Inteligencia Artificial Generativa

Fernando Fetis Riquelme

Otoño, 2026

Índice general

Parte I

Introducción

Capítulo 1

Overview y conceptos previos

1.1 Modelos generativos modernos

La inteligencia artificial ha pasado por diversas etapas históricas, incluyendo dos periodos conocidos como **inviernos de la IA**, caracterizados por una disminución considerable en la inversión y avances significativos en el área. Actualmente, podríamos afirmar que la IA vive su mejor momento, principalmente gracias al auge de arquitecturas neuronales eficientes, destacando especialmente los modelos tipo Transformer. Estos avances han sido impulsados por mejoras significativas en hardware (GPU) y una disponibilidad masiva de datos no supervisados, los cuales permiten, por ejemplo, preentrenar grandes modelos de lenguaje (LLMs) para que adquieran conocimiento del mundo.

En este primer capítulo se revisarán de forma general los distintos paradigmas generativos modernos, los cuales abarcan desde los modelos autorregresivos (usado en LLMs y agentes) hasta los modelos de difusión y flow matching (usado para la generación de imágenes y videos). En este capítulo también se entregará la notación utilizada a lo largo del libro, y se repasarán algunos conceptos básicos de probabilidades y redes neuronales, los cuales serán necesarios para el estudio de la IA generativa moderna, la cual tiene un enfoque completamente probabilístico.

Diariamente se publican cientos de artículos relacionados con IA, haciendo imposible revisar todas las contribuciones relevantes. En este libro, por lo tanto, se abordarán aquellos resultados considerados más fundamentales y relevantes, colocando énfasis en los principios subyacentes de la IA generativa más que en modelos específicos, los cuales tienden a evolucionar y quedar obsoletos rápidamente. Comprender estos principios resulta indispensable para distinguir entre el hype publicitario y las auténticas innovaciones científicas en este campo.

Por otro lado, entender en profundidad el funcionamiento interno de estos modelos permite implementar soluciones propias desde cero, lo cual es particularmente relevante en contextos donde la privacidad o la personalización resulta esencial para el trabajo que se busca realizar. Además, muchas veces los modelos del estado del arte son publicados únicamente con los pesos de las redes neuronales, sin incluir, por ejemplo, los loops de entrenamiento y/o inferencia, los cuales son indispensables para poder utilizar estos modelos en la práctica.

1.1.1 Qué puede generar un modelo generativo

Un modelo generativo es un tipo de modelo de machine learning cuyo objetivo es aprender la distribución subyacente de los datos, denotada como $p_{\text{data}}(x)$, a partir de ejemplos generados por dicha distribución. Este aprendizaje permite generar nuevas muestras sintéticas que resultan similares a las utilizadas durante el entrenamiento. Usualmente, estos modelos son entrenados de manera no supervisada (o auto-supervisada), lo que ha permitido, por ejemplo, poder utilizar gran parte del contenido que hay internet para entrenar modelos de lenguaje o de visión.

El desarrollo de la IA generativa moderna comenzó alrededor del año 2014 con la aparición del autoencoder variacional (VAE) para imágenes [1] y los modelos seq2seq para texto [2]. Desde entonces, las capacidades de estos modelos han evolucionado rápidamente, alcanzando resultados que antes se creían exclusivos del ser humano. Algunas tareas que permiten resolver los modelos generativos actuales son las siguientes:

- **Creación y modificación de imágenes**: inpainting (modificar solo una porción de una imagen), colorización (pasar una imagen en blanco y negro a color), superresolución (aumentar la resolución de una imagen), outpainting (extensión de imágenes más allá de sus bordes), mejora de calidad, interpolación semántica entre imágenes, transferencia de estilos, etc.

Figura 1: Tarea de inpainting. Imagen obtenida desde [3].

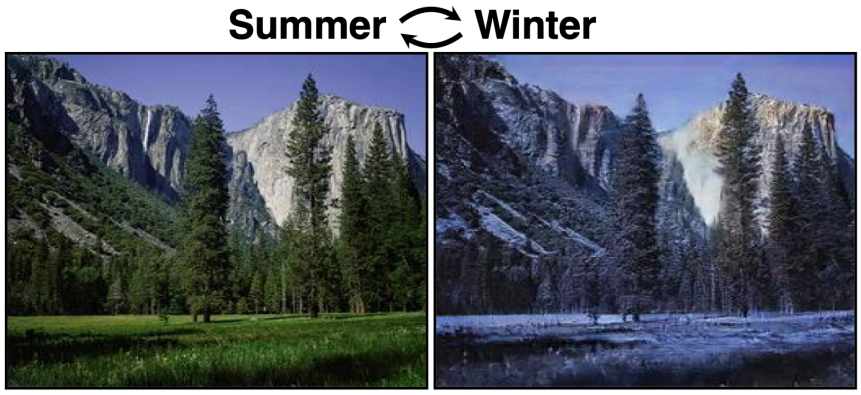

Figura 2: Tarea de transferencia de estilo. Imagen obtenida desde [4].

- **Generación de sonido y video**: clonación de voz, composición musical, generación de videos a partir de descripciones textuales, extensión o edición de videos, etc.

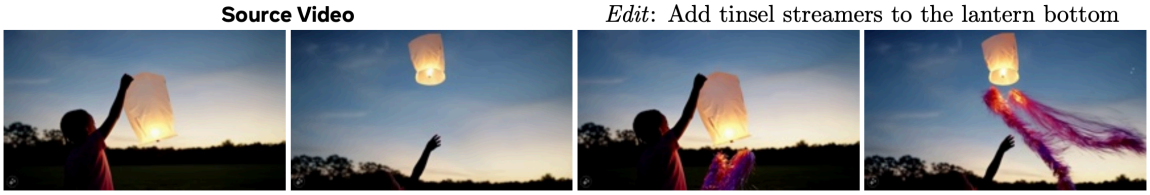

Figura 3: Edición de video mediante un prompt. Imagen obtenida desde [5].

- **Simulaciones de juegos**: generación de movimientos óptimos en ajedrez, NPCs personalizables, creación dinámica de escenarios de juego en tiempo real, etc. Más aún, este tipo de problemas puede ser incluido en una línea de investigación llamada **world models** [6], donde se busca modelar un ambiente completo, en el cual un agente puede tomar decisiones que influyen en el ambiente y en su propio estado interno.

Figura 4: Simulación del juego DOOM con creación de ambiente en tiempo real. Imagen obtenida desde [7].

- **Robótica potenciada por LLMs**: uso de modelos tipo GPT para que robots interpreten y ejecuten instrucciones dadas en lenguaje natural.

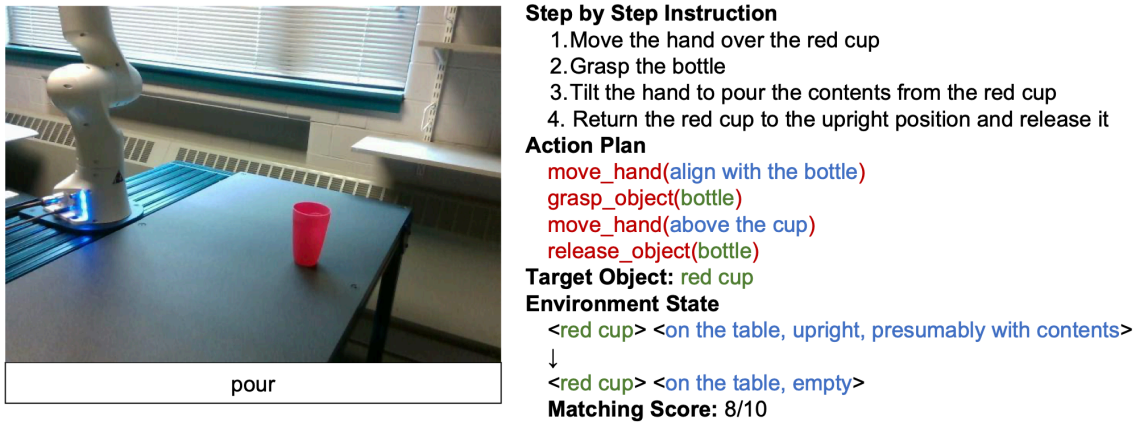

Figura 5: Robot usando un LLM para resolver una tarea. Imagen obtenida desde [8].

- **Predicción de estructuras moleculares**: diseño de fármacos, predicción del plegamiento de proteínas (como AlphaFold [9], reconocido con el premio Nobel de Química en 2024), generación de secuencias genéticas y creación de nuevos materiales con propiedades avanzadas (e.g., más resistentes o con capacidad autoregenerativa), etc.

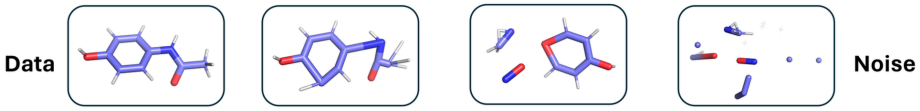

Figura 6: Modelo de difusión utilizado para la generación molecular. Imagen obtenida desde [10].

- **Agentes autónomos**: planificación para lograr un objetivo, uso de herramientas, análisis y toma de decisiones, etc.

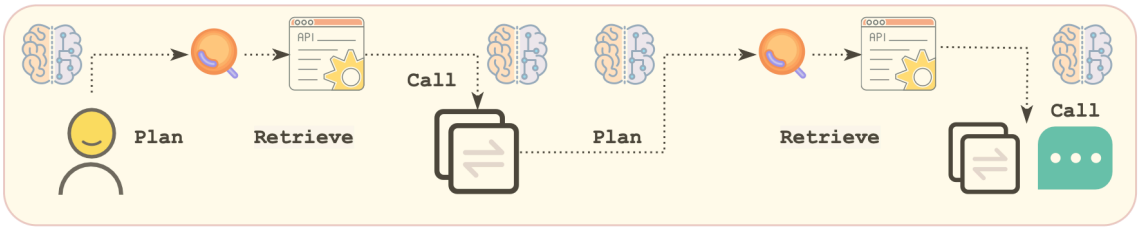

Figura 7: Planificación y uso de herramientas para completar una tarea. Imagen obtenida desde [11].

- **Investigaciones científicas**: predicción de estados cerebrales, simulación de procesos de diferenciación celular, demostración automática de teoremas, modelos del clima, resolución numérica de ecuaciones diferenciales parciales (EDPs), detección de anomalías astronómicas, etc.

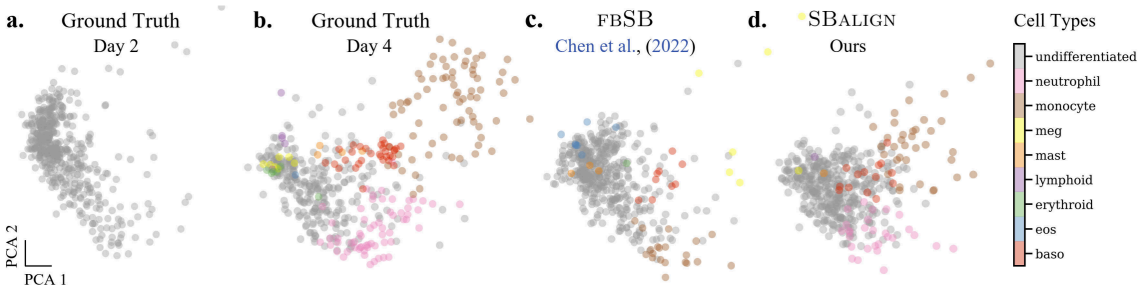

Figura 8: Predicción de diferenciación celular usando puentes de Schrödinger. Imagen obtenida desde [12].

Si bien todas estas tareas parecen ser muy diferentes entre sí, todas comparten los mismos principios subyacentes para su resolución mediante modelos generativos. Es esperable que en un futuro no muy lejano, los modelos generativos puedan aplicarse para otro tipo de problemas como la creación de nuevos sabores y olores, fabricación de medicamentos personalizados, estudio de teorías científicas novedosas, implantación de recuerdos artificiales e incluso formas de vida alternativas. Del mismo modo, este tipo de modelos

podrá, eventualmente, contribuir al desarrollo de nuevos sistemas políticos para mejorar la gobernanza, formular estrategias económicas innovadoras, e incluso encontrar soluciones a grandes problemas como lo es el calentamiento global o la hambruna en países menos desarrollados.

Perspectivas para estudiar los modelos generativos

Los modelos generativos actuales pueden ser estudiados desde distintos ángulos, por ejemplo:

- **Perspectiva económica y política**: consumo energético y monetario (e.g., se estima que entrenar GPT 4 costó 78 millones de dólares y generó lo equivalente a 15 mil toneladas de CO_2). Por otro lado, el 2024 Joe Biden declaró a la IA como un asunto de seguridad nacional, mientras que el 2025, Donald Trump anunció una inversión de 500 mil millones de dólares en IA con el proyecto Stargate (comparar, por ejemplo, con el programa Apolo, en el cual se invirtieron 180 mil millones de dólares).
- **Perspectiva ética**: sesgos (género, raza, políticos), mal uso (deepfake, filtración de datos, ataques terroristas, ataques adversarios) e infracciones de copyright (diseño, libros, música, intelectual, etc.).
- **Interés filosófico**: cómo definir AGI/ASI, qué es inteligencia, entender, pensar o sentir.
- **Interés teórico**: ¿puede un modelo realmente crear algo nuevo o solo interpola lo que aprendió? ¿Es posible evitar la alucinación de los LLMs? Los modelos de difusión, autoencoders variacionales y modelos basados en flujos tienen muy buen background matemático, lo que permite estudiar más fácilmente sus propiedades teóricas (e.g., existen resultados de convergencia, generalización a otros espacios y extensión a variedades diferenciables). Sin embargo, el entendimiento de los LLMs es muy limitado hoy en día, por lo que la mayoría de propiedades conocidas en este tipo de modelos son solo empíricas, sin ningún fundamento técnico que garantice la presencia o ausencia de estas propiedades.

En este libro solo se mencionarán algunos de estos puntos, sin entrar en detalle en ninguno de ellos ya que hoy en día no hay un consenso claro acerca de estos temas. Es esperable que en los próximos años se sigan desarrollando estos temas y se obtenga la madurez y robustez suficiente para estudiar estos tópicos de manera más formal.

1.1.2 Paradigmas modernos de la IA generativa

La inteligencia artificial generativa ha avanzado enormemente en la última década (e.g., en 2019, GPT 2 no podía contar hasta 10 de forma correcta), impulsada principalmente por el desarrollo de modelos cada vez más potentes, los cuales consiguen realizar tareas que antes

solo se creían posible para los humanos. Sin embargo, muchos de estos modelos modernos se forman ensamblando sub-modelos, muchas veces entrenados de forma independiente, donde cada uno se especializa en una tarea específica. Por ejemplo, para generar una imagen a partir de un texto descriptivo, el modelo DALL-E 2 de OpenAI [13] combina un modelo autorregresivo tipo Transformer para procesar el texto y luego utiliza un modelo de difusión para generar la imagen usando el texto procesado.

Por otro lado, cada uno de estos modelos individuales que componen un modelo mayor, suelen seguir alguno de los paradigmas modernos de IA generativa, los cuales son principalmente 6 (ordenados, a mi juicio, de forma ascendente en dificultad):

- Modelos autorregresivos.
- Redes generativas adversarias.
- Autoencoders variacionales.
- Modelos basados en score.
- Modelos basados en flujo.
- Modelos de difusión.

Es importante mencionar que tanto los modelos basados en flujo como los modelos de difusión pueden ser formulados en su versión temporal continua, lo cual aumenta considerablemente su dificultad y desarrollo. A continuación se describirá de forma general cada uno de estos paradigmas, los cuales serán estudiados en detalle en los próximos capítulos.

Modelos autorregresivos

Este es el paradigma generativo de facto para la generación de texto (aunque en los últimos años también se han propuesto enfoques basados en difusión). El enfoque autorregresivo consiste en ir generando cada palabra de una secuencia de texto de forma individual, utilizando las palabras ya generadas anteriormente. Más precisamente, si $(x_1, ..., x_t)$ es la secuencia de tokens (e.g., palabras o letras) generadas hasta el instante $t \in \mathbb{N}$, un modelo autorregresivo generará la siguiente palabra x_{t+1} usando una red neuronal que toma como entrada las t palabras anteriores, $(x_1, ..., x_t)$.

Una característica importante de este tipo de modelos es que también pueden, a priori, procesar otro tipo de secuencias como secuencias de sonido (e.g., para generar música), secuencias temporales (e.g., para predecir variaciones en los mercados), o secuencias de aminoácidos (e.g., para generar proteínas biológicamente plausibles), siempre y cuando se encuentre una forma eficiente de obtener un **vector de embedding** (i.e., una representación vectorial) para el tipo de dato que se esté utilizando. Además, hoy en día las redes neuronales utilizadas en este paradigma suelen ser de tipo Transformer [14] ya que esta arquitectura ha mostrado funcionar mejor que otras arquitecturas anteriores como la GRU [15] o la LSTM [16].

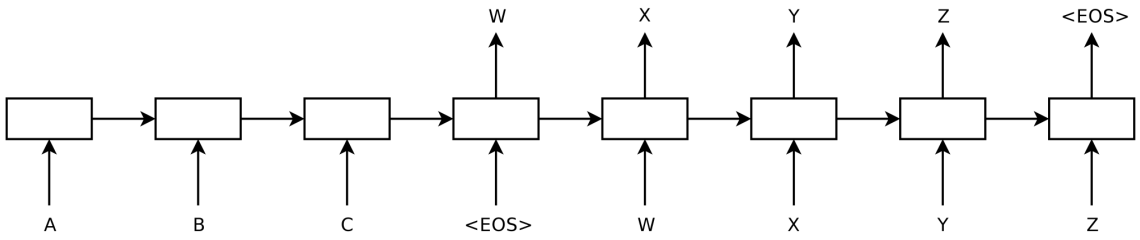

Figura 9: Modelo seq2seq, donde la entrada inicial (prompt) es ABC y la secuencia generada de forma autorregresiva es WXYZ. Imagen obtenida desde [2].

Redes generativas adversarias

Las redes generativas adversarias (GANs) están formadas por dos componentes. El primer componente es una red neuronal discriminativa (i.e., un clasificador) que busca reconocer si la muestra x que recibe como entrada es una imagen real o una imagen generada artificialmente. El segundo componente (generador) es otra red neuronal que busca aprender a generar muestras x similares a las que hay en el conjunto de entrenamiento (el clasificador debería diferenciar este tipo de muestras de las muestras reales si estuviese bien entrenado). Para lograr esto, el modelo generador es entrenado buscando que el modelo discriminativo se equivoque. Luego del entrenamiento, se desecha el modelo discriminador y solo se conserva el modelo generador, el cual aprendió a generar muestras similares a las que se usaron durante el entrenamiento.

Dada la naturaleza competitiva de las GANs (red generadora vs. red discriminadora), el entrenamiento de este tipo de modelos es muy inestable y, de hecho, el valor de la función objetivo puede divergir. Si bien se han propuesto técnicas para aminorar estos problemas, las GANs también tienen otros problemas como aprender a generar siempre una misma imagen que se sabe que es capaz de engañar al clasificador. Problemas de esta naturaleza también provocan que las GANs no cubran, por lo general, todo el soporte de la distribución, concentrando su proceso de generación en una región pequeña del soporte real (**mode collapse**). Sin embargo, este fue el paradigma principal para generar imágenes antes de la llegada de los modelos de difusión.

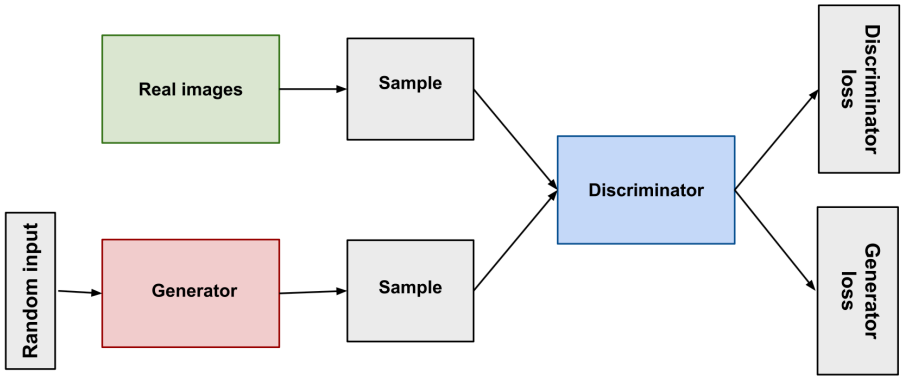

Figura 10: Un modelo discriminativo (clasificador) aprende a diferenciar muestras reales de muestras generadas aleatoriamente. Imagen obtenida desde [17].

Autoencoders variacionales

Los autoencoders variacionales (VAEs) son modelos generativos que permiten codificar una entrada x (e.g., una imagen) en un vector latente z (más precisamente, en una distribución $q_\phi(z \mid x)$), el cual luego puede ser decodificado para reconstruir la muestra original (usando una distribución $p_\theta(x \mid z)$). La red neuronal que realiza la transformación $x \mapsto z$ se llama **encoder**, mientras que la red neuronal que realiza la transformación $z \mapsto x$ se llama **decoder**. Ambas redes se entrenan de manera conjunta, optimizando una cantidad conocida como **ELBO**, la cual funciona como una función objetivo proxy al objetivo clásico de máxima verosimilitud.

Una vez el VAE está entrenado, es posible generar una nueva muestra x comenzando con un vector latente inicial z cualquiera, y pasando este vector por el decoder $z \mapsto x$.

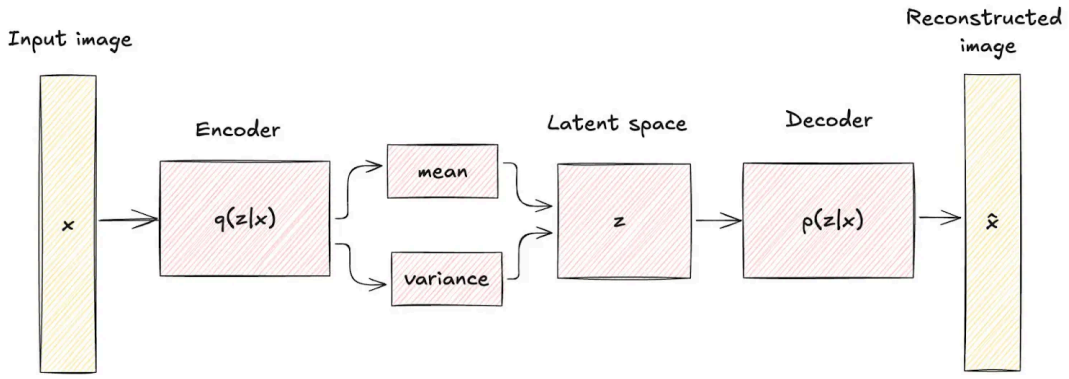

Figura 11: El encoder de un VAE predice la media y varianza de una variable latente z gaussiana. Por otro lado, el decoder aprende a reconstruir la imagen a partir de la variable latente que generó el encoder. Imagen obtenida desde [18].

Modelos basados en score

Los **modelos basados en energía** (EBMs) consisten en modelar la función de densidad $p_\theta(x)$ mediante otra cantidad $E_\theta(x)$ llamada **energía**. Este cambio de variable permite saltarse las restricciones típicas que se tienen al modelar directamente una distribución de probabilidad (positividad y que integre 1), las cuales suelen ser difíciles de imponer en una red neuronal.

Si bien este tipo de modelos puede ser estudiado desde una perspectiva más clásica (e.g., estudiando máquinas de Boltzmann), el enfoque moderno consiste en reformular estos modelos para que aprendan la cantidad $\nabla_x \log p_\theta(x)$ (llamada **score**) en vez de la función de energía. Si bien por ahora no son claras las ventajas de conocer esta cantidad, este gradiente resultará ser muy importante para poder realizar generación condicional en los modelos de difusión.

Modelos basados en flujo

Esta familia de modelos consiste en aplicar consecutivamente un conjunto de funciones simples para poder transformar un ruido inicial z_0 (e.g., una muestra gaussiana) en una muestra de la distribución $p_{\text{data}}(x)$ que se quiere aprender. El principal problema de este paradigma es que las funciones que se aplican deben ser invertibles (de hecho, se verá que deben ser difeomorfismos), lo cual es una restricción difícil de conseguir en una red neuronal.

Por otro lado, su formulación a tiempo continuo (donde la dinámica de evolución viene dada por una ecuación diferencial ordinaria) no tiene este tipo de restricciones y, de hecho, tiene muy buenas propiedades. Las técnicas de flow matching [19] y rectified flows [20] son ejemplos de modelos basados en flujo a tiempo continuo, las cuales se podrían considerar como el estado del arte actual para la generación de imágenes y video (ver, por ejemplo, FLUX.1 Kontext [21]).

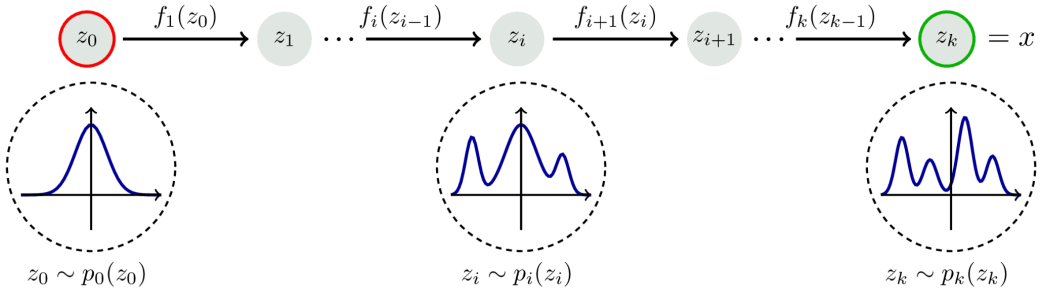

Figura 12: Transformación de un ruido gaussiano inicial en una muestra de la distribución que se busca aprender. Imagen obtenida desde [22].

Modelos de difusión

Los modelos de difusión consisten en corromper una imagen inyectándole ruido de forma progresiva, para así entrenar un modelo neuronal que aprenda a deshacer el ruido inyectado (modelo de denoising). La cantidad de iteraciones de inyección de ruido debe ser suficientemente grande para que, en el último paso, la imagen corrompida sea similar a una muestra gaussiana. De esta forma, con el modelo de denoising ya entrenado, es posible generar una nueva imagen comenzando desde una muestra gaussiana inicial, para luego aplicar el modelo de denoising de forma iterativa hasta llegar a una imagen sin ruido.

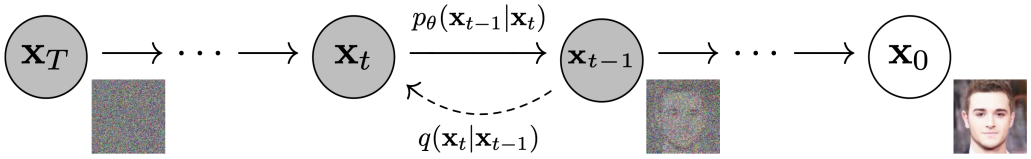

Figura 13: Proceso de denoising de un modelo de difusión. Imagen obtenida desde [23].

1.1.3 Orden histórico de los paradigmas modernos

Tal como se suele considerar que AlexNet [24] (2012) marcó el comienzo de la revolución del deep learning, se puede considerar que los autoencoders variacionales [1] (finales del 2013) marcaron el comienzo de la IA generativa moderna (aunque también se podría considerar otros modelos como las GANs [25] o los modelos seq2seq [2]). El siguiente diagrama temporal permite poner en contexto algunos de los trabajos más importantes de cada paradigma generativo. Notar que también se incluyen algunos trabajos fundacionales de deep learning como lo es la técnica de dropout [26] o el optimizador Adam [27]:

```
gantt
    title Orden histórico de los modelos generativos separados por
paradigma
    dateFormat DD-MM-YYYY
    axisFormat %Y

     section Modelos autorregresivos
        Seq2seq y atención :milestone, 07-09-2014, 0d
        PixelRNN :milestone, 25-01-2016, 0d
        Transformer :milestone, 12-06-2017, 0d
        GPT 1 :milestone, 15-05-2018, 0d
        BERT :milestone, 11-10-2018, 0d
        GPT 2 :milestone, 14-02-2019, 0d
        Bard :milestone, 29-10-2019, 0d
        GPT 3 :milestone, 28-05-2020, 0d
```

```
    Vision Transformer :milestone, 22-10-2020, 0d
    DALL-E 1 :milestone, 24-02-2021, 0d
    ChatGPT :milestone, 30-11-2022, 0d
    LLaMA 1 :milestone, 27-02-2023, 0d
    GPT-4 :milestone, 14-03-2023, 0d
    Claude 1 :milestone, 14-03-2023, 0d
    LLaMA 2 :milestone, 18-07-2023, 0d
    Grok 1 :milestone, 03-11-2023, 0d
    Gemini 1 :milestone, 08-02-2024, 0d
    LLaMA 3 :milestone, 31-07-2024, 0d
    Deepseek-R1 :milestone, 22-01-2025, 0d
    GPT-4.5 :milestone, 27-02-2025, 0d
    LLaMA 4 :milestone, 05-04-2025, 0d
    GPT-5: milestone, 07-08-2025, 0d
    Grok 4.1: milestone, 17-11-2025, 0d
    Claude Opus 4.6: milestone, 05-02-2026, 0d
    Gemini 3.1 Pro: milestone, 19-02-2026, 0d

section Redes generativas adversarias
    GAN :milestone, 10-06-2014, 0d
    DCGAN :milestone, 19-11-2015, 0d
    CycleGAN :milestone, 30-03-2017, 0d
    ProGAN :milestone, 27-10-2017, 0d
    SAGAN :milestone, 21-05-2018, 0d
    StyleGAN :milestone, 12-12-2018, 0d
    StyleGAN 2 :milestone, 09-12-2019, 0d
    VQ-GAN :milestone, 17-12-2020, 0d
    StyleGAN 3 :milestone, 23-06-2021, 0d

section Autoencoders variacionales
    VAE :milestone, 20-12-2013, 0d
    VQ-VAE :milestone, 02-11-2017, 0d

section Modelos de difusión
    Sohl-Dickstein et al. :milestone, 12-03-2015, 0d
    DDPM :milestone, 19-06-2020, 0d
    DM SDE :milestone, 20-10-2020, 0d
    Diffusion Transformer :milestone, 19-12-2020, 0d
    Classifier guidance :milestone, 11-05-2021, 0d
    DALL-E 2 :milestone, 13-04-2022, 0d
    Imagen :milestone, 23-05-2022, 0d
    Stable Diffusion 1 :milestone, 15-08-2022, 0d
    Stable Diffusion 2 :milestone, 23-11-2022, 0d
```

```
        DALL-E 3 :milestone, 10-08-2023, 0d
        Imagen 2 :milestone, 15-12-2023, 0d
        Sora :milestone, 15-02-2024, 0d
        Veo 1 :milestone, 15-04-2024, 0d
        Imagen 3 :milestone, 13-08-2024, 0d
        Veo 2 :milestone, 16-12-2024, 0d
        Veo 3 :milestone, 23-05-2025, 0d
        Nano banana: milestone, 25-08-2025, 0d
        Nano banana 2: milestone, 26-02-2026, 0d

    section Flujos normalizantes continuos
        Neural ODE :milestone, 19-06-2018, 0d
        Rectified Flow :milestone, 07-09-2022, 0d
        Flow Matching :milestone, 06-10-2022, 0d
        Stable Diffusion 3 :milestone, 05-03-2024, 0d
        Flux.1 :milestone, 01-08-2024, 0d
        MovieGen :milestone, 17-10-2024, 0d
        FLUX.1 Kontext :milestone, 29-05-2025, 0d
        Qwen-image :milestone, 10-02-2026, 0d

    section Otros
        Dropout :milestone, 14-06-2014, 0d
        Adam :milestone, 22-12-2014, 0d
        BatchNorm :milestone, 11-02-2015, 0d
        UNet :milestone, 18-05-2015, 0d
        ResNet :milestone, 10-12-2015, 0d
        LayerNorm :milestone, 21-06-2016, 0d
        GroupNorm :milestone, 22-03-2018, 0d
        World models :milestone, 27-03-2018, 0d
        RAG :milestone, 22-05-2020, 0d
        LoRA :milestone, 17-06-2021, 0d
        CoT :milestone, 08-01-2022, 0d
        ReAct :milestone, 06-10-2022, 0d
        Toolformer :milestone, 09-02-2023, 0d
        Test-time compute :milestone, 06-08-2024, 0d
```

1.2 Repaso previo

En esta sección se definirá la notación general usada a lo largo del libro y se repasarán algunos conceptos de probabilidades y redes neuronales que serán utilizados en el estudio de los distintos paradigmas generativos. Otros conceptos más específicos y ortogonales a la IA generativa serán estudiados en los capítulos respectivos donde se utilicen. Por ejemplo,

los LLMs hacen uso de técnicas de reinforcement learning, mientras que las GANs pueden vincularse con conceptos de teoría de juegos. Por otro lado, los modelos basados en energía, los VAEs y los modelos de difusión tiene conexiones con la física estadística y termodinámica mediante la distribución de Boltzmann y la energía libre de Helmholtz. Además, muchos de los paradigmas a estudiar pueden ser conectados con tópicos de transporte óptimo cuando se estudian con suficiente profundidad.

A lo largo de todo el libro, $\mathcal{M}_{m,n}(\mathbb{R})$ denotará el conjunto de matrices de tamaño $m \times n$ (con valores en $\mathbb{R}$), mientras que los vectores se considerarán siempre verticales ($\mathbb{R}^D \cong \mathcal{M}_{D,1}(\mathbb{R})$). La matriz identidad de tamaño $n \times n$, $\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \in \mathcal{M}_{n,n}(\mathbb{R})$ será denotada por I_n. Para dos vectores $x, y \in \mathbb{R}^D$, su producto punto $x \cdot y = \sum_{d=1}^{D} x_d\, y_d$ será denotado por $\langle x, y \rangle$, mientras que la notación $x \odot y \in \mathbb{R}^D$ representará el producto de Hadamard, es decir, el producto coordenada a coordenada: $(x \odot y)_d = x_d\, y_d$ para $d \in \{1, ..., D\}$.

Para un conjunto finito A, su cardinal (i.e., cantidad de elementos) será denotado por $|A|$. Para dos conjuntos A y B, el producto cruz $A \times B = \{(a, b) : a \in A, b \in B\}$ representará el conjunto de todos los posibles pares que se pueden formar con elementos de A y B, mientras que la notación $f : A \to B$ indicará una función con dominio A y codominio B. Por otro lado, para $a, b \in \mathbb{R}$, la notación $a \ll b$ indicará que a es mucho menor que b (análogo para $\gg$). Además, los logaritmos serán considerados siempre naturales (i.e., con base e). En particular, los conceptos de teoría de la información (e.g., entropía) serán medidos en nats. La notación de superíndice (e.g., $x^1, ..., x^N$) será usada para indexar un conjunto de $N \in \mathbb{N}$ elementos, y no debe confundirse con el operador potencia. En cambio, la notación de subíndice (e.g., $x_1, ..., x_D$) por lo general será usada para recorrer las componentes de un vector $x \in \mathbb{R}^D$.

En cuanto a operadores diferenciales, para un campo escalar $f : \mathbb{R}^D \to \mathbb{R}$, su gradiente será denotado como $\nabla_x f(x) \coloneqq \begin{pmatrix} \frac{\partial f}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_D}(x) \end{pmatrix} \in \mathbb{R}^D$, mientras que para un campo vectorial $F : \mathbb{R}^N \to \mathbb{R}^M$ su matriz jacobiana será denotada como $\mathrm{D}_x F(x) \coloneqq \begin{pmatrix} \nabla_x F_1(x)^\top \\ \vdots \\ \nabla_x F_M(x)^\top \end{pmatrix} \in \mathcal{M}_{M,N}(\mathbb{R})$ (i.e., para $F(x) = (F_1(x), ..., F_M(x))$, cada entrada es $(\mathrm{D}_x F(x))_{ij} = \frac{\partial F_i}{\partial x_j}(x)$).

1.2.1 Probabilidades

De manera informal, una **variable aleatoria** x es una función que puede tomar distintos valores de acuerdo a una **distribución de probabilidad** $p(x)$, la cual se interpreta de dos formas distintas, dependiendo de si se está trabajando con una variable aleatoria discreta o continua. En ambos casos, $p(x)$ corresponde a una función asociada a la probabilidad

de que x tome cierto valor. De esta forma, para indicar que x sigue una distribución de probabilidad $p(x)$, se suele escribir $x \sim p(x)$, independientemente de si la variable aleatoria es discreta o continua.

El conjunto donde $p(x)$ es positivo se denomina **soporte**, $\text{supp}(x) \coloneqq \{x : p(x) > 0\}$, e indica los distintos valores que una variable aleatoria puede tomar. Cuando el soporte de una variable aleatoria es finito (i.e., $|\text{supp}(x)| < \infty$), la variable aleatoria se dice **discreta**, y $p(x)$ se llama **función de masa** ya que indica cuánta masa (probabilidad) le asigna la variable aleatoria a cada elemento del soporte. Por otro lado, si x es una variable aleatoria en $\mathbb{R}^D$ que puede tomar una cantidad continua de valores, entonces la variable aleatoria se dice **continua**, y $p(x)$ se llama **función de densidad**. El cambio de nombre en ambos casos se debe a que, si bien $p(x)$ tiene el mismo propósito en ambos casos, la forma de operar con $p(x)$ cambia dependiendo de si se está trabajando con una variable discreta o una variable continua.

Si bien se pueden combinar ambos tipos de variables para trabajar con un tipo de dato mixto (discreto y continuo), no suele ser necesario revisar esta extensión para el estudio de los modelos generativos, lo cual requeriría introducir algunos conceptos de teoría de la medida. En consecuencia, siempre se asumirá que cada variable aleatoria es, o bien discreta, o bien continua.

Variables aleatorias discretas

Si $x \sim p(x)$ es una variable aleatoria discreta con $\text{supp}(x) = \{k_1, ..., k_N\}$, entonces la función de masa $p(x)$ es una función $p : \{k_1, ..., k_N\} \to [0, 1]$, la cual indica la probabilidad de que x tome un cierto valor de su soporte, es decir, $\mathbb{P}(x = k_n) = p(k_n)$. Notar que, por definición de medida de probabilidad, es necesario que $p(k_n) \geq 0$ y que $\sum_{n=1}^{N} p(k_n) = 1$.

Por otro lado, la igualdad $p(k_n) = \mathbb{P}(x = k_n)$ muchas veces motiva a usar la notación $p(x = k_n)$, la cual es útil para evitar ambigüedades cuando se está trabajando con más de una variable aleatoria. Además, dado que el soporte es finito, se pueden almacenar todas las salidas de la función de masa $p(x)$ en un **vector de probabilidades** $(p(k_1), ..., p(k_N)) \in [0, 1]^N$, el cual muchas veces, y abusando de notación, es denotado también por p.

A lo largo de todo el libro, se denotará como Δ^N al ***N*-símplex**, el cual es el conjunto de vectores de probabilidad de largo $N \in \mathbb{N}$:

$$\Delta^N \coloneqq \left\{ (p_1, ..., p_N) \in \mathbb{R}^N : \left(\sum_{n=1}^{N} p_n = 1 \right) \wedge (p_n \geq 0, \forall n \in \{1, ..., N\}) \right\}$$

La distribución discreta más simple es la **distribución de Bernoulli**, en la cual la variable aleatoria x solo puede tomar los valores 0 y 1. Esta distribución permite modelar escenarios

donde solo puede haber dos resultados distintos y excluyentes, como en el lanzamiento de una moneda, o el éxito/fracaso de algún experimento.

Se dice que $x \sim \text{Bernoulli}(r)$, con $r \in [0,1]$, si $\text{supp}(x) = \{0,1\}$ con $\mathbb{P}(x=1) = r$ y $\mathbb{P}(x=0) = 1-r$, es decir:

$$\begin{aligned} p(x) &= \begin{cases} r & \text{si } x = 1 \\ 1-r & \text{si } x = 0 \end{cases} \\ &= r^x(1-r)^{1-x}, \quad x \in \{0,1\} \end{aligned}$$

La **distribución categórica** es la extensión natural de la distribución de Bernoulli al caso donde la variable aleatoria puede tomar N posibles valores. Se dice que $x \sim \text{Categorical}(r)$, con $r \in \Delta^N$, si $\text{supp}(x) = \{k_1, ..., k_N\}$ y $\mathbb{P}(x = k_n) = r_n$, es decir, $p(k_n)$ es la k-ésima coordenada del vector r. En el siguiente gráfico se pueden ver 100 muestras generadas para 3 distribuciones categóricas distintas. Se observa que los 3 gráficos de frecuencia son consistentes con la función de masa $p(x)$ que induce cada vector de probabilidades $r \in \Delta^3$:

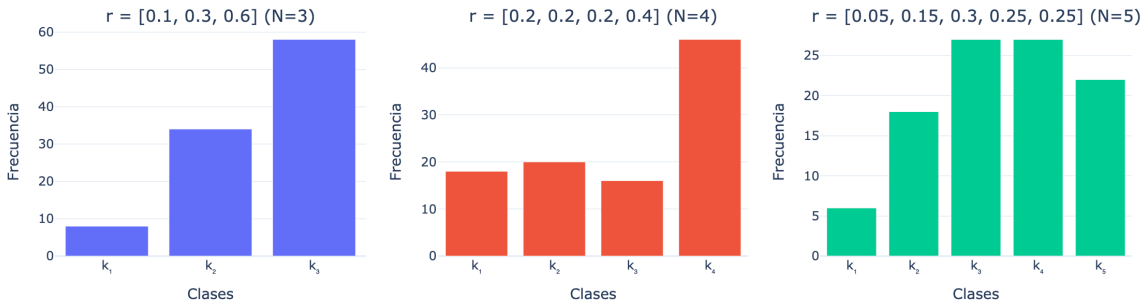

Figura 14: 100 muestras generadas para tres distribuciones categóricas con distintos vectores de probabilidad $r \in \Delta^3$.

Si bien la distribución categórica es lo suficientemente general para englobar todas las distribuciones posibles de variables aleatorias discretas, hay algunos casos particulares frecuentes, los cuales reciben nombres distintivos (e.g., la **distribución binomial**). Además, también es usual incluir dentro de las distribuciones discretas a aquellas variables aleatorias que pueden tomar una cantidad infinita numerable (no continua) de valores (e.g., $\text{supp}(x) = \mathbb{N}$ para la **distribución de Poisson**). Sin embargo, no será necesario considerar estos casos para los temas que se estudiarán a lo largo del libro.

Variables aleatorias continuas

Cuando se trabaja con variables aleatorias continuas, es usual considerar que estas son **no atómicas**, es decir, la probabilidad de tomar un valor en específico es siempre 0 (e.g., si se elige al azar un número en el intervalo $[0,1]$, la probabilidad de elegir exactamente

el valor 0.8 es cero). Por lo tanto, para este tipo de variables, lo que se suele calcular es la probabilidad de que la variable aleatoria tome un valor dentro de un conjunto de posibles valores. Más precisamente, si $x \sim p(x)$ es una variable aleatoria continua en $\mathbb{R}^D$ (i.e., $\text{supp}(x) \subset \mathbb{R}^D$ es no numerable), entonces la función de densidad $p(x)$ es una función $p : \mathbb{R}^D \to \mathbb{R}_+$, la cual permite calcular la probabilidad de que x esté en un cierto conjunto $A \subset \mathbb{R}^D$ mediante integración:

$$\mathbb{P}(x \in A) = \int_A p(x)\,\mathrm{d}x$$

Notar que, por definición de probabilidad, la función de densidad $p(x)$ necesariamente debe cumplir que $\int_{\mathbb{R}^D} p(x)\,\mathrm{d}x = 1$, lo cual corresponde a la probabilidad calculada sobre todo el soporte de la variable aleatoria. Por ejemplo, para $D = 1$, la función de densidad

$$p(x) = \begin{cases} \frac{10-x^2}{42} & \text{si } x \in [-3, 3] \\ 0 & \text{si } x \notin [-3, 3] \end{cases}$$

es efectivamente una función de densidad (con $\text{supp}(x) = [-3, 3]$) ya que

$$\int_{\mathbb{R}} p(x)\,\mathrm{d}x = \int_{-3}^{3} \frac{10 - x^2}{42}\,\mathrm{d}x = \frac{1}{42}\left(10x - \frac{x^3}{3}\right) \Big|_{x=-3}^{x=3} = 1$$

Notar que la división por 42 se realiza únicamente para forzar que la función $p(x)$ integre 1 (sin esta normalización, la integral daría 42). Por otro lado, dado que la probabilidad de que $x \in [a, b]$ es $\mathbb{P}(a \leq x \leq b) = \int_a^b p(x)\,\mathrm{d}x$, se puede calcular, por ejemplo, que $\mathbb{P}(-2 \leq x \leq -1) \approx 0.18 < 0.23 \approx \mathbb{P}(0 \leq x \leq 1)$, lo cual es directo de ver al notar que el área bajo la función de densidad que cubre el intervalo $[-2, -1]$ es menor al área que cubre el intervalo $[0, 1]$:

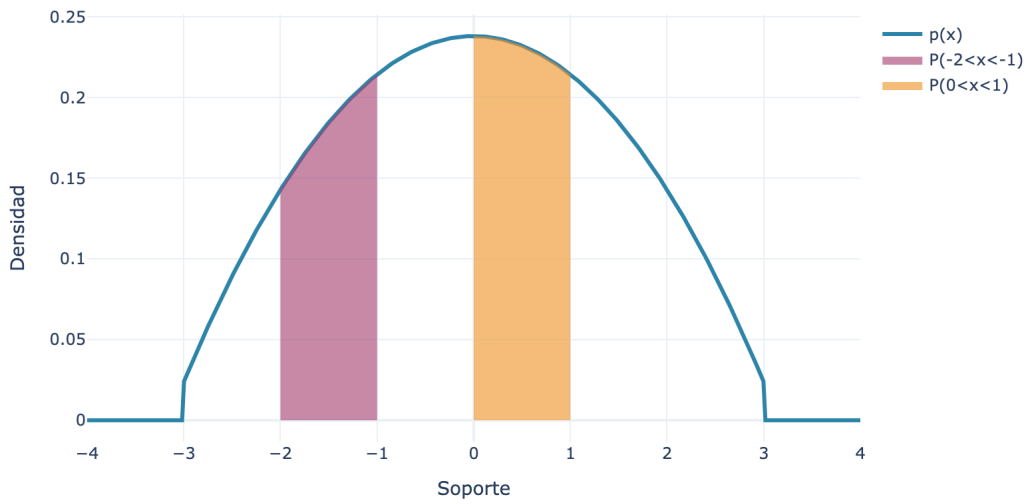

Figura 15: Función de densidad $p(x) = (10 - x^2)/42$ sobre $[-3, 3]$, con las áreas correspondientes a $\mathbb{P}(-2 \leq x \leq -1)$ y $\mathbb{P}(0 \leq x \leq 1)$.

Por lo general, nunca será necesario calcular estas integrales a mano ya que se casi siempre se trabajará, al menos en este libro, con la distribución gaussiana, la cual tiene buenas propiedades que evita tener que calcular integrales (además, la densidad gaussiana no tiene primitiva analítica, por lo que habría que recurrir a métodos numéricos para el cálculo de integrales gaussianas). Por otro lado, notar que las funciones de densidad no poseen la restricción $p(x) \leq 1$, para todo $x \in \mathbb{R}^D$ ya que $p(x)$ no representa directamente una probabilidad (como sí ocurre con las funciones de masa en el caso discreto).

Por otra parte, muy pocas veces se tendrá que $D = 1$ ya que, usualmente, la dimensión del espacio sobre el que se definirán distribuciones de probabilidad es considerablemente alto (e.g., para imágenes, cada canal de cada pixel cuenta como una dimensión diferente). Por lo tanto, la tarea de modelar distribuciones en estos espacios es una tarea no trivial debido a que, para muchos métodos estadísticos clásicos, es usual encontrarse con la **maldición de la dimensionalidad**, la cual provoca que algunos problemas no se puedan resolver eficientemente en alta dimensión, ya sea porque se necesita una cantidad exponencial de datos para obtener buenos estimadores, o bien porque se necesita una cantidad exponencial de tiempo para resolver el problema.

La **distribución gaussiana** (o **distribución normal**) es una distribución de probabilidad definida sobre todo $\mathbb{R}$ (i.e., su soporte es $(-\infty, \infty)$), cuya función de densidad viene dada por

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right),$$

donde $\mu \in \mathbb{R}$ y $\sigma^2 > 0$ son dos parámetros que definen diferentes distribuciones gaussianas, por lo que se suele denotar $x \sim \mathcal{N}(\mu, \sigma^2)$ para diferenciarlas. Posteriormente, se verá que μ resulta ser el valor esperado de la distribución, mientras que σ^2 resulta ser su varianza (o, equivalentemente, σ es su desviación estándar). Además, notar que el único rol de la constante $\frac{1}{\sqrt{2\pi\sigma^2}} > 0$ es, al igual que en el ejemplo anterior, normalizar la función de densidad para que $\int_{\mathbb{R}} p(x) = 1$ (esto se puede verificar con un cambio de variables a coordenadas polares). Por lo tanto, la parte que define el comportamiento de la distribución gaussiana es el término exponencial. Se observan varias cosas:

- La exponencial de una cantidad negativa (en este caso, $-\frac{1}{2\sigma^2}(x-\mu)^2$) decae muy rápido, por lo que la masa (probabilidad) suele estar concentrada en una porción pequeña del espacio (se dice que la gaussiana es una distribución de **cola ligera**, a diferencia de otras distribuciones que decaen más lento y son de **cola pesada**).
- El valor máximo de $\exp(-x)$ se alcanza en $x = 0$, por lo que la masa de la distribución $p(x)$ está concentrada alrededor del punto $x = \mu$. Más aún, como este máximo es único ($p(x)$ es estrictamente cóncava), la distribución es **unimodal** (i.e., hay un único punto con mayor densidad).
- Como $p(x)$ es una función simétrica con respecto a μ (i.e., $p(x-\mu) = p(\mu - x)$), el gráfico de $p(x)$ es simétrico con respecto a μ.
- A medida que aumenta el valor del parámetro $\sigma^2 > 0$, el término $\frac{1}{2\sigma^2}(x-\mu)^2$ se vuelve más chico, por lo que la densidad $p(x)$ decae más lento. En consecuencia, el gráfico es menos empinado ya que la masa se reparte de manera más uniforme entre el soporte.
- Si bien la función exponencial decae rápido, $p(x) > 0$ para todo $x \in \mathbb{R}$ (i.e., $\text{supp}(x) = \mathbb{R}$), por lo que todo punto de $\mathbb{R}$ tiene probabilidad positiva de ser generado por una distribución gaussiana, aunque para puntos muy alejados de μ, la probabilidad es extremadamente baja si el parámetro σ^2 es pequeño.
- La probabilidad del evento $x \in A$ (para $A \subset \mathbb{R}$) está dada por la integral $\int_A \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) \mathrm{d}x$, cuyo valor no se puede obtener de forma cerrada (ya que no existe una forma analítica para $\int e^{x^2} \mathrm{d}x$). En consecuencia, para calcular probabilidades sobre esta distribución se suelen usar algoritmos numéricos para aproximar la integral.
- Si bien el cálculo de probabilidades requiere integración numérica, es útil recordar algunas probabilidades típicas asociadas a la distribución gaussiana. Mediante integración se puede calcular la probabilidad de que una variable gaussiana no se desvíe más allá de una cierta cantidad de su valor medio μ. Para $x \sim \mathcal{N}(\mu, \sigma^2)$, se puede probar que:
  - $\mathbb{P}(|x-\mu| \leq \sigma) \approx 68\%$

- $\mathbb{P}(|x-\mu| \leq 2\sigma) \approx 95\%$
- $\mathbb{P}(|x-\mu| \leq 3\sigma) \approx 99.8\%$

Estas probabilidades permiten tener una noción acerca de las regiones donde se concentrarán las muestras generadas a partir de una variable aleatoria $x \sim \mathcal{N}(\mu, \sigma^2)$.

- Dado que la distribución gaussiana es no atómica (i.e., no le asigna masa a puntos individuales), los resultados anteriores siguen siendo válidos si se consideran las desigualdades de manera estricta (i.e., cambiando $\leq$ por $<$). De hecho, para distribuciones no atómicas siempre se tendrá que $\mathbb{P}(x \leq t) = \mathbb{P}(x < t)$ (y, por lo tanto, también se tendrá que $\mathbb{P}(x \geq t) = \mathbb{P}(x > t)$ ya que $\mathbb{P}(x \geq t) = 1 - \mathbb{P}(x < t)$).

Con respecto a la generación de muestras desde una distribución gaussiana, se conocen varios métodos para hacerlo, siendo el método de Box-Muller el más usual. En este libro, se asumirá que siempre se pueden generar (y de forma eficiente) muestras desde una distribución normal ya que los paquetes de Python que se usarán (e.g., NumPy o PyTorch) tienen implementados estos métodos.

En la siguiente figura se observan 3 distribuciones gaussianas diferentes, con 50 muestras generadas desde cada distribución. Se observa que para un valor $\sigma^2 > 0$ pequeño, la mayor parte de las muestras está concentrada de la media, mientras que para valores más altos, las muestras generadas oscilan en un rango mayor.

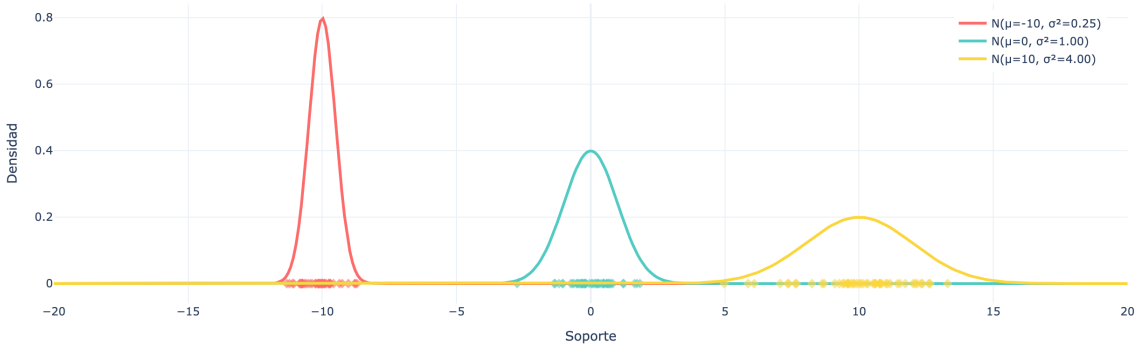

Figura 16: 50 muestras generadas desde 3 distribuciones gaussianas con distintos valores de σ^2.

La distribución gaussiana es, por lejos, la distribución más popular entre todas las distribuciones de probabilidad que existen. Esto se debe, en parte, a que esta distribución tiene muy buenas propiedades que la hacen una elección cómoda para trabajar, además de que suele estar presente en muchos resultados teóricos en probabilidades (e.g., en el teorema central del límite). Algunas propiedades que serán útiles a lo largo del libro son las siguientes:

- Sus parámetros μ y σ^2 son totalmente interpretables. Más precisamente, el parámetro μ corresponde a la esperanza de la distribución, mientras que el parámetro σ^2 corresponde a su varianza.
- Si $x \sim \mathcal{N}(\mu, \sigma^2)$, entonces $ax + b \sim \mathcal{N}(a\mu + b, a^2\sigma^2)$, es decir, trasladar afínmente una v.a. gaussiana x equivale a trasladar afínmente la función de media y ponderar cuadráticamente su varianza.
- Si $x_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ y $x_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$, entonces $x_1 + x_2 \sim \mathcal{N}(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$. En particular, la familia de variables aleatorias gaussianas es cerrada bajo la suma (i.e., la suma de gaussianas es gaussiana).
- Su función de densidad es fácil de optimizar. Más precisamente, $\log p(x)$ es una función cóncava (por lo que la función p se dice **log-cóncava**), lo que implica que no hay máximos locales donde quedarse atrapado durante la maximización de $\log p(x)$, la cual es precisamente la función objetivo del enfoque de máxima verosimilitud que se estudiará al revisar redes bayesianas.
- El promedio de variables aleatorias i.i.d. converge (en distribución) a una variable aleatoria gaussiana. Esto se conoce como el **teorema del límite central** y es uno de los motivos por el cual la distribución gaussiana aparece en tantos lugares, muchas veces de manera inesperada.
- Es la distribución de máxima entropía dentro de la familia de distribuciones cuyo soporte es todo $\mathbb{R}$ (y que además tienen varianza finita). Esto se puede interpretar diciendo que la distribución gaussiana es la que más aleatoriedad tiene o la distribución que menos información a priori asume cuando se interpreta como el prior de una variable aleatoria, lo cual es usual en modelos generativos de variable latente.
- Existen todos sus momentos: los conceptos de esperanza y varianza pueden ser extendidos a un concepto más general denominado **momento**. La distribución gaussiana tiene bien definidos todos sus momentos, lo cual no es algo que siempre ocurra y muchas veces es una propiedad deseable, ya sea por temas prácticos (e.g., para conocer más información de la variable aleatoria) o temas teóricos (la existencia de los momentos muchas veces es una hipótesis necesaria en teoremas de probabilidades y teoría de la medida).

Es posible extender la distribución gaussiana al caso multidimensional, $D > 1$. Una forma fácil es considerar una variable aleatoria $x = (x_1, ..., x_D)$ en $\mathbb{R}^D$, donde cada componente sigue una distribución gaussiana (i.e., $x_d \sim \mathcal{N}(\mu_d, \sigma_d^2)$ para todo $d \in \{1, ..., D\}$). Esto se suele denotar como $x \sim \mathcal{N}(\mu, \Sigma)$, donde ahora $\mu = (\mu_1, ..., \mu_D) \in \mathbb{R}^D$ es el vector de medias, mientras que la matriz diagonal $\Sigma = \text{diag}(\sigma_1^2, ..., \sigma_D^2) \in \mathcal{M}_{D,D}(\mathbb{R})$ almacena las varianzas de cada coordenada. En general, esta matriz puede no ser diagonal, pero siempre debe ser simétrica (i.e., $\Sigma = \Sigma^\top$) y definida positiva (i.e., sus valores propios son todos positivos) o, al menos, semidefinida positiva si se permite el caso degenerado, donde la variable aleatoria se vuelve determinista en alguna(s) dirección(es) del espacio. Sin embargo, siempre se asumirá

que Σ es no degenerada ya que esto permite poder definir una función de densidad. En este caso, la función de densidad de una variable aleatoria $x = (x_1, ..., x_D) \sim \mathcal{N}(\mu, \Sigma)$ es

$$p(x) = \frac{1}{\sqrt{(2\pi)^D \det(\Sigma)}} \exp\left(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\right),$$

donde $\det(\Sigma) > 0$ es el determinante de la matriz $\Sigma \in \mathcal{M}_{D,D}(\mathbb{R})$. Notar la similitud de esta función de densidad con la función de densidad de la gaussiana unidimensional: la constante $\frac{1}{\sqrt{(2\pi)^D \det(\Sigma)}} > 0$ es la constante de normalización para que $\int_{\mathbb{R}^D} p(x)\,\mathrm{d}x = 1$, mientras que la forma cuadrática dentro de la exponencial es una extensión de la cantidad $\frac{(x-\mu)^2}{\sigma^2}$ en la gaussiana unidimensional. Más aún, cuando $\Sigma = \sigma^2 \mathrm{I}_D$ (se dice que la gaussiana es **esférica** o **isotrópica**), se tiene que $\frac{-1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu) = \frac{-1}{2\sigma^2}\|x-\mu\|^2$, obteniendo la generalización natural de la cantidad $-\frac{1}{2\sigma^2}(x-\mu)^2$. Por otro lado, la matriz Σ^{-1} está bien definida ya que la matriz Σ es invertible por ser una matriz definida positiva. Si Σ solo fuera semidefinida positiva (i.e., si 0 es un valor propio de Σ), la matriz no sería invertible y la función de densidad $p(x)$ no estaría definida. Estos casos no se considerarán en el libro, por lo que se puede asumir siempre que la función de densidad existe.

En la siguiente figura, el gráfico de la izquierda muestra la función de densidad y algunas muestras de una gaussiana bidimensional $x \sim \mathcal{N}(\mu, \Sigma)$ con $\mu = (-1, 2)^\top$ y $\Sigma = \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix}$, es decir, $x_1 \sim \mathcal{N}(-1, 3)$ y $x_2 \sim \mathcal{N}(2, 1)$. Se observa que la primera componente de las muestras son más dispersas, lo que se debe a que la varianza de x_1 es mayor que la varianza de x_2. Además, las curvas de nivel de $p(x)$ (conjunto de puntos que tienen la misma densidad) forman elipses alrededor de la media. El hecho de que estas elipses no estén inclinadas (sus semi-ejes son paralelos a los ejes cartesianos) se debe a que las componentes de $x = (x_1, x_2)$ fueron definidas de forma independiente (Σ es matriz diagonal), por lo que el valor de una no influye en el valor de la otra (en particular, conocer el valor de una de las coordenadas no entrega ninguna información sobre el valor de la otra coordenada). Cuando las coordenadas de una variable aleatoria están relacionadas entre sí, se dice que hay una noción de dependencia entre ellas, ya que el valor de una componente afecta el valor de la otra. Este comportamiento se puede ver en el gráfico de la derecha, donde ahora se considera $\Sigma = \begin{pmatrix} 3 & 1 \\ 1 & 1 \end{pmatrix}$. Notar que el hecho de que esta matriz no sea diagonal induce una rotación en el mapa de calor asociado a la función de densidad, lo cual podría justificarse estudiando la cónica que induce la forma cuadrática asociada a la función de densidad de una gaussiana multivariable. Sin embargo, más abajo se precisará bien qué indican los valores fuera de la diagonal principal de Σ.

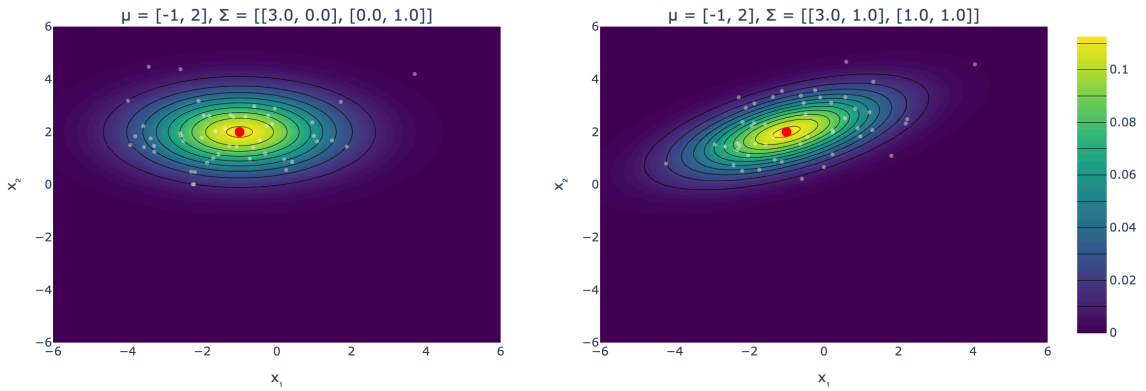

Figura 17: Funciones de densidad y muestras para dos gaussianas bidimensionales: a la izquierda, con matriz de covarianza diagonal; a la derecha, con componentes dependientes.

Probabilidad condicional y dependencia

En modelos complejos donde hay muchas variables aleatorias interactuando entre sí, es usual conocer algunos valores de esas variables aleatorias debido a que sus valores ya fueron determinados (i.e., el experimento aleatorio que definía su valor ya ocurrió y se conoce su resultado) o bien, fueron impuestos por algún motivo. A modo de ejemplo, se puede considerar un modelo de dos variables aleatorias, $x = (x_1, x_2) \sim p(x_1, x_2)$, donde x_1 es una variable aleatoria que indica si una persona lleva paraguas ($x_1 = 1$) o no ($x_1 = 0$), mientras que x_2 indica si llueve ($x_2 = 1$) o no ($x_2 = 0$). Por lo general, el valor más probable para x_1 es 0 (asumiendo que por lo general no llueve), pero esto cambia si se sabe que $x_2 = 1$ ya que, en este caso, aumenta considerablemente la probabilidad de que $x_1 = 1$. Es decir, la distribución de x_1 cambia si se conoce el valor de x_2, por lo que se dice que x_1 depende de x_2.

Dadas dos variables aleatorias, $x \sim p(x)$ e $y \sim p(y)$, la notación $p(x \mid y)$ corresponde a la distribución que sigue la variable aleatoria x cuando se conoce el valor de la variable aleatoria y. Esta función (de masa en el caso discreto y de densidad en el caso continuo) está definida como

$$p(x \mid y) := \frac{p(x, y)}{p(y)}$$

Notar que esta definición tiene sentido: conocer el valor de y limita el posible espacio de valores que puede tomar la variable x (lo cual se refleja imponiendo el valor de y en la distribución conjunta $p(x, y)$), mientras que la división por $p(y)$ busca normalizar la nueva distribución para x, recortando el espacio muestral únicamente a aquellos eventos donde y toma el valor indicado. Por ejemplo, en el caso anterior, la variable aleatoria que indica

si una persona lleva paraguas o no en un día lluvioso es $p(x_1 \mid x_2 = 1) = \frac{p(x_1, x_2=1)}{p(x_2=1)}$, donde $p(x_1, x_2 = 1)$ es la probabilidad de que la persona lleve paraguas (o no) y que además esté lloviendo, mientras que $p(x_2 = 1)$ es la probabilidad de que esté lloviendo, sin considerar si la persona lleva paraguas o no.

Dos variables aleatorias $x \sim p(x)$ e $y \sim p(y)$ se dicen **independientes** (denotado como $x \perp y$) si el valor de una no influye en el otro, es decir: $p(x \mid y) = p(x)$. Notar que, por definición de probabilidad condicional, esto indica que $p(x) = \frac{p(x,y)}{p(y)}$, por lo que $x \perp y$ es equivalente a que $p(x, y) = p(x)p(y)$. Esto muestra, además, que la independencia es simétrica, es decir, si $x \perp y$, entonces $y \perp x$ (i.e., $p(y \mid x) = p(y)$). Si dos variables aleatorias no son independientes, se dice que son **dependientes**. Por ejemplo, si x es una variable aleatoria discreta asociada a lanzar una moneda e y es una variable aleatoria discreta asociada a lanzar un dado, dado que ambas variables son independientes entre sí, la probabilidad de que la moneda salga cara y en el dado salga un 4 es $\mathbb{P}(x = \text{cara}, y = 4) = \mathbb{P}(x = \text{cara}) \cdot \mathbb{P}(y = 4) = \frac{1}{2} \cdot \frac{1}{6} = \frac{1}{12}$, lo cual es consistente con la probabilidad que se obtiene al trabajar con la distribución conjunta, donde el total de pares $(\text{moneda}, \text{dado})$ posibles es $2 \cdot 6 = 12$, mientras que la cantidad de pares favorables es 1 (solo uno de los pares tiene $x = \text{cara} \wedge y = 4$), por lo que, por la regla de Laplace, $\mathbb{P}(x = \text{cara}, y = 4) = \frac{1}{12}$.

Cuando se tiene una **distribución conjunta** $p(x, y)$ (i.e., la función de masa o de densidad asociada a cada combinación (x, y) de valores que pueden tomar las variables aleatorias), es posible conocer la **distribución marginal** $p(x)$, la cual corresponde a la distribución que sigue la variable aleatoria x de forma independiente, sin saber el valor que toma y (lo que ocurre, por ejemplo, cuando y es una variable oculta). Para esto, es necesario sumar (o integrar en el caso continuo) sobre todos los posibles valores que puede tomar la variable aleatoria y. En el caso discreto, si $\text{supp}(y) = \{y_1, ..., y_N\}$:

$$p(x) = \sum_{n=1}^{N} p(x, y = y_n)$$

Por ejemplo, para la gaussiana bidimensional usada más arriba, donde $x \sim \mathcal{N}(\mu, \Sigma)$ con $\mu = (-1, 2)^\top$ y $\Sigma = \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix}$, ambas distribuciones marginales, $p(x_1)$ y $p(x_2)$, siguen una distribución gaussiana, donde cada media está contenida en el vector de media μ y las varianzas vienen dadas en la diagonal de la matriz de covarianzas Σ. Esto mismo ocurre para la segunda variable aleatoria que se consideró, la cual tenía una matriz de covarianzas $\Sigma = \begin{pmatrix} 3 & 1 \\ 1 & 1 \end{pmatrix}$, mostrando que el único efecto de los valores fuera de la diagonal de Σ es rotar el elipsoide generado para inducir dependencia entre las componentes de la distribución. En la siguiente figura se vuelven a visualizar ambas funciones de densidad, donde además se agregan las densidades marginales de cada componente:

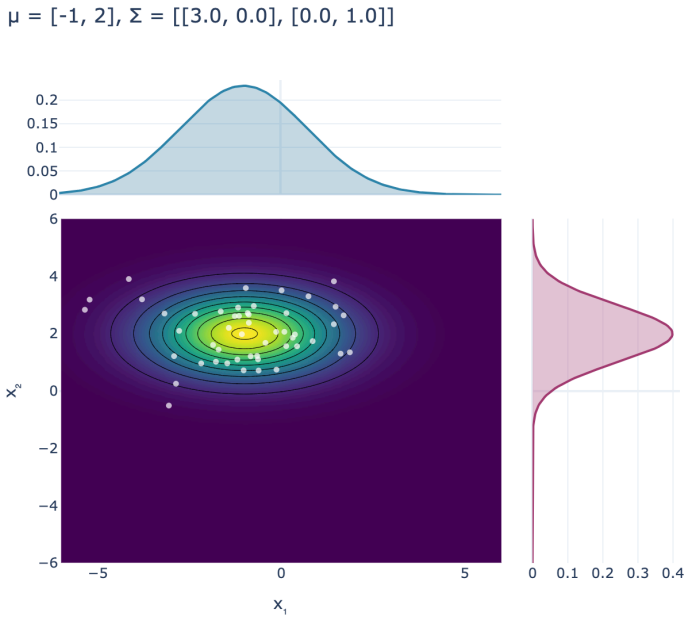

Figura 18: Gaussiana bidimensional con Σ diagonal y sus distribuciones marginales.

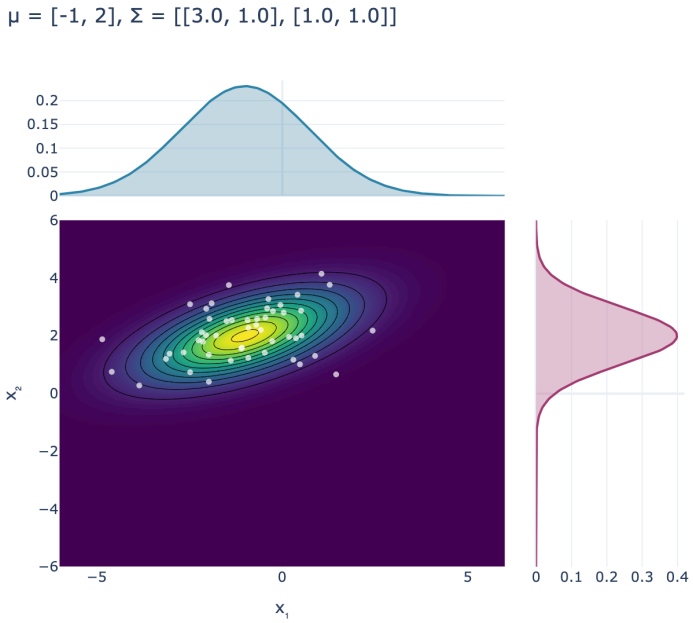

Figura 19: Gaussiana bidimensional con componentes dependientes y sus distribuciones marginales.

Se observa que las marginales en ambos casos coinciden, aunque la distribución conjunta no sea la misma. Esto muestra que la distribución conjunta $p(x, y)$ contiene más información

que las distribuciones individuales $p(x)$ y $p(y)$ ya que al marginalizar se pierden las nociones de dependencia entre ambas variables. Sin embargo, si las variables x e y son independientes, entonces $p(x, y) = p(x)p(y)$ por lo que, en este caso, las distribuciones marginales contienen la misma información que la distribución conjunta.

Una consecuencia directa de la definición de probabilidad condicional es el cálculo de distribuciones posteriores. Dado que $p(x, y) = p(x)p(y \mid x)$ y $p(x, y) = p(y)p(x \mid y)$, entonces se tiene que

$$p(y \mid x) = \frac{p(y)p(x \mid y)}{p(x)}$$

Esta propiedad se conoce como **regla de Bayes**, y permite calcular la **distribución posterior** $p(y \mid x)$ conociendo únicamente la distribución condicional opuesta, $p(x \mid y)$, y las marginales $p(x)$ y $p(y)$. Por ejemplo, un análisis estadístico simple puede permitir estimar la probabilidad de que una persona enferma tenga tos, por lo que, usando la regla de Bayes, se podría estimar la probabilidad de que una persona con tos esté enferma. Es importante mencionar que el hecho de usar la regla de Bayes no tiene relación alguna con ser bayesiano ya que esto último es simplemente una decisión filosófica acerca de cómo interpretar las probabilidades (más precisamente, de cómo interpretar los axiomas de Kolmogorov), donde la probabilidad representa el grado de confianza o creencia en la ocurrencia de un evento, a diferencia del enfoque frecuentista, donde la probabilidad de un evento corresponde a la regla de Laplace cuando la cantidad de experimentos tiende a infinito.

Por otro lado, la propiedad $p(x, y) = p(x)p(y \mid x)$ puede ser extendida a un conjunto de variables aleatorias, donde la distribución conjunta se descompone en un producto de probabilidades condicionales. En efecto, se puede probar por inducción que

$$\begin{aligned} p(x_1, ..., x_N) &= p(x_1)p(x_2 \mid x_1)p(x_3 \mid x_1, x_2) \cdots p(x_N \mid x_1, ..., x_{N-1}) \\ &= p(x_1) \prod_{n=2}^{N} p(x_n \mid x_1, ..., x_{n-1}) \end{aligned}$$

Esta propiedad se conoce como **regla de la cadena** (de las probabilidades), y no debe confundirse con la regla de la cadena de las derivadas.

Esperanza y varianza

En esta sección se estudiarán los conceptos de esperanza y varianza, los cuales son casos específicos de un concepto más general llamado momento. Tanto la esperanza como la varianza jugarán un rol fundamental en todos los paradigmas generativos que se estudiarán.

Dada una variable $x \sim p(x)$, su **esperanza** o **valor esperado** corresponde al valor que uno esperaría obtener al promediar muchas muestras generadas desde x. Si $x \sim p(x)$ es una variable discreta con $\text{supp}(x) = \{k_1, ..., k_N\}$, su esperanza se define como la suma ponderada sobre su soporte, donde el ponderador de cada sumando corresponde a la masa del respectivo átomo:

$$\mathbb{E}_{x\sim p(x)}[x] \coloneqq \sum_{n=1}^{N} k_n \, p(x = k_n)$$

Por ejemplo, si $x \sim \text{Bernoulli}(r)$, entonces $\mathbb{E}_{x\sim p(x)}[x] = 0 \cdot p(x = 0) + 1 \cdot p(x = 1) = r$, lo cual es lo esperado cuando se interpreta la distribución de Bernoulli como el lanzamiento de una moneda cargada. De forma más general, para una función genérica $f : \{k_1, ..., k_N\} \to \mathbb{R}$, se tiene que $\mathbb{E}_{x\sim p(x)}[f(x)] = \sum_{n=1}^{N} f(k_n) \cdot p(x = k_n)$.

En el caso continuo donde $p : \mathbb{R}^D \to \mathbb{R}_+$ es una función de densidad, la esperanza de la variable aleatoria $x \sim p(x)$ se define de forma análoga:

$$\mathbb{E}_{x\sim p(x)}[x] \coloneqq \int_{\mathbb{R}^D} x \, p(x) \, \mathrm{d}x$$

De forma general, se tiene que $\mathbb{E}_{x\sim p(x)}[f(x)] = \int_{\mathbb{R}^D} f(x) \, p(x) \, \mathrm{d}x$. A lo largo del libro, usualmente se denotará $\mathbb{E}_{x\sim p(x)}[x]$ como $\mathbb{E}_{p(x)}[x]$ para no sobrecargar la notación. Más aún, cuando no hay ambigüedad, se suele denotar simplemente como $\mathbb{E}[x]$, aunque se evitará esta notación ya que puede ser poco didáctica.

Por otro lado, si una variable aleatoria no participa dentro del argumento de la esperanza, se puede omitir su referencia:

$$\begin{aligned}
\mathbb{E}_{p(x,y)}[f(x)] &= \int_{\mathbb{R}^D \times \mathbb{R}^D} f(x) \, p(x, y) \, \mathrm{d}(x, y) \\
&= \int_{\mathbb{R}^D} \left(f(x) \, p(x) \int_{\mathbb{R}^D} p(y \mid x) \, \mathrm{d}y \right) \mathrm{d}x \\
&= \mathbb{E}_{p(x)}[f(x)]
\end{aligned}$$

En la segunda igualdad se usó que $p(x, y) = p(x)p(y \mid x)$ junto con el teorema de Fubini, mientras que en la tercera igualdad se usó que $\int_{\mathbb{R}^D} p(y \mid x) \, \mathrm{d}y = 1$ ya que $p(y \mid x)$ es una densidad de probabilidad (con respecto a y). Usando esta propiedad, es directo ver que la esperanza es un operador lineal. Para $\alpha, \beta \in \mathbb{R}$:

$$\mathbb{E}_{p(x,y)}[\alpha f(x) + \beta g(y)] = \alpha \, \mathbb{E}_{p(x)}[f(x)] + \beta \, \mathbb{E}_{p(y)}[g(y)]$$

De hecho, si x e y son variables independientes, también se tiene que $\mathbb{E}_{p(x,y)}[xy] = \mathbb{E}_{p(x)}[x]\,\mathbb{E}_{p(y)}[y]$ (la recíproca no es cierta: poder separar el producto no implica independencia). Además, la definición de esperanza permite escribir el proceso de marginalización descrito anteriormente como una esperanza:

$$\begin{aligned} p(x) &= \int_{\mathbb{R}^D} p(x, y)\,\mathrm{d}y \\ &= \int_{\mathbb{R}^D} p(x \mid y)\,p(y)\,\mathrm{d}y \\ &= \mathbb{E}_{p(y)}[p(x \mid y)] \end{aligned}$$

Esta expresión tiene sentido ya que consiste en promediar todas las posibles densidades que tendría x para los distintos valores que puede tomar y.

Un problema que muchas veces provoca confusión es reconocer cuál es la variable aleatoria con la que se está tomando esperanza. Por ejemplo, en la expresión $\mathbb{E}_{p(x|y)}[f(x, y)]$, la variable aleatoria es x y su distribución, $p(x \mid y)$, asume que y es un valor fijo y conocido (i.e., $\mathbb{E}_{p(x|y)}[f(x, y)]$ es una función de y). Si y también es una variable aleatoria con $y \sim p(y)$, se puede promediar la esperanza anterior sobre los posibles valores de y, lo que resulta ser equivalente a calcular la esperanza de la distribución conjunta:

$$\begin{aligned} \mathbb{E}_{p(y)}\left[\mathbb{E}_{p(x|y)}[f(x, y)]\right] &= \int_{\mathbb{R}^D} \left(\int_{\mathbb{R}^D} f(x, y)\,p(x \mid y)\,\mathrm{d}x \right) p(y)\,\mathrm{d}y \\ &= \int_{\mathbb{R}^D \times \mathbb{R}^D} f(x, y)\,p(x, y)\,\mathrm{d}(x, y) \\ &= \mathbb{E}_{p(x,y)}[f(x, y)] \end{aligned}$$

Más en general, la esperanza sobre un conjunto de variables aleatorias se puede factorizar de manera análoga a como se pueden factorizar las distribuciones conjuntas usando la regla de la cadena. Esta propiedad será importante para el desarrollo de los modelos generativos ya que permitirá, por ejemplo, escribir expresiones como

$$\mathbb{E}_{p(x_1,x_2,x_3)}[f(x_1, x_2, x_3)] = \mathbb{E}_{p(x_1)}\left[\mathbb{E}_{p(x_2|x_1)}\left[\mathbb{E}_{p(x_3|x_1,x_2)}[f(x_1, x_2, x_3)]\right]\right]$$

Dada la importancia de la esperanza en el desarrollo de los modelos generativos (y en machine learning en general), es importante poder calcular esta cantidad de forma eficiente (e.g., para entrenar una red neuronal). Si bien muchas veces esto no es posible, la ley de los grandes números permite aproximar esta cantidad utilizando muestras de la distribución

con respecto a la que se calcula la esperanza. Más precisamente, si $x \sim p(x)$ es una variable aleatoria desde la cual es fácil generar muestras, entonces

$$\mathbb{E}_{p(x)}[f(x)] \approx \frac{1}{N}\sum_{n=1}^{N} f(x^n), \quad \text{donde} \quad x^n \sim p(x), \text{para todo } n \in \{1, ..., N\}$$

Esta aproximación se suele llamar **estimación de Monte Carlo**, y solo requiere poder generar muestras desde $p(x)$. Además, este promedio es un estimador insesgado y consistente para la esperanza $\mathbb{E}_{p(x)}[f(x)]$, por lo que la aproximación mejora a medida que se utilizan más muestras desde $p(x)$. Sin embargo, en los modelos que se revisarán será usual utilizar una única muestra para la estimación.

La **varianza** de una variable aleatoria $x \sim p(x)$ en $\mathbb{R}$ corresponde a la desviación (cuadrática) media de la variable aleatoria con respecto a su media $\mu = \mathbb{E}_{p(x)}[x]$. Por lo tanto, la varianza es una medida de dispersión o aleatoriedad de la variable aleatoria:

$$\begin{aligned}
\mathbb{V}(x) &:= \mathbb{E}_{p(x)}\left[(x-\mu)^2\right] \\
&= \mathbb{E}_{p(x)}\left[x^2 - 2x\mu + \mu^2\right] \\
&= \mathbb{E}_{p(x)}\left[x^2\right] - 2\mu\,\mathbb{E}_{p(x)}[x] + \mathbb{E}_{p(x)}\left[\mu^2\right] \\
&= \mathbb{E}_{p(x)}\left[x^2\right] - \mu^2
\end{aligned}$$

La primera igualdad es la definición usual de varianza, mientras que la última igualdad es una expresión equivalente que resulta ser útil en algunos casos. A modo de ejemplo, si $x \sim \text{Bernoulli}(r)$, recordando que $\mathbb{E}_{p(x)}[x] = r$, entonces $\mathbb{V}(x) = (0-r)^2\, p(x=0) + (1-r)^2\, p(x=1) = r(1-r)$.

Una pregunta natural es por qué considerar en la definición la desviación de la media elevada a 2 y no considerar solamente la desviación absoluta $\mathbb{E}_{p(x)}[\![x-\mu]\!]$. La respuesta más simple a esto es porque es conveniente: definir la varianza usando un exponente cuadrático permite obtener buenas propiedades matemáticas. Por ejemplo, si x e y son variables aleatorias independientes, entonces se puede probar que $\mathbb{V}(\alpha x + \beta y) = \alpha^2\ \mathbb{V}(x) + \beta^2\ \mathbb{V}(y)$, lo cual no sería cierto si se considera otro exponente. Este fenómeno también se ve en otros lugares: al momento de definir la distribución gaussiana se utiliza una diferencia cuadrática dentro de la función exponencial y no simplemente el valor absoluto (como sí lo hace la distribución de Laplace) ya que el exponente cuadrático le da buenas propiedades a la distribución normal (e.g., diferenciabilidad). Esto también se observa al realizar regresión lineal basada en mínimos cuadrados, donde esta elección permite encontrar el regresor óptimo de forma cerrada.

Por otro lado, el concepto de varianza puede extenderse al concepto de covarianza, el cual permite comparar dos variables aleatorias. Si $x \sim p(x)$ e $y \sim p(y)$ son dos variables aleatorias en $\mathbb{R}$ con medias μ_x y μ_y respectivamente, la covarianza entre x e y mide el grado de desviación conjunta de cada variable con respecto a sus medias:

$$\text{Cov}(x, y) \coloneqq \mathbb{E}_{p(x,y)}\big[(x - \mu_x)(y - \mu_y)\big]$$

Notar que si x aumenta/disminuye su valor al mismo tiempo que y aumenta/disminuye su valor, entonces hay una covarianza positiva, mientras que si una de las variables aumenta su valor cuando la otra variable disminuye (o viceversa), entonces se tiene una covarianza negativa. En particular, $\text{Cov}(x, x) = \mathbb{V}(x) \geq 0$.

Por otra parte, notar que $\text{Cov}(x, y) = 0$ no implica necesariamente que $x \perp y$ ya que la covarianza solo puede captar relaciones lineales entre las variables (e.g., si $(x, y) \in \mathbb{R}^2$ representan coordenadas de un círculo unitario, $\text{Cov}(x, y) = 0$ pero las variables sí estarán relacionadas ya que $x^2 + y^2 = 1$).

En general, cuando se tiene un vector aleatorio $x = (x_1, ..., x_D) \sim p(x)$, es posible construir su matriz de covarianzas $\text{Cov}(x) \in \mathcal{M}_{D,D}(\mathbb{R})$ cuyas entradas son $\text{Cov}(x)_{ij} \coloneqq \text{Cov}\big(x_i, x_j\big)$. En particular, la diagonal de esta matriz contiene las varianzas de cada componente de x. Además, notando que $\text{Cov}\big(x_i, x_j\big) = \text{Cov}\big(x_j, x_i\big)$, se tiene que la matriz de covarianzas siempre es simétrica.

Por otro lado, la magnitud de $\text{Cov}(x, y)$ no indica el grado de relación entre las variables ya que la covarianza depende fuertemente de la escala de los valores que toman las variables x y y. Sin embargo, el **coeficiente de correlación de Pearson** normaliza esta cantidad para tener un valor más interpretable:

$$\rho(x, y) \coloneqq \frac{\text{Cov}(x, y)}{\sqrt{\mathbb{V}(x)}\sqrt{\mathbb{V}(y)}} \in [-1, 1]$$

En esta definición, la cantidad $|\rho(x, y)| \in [0, 1]$ indica el grado de correlación entre las variables. Un valor $\rho(x, y)$ cercano a 1 indica una dependencia casi lineal entre las variables, mientras que un valor más cercano a 0 indica un patrón más difuso entre las variables aleatorias.

1.2.2 Redes neuronales

Una **red neuronal** es una función $f : \mathbb{R}^D \to \mathbb{R}^C$ que se puede entrenar (i.e., se puede cambiar su comportamiento) para aprender a replicar otra función $f_{p_{\text{data}}} : \mathbb{R}^D \to \mathbb{R}^C$, usualmente desconocida, utilizando un conjunto de entrenamiento $\mathcal{D} = \big\{(x^1, y^1), ..., (x^N, y^N)\big\} \subset$

$\mathbb{R}^D \times \mathbb{R}^C$, donde cada instancia de entrenamiento (x^n, y^n) es de la forma $y^n = f_{p_{\text{data}}}(x^n)$. De este modo, cuando la red neuronal f esté entrenada, esta tendrá un comportamiento similar a la función desconocida $f_{p_{\text{data}}}$ (i.e., $f \approx f_{p_{\text{data}}}$ en algún sentido preciso).

En general, una red neuronal está formada por la composición de $K \in \mathbb{N}$ funciones individuales, $f_k : \mathbb{R}^{D_{k-1}} \to \mathbb{R}^{D_k}$ (con $D_0 = D$ y $D_K = C$), es decir:

$$f(x) = f_K(f_{K-1}(...(f_1(x))...))$$

Las funciones $f_k : \mathbb{R}^{D_{k-1}} \to \mathbb{R}^{D_k}$ suelen estar definidas por expresiones matemáticas simples (e.g., transformaciones afines) y suelen llamarse **bloques** o **capas**, por lo que al entero $K > 1$ se le llama **número de capas** de la red neuronal. Cada capa f_k tiene asociado un conjunto de **parámetros** θ_k que definen el comportamiento de la función, y que van cambiando durante el entrenamiento de la red neuronal. Sin pérdida de generalidad, siempre se asumirá que los parámetros de cada capa f_k son vectores en $\mathbb{R}^{P_k}$. De esta forma, se dice que la red neuronal tiene $P = \sum_{k=1}^{K} P_k$ parámetros. Más aún, concatenando los K parámetros en un único vector, se puede considerar que $\theta = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_K \end{pmatrix} \in \mathbb{R}^P$ es el **vector de parámetros** de la red neuronal, lo que motiva a utilizar la notación f_θ para indicar una red neuronal con (vector de) parámetros θ.

A lo largo del libro, las redes neuronales serán, por lo general, usadas para ajustar los parámetros de una distribución de probabilidad $p(x)$ (notar el alcance de nombre entre los **parámetros** de la red neuronal y los **parámetros** de la distribución). Por ejemplo, si $p_{\text{data}}(x) \sim \mathcal{N}(\mu_{\text{data}}, \Sigma_{\text{data}})$ es una variable aleatoria gaussiana con parámetros μ_{data} y Σ_{data} desconocidos, se podrían usar dos redes neuronales, μ_θ y Σ_θ, para estimar estos parámetros. Para esto será suficiente contar con un conjunto de entrenamiento $\mathcal{D} = \{x^1, ..., x^N\}$ compuesto por muestras generadas a partir de la distribución $p_{\text{data}}(x)$. De esta forma, si la red queda bien entrenada, la distribución $p_\theta(x) \sim \mathcal{N}(\mu_\theta, \Sigma_\theta)$ sería una buena aproximación de la distribución desconocida $p_{\text{data}}(x)$.

Entrenamiento

Entrenar una red neuronal $f_\theta : \mathbb{R}^D \to \mathbb{R}^C$ usando un conjunto de entrenamiento $\mathcal{D} = \{(x^1, y^1), ..., (x^N, y^N)\} \subset \mathbb{R}^D \times \mathbb{R}^C$ significa adaptar sus parámetros, $\theta \in \mathbb{R}^P$, de tal forma que $f_\theta(x^n) \approx y^n$, para todo $n \in \{1, ..., N\}$. Este proceso se llama (pre-)**entrenamiento** y, para los grandes modelos actuales, suele requerir una gran cantidad de datos de entrenamiento.

Para realizar el entrenamiento es necesario elegir una función de pérdida $L : \mathbb{R}^C \times \mathbb{R}^C \to \mathbb{R}$ que mida la discrepancia entre el valor real, $y \in \mathbb{R}^C$, y el predicho por la red neuronal, $\hat{y} \in \mathbb{R}^C$ (e.g., $L(\hat{y}, y) = \|y - \hat{y}\|^2$) para una entrada $x \in \mathbb{R}^D$ cualquiera. Con esto, para cada par

$(x^n, y^n) \in \mathcal{D}$, se compara la salida de la red neuronal, $\hat{y}^n = f_\theta(x^n)$, con el valor real, y^n. Promediando (o sumando) las pérdidas individuales se puede obtener el valor de la función de pérdida total:

$$\mathcal{L}_\mathcal{D}(\theta) = \frac{1}{N} \sum_{n=1}^{N} L(f_\theta(x^n), y^n)$$

Con esta función definida, se pueden ajustar los parámetros de la red neuronal de tal forma que minimicen el valor de $\mathcal{L}_\mathcal{D}(\theta)$. Para esto, lo más usual es usar algoritmos de gradiente, donde el más simple de estos algoritmos es **descenso del gradiente**, el cual comienza con un vector de parámetros $\theta \in \mathbb{R}^P$ aleatorio y luego realiza iteraciones de la forma

$$\theta \leftarrow \theta - \gamma \, \nabla_\theta \mathcal{L}_\mathcal{D}(\theta)$$

La constante $\gamma > 0$ se llama **tasa de aprendizaje** y define a qué velocidad se actualizan los parámetros durante el entrenamiento (también es posible considerar una tasa de aprendizaje individual para cada parámetro). De manera intuitiva, el algoritmo de descenso del gradiente cumple su función de minimizar $\mathcal{L}_\mathcal{D}(\theta)$ debido a que el gradiente $\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) \in \mathbb{R}^P$ apunta en la dirección de máximo crecimiento de $\mathcal{L}_\mathcal{D}(\theta) \in \mathbb{R}$, por lo que $-\nabla_\theta \mathcal{L}_\mathcal{D}(\theta)$ apunta en la dirección de máximo decrecimiento. Con esto, si $\gamma > 0$, entonces $\theta - \gamma \, \nabla_\theta \mathcal{L}_\mathcal{D}(\theta)$ es un nuevo vector de parámetros que entrega un menor valor para la función de pérdida $\mathcal{L}_\mathcal{D}$.

Otros optimizadores (e.g., Adam [27] o Prodigy [28]) cambian la expresión matemática de las iteraciones, pero siempre necesitan conocer el gradiente $\nabla_\theta \mathcal{L}_\mathcal{D}(\theta)$, el cual es computado utilizando la regla de la cadena (de la derivada) sobre la red neuronal $f_\theta(x) = f_K(f_{K-1}(...(f_1(x))...))$. La forma usual de realizar este cálculo en cadena es mediante un algoritmo de programación dinámica llamado **backpropagation**, el cual viene implementado en todas las librerías de Python enfocadas en redes neuronales y diferenciación automática (e.g., PyTorch, TensorFlow o JAX).

Redes fully connected

Dependiendo del tipo de capa neuronal $f_k : \mathbb{R}^{D_{k-1}} \to \mathbb{R}^{D_k}$ que se utilice, la red neuronal recibe diferentes nombres. Por ejemplo, si todas las capas son de la forma $f_k(x) = \Phi(A_k x + b_k)$ (con Φ una función no lineal), entonces la red se dice **fully connected**, mientras que si en vez de transformaciones afines generales se utilizan convoluciones, la red se dice **convolucional**.

Para ejemplos simples, la red fully connected es más que suficiente, por lo que suele ser el tipo de red neuronal que se usa como baseline, mientras que para trabajar con imágenes, donde $D = \text{ancho} \times \text{alto} \times \text{n° canales}$, las redes convolucionales funcionan mejor. Por otro lado, para trabajar con secuencias de texto o de otro tipo, se suele utilizar otro tipo de

arquitectura, como las arquitecturas recurrentes (e.g., GRU [29]). Sin embargo, hoy en día, las arquitecturas recurrentes han sido sustituidas por la arquitectura **Transformer** [14], la cual puede considerarse como uno de los trabajos más importantes en el deep learning moderno (aunque el paper original no sirve para predecir todo el potencial que tuvo a futuro).

El tipo de red neuronal más simple es la red **fully connected** (FC), también llamada **multilayer perceptron** (MLP). En este tipo de red neuronal cada bloque $f_k : \mathbb{R}^{D_{k-1}} \to \mathbb{R}^{D_k}$ es de la forma

$$f_k(x) = \Phi(A_k x + b_k),$$

donde la **matriz de pesos** $A_k \in \mathcal{M}_{D_k, D_{k-1}}(\mathbb{R})$ y el **vector de sesgos** $b_k \in \mathbb{R}^{D_k}$ son los parámetros de la capa lineal f_k (cada capa tiene $D_k \times D_{k-1} + D_k$ parámetros). La función $\Phi : \mathbb{R}^{D_k} \to \mathbb{R}^{D_k}$ suele ser una misma función escalar (no lineal) que se aplica en cada una de las coordenadas del **vector de pre-activación**, $\hat{y}_k = A_k x + b_k \in \mathbb{R}^{D_k}$, es decir: $\Phi(y) = \Big(\phi(y_1), ..., \phi\Big(y_{D_k}\Big)\Big)$, donde $\phi : \mathbb{R} \to \mathbb{R}$ es una función no lineal fija que se aplica en cada coordenada de $y \in \mathbb{R}^{D_k}$, llamada **función de activación**.

Las dimensiones intermedias, $D_1, ..., D_{K-1}$, pueden ser todas iguales, decrecer y luego volver a crecer (como en un autoencoder) o seguir otro tipo de patrón. Por otro lado, en muchos modelos, el vector de sesgo $b_k \in \mathbb{R}^{D_k}$ se suele omitir (haciendo esto, la GPU puede realizar los cálculos más eficientemente), mientras que la matriz de pesos $A_k \in \mathcal{M}_{D_k, D_{k-1}}(\mathbb{R})$ se suele inicializar con un valor cercano a cero, donde la inicialización de Xavier [30] y la inicialización de He [31] son las más usadas.

Una forma natural de motivar este tipo de redes neuronales es pensar en el problema de encontrar el patrón más simple posible (basándose en el principio de parsimonia) que se pueda repetir varias veces para lograr ajustar cualquier función. Se puede demostrar que la función más simple que cumple este requisito es la función afín $f(x) = Ax + b$. Más precisamente, se puede demostrar que este tipo de redes neuronales puede aproximar (en el sentido de la convergencia uniforme sobre compactos) a cualquier función continua $f : \mathbb{R}^D \to \mathbb{R}^C$ si es que no se pone límite en el número de parámetros. Este tipo de resultados se conocen como **teoremas de aproximación universal**, y hoy en día se conocen muchas extensiones y mejoras al resultado mencionado.

Es importante destacar que este tipo de red neuronal se puede extender fácilmente a más entradas, lo cual será útil al estudiar **modelos generativos condicionales**, donde las entradas adicionales son interpretadas como información adicional para la generación (e.g., un prompt para un modelo que genera imágenes o una imagen para tareas de transferencia de estilo). La forma usual de insertar información adicional a una red neuronal es inyectándola directamente en cada uno de los bloques de la red. Así, si la información adicional

es un vector $c \in \mathbb{R}^J$, cada bloque $f_k : \mathbb{R}^{D_{k-1}} \to \mathbb{R}^{D_k}$ se puede sustituir por otro bloque $f_k : \mathbb{R}^{D_{k-1}} \times \mathbb{R}^J \to \mathbb{R}^{D_k}$ que ahora también procese la entrada adicional c. En el caso de una red fully connected, este entrada adicional puede ser inyectada de la misma forma que se procesa la entrada original, $x \in \mathbb{R}^{D_{k-1}}$, añadiendo una matriz adicional de parámetros:

$$f_k(x, c) = \Phi(A_k x + B_k c + d_k),$$

donde $A_k \in \mathcal{M}_{D_k, D_{k-1}}(\mathbb{R})$, $B_k \in \mathcal{M}_{D_k, J}(\mathbb{R})$ y $d_k \in \mathbb{R}^{D_k}$ son los parámetros del bloque f_k.

Es importante notar que esta forma de calcular $f_k(x, c)$ es equivalente a concatenar los vectores de entrada, $x \in \mathbb{R}^{D_{k-1}}$ y $c \in \mathbb{R}^J$, y trabajar con un único vector extendido $\tilde{x} = \binom{x}{c} \in \mathbb{R}^{D_{k-1}+J}$ definiendo $f_k(\tilde{x}) = \Phi\left(\tilde{A}_k \tilde{x} + d_k\right)$, donde $\tilde{A}_k = (A_k \mid B_k) \in \mathcal{M}_{D_k, D_{k-1}+J}(\mathbb{R})$ es la concatenación horizontal de las matrices A_k y B_k. Esta observación es importante ya que es usual encontrarse con ambos tipos de implementaciones, por lo que es importante reconocer que son equivalentes.

Clasificación con redes neuronales

En esta sección se implementará, a modo de ejemplo, una red neuronal para resolver el problema de clasificación utilizando un dataset de entrenamiento $\mathcal{D} = \{(x^1, y^1), ..., (x^N, y^N)\}$ compuesto por imágenes $x \in \mathbb{R}^D$ y etiquetas de clase $y \in \{M_1, ..., M_C\}$ (e.g., $y = M_c$ indica que la imagen x pertenece a la clase número c dentro de un conjunto de C clases). Para formular este problema usando redes neuronales, es usual considerar que la etiqueta de cada muestra $x \in \mathbb{R}^D$ fue generada a partir de una distribución condicional desconocida, $p_{\text{data}}(y \mid x)$. Notar que la variable aleatoria y (condicional a una imagen x) es discreta ya que $\text{supp}(y) = \{M_1, ..., M_C\}$, por lo que se puede considerar, sin pérdida de generalidad, que $p_{\text{data}}(y \mid x) \sim \text{Categorical}(r_{\text{data}})$, donde $r_{\text{data}} \in \Delta^C$ es un vector de probabilidad desconocido.

Dado que el objetivo del problema de clasificación es predecir a qué clase $y \in \{M_1, ..., M_C\}$ pertenece una imagen $x \in \mathbb{R}^D$ dada, es suficiente entrenar una red neuronal $r_\theta : \mathbb{R}^D \to [0, 1]^C$ que defina el vector de probabilidades de un modelo $p_\theta(y \mid x) \sim \text{Categorical}(r_\theta(x))$ utilizando el conjunto de entrenamiento $\mathcal{D}$. Este tipo de modelos, llamados **modelos discriminativos**, utiliza un tipo de aprendizaje conocido como **aprendizaje supervisado**, donde cada muestra del dataset, $x \in \mathbb{R}^D$, tiene asociada una etiqueta, $y \in \{M_1, ..., M_C\}$.

Es importante notar que para cualquier imagen de entrada $x \in \mathbb{R}^D$, la salida de la red neuronal, $r_\theta(x)$, debe ser un vector de probabilidad (i.e., $r_\theta(x)_c \geq 0$ para todo $c \in \{1, ..., C\}$ y $\sum_{c=1}^{C} r_\theta(x)_c = 1$), lo cual no necesariamente ocurre en la salida de una red neuronal, por ejemplo, fully connected. Para solucionar esto, es usual aplicar la función softmax a la salida de la red neuronal, la cual transforma todas las coordenadas en positivas (utilizando la función exponencial) y luego fuerza a que sumen 1 dividiendo cada coordenada por la

suma de todo el vector. Más precisamente, para un vector $r \in \mathbb{R}^C$ cualquiera, se define $\text{softmax} : \mathbb{R}^C \to \texttt{simplex}^C$ como

$$\text{softmax}(s) := \left(\frac{e^{s_1}}{\sum_{c=1}^{C} e^{s_c}}, ..., \frac{e^{s_C}}{\sum_{c=1}^{C} e^{s_c}} \right)$$

donde es directo ver que $\text{softmax}(s)_c \in [0, 1]$ para todo $c \in \{1, ..., C\}$ y que $\sum_{c=1}^{C} \text{softmax}(s)_c = 1$, por lo que efectivamente $r = \text{softmax}(s) \in \texttt{simplex}^C$.

Para implementar y entrenar una red neuronal para el problema de clasificación de imágenes se utilizarán las siguientes librerías:

```
import random
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchsummary import summary
```

El objetivo será entrenar un clasificador para el dataset Fashion MNIST [32], el cual consiste en imágenes de $C = 10$ tipos diferentes de prendas de vestir:

```
dataset = datasets.FashionMNIST('data', download=True,
transform=transforms.ToTensor(), train=True)
labels = ['Camiseta/Top', 'Pantalón', 'Suéter', 'Vestido', 'Abrigo',
'Sandalia', 'Camisa', 'Zapatilla', 'Bolso', 'Botín']

print('Tamaño dataset:', len(dataset))
print('Cantidad de clases:', len(labels))
```

```
Tamaño dataset: 60000 Cantidad de clases: 10
```

Las etiquetas dentro de las muestras por lo general están asignadas con números enteros (del 0 al 9 en este caso), por lo que el nombre real de las clases deben ser obtenidos desde la lista `labels`. Por otro lado, la transformación `transforms.ToTensor()` es aplicada a cada elemento x del dataset y su objetivo es transformar una imagen (objeto tipo `PIL.Image.Image`) en un objeto tipo `torch.Tensor`, el cual puede verse como la versión PyTorch de los `np.ndarray` de NumPy (la principal diferencia es que los tensores de PyTorch pueden rastrear el grafo computacional para el cálculo de los gradientes usados

durante el entrenamiento). Además, la transformación `transforms.ToTensor()` escala linealmente todos los pixeles de x al intervalo $[0, 1]$.

Para acceder a una muestra del dataset, basta con recorrer sus índices:

```
n = random.randint(0, len(dataset))
x, y = dataset[n]

print(f'x | Tipo: {type(x)} - dimensiones: {x.shape}')
print(f'y | Tipo: {type(y)} - valor: {y}')

plt.imshow(x.permute(1,2,0), cmap='gray_r')
plt.title(f'Etiqueta: {labels[y]}')
plt.axis('off')
plt.show()
```

```
x | Tipo: <class “torch.Tensor”> - dimensiones: torch.Size([1, 28,
28]) y | Tipo: <class “int”> - valor: 9
```

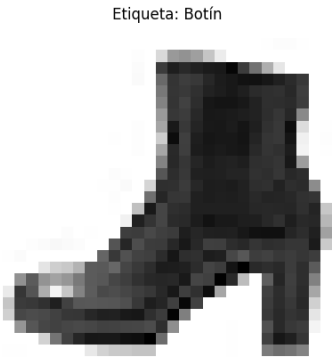

Figura 20: Ejemplo de muestra del dataset Fashion MNIST.

Notar que PyTorch coloca la dimensión de los canales al comienzo del tensor `x`, por lo que es necesario permutar esa dimensión para poder visualizar la imagen con `matplotlib` (que espera la dimensión de los canales al final).

Para el entrenamiento de la red neuronal, es usual usar sub-conjuntos del dataset de entrenamiento $\mathcal{D}$ (llamados batches) en vez de todo el conjunto en cada actualización de parámetros. Esta variante del algoritmo del descenso del gradiente se conoce como **descenso del gradiente estocástico** (SGD) y por lo general permite entrenamientos más eficientes. En PyTorch, la clase que genera los batches de entrenamiento se llama `DataLoader`:

```
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

x, y = next(iter(dataloader))
print('Dimensiones x:', x.size())
print('Dimensiones y:', y.size())
```

```
Dimensiones x: torch.Size([32, 1, 28, 28]) Dimensiones y:
torch.Size([32])
```

Notar que el dataloader retorna un batch de muestras y un batch de etiquetas. En ambos casos, la dimensión del batch siempre es la primera dimensión ya que las redes neuronales de PyTorch (casi) siempre esperan entradas con ese formato.

Como red neuronal se utilizará una red fully connected de 2 capas, donde la dimensión interna (valor D_1) será indicada por el parámetro `hidden_dim`:

```
class MLP(nn.Module):

    def __init__(self, input_dim: int, output_dim: int, hidden_dim: int) ->
None:
        super().__init__()
        self.fc1 = nn.Linear(in_features=input_dim,
out_features=hidden_dim)
        self.fc2 = nn.Linear(in_features=hidden_dim,
out_features=output_dim)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = nn.Flatten()(x)
        x = nn.ReLU()(self.fc1(x))
        x = nn.Softmax(dim=-1)(self.fc2(x))
        return x
```

- Ejemplo de uso.

```
input_dim, output_dim, hidden_dim, batch_size = 64, 8, 32, 256

mlp = MLP(input_dim, output_dim, hidden_dim)
x = torch.rand([batch_size, input_dim])
y = mlp(x)

assert y.shape == torch.Size([batch_size, output_dim])
```

Dado que las entradas son imágenes, la dimensión de entrada será el producto entre la resolución y la cantidad de canales que tiene la imagen:

```
first_image, first_label = dataset[0]
channels, height, width = first_image.size()

input_dim = channels * height * width
output_dim = len(labels)
hidden_dim = 4

mlp = MLP(input_dim, output_dim, hidden_dim)
summary(mlp, input_size=(input_dim,))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                    [-1, 4]           3,140
            Linear-2                   [-1, 10]              50
================================================================
Total params: 3,190
Trainable params: 3,190
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.01
Estimated Total Size (MB): 0.02
----------------------------------------------------------------
```

Figura 21: Resumen de parámetros de la red neuronal.

Notar que la cantidad de parámetros es consistente: la primera capa lineal tiene $4 \times 784 + 4 = 3\,140$ parámetros (donde $784 = 1 \times 28 \times 28$), mientras que la segunda capa tiene $10 \times 4 + 10 = 50$ parámetros. Además, considerando que cada parámetro ocupa 32 bits de memoria (precisión por defecto en PyTorch), se necesitan $3\,190 \times \frac{32}{8} = 12\,760$ bytes ($\approx$ 12 kilobytes o 0.012 megabytes) de memoria para almacenar el valor de sus parámetros. Sin embargo, la cantidad de memoria para almacenar un modelo moderno puede escalar hasta los cientos de gigabytes (e.g., LLaMA 3.1-405B requiere 1 620 gigabytes en precisión completa), mientras que la cantidad de memoria necesaria (en GPU) para entrenar un modelo puede ser varias veces la memoria necesaria para solo hacer inferencia debido a que durante el entrenamiento también es necesario almacenar otros tensores asociados al cálculo de gradientes y a valores temporales del optimizador.

La función de pérdida que se utilizará será la entropía cruzada [33], la cual es una función que permite comparar dos distribuciones de probabilidad (en este caso, $p_\theta(y \mid x)$ y $p_{\text{data}}(y \mid x)$). Esta función de pérdida será estudiada más en detalle cuando se implemente un modelo autorregresivo ya que es la función de pérdida estándar para entrenar modelos de lenguaje.

A cada iteración completa que se le de al dataset de entrenamiento $\mathcal{D}$ se le suele llamar **época**, y la cantidad de épocas que se entrena cada modelo depende siempre del tipo de datos y el tamaño del modelo. En este caso, será suficiente con entrenar el modelo una época. Además, el entrenamiento se puede realizar en CPU o en GPU, donde la segunda opción siempre es la preferida para grandes modelos debido a que acelera considerablemente el entrenamiento. Para esto se define la variable `DEVICE` que se fija como `cuda` (GPU de Nvidia) cuando sea posible.

El loop de entrenamiento es el siguiente:

```
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def train(net: nn.Module, optimizer: optim.Optimizer, dataloader:
DataLoader, epochs: int) -> None:

    net.to(DEVICE)
    net.train()

    loss_fn = nn.CrossEntropyLoss()

    for epoch in range(epochs):
        for x, y in dataloader:
            x, y = x.to(DEVICE), y.to(DEVICE)
            output = net(x)
            loss = loss_fn(output, y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

Como optimizador se elegirá Adam [34] ya que suele ser el optimizador usado por defecto. Para instanciarlo es necesario indicarle los parámetros que debe ir actualizando en cada iteración:

```
optimizer = torch.optim.Adam(mlp.parameters())
train(mlp, optimizer, dataloader, epochs=1)
```

Una vez el modelo está entrenado es posible utilizarlo para predecir la clase de una nueva imagen. Para esto, basta pasar la imagen por la red neuronal y elegir la clase con mayor probabilidad (aunque también se podría elegir aleatoriamente la clase de acuerdo a la distribución $p_\theta(y \mid x)$ que entrega la red neuronal). La siguiente función `make_prediction`

recibe una lista de imágenes y predice a qué clase pertenecen para así comparar la predicción con la clase real:

```
def make_prediction(net: nn.Module, images: list[torch.Tensor],
true_labels: list[int]) -> None:

    net.eval()
    plt.figure(figsize=(16, 7))

    with torch.no_grad():

        input_batch = torch.stack(images).to(DEVICE)
        output_batch = net(input_batch)
        label_preds = output_batch.argmax(dim=1)
        probs = output_batch.max(dim=1).values

        for i in range(len(images)):
            label_pred = label_preds[i].item()
            prob = probs[i].item()

            plt.subplot(2, 5, i + 1)
            plt.imshow(images[i].permute(1, 2, 0), cmap='gray_r')
            plt.axis('off')
            plt.title(f'Etiqueta real:
{labels[true_labels[i]]}\nPredicción: {labels[label_pred]}
({100*prob:.0f}%)')
```

Es importante mencionar que hacer inferencia utilizando imágenes desde el conjunto de entrenamiento no permite evaluar el overfitting del modelo. Sin embargo, se utilizará este mismo dataset por simplicidad. Para hacer inferencia, basta llamar a la función implementada:

```
indices = random.sample(range(len(dataset)), 10)

images = [dataset[i][0] for i in indices]
true_labels = [dataset[i][1] for i in indices]

make_prediction(mlp, images, true_labels)
```

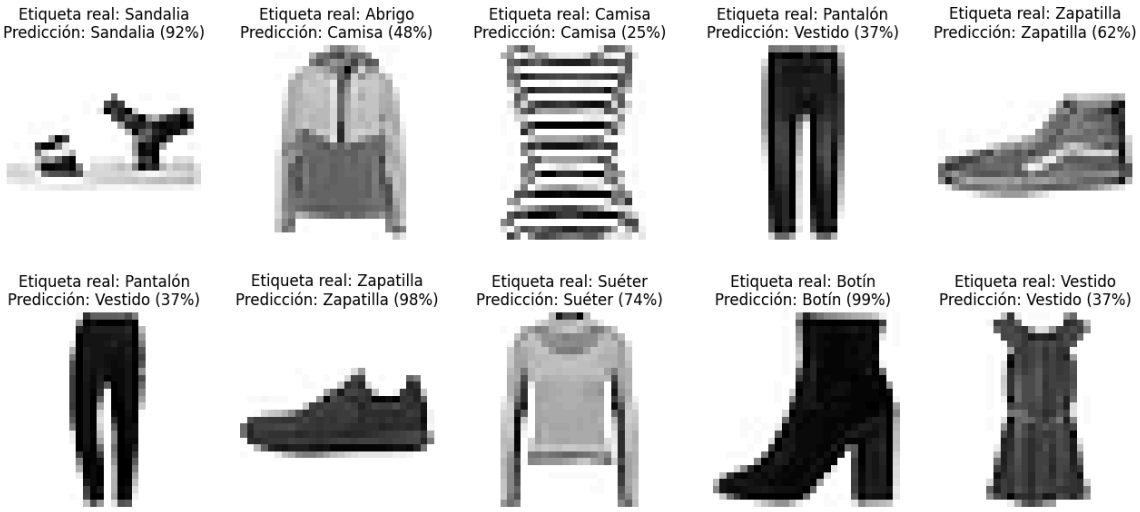

Figura 22: Predicciones realizadas por la red neuronal para 10 imágenes del dataset Fashion MNIST.

Se observa que la red neuronal fue capaz de aprender a clasificar las imágenes, con un cierto grado de error. Aumentando la cantidad de neuronas en `hidden_dim` se puede mejorar más esta precisión.

Es importante no olvidar que la salida de la red neuronal, $r_\theta(x) \in [0,1]^{10}$, es el vector de probabilidades de una distribución $p_\theta(y \mid x)$ sobre las clases y no únicamente una predicción de a qué clase pertenece x (aunque, en este caso, se realizó la predicción eligiendo la clase con mayor probabilidad). Esto es importante ya que en algunos modelos generativos, como los usados para la generación de texto, elegir siempre la clase con mayor probabilidad no suele ser la mejor opción.

Por otro lado, dado que se está trabajando con imágenes, resultaría más conveniente utilizar una red convolucional para procesar la entrada. Este tipo de redes neuronales será recordada al estudiar arquitecturas neuronales para GANs.

En el próximo capítulo se comenzará el estudio de las redes bayesianas, las cuales son el marco teórico general sobre el cual se pueden formular (casi) todos los modelos generativos estudiados a lo largo del libro.

Capítulo 2

Redes bayesianas

Dada la diversidad de paradigmas y la modularidad de los modelos actuales, resulta útil realizar el estudio bajo la perspectiva unificada de los **modelos gráficos probabilísticos**, lo que permite obtener un entendimiento más holístico de la IA generativa moderna y entender más fácilmente las similitudes y diferencias que hay entre los distintos paradigmas generativos. Por otro lado, este enfoque es natural dado que todos los modelos generativos modernos suelen ser formulados y analizados en términos probabilísticos. Más aún, todos los paradigmas que se estudiarán (a excepción de los modelos basados en score) pueden ser vistos como un tipo particular de modelo gráfico denominado **red bayesiana**. En consecuencia, este capítulo comienza estudiando algunos aspectos importantes de este marco teórico, para luego revisar el problema de inferencia en redes bayesianas, el cual puede interpretarse como el proceso de entrenamiento en la jerga usual del machine learning.

2.1 Introducción a las redes bayesianas

Un modelo gráfico consiste, a grandes rasgos, en poder representar las relaciones de dependencia e independencia entre las variables de un modelo probabilístico utilizando un grafo. Esta representación visual permite entender rápidamente cómo están relacionadas las variables, ya sean visibles u ocultas, dentro de un modelo probabilístico complejo como, por ejemplo, en un modelo de Markov oculto usado para la tarea part-of-speech tagging en texto.

Dado un modelo probabilístico con distribución conjunta $p(x_1, ..., x_N)$, el modelo gráfico asociado a este modelo probabilístico es un grafo cuyos nodos representan las variables aleatorias $x_1, ..., x_N$ del modelo (hay un nodo por cada variable), mientras que la existencia de un arco entre dos nodos indica que hay una relación de dependencia (en el sentido probabilístico) entre las variables aleatorias asociadas a esos nodos.

Los modelos gráficos se pueden clasificar en dos grandes grupos, dependiendo del tipo de grafo que utilicen para representar al modelo probabilístico:

- **Red bayesiana o belief networks:** en este caso, se utiliza un grafo dirigido (i.e., cada arco tiene una dirección) para representar la relación de dependencia entre variables. En este tipo de modelos, un arco dirigido desde (el nodo) x_i hacia (el nodo) x_j indica que hay una relación de dependencia entre (la variable) x_i y (la variable)

x_j. Se verá que el grafo dirigido asociado a un modelo probabilístico siempre debe ser acíclico para que esta interpretación tenga sentido.

- **Redes de Markov o Markov random fields (MRF):** en este tipo de modelos se utiliza un grafo no dirigido (i.e., no se diferencia entre un nodo origen y un nodo destino), por lo que ya no hay una jerarquía en las relaciones de dependencia. En consecuencia, la factorización de la distribución conjunta para este tipo de modelos se vuelve un poco diferente ya que se debe realizar sobre los cliques maximales del grafo, mientras que los términos que se multiplican ya no se interpretan como distribuciones, si no que se interpretan como **potenciales**. Este tipo de modelos no es muy importante para la IA generativa moderna, por lo que solo se comentará brevemente al estudiar los modelos basados en energía. Es importante mencionar que varios modelos generativos clásicos como las **máquinas de Boltzmann** o las **redes de Hopfield** (Nobel de física, 2024) son modelos de este tipo.

2.1.1 Ejemplo inicial

Para introducir la idea de red bayesiana, se comenzará con un ejemplo que modela la probabilidad de aprobar o no un curso, considerando únicamente si un estudiante estudia o no para el examen, y su rendimiento en las preguntas el examen. Para esto, se considerarán 4 variables aleatorias binarias (i.e., Bernoulli), x_1, x_2, x_3 y x_4, las cuales representarán los siguientes escenarios:

- **Variable aleatoria x_1:** el estudiante estudia para el examen.
- **Variable aleatoria x_2:** el estudiante responde bien las preguntas teóricas.
- **Variable aleatoria x_3:** el estudiante responde bien las preguntas prácticas.
- **Variable aleatoria x_4:** el estudiante aprueba el curso.

Como es usual, se asociará la respuesta «sí» al valor 1 y la respuesta «no» al valor 0. El siguiente grafo dirigido representa la relación entre las 4 variables aleatorias que considera el modelo:

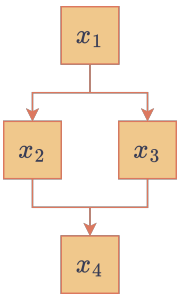

Figura 23: Red bayesiana del ejemplo del estudiante.

Este grafo indica que las variables aleatorias x_2 y x_3 dependen de x_1, mientras que la variable aleatoria x_4 depende de las variables aleatorias x_2 y x_3. Más precisamente, este grafo se interpreta factorizando la distribución conjunta de las 4 variables aleatorias como

$$p(x_1, x_2, x_3, x_4) = p(x_1)\, p(x_2 \mid x_1)\, p(x_3 \mid x_1)\, p(x_4 \mid x_2, x_3)$$

Es útil notar algunas independencias condicionales que induce esta factorización:

- La variable aleatoria x_3 es condicionalmente independiente, dado el valor de x_1, de la variable aleatoria x_2: $p(x_3 \mid x_1, x_2) = p(x_3 \mid x_1)$ o, equivalentemente, $p(x_2, x_3 \mid x_1) = p(x_2 \mid x_1)\, p(x_3 \mid x_1)$. Esta independencia indica que responder bien o no en las preguntas prácticas es independiente de responder bien o no en las preguntas teóricas, si es que se sabe si el estudiante estudió o no.
- La variable aleatoria x_4 es condicionalmente independiente de la variable aleatoria x_1, dados los valores de x_2 y x_3: $p(x_4 \mid x_1, x_2, x_3) = p(x_4 \mid x_2, x_3)$ o, equivalentemente, $p(x_1, x_4 \mid x_2, x_3) = p(x_1 \mid x_2, x_3)\, p(x_4 \mid x_2, x_3)$. Esta independencia indica que aprobar o no el examen no depende de si el estudiante estudió o no, si es que se sabe si respondió bien o no las preguntas teóricas y las preguntas prácticas.

Estas independencias condicionales se observan en la simplificación de la factorización de la distribución conjunta que se obtiene cuando se factoriza usando la regla de la cadena (la cual no asume ninguna independencia, solo utiliza la definición de probabilidad condicional):

$$p(x_1, x_2, x_3, x_4) = p(x_1)\, p(x_2 \mid x_1) \underbrace{p(x_3 \mid x_1, x_2)}_{p(x_3 \mid x_1)} \underbrace{p(x_4 \mid x_1, x_2, x_3)}_{p(x_4 \mid x_2, x_3)}$$

Notar que para definir cada una de las distribuciones condicionales, es necesario definir una distribución para cada posible valor de las variables condicionales. Por ejemplo, la distribución $p(x_4 \mid x_1, x_2, x_3)$ necesitaría definir $2 \cdot 2 \cdot 2 = 8$ distribuciones binarias sobre x_4 (una para cada posible valor de $(x_1, x_2, x_3) \in \{0, 1\}^3$), mientras que $p(x_4 \mid x_2, x_3)$ necesita definir solo $2 \cdot 2 = 4$ distribuciones binarias sobre x_4. Esto muestra cómo las hipótesis de independencia ayudan a reducir la cantidad de parámetros necesarios para definir una distribución conjunta.

Si se conocen todas las distribuciones de la factorización, es directo responder preguntas como ¿cuál es la probabilidad de responder bien las preguntas teóricas si se estudió? o ¿cuál es la probabilidad de aprobar el curso si respondió bien las preguntas prácticas y teóricas? ya que bastaría con mirar los parámetros de las distribuciones. Sin embargo, otras preguntas requieren hacer cálculos adicionales como marginalización o cálculo de posteriores. Algunas preguntas de este tipo son:

- ¿Cuál es la probabilidad de que el estudiante responda bien las preguntas teóricas?

- ¿Cuál es la probabilidad de que el estudiante haya estudiado si no respondió bien las preguntas teóricas?
- ¿Cuál es la probabilidad de responder bien tanto las preguntas teóricas como las preguntas prácticas?
- ¿Cuál es la probabilidad de responder bien las preguntas prácticas si se respondieron bien las preguntas teóricas?

Para responder las preguntas se asignarán probabilidades de juguete a cada una de las distribuciones que definen la probabilidad conjunta $p(x_1, x_2, x_3, x_4)$. Notar que la suma de las probabilidades de cada fila debe ser 1 para representar una distribución de probabilidad válida.

$p(x_1 = 0)$	$p(x_1 = 1)$
0.10	0.90

x_1	$p(x_2 = 0 \mid x_1)$	$p(x_2 = 1 \mid x_1)$
0	0.80	0.20
1	0.25	0.75

x_1	$p(x_3 = 0 \mid x_1)$	$p(x_3 = 1 \mid x_1)$
0	0.70	0.30
1	0.20	0.80

x_2	x_3	$p(x_4 = 0 \mid x_2, x_3)$	$p(x_4 = 1 \mid x_2, x_3)$
0	0	0.95	0.05
1	0	0.35	0.65
0	1	0.40	0.60
1	1	0.01	0.99

¿Cuál es la probabilidad de que el estudiante responda bien las preguntas teóricas?

Se debe calcular $p(x_2 = 1)$. Esto se puede hacer marginalizando la distribución conjunta:

$$
\begin{aligned}
p(x_2 = 1) &= \sum_{x_1, x_3, x_4 \in \{0,1\}} p(x_1, x_2 = 1, x_3, x_4) \\
&= \sum_{x_1, x_3, x_4 \in \{0,1\}} p(x_1)\, p(x_2 = 1 \mid x_1)\, p(x_3 \mid x_1)\, p(x_4 \mid x_2 = 1, x_3) \\
&= \sum_{x_1 \in \{0,1\}} \left[p(x_1)\, p(x_2 = 1 \mid x_1) \sum_{x_3 \in \{0,1\}} \left(p(x_3 \mid x_1) \sum_{x_4 \in \{0,1\}} p(x_4 \mid x_2 = 1, x_3) \right) \right] \\
&= \sum_{x_1 \in \{0,1\}} p(x_1)\, p(x_2 = 1 \mid x_1) \\
&= p(x_1 = 0)\, p(x_2 = 1 \mid x_1 = 0) + p(x_1 = 1)\, p(x_2 = 1 \mid x_1 = 1) \\
&= 0.10 \times 0.20 + 0.90 \times 0.75 \\
&= 0.695
\end{aligned}
$$

Es decir, sin observar ninguna variable a priori (e.g., sin saber si el estudiante estudió o no), es cerca de un 70% probable que el estudiante responda bien las preguntas teóricas. Notar que para pasar a la cuarta igualdad se utilizó que $\sum_{k \in \text{supp}(x)} p(x = k) = 1$ para la distribución $p(x_4 \mid x_2 = 1, x_3)$ y luego para la distribución $p(x_3 \mid x_1)$.

¿Cuál es la probabilidad de que el estudiante haya estudiado si no respondió bien las preguntas teóricas?

Se debe calcular $p(x_1 = 1 \mid x_2 = 0)$. Para esto, se puede usar la definición de probabilidad condicional y luego marginalizar la distribución conjunta:

$$
\begin{aligned}
p(x_1 = 1 \mid x_2 = 0) &= \frac{p(x_1 = 1, x_2 = 0)}{p(x_2 = 0)} \\
&= \frac{1}{p(x_2 = 0)} \sum_{x_3, x_4 \in \{0,1\}} p(x_1 = 1, x_2 = 0, x_3, x_4) \\
&= \frac{1}{p(x_2 = 0)} \sum_{x_3, x_4 \in \{0,1\}} p(x_1 = 1)\, p(x_2 = 0 \mid x_1 = 1)\, p(x_3 \mid x_1 = 1)\, p(x_4 \mid x_2 = 0, x_3) \\
&= \frac{1}{p(x_2 = 0)}\, p(x_1 = 1)\, p(x_2 = 0 \mid x_1 = 1) \sum_{x_3 \in \{0,1\}} \left(p(x_3 \mid x_1 = 1) \sum_{x_4 \in \{0,1\}} p(x_4 \mid x_2 = 0, x_3) \right) \\
&= \frac{p(x_1 = 1)\, p(x_2 = 0 \mid x_1 = 1)}{p(x_2 = 0)} \\
&= \frac{0.25 \times 0.90}{0.305} \\
&\approx 0.74
\end{aligned}
$$

En la penúltima igualdad se usó el resultado de la pregunta anterior para obtener $p(x_2 = 0) = 1 - p(x_2 = 1) = 0.305$.

Por otro lado, notar que esta distribución condicional no está en la factorización entregada por el grafo y, de hecho, tiene la dirección contraria al orden que induce el grafo. Por lo tanto, también se podría calcular esta probabilidad posterior utilizando la regla de Bayes:

$$p(x_1 = 1 \mid x_2 = 0) = \frac{p(x_2 = 0 \mid x_1 = 1)\, p(x_1 = 1)}{p(x_2 = 0)}$$

Llegando a la misma expresión que antes.

¿Cuál es la probabilidad de responder bien tanto las preguntas teóricas como las preguntas prácticas?

Se debe calcular $p(x_2 = 1, x_3 = 1)$. Esto se puede hacer, al igual que antes, marginalizando la distribución conjunta:

$$\begin{aligned}
p(x_2 = 1, x_3 = 1) &= \sum_{x_1, x_4 \in \{0,1\}} p(x_1, x_2 = 1, x_3 = 1, x_4) \\
&= \sum_{x_1, x_4 \in \{0,1\}} p(x_1)\, p(x_2 = 1 \mid x_1)\, p(x_3 = 1 \mid x_1)\, p(x_4 \mid x_2 = 1, x_3 = 1) \\
&= \sum_{x_1 \in \{0,1\}} \left[p(x_1)\, p(x_2 = 1 \mid x_1)\, p(x_3 = 1 \mid x_1) \sum_{x_4 \in \{0,1\}} p(x_4 \mid x_2 = 1, x_3 = 1) \right] \\
&= \sum_{x_1 \in \{0,1\}} p(x_1)\, p(x_2 = 1 \mid x_1)\, p(x_3 = 1 \mid x_1) \\
&= p(x_1 = 0)\, p(x_2 = 1 \mid x_1 = 0)\, p(x_3 = 1 \mid x_1 = 0) + p(x_1 = 1)\, p(x_2 = 1 \mid x_1 = 1)\, p(x_3 = 1 \mid x_1 = 1) \\
&= 0.10 \times 0.20 \times 0.30 + 0.90 \times 0.75 \times 0.80 \\
&= 0.546
\end{aligned}$$

Notar que $p(x_2 = 1, x_3 = 1) \neq p(x_2 = 1)\, p(x_3 = 1)$ ya que x_2 y x_3 solo son independientes cuando se conoce el valor de x_1 (i.e., son condicionalmente independientes).

¿Cuál es la probabilidad de responder bien las preguntas prácticas si se respondieron bien las preguntas teóricas?

Se debe calcular $p(x_3 = 1 \mid x_2 = 1)$. Notar que $p(x_3 = 1 \mid x_2 = 1) \neq p(x_3 = 1)$ ya que, como se mencionó anteriormente, x_2 y x_3 solo son independientes cuando se conoce el valor de x_1. La cantidad buscada se puede obtener por definición de probabilidad condicional:

$$
\begin{aligned}
p(x_3 = 1 \,|\, x_2 = 1) &= \frac{p(x_2 = 1, x_3 = 1)}{p(x_2 = 1)} \\
&= \frac{0.546}{0.695} \\
&\approx 0.79
\end{aligned}
$$

Donde se usaron los resultados de las preguntas anteriores.

Los desarrollos anteriores muestran que siempre se puede seguir el mismo procedimiento de marginalización para obtener alguna probabilidad deseada. Sin embargo, en muchos casos estos cálculos de pueden hacer de manera más eficiente utilizando la regla de Bayes cuando se busca conocer una distribución posterior.

Por otra parte, desde la perspectiva del grafo asociado al modelo probabilístico, sus nodos son $V = \{x_1, x_2, x_3, x_4\}$, mientras que sus arcos son $E = \{(x_1, x_2), (x_1, x_3), (x_2, x_4), (x_3, x_4)\} \subset V \times V$. Además, x_1 no tiene nodos padres, x_2 y x_3 tienen a x_1 como único nodo padre, y x_4 tiene a x_2 y a x_3 como nodos padres.

Notar que la estructura del grafo anterior puede ser usada para representar otros escenarios con una dinámica similar: una primera variable influye en el valor de otras dos variables condicionalmente independientes, las cuales a su vez influyen en una cuarta variable. Por otro lado, en este ejemplo se conocen las distribuciones de cada nodo (cada variable aleatoria sigue una distribución Bernoulli, cuyos parámetros se pueden deducir de las tablas), lo que permitió responder preguntas (i.e., hacer inferencia bayesiana) acerca del modelo. En los modelos generativos modernos no se conoce a priori la distribución de los datos (e.g., texto o imágenes), por lo que esta suele ser aprendida por una red neuronal utilizando muestras de entrenamiento generadas desde la distribución desconocida. Una vez se tiene el modelo probabilístico entrenado, se pueden generar nuevas muestras a partir de él, las cuales serán similares a las muestras usadas durante el entrenamiento ya que el modelo habrá aprendido a replicar la distribución original de los datos, la cual, en un comienzo, era desconocida. De esta forma, es posible utilizar redes neuronales para generar texto, imágenes u otro tipo de dato.

2.1.2 Formulación de una red bayesiana

Para dar una definición más precisa de lo que es una red bayesiana, es importante recordar que toda la información acerca de cómo se relacionan las distintas variables aleatorias dentro de un modelo probabilístico está contenida en su distribución conjunta $p(x_1, ..., x_N)$, la cual es más informativa que el conjunto de marginales $p(x_1), ..., p(x_N)$ ya que las marginales pierden las relaciones entre las variables. Sin embargo, muchas veces los modelos suelen incluir hipótesis de independencia entre ellas para regularizar el modelo haciéndolo más

simple, lo que en particular ayuda a evitar el overfitting (un modelo más débil no puede gastar capacidad en memorizar muestras de entrenamiento). Por otro lado, las hipótesis de independencia permiten disminuir la cantidad de parámetros necesarios para definir la distribución conjunta $p(x_1, ..., x_N)$. A modo de ejemplo, si las N variables $x_1, ..., x_N$ son binarias, la cantidad de combinaciones distintas entre ellas es 2^N, por lo que la distribución conjunta $p(x_1, ..., x_N)$ necesitaría una cantidad exponencial (en el número de variables) de valores para poder ser definida completamente. Con hipótesis de independencia, este costo exponencial puede ser disminuido significativamente, llegando incluso a un costo lineal en el caso extremo donde todas las variables se asumen independientes entre sí (como ocurre en los unigramas de texto). Para formular las hipótesis de independencia, es necesario recordar algunos conceptos elementales de teoría de grafo.

Un grafo dirigido es una estructura matemática compuesta por un conjunto de nodos V y por un conjunto de arcos E. Cada arco del grafo se representa como una flecha que va de un nodo a otro. Matemáticamente, un arco se suele denotar como un par ordenado de nodos, es decir, la afirmación $(u, v) \in E$ indica que el grafo posee un arco dirigido desde el nodo $u \in V$ hacia el nodo $v \in V$. En particular, esta notación indica que $E \subset V \times V$. Las siguientes definiciones son importantes para formalizar el concepto de red bayesiana:

- **Camino dirigido:** una secuencia de vértices $(w_1, ..., w_k) \in V^k$ se dice que es un camino dirigido desde $w_1 \in V$ hasta $w_k \in V$ si al recorrer los vértices de la secuencia (en el orden dado por la secuencia) se sigue el orden dado por los arcos, es decir, $(w_i, w_{i+1}) \in E$ para todo $i \in \{1, ..., k-1\}$.
- **Grafo dirigido acíclico (DAG):** un grafo dirigido se dice acíclico si para todo nodo $v \in V$, no es posible encontrar un camino dirigido que empiece y termine en v. Es decir, no se pueden formar caminos dirigidos cerrados (este tipo de caminos se llama ciclo).
- **Nodos padres:** los DAG inducen una jerarquía sobre el conjunto de los nodos del grafo. Para un nodo $v \in V$, sus nodos padres son los nodos que están directamente conectados a él, es decir, $\mathrm{Pa}(v) \coloneqq \{u \in V : (u, v) \in E\}$.
- **Orden topológico:** los DAG inducen un orden (parcial) sobre V a partir de la jerarquía impuesta por la orientación de los arcos. Este orden se denomina orden topológico y siempre se asumirá, sin pérdida de generalidad, que los vértices de un DAG están enumerados de acuerdo al orden topológico del grafo, es decir, $V = \{x_1, ..., x_N\}$, donde $N = |V|$ es la cantidad de nodos, y $x_j \notin \mathrm{Pa}(x_i)$ si $j > i$.

Dado un conjunto de variables aleatorias, $x_1, ..., x_N$, la idea general de una red bayesiana es poder representar la factorización de su distribución conjunta $p(x_1, ..., x_N)$ como un DAG (V, E), donde sus nodos son las variables aleatorias del modelo probabilístico (i.e., $V = \{x_1, ..., x_N\}$), mientras que la presencia de un arco $\big(x_i, x_j\big) \in E$ dirigido desde el nodo $x_i \in V$ hacia el nodo $x_j \in V$ indica que la variable x_i influye directamente en el valor de la

variable x_j (o que x_j depende directamente de x_i). Más precisamente, la noción de independencia que induce un DAG, denominada **propiedad de Markov**, es $p(x_n \mid x_1, ..., x_{n-1}) = p(x_n \mid \mathrm{Pa}(x_n))$, donde $\mathrm{Pa}(x_n) \subset \{x_1, ..., x_{n-1}\}$. Esta noción de independencia indica que $x_i \perp x_{\mathrm{pred}(x_i) \setminus \mathrm{Pa}(x_i)} \mid x_{\mathrm{Pa}(x_i)}$, es decir, todo nodo es independiente de sus ancestros (nodos hacia arriba en la jerarquía) si se conoce el valor de sus nodos padres. De esta forma, sustituyendo las propiedades de independencia en la factorización que entrega la regla de la cadena, $p(x_1, ..., x_N) = \prod_{n=1}^{N} p(x_n \mid x_1, ..., x_{n-1})$, la distribución conjunta $p(x_1, ..., x_N)$ del modelo probabilístico asociado al DAG (V, E) se factoriza como

$$p(x_1, ..., x_N) = \prod_{n=1}^{N} p(x_n \mid \mathrm{Pa}(x_n))$$

Notar que los nodos raíces del grafo (nodos x_n donde $\mathrm{Pa}(x_n) = \emptyset$) definen probabilidades incondicionales $p_\theta(x_n \mid \mathrm{Pa}(x_n)) = p_\theta(x_n)$, por lo que (las distribuciones de) estas variables se suelen llamar priors, mientras que las variables donde $\mathrm{Pa}(x_n) \neq \emptyset$ interactúan de forma condicional con sus nodos padres, permitiendo muchas veces obtener un modelo intuitivo e interpretable. Por otro lado, entendiendo $(x_i, x_j) \in E$ como que x_i causa x_j (en un sentido informal), no tendría sentido que también $(x_j, x_i) \in E$ (i.e., que x_j cause x_i). Esta observación permite entender por qué se necesita que el grafo (V, E) sea acíclico para poder ser interpretado como una red bayesiana.

En el caso de trabajar únicamente con variables discretas (e.g., categóricas), los parámetros de las distribuciones condicionales $p(x_n \mid \mathrm{Pa}(x_n))$ se pueden almacenar en tablas de forma similar a las tablas construidas en el ejercicio inicial. Sin embargo, los modelos generativos modernos requieren aprender distribuciones mucho más complejas, usualmente continuas, por lo que se suele utilizar redes neuronales que se sabe que funcionan bien en tareas que requieren capturar patrones en alta dimensión. Por lo general, cada factor $p(x_n \mid \mathrm{Pa}(x_n))$ de una red bayesiana suele seguir una distribución típica, donde sus parámetros (de la distribución, no de la red neuronal) son aprendidos por una red neuronal cuya entrada son los valores de las variables condicionantes (i.e., los valores de las variables en $\mathrm{Pa}(x_n)$), mientras que la salida son los parámetros que requiere la distribución $p(x_n \mid \mathrm{Pa}(x_n))$ para quedar totalmente definida. A modo de ejemplo:

- Si $x = (x_1, ..., x_N) \in \{0, 1\}^N$ es una imagen (monocromática), donde $N = \text{ancho} \times \text{alto}$, y x_n es el n-ésimo pixel de la imagen (cuyo valor es 0 o 1 por simplicidad), entonces cada pixel x_n puede ser considerado como una distribución Bernoulli, $x_n \sim \mathrm{Bernoulli}(r_n)$, donde $r_n \in [0, 1]$ es el parámetro de la distribución, cuyo valor depende de los pixeles en $\mathrm{Pa}(x_n)$. Este parámetro suele ser determinado por una red neuronal (con salida sigmoidal para estar en el intervalo $[0, 1]$), cuya entrada son los valores de los pixeles vecinos $\mathrm{Pa}(x_n)$ que el modelo considere. Por ejemplo, una capa convo-

lucional considera como $\text{Pa}(x_n)$ únicamente a los pixeles que están dentro del campo receptivo de la convolución.

- Si $x = (x_1, ..., x_N) \in \mathbb{R}^N$ es una variable continua (y cada componente x_n es un escalar irrestricto), es usual considerar $p(x_n \mid \text{Pa}(x_n)) \sim \mathcal{N}(\mu_n, \sigma_n^2)$, donde los parámetros $\mu_n \in \mathbb{R}$ y $\sigma_n^2 > 0$ dependen de los valores de las variables en $\text{Pa}(x_n)$. Al igual que antes, estos parámetros suelen estar determinados por redes neuronales cuyas entradas son los valores de las variables en $\text{Pa}(x_n)$.

Notar que en todos los ejemplos se ha considerado que cada x_n es una variable aleatoria unidimensional. En general, cada variable x_n por sí sola puede ser un vector aleatorio de varias dimensiones, por lo que no se diferenciará, como es usual en machine learning, entre variable aleatoria y vector aleatorio. Más aún, cada nodo $x_n \in V$ puede estar representando a un conjunto de variables aleatorias que, por sus propiedades de independencia, pueden agruparse en un mismo nodo.

Por otro lado, es importante mencionar que las redes bayesianas no se llaman así debido a que asumen la interpretación bayesiana. El nombre se debe a que utilizan la regla de Bayes para hacer inferencia sobre las variables desconocidas.

Generación de muestras

Dada una red bayesiana $p(x_1, ..., x_N) = \prod_{n=1}^{N} p(x_n \mid \text{Pa}(x_n))$, donde las distribuciones $p(x_n \mid \text{Pa}(x_n))$ son todas conocidas (y con parámetros conocidos o aprendidos con una red neuronal), entonces el algoritmo de **ancestral sampling** permite generar una nueva muestra desde la distribución conjunta $p(x_1, ..., x_N)$. Si bien esto no siempre es necesario (e.g., en el ejemplo inicial no es claro para qué sería útil generar muestras a partir de esa red bayesiana), en IA generativa el sampling es la tarea más importante luego del entrenamiento ya que permite, por ejemplo, generar nuevas respuestas si se tiene un LLM ya entrenado o generar una nueva imagen si se tiene un modelo de difusión ya entrenado.

El algoritmo de ancestral sampling comienza, naturalmente, generando muestras desde los nodos raíces (que funcionan como semillas) y luego utiliza estas muestras para generar nuevas muestras de los nodos hijos. Este procedimiento se repite jerárquicamente (i.e., con el orden de generación siguiendo el orden topológico del grafo) hasta llegar al último nodo, obteniendo así una muestra desde todos los nodos del grafo. La muestra conjunta resultante de este procedimiento, $(x_1, ..., x_N)$, es una muestra generada desde la distribución conjunta $p(x_1, ..., x_n)$.

Es importante destacar que el algoritmo de ancestral sampling asume que se sabe cómo generar nuevas muestras desde cada factor $p(x_n \mid \text{Pa}(x_n))$ de la red bayesiana. Este es otro motivo por el cual los modelos gráficos suelen elegir distribuciones condicionales simples (e.g., categóricas o gaussianas) en sus formulaciones.

En los paradigmas generativos que se revisarán, el sampling será realizado casi siempre usando ancestral sampling: en modelos de variable latente como las GANs o los VAEs, se comienza generando una muestra inicial desde $z \sim p(z)$ y luego se utiliza esta muestra para generar desde $p(x \mid z)$ usando la red neuronal entrenada (llamada generador en el caso de la GAN y decoder en el caso del VAE). En modelos secuenciales como los ARMs o los DMs, se comienza con una muestra inicial desde el nodo raíz (token inicial en ARMs y ruido inicial en DMs) y luego se decodifica iterativamente de forma secuencial (de forma causal en los ARMs y de forma anticausal en los DMs). En los modelos basados en score y EBMs en general (que no son redes bayesianas), dado que solo se suele conocer una cantidad proporcional a $p(x)$, la generación no es realizada con ancestral sampling, si no que se usan técnicas de Markov chain Monte Carlo (MCMC) ya que este tipo de algoritmos no necesita conocer la constante de normalización.

2.1.3 Algunas redes bayesianas usuales

La generalidad de las redes bayesianas permite ver varios modelos clásicos de machine learning, que usualmente se enseñan de manera independiente, como casos particulares de redes bayesianas.

Mixturas

Un modelo de mezclas (mixtura) consiste en combinar N distribuciones independientes, $p_1(x), ..., p_N(x)$, usando una variable latente $p(z) \sim \text{Categorical}(\{1, ..., N\})$ que elige al azar una de las distribuciones para generar la muestra observable x. Más precisamente, la distribución marginal asociada a la variable observable $p(x)$, es una combinación convexa de las distribuciones que se buscan mezclar:

$$p(x) = \sum_{n=1}^{N} p(z = n)\, p_n(x)$$

Notar que esta distribución marginal se obtiene al marginalizar x en el modelo de variable latente estándar, $p(x, z) = p(z)\, p(x \mid z)$, donde $p(z)$ es una distribución categórica y $p(x \mid z = n) = p_n(x)$. En consecuencia, el DAG asociado a una mixtura es el siguiente:

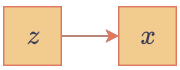

Figura 24: DAG asociado a una mixtura.

Las mixturas más usuales, llamadas **mixturas gaussianas**, son las que consideran $p_n(x) \sim \mathcal{N}(\mu_n, \Sigma_n)$. Por otro lado, es útil notar que los modelos de variable latente más generales,

donde z es una variable latente continua, pueden ser vistos como una mixtura infinita de distribuciones.

Naïve Bayes

Naïve Bayes es un clasificador clásico que, a diferencia de otros clasificadores que aprenden directamente una distribución $p(y \mid x)$ (como la regresión logística o los clasificadores usuales basados en redes neuronales), utiliza un enfoque generativo (i.e., aprende una distribución conjunta $p(x, y)$ en vez de una distribución discriminativa $p(y \mid x)$). Este clasificador factoriza la distribución conjunta de una muestra $x \in \mathbb{R}^D$ y su respectiva clase, $y \in \{1, ..., C\}$, como

$$p(x, y) = p(y) \prod_{d=1}^{D} p(x_d \mid y)$$

Es decir, esta red bayesiana asume que $x_i \perp x_j \mid y$: las características (coordenadas) de una muestra $x \in \mathbb{R}^D$ son condicionalmente independientes si es que se conoce la etiqueta de clase $y \in \{1, ..., C\}$ (de aquí viene la parte ingenua o naïve). El DAG asociado a esta red bayesiana es el siguiente:

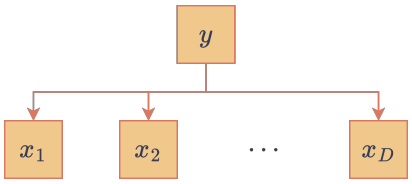

Figura 25: DAG asociado al clasificador Naïve Bayes.

Notar que en este DAG se separó la variable $x \in \mathbb{R}^D$ en sus componentes para poder separar las relaciones de independencia de cada coordenada. En otros grafos es usual agrupar todo el vector aleatorio $x \in \mathbb{R}^D$ en un mismo nodo cuando este puede ser visto como un mismo objeto en el sentido de las relaciones de dependencia e independencia.

En esta red bayesiana $p(y)$ es una distribución categórica (ya que y es una etiqueta de clase), mientras que cada distribución $p(x_d \mid y)$ dependerá de la naturaleza de la variable aleatoria x_d. Para realizar clasificación, este modelo calcula la posterior $p(y \mid x)$ mediante la regla de Bayes y elige la clase más probable.

Cadenas de Markov

Cuando las variables del modelo, $(x_1, ..., x_T)$, tienen un comportamiento o interpretación secuencial, es usual considerar ciertas hipótesis de causalidad o independencia temporal. Una familia de modelos simples pero muy útiles en la práctica son las **cadenas de Markov**

(de primer orden), las cuales asumen que el futuro es independiente del pasado si se conoce el presente, es decir: $p(x_{t+1} \mid x_1, ..., x_t) = p(x_{t+1} \mid x_t)$. En consecuencia, este tipo de modelos factoriza la distribución conjunta como

$$p(x) = p(x_1) \prod_{t=1}^{T-1} p(x_{t+1} \mid x_t)$$

Dada la forma en la que se van generando las muestras según el algoritmo de ancestral sampling, a este tipo de modelos también se les dice **autorregresivos de primer orden**. Una elección usual, y que muchas veces es asumida sin siquiera mencionarlo, es considerar que la cadena de Markov es **homogénea**, es decir, $p(x_{t+1} \mid x_t)$ es independiente de t para todo $t \in \{1, ..., T-1\}$ (lo que puede verse como una forma de parameter tying). En este caso, si además cada variable $(x_1, ..., x_T)$ es discreta, el kernel de transición $p(x_{t+1} \mid x_t)$ puede ser guardado en una matriz estocástica, llamada **matriz de transición**: $P_{ij} = p(x_2 = j \mid x_1 = i) = p(x_{t+1} = j \mid x_t = i)$, para todo $t \in \{1, ..., T-1\}$. El DAG asociado a una cadena de Markov es el siguiente:

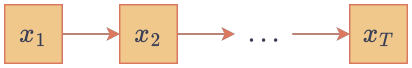

Figura 26: DAG asociado a una cadena de Markov.

La condición de Markov se puede generalizar a un orden $k \in \mathbb{N}$, donde cada variable depende de las k variables anteriores, lo cual se representa con la hipótesis de independencia $p(x_{t+1} \mid x_1, ..., x_t) = p(x_{t+1} \mid x_{t-k+1}, ..., x_t)$. En particular, si se están modelando secuencias de texto, cuando $k = 1$ al modelo se le suele llamar **bigrama** y para $k = 2$ se le llama **trigrama**. En general, se les llama $(k+1)$-grama a los modelos de Markov de orden k que modelan secuencias de texto utilizando $k+1$ gramas (unidad básica de texto, como una letra o una palabra). En el caso extremo $k = 0$, donde todos los tokens son independientes (pensar, por ahora, un token como una palabra dentro de un texto), el modelo se llama **unigrama** o **bag-of-words** (BoW). El otro caso extremo, $k = T$, corresponde a no realizar ninguna hipótesis de independencia, recuperando la regla de la cadena.

Por otro lado, los procesos forward y backward asociados a los modelos de difusión son cadenas de Markov, al igual que los flujos normalizantes. En general, la forma relativamente simple que tienen las cadenas de Markov para modelar procesos secuenciales permite poder usarlas en una amplia cantidad de casos, por lo que este tipo de procesos suelen ser estudiados de forma especializada.

Modelos de Markov ocultos

Los modelos de Markov ocultos (HMM) son, al igual que las cadenas de Markov, redes bayesianas útiles para modelar secuencias de variables $x = (x_1, ..., x_T)$. La principal diferencia es que los HMM asumen la existencia de una cadena de Markov latente, $z = (z_1, ..., z_T)$, la cual va generando las observaciones visibles $x = (x_1, ..., x_T)$, donde cada elemento x_t generado depende únicamente de la variable latente z_t. Es decir:

$$p(z, x) = p(z_1) \prod_{t=1}^{T-1} p(z_{t+1} \mid z_t) \prod_{t=1}^{T} p(x_t \mid z_t)$$

En esta factorización, $p(z_1)$ es la distribución inicial sobre los estados latentes, $p(z_t \mid z_{t-1})$ es el kernel de transición entre los estados latentes y $p(x_t \mid z_t)$ es la **distribución de emisión**, la cual modela cómo cada estado latente genera su respectiva observación. El DAG asociado a un HMM es el siguiente:

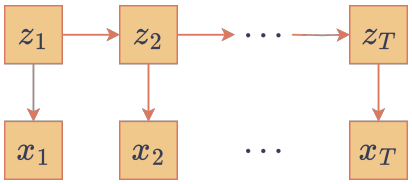

Figura 27: DAG asociado a un modelo de Markov oculto.

Si bien los HMMs se pueden usar en diversos campos, en NLP es usual ver este tipo de modelos en tareas de etiquetado de secuencias de texto como POS tagging (busca etiquetar cada palabra con su categoría gramatical como sustantivo o verbo) y NER (busca identificar palabras que son entidades como personas o lugares), donde es necesario asignarle una clase a cada uno de los tokens del texto a analizar. Por otro lado, si bien este tipo de modelos podría usarse para generación de texto, la independencia condicional entre los tokens $x_1, ..., x_T$ no permitiría generar frases con mucho sentido.

2.2 Modelos generativos modernos

2.2.1 Formulación naïve

Muchos de los modelos generativos actuales están basados en redes neuronales entrenadas para aprender a aproximar una distribución desconocida $p_{\text{data}}(x)$ utilizando únicamente un conjunto de entrenamiento, $\mathcal{D} = \{x^1, ..., x^K\}$, donde (se asume que) cada muestra $x^k \in \mathbb{R}^D$ fue generada de manera independiente a partir de $p_{\text{data}}(x)$. Una forma directa de hacer esto es considerar la **distribución empírica**, la cual le asigna una masa uniforme únicamente a

las muestras observadas en el entrenamiento. Es decir, la función de masa de la distribución empírica es

$$p_{\mathcal{D}}(x) = \frac{1}{K}\sum_{k=1}^{K} \delta_{x^k}(x),$$

donde $\delta_{x^k}(x) \coloneqq \begin{cases} 1 \text{ si } x=x^k \\ 0 \text{ si no} \end{cases}$ es la **medida de Dirac** centrada en la muestra $x^k \in \mathbb{R}^D$.

Sin embargo, esta distribución es poco útil para ser usada en un proceso generativo ya que solo asigna masa a las muestras observadas en el conjunto de entrenamiento $\mathcal{D}$, lo que no permite variabilidad en la generación más allá de este conjunto. Una posible regularización a este enfoque consiste en utilizar **kernel density estimation** (KDE), lo cual puede verse como una versión suavizada de la distribución empírica. Si $\mathcal{K} : \mathbb{R} \to \mathbb{R}_+$ es un **kernel de densidad** (i.e., una función de densidad con $\mathbb{E}_{x\sim\mathcal{K}}[x] = 0$), entonces la distribución que entrega KDE es un promedio de estos kernels centrados en cada una de las muestras en $\mathcal{D}$:

$$\text{KDE}_{\mathcal{D}}(x) = \frac{1}{K}\sum_{k=1}^{K} \mathcal{K}\big(x - x^k\big)$$

La función $x \mapsto \mathcal{K}\big(x - x^k\big)$ suele interpretarse como una versión suave de la medida de Dirac, $x \mapsto \delta_{x^k}(x)$, debido a que $\mathcal{K}\big(x - x^k\big)$ suele entregar masa a los vecinos de x^k (generalmente decayendo a medida que se alejan de x^k), mientras que $\delta_{x^k}(x)$ concentra toda su masa en x^k.

En las siguientes figuras se pueden ver ejemplos de KDE para $D = 1$ (izquierda) y $D = 2$ (derecha). En ambos casos, $\mathcal{D}$ está formado por 5 puntos generados al azar sobre el intervalo $[0, 1]$ (izquierda) y sobre el cuadrado $[0, 1] \times [0, 1]$ (derecha). El kernel usado en ambos casos es un kernel gaussiano, $\mathcal{K}(x) = \frac{1}{\sqrt{2\pi h^2}} \exp\Big(-\frac{\|x\|^2}{2h^2}\Big)$ (con $h > 0$ un hiperparámetro), el cual permite colocar una campana gaussiana en cada uno de los puntos de $\mathcal{D}$, de forma similar a como ocurre en una mixtura gaussiana con prior uniforme. Dado que el kernel depende únicamente de la cantidad $r = \|x\|$, este kernel también se suele llamar **kernel de base radial** (RBF kernel) ya que, como se observa en el mapa de calor, genera curvas de nivel circulares.

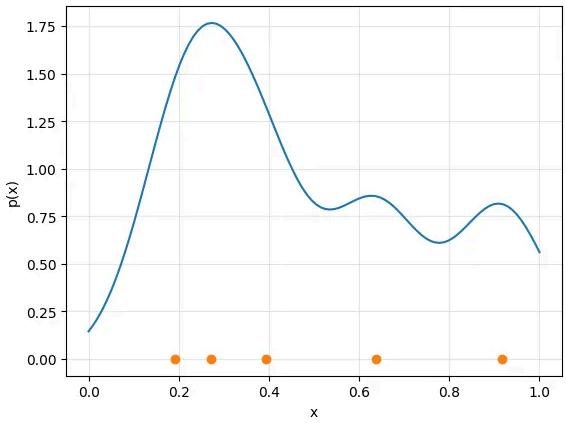

Figura 28: KDE en $D = 1$.

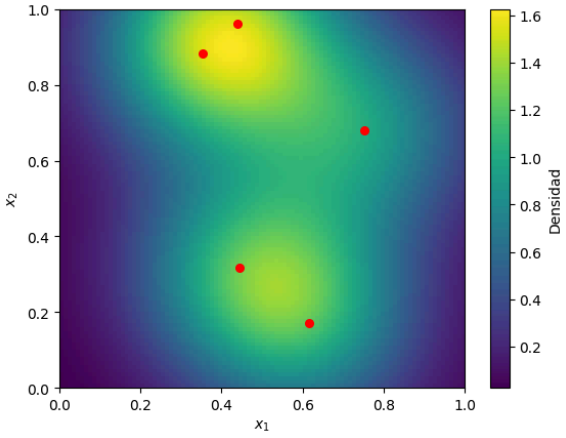

Figura 29: KDE en $D = 2$.

En ambos casos, el hiperparámetro $h > 0$ (asociado a la desviación estándar en una distribución gaussiana) determina a qué velocidad decae la densidad en $\mathbb{R}^D$ a medida que las posibles muestras se alejan de las muestras observadas en $\mathcal{D}$: un valor h pequeño puede capturar con precisión variaciones locales, pero puede producir overfitting al poder capturar también el ruido en las muestras, mientras que un h más grande, si bien genera una densidad más suave, pero puede producir underfitting al limitarse a distribuciones de baja frecuencia.

2.2.2 Paradigmas generativos actuales

Si bien KDE puede funcionar bien en baja dimensión, sufre de la maldición de la dimensionalidad, por lo que no se puede utilizar en contextos como generación de texto o imágenes. Más aún, no es claro cómo utilizar este enfoque para tareas más complejas

como edición de imágenes o chatbots. En consecuencia, hay que buscar enfoques más robustos, como aquellos basados en verosimilitud o que se haya probado que, al menos en la práctica, funcionan bien. Dentro de esta familia de modelos, se encuentran los 6 paradigmas mencionados en la introducción. A excepción de los modelos basados en energía, todos estos paradigmas pueden verse como un tipo específico de red bayesiana:

Modelos autorregresivos (ARMs)

Este tipo de modelos se suele utilizar para modelar la generación de secuencias temporales. Si $x = (x_1, ..., x_T)$ es una secuencia de variables (se asumirá T fijo por simplicidad, pensado como un largo de secuencia máximo), la red bayesiana asociada a un ARM corresponde a factorizar la distribución conjunta siguiendo la regla de la cadena y no asumir, en principio, ninguna independencia (aunque impone como orden topológico del grafo al orden temporal de las variables):

$$p(x) = p(x_1) \prod_{t=1}^{T-1} p(x_{t+1} \mid x_1, ..., x_t)$$

El DAG que representa esta red bayesiana es el siguiente:

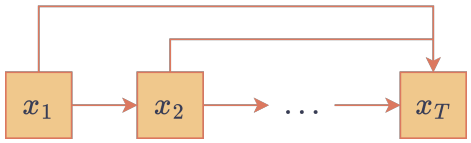

Figura 30: DAG asociado a un modelo autorregresivo.

Hasta hace no mucho tiempo, las distribuciones condicionales $p(x_{t+1} \mid x_1, ..., x_t)$ solían ser aprendidas usando redes neuronales recurrentes como la LSTM o la GRU. Sin embargo, hoy en día es más usual el uso de arquitecturas tipo Transformer debido a sus ventajas de paralelización y memoria temporal. Por otro lado, el hecho de no tener variables ocultas permite entrenar estos modelos usando el criterio de máxima verosimilitud, lo cual deja de ser posible en los siguientes paradigmas (al menos de forma exacta).

Autoencoders variacionales (VAEs)

Esta familia de modelos formula el proceso generativo como un modelo de variable latente, $p(x, z) = p(z)\, p(x \mid z)$, con la particularidad de que al mismo tiempo también aprenden otro modelo $q(z \mid x)$ que es entrenado para estimar la distribución posterior $p(z \mid x)$. Si bien estos modelos no pueden ser optimizados usando el criterio de máxima verosimilitud (debido a que se vuelve intratable en modelos complejos), el uso de estos dos modelos permite

encontrar una función de costo robusta, llamada ELBO, la cual sí se puede calcular de manera eficiente para el entrenamiento. Los DAGs de un VAE son los siguientes:

Figura 31: DAGs asociados a un VAE.

Los parámetros de la distribución condicional $p(x \mid z)$ (que genera una muestra x a partir de un valor de la variable latente z) suelen ser aprendidos por una red neuronal, llamada **decoder**, la cual recibe como entrada el valor de la variable latente $z \in \mathbb{R}^L$. Por otro lado, los parámetros de la distribución condicional $q(z \mid x)$ (que recupera la variable latente z a partir de una muestra observada x) suelen ser aprendidos por otra red neuronal diferente, llamada **encoder**, la cual recibe como entrada una muestra $x \in \mathbb{R}^D$. Considerando que usualmente se utiliza $L \ll D$, este tipo de modelos tiene forma de autoencoder, de donde viene su nombre. La parte variacional tiene que ver con su función de costo, la cual resulta ser una cota inferior de la log-verosimilitud que se obtiene usando inferencia variacional.

Redes generativas adversarias (GANs)

Las GANs también son modelos de variable latente de la forma $p(x, z) = p(z)\, p(x \mid z)$ solo que, a diferencia de los VAEs que además incluyen un encoder, este tipo de modelos incluye un clasificador externo, $q(y \mid x)$, que es desechado después del entrenamiento. La idea principal de una GAN es que el clasificador $q(y \mid x)$ aprenda a reconocer si una muestra dada, $x \in \mathbb{R}^D$, es una muestra artificial ($y = 0$) generada desde $p(x \mid z)$ o si es una muestra real ($y = 1$) generada desde la distribución desconocida $p_{\text{data}}(x)$. De este modo, mientras el clasificador es entrenado para esta tarea, el generador es entrenado para engañar al clasificador (i.e., busca que identifique las muestras generadas desde $p(x \mid z)$ como reales), lo que genera como consecuencia que el generador aprenda a imitar muy bien las muestras que se generan desde $p_{\text{data}}(x)$. Los DAGs asociados a una GAN son muy similares a los de un VAE:

Figura 32: DAGs asociados a una GAN.

En este caso, al modelo $p(x \mid z)$ se le llama **generador** (no se usa la interpretación encoder--decoder), mientras que al clasificador $q(y \mid x)$ se le suele llamar **discriminador**.

Modelos de difusión (DMs)

Este tipo de modelos está compuesto por dos distribuciones distintas. Una primera distribución (fija) inyecta ruido de manera progresiva a muestras de $p_{\text{data}}(x)$ hasta llegar a una

imagen de puro ruido. Dado que el ruido se va inyectando directamente sobre la imagen ruidosa anterior, este proceso de destrucción de información puede ser expresado como una cadena de Markov:

$$q(x, z_1, ..., z_T) = p_{\text{data}}(x) \prod_{t=1}^{T} q(z_t \mid z_{t-1}),$$

donde $q(z_t \mid z_{t-1})$ (con $z_0 = x$) son transiciones ruidosas tales que $p(z_T) \approx \mathcal{N}(0, \mathrm{I}_D)$. Al mismo tiempo, otra red bayesiana es entrenada para aprender a deshacer el proceso de inyección de ruido. Esta red bayesiana, cuyos parámetros son aprendidos por una red neuronal, también es una cadena de Markov pero hacia atrás en el tiempo (ya que busca revertir la cadena de Markov que inyecta el ruido):

$$p(x, z_1, ..., z_T) = p(z_T) \prod_{t=1}^{T} p(z_{t-1} \mid z_t),$$

donde $p(z_T) \sim \mathcal{N}(0, \mathrm{I}_D)$ es el nodo raíz de esta red bayesiana. Los DAGs asociados a ambos modelos gráficos son los siguientes:

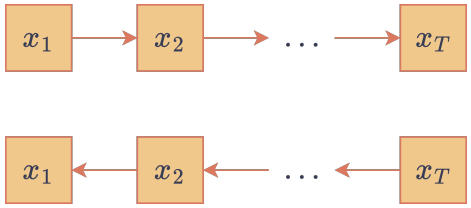

Figura 33: DAGs asociados a un modelo de difusión.

Una vez se tiene entrenada la red neuronal para $p(x, z_1, ..., z_T)$, se puede generar una nueva muestra comenzando con una muestra generada desde $p(z_T) \sim \mathcal{N}(0, \mathrm{I}_D)$ y aplicando el proceso de denoising aprendido para llegar a una muestra $x \in \mathbb{R}^D$, la cual debería parecerse a una muestra de $p_{\text{data}}(x)$ si el modelo de difusión fue bien entrenado.

Flujos normalizantes (NFs)

El paradigma de los flujos normalizantes modela una distribución de probabilidad $p(x \mid z)$ mediante la transformación de una variable aleatoria simple y bien conocida z a través de una serie de funciones invertibles y diferenciables. La red bayesiana asociada a estos modelos puede verse, al igual que los modelos anteriores, como un modelo de variable latente $p(x, z) = q(z)\, p(x \mid z)$, donde la distribución $p(x \mid z)$ es determinista, es decir, $x = f(z)$, cuando se conoce el valor de $z \sim q(z)$. Dado que la transformación se considera

invertible y de inversa diferenciable (i.e., f es un difeomorfismo), el teorema de cambio de variable permite obtener la densidad $p(x)$ a partir de la densidad $q(z)$:

$$p(x) = q(f^{-1}(x)) \left|\det(\mathrm{D}_x f^{-1}(x))\right|$$

Por lo general, la transformación $f : \mathbb{R}^D \to \mathbb{R}^D$ se construye componiendo varias transformaciones más simples, $f(z) = (f_N \circ \cdots \circ f_1)(z)$, por lo que el jacobiano $\mathrm{D}_x f^{-1}(x) \in \mathcal{M}_{D,D}(\mathbb{R})$ se puede descomponer, de acuerdo a la regla de la cadena (para derivadas), como el producto de varios jacobianos individuales.

El DAG asociado a un modelo basado en flujos normalizantes se puede ver, dependiendo de la dirección temporal que se elija, de dos formas distintas:

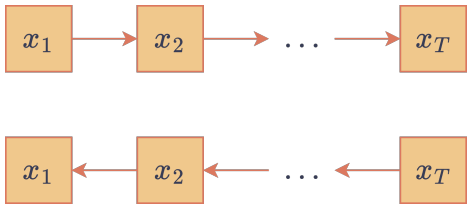

Figura 34: DAGs asociados a un modelo de flujos normalizantes.

Por último, es importante notar que cada paradigma utiliza un enfoque distinto para resolver el problema de aprender una distribución desconocida, cada uno aprovechando suposiciones e hipótesis de independencia distintas, lo cual le otorga a cada enfoque propiedades únicas, tanto en su forma de entrenar, como en el tipo de generación que realiza. Por otro lado, es interesante mencionar que todos los modelos tienen, de un modo u otro, alguna conexión con la física termodinámica: los ARMs usan la idea de temperatura para controlar la diversidad en la generación; los VAEs y los DMs entrenan una cantidad llamada ELBO, la cual puede relacionarse con la energía libre de Helmholtz; y los EBMs se basan en escribir la función de densidad como una distribución de Boltzmann y modelan directamente una función de energía.

2.2.3 Modelos generativos condicionales

Hasta el momento, todos los modelos generativos han sido formulados para aprender una distribución $p(x)$ que aproxime bien a otra distribución desconocida, $p_{\text{data}}(x)$. Sin embargo, en muchos casos prácticos se dispone de información adicional que se quiere que influya en la generación de los datos. Esto da lugar a los **modelos generativos condicionales**, donde en lugar de modelar una distribución incondicional $p(x)$, se modela una distribución condicional $p(x \,|\, y)$, con y representando una variable observada que actúa como entrada adicional al modelo y que busca modificar el comportamiento de la distribución. Este tipo

de condicionamiento es el que permite generar imágenes dada una descripción o conversar con un chatbot (en vez de generar texto libre). En estos casos, a la condición y se le suele llamar **prompt**.

Los modelos condicionales, junto a la capacidad de multimodalidad (esto es, procesar varias modalidades: texto, imágenes, sonido, videos, etc.), permiten resolver algunas tareas que parecen difíciles (o hasta imposibles) de resolver con otros enfoques más clásicos, por ejemplo:

- **Traducción de texto:** la arquitectura Transformer [14] (y otras arquitecturas seq2seq [2]) son diseñadas para aprender un modelo condicional $p(x \mid y)$ que es capaz de traducir un texto de entrada y a otro idioma (texto de salida x). En la arquitectura Transformer el texto es codificado mediante un mecanismo de auto-atención, mientras que la información condicional es inyectada al modelo mediante un mecanismo de atención cruzada.
- **Generación de imágenes a partir de texto:** modelos como DALL-E [35] y Stable Diffusion [3] permiten generar imágenes a partir de descripciones textuales. En este caso, los modelos aprenden una distribución condicional $p(x \mid y)$ donde x es la imagen generada e y es el texto dado como entrada (prompt). En el caso de DALL-E, $p(x \mid y)$ es modelado por un modelo autorregresivo, mientras que en Stable Diffusion, $p(x \mid y)$ es modelado por un modelo de difusión. El éxito de los modelos de difusión llevó a OpenAI a cambiar a este paradigma para diseñar DALL-E 2 [36] y DALL-E 3 [37].
- **Traducción y modificación de imágenes:** modelos como pix2pix [38] y CycleGAN [4] (ambos modelos tipo GAN) o FLUX.1 Kontext [21] (modelo de flow matching) permiten realizar tareas sobre imágenes como colorización de imágenes en escala de grises e inpainting (imputación de píxeles, por ejemplo, para corregir detalles estéticos). En todos estos casos, x es la imagen generada e y es la imagen de entrada. De forma similar se pueden modelar otras tareas como super-resolución.
- **Automatic speech recognition (ASR):** modelos como Whisper [39] (modelo tipo Transformer que procesa secuencias de audio en vez de secuencias de texto) permiten transcribir grabaciones de voz modelando una distribución condicional $p(x \mid y)$, donde y es una pista de audio y x es una secuencia de texto. La tarea inversa, la cual consiste en modelar $p(y \mid x)$, permitiría resolver la tarea de generar un audio de voz a partir del texto que se debe hablar en el audio.

En este tipo de modelos, la generación de nuevas muestras depende explícitamente del valor del factor condicionante y, el cual es necesario para controlar y dirigir el proceso de generación dependiendo de qué valor se entregue como entrada adicional al modelo. Por otro lado, es importante notar que cualquier red bayesiana incondicional, $p(x_1, ..., x_N) = \prod_{n=1}^{N} p(x_n \mid \mathrm{Pa}(x_n))$, puede ser transformada en una red bayesiana condicional, agregando y a cada uno de los factores:

$$p(x_1, ..., x_N \mid y) = \prod_{n=1}^{N} p(x_n \mid \mathrm{Pa}(x_n), y)$$

Más aún, dado que cada factor $p(x_n \mid \mathrm{Pa}(x_n))$ suele ser aprendido por una red neuronal (cuya entrada son los valores de las variables aleatorias $\mathrm{Pa}(x_n)$), la misma arquitectura neuronal usada para aprender (los parámetros de) $p(x_n \mid \mathrm{Pa}(x_n))$ suele ser adaptada para aprender (los parámetros de) la distribución condicional $p(x_n \mid \mathrm{Pa}(x_n), y)$. Para esto, basta modificar la red neuronal para poder inyectar, de algún modo, el valor de y en el input. Una forma fácil y que muchas veces funciona es concatenar el valor de y al resto de entradas de la red neuronal, aunque también es posible utilizar mecanismos más complejos. Por ejemplo, Stable Diffusion inyecta el prompt al proceso de generación utilizando un mecanismo de atención cruzada, mientras que la arquitectura DiT [40] (Diffusion Transformer) inyecta la condición mediante moduladores tipo FiLM [41].

Esta facilidad para transformar una arquitectura incondicional en una arquitectura condicional ha permitido un rápido desarrollo de los modelos generativos multimodales (MLLMs). Encontrando formas eficientes de representar imágenes como vectores (o tensores de algún rango mayor), un modelo de lenguaje como GPT 4 puede ser transformado a un modelo que permita recibir adicionalmente imágenes como entrada, al igual que, por ejemplo, el modelo GPT 4o. De forma análoga se puede condicionar cualquier modelo generativo basado en redes neuronales con respecto a otras modalidades como video, temperatura, movimiento, etc., siempre y cuando se conozca una forma eficiente de inyectar la información condicional en la red neuronal. A la representación vectorial de una entrada en alguna modalidad específica se le suele llamar **vector de embedding**, mientras que al modelo que transforma dicha modalidad en una representación vectorial se le suele llamar **modelo de embedding** o **capa de embedding** en el caso de estar dentro de un modelo más grande.

Para mayor simplicidad, por lo general se considerarán modelos generativos incondicionales al momento de formular los distintos paradigmas, pero siempre hay que tener en cuenta que el enfoque utilizado actualmente (basado en redes neuronales) permite añadir condiciones adicionales al modelo gráfico como entradas adicionales a las redes neuronales sin mayor complejidad. Más aún, por lo general uno siempre desea tener un modelo generativo condicional ya que no suele ser suficiente tener un modelo que genere objetos (e.g., imágenes o texto) de forma libre, sino que se busca poder guiar el proceso de generación (e.g., mediante un prompt) para obtener resultados útiles para el usuario final.

Por último, es importante notar que para construir un modelo generativo condicional es necesario aprender una distribución $p(x \mid y)$ (que se puede considerar como el opuesto bayesiano de un clasificador), por lo que es necesario, al igual que en el caso supervisado, tener pares condición-muestra para el entrenamiento (aunque no siempre; algunos modelos

condicionales siguen un enfoque autosupervisado). Sin embargo, la obtención de estos datos etiquetados muchas veces es más fácil que en un clasificador clásico. Por ejemplo, para obtener un dataset de imágenes con texto descriptivo, se pueden obtener los captions a partir del texto alternativo de imágenes sacadas de internet.

Relación con los modelos discriminativos

Por lo general, un curso clásico de machine learning o deep learning se enfoca principalmente en estudiar tópicos de aprendizaje supervisado donde, por ejemplo, una red neuronal aprende a discriminar a partir de múltiples ejemplos etiquetados. Es decir, el modelo aprende una distribución condicional $p(y \mid x)$ que aprende a reconocer el grupo al que pertenece una muestra. Por esto, a este tipo de modelos se les llama **modelos discriminativos**.

Dado un modelo generativo $p(x, y)$, siempre es posible, al menos en teoría, obtener un modelo discriminativo considerando que $p(y \mid x) = \frac{p(x,y)}{\int p(x,y)\,\mathrm{d}y}$. Sin embargo, si la tarea objetivo es de tipo discriminativa (e.g., un clasificador), por lo general se obtiene un mejor desempeño entrenando directamente un modelo $p(y \mid x)$ en vez de adaptar un modelo generativo $p(x, y)$ a su versión discriminativa. Esto es esperable ya que ambos enfoques siguen paradigmas de entrenamiento distintos, donde los modelos discriminativos son entrenados precisamente para rendir bien en la tarea de clasificación. Por otro lado, un modelo discriminativo $p(y \mid x)$ solo se enfoca en aprender a clasificar objetos en categorías predefinidas, sin modelar la estructura subyacente de los datos mediante el aprendizaje de la distribución $p(x)$. En cambio, un modelo discriminativo obtenido a partir de un modelo generativo aprende ambas distribuciones al aprender la distribución conjunta $p(x, y)$. En consecuencia, los modelos generativos suelen tener un mejor entendimiento del mundo que los modelos discriminativos, ya que un modelo discriminativo no puede, por ejemplo, generar nuevas muestras a partir de una clase determinada. Además, los modelos discriminativos pueden ser vulnerables a ataques adversarios, donde se ha demostrado que pequeñas modificaciones en la entrada pueden cambiar totalmente la predicción. En cambio, un modelo discriminativo obtenido a partir de uno generativo podría ser más robusto a estas perturbaciones. Más aún, en problemas de aprendizaje semisupervisado, donde hay pocos ejemplos etiquetados y muchos sin etiquetar, los modelos generativos pueden ser útiles para mejorar el clasificador final.

Otra observación importante es que hasta hace no muchos años, los principales avances en deep learning (e.g., AlexNet, ResNet y EfficientNet) correspondían a modelos usados para aprendizaje supervisado, ya que se daba por hecho que las tareas generativas estaban limitadas solo a los humanos. Sin embargo, en los últimos diez años, el mayor progreso ha estado dominado por los modelos generativos, los cuales son, usualmente, de naturaleza no supervisada. Parte de este rápido desarrollo es gracias a las técnicas desarrolladas

en modelos discriminativos, las cuales han sido adaptadas a modelos generativos. Dentro de estas técnicas se encuentran muchas de las mejoras en arquitecturas neuronales (e.g., ReLU, bloques residuales, dropout, BatchNorm), pero también se encuentran mejoras en los frameworks y en el hardware usado.

2.2.4 Modelos de variable latente

En el ejemplo inicial, todas las variables del modelo pueden ser observadas (es decir, además de conocer sus distribuciones, se puede conocer el valor que toman), por lo que se dice que el modelo es completamente observable. Sin embargo, en modelos probabilísticos más complejos, es común que algunas variables del sistema no sean directamente observables, sino que sean **variables latentes** (también llamadas **variables ocultas**), las cuales se suelen considerar como variables ficticias que influyen sobre las variables observadas, permitiendo, además, explicar la variabilidad de los datos observados. A modo de ejemplo, las imágenes del dataset CelebA [42] (dataset de imágenes de caras) se representan como vectores de dimensión $D = 178 \times 218 \times 3 = 116\,412$. Sin embargo, uno puede hipotetizar que las imágenes fueron generadas transformando una variable aleatoria oculta (que vive en un espacio de dimensión $L < D$) al espacio ambiente $\mathbb{R}^D$. Este espacio latente podría ser un conjunto de variables más primitivas como el sexo de la persona en la fotografía, su color de pelo, el color de sus ojos, su color de piel, si usa lentes o no, si usa sombrero, etc., y las distintas imágenes que se puedan obtener una vez se definen estas variables latentes vienen dadas por la aleatoriedad del proceso de generación.

El uso de modelos de variables ocultas se suele justificar mediante la **manifold hypothesis**, la cual propone que los datos observados viven en un espacio ambiente de alta dimensión, $\mathbb{R}^D$, pero que realmente provienen de un espacio (más precisamente, una variedad diferenciable) de menor dimensión, $\mathbb{R}^L$. La hipótesis de la variedad es una hipótesis bastante aceptada (con evidencia empírica) y permite, entre otras cosas, explicar la aparente ausencia de la maldición de la dimensionalidad (a.k.a. efecto Hughes) al entrenar redes neuronales. Por otro lado, el uso de los modelos de variable latente permite construir formas ingeniosas de formular distintos enfoques generativos, los cuales muchas veces no tienen relación entre sí más que ser modelos de variable latente. Por ejemplo, las GANs y los VAEs son dos tipos de modelo generativo de variable latente que funcionan de manera muy diferente. Cada uno de estos paradigmas, al ser entrenados bajo criterios distintos, pueden aprender comportamientos y patrones distintos, incluso si son entrenados sobre un mismo conjunto de datos. Por ejemplo, en el caso de las GANs y los VAEs, se observa que las GANs son buenas generando de forma clara los bordes de los objetos, mientras que los VAEs generan bordes más borrosos. Sin embargo, las GANs tienen dificultades para aprender muestras con mayor variabilidad, mientras que los VAEs son capaces de capturar mejor el soporte de la distribución desconocida $p_{\text{data}}(x)$ que se busca aproximar.

En la siguiente figura se ve el dataset swiss roll [43], el cual será usado en todas las implementaciones iniciales (excepto en los ARMs) para introducir cada uno de los paradigmas generativos. En línea con la hipótesis de la variedad, se observa que, si bien las muestras viven en el espacio ambiente $\mathbb{R}^3$, estas siguen una estructura que pareciera poder explicarse, al menos de forma aproximada, con solo dos grados de libertad.

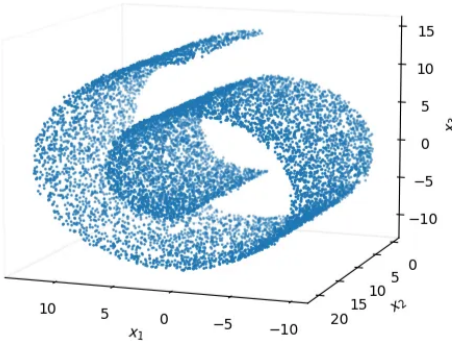

Figura 35: Swiss roll [43].

Más en general, se han encontrado dimensiones intrínsecas para algunos datasets clásicos, lo que valida empíricamente la hipótesis de la manifold. Por ejemplo, para el dataset MNIST se ha probado experimentalmente que su dimensión intrínseca es cercana a 12, lo cual es considerablemente menor que la cantidad de pixeles de las imágenes de este dataset, donde $D = 728$. Por otro lado, en los casos donde la hipótesis de la variedad no se cumple (o donde no existe realmente una variable latente), el considerar un modelo de variable latente no perjudica la formulación ya que si el modelo real desconocido no fuese de variable latente, siempre se puede terminar aprendiendo que $x \perp z$, es decir, $p(x \mid z) = p(x)$.

Vista como una red bayesiana, la distribución conjunta de un modelo de variable latente se factoriza de forma causal como $p(x, z) = p(z)\, p(x \mid z)$, donde $p(z)$ es el prior (se llama así ya que no está condicionado a nada) sobre la variable latente z y $p(x \mid z)$ es la distribución condicional de los datos observados dado el valor de la variable latente z (que no es lo mismo que la marginal $p(x) = \int_{\mathbb{R}^L} p(x, z)\, \mathrm{d}z$). En estos casos, el modelo $p(x \mid z)$ se interpreta como un modelo generador, dada una semilla inicial z, por lo que también se le conoce como **decoder**. Es importante notar que $x \in \mathbb{R}^D$ está representando todas las variables observables unidas en una única variable con el fin de simplificar la notación. Por ejemplo, para un modelo que modifica una imagen de acuerdo a una descripción, x podría ser un vector formado por algún embedding de la imagen original y del texto con las instrucciones de edición. El mismo principio aplica para $z \in \mathbb{R}^L$, por lo que a z se le puede decir la variable latente o las variables latentes.

Por otra parte, a excepción de los modelos autorregresivos (usados en texto) y los modelos basados en energía (ya no tan usados en IA generativa, al menos en su versión original),

todos los paradigmas generativos modernos son modelos de variable latente, por lo que es necesario hacer hipótesis sobre la distribución prior $p(z)$. Por lo general, es usual considerar $p(z) \sim \mathcal{N}(0, \mathrm{I}_L)$ ya que funciona bien al desarrollar la matemática de los distintos modelos y, además, es una distribución desde la que es fácil generar muestras. Esto último es una propiedad importante ya que prácticamente todos los modelos de variable latente (como las GANs, los VAEs o los modelos de difusión) funcionan generando una muestra latente inicial $z \sim p(z)$ y luego generan una nueva muestra (e.g., una imagen) utilizando el valor de z en la distribución $p(x \mid z)$. Además, al estar considerando una matriz de covarianzas diagonal en $p(z) \sim \mathcal{N}(0, \mathrm{I}_L)$, se está imponiendo explícitamente que todas las coordenadas de la variable latente sean independientes entre sí, lo cual no es raro si se interpretan las variables latentes como el conjunto de características esenciales que poseerán las muestras que se generen desde $p(x \mid z)$.

Desde el punto de vista de la generación en modelos de variable latente, se observa que el algoritmo de ancestral sampling sí entrega muestras de la distribución marginal $p_\theta(x)$ ya que

$$p_\theta(x) = \int_{\mathbb{R}^L} p_\theta(x \mid z)\, p(z)\, \mathrm{d}z = \mathbb{E}_{p(z)}[p_\theta(x \mid z)],$$

por lo que se puede ver el sampling $z \sim p(z) \to x \sim p(x \mid z)$ como una aproximación de Monte Carlo (con una única muestra) de la esperanza $\mathbb{E}_{p(z)}[p_\theta(x \mid z)]$.

A continuación se revisarán algunas de las ventajas que poseen los modelos generativos basados en variable latente.

Reducción de la dimensionalidad

Si bien el proceso generativo de un modelo de variable latente es de la forma $z \to p(x \mid z)$, es posible considerar la dirección opuesta, $x \to p(z \mid x)$, con el fin de recuperar la información primitiva que generó una muestra $x \in \mathbb{R}^D$. Esta distribución posterior $p(z \mid x)$ puede interpretarse como una representación (estocástica) más compacta de x, la cual muchas veces es interpretable o útil. Por ejemplo, si $p(x)$ es una distribución que genera imágenes aleatorias de círculos de color negro con una resolución de $D = 256 \times 256 = 65536$, el proceso de generación se podría reducir a una variable latente z en $\mathbb{R}^3$ que genera ternas (r, h, k) con $r > 0$ indicando el radio del círculo y $(h, k) \in \mathbb{R}^2$ las coordenadas de su centro. Con esto, la distribución generadora $p(x \mid z)$ puede ser la función determinista que grafica el respectivo círculo y genera la imagen.

La tarea de aprender buenas representaciones latentes se denomina **representation learning** y puede ser vista como una forma robusta de hacer reducción de dimensionalidad. Por ejemplo, el modelo Stable Diffusion [3] de Stability AI entrena un modelo de variable

latente (tipo VAE) para luego poder entrenar otro modelo generativo más potente (tipo difusión) en el espacio latente del modelo anterior. Esta técnica de aplicar un modelo generativo (e.g., difusión) en el espacio latente de otro modelo (e.g., un VAE) es usada prácticamente en todos los modelos generativos de variable latente que hay hoy en día ya que permite disminuir el costo y tiempo de entrenamiento considerablemente al poder trabajar con representaciones de menor dimensión.

Interpolación semántica en el espacio latente

La representación latente $p(z \mid x)$ de una muestra $x \in \mathbb{R}^D$, aparte de actuar como una representación más compacta de x, suele ser semánticamente más rica que la representación original, en el sentido que es posible desplazarse en el espacio latente para cambiar algunas propiedades semánticas de la muestra, entregando una forma de **interpolación semántica** entre dos objetos.

Si $x_0, x_1 \in \mathbb{R}^D$ son dos imágenes, una tarea natural es querer interpolar entre estas dos imágenes, realizando una transición suave de una imagen a otra. Un enfoque naïve es interpolar linealmente en el espacio de pixeles, donde se considera como imagen interpolada a $x_t = tx_1 + (1-t)x_0$, para $t \in [0, 1]$. Esta interpolación consiste en ir desplazando cada pixel a lo largo de la recta que unen los pixeles homólogos en las imágenes x_0 y x_1. Sin embargo, dado que los valores de los pixeles representan los colores de la imagen, esta interpolación es una interpolación lineal entre la intensidad de color de cada pixel individual, los cuales no contienen información semántica acerca de la imagen, por lo que la imagen interpolada, $x_t \in \mathbb{R}^D$, suele ser una imagen sin mucho sentido.

Un enfoque que funciona mejor consiste en utilizar un modelo generativo de variable latente, $p(x, z) = p(z)\, p(x \mid z)$. De este modo, si una función $x \mapsto z = \text{encoder}(x)$ genera una muestra desde la distribución posterior $p(z \mid x)$ (distribución obtenida, por ejemplo, usando la regla de Bayes), se pueden obtener representaciones latentes de x_0 y x_1 mediante $z_0 = \text{encoder}(x_0)$ y $z_1 = \text{encoder}(x_1)$ respectivamente. Con estas representaciones, se puede realizar una interpolación en el espacio latente del modelo generativo mediante $z_t = tz_1 + (1-t)z_0 \in \mathbb{R}^L$. Luego, si $z \mapsto x = \text{decoder}(z)$ genera una muestra desde la distribución $p(x \mid z)$, entonces $x_t = \text{decoder}(z_t) \in \mathbb{R}^D$ es una imagen interpolada pero en el espacio latente de las imágenes x_0 y x_1. Dado que el espacio latente suele estar asociado a atributos específicos de las muestras que se generarán, la interpolación en el espacio latente suele ser mucho más natural que la interpolación obtenida en el espacio de pixeles debido a que resulta ser una interpolación de propiedades esenciales de las muestras a generar y no solo una interpolación pixel a pixel. Por otro lado, si bien se podrían realizar otro tipo de interpolaciones en el espacio latente (e.g., slerp), la interpolación lineal suele ser suficiente. De hecho, algunos estudios han probado empíricamente que la variedad latente aprendida por los modelos de

variable latente a menudo tiene una curvatura cercana a cero, por lo que se asemeja a un espacio euclidiano, donde la interpolación lineal es el método de interpolación natural.

En la siguiente imagen se observan interpolaciones semánticas en el espacio latente de una GAN, donde cada fila realiza la interpolación entre las imágenes de la primera y última columna:

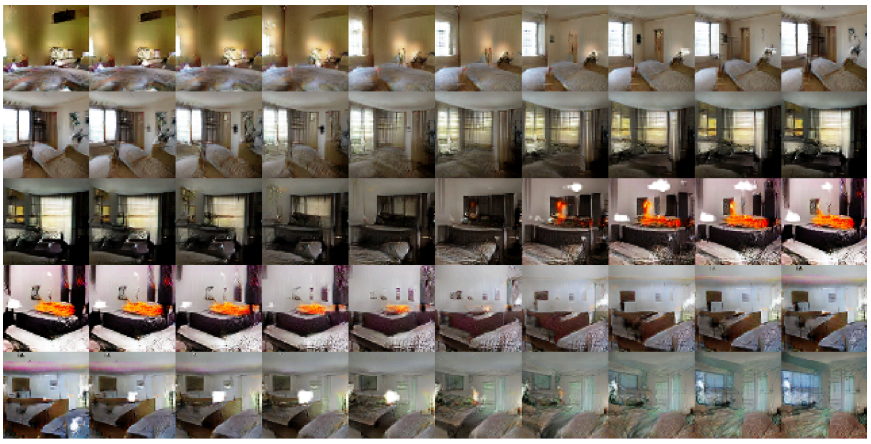

Figura 36: Interpolación latente usando una DCGAN. Imagen obtenida desde [44].

Otra herramienta que entrega la continuidad del espacio latente es la posibilidad de modificar la intensidad de algún atributo de una muestra. Por ejemplo, en la siguiente imagen se logra modificar la intensidad del atributo sonrisa de la imagen que está en la primera columna:

Figura 37: Modificación de atributos usando un VAE. Imagen obtenida desde [45].

Por otro lado, es importante mencionar que por lo general no es posible encontrar una función $x \mapsto z = \text{encoder}(x)$ para las GANs, por lo que este procedimiento solo se puede

realizar cuando se conoce de antemano la variable latente que genera a la respectiva muestra x. En cambio, los VAEs cuentan desde un comienzo con un modelo encoder, $q(z \mid x)$, por lo que sí es posible realizar este tipo de interpolaciones para muestras genéricas.

2.3 Estimación de parámetros

Dada una red bayesiana, para conocer la distribución conjunta $p(x_1, ..., x_N)$ es necesario y suficiente conocer cada uno de los factores $p(x_n \mid \text{Pa}(x_n))$ que componen la factorización que asume la red bayesiana, $p(x_1, ..., x_N) = \prod_{n=1}^{N} p(x_n \mid \text{Pa}(x_n))$. Si bien el desarrollo realizado hasta el momento permite considerar variables con valores desconocidos (variables latentes), siempre se ha asumido que todos los parámetros de las distribuciones $p(x_n \mid \text{Pa}(x_n))$ son conocidos (e.g., si uno de los factores $p(x_n \mid \text{Pa}(x_n))$ es una distribución gaussiana, se conocen los valores de su media y de su varianza para los distintos valores de las variables $\text{Pa}(x_n)$). Sin embargo, en los modelos de IA generativa lo usual es no conocer dichos parámetros y solo contar con un conjunto de muestras (dataset de entrenamiento) generadas desde la distribución desconocida $p_{\text{data}}(x_1, ..., x_N)$ que se busca aprender. El proceso de estimar los valores de estos parámetros desconocidos (lo que usualmente se logra entrenando redes neuronales) se denomina **inferencia**, y la forma más usual de hacer inferencia (puntual) es siguiendo el criterio de máxima verosimilitud.

De manera general, inferir es la acción de obtener una conclusión lógica a partir de un conjunto de premisas que se asumen como ciertas. La inferencia estadística corresponde a realizar inferencia sobre una distribución a partir de un conjunto de muestras observadas, donde el objetivo más usual es inferir los parámetros de la distribución que (supuestamente) generó los datos. Dentro de la inferencia estadística se encuentra la inferencia bayesiana, donde se utiliza la regla de Bayes para hacer inferencia sobre variables desconocidas. De esta forma, la inferencia bayesiana evalúa la probabilidad de que una hipótesis sea cierta a partir de una creencia a priori sobre la hipótesis y de un conjunto de muestras observadas, las cuales actualizan la información asumida por el prior previo a la observación.

Sobre una red bayesiana, se pueden realizar 3 tipos de inferencia bayesiana: inferencia sobre variables no observadas (i.e., estimar el valor de variables desconocidas), inferencia sobre los parámetros de alguna de las distribuciones $p(x_n \mid \text{Pa}(x_n))$ del grafo, e inferencia sobre el grafo mismo (i.e., aprender la estructura del grafo), aunque esta última tarea de inferencia no se trabajará. Para la tarea de inferencia sobre los valores de variables desconocidas, se puede considerar como ejemplo un modelo de variable latente $p(x, z) = p(z)\, p(x \mid z)$, donde se asume que se conoce el valor de la variable observable x pero no el valor de la variable latente z. De este modo, se puede inferir la distribución posterior $p(z \mid x)$ mediante la regla de Bayes:

$$p(z \mid x) = \frac{p(x \mid z)\, p(z)}{p(x)}$$

Notar que la distribución marginal $p(x) = \int_{\mathbb{R}^L} p(x, z)\, \mathrm{d}z$ (o $p(x) = \sum_z p(x, z)$ si z es variable latente discreta) es costosa de obtener ya que requiere integrar (sumar en el caso discreto) sobre todos los posibles valores de la variable latente z.

En las tareas de IA generativa, la red bayesiana de cada paradigma generativo suele ser conocida, por lo que no se debe hacer inferencia sobre el grafo. En cambio, el foco suele estar en hacer inferencia sobre los parámetros de las distribuciones del modelo gráfico, los cuales permitirán, posteriormente, generar nuevos datos utilizando las distribuciones aprendidas. Los modelos modernos consideran, por lo general, que cada uno de los factores $p(x_n \mid \mathrm{Pa}(x_n))$ sigue una distribución simple (e.g., una distribución gaussiana si x_n es una variable continua o una distribución Bernoulli si x_n es binario), lo que facilita el desarrollo de la matemática al construir las funciones de pérdida o demostrar propiedades necesarias para la formulación de los modelos. Dada la simpleza de las distribuciones comúnmente utilizadas, la complejidad de las distribuciones $p(x_n \mid \mathrm{Pa}(x_n))$ debe ser trasladada a los parámetros que las definen, lo cual se suele conseguir utilizando redes neuronales que estimen dichos parámetros. Por ejemplo, en el caso continuo donde se suele elegir $p(x_n \mid \mathrm{Pa}(x_n)) \sim \mathcal{N}(\mu_n, \sigma_n^2)$, los parámetros $\mu_n \in \mathbb{R}$ y $\sigma_n^2 > 0$ suelen ser las salidas de redes neuronales cuyas entradas son los valores de las variables en $\mathrm{Pa}(x_n)$. Sin embargo, también es usual fijar algunos parámetros menos relevantes como las varianzas de las distribuciones gaussianas para poder disminuir la complejidad del modelo.

Si $\theta_n \in \mathbb{R}^{P_n}$ denota el vector de parámetros desconocidos de la distribución $p(x_n \mid \mathrm{Pa}(x_n))$ y $\theta = (\theta_1, ..., \theta_N) \in \mathbb{R}^P$ (con $P = \sum_{n=1}^N P_n$) denota el vector de todos los parámetros desconocidos de la red bayesiana, la forma usual de hacer inferencia sobre θ es considerar que estos parámetros también son variables del modelo cuyo valor no se conoce. De esta forma, se puede hacer inferencia sobre los parámetros haciendo inferencia sobre estas nuevas variables aleatorias. La distribución conjunta de este nuevo modelo extendido se factoriza como:

$$\begin{aligned} p(x, \theta) &= p(\theta)\, p(x \mid \theta) \\ &= p(\theta) \prod_{n=1}^{N} p(x_n \mid \mathrm{Pa}(x_n), \theta_n) \end{aligned}$$

La distribución $p(\theta)$ es un **prior** sobre los parámetros (recordar que se están tratando los parámetros como variables aleatorias), el cual se puede considerar uniforme ($p(\theta) \propto 1$) si no se quiere sesgar la estimación de los parámetros hacia ningún valor. La cantidad $p(x \mid \theta)$ se llama **verosimilitud** (de los parámetros) e indica la probabilidad de generar la

muestra observada x cuando se considera el vector $\theta = (\theta_1, ..., \theta_N)$ como los parámetros de la red bayesiana. De esta forma, se puede obtener la distribución posterior $p(\theta \mid x)$ para los parámetros $\theta \in \mathbb{R}^P$ luego de conocer el valor de la variable conocida x. Por regla de Bayes:

$$p(\theta \mid x) = \frac{p(x \mid \theta)\, p(\theta)}{p(x)},$$

donde $p(x \mid \theta) = \prod_{n=1}^{N} p(x_n \mid \mathrm{Pa}(x_n), \theta_n)$ es la verosimilitud y la constante de normalización $p(x) = \int_{\mathbb{R}^P} p(x, \theta)\, \mathrm{d}\theta$ es una cantidad independiente de θ, la cual representa la probabilidad media de observar el valor conocido x a lo largo de todos los posibles modelos (cada uno determinado por su respectivo valor de θ).

De aquí en adelante se asumirá siempre que cada vector $\theta_n \in \mathbb{R}^{P_n}$ (parámetros de la distribución $p(x_n \mid \mathrm{Pa}(x_n))$) será estimado usando una red neuronal cuyas entradas serán los valores de las variables en $\mathrm{Pa}(x_n)$ y la salida será la estimación de la red neuronal para el vector de parámetros $\theta_n \in \mathbb{R}^{P_n}$. Como es usual, si existe alguna restricción para $\theta_n \in \mathbb{R}^{P_n}$, esta será impuesta en la última capa de la red neuronal. Por ejemplo, si θ_n debe ser un vector de probabilidad, es usual utilizar la función softmax en la salida, mientras que si solo es un escalar que debe estar en el intervalo $[0, 1]$ (e.g., para el parámetro de una distribución Bernoulli), es usual utilizar la función logística (llamada sigmoide en deep learning), $\sigma(x) \coloneqq \frac{1}{1+e^{-x}}$. También es usual utilizar otras funciones de activación en la salida como por ejemplo $\tanh(x) \coloneqq \frac{e^x - e^{-x}}{e^x + e^{-x}}$, la cual restringe la salida al intervalo $[-1, 1]$ (notar que $\tanh(x) = 2\sigma(2x) - 1$), o $\mathrm{relu}(x) \coloneqq \max\{0, x\}$, la cual filtra los valores negativos.

2.3.1 Máximo a posteriori

Cuando se realiza inferencia bayesiana sobre el vector de parámetros $\theta \in \mathbb{R}^P$, la regla de Bayes entrega una distribución posterior $p(\theta \mid x)$, la cual corrige la distribución que asume el prior $p(\theta)$ (recordar que ahora se está tratando a θ como variable aleatoria) teniendo en consideración el valor que tomó la variable x. Sin embargo, muchas veces lo que realmente se busca es una estimación puntual $\hat{\theta} \in \mathbb{R}^P$ de los parámetros y no una distribución completa (aunque esto último es más informativo que una estimación puntual). Más aún, la estimación usualmente no se realiza utilizando una única observación $x \in \mathbb{R}^D$, sino que se utiliza un conjunto de observaciones, $\mathcal{D} = \{x^1, ..., x^K\} \subset \mathbb{R}^D$, las que se asumen como muestras provenientes desde $p(x)$. En consecuencia, la distribución posterior que se obtiene para los parámetros $\theta \in \mathbb{R}^P$ es $p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta)\, p(\theta)}{p(\mathcal{D})}$, donde $p(\mathcal{D} \mid \theta)$ es la verosimilitud de θ una vez se conoce el conjunto de observaciones $\mathcal{D}$.

Un enfoque natural para elegir una estimación puntual de los parámetros θ a partir de la posterior $p(\theta \mid \mathcal{D})$ es el de **máximo a posteriori** (MAP), el cual consiste en elegir a θ

como el vector de parámetros más probable luego de conocer las observaciones en $\mathcal{D}$ (i.e., como la moda de $p(\theta \mid \mathcal{D})$):

$$\begin{aligned}\hat{\theta}_{\text{MAP}} &= \arg\max_{\theta} p(\theta \mid \mathcal{D}) \\ &= \arg\max_{\theta} p(\mathcal{D} \mid \theta)\, p(\theta)\end{aligned}$$

Es importante notar que este enfoque entrega, por primera vez, una función objetivo con la cual se pueden entrenar las redes neuronales que estiman los parámetros $\theta_n \in \mathbb{R}^{P_n}$ de cada distribución $p(x_n \mid \text{Pa}(x_n), \theta_n)$ de la red bayesiana.

Por otro lado, en vez de maximizar directamente la posterior $p(\theta \mid \mathcal{D})$, es más usual maximizar $\log p(\theta \mid \mathcal{D})$. Si bien ambos problemas son equivalentes (ya que el logaritmo es estrictamente creciente), en la siguiente sección se verá que el segundo problema es más estable numéricamente, lo cual es importante para un entrenamiento estable de las redes neuronales. De esta forma, el problema de optimización para el enfoque MAP es el siguiente:

$$\hat{\theta}_{\text{MAP}} = \arg\max_{\theta} \log p(\mathcal{D} \mid \theta) + \log p(\theta)$$

Por simplicidad, de aquí en adelante se usará la notación estándar $p_\theta(x) \coloneqq p(x \mid \theta)$ para indicar la distribución que sigue x cuando se fijan los parámetros al valor $\theta \in \mathbb{R}^P$. Del mismo modo, para cada factor de la red bayesiana, se escribirá $p_{\theta_n}(x_n \mid \text{Pa}(x_n)) \coloneqq p(x_n \mid \text{Pa}(x_n), \theta_n)$. Además, si una distribución no lleva un símbolo de parámetro como subíndice, se asumirá que todos sus parámetros son conocidos (i.e., son distribuciones fijas). Esto último será usual al fijar, por ejemplo, distribuciones priors como una distribución estándar (e.g., $p_\theta(z) = p(z) \sim \mathcal{N}(0, \text{I}_L)$ en un modelo de variable latente).

Relación con weight decay

Reescribir la maximización de $p(\theta \mid x)$ como la maximización de $\log p(\theta \mid x)$ permite ver que algunas elecciones para el prior $p(\theta)$ son equivalentes a aplicar técnicas de regularización clásicas durante la maximización de la verosimilitud. Por ejemplo, si se considera un prior gaussiano $p(\theta) \sim \mathcal{N}\big(0, \frac{1}{\lambda}\text{I}_P\big)$, con $\lambda > 0$ un hiperparámetro fijo, MAP resulta ser equivalente a maximizar la log-verosimilitud $\log p_\theta(\mathcal{D})$ incluyendo regularización l^2 sobre los parámetros (llamado **weight decay** en PyTorch), lo cual es directo al ver que los problemas de optimización son equivalentes:

$$\begin{aligned}\log p(\theta \mid \mathcal{D}) &= \log p_\theta(\mathcal{D}) + \log p(\theta) \\ &= \log p_\theta(\mathcal{D}) - \frac{\lambda}{2}\|\theta\|^2 + \text{constante},\end{aligned}$$

donde se usó que $p(\theta) \propto \exp\left(-\frac{\lambda}{2}\|\theta\|^2\right)$. Se observa que el parámetro de varianza inversa (precisión), $\lambda > 0$, define la importancia que se le da al término regularizador, lo cual es esperable, ya que una varianza pequeña ($\lambda \gg 0$) concentra fuertemente su masa alrededor de la media $0 \in \mathbb{R}^P$ de la distribución prior $p(\theta)$, mientras que una varianza alta ($\lambda \approx 0$) distribuye más uniformemente la masa sobre el soporte. De forma análoga, considerar un prior de Laplace equivale a utilizar LASSO (regularización l^1) sobre los parámetros. Notar que, en ambos casos, los priors están centrados alrededor del origen sesgando al modelo a usar parámetros pequeños, lo cual se espera que disminuya tanto la complejidad del modelo (para evitar overfitting) como su varianza (para un entrenamiento más estable).

También es posible hacer otras elecciones de priors convenientes para la optimización. Una alternativa usual en machine learning clásico es el uso de priors conjugados, donde se elige un prior $p(\theta)$ con una forma específica tal que la posterior $p(\theta \mid \mathcal{D}) \propto p_\theta(\mathcal{D})\, p(\theta)$ pertenezca a la misma familia, lo que resulta conveniente para el cálculo de productos que entrega la regla de Bayes. Sin embargo, si bien este enfoque suele entregar interpretabilidad al modelo, no se suele usar en modelos de IA generativa debido a que se prefiere elegir, implícitamente, un prior gaussiano al regularizar el entrenamiento usando weight decay.

Por último, es importante mencionar que algunas veces, sobre todo en la comunidad de machine learning, la expresión hacer inferencia se usa para referirse a la acción de realizar una predicción con un modelo ya entrenado y no a ajustar los parámetros de un modelo, a lo cual se le suele decir entrenar el modelo. Estos dos usos para el término inferencia no son contradictorios entre sí ya que realizar una predicción con un modelo puede verse como hacer inferencia sobre variables no conocidas. Sin embargo, es importante tener presente que, al menos desde una perspectiva bayesiana, entrenar un modelo también es una forma de realizar inferencia.

2.4 Criterio de máxima verosimilitud

Cuando se utiliza un prior uniforme sobre los parámetros, $p(\theta) \propto 1$, el criterio de MAP se reduce al **criterio de máxima verosimilitud**, el cual es el enfoque estándar para la optimización de modelos probabilísticos en machine learning:

$$\hat{\theta}_{\text{MLE}} = \arg\max_{\theta} \log p_\theta(\mathcal{D})$$

En IA generativa, todos los modelos que se estudiarán, a excepción de las GANs, se suelen entrenar siguiendo un enfoque relacionado, en algún sentido, con el enfoque de máxima verosimilitud.

Por otro lado, es usual asumir que todas las muestras en $\mathcal{D} = \{x^1, ..., x^K\}$ fueron generadas de forma independiente a partir de una distribución fija y desconocida, $p_{\text{data}}(x)$, por lo que

se dice que las muestras en $\mathcal{D}$ son independientes e idénticamente distribuidas (i.i.d.). Esta suposición motiva a considerar siempre una hipótesis de independencia condicional entre las muestras, $x^i \perp x^j \mid \theta$, lo cual permite escribir $p_\theta(x^i, x^j) = p_\theta(x^i)\, p_\theta(x^j)$. Más aún, esta independencia condicional permite descomponer la log-verosimilitud sobre todo el dataset de entrenamiento, $\log p_\theta(\mathcal{D})$, como una suma de log-verosimilitudes individuales:

$$\begin{aligned}\log p_\theta(\mathcal{D}) &= \log \prod_{k=1}^{K} p_\theta(x^k) \\ &= \sum_{k=1}^{K} \log p_\theta(x^k)\end{aligned}$$

Notar que esta separación en una suma es uno de los principales beneficios de optimizar la log-verosimilitud en vez de la verosimilitud directamente (donde quedaría un producto de verosimilitudes). Por otro lado, la mayoría de las distribuciones comúnmente usadas en machine learning (e.g., gaussiana o exponencial) son log-cóncavas (i.e., el logaritmo de su función de densidad es una función cóncava), por lo que para maximizar la log-verosimilitud $\log p_\theta(\mathcal{D})$ basta derivar e igualar a cero (i.e., la condición de 1º orden necesaria para la optimalidad también es suficiente). Además, dado que las probabilidades están siempre en el intervalo $[0, 1]$, el producto de muchas probabilidades (e.g., en $\prod_{k=1}^{K} p_\theta(x^k)$) puede provocar problemas numéricos (y, por lo tanto, un entrenamiento inestable) al estar trabajando con números muy pequeños. Usando logaritmos, el producto se transforma en suma y el intervalo de probabilidades $(0, 1]$ se estira a todo el semieje real $(-\infty, 0]$.

Para fijar conceptos, dado un dataset $\mathcal{D}$, la función $l_\mathcal{D}(\theta) \coloneqq \log p_\theta(\mathcal{D})$ se denomina función de log-verosimilitud, y es una función de los parámetros $\theta \in \mathbb{R}^P$ y no de los datos observados (el conjunto de muestras $\mathcal{D}$ se considera fijo, y cada conjunto $\mathcal{D}$ define una función de log-verosimilitud distinta). Notar que la función de log-verosimilitud no es una (log-)distribución ya que, si bien $p_\theta(\mathcal{D}) \coloneqq p(\mathcal{D} \mid \theta)$ es una distribución sobre $\mathcal{D}$, no lo es sobre θ debido a que $\int_{\mathbb{R}^P} p_\theta(\mathcal{D})\, \mathrm{d}\theta \neq 1$. Por otro lado, el cálculo de la log-verosimilitud $\log p_\theta(\mathcal{D})$ puede realizarse de forma secuencial (i.e., una muestra a la vez) o de forma paralela en batches, ya que la descomposición $\log p_\theta(\mathcal{D}) = \sum_{k=1}^{K} \log p_\theta(x^k)$ permite evaluar la log-verosimilitud de forma individual en cada muestra y luego sumar las salidas. Más aún, cuando el modelo no es de variable latente (i.e., todas las variables, a excepción de los parámetros, son observadas durante el entrenamiento), la optimización de los parámetros $\theta = (\theta_1, ..., \theta_N)$ puede realizarse de forma independiente en cada nodo de la red bayesiana $p(x) = \prod_{n=1}^{N} p_{\theta_n}(x_n \mid \mathrm{Pa}(x_n))$, lo que permite paralelizar el proceso de entrenamiento por nodos (e.g., se podría computar cada nodo en una GPU distinta si se están usando redes neuronales). En efecto, para una muestra de entrenamiento $x = (x_1, ..., x_N) \in \mathcal{D}$, su log--verosimilitud se descompone como:

$$\begin{aligned}\log p_\theta(x) &= \log\left(\prod_{n=1}^{N} p_{\theta_n}(x_n \mid \mathrm{Pa}(x_n))\right) \\ &= \sum_{n=1}^{N} \log p_{\theta_n}(x_n \mid \mathrm{Pa}(x_n)),\end{aligned}$$

por lo que, para obtener el estimador de máxima verosimilitud (MLE), se puede calcular individualmente el MLE para cada parámetro $\theta_n \in \mathbb{R}^{P_n}$ mediante $\arg\max_{\theta_n} \log p_{\theta_n}(x_n \mid \mathrm{Pa}(x_n))$ ya que θ_n no influye en ningún otro sumando de la log-verosimilitud $\log p_\theta(x)$. Del mismo modo, si se realiza inferencia usando el conjunto de observaciones $\mathcal{D} = \{x^1, ..., x^K\}$, el MLE para el parámetro θ_n se obtiene resolviendo el problema de optimización

$$\arg\max_{\theta_n} \sum_{k=1}^{K} \log p_{\theta_n}\left(x_n^k \,\middle|\, \mathrm{Pa}(x_n)\right)$$

La familia de modelos descrita anteriormente se denominan **modelos totalmente observados** ya que todas las variables del modelo, $x = (x_1, ..., x_N)$, son observadas en cada una de las muestras contenidas en el dataset de entrenamiento $\mathcal{D}$. Este tipo de modelos permite obtener la densidad de una muestra de forma exacta y tratable (solo se deben sumar N términos). Más aún, el gradiente de la log-verosimilitud, $\nabla_{\theta_n} \log p_\theta(x) = \nabla_{\theta_n} \log p_{\theta_n}(x_n \mid \mathrm{Pa}(x_n)) \in \mathbb{R}^{P_n}$, se puede obtener fácilmente mediante backpropagation, por lo que es fácil hacer inferencia sobre los parámetros de un modelo totalmente observado usando una red neuronal entrenada mediante el criterio de máxima verosimilitud. Más aún, si se trabaja con mini-batches de entrenamiento, $\mathcal{B} \subset \mathcal{D}$, entonces la cantidad $\frac{1}{|\mathcal{B}|} \log p_\theta(\mathcal{B})$ es un estimador insesgado de la verosimilitud media, $\frac{1}{|\mathcal{D}|} \log p_\theta(\mathcal{D})$, mientras que $\frac{1}{|\mathcal{B}|} \nabla_\theta \log p_\theta(\mathcal{B})$ es un estimador insesgado del gradiente $\frac{1}{|\mathcal{D}|} \nabla_\theta \log p_\theta(\mathcal{D})$. Esta observación permite, por ejemplo, interpretar a $\nabla_\theta \log p_\theta(\mathcal{B})$ como un gradiente estocástico que le entrega aleatoriedad al modelo (donde la aleatoriedad ocurre en la elección de las muestras que forman el batch $\mathcal{B} \subset \mathcal{D}$), lo cual muchas veces ayuda a evitar que el modelo se quede atrapado en un óptimo local de la función objetivo a optimizar. Por otro lado, el entrenamiento usando batches muchas veces es impuesto por restricciones de hardware, ya que para grandes modelos es usual no poder cargar todos los datos disponibles en la GPU para realizar el entrenamiento.

En conclusión, el criterio de máxima verosimilitud, $\hat{\theta} = \arg\max_\theta \log p_\theta(\mathcal{D})$, es el enfoque más común en modelos totalmente observados como los modelos autorregresivos (usados para generar texto) o los modelos basados en flujos (usados para generar imágenes). Esta función de pérdida permite optimizar los modelos de manera estable (dentro de lo posible; los Transformers a gran escala muchas veces presentan inestabilidades durante el entrenamiento aunque se entrenen usando verosimilitud), lo cual no siempre es así. Por ejemplo, las GANs, que entrenan un objetivo que no está basado en verosimilitud, sufren de muchas

inestabilidades durante su entrenamiento, principalmente por la naturaleza adversativa de su entrenamiento.

2.4.1 Ejemplos

En algunos casos simples es posible encontrar el estimador de máxima verosimilitud de forma cerrada, el cual, aparte de ser único, también suele ser interpretable y natural. En esta subsección se revisarán algunos de estos casos.

MLE para la distribución gaussiana

Si $\mathcal{D} = \{x^1, ..., x^K\} \subset \mathbb{R}$ es un dataset de muestras independientes, se puede considerar el modelo $p_\theta(x) \sim \mathcal{N}(\mu_\theta, \sigma^2)$, donde $\mu_\theta \in \mathbb{R}$ es su media (desconocida) y $\sigma^2 > 0$ es su varianza, la cual, por simplicidad, se asumirá conocida. Para buscar el estimador de máxima verosimilitud para el parámetro μ_θ notar que

$$\begin{aligned}\log p_\theta(\mathcal{D}) &= \sum_{k=1}^{K} \log p\big(x^k\big) \\ &= \sum_{k=1}^{K} \frac{-1}{2\sigma^2}\big(x^k - \mu_\theta\big)^2 + \text{constante}\end{aligned}$$

Para obtener $\arg\max_\theta \log p_\theta(\mathcal{D})$ se puede derivar la log-verosimilitud anterior e igualarla a 0 (condición de 1º orden):

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}\mu_\theta} \log p_\theta(\mathcal{D}) &= \sum_{k=1}^{K} \frac{-1}{\sigma^2}\big(x^k - \mu_\theta\big) \\ &= \frac{-1}{\sigma^2}\left(\sum_{k=1}^{K} x^k - K \cdot \mu_\theta\right) \\ &= 0\end{aligned}$$

Por lo que basta despejar $\mu_\theta \in \mathbb{R}$ para obtener el estimador de máxima verosimilitud:

$$\mu_\theta = \frac{1}{K}\sum_{k=1}^{K} x^k$$

Es decir, el MLE para la media de un modelo gaussiano es simplemente la media empírica de las muestras. Si bien se podría verificar que esta cantidad es un máximo mirando la segunda derivada (o el Hessiano en más dimensiones), aquí no es necesario porque la función a maximizar es cóncava (ya que la suma de funciones cóncavas es cóncava). Por otro lado, si

también se quisiera estimar el parámetro $\sigma^2 > 0$, se puede realizar el mismo procedimiento para llegar a que el MLE de la varianza es simplemente la varianza empírica de las muestras en $\mathcal{D}$.

MLE para la distribución de Bernoulli

Si $\mathcal{D} = \{x^1, ..., x^K\} \subset \{0, 1\}$ es un dataset de muestras binarias independientes, se puede considerar el modelo $p_\theta(x) \sim \text{Bernoulli}(r_\theta)$, donde $r_\theta \in [0, 1]$ es el parámetro (desconocido) de la distribución. Al igual que antes, este parámetro puede ser estimado usando el criterio de máxima verosimilitud. Para esto, notar que:

$$\begin{aligned}\log p_\theta(\mathcal{D}) &= \sum_{k=1}^{K} \log p\big(x^k\big) \\ &= \sum_{k=1}^{K} \big(x^k \log(r_\theta) + \big(1 - x^k\big) \log(1 - r_\theta)\big)\end{aligned}$$

Al igual que antes, para obtener $\arg\max_\theta \log p_\theta(\mathcal{D})$ se derivará la log-verosimilitud anterior y se igualará a cero:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}r_\theta} \log p_\theta(\mathcal{D}) &= \sum_{k=1}^{K} \left(\frac{x^k}{r_\theta} - \frac{1 - x^k}{1 - r_\theta} \right) \\ &= \frac{1}{r_\theta} \sum_{k=1}^{K} x^k - \frac{1}{1 - r_\theta} \sum_{k=1}^{K} \big(1 - x^k\big) \\ &= 0\end{aligned}$$

Por lo que basta despejar $r_\theta \in [0, 1]$ para obtener el estimador de máxima verosimilitud del parámetro desconocido:

$$\begin{aligned}&(1 - r_\theta) \sum_{k=1}^{K} x^k - r_\theta \sum_{k=1}^{K} \big(1 - x^k\big) = 0 \\ &\implies \sum_{k=1}^{K} x^k = r_\theta \left(\sum_{k=1}^{K} x^k + \sum_{k=1}^{K} \big(1 - x^k\big) \right) = r_\theta \cdot K \\ &\implies r_\theta = \frac{1}{K} \sum_{k=1}^{K} x^k\end{aligned}$$

Es decir, el MLE para una distribución de Bernoulli corresponde a la fracción de muestras con valor 1 con respecto al total de muestras en $\mathcal{D}$, lo cual tiene sentido, por ejemplo, al

interpretar el experimento como lanzamiento de monedas, donde la probabilidad de obtener cara, $p(x = 1) = r_\theta$, es (cuando $K \to \infty$) la cantidad de caras que se obtuvieron sobre el total de lanzamientos.

Por otro lado, al igual que en el caso gaussiano, este MLE para r_θ se puede ver como la media empírica de $\mathcal{D}$, la cual, a su vez, puede interpretarse como un estimador de la media real $\mathbb{E}_{p_{\text{data}}(x)}[x] = r_{\text{data}}$ (asumiendo que $p_{\text{data}}(x) \sim \text{Bernoulli}(r_{\text{data}})$, con $r_{\text{data}} \in [0, 1]$ desconocido).

MLE para la regresión lineal

Para un dataset supervisado $\mathcal{D} = \{(x^1, y^1), ..., (x^K, y^K)\} \subset \mathbb{R}^D \times \mathbb{R}$ (donde se asume que $y^k \sim p_{\text{data}}(y^k \mid x^k)$ para una entrada $x^k \in \mathbb{R}^D$ dada), se puede considerar el modelo gaussiano lineal, $p(y \mid x) \sim \mathcal{N}(\theta^\top x, \sigma^2)$ (i.e., $y = \theta^\top x + \epsilon$, con $\epsilon \sim \mathcal{N}(0, \sigma^2)$ un error gaussiano aditivo), donde $\theta \in \mathbb{R}^D$ es el parámetro desconocido a estimar y $\sigma^2 > 0$ es un hiperparámetro fijo (aunque también podría ser estimado por máxima verosimilitud). Al igual que antes, se puede obtener el estimador de máxima verosimilitud para $\theta \in \mathbb{R}^D$, el cual coincide con el regresor lineal óptimo para mínimos cuadrados[1], $\hat{\theta} = (X^\top X)^{-1} X^\top Y \in \mathbb{R}^D$, donde $X \in \mathcal{M}_{K,D}(\mathbb{R})$ es la matriz cuya k-ésima fila corresponde a la muestra $x^k \in \mathbb{R}^D$, mientras que $Y \in \mathbb{R}^K$ es el vector cuya k-ésima coordenada es $y^k \in \mathbb{R}$. Notar que, al igual que antes, este MLE también es natural ya que la matriz $X^\dagger := (X^\top X)^{-1} X^\top \in \mathcal{M}_{D,K}(\mathbb{R})$ corresponde a la pseudoinversa de Moore-Penrose de la matriz $X \in \mathcal{M}_{K,D}(\mathbb{R})$, por lo que $\hat{\theta} = X^\dagger Y$ puede verse como un intento de despejar $\theta \in \mathbb{R}^D$ en el sistema lineal sobredeterminado $Y = X\theta$. Más aún, en línea con lo revisado en la sección anterior, si se agrega un prior gaussiano al parámetro desconocido $\theta \in \mathbb{R}^D$, el estimador MAP coincide con el regresor lineal de mínimos cuadrados con regularización cuadrática (i.e., ridge regression).

2.4.2 Verosimilitud en modelos de variable latente

La principal dificultad de entrenar un modelo de variable latente por máxima verosimilitud es que la cantidad $\log p_\theta(x)$ es costosa de computar. En efecto, para una red bayesiana de variable latente, $p_\theta(x, z) = p_\theta(z)\, p_\theta(x \mid z)$, el cálculo de la log-verosimilitud $\log p_\theta(x)$ requiere marginalizar sobre la variable latente z:

$$\begin{aligned} p_\theta(x) &= \int_{\mathbb{R}^L} p_\theta(x, z)\, \mathrm{d}z \\ &= \int_{\mathbb{R}^L} p_\theta(z)\, p_\theta(x \mid z)\, \mathrm{d}z \end{aligned}$$

[1] Es decir, minimiza el error cuadrático medio $\text{MSE}_{\mathcal{D}}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} (y - \theta^\top x)^2$, el cual es un problema que no necesita nociones probabilísticas.

Esta integral (o suma si z es una variable latente discreta) es muy costosa de calcular, incluso aproximándola numéricamente (hay varias formas de hacerlo; una usual es escribir la igualdad anterior como $p_\theta(x) = \mathbb{E}_{p_\theta(z)}[p_\theta(x \mid z)]$, lo que permite usar una aproximación de Monte Carlo usando muestras desde $p_\theta(z)$), por lo que la verosimilitud para este tipo de modelos se considera intratable en la práctica. Esta dificultad aumenta aún más si se considera que, por lo general, la verosimilitud se debe calcular para un dataset de muestras, $\mathcal{D} = \{x^1, ..., x^K\}$, y en cada iteración del algoritmo de gradiente (asumiendo que los parámetros están siendo aprendidos por redes neuronales).

Por otro lado, la dificultad de calcular $p_\theta(x)$ se hereda al querer calcular el gradiente $\nabla_\theta \log p_\theta(x)$ (necesario para el entrenamiento de las redes neuronales) y al querer hacer inferencia sobre z mediante $p_\theta(z \mid x) = p_\theta \frac{x,z}{p_\theta}(x)$. Además, la marginalización necesaria para calcular $p_\theta(x)$ elimina la posibilidad de optimizar cada nodo del DAG de forma individual como sí ocurre en los modelos completamente observables. En efecto:

$$\begin{aligned}
\log p_\theta(\mathcal{D}) &= \sum_{k=1}^{K} \log p_\theta\left(x^k\right) \\
&= \sum_{k=1}^{K} \log \left(\int_{\mathbb{R}^L} p_\theta(z)\, p_\theta\left(x^k \mid z\right) \mathrm{d}z \right) \\
&= \sum_{k=1}^{K} \log \left(\int_{\mathbb{R}^L} p_\theta(z) \prod_{n=1}^{N} p_{\theta_n}\left(x_n \mid \mathrm{Pa}\left(x_n^k\right)\right) \mathrm{d}z \right)
\end{aligned}$$

Dado que el logaritmo no distribuye sobre la integral, la log-verosimilitud ya no se puede descomponer en una suma que separe los parámetros a optimizar, $\theta = (\theta_1, ..., \theta_N)$. Más aún, la log-verosimilitud que se obtiene es difícil de optimizar debido a su estructura, la cual además provoca que se pierda la log-concavidad que se suele tener para distribuciones sencillas. En consecuencia, los modelos de variable latente por lo general son entrenados utilizando enfoques alternativos al de máxima verosimilitud. Por ejemplo, los VAEs y los modelos de difusión utilizan un enfoque variacional para optimizar una cota inferior de la log-verosimilitud (llamada ELBO), mientras que las GANs utilizan un entrenamiento alternado entre dos modelos que compiten entre sí (de hecho, este tipo de modelos no modela una función de verosimilitud de forma explícita).

Por otro lado, un algoritmo clásico para entrenar un modelo de variable latente genérico es el algoritmo de expectation-maximization (EM), el cual estima las variables latentes a partir de los datos (paso E) y luego realiza inferencia sobre los parámetros usando los valores estimados para las variables latentes (paso M). Otro tipo de modelos, como los modelos basados en energía, reparametrizan la verosimilitud de forma conveniente para luego poder

optimizarla sin tener que calcularla de forma explícita, y luego utilizan algoritmos tipo Markov chain Monte Carlo (MCMC) para poder generar muestras.

2.4.3 Relación con teoría de la información

En esta sección se revisará la conexión y equivalencia entre el enfoque de máxima verosimilitud y algunos conceptos principales de la teoría de la información. Para esto, se comenzará construyendo la función de información de Shannon.

Información de Shannon y entropía

De manera intuitiva, se puede definir la información de un evento aleatorio como la cantidad de sorpresa o conocimiento no trivial que reporta el saber que el evento ocurrió. De esta forma, la ocurrencia de un evento poco probable (e.g., hoy hubo un terremoto) es muy informativo, mientras que eventos más comunes o de ocurrencia más esperable (e.g., mañana saldrá el sol) entregan poca información. Para definir formalmente este concepto, se construirá una función $\mathrm{I} : \mathcal{B} \to \mathbb{R}$ (con $\mathcal{B}$ el conjunto de todos los eventos posibles) que mida la cantidad de información que contiene un evento de acuerdo a su probabilidad de ocurrencia. Para esto, se considerarán los siguientes requisitos naturales sobre la función de información:

1. La información de un evento solo debe depender de su probabilidad. Es decir, para todo evento $A \in \mathcal{B}$, $\mathrm{I}(A) = f(\mathbb{P}(A))$ para alguna función continua $f : [0, 1] \to \mathbb{R}$.
2. La información debe ser no negativa. Además, es nula solo si el evento es completamente seguro. Es decir, para todo evento $A \in \mathcal{B}$, $\mathrm{I}(A) \geq 0$, con igualdad si y solo si $\mathbb{P}(A) = 1$.
3. La información que entrega el saber la ocurrencia de dos eventos independientes es la suma de las informaciones individuales. Es decir, para $A \perp B$ eventos independientes, se debe cumplir que $\mathrm{I}(A \cap B) = \mathrm{I}(A) + \mathrm{I}(B)$, donde $A \cap B \in \mathcal{B}$ es el evento de que ocurran los eventos A y B.

Para encontrar la función $f : [0, 1] \to \mathbb{R}$ correcta se puede comenzar considerando dos eventos independientes, $A \perp B$. Por los requisitos 3 y 1:

$$\begin{aligned} \mathrm{I}(A \cap B) &= \mathrm{I}(A) + \mathrm{I}(B) \\ &= f(\mathbb{P}(A)) + f(\mathbb{P}(B)) \end{aligned}$$

Por otra parte, como $A, B \in \mathcal{B}$ son eventos independientes, también se tiene que $\mathbb{P}(A \cap B) = \mathbb{P}(A)\,\mathbb{P}(B)$. Por lo tanto:

$$\begin{aligned}\mathrm{I}(A \cap B) &= f(\mathbb{P}(A \cap B)) \\ &= f(\mathbb{P}(A)\,\mathbb{P}(B))\end{aligned}$$

Luego, igualando ambas expresiones para $\mathrm{I}(A \cap B)$, se observa que la función $f : [0,1] \to \mathbb{R}$ buscada debe cumplir que $f(x) + f(y) = f(xy)$ para $x, y \in [0,1]$. Una función continua natural que cumple esto es el logaritmo, $f(x) = c \cdot \log(x)$, para cualquier constante $c \in \mathbb{R}$. Más aún, la ecuación funcional de Cauchy permite demostrar que el logaritmo es la única función continua que cumple esta propiedad. Por lo tanto, la definición natural para la información de un evento $A \in \mathcal{B}$ es $\mathrm{I}(A) = c \cdot \log \mathbb{P}(A)$, para algún valor apropiado de $c \in \mathbb{R}$.

Por otro lado, por el requisito de positividad, y notando que $\log \mathbb{P}(A) \leq 0$ para cualquier evento $A \in \mathcal{B}$, necesariamente se debe elegir una constante $c < 0$. Además, considerando esta constante de la forma $c = -\frac{1}{\log} b$ para $b > 1$, la propiedad de cambio de base de los logaritmos permite definir la **función de información de Shannon** (para una base fija $b > 1$) como

$$\mathrm{I}(A) := -\log_b \mathbb{P}(A),$$

donde $A \in \mathcal{B}$ es un evento cualquiera. Cuando $b = 2$ la información se mide en bits, mientras que cuando $b = e$ (constante de Euler), la información se mide en nats. Sin embargo, la base que se utilice es irrelevante ya que las definiciones solo difieren por una constante multiplicativa. En este libro se considerará siempre $b = e$ para trabajar con el logaritmo natural, lo cual es la elección estándar en machine learning.

Teniendo definido el concepto de información de un evento, es posible definir la entropía de una distribución $p(x)$ como la información esperada para una variable $x \sim p(x)$ que sigue dicha distribución:

$$\begin{aligned}\mathrm{H}(p) &:= \mathbb{E}_{p(x)}[\mathrm{I}(x)] \\ &= \mathbb{E}_{p(x)}[-\log p(x)] \\ &= -\int_{\mathbb{R}^D} \log p(x)\, p(x)\, \mathrm{d}x,\end{aligned}$$

donde en la última igualdad se asumió que $x \sim p(x)$ es una variable aleatoria continua. En el caso discreto, en cambio, $\mathrm{H}(p) = -\sum_{n=1}^{N} \log p(k_n) \cdot p(k_n)$, si se considera $\mathrm{supp}(x) = \{k_1, ..., k_N\}$.

Recordando la interpretación de la información de Shannon dada anteriormente, la entropía de una variable aleatoria $x \sim p(x)$ se puede interpretar como el grado de aleatoriedad o incertidumbre que esta posee. Por ejemplo, para $x \sim \mathrm{Bernoulli}(r)$, $\mathrm{H}(p) = -r \cdot \log(r) -$

$(1-r)\cdot\log(1-r)$. Se puede verificar que esta cantidad es máxima cuando $r=\frac{1}{2}$ (máxima incertidumbre) y es mínima cuando $r=0$ o $r=1$ (no hay incertidumbre). De forma similar, la distribución uniforme $x\sim\text{Uniform}([a,b])$ es la de máxima entropía (incertidumbre) entre todas las distribuciones con soporte acotado, mientras que la distribución gaussiana es la de máxima incertidumbre entre todas las distribuciones con soporte $\mathbb{R}$ (y de varianza finita).

Entropía cruzada y divergencia KL

Teniendo definido el concepto de entropía, la entropía cruzada entre dos distribuciones, $p(x)$ y $q(x)$, se define como la información esperada de x pero cuando se considera la distribución p en el cálculo de esperanza y la distribución q en el cálculo de información:

$$\text{CE}(p,q) \coloneqq \mathbb{E}_{p(x)}[-\log q(x)] = -\int_{\mathbb{R}^D} \log q(x)\, p(x)\, \mathrm{d}x$$

Si bien el concepto de entropía cruzada parece menos interpretable que el de entropía, este surge naturalmente cuando se consideran las motivaciones originales de Shannon al introducir la teoría de la información, donde el objetivo principal es poder codificar distintos símbolos (e.g., palabras de un vocabulario) de manera eficiente (símbolos frecuentes se codifican con códigos cortos, mientras que símbolos poco usuales se codifican con códigos más extensos). Bajo esta mirada, el primer teorema de Shannon indica que la entropía $\text{H}(p)$ indica cuántos bits por símbolo (considerando $b=2$) se necesitan en promedio para codificar un mensaje (sin pérdida de información), asumiendo que los símbolos son generados según la distribución $x\sim p(x)$. De esta forma, la entropía cruzada $\text{CE}(p,q)$ corresponde a la cantidad de bits que se requieren (en promedio) para codificar un símbolo cuando el esquema de codificación se optimiza asumiendo que los símbolos siguen una distribución $y\sim q(y)$, cuando en realidad siguen una distribución $x\sim p(x)$.

Por otro lado, la interpretación de compresibilidad de la entropía cruzada no es estrictamente necesaria en el estudio de redes bayesianas ya que, usualmente, la entropía cruzada solo se menciona por la relación que surge al compararla con la verosimilitud $\log p_\theta(\mathcal{D})$ para un conjunto de entrenamiento i.i.d., $\mathcal{D}=\{x^1,...,x^K\}$. En efecto, se puede ver que buscar el estimador de máxima verosimilitud, $\max_\theta \log p_\theta(\mathcal{D})$, equivale a minimizar la entropía cruzada entre la distribución empírica $p_{\mathcal{D}}(x)=\frac{1}{K}\sum_{k=1}^{K}\delta_{x^k}(x)$ y la distribución del modelo que se está ajustando, $p_\theta(x)$. Para ver esto, notar que

$$\begin{aligned}\mathrm{CE}(p_{\mathcal{D}}, p_\theta) &= -\mathbb{E}_{p_{\mathcal{D}}(x)}[\log p_\theta(x)] \\ &= -\frac{1}{K}\sum_{k=1}^{K} \log p_\theta\left(x^k\right) \\ &= -\frac{1}{K}\log\left(\prod_{k=1}^{K} p_\theta\left(x^k\right)\right) \\ &= -\frac{1}{K}\log p_\theta(\mathcal{D}),\end{aligned}$$

por lo que $\arg\max_\theta \log p_\theta(\mathcal{D}) = \arg\min_\theta \mathrm{CE}(p_{\mathcal{D}}, p_\theta)$. Por otro lado, recordando que $\mathbb{E}_{p_{\text{data}}(x)}[...]$ puede ser aproximada mediante una estimación de Monte Carlo usando las muestras de $\mathcal{D}$, también se puede considerar el estimador de máxima verosimilitud como una aproximación de la solución del problema $\min_\theta \mathrm{CE}(p_{\text{data}}, p_\theta)$.

Por otro lado, retomando la interpretación de la entropía cruzada $\mathrm{CE}(p, q)$ como la cantidad de bits promedio que requiere codificar un símbolo considerando una distribución $y \sim q(y)$ cuando en realidad la distribución real es $x \sim p(x)$, se puede descomponer la cantidad $\mathrm{CE}(p, q)$ en un término asociado a la cantidad de información irreducible (asociada a la propia información de $p(x)$) y en un término asociado al error de considerar como distribución de símbolos a $q(x)$ en vez de $p(x)$:

$$\begin{aligned}\mathrm{CE}(p, q) &= \mathbb{E}_{p(x)}[-\log q(x)] \\ &= \mathbb{E}_{p(x)}[-\log p(x)] + \mathbb{E}_{p(x)}\left[\log\left(\frac{p(x)}{q(x)}\right)\right] \\ &= \mathrm{H}(p) + \mathrm{D}_{\mathrm{KL}}(p \,\|\, q),\end{aligned}$$

donde $\mathrm{D}_{\mathrm{KL}}(p \,\|\, q)$ es la **divergencia de Kullback-Leibler** (también llamada entropía relativa) entre $p(x)$ y $q(x)$:

$$\begin{aligned}\mathrm{D}_{\mathrm{KL}}(p \,\|\, q) &:= \mathbb{E}_{p(x)}\left[\log\left(\frac{p(x)}{q(x)}\right)\right] \\ &= \int_{\mathbb{R}^D} \log\left(\frac{p(x)}{q(x)}\right) p(x)\,\mathrm{d}x\end{aligned}$$

Por lo tanto, el operador de Kullback-Leibler puede interpretarse como una medida de discrepancia entre las distribuciones $p(x)$ y $q(x)$, donde la comparación se realiza con el cociente $\frac{p(x)}{q(x)}$. En efecto, si las dos distribuciones son similares, entonces el cociente es cercano a 1, por lo que $\log\left(\frac{p(x)}{q(x)}\right) \approx 0$. En particular, $\mathrm{D}_{\mathrm{KL}}(p \,\|\, q) = 0$ cuando $p = q$.

Por otro lado, se puede probar que la divergencia de Kullback-Leibler es siempre no negativa (esto se conoce como desigualdad de Gibbs). Más aún, dado que $\mathrm{CE}(p_{\mathcal{D}}, p_\theta) = \mathrm{H}(p_{\mathcal{D}}) + \mathrm{D}_{\mathrm{KL}}(p_{\mathcal{D}} \,\|\, p_\theta)$, con $\mathrm{H}(p_{\mathcal{D}}) \geq 0$ una constante independiente de θ, se puede obtener la siguiente equivalencia al enfoque de máxima verosimilitud:

$$\arg\max_\theta \log p_\theta(\mathcal{D}) = \arg\min_\theta \mathrm{D}_{\mathrm{KL}}(p_{\mathcal{D}} \,\|\, p_\theta)$$

En resumen, buscar el estimador de máxima verosimilitud resulta ser equivalente a minimizar la entropía cruzada $\mathrm{CE}(p_{\mathcal{D}}, p_\theta)$, lo que a su vez resulta ser equivalente a minimizar la divergencia $\mathrm{D}_{\mathrm{KL}}(p_{\mathcal{D}} \,\|\, p_\theta)$. Por otra parte, si bien la divergencia de Kullback-Leibler puede verse como una medida de discrepancia entre dos distribuciones de probabilidad, no es una función de distancia (o métrica) en el sentido estricto. En particular, no es simétrica ($\mathrm{D}_{\mathrm{KL}}(p \,\|\, q) \neq \mathrm{D}_{\mathrm{KL}}(q \,\|\, p)$) ni cumple la desigualdad triangular ($\mathrm{D}_{\mathrm{KL}}(p \,\|\, q) \neq \mathrm{D}_{\mathrm{KL}}(p \,\|\, r) + \mathrm{D}_{\mathrm{KL}}(r \,\|\, q)$). La siguiente figura muestra cómo cambia el argumento minimizante dependiendo del orden que se utilice para las distribuciones dentro de la divergencia:

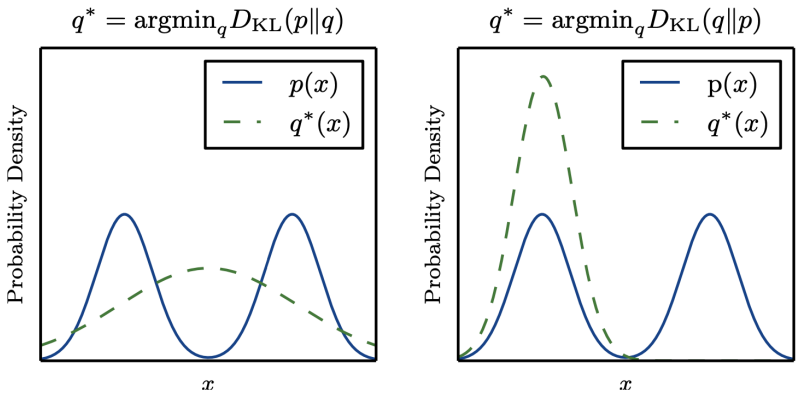

Figura 38: Efecto del orden de los argumentos en la divergencia de Kullback-Leibler sobre el argumento minimizante [46].

En particular, para el caso de la izquierda se observa la creencia usual de que optimizar un modelo generativo maximizando la log-verosimilitud fuerza al modelo a capturar todas las modas de la distribución de los datos de entrenamiento.

El hecho de que la divergencia de Kullback-Leibler no sea una distancia no permite aplicar distintos resultados conocidos sobre espacios métricos, lo cual muchas veces es útil para obtener garantías teóricas (e.g., convergencia) sobre los temas estudiados. Además, si bien es posible adaptar la divergencia de Kullback-Leibler para construir la distancia de Jensen-Shannon (la cual aparece en el estudio de las GANs), $d_{\mathrm{JS}}(p, q)^2 = \frac{1}{2}\mathrm{D}_{\mathrm{KL}}\big(p \,\|\, \frac{p+q}{2}\big) + \frac{1}{2}\mathrm{D}_{\mathrm{KL}}\big(q \,\|\, \frac{p+q}{2}\big)$, esta distancia parece un poco forzada y tampoco tiene muy buenas propiedades como la distancia de Wasserstein, la cual se definirá al revisar algunos tópicos de

transporte óptimo. Por otro lado, si bien la divergencia de Kullback-Leibler no es una distancia, sí es una función de divergencia, lo que permitirá definir las f-GANs, las cuales pueden verse como una generalización de las GANs para distintos funcionales de optimización.

2.4.4 Ejemplo usando una red neuronal

En esta última subsección se implementará un modelo probabilístico de juguete para poder aproximar un estimador de máxima verosimilitud usando una red neuronal. Las librerías que se usarán son las siguientes:

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
```

Como red bayesiana se usará una mixtura (modelo de variable latente $p(x, z) = p(z)\, p(x \mid z)$, donde z es una variable categórica) de dos componentes gaussianos unidimensionales, es decir:

$$p(x) = p(z = 0)\, p_0(x) + p(z = 1)\, p_1(x),$$

donde $p_0(x) \sim \mathcal{N}(\mu_0, \sigma_0^2)$ y $p_1(x) \sim \mathcal{N}(\mu_1, \sigma_1^2)$, mientras que $p(z) \sim \text{Bernoulli}(r)$, con $r \in [0, 1]$ un parámetro que indica el peso asignado a cada distribución. En consecuencia:

$$p(x) = (1 - r) \cdot \frac{1}{\sqrt{2\pi\sigma_0^2}} \exp\left(-\frac{(x - \mu_0)^2}{2\sigma_0^2}\right) + r \cdot \frac{1}{\sqrt{2\pi\sigma_1^2}} \exp\left(-\frac{(x - \mu_1)^2}{2\sigma_1^2}\right)$$

La siguiente clase `Mixture` implementa esta red bayesiana, asumiendo todos los parámetros como conocidos:

```
class Mixture:

    def __init__(self, r: float, gaussians: list[tuple[float, float]]) ->
None:
        self.r = r
        self.gaussians = gaussians  # (mu_0,sigma_0), (mu_1,sigma_1).

    def generate_data(self, n_samples: int = 1000) -> torch.Tensor:
        r = torch.full([n_samples], self.r)
        z = torch.bernoulli(r)
        x = torch.empty(n_samples)
        x[z == 0] = torch.normal(*self.gaussians[0], [(z == 0).sum()])
```

```
        x[z == 1] = torch.normal(*self.gaussians[1], [(z == 1).sum()])
        return x

    @staticmethod
    def p_x(x: torch.Tensor, r: float | torch.Tensor, gaussians:
list[tuple[float, float]]) -> torch.Tensor:
        mu = gaussians[0][0], gaussians[1][0]
        var = gaussians[0][1]**2, gaussians[1][1]**2

        p0 = (2*torch.pi*var[0])**(-1/2) * torch.exp(-1/(2*var[0]) * (x-
mu[0])**2)
        p1 = (2*torch.pi*var[1])**(-1/2) * torch.exp(-1/(2*var[1]) * (x-
mu[1])**2)
        p = (1 - r) * p0 + r * p1
        return p
```

El método `generate_data` genera muestras por ancestral sampling: primero decide desde qué clase generará cada muestra (sampleando desde $p(z) \sim \text{Bernoulli}(r)$), y luego genera las muestras observadas desde las respectivas distribuciones gaussianas. En este caso, se utilizarán como parámetros reales a $(\mu_0, \sigma_0) = (-1, 1)$ y a $(\mu_1, \sigma_1) = (3, 2)$, mientras que como prior de clase se fijará $r = 0.8$. Con estos parámetros fijos, se generarán $|\mathcal{D}| = 1000$ muestras desde la red bayesiana:

```
r = 0.8
gaussians = [(-1, 1), (3, 2)]

# Datos:
dataset = Mixture(r, gaussians)
x = dataset.generate_data()

# Gráfico:
x_plot = torch.linspace(min(x), max(x), 1000)
density = Mixture.p_x(x_plot, r, gaussians)

plt.figure(figsize=(8, 5))
plt.plot(x_plot, density)
plt.hist(x, bins=100, density=True, alpha=0.3)
plt.xlabel('x')
plt.ylabel('Densidad')
plt.grid(alpha=0.3)
plt.show()
```

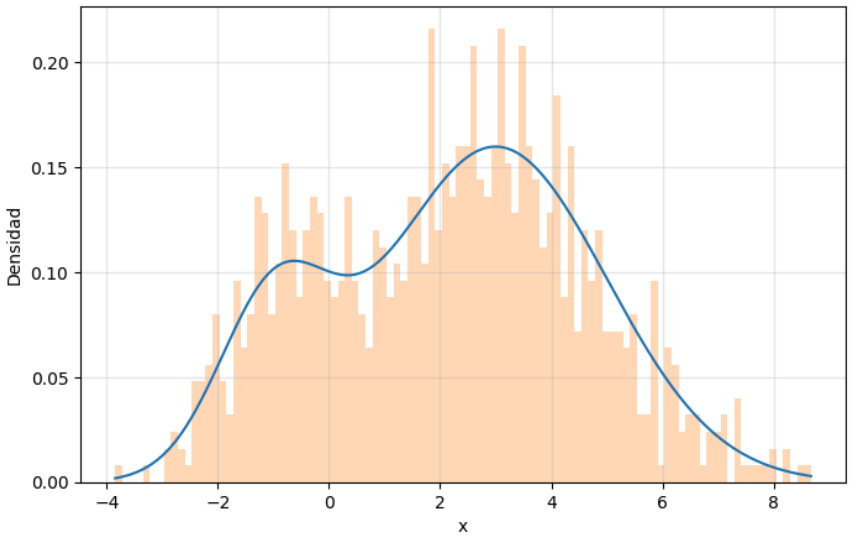

Figura 39: Densidad teórica de la mixtura gaussiana (curva) junto al histograma de las $|\mathcal{D}| = 1000$ muestras generadas por ancestral sampling.

En el gráfico de las muestras se aplicó KDE, lo que permite, en este caso, identificar claramente los valores de los parámetros μ_0 y μ_1. Para ejemplificar el problema de inferencia, se supondrá que el prior de clase r es desconocido, mientras que el resto de parámetros se considerarán conocidos por simplicidad. De esta forma, se usarán las muestras en $\mathcal{D} \subset \mathbb{R}$ para inferir el valor del parámetro $r \in [0, 1]$ usando una red neuronal r_θ con salida sigmoidal. Notar que la distribución $p_\theta(z)$ en $p_\theta(x, z) = p_\theta(z)\, p(x \mid z)$ es incondicional, por lo que la red neuronal no recibe ninguna entrada:

```
class PriorEstimator(nn.Module):

    def __init__(self, init_val: float = 0.5) -> None:
        super().__init__()
        self.logit_r = nn.Parameter(torch.tensor(init_val))

    def forward(self) -> torch.Tensor:
        return torch.sigmoid(self.logit_r)
```

La red neuronal $r_\theta \in [0, 1]$ será optimizada para maximizar la verosimilitud (usando el método `p_x` definido en la clase `Mixture`). Para esto, un detalle importante y que es usado en la práctica es que en vez de maximizar directamente la log-verosimilitud $\log p_\theta(\mathcal{D})$, se suele optimizar la cantidad $\frac{1}{|\mathcal{D}|} \log p_\theta(\mathcal{D})$ ya que al normalizar la log-verosimilitud por el tamaño del dataset de entrenamiento, esta cantidad se vuelve independiente de $|\mathcal{D}|$, lo que evita tener que ajustar la tasa de aprendizaje de acuerdo al tamaño de cada dataset (o batch) de entrenamiento:

```
def train_loop(model: PriorEstimator, optimizer: optim.Optimizer, x:
torch.Tensor, n_epochs: int = 500) -> tuple[list[float], list[float]]:

    normalized_nlls, r_preds = [], []

    for epoch in range(n_epochs):
        r_pred = model()
        p_x = Mixture.p_x(x, r_pred, gaussians)
        normalized_nll = - p_x.log().mean()

        optimizer.zero_grad()
        normalized_nll.backward()
        optimizer.step()

        normalized_nlls.append(normalized_nll.item())
        r_preds.append(r_pred.item())

    return normalized_nlls, r_preds
```

En la función `train_loop` se está almacenando y retornando la dinámica de entrenamiento completa para luego poder visualizarla. Para realizar el entrenamiento solo falta instanciar la red neuronal y un optimizador asociado a su único parámetro:

```
model = PriorEstimator()
optimizer = optim.SGD(model.parameters(), lr=1)

train_loop(model, optimizer, x)

print(f'Estimación: {model().item():.4f}')
print(f'Valor real: {dataset.r}')
```

```
Estimación: 0.7847 Valor real: 0.8
```

Se observa que el valor predicho para el parámetro del prior $p_\theta(z) \sim \text{Bernoulli}(r_\theta)$ es cercano al valor real. De hecho, la convergencia puede ocurrir en pocas iteraciones, dependiendo de la tasa de aprendizaje que se utilice:

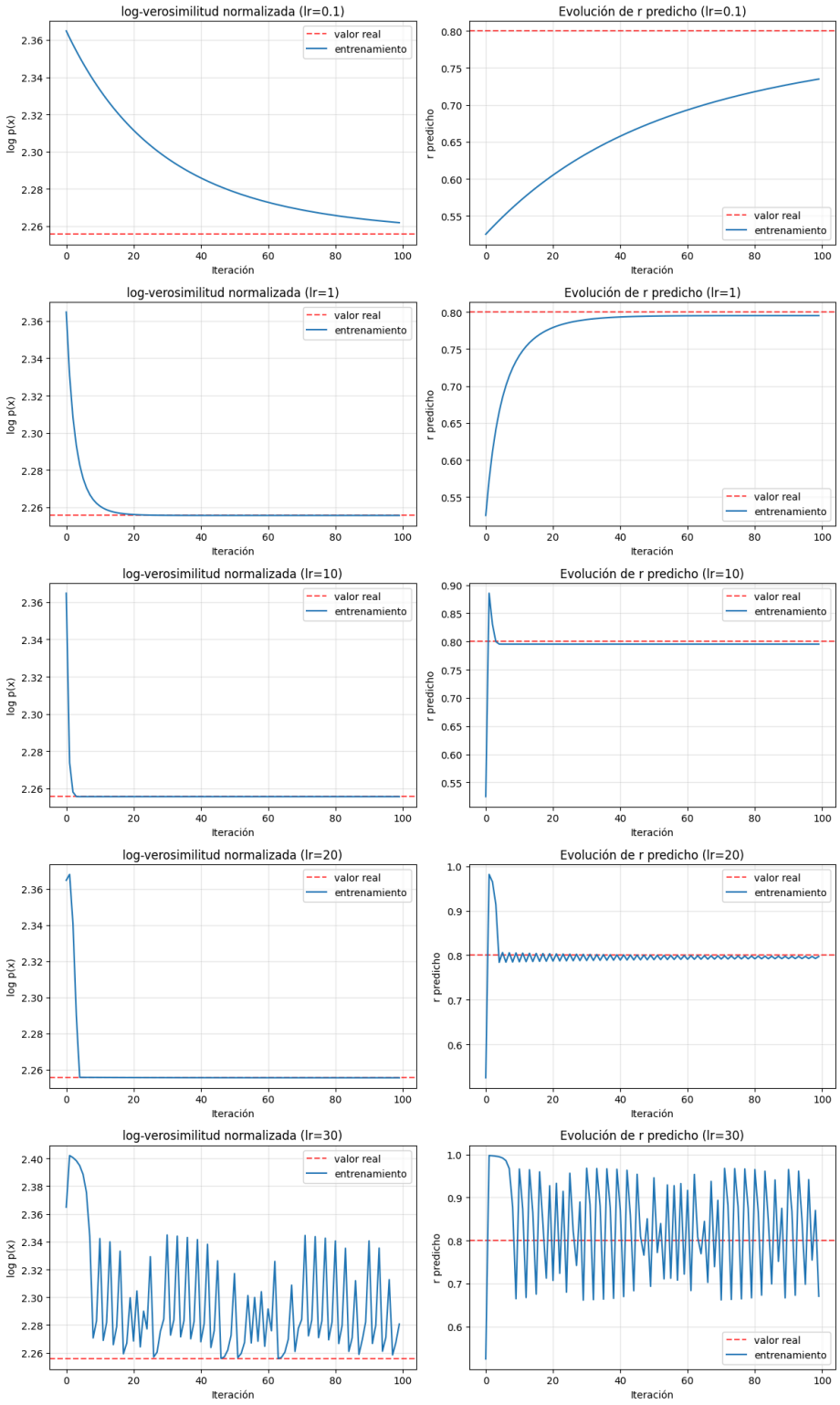

Figura 40: Dinámica de entrenamiento de r_θ para distintos valores de la tasa de aprendizaje: evolución de la log-verosimilitud negativa normalizada (columna izquierda) y del parámetro predicho (columna derecha).

Se observa que la dinámica de entrenamiento varía considerablemente dependiendo del valor de la tasa de aprendizaje. Una tasa de aprendizaje pequeña puede provocar un entrenamiento lento, mientras que una tasa de aprendizaje muy alta puede provocar una dinámica de entrenamiento muy errática al estar variando demasiado los parámetros en cada iteración. Esto se observa en los últimos gráficos de la columna derecha, donde el parámetro $r_\theta \in [0, 1]$ oscila con alta amplitud alrededor del valor real.

Este ejemplo muestra que incluso los modelos generativos de juguete (como este, que es de un solo parámetro) pueden tener una dinámica de entrenamiento inestable si no se ajustan bien los hiperparámetros. En modelos más grandes, este ajuste puede ser aún más delicado, por lo que es usual usar heurísticas para ajustar su valor.

Parte II

LLMs y agentes

Capítulo 3

Modelos de lenguaje y arquitectura GPT

En este capítulo revisaremos las bases de los modelos de lenguaje actuales. Para esto, comenzaremos revisando la formulación autorregresiva de una secuencia de texto y concluiremos con la implementación de un modelo GPT [47] de juguete en español. El foco estará puesto en realizar implementaciones limpias y sencillas, al mismo tiempo que se formula cada elemento de manera precisa, lo cual muchas veces es pasado por alto en la literatura de este tipo de modelos.

Las librerías que se utilizarán son las siguientes:

```
import torch
from torch import nn, optim
from torch.utils.data import Dataset, DataLoader
from dataclasses import dataclass
import matplotlib.pyplot as plt
import tqdm
import re

DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

3.1 Formulación de un modelo de lenguaje

3.1.1 Tokenización

Una secuencia de tokens es un conjunto ordenado finito, $S = (w_1, ..., w_T)$, donde los elementos $w_1, ..., w_T$ son símbolos (e.g., letras o palabras), denominados **tokens**, que pertenecen a un vocabulario común, $\mathcal{V}_0 = \{a_1, ..., a_K\}$ (e.g., letras del abecedario o palabras de un idioma). Por lo general, el vocabulario base $\mathcal{V}_0$ suele ser extendido a otro vocabulario $\mathcal{V} \supset \mathcal{V}_0$ para incluir algunos **tokens especiales** que son utilizados durante el entrenamiento e inferencia de un modelo generativo para texto. En este capítulo, se considerarán los siguientes tokens especiales:

- **Begin of sentence ($\langle$ BOS $\rangle$):** token especial agregado al comienzo de cada secuencia. Se utilizará como token inicial durante la generación.
- **End of sentence ($\langle$ EOS $\rangle$):** token especial agregado al final de cada secuencia. Se utilizará para saber cuándo detener la generación de texto.
- **Unknown ($\langle$ UNK $\rangle$):** token usado para representar tokens desconocidos (fuera de $\mathcal{V}_0$). Permitirá procesar símbolos desconocidos durante el entrenamiento e inferencia.
- **Padding ($\langle$ PAD $\rangle$):** token agregado al final de algunas secuencias para forzar que tengan un largo predefinido. Permitirá realizar el entrenamiento en batches de secuencias.

Por lo tanto, el vocabulario extendido que se utilizará en este capítulo es el siguiente:

$$\mathcal{V} = \left\{ a_1, ..., a_K, \underbrace{\langle \text{BOS} \rangle}_{a_{K+1}}, \underbrace{\langle \text{EOS} \rangle}_{a_{K+2}}, \underbrace{\langle \text{UNK} \rangle}_{a_{K+3}}, \underbrace{\langle \text{PAD} \rangle}_{a_{K+4}} \right\}.$$

Además, se denotará como $\mathcal{V}^*$ al conjunto de todas las secuencias finitas de tokens formadas por elementos del vocabulario $\mathcal{V}$, es decir:

$$\begin{aligned} \mathcal{V}^* &:= \{S = (w_1, ..., w_T) : w_1, ..., w_T \in \mathcal{V}, T \in \mathbb{N}\} \\ &= \bigcup_{T \in \mathbb{N}} \mathcal{V}^T, \end{aligned}$$

donde la notación $\mathcal{V}^T := \underbrace{\mathcal{V} \times \cdots \times \mathcal{V}}_{T \text{ veces}}$ representa el conjunto de secuencias de largo T (i.e., $\mathcal{V}^T = \{S = (w_1, ..., w_T) : w_1, ..., w_T \in \mathcal{V}\}$).

La siguiente clase `Tokenizer` define un vocabulario extendido, `vocabulary`, y utiliza el método `encode` para mapear un string a su respectiva secuencia de tokens (más precisamente, a su secuencia de índices de tokens según el orden del vocabulario). De forma inversa, el método `decode` transforma una lista de índices en el string de símbolos:

```
class Tokenizer:

    def __init__(self, corpus_vocabulary: list[str]) -> None:
        self.vocabulary = corpus_vocabulary + ['<BOS>', '<EOS>', '<UNK>',
'<PAD>']
        self.vocab_size = len(self.vocabulary)

        self.token_to_id = {token: id for id, token in
enumerate(self.vocabulary)}
        self.bos_id = self.token_to_id['<BOS>']
        self.eos_id = self.token_to_id['<EOS>']
```

```
        self.unk_id = self.token_to_id['<UNK>']
        self.pad_id = self.token_to_id['<PAD>']

    def encode(self, text: str) -> list[int]:
        seq_tokens = re.findall(r'\d|[^\w\s]|\w+|\s', text)
        seq_ids = [self.token_to_id.get(token, self.unk_id) for token in
seq_tokens]
        return seq_ids

    def decode(self, seq_ids: list[int]) -> str:
        seq_tokens = ''.join(self.vocabulary[i] for i in seq_ids)
        return seq_tokens
```

- Ejemplo de uso.

```
vocabulary = ['a', 'b', 'c', 'd', ' ']
tokenizer = Tokenizer(vocabulary)

print(f'<BOS> ID: {tokenizer.bos_id}.')
print(f'<EOS> ID: {tokenizer.eos_id}.')
print(f'<UNK> ID: {tokenizer.unk_id}.')
print(f'<PAD> ID: {tokenizer.pad_id}.')

token_seq = 'a b c d e'
id_seq = tokenizer.encode(token_seq)
print(f'encode("{token_seq}") = {id_seq}.')
print(f'decode({id_seq}) = "{tokenizer.decode(id_seq)}".')
```

```
<BOS> ID: 5. <EOS> ID: 6. <UNK> ID: 7. <PAD> ID: 8. encode(«a
b c d e») = [0, 4, 1, 4, 2, 4, 3, 4, 7]. decode([0, 4, 1, 4,
2, 4, 3, 4, 7]) = «a b c d <UNK>».
```

Para entrenar un modelo de lenguaje, es necesario contar con un conjunto de entrenamiento $\mathcal{D} = \{S^1, ..., S^N\} \subset \mathcal{V}^*$ formado por secuencias de tokens definidas sobre un mismo vocabulario $\mathcal{V}$. Además, para poder trabajar con batches de secuencias, asumiremos que todas las secuencias de entrenamiento tienen un mismo largo $T \in \mathbb{N}$:

- Las secuencias de largo mayor a T serán truncadas hasta el largo máximo T.
- Las secuencias de largo menor a T serán extendidas agregando el token especial $\langle \text{PAD} \rangle$ tantas veces como sea necesario para alcanzar el largo T.

En las implementaciones, el parámetro T será representado por la el atributo `seq_length`.

El vocabulario $\mathcal{V}_0$ consistirá en todas las palabras distintas que se encuentran en el corpus contenido en el archivo `data.txt`, el cual consiste en un conjunto de cuentos en español, separados por un salto de línea. En consecuencia, cada secuencia del conjunto de entrenamiento, $S = (w_1, ..., w_T) \in \mathcal{D}$, consistirá en un cuento obtenido desde el corpus (truncado o extendido hasta un largo $T \in \mathbb{N}$), con los tokens especiales agregados donde corresponda. El dataset original fue descargado desde el repositorio karen-pal/borges y se preprocesó para eliminar todos los tildes y caracteres poco frecuentes. La siguiente clase `TextDataset` implementa esta lógica:

```
class TextDataset(Dataset):

    def __init__(self, filename: str, seq_length: int) -> None:

        with open(filename, 'r', encoding='utf-8') as file:
            corpus = file.read()

        corpus_vocabulary = sorted(set(re.findall(r'\d|[^\w\s]|\w+|\s',
corpus)))
        self.tokenizer = Tokenizer(corpus_vocabulary)

        sentences = [sentence.strip() for sentence in corpus.split('\n') if
sentence.strip()]
        self.data = [[self.tokenizer.bos_id] +
self.tokenizer.encode(sentence) + [self.tokenizer.eos_id] for sentence in
sentences]

        self.seq_length = seq_length

    def __len__(self) -> int:
        return len(self.data)

    def __getitem__(self, n: int) -> torch.Tensor:

        seq_ids = self.data[n][:]

        if len(seq_ids) > self.seq_length:
            seq_ids = seq_ids[:self.seq_length]
        else:
            seq_ids += [self.tokenizer.pad_id] * (self.seq_length -
len(seq_ids))

        return torch.tensor(seq_ids)
```

Con esta clase se puede inicializar un dataset para el texto contenido en el archivo `data.txt`:

```
seq_length, batch_size = 256, 32

dataset = TextDataset('data.txt', seq_length)
dataloader = DataLoader(dataset, batch_size, shuffle=True, drop_last=True)

print(f'Tamaño del dataset: {len(dataset)} secuencias.')
print(f'Tamaño del dataloader: {len(dataloader)} batches.')
print(f'Tamaño del vocabulario: {dataset.tokenizer.vocab_size} tokens.')
```

```
Tamaño del dataset: 719 secuencias. Tamaño del dataloader: 22
batches. Tamaño del vocabulario: 60198 tokens.
```

Un ejemplo de secuencia contenida en este dataset es el siguiente (solo se mostrarán los primeros 40 tokens por espacio):

```
seq_ids = dataset[0]
seq_tokens = dataset.tokenizer.decode(seq_ids[:40].tolist())
print(seq_tokens)
```

```
Los enanos tienen una especie de sexto sentido que les permite
reconocerse a primera vista.
```

3.1.2 Formulación autorregresiva

Dada una secuencia de tokens, $S = (w_1, ..., w_T) \in \mathcal{V}^*$, un modelo autorregresivo $p_\theta(S)$ descompone la probabilidad que le asigna a la secuencia S de forma causal, expresando la probabilidad conjunta $p_\theta(S) = p_\theta(w_1, ..., w_T)$ como el producto de las probabilidades de cada token de S dados los tokens que lo anteceden:

$$p_\theta(w_1, ..., w_T) = p_\theta(w_1) \prod_{t=1}^{T-1} p_\theta(w_{t+1} \,|\, w_1, ..., w_t),$$

donde $p_\theta(w_1)$ es la distribución que el modelo le asigna al primer token de S, mientras que $p_\theta(w_{t+1} \,|\, w_1, ..., w_t)$ es la distribución del $(t+1)$-ésimo token de S dados los t tokens anteriores. Además, dado que el primer token siempre es $\langle \text{BOS} \rangle \in \mathcal{V}$, se puede fijar $p_\theta(w_1) = p(w_1) = \delta_{\langle \text{BOS} \rangle}(w_1)$, por lo que solo es necesario diseñar una red neuronal que

aprenda (los parámetros de) la distribución condicional $p_\theta(w_{t+1} \,|\, w_1, ..., w_t)$, la cual será utilizada posteriormente para generar nuevas secuencias de tokens. Notar que, a priori, la descomposición causal no realiza ninguna suposición de independencia sobre los tokens que componen la secuencia S (como sí ocurre, por ejemplo, en las cadenas de Markov), aunque sí impone que el orden topológico del DAG asociado a esta red bayesiana sea el orden temporal de las variables.

Por otra parte, dado que cada distribución condicional $p_\theta(w_{t+1} \,|\, w_1, ..., w_t)$ es una distribución de probabilidades sobre el vocabulario $\mathcal{V}$ (el cuwal es finito), se puede considerar, sin pérdida de generalidad, que $p_\theta(w_{t+1} \,|\, w_1, ..., w_t) \sim \text{Categorical}(\mathcal{V}; \lambda_\theta(w_1, ..., w_t))$ es una distribución categórica sobre $\mathcal{V}$ con vector de probabilidades $\lambda_\theta(w_1, ..., w_t) \in \Delta^{|\mathcal{V}|}$, es decir:

$$p_\theta(w_{t+1} = a_k \,|\, w_1, ..., w_t) = \lambda_\theta(w_1, ..., w_t)_k \in [0, 1], \quad \text{para todo} \quad k \in \{1, ..., |\mathcal{V}|\}.$$

En consecuencia, para entrenar un modelo de lenguaje autorregresivo solo hace falta entrenar un modelo condicional que aprenda el vector de probabilidad $\lambda_\theta(w_1, ..., w_t) \in \Delta^{|\mathcal{V}|}$, los cuales son parametrizados mediante una red neuronal $\lambda_\theta : \mathcal{V}^* \to [0, 1]^{|\mathcal{V}|}$, la cual recibe una secuencia de tokens $(w_1, ..., w_t) \in \mathcal{V}^*$ (llamada **contexto**) y entrega una distribución sobre $\mathcal{V}$ asociada al próximo token en el tiempo $t + 1$. Por otro lado, dado que $\lambda_\theta(w_1, ..., w_t) \in \Delta^{|\mathcal{V}|}$ es un vector de probabilidades, se debe que cumplir que $\sum_{k=1}^{|\mathcal{V}|} \lambda_\theta(w_1, ..., w_t)_k = 1$. Esto último se consigue aplicando la función softmax al final de la red neuronal.

3.2 Inferencia usando una RNN

3.2.1 Vectores de embedding y RNN

Dado que $\mathcal{V}$ es un conjunto abstracto sin ninguna estructura matemática definida, para implementar un modelo de lenguaje utilizando redes neuronales, es necesario obtener una representación continua para cada uno de los tokens del vocabulario discreto $\mathcal{V}$ ya que, de lo contrario, conceptos como derivada o gradiente no estarán bien definidos (al menos en su forma usual). Para esto, es usual asociar cada token $a \in \mathcal{V}$ con un vector $\text{emb}(a) \in \mathbb{R}^D$, denominado **vector de embedding**. Más precisamente, el operador de embedding, $\text{emb} : \mathcal{V} \to \mathbb{R}^D$, mapea el k-ésimo token del vocabulario $\mathcal{V}$ a la k-ésima fila de una matriz $E \in \mathcal{M}_{|\mathcal{V}|,D}(\mathbb{R})$, denominada **matriz de embedding**, la cual es aprendida conjuntamente con el resto de los parámetros del modelo neuronal durante el entrenamiento. Notar que el operador de embedding asume que el vocabulario $\mathcal{V}$ tiene un orden fijo (i.e., se debería escribir $\mathcal{V} = \left(a_1, ..., a_{|\mathcal{V}|}\right)$ en vez de $\mathcal{V} = \left\{a_1, ..., a_{|\mathcal{V}|}\right\}$, pero se seguirá usando la notación de conjunto), lo cual se cumple en la clase `Tokenizer`, donde cada símbolo tiene asociado un índice, los cuales se almacenan en el atributo `token_to_id`. En PyTorch, se puede definir una capa de embedding utilizando la clase `nn.Embedding`, la cual implementa exactamente

esta lógica. La dimensión de embedding $D \in \mathbb{N}$ será denotada como `embedding_dim`, aunque también se le suele llamar **hidden dim**.

- Ejemplo de uso.

```
vocab_size, embedding_dim = dataset.tokenizer.vocab_size, 384
emb = nn.Embedding(vocab_size, embedding_dim)

x = torch.randint(vocab_size, size=[batch_size, seq_length])
y = emb(x)

assert y.shape == (batch_size, seq_length, embedding_dim)
assert emb.weight.shape == (vocab_size, embedding_dim)
```

Por otro lado, aplicar el operador de embedding al k-ésimo token del vocabulario, $a_k \in \mathcal{V}$, es equivalente al realizar el producto matricial $E^\top y_k$, donde $y_k \in \mathbb{R}^{|\mathcal{V}|}$ es el vector one-hot asociado al token a_k (i.e., es el k-ésimo vector de la base canónica de $\mathbb{R}^{|\mathcal{V}|}$). De esta forma, una capa de embedding puede verse como una capa lineal cuya entrada, $a_k \in \mathcal{V}$, es codificada mediante el vector $y_k \in \mathbb{R}^{|\mathcal{V}|}$.

Con respecto a la elección de la arquitectura neuronal a utilizar, es importante notar que el contexto $(w_1, ..., w_t) \in \mathcal{V}^*$ (entrada a la red neuronal) es de largo variable, por lo que no es posible utilizar una red fully connected estándar para parametrizar la distribución condicional $p_\theta(w_{t+1} \mid w_1, ..., w_t)$ (aunque sí se podría utilizar una red convolucional 1D), por lo que hay que usar otro tipo de arquitectura. Una opción frecuente (al menos hasta antes de la aparición de la arquitectura Transformer) es utilizar una red neuronal recurrente (RNN), la cual procesa el contexto $(w_1, ..., w_t) \in \mathcal{V}^*$ comenzando con un vector inicial $h_0 = 0 \in \mathbb{R}^D$ (o cualquier otro valor), y luego procesa iterativamente cada uno de los tokens del contexto $(w_1, ..., w_t)$, desde el primero hasta el último:

$$h_s = f_\theta(w_s, h_{s-1}), \quad \text{para todo} \quad 1 \leq s \leq t,$$

donde $f_\theta : \mathcal{V} \times \mathbb{R}^D \to \mathbb{R}^D$ es una red neuronal cuya entrada es un token del vocabulario $\mathcal{V}$ y el vector asociado a la iteración anterior. De este modo, se puede considerar la última iteración, $h_t \in \mathbb{R}^D$, para definir $\lambda_\theta(w_1, ..., w_t) = g_\theta(h_t) \in [0, 1]^{|\mathcal{V}|}$, donde $g_\theta : \mathbb{R}^D \to [0, 1]^{|\mathcal{V}|}$ es una función que transforma el último vector $h_t \in \mathbb{R}^D$ en un vector de probabilidades sobre $\mathcal{V}$ (a los bloques finales que cumplen esta función se les suele llamar **cabezales de lenguaje**).

Un ejemplo simple (que será usado como baseline) es la red de Elman clásica [48], la cual considera

- $f_\theta(w_s, h_{s-1}) = \tanh(A\,\mathrm{emb}(w_s) + B\,h_{s-1} + d)$.
- $g_\theta(h_t) = \mathrm{softmax}(C\,h_t + e)$.

Notar que se abusó de notación en $f_\theta(w_s, h_{s-1})$, ya que la función de activación $\tanh : \mathbb{R} \to [-1, 1]$ es aplicada coordenada a coordenada.

De esta forma, los parámetros entrenables del modelo son la capa de embedding emb (la cual aprende una matriz $E \in \mathcal{M}_{|\mathcal{V}|,D}(\mathbb{R})$), las matrices $A, B \in \mathcal{M}_{D,D}(\mathbb{R})$, $C \in \mathcal{M}_{|\mathcal{V}|,D}(\mathbb{R})$, y los vectores de sesgo $d \in \mathbb{R}^D$, $e \in \mathbb{R}^{|\mathcal{V}|}$. Para efectos de implementación, no se agregará la capa softmax en la salida del modelo. Es decir, el modelo no retornará directamente un vector de probabilidad, si no que retornará un vector no normalizado ($C\, h_t + e \in \mathbb{R}^{|\mathcal{V}|}$ en este caso) denominado **vector de logits**. Esto permitirá calcular la función de pérdida (log--verosimilitud) de forma más eficiente ya que, como se verá en la sección de entrenamiento, la función logaritmo deshace la normalización realizada por la función softmax.

```
class GenerativeRNN(nn.Module):

    def __init__(self, vocab_size: int, embedding_dim: int) -> None:
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, embedding_dim, batch_first=True)
        self.lm_head = nn.Linear(embedding_dim, vocab_size)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        seq_embedding = self.embedding(x.long())
        rnn_output, _ = self.rnn(seq_embedding)
        logits = self.lm_head(rnn_output)
        return logits
```

- Ejemplo de uso.

```
vocab_size, embedding_dim = dataset.tokenizer.vocab_size, 384
rnn = GenerativeRNN(vocab_size, embedding_dim)

x = torch.randint(vocab_size, size=[batch_size, seq_length])
y = rnn(x)

assert y.shape == (batch_size, seq_length, vocab_size)
```

En este capítulo, se utilizará una red recurrente con $D = 384$:

```
rnn = GenerativeRNN(dataset.tokenizer.vocab_size, embedding_dim=384)

n_params = sum(param.numel() for param in rnn.parameters()) / 1e6
print(f'Cantidad de parámetros: {n_params:.3} millones.')
```

```
Cantidad de parámetros: 46.6 millones.
```

3.2.2 Entrenamiento

Para entrenar una red neuronal $\lambda_\theta : \mathcal{V}^* \to [0,1]^{|\mathcal{V}|}$ que aprenda el vector de probabilidades de la distribución $p_\theta(w_{t+1} \mid w_1, ..., w_t) \sim \text{Categorical}(\mathcal{V}; \lambda_\theta(w_1, ..., w_t))$, el enfoque de optimización natural es el de máxima verosimilitud, el cual consiste en maximizar la cantidad $\log p_\theta(\mathcal{D})$ definida sobre el conjunto de entrenamiento i.i.d. $\mathcal{D} = \{S^1, ..., S^N\} \subset \mathcal{V}^*$. Sin embargo, considerando que $\log p_\theta(\mathcal{D}) \in (-\infty, 0]$, es más usual formular este problema de manera equivalente como la minimización de $-\log p_\theta(\mathcal{D}) \geq 0$, es decir, la red neuronal es optimizada optimizando

$$\min_\theta - \sum_{S \in \mathcal{D}} \log p_\theta(S).$$

De acuerdo a la factorización causal que realizan los modelos autorregresivos, para una secuencia $S = (w_1, ..., w_T) \in \mathcal{V}^*$, su log-verosimilitud se puede escribir como la suma de las log-probabilidades que el modelo le asigna a cada token de la secuencia, dados los tokens anteriores como contexto:

$$\begin{aligned}\log p_\theta(S) &= \log \left(p_\theta(w_1) \prod_{t=1}^{T-1} p_\theta(w_{t+1} \mid w_1, ..., w_t) \right) \\ &= \cancel{\log p(w_1)} + \sum_{t=1}^{T-1} \log p_\theta(w_{t+1} \mid w_1, ..., w_t),\end{aligned}$$

donde en la última igualdad, $\log p(w_1) = 0$ ya que $p(w_1) = \delta_{\langle \text{BOS} \rangle}(w_1) = 1$ (todas las secuencias en $\mathcal{D}$ comienzan con $w_1 = \langle \text{BOS} \rangle$). Sin embargo, una práctica usual es optimizar la log-verosimilitud normalizada por dos cantidades que influyen directamente en la función de costo $-\log p_\theta(\mathcal{D})$:

- **Por el tamaño del batch $\mathcal{D}$:** esto permite que la cantidad $\log p_\theta(\mathcal{D})$ sea independiente del tamaño del batch, por lo que no hay que ajustar la tasa de aprendizaje según esta cantidad.
- **Por el largo de cada secuencia $S \in \mathcal{D}$:** esto cambia levemente el problema para optimizar la log-verosimilitud promedio por token (en vez de la log-verosimilitud de las secuencias), lo cual es importante ya que $\log p_\theta(S^1) \geq \log p_\theta(S^2)$ si $|S^1| \leq |S^2|$ (i.e., sin normalizar, el modelo le da mayor verosimilitud a secuencias cortas).

En resumen, si se cuenta con un dataset de entrenamiento $\mathcal{D} = \{S^1, ..., S^N\} \subset \mathcal{V}^*$, donde cada secuencia $S^n = \left(w_1^{(n)}, ..., w_{T_n}^{(n)}\right) \in \mathcal{D}$ es de largo $T_n \in \mathbb{N}$, el problema de optimización estándar para el entrenamiento de un LLM es el siguiente:

$$\min_{\theta} -\frac{1}{N}\sum_{n=1}^{N}\frac{1}{T_n}\sum_{t=1}^{T_n-1}\log p_\theta\left(w_{t+1}^{(n)} \,\middle|\, w_1^{(n)},...,w_t^{(n)}\right).$$

Si bien en esta implementación todos los batches tendrán el mismo tamaño y todas las secuencias tendrán el mismo largo (gracias al truncamiento y padding), en entrenamientos más eficientes no siempre es así. Por ejemplo, para ahorrar computación y memoria durante el entrenamiento, se pueden formar batches de secuencias de largos similares para minimizar el uso de truncamiento y padding. Del mismo modo, los batches también pueden ser de diferente tamaño, sobretodo cuando se indica `drop_last=False` al instanciar el DataLoader, donde el último batch puede ser de un tamaño menor que el resto debido a la falta de más muestras en el dataset.

Por otra parte, es importante notar que, para una secuencia $S = (w_1, ..., w_T) \in \mathcal{D}$, esta función de pérdida no necesita conocer todo el vector de probabilidades $\lambda_\theta(w_1, ..., w_t) \in \Delta^{|\mathcal{V}|}$ asociado a la distribución $p_\theta(w_{t+1} \,|\, w_1, ..., w_t)$, si no que es suficiente conocer su valor en la coordenada correspondiente al token $w_{t+1} \in \mathcal{V}$ correcto. En consecuencia, es suficiente implementar el cálculo del logaritmo y softmax del vector de logits solo para la coordenada que se necesita (y no para toda la salida de la red neuronal), lo cual resulta computacionalmente más eficiente.

La clase `nn.CrossEntropyLoss` de PyTorch permite calcular eficientemente esta función objetivo cuando la red neuronal está implementada para retornar el vector de logits (i.e., cuando no se aplica la función softmax en la salida de la red neuronal). En efecto, si `logits` representa la salida de la red neuronal para un batch de secuencias dado como input y `targets` es un tensor con los índices de los tokens que debería predecir el modelo, el valor obtenido con `nn.CrossEntropyLoss` equivale a la función de costo anterior:

```
batch_size, seq_length, vocab_size = 32, 256, dataset.tokenizer.vocab_size

logits = torch.randn(batch_size, seq_length, vocab_size)
targets = torch.randint(0, vocab_size, size=(batch_size, seq_length))

# Cálculo de la función de pérdida paso a paso:
probs = logits.softmax(dim=-1)
target_probs = probs.gather(dim=-1,
index=targets.unsqueeze(-1)).squeeze(-1)

log_likelihood = target_probs.log().sum(dim=-1)
normalized_log_likelihood = log_likelihood / seq_length
manual_loss = - normalized_log_likelihood.mean()

# Cálculo de la función de pérdida con CrossEntropyLoss:
```

```
loss_fn = nn.CrossEntropyLoss()
logits_reshaped = logits.view(batch_size * seq_length, vocab_size)
targets_reshaped = targets.view(batch_size * seq_length)
direct_loss = loss_fn(logits_reshaped, targets_reshaped)

assert torch.isclose(manual_loss, direct_loss)
```

Un último detalle importante al momento de optimizar este tipo de modelos es con respecto a los tokens de padding, $\langle \text{PAD} \rangle \in \mathcal{V}$. Dado que este token solo se incluye en las secuencias para extenderlas hasta un cierto largo predefinido $T \in \mathbb{N}$, su predicción no debe incluirse en el cálculo de la función de pérdida ya que no forman parte de la secuencia original, ni se busca aprender la dinámica de extender las secuencias con tokens de padding luego de generar el token final, $\langle \text{EOS} \rangle \in \mathcal{V}$. Esta omisión se consigue incluyendo el argumento `ignore_index=pad_id` al instanciar la clase `nn.CrossEntropyLoss`.

Con todo esto, el loop de entrenamiento de un modelo de lenguaje es el siguiente:

```
def train_model(
        model: nn.Module,
        optimizer: optim.Optimizer,
        dataloader: DataLoader,
        epochs: int,
        ckpt_filename: str) -> None:

    model.to(DEVICE)
    model.train()

    pad_id = dataloader.dataset.tokenizer.pad_id
    loss_fn = nn.CrossEntropyLoss(ignore_index=pad_id)

    training = {'losses': [], 'model': None}

    try:
        progressbar = tqdm.trange(epochs)
        for epoch in progressbar:

            for seq_batch in dataloader:

                seq_batch = seq_batch.to(DEVICE)
                x_batch, y_batch = seq_batch[:, :-1], seq_batch[:, 1:]

                logits = model(x_batch)

                batch_size, seq_length, vocab_size = logits.shape
```

```
                logits = logits.reshape(batch_size * seq_length,
vocab_size)
                y_batch = y_batch.reshape(batch_size * seq_length)

                loss = loss_fn(logits, y_batch)

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                training['losses'].append(loss.item())
                progressbar.set_postfix(loss=loss.item())

    except KeyboardInterrupt:
        print('Entrenamiento interrumpido.')

    training['model'] = model.state_dict()
    torch.save(training, ckpt_filename)
```

Se entrenará la RNN definida anteriormente durante 32 épocas utilizando AdamW [49], el cual es un optimizador usual para LLMs (aunque hoy en día también se utilizan otras opciones más modernas):

```
rnn_optimizer = optim.AdamW(rnn.parameters())
train_model(rnn, rnn_optimizer, dataloader, epochs=32,
ckpt_filename='rnn_training.pt')
```

El entrenamiento toma aproximadamente 5 minutos en Google Colab (usando una GPU NVIDIA Tesla T4). Una vez concluido el entrenamiento, se puede cargar el modelo y ver la evolución de la función de costo a lo largo del entrenamiento:

```
rnn_training = torch.load('rnn_training.pt', DEVICE, weights_only=True)
rnn.load_state_dict(rnn_training['model'])

plt.figure(figsize=(10, 5))
plt.plot(rnn_training['losses'])
plt.xlabel('Iteración')
plt.ylabel('Entropía cruzada media')
plt.grid(alpha=0.3)
plt.show()
```

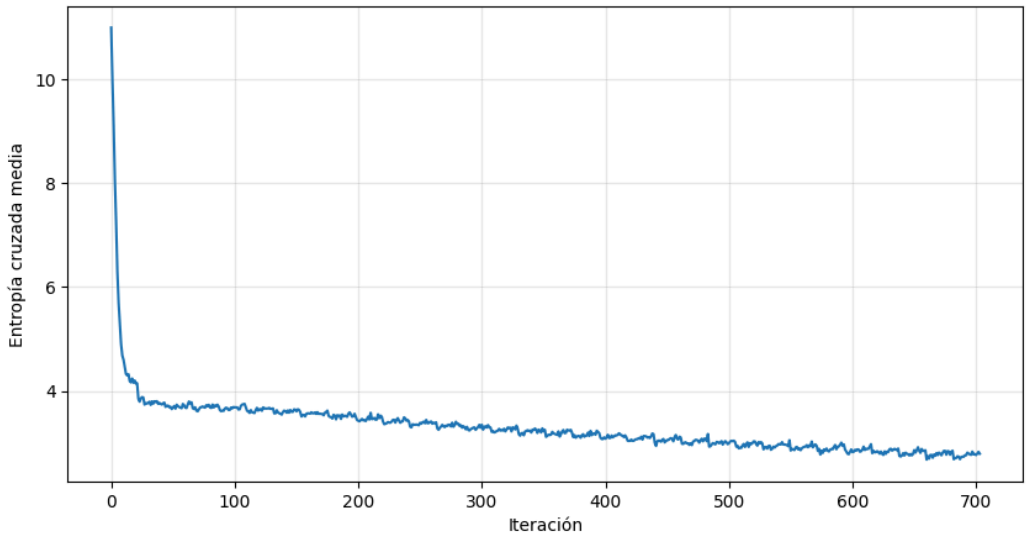

Figura 41: Evolución de la entropía cruzada durante el entrenamiento de la RNN generativa.

Notar que, en un comienzo, la entropía cruzada es cercana a $\log(|\mathcal{V}|) \approx 11$ nats, que es lo máximo que puede valer cuando el modelo asigna probabilidades uniformes a cada token.

3.2.3 Generación

Para la generación de nuevas secuencias de tokens, se puede utilizar el algoritmo de ancestral sampling revisado en la entrada de redes bayesianas. En el caso del enfoque autorregresivo, la generación comienza con el token inicial $w_1 = \langle \text{BOS} \rangle$ (recordar que se fijó $p(w_1) = \delta_{\langle \text{BOS} \rangle}(w_1)$) y luego se debe evaluar iterativamente el modelo neuronal entrenado para predecir el siguiente token, utilizando como contexto la secuencia de tokens que ya han sido generados. La generación de nuevos tokens continúa hasta que se genera el token $\langle \text{EOS} \rangle$ o bien, hasta que se genera una cantidad máxima de tokens. Del mismo modo, se podría realizar generación condicional comenzando con una secuencia de tokens iniciales (llamada **prompt**), $S_{\text{prompt}} = (w_1, ..., w_t)$, lo cual es una tarea usual en aplicaciones como chatbots o agentes.

Dado que el modelo neuronal retorna (los logits de) la distribución $p_\theta(w_{t+1} \,|\, w_1, ..., w_t) \sim \text{Categorical}(\mathcal{V}; \lambda_\theta(w_1, ..., w_t))$ asociada al siguiente token dado el contexto actual, se debe elegir un criterio para decidir cuál va a ser el próximo token a partir de la distribución entregada por el modelo. Si $S = (w_1, ..., w_t) \in \mathcal{V}^*$ son los tokens de entrada (contexto) y $\lambda_\theta(S) \in \Delta^{|\mathcal{V}|}$ es el vector de probabilidades (luego de aplicar softmax) que entrega el modelo neuronal para el próximo token (i.e., $p_\theta(w_{t+1} = a_k \,|\, S) = \lambda_\theta(S)_k$, para todo $k \in \{1, ..., |\mathcal{V}|\}$), una primera idea podría ser elegir el siguiente token $w_{t+1} \in \mathcal{V}$ como el token más probable según el modelo, es decir:

$$w_{t+1} = a_k, \quad \text{donde} \quad k = \underset{k \in \{1,..., |\mathcal{V}|\}}{\arg\max} \lambda_\theta(S)_k$$

Una limitación de este enfoque, denominado **greedy decoding**, es que la elección del token más probable no es necesariamente una decisión óptima ya que, si bien la probabilidad de este token es alta, puede ocurrir que todas las futuras probabilidades sean bajas producto de haber elegido el token más probable en el momento (de ahí viene el adjetivo greedy). Para permitir más variabilidad en la generación, se puede elegir el siguiente token generando una muestra desde la distribución categórica $\text{Categorical}(\mathcal{V}; \lambda_\theta(S))$ (e.g., utilizando `torch.multinomial`) y definir w_{t+1} como la muestra generada. Esto garantiza que el modelo genere siempre secuencias nuevas, aunque reciba la misma secuencia inicial. Sin embargo, dado que $\lambda_\theta(S)_k > 0$ para todo $k \in \{1, ..., |\mathcal{V}|\}$ (ya que este vector se determina usando softmax y $e^l > 0, \forall l \in \mathbb{R}$), el sampling desde la distribución categórica usando $\lambda_\theta(S)$ puede generar, eventualmente, tokens muy poco probables, lo que muchas veces se traduce en secuencias de texto sin sentido debido a la propagación del error de generación. Una solución inmediata a este problema es permitir elegir solamente los $K \geq 1$ tokens más probables, redistribuyendo la masa de los $|\mathcal{V}| - K$ tokens restantes: si $\mathcal{I} = \{k_1, ..., k_K\} \subset \{1, ..., |\mathcal{V}|\}$ son las K coordenadas donde $\lambda_\theta(S) \in \Delta^{|\mathcal{V}|}$ toma los valores más altos, se puede construir el vector de probabilidad truncado, $\tilde{\lambda}_\theta(S) \in \Delta^{|\mathcal{V}|}$, cuya k-esima coordenada es

$$\tilde{\lambda}_\theta(S)_k = \begin{cases} 0 & \text{si} \quad k \notin \mathcal{I} \\ \lambda_\theta \frac{(S)_k}{\sum_{i \in \mathcal{I}} \lambda_\theta(S)_i} & \text{si} \quad k \in \mathcal{I} \end{cases}.$$

Luego, la técnica de **top-K sampling** elige el siguiente token generando una muestra desde la distribución $\text{Categorical}\big(\mathcal{V}; \tilde{\lambda}_\theta(S)\big)$. Notar que si $K = 1$ se recupera la generación greedy, mientras que si $K = |\mathcal{V}|$ se recupera el sampling categórico usando el vector de probabilidades original, $\lambda_\theta(S) \in \Delta^{|\mathcal{V}|}$. Por otro lado, es importante mencionar que el escalar $K \in \mathbb{N}$ definido aquí no tiene relación con el escalar $K \in \mathbb{N}$ definido al comienzo del capítulo (usado para denotar la cardinalidad del vocabulario base, $\mathcal{V}_0 = \{a_1, ..., a_K\}$), solo es un alcance de nombre.

Otra heurística comúnmente utilizada corresponde a aplicar la idea de temperatura, la cual modifica el vector de probabilidades $\lambda_\theta(S)$ para volverlo más determinista (baja temperatura) o más uniforme (alta temperatura). Más precisamente, si $l \in \mathbb{R}^{|\mathcal{V}|}$ corresponde al vector de logits entregado por la red neuronal (i.e., el vector previo a aplicar softmax) y $T > 0$ es un escalar, denominado **temperatura**, se puede considerar una versión escalada del vector de logits, $\tilde{l} = \frac{l}{T} \in \mathbb{R}^{|\mathcal{V}|}$, y aplicar softmax a esta nueva cantidad para obtener el vector de probabilidades desde el que se realiza el sampling del próximo token (ya sea usando todo el vector o usando top-K sampling). El objetivo de esta ponderación es llevar todos los logits a una escala similar (si $T > 1$) o destacar aún más el logit del token más

probable (si $T < 1$). Notar que el escalar de temperatura $T \in \mathbb{R}_{++}$ definido aquí no tiene relación con el escalar $T \in \mathbb{N}$ usado para definir el largo de las secuencias, solo es un alcance de nombre.

En la siguiente figura se observa cómo cambia el nuevo vector de probabilidades, $\text{softmax}(\tilde{l}) \in \Delta^{|\mathcal{V}|}$, para distintos valores de temperatura, donde $T = 1$ corresponde al vector de probabilidades $\lambda_\theta(S)$ que se obtendría sin aplicar esta técnica:

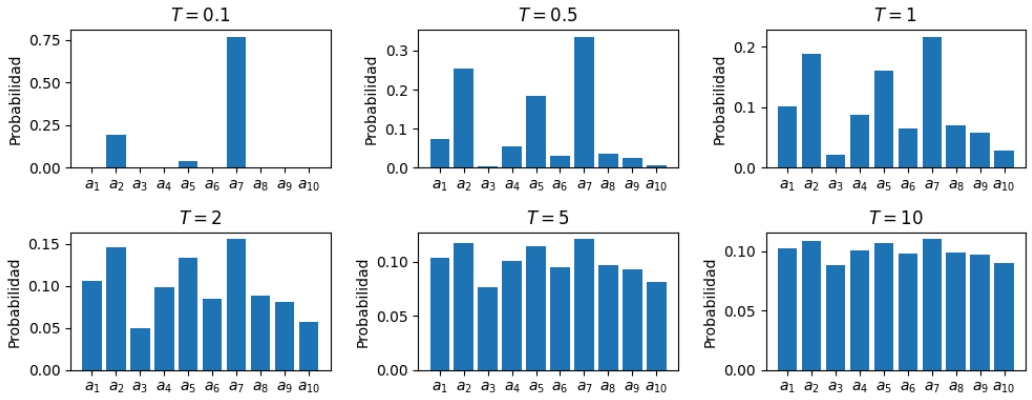

Figura 42: Vector de probabilidades obtenido al aplicar softmax a un vector de logits reescalado por distintos valores de temperatura.

- Código para generar la figura.

```
vocab_size = 10
vocabulary = [f'$a_{{{k}}}$' for k in range(1, vocab_size + 1)]
logits = torch.randn(vocab_size)
temperatures = [0.1, 0.5, 1, 2, 5, 10]

fig, axes = plt.subplots(2, 3, figsize=(10, 4))
axes = axes.flatten()

for i, t in enumerate(temperatures):
    scaled_logits = logits / t
    prob = scaled_logits.softmax(-1)
    axes[i].bar(vocabulary, prob)
    axes[i].set_title(f'$T={t}$')
    axes[i].set_ylabel(f'Probabilidad')

plt.tight_layout()
plt.show()
```

Notar que si $T \to 0$ se recupera la generación greedy, mientras que si $T \to \infty$ el vector de probabilidades se vuelve uniforme (con una probabilidad $\frac{1}{|\mathcal{V}|}$ para cada token). Por lo

tanto, el parámetro de temperatura $T > 0$ permite controlar el trade-off entre versatilidad y confianza en la generación, lo cual es una herramienta muy útil en algunos contextos. Por ejemplo, para la generación de código es usual elegir una baja temperatura, mientras que para la generación de textos narrativos, puede ser preferible utilizar una temperatura más elevada.

Por último, otra heurística usual es penalizar la probabilidad de tokens que ya han sido generados o que estén en la secuencia inicial. Esta técnica se aplica debido a que es común que los modelos de lenguaje basados en redes neuronales entren en ciclos de repetición de un mismo token o secuencias de tokens. Si $c(a_k) \in \mathbb{N}$ es la cantidad de veces que el símbolo $a_k \in \mathcal{V}$ ha aparecido en la secuencia generada hasta el momento, se pueden escalar los logits $l \in \mathbb{R}^{|\mathcal{V}|}$ mediante

$$\tilde{l}_k = \begin{cases} \frac{1}{\gamma^{c(a_k)}} \cdot l_k & \text{si} \quad l_k > 0 \\ \gamma^{c(a_k)} \cdot l_k & \text{si} \quad l_k \leq 0 \end{cases},$$

donde el hiperparámetro $\gamma \geq 1$ controla el nivel de penalización. Más específicamente, si $\gamma > 1$, entonces los logits de tokens repetidos decrecen, reduciendo su probabilidad tras aplicar softmax.

Todas estas técnicas pueden ser incluidas dentro de un mismo loop de generación:

```
def generate_tokens(
        model: nn.Module,
        context: str,
        tokenizer: Tokenizer,
        temperature: float = 0.8,
        top_k: int = 50,
        max_tokens: int = 256,
        repetition_penalty: float = 1.01) -> str:

    model.to(DEVICE)
    model.eval()

    seq_id = [tokenizer.bos_id] + tokenizer.encode(context)
    seq_id = torch.tensor(seq_id, device=DEVICE)

    with torch.no_grad():
        for _ in range(max_tokens):
            logits = model(seq_id.unsqueeze(0))[0, -1, :]

            token_counts = torch.bincount(seq_id, minlength=logits.size(0))
            for token_id, count in enumerate(token_counts):
```

```
                if count > 0:
                    if logits[token_id] > 0:
                        logits[token_id] /= (repetition_penalty ** count)
                    else:
                        logits[token_id] *= (repetition_penalty ** count)

            if temperature == 0:
                next_token = torch.argmax(logits, dim=-1, keepdim=True)
            else:
                logits = logits / temperature

                top_k_logits, top_k_indices = torch.topk(logits, top_k)
                probs = top_k_logits.softmax(dim=-1)
                next_token = top_k_indices[torch.multinomial(probs,
num_samples=1)]

            seq_id = torch.cat((seq_id, next_token), dim=0)

            if next_token in (tokenizer.eos_id, tokenizer.pad_id):
                break

    return tokenizer.decode(seq_id.tolist())
```

Notar que la técnica de penalización por repetición se aplica antes de aplicar la de temperatura. Por otro lado, la implementación realizada no es muy eficiente: hay cálculos que podrían reutilizarse (p.g. usando programación dinámica), y en la parte de penalización por repetición, el loop con `enumerate(token_counts)` itera sobre todo el vocabulario (se podría hacer más eficiente operando solo sobre los tokens que efectivamente aparecen). En la entrada de LLMs y agentes se revisarán algunas heurísticas más avanzadas para un entrenamiento e inferencia más eficiente.

Usando la función anterior y el modelo neuronal ya entrenado, se puede realizar generación condicional continuando una secuencia de texto dada:

```
context = 'Habia una vez'

for n in range(10):
    new_tokens = generate_tokens(rnn, context, dataset.tokenizer)
    print(new_tokens)
```

```
Habia una vez por la luz del otro, de la puerta y de los detalles de
los ojos, y el que no se acerco el caso. Habia una vez que me dijo
```

```
que habia que en los ojos del mundo, y de la hora. Y es de otro.
Habia una vez que como el dia los ojos de la tarde y su vida. Ya son
menos. Pero, se sento en que, que lo pocos de los cuatro. Como el
brazo, de los dedos, el sol, las cosas. En la ultima hora, habia?
Habia una vez una manera de las yerbas que habia sido tan dias, y a
la puerta, y el cuarto. ¿Que dio a el espacio de la mañana. Habia una
vez un rio para el muro y la luz. No puede la mañana, al instante…
¿Es siempre por la costa. Sus noche, se llamaba a la tierra, sin
la pequeña spicula…… Habia una vez que solo como a los dos años,
no se ve sus amigos, es la casa de los dos del sol. Todo el mundo,
despues, no, las pesadas mujer, ¡Era con una mesa,……. Habia una vez
que no se me dijo nunca, de la puerta que en la muerte, como que?
¡No….. Habia una vez ese que no se ha que se encamino a quien estaba
en el instante. Las calles y a la ciudad. Habia una vez por la costa
del hombre del brazo, aunque no hace que su lado que se le habia
veces se trata. Habia una vez su luz del corredor de los hombres
para la puerta, y se no de las manos, pues, la calle. Para entre la
puerta, a la cabeza, que?
```

En la generación anterior se observa que, si bien el modelo fue capaz de aprender a separar cada palabra por un espacio, el texto que genera es incoherente. Si bien es posible obtener mejores resultados cambiando algunos hiperparámetros de generación (o usando una RNN más profunda), en la segunda parte de este capítulo se estudiará la arquitectura GPT, la cual define el esqueleto común de prácticamente todos los LLMs modernos.

Otras heurísticas de generación

Pendiente: beam search, nucleus sampling, min-p sampling, penalización por n-gramas, sampling especulativo, contrastive search.

3.3 Arquitectura GPT

A pesar de que las RNNs permiten procesar secuencias de largo variable, tienen limitaciones importantes que no permiten extenderlas fácilmente a escenarios más complejos como los que enfrentan los LLMs de hoy en día. Por ejemplo, omitiendo algunos componentes por simplicidad, una RNN está compuesta esencialmente por iteraciones de la forma $h_s = A\,\mathrm{emb}(w_s) + B\,h_{s-1}$, donde se observa que la matriz $B \in \mathcal{M}_{D,D}(\mathbb{R})$ es multiplicada por sí misma varias veces, dependiendo del largo de la secuencia que se esté procesando. Esta multiplicación reiterada provoca que el gradiente $\frac{\partial L}{\partial B}$ (con L la función de pérdida) crezca de manera irrestricta (**exploding gradient problem**) o bien, decrezca hasta hacerse

cercano a cero (**vanishing gradient problem**), lo cual dependerá del radio espectral de la matriz B. Ambos problemas dificultan el entrenamiento de este tipo de modelos ya que un gradiente irrestricto provoca un entrenamiento inestable (que puede incluso divergir), mientras que un gradiente nulo no permite actualizar los parámetros de la red neuronal por algoritmos de gradiente. Más aún, el tamaño de estos gradientes no es uniforme a lo largo de las capas de la red neuronal, por lo que una tasa de aprendizaje pequeña (e.g., para aminorar la inestabilidad del exploding gradient) puede ser contraproducente para el desvenecimiento de gradiente observado en las primeras capas.

Si bien se han propuesto algunas modificaciones para mitigar estos problemas (e.g., **gradient clipping** para el exploding gradient problem o arquitecturas con compuertas como GRU o LSTM para el vanishing gradient problem), las RNNs siguen teniendo limitaciones intrínsecas debido a su forma recursiva. En efecto, su procesamiento secuencial ralentiza considerablemente el entrenamiento de este tipo de modelos ya que cada secuencia se debe procesar token a token (i.e., no se pueden procesar todos los tokens en paralelo). Por otro lado, toda la información de la secuencia que se está procesando debe poder almacenarse en un vector $h_t \in \mathbb{R}^D$, lo cual es inviable para secuencias largas, provocando que el modelo empiece a olvidar la información de los primeros tokens de la secuencia a medida que se avanza en la recursion (en particular, las RNNs tienen un sesgo hacia los tokens más recientes). La arquitectura Transformer [14], y en particular los modelos tipo GPT, solucionan (al menos en principio) las limitaciones anteriores:

- Las secuencias $S = (w_1, ..., w_T) \in \mathcal{V}$ se procesan en paralelo, es decir, se pueden procesar todos los tokens de todas las secuencias de un batch de entrenamiento en una sola llamada a la red neuronal. En particular, esto permite escalar el entrenamiento en tamaño del dataset y largo de las secuencias. Sin embargo, la generación sigue siendo secuencial (token a token).
- No se produce el problema de vanishing/exploding gradient gracias a que se insertan capas de normalización y conexiones residuales entre las capas de la red neuronal.
- No se olvida información al procesar la secuencia ya que cada token puede poner atención a la información que contienen todos los otros tokens de la secuencia.

En esencia, la arquitectura GPT consiste en mejorar una secuencia de embeddings iniciales, $(x_1, ..., x_T) \subset \mathbb{R}^D$ (asociados a una secuencia de tokens $S = (w_1, ..., w_T) \in \mathcal{V}^*$ dados como entrada), pasándolos a través de una serie de bloques neuronales, llamados bloques Transformer, los cuales buscan enriquecer cada uno de estos embeddings con información adicional obtenida de los tokens vecinos dentro de la secuencia. De este modo, al pasar por el último bloque Transformer, los embeddings obtenidos son lo suficientemente ricos en información como para poder pasarlos por un cabezal de lenguaje simple (similar al usado en la RNN) para realizar la predicción del próximo token. Cada uno de estos

bloques Transformer, que son el elemento principal de la arquitectura GPT, está formado principalmente por dos componentes:

- **Módulo de self-attention:** en este sub-módulo cada token interactúa con los otros tokens de la secuencia para obtener una representación más acorde al contexto.
- **Módulo feedforward:** red fully connected de dos capas usada para mejorar la representación obtenida en el módulo de auto-atención. Este sub-módulo es aplicado de forma independiente a cada token.

Adicionalmente, la arquitectura GPT tiene otros componentes: al comienzo de la arquitectura se debe incluir una capa de embedding inicial (llamada **token embedding**), a la cual se le suma una capa de embedding posicional (llamada **positional encoding**), la cual permitirá inyectar la información acerca de la posición en la que se encuentra cada token. A la salida del último bloque Transformer, se anexa un cabezal de lenguaje similar al usado en la RNN implementada anteriormente. Adicionalmente, cada uno de los componentes que forman la arquitectura GPT incluye sub-módulos menores como capas de normalización (en este caso, layer normalization) y dropout.

En esta sección se explicará e implementará cada uno estos componentes, intentando dar una motivación natural para cada uno de ellos. La siguiente clase `GPTConfig` contiene todos los hiperparámetros que definen la arquitectura GPT (se utilizarán valores pequeños para poder realizar el entrenamiento en Google Colab):

```
@dataclass
class GPTConfig:
    vocab_size: int = dataset.tokenizer.vocab_size
    context_window: int = 256
    embedding_dim: int = 384
    num_layers: int = 6
    num_heads: int = 6
    head_dim: int = 64
    ff_factor: int = 4
    dropout: float = 0.1

config = GPTConfig()
```

3.3.1 Layer normalization

El primer módulo que se revisará será la capa de normalización, la cual será añadida al comienzo de cada componente de la arquitectura (aunque originalmente se aplicaba al final de cada componente).

Al igual como ocurre en todas las redes neuronales profundas, la aplicación consecutiva de varias capas neuronales suele afectar el entrenamiento del modelo (e.g., por vanishing/exploding gradient). Para mitigar estos problemas, es usual agregar capas de normalización entremedio de los bloques de la red. Por ejemplo, en redes convolucionales es usual utilizar la técnica de batch normalization [50] (o más frecuentemente hoy en día, group normalization [51]), la cual normaliza cada canal de un batch de imágenes para que tenga media nula y varianza unitaria.

Sin embargo, no es obvio cómo aplicar esta técnica en modelos secuenciales ya que, en general, las secuencias de tokens no suelen tener el mismo largo y muchas veces los batches de entrenamiento deben ser pequeños debido a limitaciones de memoria en GPU, lo cual provoca que los estadísticos calculados no sean precisos, perjudicando el entrenamiento.

Por estos motivos, en modelos de lenguaje y en arquitecturas tipo Transformer es usual utilizar otra técnica de normalización más natural para este tipo de modelos, conocida como layer normalization [52], la cual normaliza cada token dentro de una secuencia de forma individual, por lo que su normalización no depende de otras secuencias dentro de un batch de entrenamiento ni de otros tokens de la secuencia que se está procesando. Más precisamente, para un vector de embedding $x \in \mathbb{R}^D$, layer normalization primero calcula su media $\mu \in \mathbb{R}$ y varianza $\sigma^2 \geq 0$ a lo largo de las dimensiones del embedding x:

$$\mu = \frac{1}{D}\sum_{d=1}^{D} x_d \qquad\qquad \sigma^2 = \frac{1}{D}\sum_{d=1}^{D} (x_d - \mu)^2.$$

Luego, normaliza el vector de embedding coordenada a coordenada usando los valores de μ y σ. Es decir, la d-ésima coordenada del embedding normalizado, $\tilde{x} \in \mathbb{R}^D$, viene dada por

$$\tilde{x}_d = \frac{x_d - \mu}{\sqrt{\sigma^2 + \epsilon}},$$

donde $\epsilon > 0$ es un escalar fijo y pequeño utilizado para estabilidad numérica. Finalmente, de forma análoga a lo realizado por batch normalization, layer normalization aplica una transformación afín a cada coordenada del embedding normalizado:

$$\text{LayerNorm}(x) := \alpha \odot \tilde{x} + \beta,$$

donde $\alpha, \beta \in \mathbb{R}^D$ son parámetros entrenables del módulo LayerNorm y $\odot$ es el producto de Hadamard (i.e., $\alpha \odot \tilde{x} \in \mathbb{R}^D$ con $(\alpha \odot \tilde{x})_d = \alpha_d \, \tilde{x}_d$). La motivación detrás de esta última transformación es poder permitirle al modelo deshacer la normalización anterior. El siguiente diagrama resume este proceso:

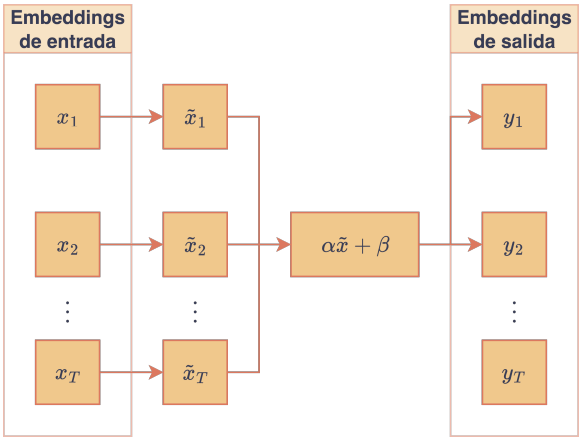

Figura 43: Diagrama del módulo de layer normalization aplicado a un vector de embedding.

La siguiente clase `LayerNorm` implementa esta técnica de normalización:

```
class LayerNorm(nn.Module):

    def __init__(self, emb_dim: int, epsilon: float = 1e-5) -> None:
        super().__init__()

        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))
        self.epsilon = epsilon

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        x_norm = (x - mean) / torch.sqrt(var + self.epsilon)
        return self.scale * x_norm + self.shift
```

- Ejemplo de uso.

```
ln = LayerNorm(config.embedding_dim)
x = torch.randn(batch_size, seq_length, config.embedding_dim)
y = ln(x)

assert y.shape == (batch_size, seq_length, config.embedding_dim)
assert y.mean(dim=-1).allclose(torch.zeros(batch_size, seq_length),
atol=1e-5)
assert y.var(dim=-1).allclose(torch.ones(batch_size, seq_length),
atol=1e-2)
```

Notar que los parámetros α y β son inicializados como el vector $1 \in \mathbb{R}^D$ y el vector $0 \in \mathbb{R}^D$ respectivamente, lo cual garantiza que, en un comienzo, estos parámetros no tengan ningún

efecto posterior a la normalización, lo cual estabiliza el entrenamiento. Por otro lado, la transformación afín realizada luego de la normalización, $\alpha \odot \tilde{x} + \beta$, es aplicada sobre todos tokens de la secuencia (y sobre todas las secuencias de un batch de entrenamiento) usando los mismos parámetros $\alpha, \beta \in \mathbb{R}^D$ ya que esta capa de normalización es local y se aplica de forma individual para cada token.

Es importante mencionar que la implementación realizada en la clase `LayerNorm` tiene el mismo comportamiento que la clase `nn.LayerNorm` de PyTorch cuando se utiliza para procesar secuencias, pero no tiene el mismo comportamiento cuando es utilizada para batches imágenes (donde los tensores suelen ser de orden 4). Sin embargo, como se mencionó anteriormente, en imágenes es usual considerar otras técnicas de normalización como `nn.BatchNorm2d` o `nn.GroupNorm`.

3.3.2 Self attention

El componente clave de la arquitectura GPT corresponde al mecanismo de auto-atención (self-attention), el cual está encargado de mejorar los embeddings de una secuencia utilizando la información contextual de cada token. Más precisamente, dada una secuencia de vectores de embeddings, $(x_1, ..., x_T) \subset \mathbb{R}^D$, asociada a una secuencia de tokens $S = (w_1, ..., w_T) \in \mathcal{V}^*$, el objetivo del mecanismo de auto-atención es obtener otra secuencia de embeddings, $(y_1, ..., y_T) \subset \mathbb{R}^P$, donde cada embedding $y_t \in \mathbb{R}^P$ corresponderá a una versión mejorada del respectivo embedding inicial, $x_t \in \mathbb{R}^D$, la cual incorporará información contextual de todos los otros tokens de la secuencia S. Para esto, el mecanismo de auto-atención comienza proyectando cada uno de los embeddings $x_1, ..., x_T \in \mathbb{R}^D$ sobre el espacio $\mathbb{R}^P$ usando 3 proyecciones distintas (en la implementación, el escalar $P \in \mathbb{N}$ se representa por el atributo `head_dim`), es decir, para cada $t \in \{1, ..., T\}$ se definen los vectores

$$q_t = W_Q\, x_t, \qquad k_t = W_K\, x_t, \qquad v_t = W_V\, x_t,$$

donde $W_Q, W_K, W_V \in \mathcal{M}_{P,D}(\mathbb{R})$ son las 3 matrices (parámetros entrenables) que realizan las 3 proyecciones lineales a $\mathbb{R}^P$ de los vectores de embedding $x_1, ..., x_T \in \mathbb{R}^D$. De esta forma, cada embedding mejorado, $y_i \in \mathbb{R}^P$, se calculará como una suma ponderada (más precisamente, una combinación convexa) de los vectores $v_1, ..., v_T \in \mathbb{R}^P$, donde los ponderadores (que serán obtenidos a partir de las proyecciones $q_1, ..., q_T \in \mathbb{R}^P$ y $k_1, ..., k_T \in \mathbb{R}^P$) indicarán el grado de importancia que tiene cada token de S para el token $w_i \in S$:

$$y_i = \sum_{j=1}^{T} a\big(q_i, k_j\big)\, v_j.$$

Cada ponderador $a(q_i, k_j) \in [0, 1]$ se interpreta como la **atención** que coloca el token w_i sobre el token w_j, lo cual se determinará utilizando las proyecciones $q_i \in \mathbb{R}^P$ y $k_j \in \mathbb{R}^P$. Por otro lado, notar que para que la combinación lineal que define $y_i \in \mathbb{R}^P$ sea convexa, se debe cumplir que $\sum_{j=1}^{T} a(q_i, k_j) = 1$.

La motivación de realizar 3 proyecciones distintas, q_t, k_t y v_t, para cada embedding $x_t \in \mathbb{R}^D$ de la secuencia de embeddings $(x_1, ..., x_T)$, viene de permitir distintas representaciones que se ajusten al papel que está jugando el respectivo token en cada situación (lógica prestada desde information retrieval):

- El vector $q_t \in \mathbb{R}^P$ (denominado **query**) es usado cuando el embedding $x_t \in \mathbb{R}^D$ es el que busca poner atención sobre otros vectores de embedding de la secuencia.
- El vector $k_t \in \mathbb{R}^P$ (denominado **key**) es usado cuando un token cualquiera de la secuencia busca poner atención sobre el embedding $x_t \in \mathbb{R}^D$.
- El vector $v_t \in \mathbb{R}^P$ (denominado **value**) es usado para representar la información útil del embedding $x_t \in \mathbb{R}^D$, la cual se considerará en la suma que define cada uno de los embeddings finales $(y_1, ..., y_T) \subset \mathbb{R}^P$.

Dado que cada embedding mejorado, $y_t \in \mathbb{R}^D$, incluye información acerca de todos los demás tokens de la secuencia $S = (w_1, ..., w_T)$ (y no solo del token $w_t \in \mathcal{V}$), a estos nuevos embeddings se les suele llamar **embeddings contextuales** ya que permite, por ejemplo, la desambiguación de palabras homógrafas, lo cual no es posible a partir de embeddings obtenidos al mirar los tokens individualmente.

Por otra parte, para satisfacer las restricciones de convexidad, $a(q_i, k_j) \geq 0$ y $\sum_{j=1}^{T} a(q_i, k_j) = 1$, es usual comenzar considerando escalares irrestrictos, $s(q_i, k_j) \in \mathbb{R}$, denominados **scores de atención**, los cuales luego son normalizados usando la función softmax, de manera similar a lo que se realiza para transformar un vector de logits en un vector de probabilidad. Notar que el cálculo de la función softmax debe realizarse a lo largo de la segunda variable de la función de score $s : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ ya que es esa la coordenada donde se pide que la suma sea 1:

$$a(q_i, k_j) = \text{softmax}(s(q_i, k_1), ..., s(q_i, k_T))_j = \frac{e^{s(q_i, k_j)}}{\sum_{t=1}^{T} e^{s(q_i, k_t)}}.$$

Para la elección de los scores de atención, $s(q_i, k_j) \in \mathbb{R}$, si bien se podría utilizar una red neuronal más pequeña que aprenda buenos valores (como ocurre en la atención de Bahdanau [53]), un enfoque sencillo y que funciona bien corresponde al **dot-product self-attention** (propuesto en la arquitectura original de Transformer [14]), el cual considera

$$s(q_i, k_j) = \frac{\langle q_i, k_j \rangle}{\sqrt{P}},$$

donde el producto punto indica el grado de similitud entre el par query-key (recordar la relación con la similitud coseno: $\cos \angle(x, y) := \frac{\langle x, y \rangle}{\|x\|\|y\|}$), mientras que la división por $\sqrt{P}$ (con P la dimensión de los vectores q_i y k_j) busca mantener la magnitud de los scores controlada, lo cual evita que la función softmax concentre casi toda su masa en una única posición al momento de calcular los ponderadores de atención.

El siguiente diagrama resume el mecanismo de auto-atención para una secuencia de embeddings $(x_1, ..., x_T) \subset \mathbb{R}^D$:

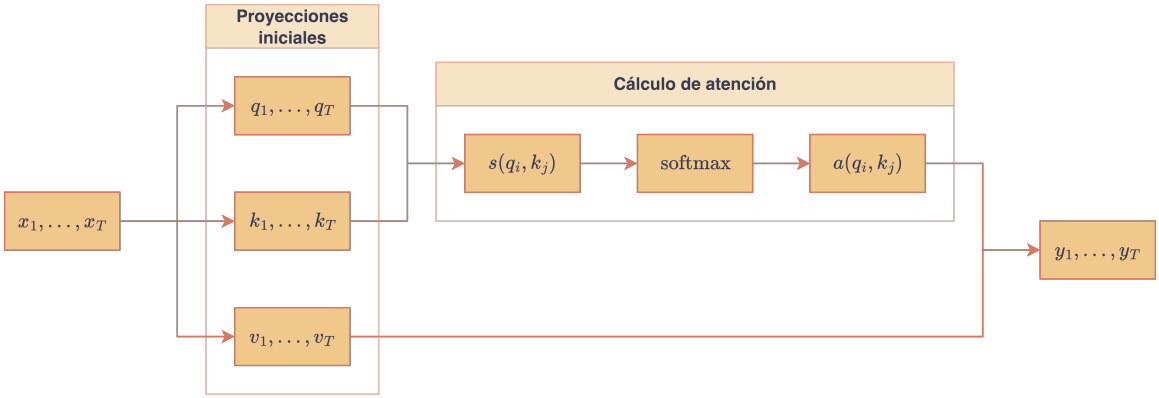

Figura 44: Diagrama del mecanismo de self-attention aplicado a una secuencia de embeddings.

Es importante notar que se pueden calcular todos los embeddings contextuales $y_1, ..., y_T \in \mathbb{R}^P$ en paralelo (hasta el momento, cada $y_t \in \mathbb{R}^P$ se debería computar de forma independiente, iterando sobre $t \in \{1, ..., T\}$). Para esto, se almacenarán las proyecciones realizadas dentro de matrices

$$Q = \begin{pmatrix} q_1^\top \\ \vdots \\ q_T^\top \end{pmatrix}, \quad K = \begin{pmatrix} k_1^\top \\ \vdots \\ k_T^\top \end{pmatrix}, \quad V = \begin{pmatrix} v_1^\top \\ \vdots \\ v_T^\top \end{pmatrix},$$

donde las 3 matrices, las cuales pertenecen a $\mathcal{M}_{T,P}(\mathbb{R})$, almacenan los respectivos embeddings proyectados, uno en cada fila, similar a como funciona una matriz de embedding. Luego, notando que $\langle q_i, k_j \rangle = \langle \text{fila}_i(Q), \text{fila}_j(K) \rangle = (QK^\top)_{ij}$, se pueden obtener los scores de atención mediante $s(q_i, k_j) = \left(\frac{QK^\top}{\sqrt{P}}\right)_{ij}$. De esta forma, si $A \in \mathcal{M}_{T,T}(\mathbb{R})$ es la **matriz de atención**, la cual almacena $A_{ij} = a(q_i, k_j)$, la observación anterior permite escribir $A = \text{softmax}_{\text{por filas}}\left(\frac{QK^\top}{\sqrt{P}}\right)$, donde $\text{softmax}_{\text{por filas}}$ indica que se debe aplicar la función softmax en

cada una de las filas de la **matriz de scores** $S = \frac{QK^\top}{\sqrt{P}} \in \mathcal{M}_{T,T}(\mathbb{R})$ (recordar que cada fila debe sumar 1, no cada columna). De este modo, si $Y \in \mathcal{M}_{T,P}(\mathbb{R})$ es la matriz que almacena los embeddings contextuales calculados por el mecanismo de atención, $(y_1, ..., y_T) \subset \mathbb{R}^P$ (un embedding en cada fila), se observa que $Y_{ij} = \sum_{k=1}^{T} a_{ik} v_{kj} = (AV)_{ij}$, por lo que se puede escribir $Y = \text{Attention}(Q, K, V)$, donde

$$\text{Attention}(Q, K, V) \coloneqq \underset{\text{por filas}}{\text{softmax}}\left(\frac{QK^\top}{\sqrt{P}}\right)V$$

es el mecanismo de atención actuando sobre los embeddings iniciales $(x_1, ..., x_T) \subset \mathbb{R}^D$, cuyas proyecciones en $\mathbb{R}^P$ se almacenan en las matrices $Q, K, V \in \mathcal{M}_{T,P}(\mathbb{R})$.

Por otro lado, las proyecciones $Q, K, V \in \mathcal{M}_{T,P}(\mathbb{R})$ también pueden ser calculadas mediante un producto matricial: si $X \in \mathcal{M}_{T,D}(\mathbb{R})$ es la matriz que almacena en sus filas los embeddings iniciales, $(x_1, ..., x_T) \subset \mathbb{R}^D$, entonces se tiene que $q_{ij} = \big(W_Q x_i\big)_j = \big\langle \text{fila}_j(W_Q), x_i \big\rangle = \big\langle \text{fila}_j(W_Q), \text{fila}_i(X) \big\rangle = \big(XW_Q^\top\big)_{ij}$ (análogo para K y V). Por lo tanto, $Q = XW_Q^\top$, $K = XW_K^\top$ y $V = XW_V^\top$, donde $W_Q, W_K, W_V \in \mathcal{M}_{P,D}(\mathbb{R})$ son las matrices de proyección definidas anteriormente.

Masked self-attention

Un detalle importante pasado por alto hasta el momento es la pérdida de causalidad al momento de computar los embeddings contextuales $y_1, ..., y_T \in \mathbb{R}^P$. En efecto, dado que el objetivo central sigue siendo construir un modelo neuronal para predecir la probabilidad del próximo token dados los tokens anteriores, cada embedding contextual y_t solo debería poder tener acceso a la información asociada a los tokens $w_1, ..., w_t \in \mathcal{V}$ y no a la información de los tokens siguientes, $w_{t+1}, ..., w_T \in \mathcal{V}$, por lo que se debe forzar que $a\big(x_i, x_j\big) = 0$ para $j > i$.

Si bien se podría sustituir directamente $A_{ij} = 0$ para $j > i$ en la matriz de atención, esto provocaría que $\sum_{j=1}^{T} a\big(q_i, k_j\big) < 1$, por lo que la combinación lineal ya no sería convexa. Una solución que evita este problema es sustituir en la matriz de scores, $S = \frac{QK^\top}{\sqrt{P}}$, cada entrada donde $j > i$ por un valor muy negativo ($-\infty$ en la implementación). De este modo, como $\lim_{s \to -\infty} e^s = 0$, se tendrá que $a\big(q_i, k_j\big) = \frac{e^{s\left(q_i, k_j\right)}}{\sum_{t=1}^{T} e^{s(q_i, k_t)}} = 0$ para $j > i$, y se seguirá cumpliendo la restricción $\sum_{j=1}^{T} a\big(q_i, k_j\big) = 1$ ya que la función softmax es aplicada luego de esta sustitución.

Esta variante de auto-atención se conoce como **masked self-attention**, y es esencial para preservar la causalidad del lenguaje. En la práctica, este tipo de exclusión se implementa construyendo una matriz auxiliar $M \in \mathcal{M}_{T,T}(\mathbb{R})$, llamada máscara, donde las entradas no

nulas de su i-ésima fila indicarán los tiempos $\{1, ..., T\}$ sobre los que podrá poner atención el embedding contextual $x_i \in \mathbb{R}^D$. En este caso, para mantener la causalidad, $M_{ij} = \begin{cases} 1 \text{ si } & j \leq i \\ 0 \text{ si } & j > i \end{cases}$ (en particular, $M \in \mathcal{M}_{T,T}(\mathbb{R})$ es una matriz triangular inferior). El siguiente diagrama muestra el mecanismo de auto-atención incluyendo el masking causal:

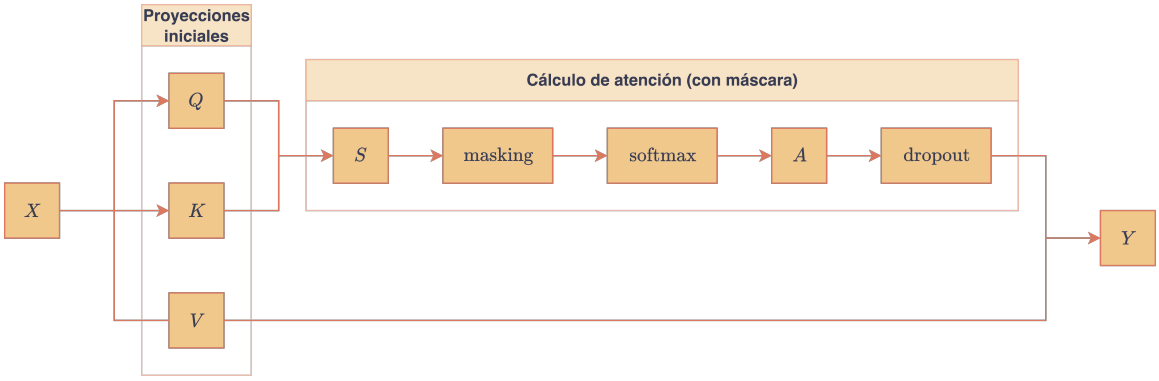

Figura 45: Diagrama del mecanismo de masked self-attention con máscara causal y dropout sobre la matriz de atención.

Notar que se incluyó una capa de dropout [26], la cual es aplicada a la matriz de atención, $A \in \mathcal{M}_{T,T}(\mathbb{R})$. Recordar que para un vector cualquiera, $x \in \mathbb{R}^D$, $\text{Dropout}_p(x) := \frac{1}{1-p} m \odot x$, donde el vector binario $m \in \{0,1\}^D$ determina sus coordenadas al azar desde $m_d \sim \text{Bernoulli}(1-p)$. Es decir, $\text{Dropout}_p(x)$ apaga el valor de ciertas coordenadas de x, impidiendo el flujo de información a través de las conexiones neuronales relacionadas con las componentes apagadas. En cambio, el factor de ponderación, $\frac{1}{1-p} \geq 1$ busca compensar el flujo perdido aumentando el valor de las coordenadas que sí quedaron encendidas. En el caso del dropout aplicado a la matriz de atención, el funcionamiento es el mismo: ciertas coordenadas de la matriz $A \in \mathcal{M}_{T,T}(\mathbb{R})$ son sustituidas por 0, mientras que las restantes son ponderadas por $\frac{1}{1-p}$, respetando (en esperanza) el cumplimiento de la restricción $\sum_{j=1}^{T} a(q_i, k_j) = 1$. En el código, el parámetro de dropout $p \in [0,1]$ se representa por el atributo `config.dropout`, el cual se compartirá de manera global en todas las capas de dropout definidas en el modelo.

Para ejemplificar el uso de dropout, se aplicará dropout (con $p = 0.2$) sobre una matriz cuadrada que simulará ser una matriz de atención (no se aplicará causalidad ni softmax por simplicidad). Notar que los valores que permanecen activos son multiplicados cada uno por $\frac{1}{1-p} = 1.25$:

```
p = 0.2
dropout = nn.Dropout(p)
matrix = torch.ones(5, 5)
matrix = dropout(matrix)
print(matrix)
```

```
tensor([[1.2500, 1.2500, 1.2500, 1.2500, 0.0000], [0.0000, 1.2500,
1.2500, 0.0000, 1.2500], [1.2500, 0.0000, 1.2500, 1.2500, 0.0000],
[1.2500, 1.2500, 0.0000, 1.2500, 1.2500], [0.0000, 1.2500, 1.2500,
1.2500, 1.2500]])
```

La motivación usual de la técnica de dropout es que el modelo aprenda a predecir la respuesta correcta aún cuando no tiene acceso a toda la información. En particular, para la matriz de atención, si $a(q_i, k_j) = 0$, entonces el embedding contextual $y_i \in \mathbb{R}^P$ (asociado al token $w_i \in S$) no podrá obtener información contextual acerca del token $w_j \in S$, independientemente de si $j \leq i$ o no. Dado que las posiciones donde ocurre el dropout se eligen de forma aleatoria en cada secuencia e iteración de entrenamiento, aplicar esta regularización en la matriz de atención fuerza a que el modelo tenga que aprender a utilizar la información de todas las posiciones del contexto para realizar una predicción. Durante la inferencia (i.e., cuando el modelo ya está entrenado y se invocó `model.eval()`), el dropout se deshabilita y todos los tokens del contexto son utilizados para el cálculo de los embeddings contextuales.

La siguiente clase implementa el mecanismo de masked self-attention. Además, aunque no es necesario hacerlo, se guardará la matriz de atención en el atributo `attn_matrix` para poder visualizarla posteriormente cuando el modelo esté en modo inferencia (recordar que el atributo `training` se hereda de la clase `nn.Module`, e indica si el modelo está en modo entrenamiento o en modo inferencia).

```
class SelfAttentionHead(nn.Module):

    def __init__(self, cfig: GPTConfig) -> None:
        super().__init__()

        self.query = nn.Linear(cfig.embedding_dim, cfig.head_dim,
bias=False)
        self.key = nn.Linear(cfig.embedding_dim, cfig.head_dim, bias=False)
        self.value = nn.Linear(cfig.embedding_dim, cfig.head_dim,
bias=False)

        self.dropout = nn.Dropout(cfig.dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:

        Q = self.query(x)
        K = self.key(x)
        V = self.value(x)
```

```
        batch_size, seq_length, head_dim = Q.shape
        mask = torch.tril(torch.ones(seq_length, seq_length)).to(x.device)

        attn = Q @ K.transpose(-1, -2) / (head_dim ** 0.5)
        attn = attn.masked_fill(mask == 0, float('-inf'))
        attn = self.dropout(attn.softmax(dim=-1))

        if self.training == False:
            self.attn_matrix = attn.detach()

        return attn @ V
```

- Ejemplo de uso.

```
sa = SelfAttentionHead(config)
x = torch.randn(batch_size, seq_length, config.embedding_dim)
y = sa(x)

assert y.shape == (batch_size, seq_length, config.head_dim)
```

Atención multicabezal

El objetivo del mecanismo de auto-atención definido anteriormente es que cada embedding contextual $y_t \in \mathbb{R}^P$ pueda resumir de mejor forma la información contenida en el vector de embedding inicial, $x_t \in \mathbb{R}^D$, haciendo uso del contexto en el que se encuentra el token $w_t \in S$ asociado al embedding x_t (i.e., usando información de los tokens vecinos). Este mecanismo puede repetirse varias veces de forma independiente sobre una misma secuencia de embeddings $(x_1, ..., x_T) \subset \mathbb{R}^D$, donde cada instancia del mecanismo de auto-atención, que se suele llamar **cabezal de atención**, tendrá sus propias matrices de proyección $W_Q, W_K, W_V \in \mathcal{M}_{P,D}(\mathbb{R})$ y, en consecuencia, su propia matriz de atención y embeddings finales, $(y_1, ..., y_T) \subset \mathbb{R}^P$. La motivación de usar varios cabezales de atención es que cada cabezal puede aprender a extraer una información contextual diferente, de forma similar a como actúan los filtros en una red convolucional.

Para una secuencia de embeddings $(x_1, ..., x_T) \subset \mathbb{R}^D$ y $H \geq 1$ cabezales de atención independientes actuando sobre esta secuencia, se obtendrán H nuevas secuencias de embeddings, $\left(y_1^{(h)}, ..., y_T^{(h)}\right) \subset \mathbb{R}^P$, con $h \in \{1, ..., H\}$. Con esto, se pueden combinar todos los cabezales para formar una secuencia de embeddings $(\tilde{y}_1, ..., \tilde{y}_T) \subset \mathbb{R}^{P \cdot H}$ de mayor dimensión concatenando los embeddings de todos los cabezales:

$$\tilde{y}_t = \left(y_t^{(1)} \cdots y_t^{(H)}\right) \in \mathbb{R}^{P \cdot H}, \quad \text{para} \quad t \in \{1, ..., T\}.$$

Además, para poder relacionar los embeddings obtenidos por cada cabezal, cada embedding extendido $\tilde{y}_t$ suele ser pasado por una capa lineal extra, lo que permite, además, recuperar la dimensión de los embeddings originales, $(x_1, ..., x_T) \subset \mathbb{R}^D$. Por lo tanto, los embeddings finales que retorna un bloque de atención multicabezal, $(u_1, ..., u_T) \in \mathbb{R}^D$, son de la forma $u_t = W_0\, \tilde{y}_t \in \mathbb{R}^D$, donde la matriz $W_0 \in \mathcal{M}_{D,P\cdot H}(\mathbb{R})$ proyecta cada embedding concatenado, $\tilde{y}_t \in \mathbb{R}^{P\cdot H}$, de vuelta al espacio $\mathbb{R}^D$. El siguiente diagrama resume esta idea:

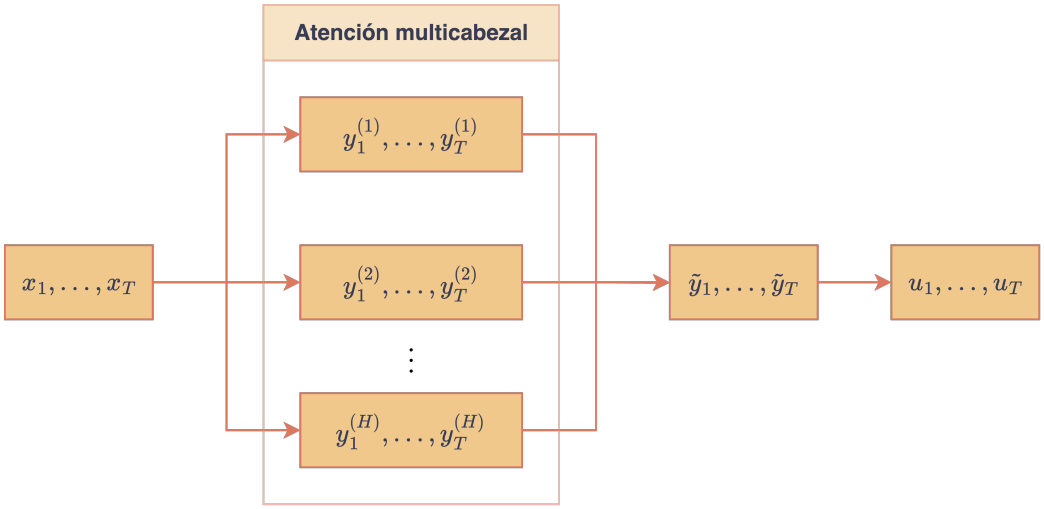

Figura 46: Diagrama del mecanismo de atención multicabezal con concatenación y proyección lineal final.

En la práctica (y en la clase de hiperparámetros `GPTConfig`) es usual considerar como dimensión de cada cabezal a $P = \frac{D}{H}$, donde se asume que $H \mid D$ (i.e., D es divisible por H). La idea de esto es considerar que la dimensión de embedding $D \in \mathbb{N}$ se reparte entre los $H \in \mathbb{N}$ cabezales de atención. En la siguiente implementación, H es representado por el atributo `num_head`:

```
class MultiHeadSelfAttention(nn.Module):

    def __init__(self, cfig: GPTConfig) -> None:
        super().__init__()
        self.heads = nn.ModuleList([SelfAttentionHead(cfig) for _ in
range(cfig.num_heads)])
        self.projection = nn.Linear(cfig.num_heads * cfig.head_dim,
cfig.embedding_dim, bias=False)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        output = torch.cat([head(x) for head in self.heads], dim=-1)
        output = self.projection(output)
        return output
```

- Ejemplo de uso.

```
mhsa = MultiHeadSelfAttention(config)
x = torch.randn(batch_size, seq_length, config.embedding_dim)
y = mhsa(x)

assert y.shape == (batch_size, seq_length, config.embedding_dim)
```

Por otra parte, es posible computar los H cabezales de atención en paralelo aplicando el mecanismo de atención sobre matrices Q, K y V más grandes. Aquí no se realizó dicha paralelización por simplicidad en la implementación, pero en la práctica sí se debería realizar para mayor eficiencia.

3.3.3 Red feedforward

El último módulo que compone un bloque Transformer de la arquitectura GPT está enfocado en pasar cada uno de los embeddings obtenidos por la atención multicabezal, $(u_1, ..., u_T) \subset \mathbb{R}^D$, por una misma red fully connected de dos capas. La idea de esta transformación final es darle una mayor capacidad a la red neuronal mediante un post--procesamiento individual de cada embedding obtenido anteriormente en el mecanismo de atención (cuya función principal era combinar los distintos tokens de la secuencia entre sí). Para cada $u_t \in \mathbb{R}^D$, $t \in \{1, ..., T\}$, el módulo feed forward calcula

$$\mathrm{FF}(u_t) = W_2\, \phi(W_1\, u_t),$$

donde $W_1 \in \mathcal{M}_{J,D}(\mathbb{R})$, $W_2 \in \mathcal{M}_{D,J}(\mathbb{R})$ son los parámetros de la red (con $J \in \mathbb{N}$ la dimensión interna del bloque feed forward) y $\phi : \mathbb{R} \to \mathbb{R}$ es una función de activación que se aplica coordenada a coordenada. El siguiente diagrama resume el uso de la red feed forward para una secuencia $(u_1, ..., u_T) \subset \mathbb{R}^D$ de embeddings, donde se ha denotado por $(r_1, ..., r_T) \subset \mathbb{R}^D$ los vectores de salida:

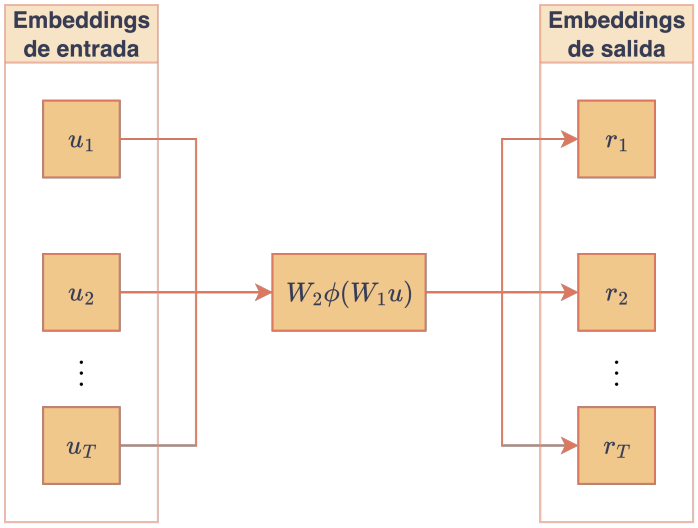

Figura 47: Diagrama de la red feedforward aplicada de forma independiente a cada embedding de la secuencia.

Notar que las dos capas lineales, al igual que todas las otras capas lineales usadas en esta arquitectura, no incluyen un término de sesgo, lo cual se ha vuelto una práctica común en los LLMs y arquitecturas neuronales modernas. Además, es usual considerar $J \gg D$ para darle más capacidad de representación a la red. Más específicamente, J suele ser un múltiplo de D (e.g., $J = 4D$), lo cual será indicado en el hiperparámetro `cfig.ff_factor`. Por otro lado, en la implementación se utilizará GELU [54] como función de activación (i.e., $\phi(x) = x \cdot \Phi(x)$, donde $\Phi : \mathbb{R} \to [0, 1]$ es la distribución acumulada de una gaussiana estándar), lo que es una práctica común para este tipo de modelos, cuya elección, hasta donde tengo entendido, está basada únicamente en resultados empíricos. La siguiente clase `FeedForward` implementa este módulo:

```
class FeedForward(nn.Module):

    def __init__(self, cfig: GPTConfig) -> None:
        super().__init__()
        self.fc1 = nn.Linear(cfig.embedding_dim, cfig.ff_factor *
cfig.embedding_dim, bias=False)
        self.activation = nn.GELU()
        self.fc2 = nn.Linear(cfig.ff_factor * cfig.embedding_dim,
cfig.embedding_dim, bias=False)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.fc1(x)
        x = self.activation(x)
```

```
        x = self.fc2(x)
        return x
```

- Ejemplo de uso

```
ff = FeedForward(config)
x = torch.randn(batch_size, seq_length, config.embedding_dim)
y = ff(x)

assert y.shape == (batch_size, seq_length, config.embedding_dim)
```

3.3.4 Bloque Transformer

Los 3 módulos definidos anteriormente, `LayerNorm`, `MultiHeadSelfAttention` y `FeedForward`, permiten construir el bloque principal de la arquitectura GPT, llamado **bloque Transformer**, el cual es repetido varias veces en la arquitectura GPT, dependiendo de la profundidad de la arquitectura.

Dada una secuencia de embeddings, $(x_1, ..., x_T) \subset \mathbb{R}^D$, almacenada en una matriz $X \in \mathcal{M}_{T,D}(\mathbb{R})$ (un vector de embedding en cada fila), un bloque Transformer entrega otra secuencia de embeddings mejorados (por el mecanismo de atención y luego la red feed forward) en una matriz de salida del mismo tamaño, $\text{TransformerBlock}(X) \in \mathcal{M}_{T,D}(\mathbb{R})$:

$$\begin{cases} Y = X + \text{Dropout}_p\big(\text{MHSA}\big(\text{LayerNorm}_{\text{por filas}}(X)\big)\big) \\ \text{TransformerBlock}(X) = Y + \text{Dropout}_p\big(\text{FF}_{\text{por filas}}\big(\text{LayerNorm}_{\text{por filas}}(Y)\big)\big) \end{cases}.$$

- La función $\text{LayerNorm}_{\text{por filas}}$ indica que se debe aplicar el módulo `LayerNorm` de forma independiente en cada fila de las matrices X e Y (i.e., se aplica en cada uno de los vectores de embedding).
- La función MHSA corresponde a aplicar el módulo `MultiHeadSelfAttention` a la secuencia de embeddings normalizados contenida en la matriz $\text{LayerNorm}_{\text{por filas}}(X) \in \mathcal{M}_{T,D}(\mathbb{R})$.
- La función $\text{FF}_{\text{por filas}}$ corresponde a aplicar el módulo `FeedForward` a cada uno de los embeddings normalizados contenidos en la matriz $\text{LayerNorm}_{\text{por filas}}(Y) \in \mathcal{M}_{T,D}(\mathbb{R})$.

Notar que, en la práctica, gracias al broadcasting automático realizado por PyTorch, las operaciones por filas se obtienen de forma automática al pasar las matrices de embeddings por los módulos `LayerNorm` y `FeedForward`. Sin embargo, se debe tener presente que dichos módulos actúan siempre de forma independiente sobre cada posición de la secuencia de embeddings.

En resumen, un bloque Transformer no es más que aplicar el mecanismo de atención multicabezal, seguido de una red feed forward. Para ambos módulos, se aplica anteriormente una

capa de normalización (para estabilizar la entrada al respectivo módulo) y posteriormente una capa de dropout (como una técnica de regularización). Además, en ambos módulos se realiza una **conexión residual** (i.e., se suma la entrada a la salida de cada módulo). En el próximo capítulo se realizará una revisión más detallada de estas decisiones de diseño y de sus efectos en la arquitectura.

El siguiente diagrama resume el funcionamiento de un bloque Transformer:

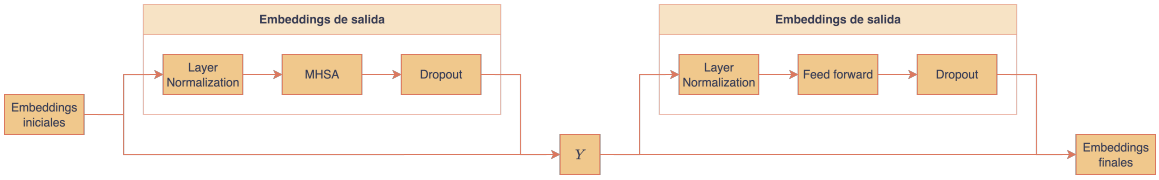

Figura 48: Diagrama de un bloque Transformer con self-attention multicabezal, red feed-forward, layer normalization, dropout y conexiones residuales.

Las capas de dropout aplicadas posterior a la atención multicabezal y a la red feedforward tienen el mismo rol que antes: descartar algunas componentes de algunos embeddings de forma aleatoria. Notar que el dropout aplicado en estas capas no descarta tokens completos como ocurre al usar masking en los scores de atención, donde el masking de un parámetro $a(q_i, k_j) \in [0, 1]$ no permite que el i-ésimo token coloque atención sobre el j-ésimo token.

La siguiente clase `TransformerBlock` implementa un bloque Transformer usando los módulos implementados anteriormente:

```
class TransformerBlock(nn.Module):

    def __init__(self, cfig: GPTConfig) -> None:
        super().__init__()
        self.norm1 = LayerNorm(cfig.embedding_dim)
        self.mhsa = MultiHeadSelfAttention(cfig)
        self.norm2 = LayerNorm(cfig.embedding_dim)
        self.ff = FeedForward(cfig)
        self.dropout = nn.Dropout(cfig.dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = x + self.dropout(self.mhsa(self.norm1(x)))
        x = x + self.dropout(self.ff(self.norm2(x)))
        return x
```

- Ejemplo de uso.

```
transformer_blocks = TransformerBlock(config)
x = torch.randn(batch_size, seq_length, config.embedding_dim)
y = transformer_blocks(x)
```

```
assert y.shape == (batch_size, seq_length, config.embedding_dim)
```

3.3.5 Clase principal

El bloque Transformer definido anteriormente es el componente principal de la arquitectura GPT (y de las arquitecturas tipo Transformer en general). Este bloque es repetido varias veces para aumentar la complejidad del modelo, de forma análoga a como se repiten los bloques ResNet [55] en una red convolucional. Sin embargo, es necesario agregar elementos auxiliares antes y después de la ejecución de los bloques Transformer:

- **Capa de embedding:** todos los módulos implementados hasta el momento (en particular, el bloque Transformer) esperan recibir una secuencia de vectores de embedding $(x_1, ..., x_T) \subset \mathbb{R}^D$ y no directamente una secuencia de tokens $(w_1, ..., w_T) \in \mathcal{V}^*$. Por lo tanto, al igual que en la RNN anterior, se debe anteponer una capa de embedding $\text{emb} : \mathcal{V} \to \mathbb{R}^D$ que transforme esta secuencia de símbolos discretos en vectores que la red neuronal pueda procesar.
- **Positional encoding:** dado que las RNNs procesan los tokens de una secuencia uno a uno, esto le entrega información al modelo acerca del orden de los respectivos tokens dentro de la secuencia. Sin embargo, los bloques Transformer no reciben, al menos de forma explícita, ninguna información acerca de la posición dentro de la secuencia que tienen los embeddings $(x_1, ..., x_T) \subset \mathbb{R}^D$ que va a procesar. Una posible opción para inyectar explícitamente esta información es agregar una segunda capa de embedding (similar a la que codifica cada símbolo del vocabulario $\mathcal{V}$ en un vector de embedding) que codifique cada posición de la secuencia, $(1, 2, ..., T) \subset \mathbb{N}$, en un vector de embedding, $\text{emb}(t) \in \mathbb{R}^D$. De esta forma, el embedding final que entra al primer bloque Transformer consistirá en sumar los embeddings asociados a los tokens de la secuencia (**token embeddings**) con los embeddings asociados a la posición absoluta de cada token dentro de la secuencia (**positional embeddings**). Notar que el uso de una capa de embedding para la posición automáticamente limita el largo máximo de las secuencias (según la cantidad de filas que tenga la matriz de embedding respectiva). Este valor máximo se conoce como **ventana de contexto**.
- **Cabezal de lenguaje:** luego de aplicar consecutivamente los bloques Transformer a las secuencias de embeddings que se van mejorando iterativamente, es necesario agregar un cabezal de lenguaje análogo al utilizado en la RNN, el cual transforma la salida del último bloque Transformer en un vector de probabilidades sobre $\mathcal{V}$, lo cual permitirá utilizar la arquitectura GPT para aprender el vector de probabilidades asociado a la distribución del próximo token dados los tokens de entrada al modelo.

El siguiente diagrama resume la arquitectura GPT completa:

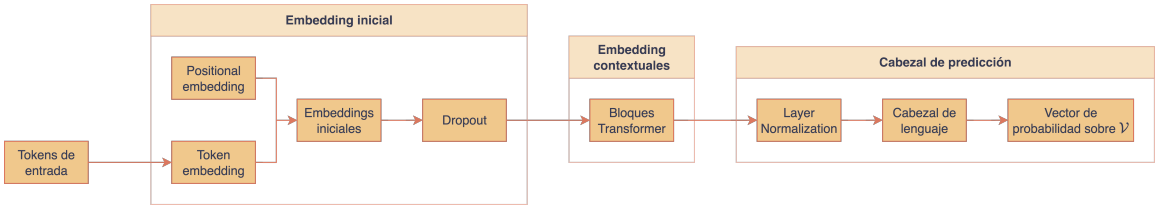

Figura 49: Diagrama de la arquitectura GPT completa [47].

Notar que se agregó una capa de dropout antes de pasar la secuencia de vectores de embedding al primer bloque Transformer. El objetivo de esta capa es, al igual que antes, forzar a que el modelo aprenda a usar todas las posiciones de la secuencia y todas las dimensiones de los vectores de embedding. Adicionalmente, se agregó una capa de normalización previo a la proyección realizada por el cabezal de lenguaje, lo cual permite estabilizar el entrenamiento.

La siguiente clase `GPT` implementa la arquitectura GPT completa utilizando los módulos implementados anteriormente:

```
class GPT(nn.Module):

    def __init__(self, cfig: GPTConfig) -> None:
        super().__init__()

        self.token_embedding = nn.Embedding(cfig.vocab_size,
cfig.embedding_dim)
        self.position_embedding = nn.Embedding(cfig.context_window,
cfig.embedding_dim)
        self.dropout = nn.Dropout(cfig.dropout)
        self.transformer_blocks = nn.Sequential(*[TransformerBlock(cfig)
for _ in range(cfig.num_layers)])
        self.norm_output = LayerNorm(cfig.embedding_dim)
        self.lm_head = nn.Linear(cfig.embedding_dim, cfig.vocab_size,
bias=False)

        self.context_window = cfig.context_window

    def forward(self, x: torch.Tensor) -> torch.Tensor:

        x = x[:, -self.context_window:]
        batch_size, seq_length = x.shape

        token_embedding = self.token_embedding(x.long())
        positional_embedding =
self.position_embedding(torch.arange(seq_length, device=x.device))
```

```
        output = token_embedding + positional_embedding
        output = self.dropout(output)
        output = self.transformer_blocks(output)
        output = self.norm_output(output)
        output = self.lm_head(output)
        return output
```

- Ejemplo de uso.

```
gpt = GPT(config)
x = torch.randint(config.vocab_size, size=[batch_size, seq_length])
y = gpt(x)

assert y.shape == (batch_size, seq_length, config.vocab_size)
```

3.3.6 Entrenamiento y generación

Para entrenar y generar nuevas secuencias de texto utilizando este modelo se puede reutilizar sin ningún cambio el código implementado anteriormente para la RNN. Esto muestra que las RNNs y la arquitectura GPT son similares en su objetivo, solo que procesan internamente las secuencias de forma diferente. En este caso, los hiperparámetros de arquitectura que se utilizarán serán los definidos en la clase `GPTConfig`:

```
gpt = GPT(config)

n_params = sum(param.numel() for param in gpt.parameters()) / 1e6
print(f'Cantidad de parámetros: {n_params:.3} millones.')
```

```
Cantidad de parámetros: 57.0 millones.
```

Para comparar de forma justa las curvas de entrenamiento del modelo GPT con la curva de la RNN, este modelo también se entrenará durante 32 épocas, y utilizando el mismo optimizador y dataloader:

```
gpt_optimizer = optim.AdamW(gpt.parameters())
train_model(gpt, gpt_optimizer, dataloader, epochs=32,
ckpt_filename='gpt_training.pt')
```

El entrenamiento de este modelo toma aproximadamente 7 minutos en Google Colab (usando, igual que antes, una GPU NVIDIA Tesla T4). Ahora se cargará el modelo y se verán las curvas de entrenamiento de ambos modelos:

```
rnn_training = torch.load('rnn_training.pt', DEVICE, weights_only=True)
gpt_training = torch.load('gpt_training.pt', DEVICE, weights_only=True)
gpt.load_state_dict(gpt_training['model'])

plt.figure(figsize=(10, 5))
plt.plot(rnn_training['losses'], label='RNN')
plt.plot(gpt_training['losses'], label='GPT')
plt.xlabel('Iteración')
plt.ylabel('Entropía cruzada')
plt.grid(alpha=0.3)
plt.legend()
plt.show()
```

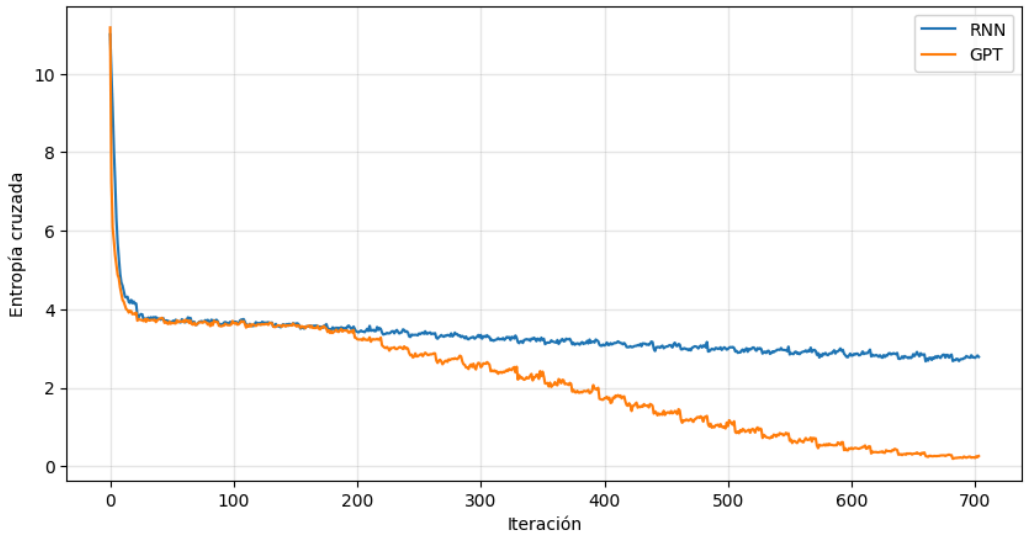

Figura 50: Comparación de las curvas de entropía cruzada durante el entrenamiento de la RNN y la arquitectura GPT.

Se observa que la complejidad de la arquitectura GPT permite obtener una entropía cruzada mucho más baja que con la RNN. Un detalle importante aquí es que al no estar midiendo la función de pérdida en un conjunto de test (i.e., en un conjunto i.i.d con respecto al conjunto de entrenamiento), no es posible ver si el entrenamiento está realizando overfitting o no.

Para la generación, se usará el mismo código de antes (con los mismos parámetros por defecto):

```
context = 'Habia una vez'

for n in range(10):
```

```
new_tokens = generate_tokens(gpt, context, dataset.tokenizer)
print(new_tokens)
```

```
Habia una vez un matrimonio que tenia un solo hijo. El hombre sembro
la mas hermosa papa en una tierra que estaba lejos de la casa que
habitaban. Habia una vez un matrimonio que tenia un niño hijo.
El hombre sembro la mas hermosa papa de una tierra. En la papa
me miraba porque levantaba la tierra. Una la ciudad habia dibujo
sombra, tenia, pero se excelsa clase de semilla. Habia una vez un
espejo de mano que cuando se quedaba solo y nadie se veia en el
se sentia de lo peor, como que no existia, y quiza tenia razon
pero los otros espejos se burlaban.. Habia una vez un matrimonio
que tenia un solo tenia verano. En esa lo esa hermosa casi por un
publica. En un vuelo un espejo de las casas saludado, Yo me quedaba
edad, habian instalado un persona, cuando. Habia una vez un espejo
de mano que cuando se quedaba solo y nadie se veia en el, razon
por su cual la cual que tenia existia, y quiza tenia razon pero
los otros espejos se burlaban de el, y, se asesorados, noches, el,
en el mismo cajon del, el. Una pierna suelta satisfechos, ajenos a
la, del alma, salir, que, un momento, era niño lo vio. Habia una
vez un niño llamado David N. El niño que niño sembro y y hermosa
y vacias, un mediano. El una muerte en la casa de la casa en la
espalda Habia una vez una viuda de buen pasar, que tenia una hija.
La muchacha era hermosa y la madre queria casarla con un hombre bien
rico. Se presentaron algunos pretendientes, pero hombres honrados,
trabajadores y acomodados, pero, y, los despedia con su musica a
otra parte, Por pelo, riquisimos….. Habia una vez un niño llamado
David N. Me conversaciones El niño Chapachapa, y en la la resortera
despertaba y capataz y admiracion en sus amigos el y vacas y resortera
la escuela, que. Habia una vez un matrimonio que tenia un solo hijo.
El hombre sembro la mas hermosa papa en una tierra que estaba lejos
de la casa. En la casa, Acababa la arboles miraba la sombra, sola.
Pero con casa se le le madre llamaron.. Habia una vez un niño llamado
David N., cuya punteria y habilidad en el manejo de la resortera
despertaba tanta envidia y admiracion, y me admiracion y la casa,
y, puño, el azar madre, firmeza y su excelsa vacias entre, con su
musica, no llamaba…
```

Se observa que el modelo, a diferencia de la RNN anterior, es capaz de generar texto con un poco más de sentido. Sin embargo, resulta ser que gran parte del texto generado corresponde

a fragmentos exactos del dataset usado para entrenar (ver archivo `data.txt`), mostrando que el modelo, si bien tiene la capacidad para aprender a generar secuencias, se sobreajustó fuertemente al conjunto de entrenamiento. Este es un comportamiento esperado en este caso ya que solo se utilizaron un poco más de 700 secuencias para entrenar el modelo. Además, al haber realizado 32 iteraciones sobre el conjunto de entrenamiento, el sobreajuste es casi inevitable (considerar que muchos LLMs realizan por lo general 1 o 2 iteraciones sobre cada dataset de entrenamiento).

Por otro lado, se observa que el tiempo de inferencia es bastante lento. Esto se debe a que el modelo es llamado cada vez que se quiere generar un nuevo token ya que, si bien el entrenamiento es paralelizable, la generación no lo es. Notar, además, que el atributo `attn_matrix` definido en el forward de la clase `SelfAttentionHead` (el cual solo se define cuando el modelo está en modo inferencia) guarda las matrices de atención en cada llamada al modelo, lo que hace aún más lenta la inferencia. En una implementación estándar no es necesario guardar las matrices (y de hecho, no se realiza para evitar el deterioro en la eficiencia), pero aquí se incluyó con fines didácticos.

Visualización de las matrices de atención

En este apartado se visualizarán las matrices de atención `attn_matrix` que se calculan cuando el modelo GPT procesa una secuencia. Si bien esto no es necesario en la práctica, permite ver cómo el modelo está procesando la entrada, entregando propiedades de interpretabilidad a la arquitectura.

La siguiente función `show_attn_matrix` grafica los valores de las matrices de atención de todos los cabezales de atención de un bloque, para una secuencia de entrada dada:

```
def show_attn_matrix(model: GPT, text: str, block: int) -> None:

    tokens = re.findall(r'\d|[^\w\s]|\w+|\s', text)
    token_ids = dataset.tokenizer.encode(text)
    x = torch.tensor(token_ids, device=DEVICE).unsqueeze(0)

    model.eval()
    _ = model(x)

    heads = model.transformer_blocks[block].mhsa.heads
    attn_matrices = [head.attn_matrix[0] for head in heads]

    fig, axes = plt.subplots(2, len(heads) // 2, figsize=(14, 9))
    fig.suptitle(f'Matrices de atención (block={block})')

    for head, ax in enumerate(axes.flatten()):
```

```
            ax.imshow(attn_matrices[head])
            ax.set_xticks(ticks=range(len(tokens)), labels=tokens)
            ax.set_yticks(ticks=range(len(tokens)), labels=tokens)
            ax.set_title(f'Head {head + 1}')

    plt.tight_layout()
    plt.show()
```

Para el 3º bloque Transformer del modelo entrenado:

```
text = 'Habia una vez, en un pueblo muy lejano'
show_attn_matrix(gpt, text, block=2)
```

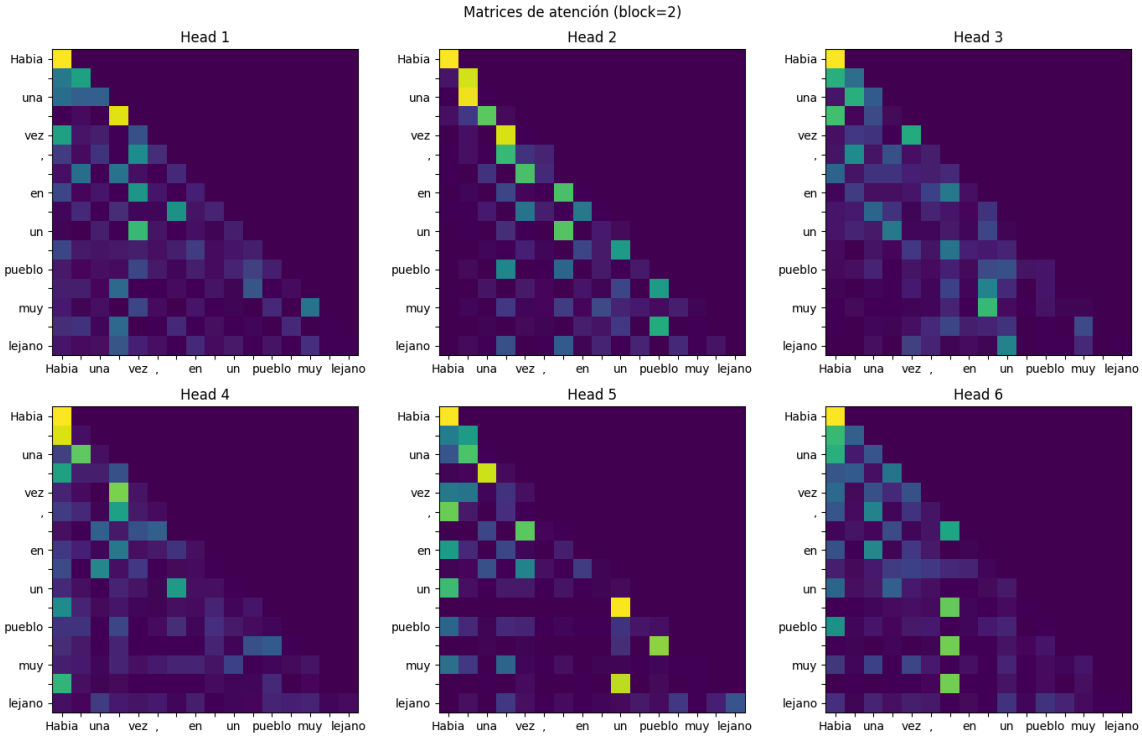

Figura 51: Matrices de atención del tercer bloque Transformer para la secuencia de entrada, donde se observa la estructura triangular inferior inducida por la máscara causal.

Se observa que todas las matrices son triangulares inferior, lo que es consistente con la causalidad impuesta en el modelo. En entradas posteriores se revisarán más en detalle algunas propiedades conocidas acerca de estas matrices de atención.

El siguiente capítulo revisará algunas variantes de la arquitectura GPT y de la arquitectura Transformer en general, las cuales han permitido extender aún más los límites de este tipo de modelos en las arquitecturas modernas. Además, se revisarán otro tipo de modelos, no

necesariamente autorregresivos, que pueden aprovechar la arquitectura Transformer para otro tipo de tareas como el procesamiento de imágenes o tareas discriminativas en general.

Parte III

Redes generativas adversarias

Capítulo 7

GANs y arquitecturas neuronales para imágenes

Las **redes generativas adversarias** (GANs) [25] son una familia de modelos generativos de variable latente, $p_\theta(x, z) = p(z)\, p_\theta(x \mid z)$, que constituyó el estado del arte en la generación de imágenes hasta la llegada de los modelos de difusión. A diferencia de otros paradigmas, las GANs no modelan explícitamente la densidad $p_\theta(x)$, sino que aprenden directamente un procedimiento para generar muestras desde $p_\theta(x)$ sin tener que evaluar dicha densidad. Esta característica clasifica a las GANs como **modelos generativos implícitos** [56], los cuales no pueden ser entrenados por enfoques que involucren la verosimilitud ya que no conocen $p_\theta(x)$. Si bien hay varios enfoques para evitar trabajar con la verosimilitud, las GANs utilizan un mecanismo de entrenamiento adversarial entre dos redes neuronales. En este capítulo se formulará e implementará una GAN básica, se introducirá la convolución transpuesta como bloque básico para el upsampling de los generadores, y se revisarán algunas de las arquitecturas neuronales más representativas de la tarea de generación de imágenes con GANs. Por otro lado, más allá del valor histórico de estos modelos, varias de las técnicas que se introdujeron aquí siguen siendo relevantes para el diseño de arquitecturas neuronales para otros paradigmas como los modelos de difusión.

7.1 Formulación de una GAN

7.1.1 Modelo generativo implícito

La red bayesiana asociada a una GAN es la red bayesiana de variable latente estándar, $p_\theta(x, z) = p(z)\, p_\theta(x \mid z)$, donde el prior latente se suele elegir como una distribución de la que es fácil generar muestras (e.g., $p(z) \sim \mathcal{N}(0, \mathrm{I}_L)$, aunque también se puede utilizar la distribución uniforme). En cambio, para la parte generadora $p_\theta(x \mid z)$, esta viene dada por una transformación determinista de z a través de una red neuronal $G_\theta : \mathbb{R}^L \to \mathbb{R}^D$ (denominada **generador**), es decir:

$$p_\theta(x \mid z) \sim \delta_{G_\theta(z)}(x).$$

De esta forma, $p_\theta(x \mid z)$ es una distribución que concentra toda su masa en el punto $G_\theta(z) \in \mathbb{R}^D$ (i.e., la única muestra que puede obtenerse desde $p_\theta(x \mid z)$ es $x = G_\theta(z) \in \mathbb{R}^D$). Esta es una primera diferencia con respecto a otros modelos generativos, donde lo usual es fijar distribuciones pertenecientes a alguna familia paramétrica (e.g., gaussianas o categóricas), y usar una red neuronal para aprender sus parámetros en vez de aprender directamente la transformación $z \mapsto x$.

7.1.2 Discriminador y entrenamiento adversarial

Para entrenar la red neuronal G_θ sin acceso a la función de verosimilitud, $p_\theta(x)$, una GAN introduce un clasificador binario auxiliar, $q_\phi(y \mid x)$, denominado **discriminador**, cuya tarea es distinguir si una muestra $x \in \mathbb{R}^D$ proviene de la distribución de datos $p_{\text{data}}(x)$ ($y = 1$, real) o de la distribución implícita $p_\theta(x)$ ($y = 0$, sintética). Naturalmente, el modelo discriminativo se elige como un clasificador binario estándar:

$$q_\phi(y \mid x) \sim \text{Bernoulli}\big(D_\phi(x)\big),$$

donde $D_\phi : \mathbb{R}^D \to [0, 1]$ es otra red neuronal independiente de $G_\theta : \mathbb{R}^L \to \mathbb{R}^D$. Mientras D_ϕ se entrena para clasificar correctamente, G_θ se entrena para engañarlo, lo que fuerza, como subproducto, a que $p_\theta(x)$ genere muestras similares a $p_{\text{data}}(x)$.

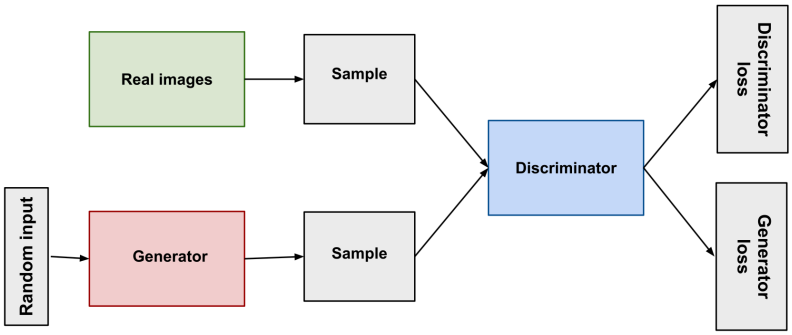

Figura 52: Diagrama de entrenamiento de una GAN. Imagen obtenida desde [17].

Es importante destacar que $q_\phi(y \mid x)$ no forma parte del modelo generativo; es solo un mecanismo de entrenamiento que se descarta una vez finalizado el ajuste del generador. La GAN, vista como modelo, sigue siendo una única red bayesiana $z \to x$.

Función objetivo

Dado que D_ϕ es un clasificador, su entrenamiento se puede realizar siguiendo el criterio de máxima verosimilitud sobre una distribución conjunta de pares (x, y) formados por imágenes reales (generadas desde $p_{\text{data}}(x)$; $y = 1$) e imágenes sintéticas (generadas desde

$p_\theta(x);\ y=0$). Si las muestras reales y sintéticas se utilizan de forma balanceada en el entrenamiento, esta distribución es

$$p_{\text{train}}(x,y) = \frac{1}{2}(p_{\text{data}}(x)\,\delta_1(y) + p_\theta(x)\,\delta_0(y)).$$

Desarrollando la log-verosimilitud del clasificador $q_\phi(y\,|\,x) \sim \text{Bernoulli}\big(D_\phi\big)$:

$$q_\phi(y \mid x) = \begin{cases} D_\phi(x) & \text{si } \ y=1 \\ 1-D_\phi(x) & \text{si } \ y=0 \end{cases} = D_\phi(x)^y\big(1-D_\phi(x)\big)^{1-y},$$

por lo que $\log q_\phi(y \mid x) = y\log D_\phi(x) + (1-y)\log\big(1-D_\phi(x)\big)$. En consecuncia, la función objetivo de una GAN es:

$$\begin{aligned}
\mathcal{L}_{\text{GAN}}(\theta,\phi) &:= \mathbb{E}_{p_{\text{train}}(x,y)}\big[\log q_\phi(y\,|\,x)\big] \\
&= \frac{1}{2}\mathbb{E}_{p_{\text{train}}(x,y)}\big[y\log D_\phi(x) + (1-y)\log\big(1-D_\phi(x)\big)\big] \\
&= \frac{1}{2}\Big(\mathbb{E}_{p_{\text{data}}(x)}\big[\log D_\phi(x)\big] + \mathbb{E}_{p_\theta(x)}\big[\log\big(1-D_\phi(x)\big)\big]\Big) \\
&= \frac{1}{2}\Big(\mathbb{E}_{p_{\text{data}}(x)}\big[\log D_\phi(x)\big] + \mathbb{E}_{p(z)}\big[\log\big(1-D_\phi(G_\theta(z))\big)\big]\Big)
\end{aligned}$$

Recordando que el discriminador (clasificador) busca maximizar su rendimiento y el generador busca que el discriminador que se equivoque, el entrenamiento de una GAN consiste en optimizar el siguiente juego minimax:

$$\min_\theta \max_\phi \mathcal{L}_{\text{GAN}}(\theta,\phi).$$

En la práctica, las esperanzas se aproximan mediante estimaciones de Monte Carlo. La primera se estima con un conjunto de entrenamiento $\mathcal{D} = \{x^1,...,x^N\} \subset \mathbb{R}^D$ de muestras i.i.d. desde $p_{\text{data}}(x)$, mientras que la segunda se estima generando muestras i.i.d. $\{z^1,...,z^N\} \subset \mathbb{R}^L$ desde $p(z) \sim \mathcal{N}(0,\mathrm{I}_L)$.

Por otro lado, es usual escribir la función objetivo de manera separada para cada red neuronal dado que muchas veces se suele modificar una (o ambas) funciones objetivo:

$$\begin{aligned}
\mathcal{L}_D(\phi) &= \mathbb{E}_{p_{\text{data}}(x)}\big[\log D_\phi(x)\big] - \mathbb{E}_{p(z)}\big[\log\big(1-D_\phi(G_\theta(z))\big)\big] \\
\mathcal{L}_G(\theta) &= \mathbb{E}_{p(z)}\big[\log\big(1-D_\phi(G_\theta(z))\big)\big],
\end{aligned}$$

donde se han omitido los factores $\frac{1}{2}$ y el término constante para θ en la verosimilitud.

Notar que esta función objetivo es **saturante** (en el sentido de que sufre de vanishing gradients), lo que es particularmente malo durante el comienzo del entrenamiento ya que en esta etapa el generador es muy débil, por lo que el discriminador puede diferenciar fácilmente una muestra real de una falsa. En consecuencia, al comienzo del entrenamiento, $D_\phi(G_\theta(z)) \approx 0$, por lo que el gradiente de $\log\big(1 - D_\phi(G_\theta(z))\big)$ respecto a θ es muy pequeño, lo que dificulta el aprendizaje.

Para resolver esto, en la práctica se entrena el generador minimizando una función de pérdida **no saturante**:

$$\mathcal{L}_G(\theta) = -\mathbb{E}_{p(z)}\left[\log D_\phi(G_\theta(z))\right].$$

Esta función de pérdida busca maximizar la probabilidad de que el discriminador clasifique las muestras falsas como reales, manteniendo el objetivo inicial del generador, pero otorgándole gradientes útiles a la red neuronal G_θ.

En la siguiente figura se puede ver la diferencia entre ambas curvas, en función de la salida del discriminador, $d = D_\phi(G_\theta(z))$:

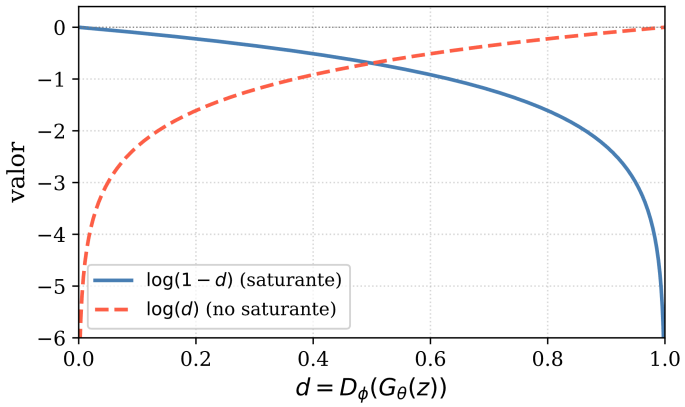

Figura 53: Función objetivo saturante y no saturante. Se observa que al comienzo del entrenamiento (cuando $d \approx 0$), la derivada de la función saturante es casi nula.

Respecto al entrenamiento basado en el juego minimax, en cada iteración se comienza optimizando los parámetros del discriminador D_ϕ varias veces (parte max), y luego se actualizan los parámetros del generador G_θ una vez (parte min). Esta estrategia, en la que el discriminador se entrena más que el generador, busca mantener al discriminador en un estado cercano al óptimo durante el entrenamiento, lo que proporciona señales de gradiente más confiables para el generador. Si el discriminador se debilita demasiado y pierde capacidad para distinguir entre datos reales y generados, el generador deja de recibir feedback útil durante el entrenamiento.

7.1.3 Implementación sobre datos bidimensionales

Para fijar la formulación se entrenará una GAN sobre un dataset bidimensional ($D = 2$). Las librerías que se utilizarán son las siguientes:

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.datasets import make_swiss_roll
```

Como distribución de datos $p_{\text{data}}(x)$ se utiliza el dataset `make_swiss_roll` de scikit-learn [43], proyectado a $\mathbb{R}^2$, con ruido gaussiano de baja varianza para añadir variabilidad:

```
def get_batch(batch_size=1000, noise=0.1):
    x, _ = make_swiss_roll(batch_size, noise=noise)
    x = x[:, [0, 2]]
    x = (x - x.mean()) / x.std()
    return torch.tensor(x).float()

# Ejemplo:
samples = get_batch()
plt.figure(figsize=(3, 3))
plt.scatter(samples[:, 0], samples[:, 1], s=1)
plt.show()
```

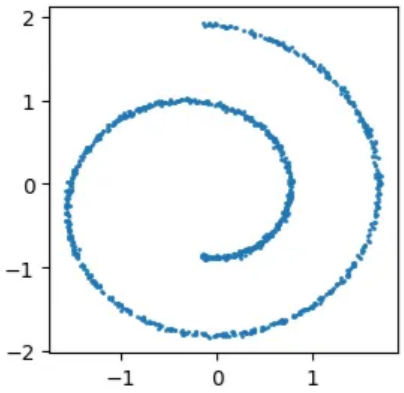

Figura 54: Muestras del dataset swiss roll proyectado a $\mathbb{R}^2$.

Para las redes neuronales $G_\theta : \mathbb{R}^L \to \mathbb{R}^D$ y $D_\phi : \mathbb{R}^D \to [0, 1]$ se utilizarán dos redes fully connected de cuatro capas. Como dimensión latente se considerará $L = 16$. Si bien lo usual es considerar $L \ll D$ (considerando la hipótesis de la variedad), en este ejemplo se considerará que $L > D$ para no colapsar la capacidad de la red neuronal del generador:

```
data_dim, latent_dim = 2, 16

generator = nn.Sequential(
    nn.Linear(latent_dim, 32), nn.ReLU(),
    nn.Linear(32, 64), nn.ReLU(),
    nn.Linear(64, 128), nn.ReLU(),
    nn.Linear(128, data_dim)
)

discriminator = nn.Sequential(
    nn.Linear(data_dim, 128), nn.ReLU(),
    nn.Linear(128, 64), nn.ReLU(),
    nn.Linear(64, 32), nn.ReLU(),
    nn.Linear(32, 1), nn.Sigmoid()
)
```

Por la naturaleza competitiva entre G_θ y D_ϕ, las GANs suelen presentar una dinámica de entrenamiento inestable, lo que ha motivado el uso hiperparámetros que se sabe, principalmente por experiencia empírica, que funcionan bien. En este caso se utilizará Adam [27] con `betas=(0.5, 0.999)`, una elección común sugerida por DCGAN (revisada más abajo):

```
def train(generator, discriminator, latent_dim, iters=5000):

    # Optimizadores:
    generator_optimizer = optim.Adam(generator.parameters(), betas=(0.5,
0.999))
    discriminator_optimizer = optim.Adam(discriminator.parameters(),
betas=(0.5, 0.999))

    for iter in range(iters):

        # Datos de entrenamiento:
        x_true = get_batch()
        x_fake = generator(torch.randn([len(x_true), latent_dim]))

        # Entrenamiento discriminador:
        loss_y1 = torch.log(discriminator(x_true)).mean()
        loss_y0 = torch.log(1-discriminator(x_fake.detach())).mean()
        loss_discriminator = - 1 / 2 * (loss_y1 + loss_y0)
        discriminator_optimizer.zero_grad()
        loss_discriminator.backward()
        discriminator_optimizer.step()

        # Entrenamiento generador (versión saturante):
```

```
        loss_generator = 1 / 2 * torch.log(1-discriminator(x_fake)).mean()
        generator_optimizer.zero_grad()
        loss_generator.backward()
        generator_optimizer.step()
train(generator, discriminator, latent_dim)
```

Una vez entrenado el modelo, se pueden generar nuevas muestras mediante ancestral sampling sobre la red bayesiana $p(z)\,p_\theta(x \mid z)$ sampleando desde $z \sim p(z)$, y luego evaluando $x = G_\theta(z)$:

```
def generate_samples(n_samples=1000):
    z = torch.randn([n_samples, latent_dim])
    samples = generator(z).detach()
    return samples

samples = generate_samples()
plt.figure(figsize=(3, 3))
plt.scatter(samples[:, 0], samples[:, 1], s=1)
plt.show()
```

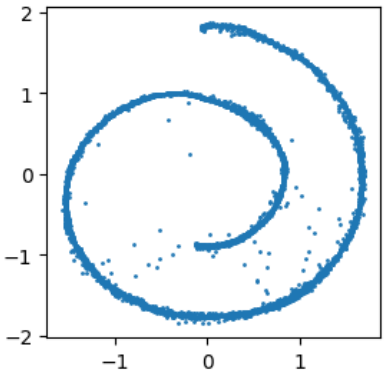

Figura 55: Muestras generadas por la GAN entrenada.

Se observa que el generador aprendió a reproducir, aproximadamente, la distribución $p_{\text{data}}(x)$ a pesar de no haber modelado explícitamente $p_\theta(x)$.

Observación 1. En la implementación anterior se incluyó una sigmoide en la salida del discriminador para que su rango sea $[0, 1]$. Una práctica más estable numéricamente es eliminar la sigmoide y trabajar directamente con los logits, usando `nn.BCEWithLogitsLoss` para el cálculo de la pérdida. La equivalencia con esta función de pérdida es directa al identificar la entropía cruzada (binaria) en la verosimilitud $\log q_\phi(y|x) = y \log D_{\phi(x)} + (1 - y) \log\big(1 - D_{\phi(x)}\big)$, por lo que:

$$\mathcal{L}_{\text{GAN}}(\theta, \phi) = -\mathbb{E}_x \left[\text{CE}\big(p_{\text{train}}(y|x), q_\phi(y|x)\big) \right]$$

Notar que esto es análogo a lo que se realiza con los modelos de lenguaje, donde usualmente no se incluye un módulo `nn.Softmax` en la salida del Transformer y se utiliza `nn.CrossEntropyLoss` para el equivalencia con la verosimilitud de una distribución categórica sobre el vocabulario.

7.1.4 Generación condicional

Al igual que toda red bayesiana, la formulación anterior se puede extender al caso condicional considerando una variable adicional $c \in \mathcal{C}$ que guíe la generación. La red bayesiana de una **GAN condicional** (CGAN) [57] se factoriza como $p_\theta(x, z \mid c) = p(z)\, p_\theta(x \mid c, z)$, donde implícitamente se ha asumido que $z \perp c$ (i.e., el prior latente es independiente del factor condicionante). Además, tanto el generador, $G_\theta : \mathcal{C} \times \mathbb{R}^L \to \mathbb{R}^D$, como el discriminador, $D_\phi : \mathcal{C} \times \mathbb{R}^D \to [0, 1]$, ahora reciben la condición $c \in \mathcal{C}$ como entrada adicional. Con estos cambios, la función objetivo de una CGAN es:

$$\mathcal{L}_{\text{CGAN}}(\theta, \phi) \coloneqq \mathbb{E}_{p_{\text{data}}(c)\, p_{\text{data}}(x \mid c)} \big[\log D_\phi(c, x)\big] + \mathbb{E}_{p(c)\, p(z)} \big[\log\big(1 - D_\phi(c, G_\theta(c, z))\big)\big],$$

donde $p(c)$ es una distribución sobre $\mathcal{C}$ que no es modelada ni aprendida (los valores de c son sampleados durante el entrenamiento). La forma en la que se inyecta c en las redes neuronales depende de su naturaleza:

- **Etiqueta de clase $c \in \{1, \ldots, K\}$.** Lo más simple es concatenar el valor de c a las entradas. Una opción más expresiva es utilizar una matriz de embedding $E \in \mathcal{M}_{K,D'}(\mathbb{R})$, de forma análoga a los modelos de lenguaje, donde la c-ésima fila de E es un vector aprendido asociado a la clase c. Para evitar parámetros adicionales se puede usar el vector one-hot de cada clase como embedding fijo.
- **Texto $c \in \mathcal{V}^*$.** Una opción simple y efectiva es pasar la secuencia de tokens por un modelo de embeddings preentrenado como CLIP, BERT o T5, y utilizar el vector resultante como entrada adicional al generador. Una alternativa más sofisticada, utilizada por modelos como Stable Diffusion [3], consiste en inyectar la secuencia de embeddings mediante atención cruzada en bloques específicos de la red.
- **Imagen $c \in \mathbb{R}^{C \times H \times W}$.** Para tareas image-to-image (e.g., colorización, super-resolución, inpainting) es usual utilizar arquitecturas especializadas como la U-Net (revisada más abajo). Además, para mezclar texto con imagen, se pueden usar arquitecturas más sofisticadas como la usada por FLUX (modelo de flow matching), donde se trabajan tokens de imagen de forma similar.

7.1.5 Implementación para MNIST

A modo de ejemplo, se implementará una CGAN sobre el dataset MNIST, donde la condición $c \in \{0, ..., 9\}$ será el dígito asociado a la imagen. Se comenzará instanciando el dataset con las imágenes normalizadas al intervalo $[-1, 1]$ (otra recomendación de DCGAN):

```
transform = transforms.Compose([
    transforms.ToTensor(),  # normalización [0, 1].
    transforms.Normalize((0.5,), (0.5,))  # normalización [-1, 1].
])

dataset = datasets.MNIST(root='data', train=True, download=True,
transform=transform)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True,
drop_last=True, num_workers=2)

# Ejemplo:
fig, axes = plt.subplots(1, 10, figsize=(10, 1.2))
for cond in range(10):
    idx = (dataset.targets == cond).nonzero()[0].item()
    x, cond_true = dataset[idx]
    img = x.permute(1, 2, 0) * 0.5 + 0.5  # desnormalización a [0, 1].
    axes[cond].imshow(img, cmap='gray_r')
    axes[cond].set_title(cond_true)
    axes[cond].axis('off')
plt.show()
```

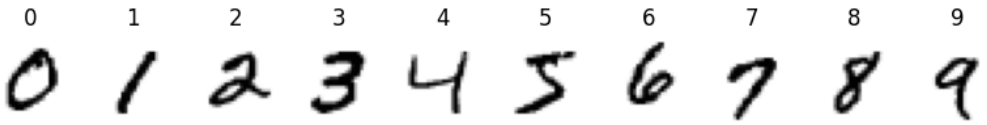

Figura 56: Muestras de MNIST con sus respectivas etiquetas.

Como redes neuronales se utilizarán MLPs simples, dejando el estudio de las arquitecturas convolucionales para la siguiente sección. Para el generador:

```
class Generator(nn.Module):

    def __init__(self, data_dim, n_classes, cond_embed_dim, latent_dim):
        super().__init__()
        self.cond_embed = nn.Embedding(n_classes, cond_embed_dim)
        self.net = nn.Sequential(
            nn.Linear(latent_dim + cond_embed_dim, 256),
nn.BatchNorm1d(256), nn.LeakyReLU(0.2),
```

```
            nn.Linear(256, 512), nn.BatchNorm1d(512), nn.LeakyReLU(0.2),
            nn.Linear(512, 1024), nn.BatchNorm1d(1024), nn.LeakyReLU(0.2),
            nn.Linear(1024, data_dim), nn.Tanh()
        )
        self.apply(self._init_weights)

    @staticmethod
    def _init_weights(m):
        if isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, 0.0, 0.02)
            nn.init.zeros_(m.bias)

    def forward(self, z, cond):
        cond = self.cond_embed(cond)
        input = torch.cat([z, cond], dim=1)
        return self.net(input)
```

Notar que se utilizaron heurísticas como inicialización de parámetros y uso de capas de normalización. Como se verá en las siguientes secciones, estas modificaciones permiten tener un entrenamiento más estable.

Para el discriminador se utilizará una red análoga, donde no se incluye `nn.Softmax` en la salida para trabajar con `nn.BCEWithLogitsLoss`:

```
class Discriminator(nn.Module):

    def __init__(self, data_dim, n_classes, cond_embed_dim):
        super().__init__()
        self.cond_embed = nn.Embedding(n_classes, cond_embed_dim)
        self.net = nn.Sequential(
            nn.Linear(data_dim + cond_embed_dim, 1024), nn.LeakyReLU(0.2),
nn.Dropout(0.3),
            nn.Linear(1024, 512), nn.LeakyReLU(0.2), nn.Dropout(0.3),
            nn.Linear(512, 256), nn.LeakyReLU(0.2), nn.Dropout(0.3),
            nn.Linear(256, 1)
        )
        self.apply(self._init_weights)

    @staticmethod
    def _init_weights(m):
        if isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, 0.0, 0.02)
            nn.init.zeros_(m.bias)
```

```
    def forward(self, x, cond):
        cond = self.cond_embed(cond)
        input = torch.cat([x.flatten(1), cond], dim=1)
        return self.net(input)
```

Para el entrenamiento, ahora se utilizará la función de pérdida no saturante (recomendación de la GAN original). Además, se utilizará label smoothing para el discriminador (similar a lo hecho en ARMs), lo que ayuda a evitar la saturación y mejorar la estabilidad del entrenamiento:

```
def train(generator, discriminator, latent_dim, epochs=50,
label_smoothing=0.9, g_steps=2):

    generator.to(DEVICE).train()
    discriminator.to(DEVICE).train()

    # Optimizadores:
    generator_optimizer = optim.Adam(generator.parameters(), lr=2e-4,
betas=(0.5, 0.999))
    discriminator_optimizer = optim.Adam(discriminator.parameters(),
lr=2e-4, betas=(0.5, 0.999))

    criterion = nn.BCEWithLogitsLoss()
    losses = {'generator': [], 'discriminator': []}

    try:
        for epoch in tqdm(range(epochs)):
            for x_true, cond_true in dataloader:
                batch_size = len(x_true)
                ones = torch.ones(batch_size, 1, device=DEVICE)
                zeros = torch.zeros(batch_size, 1, device=DEVICE)

                # Datos de entrenamiento:
                # - True:
                cond_true = cond_true.to(DEVICE)
                x_true = x_true.to(DEVICE)
                # - Fake:
                z = torch.randn(batch_size, latent_dim, device=DEVICE)
                x_fake = generator(z, cond_true)

                # Entrenamiento discriminador:
                # - Loss:
                loss_d_real = criterion(discriminator(x_true, cond_true),
ones * label_smoothing)
```

```
                loss_d_fake = criterion(discriminator(x_fake.detach(),
cond_true), zeros)
                loss_discriminator = 1/2 * (loss_d_real + loss_d_fake)
                # - Step:
                discriminator_optimizer.zero_grad()
                loss_discriminator.backward()
                discriminator_optimizer.step()

                # Entrenamiento generador (versión no saturante):
                for _ in range(g_steps):
                    z = torch.randn(batch_size, latent_dim, device=DEVICE)
                    x_fake = generator(z, cond_true)
                    loss_generator = criterion(discriminator(x_fake,
cond_true), ones)
                    generator_optimizer.zero_grad()
                    loss_generator.backward()
                    generator_optimizer.step()

                # Log:
                losses['generator'].append(loss_generator.item())
                losses['discriminator'].append(loss_discriminator.item())

    except KeyboardInterrupt:
        print('Entrenamiento interrumpido.')

    # Gráfico de entrenamiento:
    plt.plot(losses['generator'], label='Generador', alpha=0.7)
    plt.plot(losses['discriminator'], label='Discriminador', alpha=0.7)
    plt.xlabel('Step');
    plt.ylabel('Loss');
    plt.legend();
    plt.grid(alpha=0.3)
    plt.show()
```

Con esto, se pueden instanciar las redes neuronales y realizar el entrenamiento:

```
data_dim = dataset[0][0].numel()
n_classes = len(dataset.classes)
cond_embed_dim = 16
latent_dim = 64

generator = Generator(data_dim, n_classes, cond_embed_dim, latent_dim)
discriminator = Discriminator(data_dim, n_classes, cond_embed_dim)
```

```
train(generator, discriminator, latent_dim)
```

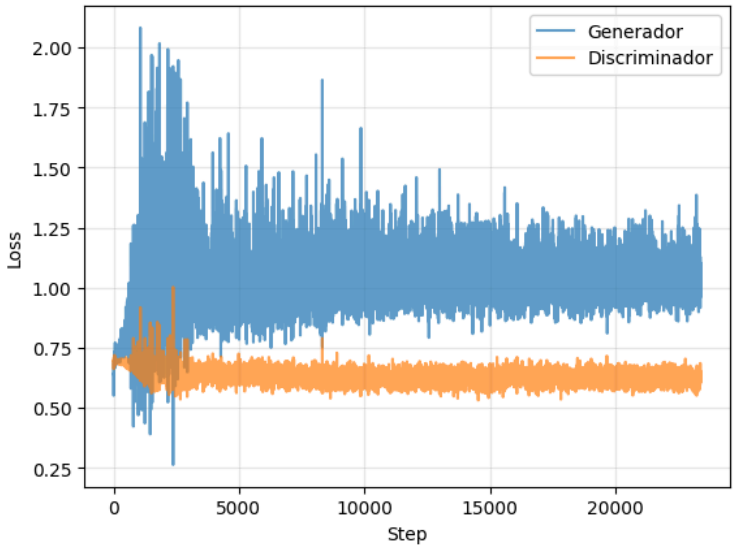

Figura 57: Dinámica de evolución de la CGAN.

Se observa que la dinámica de entrenamiento de la GAN es mucho más inestable e impredecible que la dinámica de, por ejemplo, los modelos autorregresivos. Más aún, la función de pérdida de ambos modelos no parece entregar ningún indicador obvio de cuándo detener el entrenamiento, o cómo detectar overfitting.

Para la generación, se utiliza la misma técnica de ancestral sampling usada en el caso incondicional, solo que además se debe incluir una condición $c \in \mathcal{C}$ en la red generadora:

```
@torch.no_grad()
def generate_samples(cond, n_samples):
    generator.eval()
    z = torch.randn(n_samples, latent_dim, device=DEVICE)
    cond = torch.full((n_samples,), cond, dtype=torch.long, device=DEVICE)
    samples = generator(z, cond)
    return samples.cpu()
```

Usando la función anterior, se generarán algunas muestras para cada etiqueta de clase:

```
n_samples = 15

fig, axes = plt.subplots(n_classes, n_samples, figsize=(n_samples,
n_classes))
for cond in range(n_classes):
    samples = generate_samples(cond=cond, n_samples=n_samples)
```

```
    samples = (samples * 0.5 + 0.5).clamp(0, 1).view(len(samples), 28, 28)
    for j in range(n_samples):
        axes[cond, j].imshow(samples[j], cmap='gray_r')
        axes[cond, j].axis('off')
    axes[cond, 0].set_ylabel(str(cond), fontsize=10)
plt.tight_layout()
plt.show()
```

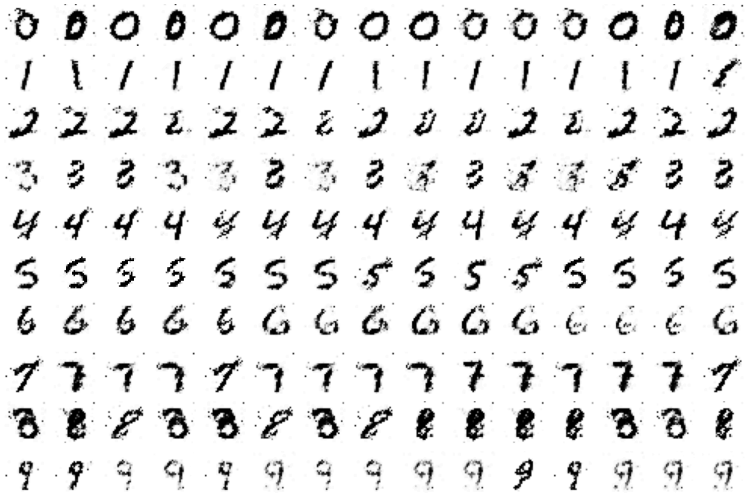

Figura 58: Muestras generadas por la CGAN para cada etiqueta de clase.

En la siguiente sección se estudiarán arquitecturas neuronales especializadas para imágenes, las cuales combinan distintas técnicas de diseño para inducir sesgos útiles en este tipo de datos. Como se verá, si bien las GANs tienen la capacidad de llegar a generar imágenes hiperrealistas, la principal limitación de este tipo de modelos es su inestabilidad durante el entrenamiento. Debido a esto, muchos de los trabajos en GANs consistieron principalmente en introducir modificaciones de arquitectura y heurísticas de optimización.

7.2 Redes convolucionales transpuestas

El proceso generativo de una GAN para imágenes consiste en transformar una muestra latente de baja dimensión, $z \in \mathbb{R}^L$, en una imagen de alta dimensión, $x = G_\theta(z) \in \mathbb{R}^D$, donde $D = C \times H \times W$ para una imagen de resolución $H \times W$ y C canales. Cuando $L \ll D$, este mapeo requiere un aumento progresivo de resolución a través de la red neuronal G_θ. Si bien esto se podría implementar con capas fully connected, este enfoque no escala bien a altas resoluciones ni aprovecha la estructura espacial de las imágenes. La operación natural que permite realizar este upsampling preservando la estructura espacial es la **convolución**

transpuesta, también llamada **convolución con stride fraccionario** (o mal llamada *deconvolución*).

7.2.1 Convolución como operador lineal

Antes de estudiar las convoluciones transpuestas, se recordará el concepto de convolución[2] usado en una red convolucional simple. Para esto, se considerará el caso simplificado unidimensional ($D = 1$) con un solo canal.

Sea $x = \left(x_1, ..., x_{D_{\text{in}}}\right) \in \mathbb{R}^{D_{\text{in}}}$ una entrada y $w = (w_1, ..., w_K) \in \mathbb{R}^K$ un **filtro convolucional** de tamaño $K \geq 1$. La convolución $\text{Conv} : \mathbb{R}^{D_{\text{in}}} \to \mathbb{R}^{D_{\text{out}}}$ con **stride** $S \geq 1$ y **padding** $P \geq 0$ produce un vector $y \in \mathbb{R}^{D_{\text{out}}}$ con componentes

$$y_j = \sum_{k=1}^{K} w_k \, x_{(j-1)S+k-P}, \quad j \in \{1, ..., D_{\text{out}}\},$$

donde $D_{\text{out}} = \left\lfloor \frac{D_{\text{in}}+2P-K}{S} \right\rfloor + 1$ y se considera $x_i = 0$ para $i \notin \{1, ..., D_{\text{in}}\}$ (zero-padding implícito). Se puede observar que la convolución es un operador lineal (i.e., $\text{Conv}(\alpha u + \beta v) = \alpha \, \text{Conv}(u) + \beta \, \text{Conv}(v)$), por lo que existe una matriz $A \in \mathcal{M}_{D_{\text{out}}, D_{\text{in}}}(\mathbb{R})$ tal que $y = Ax$. En el caso $S = 1$, $P = 0$, esta matriz tiene estructura de **Toeplitz**. Por ejemplo, para $D_{\text{in}} = 5$, $K = 3$, $S = 1$ y $P = 0$:

$$A = \begin{pmatrix} w_1 & w_2 & w_3 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 \end{pmatrix}.$$

En consecuencia, una red convolucional puede verse como una red fully connected con una matriz de pesos restringida a una estructura específica, lo que reduce drásticamente el número de parámetros (de $D_{\text{out}} D_{\text{in}}$ a K) y además induce sesgos útiles. En particular, dado que se comparten los pesos del kernel w a lo largo de todas las posiciones espaciales de la entrada, la convolución es **equivariante por traslación** (si un trozo de la entrada se traslada, la salida se traslada de la misma forma). Además, cuando la convolución se combina con operaciones de pooling, la red puede volverse **invariante por traslación local** (la salida no cambia si se traslada localmente la entrada). Estos sesgos inductivos son especialmente útiles en visión computacional, donde se asume que los patrones relevantes (bordes, texturas, objetos) aparecen en cualquier posición de la imagen, por lo que la red no debe depender de la ubicación absoluta de estos patrones. De forma similar, este tipo

[2]Más correctamente, esta operación corresponde a la **correlación cruzada**, ya que la convolución contiene el índice desfasado e invertido. En la práctica ambos conceptos son equivalentes a nivel de capacidad de aprendizaje, y el nombre de convolución es el más usual en deep learning.

de convoluciones 1D puede utilizarse para modelar el lenguaje. En este caso, el tamaño del kernel convolucional indica el tamaño de la ventana de contexto del modelo.

En el caso de más dimensión (e.g. imágenes 2D), la operación de convolución se aplica de forma independiente en cada uno de los ejes, por lo que la extensión es directa. Por otro lado, en general las convoluciones operan sobre múltiples canales de entrada (e.g. 3 canales de colores y un canal de transparencia) y producen múltiples canales de salida (cada canal un **feature map**). En estos casos, si la entrada tiene C_{in} canales de entrada, la convolución utiliza un kernel diferente por cada canal, y las contribuciones de todos los canales se suman (elemento a elemento) para producir un único canal de salida[3]. De forma dual, si se necesitan C_{out} canales de salida, se utiliza un bloque independiente de C_{in} kernels por cada canal de salida, y las salidas se concatenan a lo largo de la dimensión de canales. En consecuencia, una capa convolucional estándar necesita aprender $C_{\text{out}} \times C_{\text{in}} \times K$ parámetros (más parámetros de sesgo si se consideran).

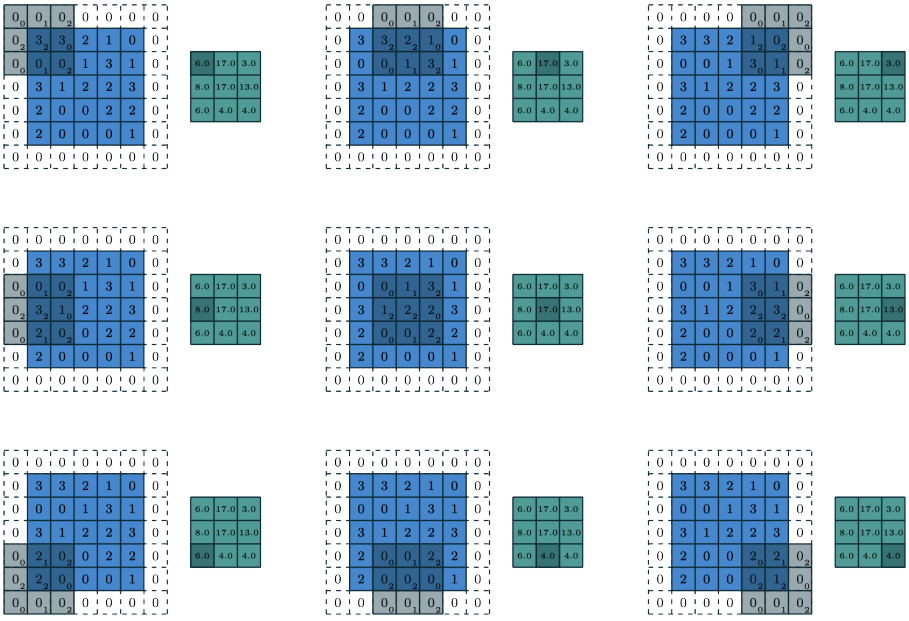

Figura 59: Convolución 2D con $K = 3$, $S = 2$ y $P = 1$ (mismo valor en ambas dimensiones).

7.2.2 Pooling

Las operaciones de **pooling** reducen la resolución espacial de un feature map sin cambiar el número de canales, aplicando una función de agregación sobre ventanas locales de la

[3]Equivalentemente, se puede considerar un único kernel $(D+1)$-dimensional que convolucione con los C_{in} canales de entrada.

entrada. Las dos operaciones más comunes son **max pooling** (retornar el valor máximo dentro de cada ventana) y **average pooling** (retornar el promedio de cada ventana). Por ejemplo, para un feature map de entrada de tamaño $C \times H \times W$, un max pooling 2×2 con $S = 2$ produce una salida de tamaño $C \times \frac{H}{2} \times \frac{W}{2}$, donde cada posición de la salida es el máximo de una ventana 2×2 de la entrada[4]. En arquitecturas típicas, el pooling se utiliza luego de convoluciones 3×3, las cuales no cambian la resolución (e.g., usando $P = 1$, $S = 1$) pero sí el número de canales.

7.2.3 Convolución transpuesta

Dada una convolución directa con matriz asociada $A \in \mathcal{M}_{D_{\text{out}}, D_{\text{in}}}(\mathbb{R})$, la **convolución transpuesta** es el operador lineal $A^\top : \mathbb{R}^{D_{\text{out}}} \to \mathbb{R}^{D_{\text{in}}}$ (i.e., $y = A^\top x$). Considerando que $D_{\text{in}} \geq D_{\text{out}}$ en una convolución tradicional, se observa que la convolución transpuesta es un operador que permite expandir la dimensión de la entrada, al mismo tiempo que se mantienen los sesgos inductivos que caracterizan a la convolución directa. En particular, este tipo de operador permite realizar un aumento de la resolución a medida que se avanza en una red neuronal generadora (como en una GAN o en un VAE).

Interpretando la convolución directa como un mecanismo que *agrega* información de la entrada (cada elemento de la salida recibe contribuciones de K elementos de la entrada), la convolución transpuesta puede interpretarse como un mecanismo que *distribuye* el valor de cada elemento de la entrada sobre K posiciones de la salida. Más aún, se puede ver que la convolución transpuesta con stride S es operacionalmente equivalente a insertar $S - 1$ ceros entre cada par de elementos consecutivos de la entrada y luego aplicar una convolución directa con stride 1 y padding $K - 1 - P$. Esta interpretación explica el nombre **stride fraccionario** (el filtro avanza efectivamente $\frac{1}{S}$ posiciones por cada elemento del input original).

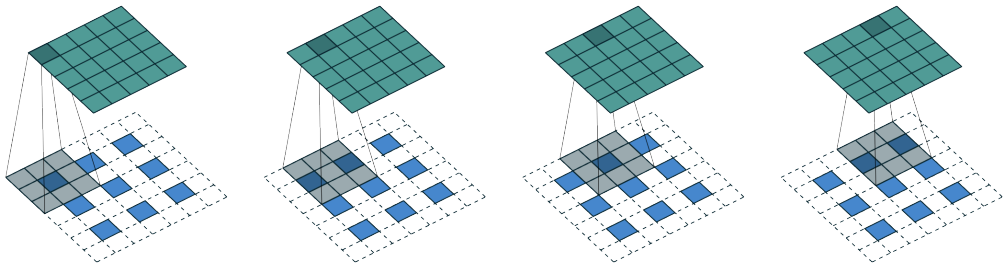

Figura 60: Convolución transpuesta 2D vista como una convolución con stride fraccionario.

[4]En particular, max pooling puede considerarse como un detector de características que se activa cuando al menos un elemento del campo receptivo posee la feature buscada (invarianza local).

Por otra parte, despejando D_{in} desde $D_{\text{out}} = \frac{D_{\text{in}}+2P-K}{S} + 1$ (asumiendo división exacta), es posible obtener la dimensión de salida que produce una convolución transpuesta con kernel de tamaño K, stride S y padding P sobre una entrada de largo D_{in}:

$$D_{\text{out}} = S(D_{\text{in}} - 1) + K - 2P.$$

En arquitecturas generativas las convolución con $K = 4$, $S = 2$, $P = 1$ y la convolución con $K = 2$, $S = 2$, $P = 0$ duplican la resolución espacial (i.e., $D_{\text{out}} = 2D_{\text{in}}$), por lo que son convoluciones transpuestas usuales en redes neuronales generativas.

Artefactos de tablero

Cuando K no es múltiplo de S, el filtro se superpone de forma desigual sobre la salida, lo que provoca que algunas posiciones reciban contribuciones de $\lceil \frac{K}{S} \rceil$ elementos de la entrada, mientras que otras reciben contribuciones de $\lfloor \frac{K}{S} \rfloor$ elementos.

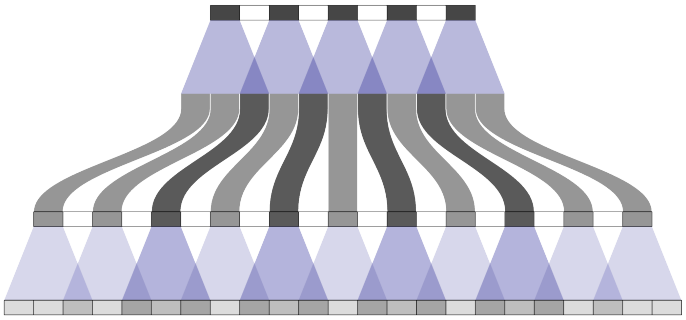

Figura 61: Periodicidad en la cantidad de contribuyentes para cada posición de salida.

Esta periodicidad se manifiesta como un patrón visual de cuadrícula en la imagen generada, conocido como **checkerboard artifact**.

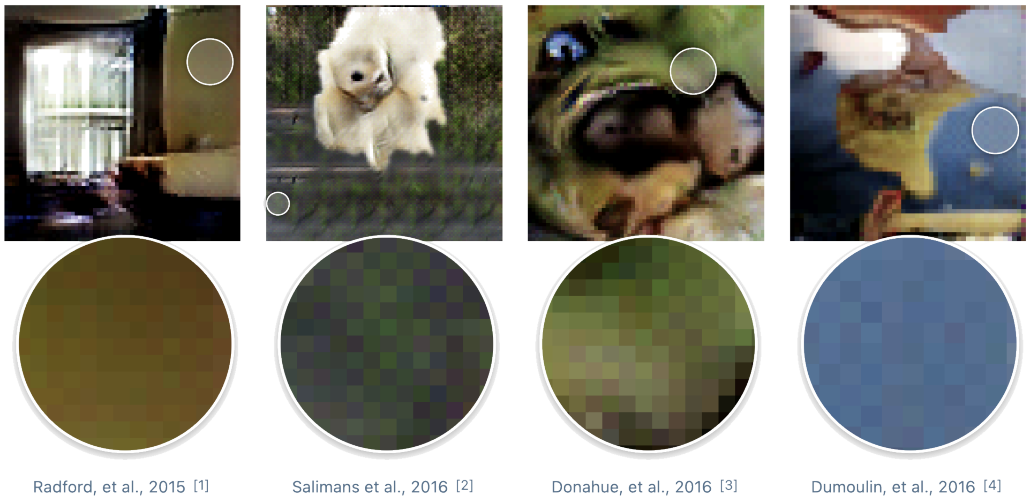

Figura 62: Ejemplos de checkboard artifacts en distintas imágenes generadas.

Existen dos estrategias usuales para evitar estos artefactos:

1. Elegir K múltiplo de S (típicamente $K = 2S$), de modo que todas las posiciones reciban el mismo número de contribuyentes.
2. Reemplazar la convolución transpuesta por una interpolación explícita (e.g., bilineal o nearest neighbor) para aumentar la resolución, seguida de una convolución que cambie la cantidad de canales pero no la resolución (e.g., $S = 1$, $K = 3$, $P = 1$).

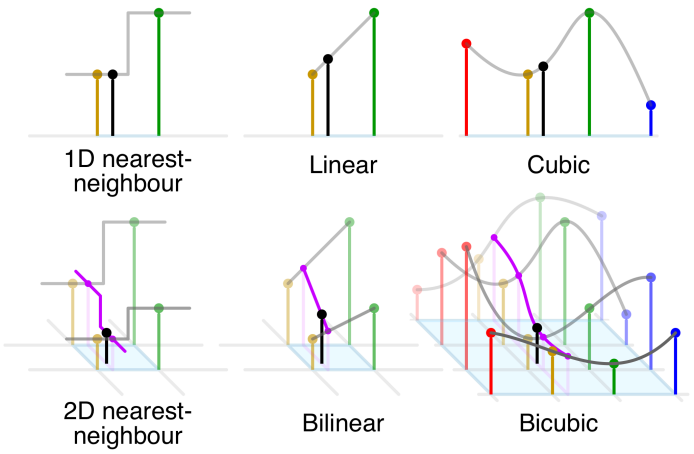

Figura 63: Comparación de interpolaciones en los casos $D = 1$ y $D = 2$.

7.3 Algunas GANs famosas

Las GANs constituyeron el estado del arte para la generación de imágenes entre los años 2014 y 2021, hasta la consolidación de los modelos de difusión. Como se mencionó, una parte importante de los trabajos sobre GANs estuvo enfocada en desarrollar mejores arquitecturas neuronales, con el objetivo de mitigar las inestabilidades del entrenamiento y propagar la información condicional de forma efectiva. Otra línea de investigación se dedicó a modificar la función de entrenamiento original, proponiendo formulaciones más estables o ligadas a divergencias con mejores propiedades geométricas (algunas divergencias serán estudiadas en el siguiente capítulo). Una tercera línea estuvo enfocada en obtener resultados teóricos sobre las GANs, como propiedades de convergencia y conexiones con teoría de juegos.

El siguiente diagrama muestra un orden temporal de los modelos tipo GAN más relevantes, junto con otros desarrollos contemporáneos como contexto histórico:

```
gantt
    dateFormat DD-MM-YYYY
    axisFormat %Y

    section Otros
        BatchNorm :milestone, 11-02-2015, 0d
        UNet :milestone, 18-05-2015, 0d
        ResNet :milestone, 10-12-2015, 0d
        LayerNorm :milestone, 21-06-2016, 0d
        GroupNorm :milestone, 22-03-2018, 0d

    section ARMs
        Seq2seq :milestone, 07-09-2014, 0d
        Transformer :milestone, 12-06-2017, 0d
        GPT :milestone, 15-05-2018, 0d
        GPT 2 :milestone, 14-02-2019, 0d

    section GANs
        GAN :milestone, 10-06-2014, 0d
        CGAN :milestone, 06-11-2014, 0d
        DCGAN :milestone, 19-11-2015, 0d
        WGAN :milestone, 26-01-2017, 0d
        Pix2Pix :milestone, 21-11-2016, 0d
        CycleGAN :milestone, 30-03-2017, 0d
        ProGAN :milestone, 27-10-2017, 0d
        SAGAN :milestone, 21-05-2018, 0d
        StyleGAN :milestone, 12-12-2018, 0d
```

En esta sección se revisarán algunas arquitecturas y modelos tipo GAN que se consideran relevantes en el campo. Si bien hay muchos otros trabajos importantes, la elección de estos modelos es principalmente debido a que algunas de sus propuestas fueron hereadadas a arquitecturas más modernas usadas en modelos de difusión (e.g. el uso de la U-Net con módulos de atención). Si bien no se seguirá el orden cronológico, cada arquitectura se motivará como respuesta a una limitación concreta de la anterior. En particular, DCGAN partirá dando reglas básicas para estabilizar el entrenamiento de GANs convolucionales, SAGAN incorporará dependencias globales para mejorar la coherencia geométrica, ProGAN permitirá escalar a mayores resoluciones mediante un entrenamiento progresivo, y StyleGAN reorganizará el generador para desacoplar factores de variación de alto y bajo nivel.

7.3.1 Deep convolutional GAN

La GAN original utilizaba redes fully connected, las cuales no aprovechan la estructura espacial de las imágenes y no escalan a resoluciones altas. **Deep convolutional GAN (DCGAN)** [44] reemplaza estas redes por arquitecturas convolucionales profundas y propone, mediante una extensa exploración empírica, un conjunto de restricciones arquitectónicas que estabilizan el entrenamiento. La contribución central de este trabajo es haber definido reglas generales para el diseño de arquitecturas para GANs. Adicionalmente, DCGAN mostró que tanto el generador como el discriminador aprenden representaciones semánticamente estructuradas, lo cual es destacable considerando que el entrenamiento es, al igual que en los modelos de lenguaje, autosupervisado (las etiquetas $y \in \{0, 1\}$ se obtienen automáticamente).

Restricciones arquitectónicas

Una técnica ampliamente utiliza en GANs es **batch normalization (BN)** [50]. Este módulo normaliza las pre-activaciones de cada neurona a media nula y varianza unitaria, y luego les aplica una transformación afín con parámetros aprendibles $\alpha_c, \beta_c \in \mathbb{R}$, con cada par de parámetros aprendido de forma independiente para cada canal.

Si $B = (x^1, ..., x^N) \in \mathbb{R}^{N \times C \times H \times W}$ es un batch de imágenes[5], durante el entrenamiento BN devuelve un tensor del mismo tamaño definido componente a componente por

$$\mathrm{bn}(B)^n_{c,h,w} := \alpha_c \frac{x^n_{c,h,w} - \mu_c}{\sqrt{\sigma_c^2 + \varepsilon}} + \beta_c,$$

[5]En general, un objeto tipo imagen es un tensor 3D formado por un conjunto de mapas de características 2D, donde cada mapa de características corresponde a un canal de la imagen. En particular, se le dirá imagen a la entrada y salida de cualquier bloque convolucional 2D.

para todo $(n, c, h, w) \in \{1, ..., N\} \times \{1, ..., C\} \times \{1, ..., H\} \times \{1, ..., W\}$, donde

$$\mu_c \coloneqq \frac{1}{NHW} \sum_{n=1}^{N} \sum_{h=1}^{H} \sum_{w=1}^{W} x^n_{c,h,w} \in \mathbb{R}, \qquad \sigma_c^2 \coloneqq \frac{1}{NHW} \sum_{n=1}^{N} \sum_{h=1}^{H} \sum_{w=1}^{W} \left(x^n_{c,h,w} - \mu_c\right)^2 \geq 0,$$

y $\varepsilon > 0$ es un hiperparámetro de estabilidad numérica. En inferencia, los estadísticos del batch, $(\mu_c, \sigma_c^2) \in \mathbb{R}^2$, $c \in \{1, ..., C\}$, se sustituyen por estimaciones globales $(\hat{\mu}_c, \hat{\sigma}_c^2) \in \mathbb{R}^2$ acumuladas durante el entrenamiento mediante una media móvil exponencial[6].

DCGAN propone construir una GAN para imágenes utilizando redes convolucionales con las siguientes cinco restricciones:

1. **Batch normalization en G_θ y D_ϕ:** se debe aplicar BN a todas las capas intermedias de G_θ y D_ϕ, excepto en la salida de G_θ (cuya distribución de píxeles no debe ser forzada a ser gaussiana) y en la entrada de D_ϕ (donde mezclar estadísticas de muestras reales y sintéticas en la primera capa degrada el clasificador).
2. **Sustitución de pooling por convoluciones con stride:** el downsampling en D_ϕ se debe realizar mediante convoluciones con $S = 2$ en vez de usar max-pooling, mientras que el upsampling en G_θ se debe realizar mediante convoluciones transpuestas con $S = 2$. En ambos casos, estos cambios le permite a la red aprender sus propias funciones de muestreo (kernels) en lugar de usar operadores fijos.
3. **Eliminación de capas fully connected ocultas:** en las primeras GANs, tanto G_θ como D_ϕ eran redes fully connected que trataban la imagen como un vector plano. En DCGAN el código latente $z \in \mathbb{R}^L$ se proyecta linealmente a un tensor en $\mathbb{R}^{C \times H \times W}$ mediante una única capa lineal seguida de un reshape, y el resto de la arquitectura opera exclusivamente con convoluciones.
4. **Activaciones en G_θ:** en el generador, todas las capas ocultas deben usar ReLU, excepto la capa de salida que debe usar tanh, cuyo rango $(-1, 1)$ es simétrico alrededor de cero[7]. Se prefiere tanh sobre la sigmoide estándar debido a que tanh tiene gradientes más fuertes cerca de los valores extremos, lo que facilita el aprendizaje de píxeles con intensidades altas o bajas. Además, el rango simétrico $(-1, 1)$ es más natural para representar imágenes normalizadas que el rango asimétrico $(0, 1)$ de la sigmoide.
5. **Activaciones en D_ϕ:** en el discriminador, todas las capas deben usar $\text{LeakyReLU}_\alpha(x) \coloneqq \begin{cases} x & \text{si } x \geq 0 \\ \alpha x & \text{si } x < 0 \end{cases}$ en lugar de ReLU (LeakyReLU con $\alpha = 0$). Esta elección garantiza gradientes no nulos en todo $\mathbb{R}$, lo que es crítico al inicio del entrenamiento, cuando G_θ produce imágenes de muy mala calidad.

[6] Tras cada batch de entrenamiento se actualiza $\hat{\mu}_c \leftarrow \lambda \hat{\mu}_c + (1 - \lambda)\mu_c$, donde $\lambda \in [0, 1]$ es un hiperparámetro de momentum. De forma análoga se actualiza $\hat{\sigma}_c^2$.

[7] Las imágenes de entrenamiento se deben normalizar al mismo rango.

Hiperparámetros de Adam

El algoritmo de optimización **Adam** [27] mantiene dos estimaciones acumuladas para cada parámetro $\theta \in \mathbb{R}^P$, las cuales corresponden al primer y segundo momento de los gradientes. Estas estimaciones se actualizan en cada step de entrenamiento mediante

$$m_t = \beta_1 \, m_{t-1} + (1 - \beta_1) \, g_t$$
$$v_t = \beta_2 \, v_{t-1} + (1 - \beta_2) \, g_t^2,$$

donde $g_t \in \mathbb{R}^P$ es el gradiente en el paso t y el cuadrado en $g_t^2 = g_t \odot g_t \in \mathbb{R}^P$ se realiza coordenada a coordenada. El parámetro $\beta_1 \in [0, 1)$ controla cuánta memoria mantiene Adam de la dirección media de los gradientes pasados (mayor β_1 implica más momentum), mientras que $\beta_2 \in [0, 1)$ controla la memoria para estimar la varianza. Tras aplicar corrección de sesgo, $\hat{m}_t = \frac{m_t}{1-\beta_1^t} \in \mathbb{R}^P$ y $\hat{v}_t = \frac{v_t}{1-\beta_2^t} \in \mathbb{R}^P$, la actualización de parámetros que realiza Adam es

$$\theta_t = \theta_{t-1} - \alpha \, \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

con las operaciones de raíz y división aplicada coordenada a coordenada. Notar que la corrección de sesgo es necesaria ya que, al inicio del entrenamiento (t pequeño), las estimaciones m_t y v_t están sesgadas hacia cero (ya que se inicializan en cero y los factores $\beta_1, \beta_2 \in [0, 1)$ son cercanos a 1). Dividir por $1 - \beta_i^t$ cancela este sesgo, ya que $(1 - \beta^t) \to 1$ cuando $t \to \infty$. La división $\frac{\widehat{m}_t}{\sqrt{\hat{v}_t}}$ normaliza cada componente del gradiente por su desviación estándar histórica, implementando una tasa de aprendizaje adaptativa por parámetro que acelera la convergencia en direcciones con gradientes consistentes, y frena el aprendizaje en direcciones ruidosas.

Si bien los valores por defecto de Adam son $(\beta_1, \beta_2) = (0.9, 0.999)$, DCGAN propone utilizar $\beta_1 = 0.5$. Durante el entrenamiento de una GAN, las redes neuronales G_θ y D_ϕ son adaptadas de manera adversativa, por lo que los gradientes del generador suelen cambiar de signo con frecuencia (ya que una dirección favorable en el paso t puede volverse perjudicial para G_θ en el paso $t + 1$ si el discriminador ha cambiado). Con $\beta_1 = 0.9$, Adam promedia gradientes a lo largo de aproximadamente 10 pasos[8] (ya que $\frac{1}{1-0.9} = 10$), por lo que, en una GAN, la actualización estándar de Adam muchas veces seguiría direcciones obsoletas, produciendo oscilaciones. DCGAN reduce β_1 a 0.5 ya que esta elección acorta

[8]La cantidad de pasos de memoria efectiva de una media móvil exponencial (EMA) con parámetro β se puede estimar como $\frac{1}{1-\beta}$. Esto proviene de observar que en la actualización $m_t = \beta m_{t-1} + (1 - \beta) g_t$, el peso del gradiente en el paso $t - k$ es proporcional a $\beta^k(1 - \beta)$, que decae exponencialmente. Luego, la suma acumulada de pesos, $\sum_{k=0}^{\infty} \beta^k = \frac{1}{1-\beta}$, entrega una medida del horizonte temporal efectivo de la EMA.

la memoria efectiva a aproximadamente $\frac{1}{1-0.5} = 2$ pasos, haciendo a Adam más reactivo a la información reciente de los gradientes.

Propiedades emergentes del espacio latente

El trabajo de DCGAN mostró que las representaciones aprendidas por una GAN, aun en un escenario autosupervisado, capturan propiedades semánticas no triviales. En particular, al utilizar las activaciones convolucionales intermedias del discriminador como entrada para un clasificador lineal entrenado sobre CIFAR-10 y SVHN, DCGAN logró obtener resultados competitivos respecto a métodos clásicos de aprendizaje no supervisado.

Más aún, en DCGAN observaron que el espacio latente exhibe una estructura algebraica con efectos semánticos. Más precisamente, si $z_0, z_1 \in \mathbb{R}^L$ son dos muestras del prior $p_\theta(z)$, la *curva de imágenes* dada por $t \mapsto x_t = G_\theta((1-t)z_0 + tz_1)$, $t \in [0,1]$, corresponde a una **interpolación semántica** entre las imágenes $G_\theta(z_0) \in \mathbb{R}^D$ y $G_\theta(z_1) \in \mathbb{R}^D$, en la cual los atributos relevantes de la imagen varían suavemente.

Figura 64: Interpolación latente en DCGAN. Imagen obtenida desde [44].

En particular, esta propiedad del espacio latente de una GAN permite realizar aritmética vectorial en el espacio latente con efectos semánticos interpretables, de forma similar a como se pueden operar los vectores de embedding de un vocabulario de texto. Por ejemplo, si $\hat{z}_{\text{hombre, con lentes}}$, $\hat{z}_{\text{hombre, sin lentes}}$ y $\hat{z}_{\text{mujer, sin lentes}}$ son promedios de vectores latentes que generan imágenes con los atributos correspondientes, entonces el vector latente

$$\hat{z}_{\text{mujer, con lentes}} = \hat{z}_{\text{hombre, con lentes}} - \hat{z}_{\text{hombre, sin lentes}} + \hat{z}_{\text{mujer, sin lentes}}$$

(y una vecindad suya) genera imágenes de mujeres con lentes, sugiriendo que G_θ aprende representaciones con cierto grado de **linealidad semántica**.

Este tipo de modificación de atributos será implementado al estudiar VAEs, donde además se verá como aumentar y disminuir la intensidad del atributo a modificar.

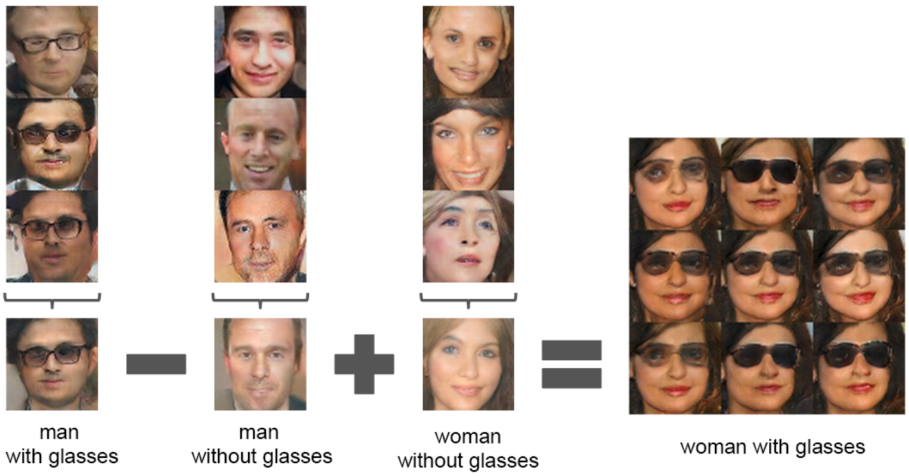

Figura 65: Aritmética vectorial en el espacio latente de DCGAN. Imagen obtenida desde [44].

7.3.2 Self-attention GAN

Si bien DCGAN produce resultados notables a 64×64, a mayor resolución presenta varias limitaciones que motivan las siguientes arquitecturas. En particular, las arquitecturas convolucionales clásicas presentan dificultades al modelar patrones con estructura geométrica compleja, donde las partes de un objeto deben mantener coherencia global (e.g., imágenes de animales). Esta limitación surge porque las convoluciones procesan información de manera local, restringiendo el modelado de dependencias de largo alcance y, si bien apilar varias capas convolucionales expande el campo receptivo, esto incrementa significativamente la profundidad y el costo computacional.

Self-attention GAN (SAGAN) [58] introduce un módulo de self-attention que captura dependencias entre cualquier par de posiciones del feature map con un único bloque, generando imágenes a 128×128 en la tarea de generación condicional (por clase) sobre ImageNet.

Módulo de self-attention

Para tratar un feature map convolucional como una secuencia, se puede considerar cada pixel como un token, donde la dimensión de embedding del token correspondería a la cantidad de canales. Si $x_1, ..., x_N \in \mathbb{R}^C$ son los $N = HW$ vectores asociados a las posiciones espaciales (con C canales), el módulo de self-attention en SAGAN parte realizando tres proyecciones lineales implementadas como convoluciones 1×1:

$$q_n = W_Q\, x_n, \quad k_n = W_K\, x_n, \quad v_n = W_V\, x_n, \quad \forall n \in \{1, ..., N\},$$

donde $W_Q, W_K, W_V \in \mathcal{M}_{\overline{C},C}(\mathbb{R})$ son matrices de parámetros y $\overline{C} \ll C$ es la dimensión del cabezal. La similitud entre las posiciones x_i y x_j se calcula mediante el producto interno $s_{ij} = \langle q_i, k_j \rangle$, y los pesos de atención se obtienen aplicando softmax por filas sobre la matriz de scores:

$$a_{ij} = \frac{e^{s_{ij}}}{\sum_{n=1}^{N} e^{s_{in}}}.$$

Luego vector de salida en la posición i agrega los valores ponderados y aplica una proyección de salida $W_O \in \mathcal{M}_{C,\overline{C}}(\mathbb{R})$ al único cabezal de atención para volver a la dimensión original:

$$o_i = W_O \sum_{j=1}^{N} a_{ij} v_j \in \mathbb{R}^C.$$

Finalmente, la salida se inyecta como una conexión residual con compuerta:

$$y_i = \gamma\, o_i + x_i.$$

El parámetro de compuerta, $\gamma \in \mathbb{R}$, es un escalar aprendible inicializado en cero. Con $\gamma = 0$, la salida del módulo de atención es $y_i = x_i$, por lo que la red se reduce a su contraparte puramente convolucional al inicio del entrenamiento. A medida que γ crece durante la optimización, el módulo incorpora información de largo alcance de forma gradual, evitando perturbaciones no locales que desestabilizarían el entrenamiento en las iteraciones iniciales.

Observación 2. A diferencia de la scaled dot-product attention de [14], SAGAN no divide los logits por $\sqrt{\overline{C}}$ (el control de magnitudes se delega a la normalización espectral[9]), y tampoco se usa layer normalization dentro del módulo.

Para elegir donde colocar los mecanismos de atención, notar que en resoluciones menores, el campo receptivo de la convolución ya cubre prácticamente todo el feature map, mientras que en resoluciones mayores el costo cuadrático de la atención se vuelve prohibitivo. Por este motivo, los autores de SAGAN sugieren insertar el módulo en el feature map de 64×64 ya que esta resolución ofrece la mejor relación costo-beneficio.

[9]Dada una capa lineal con matriz de pesos $W \in \mathcal{M}_{m,n}(\mathbb{R})$, la normalización espectral reemplaza W por $\frac{W}{\sigma_1(W)}$, donde $\sigma_1(W) = \max_{\|u\|=1} \|Wu\|$ es el mayor valor singular de W, el cual puede ser estimado eficientemente.

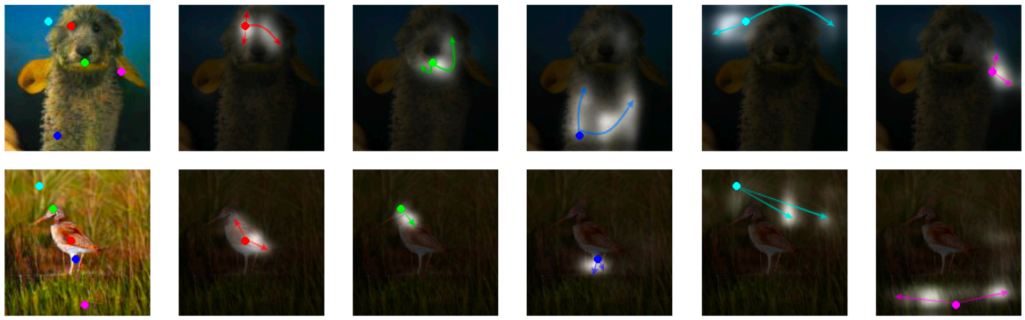

Figura 66: Visualización de los mapas de atención en SAGAN. Imagen obtenida desde [58].

Empíricamente, incorporar un mecanismo atención produce un aumento sustancial en la calidad de las imágenes generadas, especialmente en clases con patrones geométricos complejos. Debido a esto, el mecanismo de atención introducido por SAGAN ha influido notablemente en arquitecturas posteriores, como la U-Net usadas en modelos de difusión, donde se combinan bloques convolucionales con bloques de self-attention en las resoluciones intermedias siguiendo la idea de SAGAN.

Hinge aversarial loss

En la formulación estándar de una GAN, el discriminador produce una probabilidad $D_\phi(x) \in [0,1]$ y se entrena siguiendo el enfoque de máxima verosimilitud. En SAGAN, $D_\phi : \mathbb{R}^D \to \mathbb{R}$ produce un puntaje real sin restricción a $[0,1]$, y se entrena con la **hinge adversarial loss**:

$$\mathcal{L}_D(\phi) = -\mathbb{E}_{p_{\text{data}}(x)}\big[\min\big(0, -1 + D_\phi(x)\big)\big] - \mathbb{E}_{p(z)}\big[\min\big(0, -1 - D_\phi(G_\theta(z))\big)\big]$$
$$\mathcal{L}_G(\theta) = -\mathbb{E}_{p(z)}\big[D_\phi(G_\theta(z))\big].$$

Para muestras reales, el objetivo es $D_\phi(x) \geq 1$, y la penalización solo se activa cuando $D_\phi(x) < 1$. Para muestras sintéticas, el objetivo es $D_\phi(G_\theta(z)) \leq -1$, y la penalización solo se activa cuando $D_\phi(G_\theta(z)) > -1$. Notar que esta función de pérdida es análoga a la pérdida de margen de las SVMs, y evita que D_ϕ siga sobre-entrenando en ejemplos fáciles, lo que reduce la saturación de los gradientes que recibe G_θ.

7.3.3 Progressive GAN

SAGAN mejora la coherencia global de las muestras pero sigue limitado a 128×128, ya que el costo cuadrático de la atención y la dificultad de optimizar redes profundas a alta resolución impiden escalar la arquitectura directamente. **ProGAN** [59] fue el primer método en producir imágenes realistas a resolución 1024×1024. La idea central es un

curriculum learning[10] explícito, donde en lugar de entrenar los modelos directamente en la resolución final, G_θ y D_ϕ comienzan con arquitecturas simétricas que operan en resolución baja (4×4) y se van agregando pares de capas de manera progresiva hasta alcanzar la resolución objetivo.

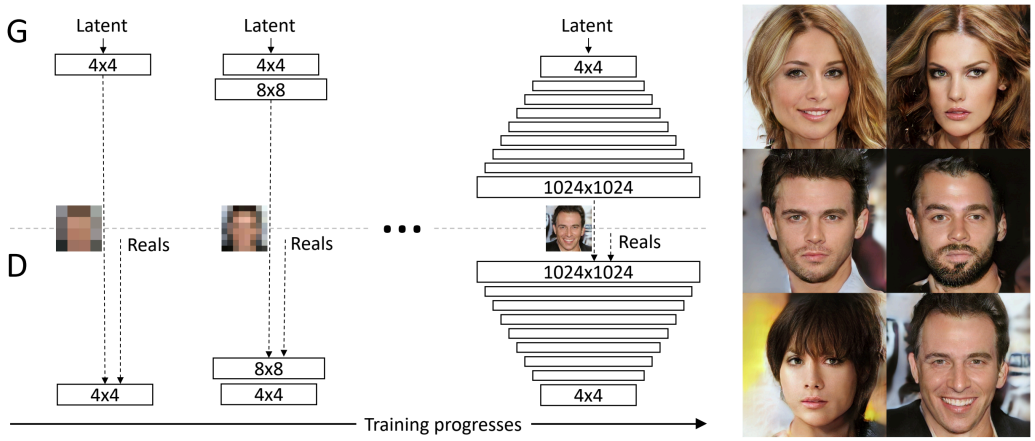

Figura 67: Arquitectura de ProGAN (izquierda) e imágenes generadas por el modelo (derecha).

Entrenamiento progresivo

Sean $G_\theta^{(r)}$ y $D_\phi^{(r)}$ las versiones de G_θ y D_ϕ a resolución $r \times r$. Al pasar de la resolución r a $2r$, ProGAN agrega un bloque convolucional en la parte superior de $G_\theta^{(r)}$ para realizar el upsampling $r \to 2r$ y, simétricamente, un bloque en la parte inferior de $D_\phi^{(r)}$ para realizar el downsampling $2r \to r$. Las imágenes reales de entrenamiento se redimensionan a la resolución actual y las nuevas capas se incorporan mediante un mecanismo de fade-in (explicado más abajo). Además, todas las capas existentes se siguen considerando entrenables al agregar las nuevas capas (esto permite interpretar el entrenamiento previo como una buena inicialización). Como resultado, los autores de ProGAN encontraron que, a igual presupuesto de cómputo, el entrenamiento progresivo es significativamente más rápido y estable que entrenar directamente en alta resolución.

Para cada nueva resolución $C_r \times r \times r$, al generador se le agrega una capa $\text{toRGB}_r : \mathbb{R}^{C_r \times r \times r} \to \mathbb{R}^{3 \times r \times r}$ al final, mientras que al discriminador se le agrega una capa $\text{fromRGB}_r : \mathbb{R}^{3 \times r \times r} \to \mathbb{R}^{C_r \times r \times r}$ al inicio. Estas capas cumplen la función de mapear entre el espacio de imágenes RGB y el espacio de features convolucionales, y viceversa. Ambas funciones son implementadas como convoluciones 1×1 que cambian la cantidad de canales sin alterar la resolución espacial.

[10] Esto es, un enfoque de aprendizaje progresivo donde la red se entrena primero en tareas más simples y luego se incrementa la dificultad.

Con respecto a la función objetivo, ProGAN es entrenado utilizando la pérdida WGAN-GP (estudiada en el próximo capítulo). También se utilizan otras heurísticas de entrenamiento y arquitectura, pero no se mencionarán ya que no son tan relevantes.

Fade-in de nuevas capas

Introducir nuevas capas de golpe altera considerablemente la función aprendida, lo que desestabiliza el entrenamiento. Para evitarlo, ProGAN utiliza una interpolación lineal controlada por un parámetro $\alpha \in [0, 1]$ que crece de 0 a 1 a lo largo de $N_{\text{fade-in}}$ steps de actualización. Más precisamente, durante la transición $r \to 2r$, la salida del generador es:

$$G_\theta^{(2r)}(z) = (1-\alpha)\,\text{toRGB}_{2r}(\text{Upsample}(h_r)) + \alpha\,\text{toRGB}_{2r}(\text{Conv}(\text{Upsample}(h_r))),$$

donde h_r es el último feature map de $G_\theta^{(r)}(z)$ a resolución r, Upsample es una interpolación determinista (e.g. nearest neighbor o bicúbica) y Conv es la nueva convolución que opera en resolución $2r \times 2r$. Simétricamente, durante la transición $2r \to r$, el discriminador se actualiza de la siguiente forma:

$$D_\phi^{(r)}(x_{2r}) = (1-\alpha)\,\text{fromRGB}_r(\text{Downsample}(x_{2r})) + \alpha\,\text{Conv}(\text{fromRGB}_{2r}(x_{2r})),$$

donde x_{2r} es la imagen real a resolución $2r$, Downsample es average pooling 2×2 y Conv es el nuevo bloque convolucional que pasa de resolución $2r \times 2r$ a $r \times r$. En $\alpha = 0$ se recuperan las arquitecturas de la resolución anterior y en $\alpha = 1$ las nuevas capas están completamente activas.

7.3.4 StyleGAN

Pendiente.

7.4 Arquitectura U-Net

Si bien las arquitecturas anteriores permiten generar imágenes en alta resolución, no es claro cómo utilizarlas de manera condicional cuando la condición c es una imagen (e.g., en tareas image-to-image). La arquitectura **U-Net** [60] es una red completamente convolucional con forma de **autoencoder**[11] que incluye **skip connections** entre los bloques del encoder y los bloques homólogos en el decoder. Por otro lado, si bien esta arquitectura fue propuesta originalmente para la tarea de **segmentación semántica** (donde cada píxel de una imagen

[11]Es decir, una red neuronal que primero comprime la información y luego la reconstruye. Estas redes serán estudiadas en más profundidad en el capítulo de autoencoders variacionales.

se clasifica individualmente), esta arquitectura es de gran importancia tanto en GANs como en modelos de difusión, donde la arquitectura debe recibir versiones más ruidosas de la imagen a generar, por lo que naturalmente se necesita una arquitectura tipo image-to--image.

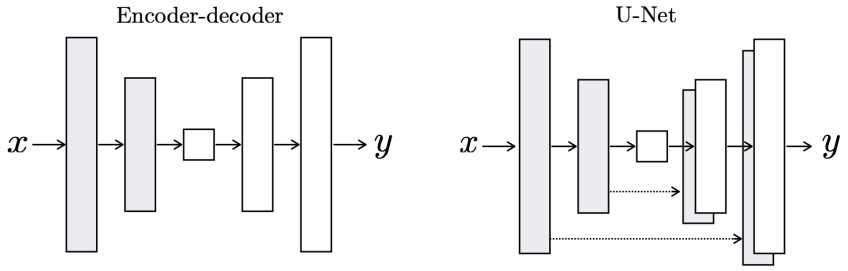

Figura 68: Comparación entre un autoencoder (izquierda) y la arquitectura U-Net (derecha). Imagen obtenida desde [38].

Por otro lado, hoy en día los modelos generativos de imágenes suelen utilizar arquitecturas tipo ViT (e.g. Diffusion Transformer), donde una imagen se *patchifica* para poder tratarla como una secuencia de tokens usando un Transformer.

El siguiente diagrama muestra una U-Net con dos bloques de bajada y dos bloques de subida. En cada bloque, la tupla (H, W, C) superior indica la resolución de entrada y la tupla inferior indica la resolución de salida del bloque:

```
graph LR

    Input["Input<br>(H, W, C)"] --> Phi0["Convolución<br>(H, W, C)<br>(H,
W, 64)"]

    Phi0 --> Down0["Downsample<br>(H, W, 64)<br>(H/2, W/2, 64)"]
    Down0 --> Phi1["Convolución<br>(H/2, W/2, 64)<br>(H/2, W/2, 128)"]
    Phi1 --> Down1["Downsample<br>(H/2, W/2, 128)<br>(H/4, W/4, 128)"]

    Down1 --> Phi2["Convolución<br>(H/4, W/4, 128)<br>(H/4, W/4, 256)"]

    Phi2 --> Up1["Upsample<br>(H/4, W/4, 256)<br>(H/2, W/2, 128)"]
    Up1 --> Psi1["Concat + Conv<br>(H/2, W/2, 256)<br>(H/2, W/2, 128)"]
    Psi1 --> Up0["Upsample<br>(H/2, W/2, 128)<br>(H, W, 64)"]
    Up0 --> Psi0["Concat + Conv<br>(H, W, 128)<br>(H, W, 64)"]

    Psi0 --> Out["Conv1×1<br>(H, W, 64)<br>(H, W, C)"]

    Phi0 -.- Psi0
```

```
    Phi1 -.- Psi1

    subgraph Encoder
        Phi0
        Down0
        Phi1
        Down1
    end

    subgraph Middle
        Phi2
    end

    subgraph Decoder
        Up1
        Psi1
        Up0
        Psi0
    end

    classDef default fill:#f9f9f9,stroke:#333,stroke-width:2px;
    classDef encoder fill:#fce4ec,stroke:#333,stroke-width:2px;
    classDef middle fill:#e3f2fd,stroke:#333,stroke-width:2px;
    classDef decoder fill:#fff8e1,stroke:#333,stroke-width:2px;

    class Phi0,Down0,Phi1,Down1 encoder;
    class Phi2 middle;
    class Up1,Psi1,Up0,Psi0 decoder;
    class Input,Out default;
```

Cada bloque de bajada del encoder se conecta a su bloque de subida homólogo en el decoder mediante una skip connection (concatenación en la dimensión de los canales). Esto facilita la transmisión directa de información de bajo nivel desde el encoder hacia el decoder, ayudando al modelo a conservar detalles espaciales finos. Las modificaciones modernas de esta arquitectura suelen incorporar bloques de self-attention en algunas resoluciones (similar a SAGAN) y mecanismos de atención cruzada para condicionamiento textual, lo cual será revisado al estudiar modelos de difusión.

7.4.1 Implementación

Dada la importancia de esta arquitectura neuronal para los modelos generativos, se entregará una implementación minimal de una U-Net, y luego se utilizará esta implementación

para entrenar un modelo de segmentación semántica sobre el dataset Oxford-IIIT Pet. Para simplificar la implementación, no se incluirán bloques de normalización ni dropout u otro tipo de regularizaciones. En el próximo capítulo se adaptará esta arquitectura para ser utiliza en modelo Pix2Pix (GAN para *traducción de imágenes*), y luego se adaptará con bloques de self-attention al implementar un modelo de difusión para imágenes.

La U-Net es una arquitectura completamente convolucional, por lo que se comenzará definiendo un módulo `ConvBlock`, formado por dos convoluciones 3×3 con $S = P = 1$ (i.e., no cambia la resolución). En caso de querer cambiar la cantidad de canales, esto se hará en la primera convolución[12].

```
class ConvBlock(nn.Module):

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(out_channels, out_channels, kernel_size=3,
padding=1),
            nn.ReLU(),
        )

    def forward(self, x):
        return self.conv(x)
```

Cada bloque de bajada del encoder, implementado en `DownBlock`, está compuesto por una `ConvBlock` (que duplica los canales sin cambiar la resolución) seguida de una operación de pooling (que divide en dos la resolución sin cambiar los canales). En consecuencia, cada `DownBlock` duplica la cantidad de canales y reduce la resolución a la mitad. Este bloque retorna, además, la salida de la convolución (antes del pooling) para que sea usada en la skip connection con el decoder en la resolución homóloga respectiva.

```
class DownBlock(nn.Module):

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = ConvBlock(in_channels, out_channels)
        self.down = nn.MaxPool2d(kernel_size=2)

    def forward(self, x):
```

[12]Esto es una decisión de diseño. De forma relacionada, el paper original de U-Net no incluye padding, por lo que las resoluciones van disminuyendo levemente. Aquí se seguirá la elección natural de usar $P = 1$.

```
        skip = self.conv(x)
        return self.down(skip), skip
```

Por otro lado, cada bloque de subida del decoder, implementado en `UpBlock`, parte duplicando la resolución y dividiendo los canales por dos usando una convolución transpuesta[13]. Luego, realiza la skip conection, concatenando lo anterior con el bloque auxiliar homólogo del encoder. Finalmente, se aplica una `ConvBlock` para ajustar a la cantidad de canales de salida deseada. En consecuencia, cada `UpBlock` reduce a la mitad la cantidad de canales y duplica la resolución.

```
class UpBlock(nn.Module):

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.up = nn.ConvTranspose2d(in_channels, in_channels // 2,
kernel_size=2, stride=2)
        self.conv = ConvBlock(in_channels, out_channels)

    def forward(self, x, skip):
        x = self.up(x)
        x = torch.cat([x, skip], dim=1)
        return self.conv(x)
```

Con estos bloques, la U-Net completa se construye de forma simétrica. Los bloques del decoder reciben las skip connections del encoder en orden inverso, y la salida del último bloque del decoder pasa por una convolución 1×1 para ajustar la cantidad de canales a la salida deseada (`n_classes` en la tarea de segmentación semántica, o 3 en la tarea de generación de imágenes).

```
class UNet(nn.Module):

    def __init__(self, in_channels, n_classes, base_ch=64):
        super().__init__()

        self.down1 = DownBlock(in_channels, base_ch)
        self.down2 = DownBlock(base_ch, base_ch * 2)
        self.down3 = DownBlock(base_ch * 2, base_ch * 4)
        self.down4 = DownBlock(base_ch * 4, base_ch * 8)

        self.bottleneck = ConvBlock(base_ch * 8, base_ch * 16)

        self.up4 = UpBlock(base_ch * 16, base_ch * 8)
```

[13]Esto se realiza para que, al concatenar con la skip connection del encoder (el cual tiene la misma resolución), la cantidad de canales coincida.

```
        self.up3 = UpBlock(base_ch * 8, base_ch * 4)
        self.up2 = UpBlock(base_ch * 4, base_ch * 2)
        self.up1 = UpBlock(base_ch * 2, base_ch)

        self.out = nn.Conv2d(base_ch, n_classes, kernel_size=1)

    def forward(self, x):
        x, skip1 = self.down1(x)
        x, skip2 = self.down2(x)
        x, skip3 = self.down3(x)
        x, skip4 = self.down4(x)

        x = self.bottleneck(x)

        x = self.up4(x, skip4)
        x = self.up3(x, skip3)
        x = self.up2(x, skip2)
        x = self.up1(x, skip1)

        return self.out(x)
```

7.4.2 Aplicación a segmentación semántica

Si bien esta arquitectura será utilizada en modelos generativos, U-Net fue propuesta originalmente para la tarea de segmentación semántica en imágenes médicas, donde se busca clasificar individualmente cada pixel de una imagen para construir un mapa de segmentación (máscara) y poder identificar diferentes estructuras o regiones de interés. Aquí se entrenará el modelo anterior sobre el dataset Oxford-IIIT Pet, el cual contiene imágenes de mascotas junto a sus mapas de segmentación (fondo, mascota y borde).

Para comenzar, todas las imágenes y máscaras se redimensionarán a resolución 256×256. Notar que las máscaras se deben interpolar utilizando métodos como `InterpolationMode.NEAREST` para evitar valores intermedios que no corresponderían a etiquetas válidas (fuera de $\{0, 1, 2\}$):

```
img_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
])

mask_transform = transforms.Compose([
    transforms.Resize((256, 256),
interpolation=transforms.InterpolationMode.NEAREST),
```

```
    transforms.PILToTensor(),
])

def transform(img, mask):
    img = img_transform(img)
    mask = mask_transform(mask).long().squeeze()
    return img, (mask - 1)

dataset = datasets.OxfordIIITPet(
    root='data', download=True, target_types='segmentation',
transforms=transform,
)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True,
drop_last=True)
```

Figura 69: Ejemplo de imagen y máscara de segmentación del dataset Oxford-IIIT Pet.

Para entrenar el modelo se seguirá el enfoque de máxima verosimilitud del clasificador minimizando la entropía cruzada, aplicada a nivel de pixel, donde la función de pérdida total es el promedio sobre todos los píxeles de las pérdidas individuales.

```
def train(model, optimizer, dataloader, n_epochs):

    model.to(DEVICE)
    model.train()

    loss_fn = nn.CrossEntropyLoss()
    losses = []

    try:
        for epoch in range(n_epochs):
            pbar = tqdm(dataloader, desc=f'Época {epoch+1}/{n_epochs}')
            for imgs, masks in pbar:
                imgs, masks = imgs.to(DEVICE), masks.to(DEVICE)
```

```
                logits = model(imgs)
                loss = loss_fn(logits, masks)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                pbar.set_postfix(loss=loss.item())
                losses.append(loss.item())

    except KeyboardInterrupt:
        print('Entrenamiento interrumpido.')
        training_log = {'model': model.state_dict(), 'losses': losses}
        torch.save(training_log, 'training.pt')
```

Finalmente, se entrenará el modelo y se graficará su dinámica de entrenamiento.

```
# Entrenamiento:
model = UNet(in_channels=3, n_classes=3)
optimizer = optim.Adam(model.parameters(), lr=1e-4)
train(model, optimizer, dataloader, n_epochs=50)

# Carga del modelo entrenado:
training_log = torch.load('training.pt', map_location=DEVICE,
weights_only=True)
model.load_state_dict(training_log['model'])
```

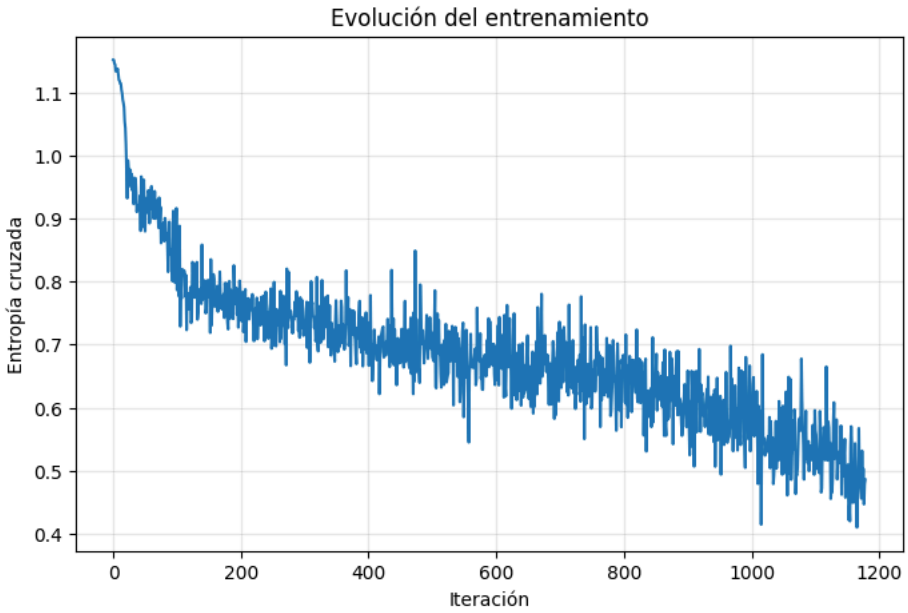

Figura 70: Curva de pérdida durante el entrenamiento de la U-Net para la tarea de segmentación.

Una vez entrenado el modelo, la predicción de la máscara para una imagen dada se obtiene aplicando arg max sobre los logits de salida en la dimensión de las clases.

```
def predict_mask(model, img):
    model.eval()
    with torch.no_grad():
        img = img.unsqueeze(0).to(DEVICE)
        logits = model(img)
        probs = torch.softmax(logits, dim=1)
        pred_mask = torch.argmax(probs, dim=1)
    return pred_mask.squeeze().cpu()
```

Figura 71: Imagen original, máscara real y predicción del modelo entrenado.

Parte IV

Autoencoders variacionales

Capítulo 9

Autoencoders clásicos e inferencia variacional

En este capítulo se revisarán los fundamentos teóricos de una familia importante de modelos generativos llamados autoencoders variacionales. Si bien esta clase de modelos fue propuesta el año 2013, hoy en día estos modelos se siguen utilizando de forma activa como parte de modelos más complejos. Para complementar la formulación teórica, se realizarán dos implementaciones minimales de un autoencoder variacional, y se revisará la idea de interpolación en el espacio latente.

Un **autoencoder variacional** (VAE) es, al igual que una GAN, una red bayesiana de variable latente que busca aprender una distribución de probabilidad desconocida, $p_{\text{data}}(x)$, a partir de un conjunto de muestras i.i.d., $\mathcal{D} = \{x^n\}_{n=1}^N \subset \mathbb{R}^D$, generadas a partir de $p_{\text{data}}(x)$. Como es usual, x representará siempre la variable aleatoria de los datos en $\mathbb{R}^D$ (en el código, D será representado mediante `data_dim`), mientras que z será la variable aleatoria latente en $\mathbb{R}^L$ (en el código, L será representado por `latent_dim`). Además, en línea con la manifold hypothesis, es común considerar $L \ll D$, aunque no es un requisito estricto para el desarrollo teórico y, de hecho, se mencionarán casos donde es útil considerar $D < L$.

Al igual que en otros modelos de variable latente, los términos de la descomposición $p_\theta(x, z) = p_\theta(z)p_\theta(x \mid z)$ son los siguientes:

- $p_\theta(z)$ se interpreta como una distribución a priori sobre la variable latente $z \in \mathbb{R}^L$.
- $p_\theta(x \mid z)$ es la distribución desde la que se generarán nuevas muestras $x \in \mathbb{R}^D$ a partir de un valor dado de la variable latente $z \in \mathbb{R}^L$.

Por lo tanto, una vez el VAE esté entrenado, se podrá generar una nueva muestra desde $p_\theta(x)$ utilizando ancestral sampling, donde se comienza generando una muestra de la variable latente $z \sim p_\theta(z)$ y luego se genera otra desde el modelo condicional $x \sim p_\theta(x \mid z)$.

9.1 Autoencoders clásicos

Antes de comenzar el estudio de los autoencoders variacionales se repasarán algunos conceptos asociados a los autoencoders clásicos (no variacionales), donde la principal diferencia

entre ambos enfoques es que los AEs clásicos son de naturaleza determinista, mientras que los AEs variacionales son de naturaleza probabilística. En particular, los autoencoders clásicos no pueden ser vistos como modelos generativos, lo cual es esperable considerando que este tipo de modelos surgió hace más de 30 años, donde el foco estaba puesto en capacidades de representación compresión, y no de generación.

Un autoencoder es, en principio, una red neuronal utilizada para aprender representaciones eficientes de un conjunto de datos. Estas representaciones son aprendidas de manera autosupervisada mediante la minimización de alguna función objetivo.

Dada una muestra $x \in \mathbb{R}^D$, un autoencoder estándar busca aprender una representación compacta $E_\phi(x) \in \mathbb{R}^L$ (con $E_\phi : \mathbb{R}^D \to \mathbb{R}^L$ una red neuronal llamada **encoder**), de tal modo que otro modelo neuronal $D_\theta : \mathbb{R}^L \to \mathbb{R}^D$ (llamado **decoder**) sea capaz de reconstruir con bastante precisión la muestra original $x \in \mathbb{R}^D$ a partir de esta representación compacta, es decir, $D_\theta\big(E_\phi(x)\big) \approx x$. Para esto, las redes neuronales E_ϕ y D_θ son entrenadas de forma conjunta utilizando cómo función de pérdida alguna métrica de discrepancia, siendo la distancia euclidiana la función de pérdida usual[14]. Más precisamente, si $\mathcal{D} = \{x^n\}_{n=1}^N \subset \mathbb{R}^D$ es el conjunto de entrenamiento, las redes neuronales son optimizadas minimizando

$$\mathcal{L}_{\text{MSE}}(\phi, \theta) = \frac{1}{N} \sum_{n=1}^{N} \left\| x^n - \big(D_\theta \circ E_\phi\big)(x^n) \right\|^2,$$

donde $\big(D_\theta \circ E_\phi\big)(x^n) = D_\theta\big(E_\phi(x^n)\big)$ es la composición del encoder con el decoder aplicado sobre la muestra $x^n \in \mathbb{R}^D$. Esta función de pérdida es natural ya que busca precisamente que la reconstrucción sea lo más cercana posible a la muestra original. Además, notar que si $L \geq D$, entonces la reconstrucción se puede realizar de forma exacta (e.g. se puede considerar $E_\phi(x) = \Big(x_1, \cdots, x_D, \underbrace{0 \cdots 0}_{L-D \text{ veces}}\Big)^\top \in \mathbb{R}^L$ y luego $D_\theta(z) = (z_1, \cdots, z_D)^\top \in \mathbb{R}^D$), por lo que este problema solo es no trivial cuando $L < D$.

A modo de ejemplo se implementará un autoencoder clásico de forma minimal usando el mismo dataset de juguete 2D usado en la introducción de GANs:

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.datasets import make_swiss_roll

def get_batch(batch_size=1000, noise=0.1):
```

[14]Notar que en este tipo de modelos no tienen sentido conceptos como la verosimilitud debido a la falta de una estructura probabilística sobre el modelo y las muestras.

```
    x, _ = make_swiss_roll(batch_size, noise=noise)
    x = x[:, [0, 2]]
    x = (x - x.mean()) / x.std()
    return torch.tensor(x).float()

# Ejemplo:
samples = get_batch()
plt.figure(figsize=(3, 3))
plt.scatter(samples[:, 0], samples[:, 1], s=1)
plt.show()
```

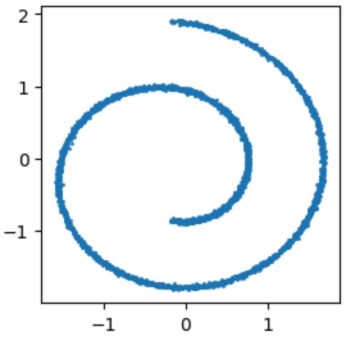

Figura 72: Muestras del dataset de juguete 2D.

Dada la baja complejidad de los datos, será suficiente considerar redes neuronales fully connected. Para el encoder $E_\phi : \mathbb{R}^D \to \mathbb{R}^L$ se utilizará la siguiente red neuronal:

```
class Encoder(nn.Module):

    def __init__(self, data_dim, latent_dim):
        super().__init__()

        self.encoder = nn.Sequential(
            nn.Linear(data_dim, 128), nn.ReLU(),
            nn.Linear(128, 64), nn.ReLU(),
            nn.Linear(64, 32), nn.ReLU(),
            nn.Linear(32, latent_dim)
        )

    def forward(self, x):
        return self.encoder(x)
```

Mientras que para el decoder $D_\theta : \mathbb{R}^L \to \mathbb{R}^D$ se utilizará la siguiente red neuronal:

```
class Decoder(nn.Module):
```

```
    def __init__(self, data_dim, latent_dim):
        super().__init__()

        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 32), nn.ReLU(),
            nn.Linear(32, 64), nn.ReLU(),
            nn.Linear(64, 128), nn.ReLU(),
            nn.Linear(128, data_dim)
        )

    def forward(self, z):
        return self.decoder(z)
```

La siguiente clase `Autoencoder` implementa el loop de entrenamiento y reconstrucción de un AE clásico con error cuadrático medio como función de costo:

```
class Autoencoder:

    def __init__(self, data_dim, latent_dim):

        self.latent_dim = latent_dim

        self.encoder = Encoder(data_dim, latent_dim)
        self.decoder = Decoder(data_dim, latent_dim)
        self.encoder_optimizer = optim.AdamW(self.encoder.parameters())
        self.decoder_optimizer = optim.AdamW(self.decoder.parameters())

    def train(self, iters):

        loss_fn = nn.MSELoss()

        for _ in range(iters):

            # MSE:
            x = get_batch()
            z = self.encoder(x)
            x_dec = self.decoder(z)
            loss = loss_fn(x_dec, x)

            # Optimización:
            self.encoder_optimizer.zero_grad()
            self.decoder_optimizer.zero_grad()
            loss.backward()
            self.encoder_optimizer.step()
```

```
            self.decoder_optimizer.step()

    def reconstruct(self, x):
        z = self.encoder(x)
        x_dec = self.decoder(z)
        return x_dec
```

Entrenando el autoencoder definido anteriormente, se puede observar que la reconstrucción de un autoencoder es casi exacta:

```
# Entrenamiento:
autoencoder = Autoencoder(data_dim=2, latent_dim=16)
autoencoder.train(iters=5000)

# Reconstrucción:
x = get_batch()
reconstruction = autoencoder.reconstruct(x).detach()

plt.figure(figsize=(3, 3))
plt.scatter(x[:, 0], x[:, 1], s=1, label='Original')
plt.scatter(reconstruction[:, 0], reconstruction[:, 1], s=1,
label='Reconstrucción')
plt.legend()
plt.show()
```

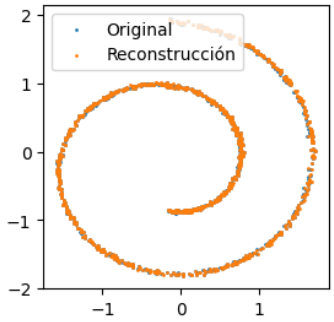

Figura 73: Reconstrucción realizada por un autoencoder clásico.

Que la reconstrucción sea prácticamente exacta es esperable ya que esa es precisamente la función objetivo sobre la que se entrenó el autoencoder. Sin embargo, se verá que esta función de pérdida no es suficiente para darle capacidades generativas a un autoencoder, lo cual se debe, esencialmente, a una falta de noción probabilística sobre la formulación del modelo.

9.1.1 Aplicaciones

Reducción de dimensionalidad

La aplicación natural de un autoencoder es la reducción de dimensionalidad, donde una muestra original (usualmente de alta dimensión) es representada mediante un vector de menor dimensión, lo cual es útil, por ejemplo, para almacenar información usando menos memoria (similar a como ocurre con la compresión JPEG), para implementar sistemas de information retrieval de imágenes (e.g. RAG), o para entrenar otros modelos de machine learning utilizando representaciones compactas de la data en vez de usar los datos originales. Esto último es el uso más común hoy en día para los VAEs, donde un modelo más grande (e.g. un modelos de difusión para imágenes o videos) es entrenado en el espacio latente de un VAE, volviendo mucho más eficiente y escalable el entrenamiento.

Detección de anomalías

Durante el proceso de aprendizaje de un AE, es esperable que el encoder E_ϕ aprenda a omitir la información común entre las instancias de entrenamiento $\mathcal{D} = \{x^n\}_{n=1}^N \subset \mathbb{R}^D$ para solo enfocarse en codificar las características distintivas de cada instancia. Al mismo tiempo, el decoder D_θ debe ser capaz de "memorizar" esta información común para incluirla durante la reconstrucción, y solo variar los detalles propios de la muestra utilizando la representación compacta dada por el encoder.

Debido a este patrón de omitir la información común entre las instancias, es esperable que la codificación y reconstrucción de instancias anómalas sea de menor calidad que la reconstrucción obtenida para las muestras durante el entrenamiento. De esta forma, el error de reconstrucción de un AE puede ser utilizado como un sistema de detección de anomalías mediante el filtrado de instancias que se alejan más de un cierto umbral del error medio de entrenamiento.

Representation learning

Pendiente.

9.1.2 Propiedades de los autoencoders

En esta subsección se revisarán, por completitud, algunas propiedades comúnmente mencionadas sobre los autoencoders clásicos.

Propiedad de Johnson-Lindenstrauss

El siguiente resultado indica que existe una función encoder que preserva, aproximadamente, la distancia euclidiana entre las muestras originales:

Teorema 3 (Johnson-Lindenstrauss). *Dado un conjunto de puntos* $\{x^1, \cdots, x^N\} \subset \mathbb{R}^D$ *y un escalar* $\epsilon \in (0,1)$*, entonces, para todo* $L \geq \mathcal{O}\left(\frac{\log N}{\epsilon^2}\right)$*, existe un mapa* $f : \mathbb{R}^D \to \mathbb{R}^L$ *tal que*

$$(1-\epsilon)\left\|x^i - x^j\right\|^2 \leq \left\|f(x^i) - f(x^j)\right\|^2 \leq (1+\epsilon)\left\|x_i - x_j\right\|^2,$$

para todo $i, j \in \{1, \cdots, N\}$*. Más aún, esta cota no se puede mejorar. Es decir, existe un conjunto* $\{x^1, \cdots, x^N\} \subset \mathbb{R}^D$ *donde se alcanza la cota.*

Este resultado indica que es posible encontrar una función de codificación tal que la distancia relativa entre las muestras originales no se distorsiona demasiado cuando son transformadas a su representación compacta de menor dimensión. Más aún, se puede probar que el problema de encontrar el mapa f está en la clase de complejidad BPP.

Relación con PCA

Pendiente.

Uso como modelos generativos

Dada la similitud de un autoencoder con un modelo de variable latente, se podría intentar utilizar el decoder como un modelo generativo de forma similar a como ocurre en una GAN. Sin embargo, dado que no hay un prior $p(z)$ para la variable latente, no es obvio qué variable latente elegir para realizar la generación. A modo de ejemplo, se intentará generar un conjunto de muestras a partir de un conjunto variables latentes gaussianas:

```
def generate_samples(autoencoder, n_samples):
    z = torch.randn(n_samples, autoencoder.latent_dim)
    x_dec = autoencoder.decoder(z)
    return x_dec

# Generación:
samples = generate_samples(autoencoder, 5000).detach()
plt.figure(figsize=(5, 5))
plt.scatter(samples[:, 0], samples[:, 1], s=1)
plt.show()
```

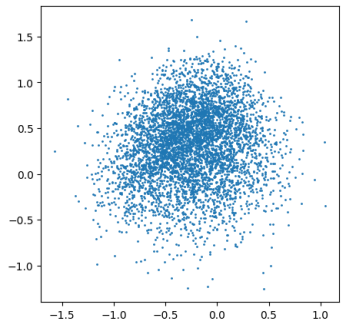

Figura 74: Muestras generadas por un autoencoder clásico.

Se observa que las muestras decodificadas a partir de muestras latentes $z \sim \mathcal{N}(0, \mathrm{I}_L)$ no corresponden a muestras similares a las observadas en el conjunto de entrenamiento `make_swiss_roll`. Esto se debe, principalmente, a que no se definió una distribución prior $p(z)$ durante el entrenamiento, por lo que la elección $z \sim \mathcal{N}(0, \mathrm{I}_L)$ es realmente arbitraria y no justificada (lo mismo ocurriría si se elige, por ejemplo, una distribución uniforme).

Una solución trivial a este problema es considerar como prior a la distribución empírica de las representaciones latentes de las muestras en $\mathcal{D}$, $p_{\mathcal{D}}(z) = \frac{1}{N} \sum_{n=1}^{N} \delta_{E_\phi(x^n)}(z)$. Sin embargo, este prior solo permitirá generar las mismas muestras usadas durante el entrenamiento, por lo que no habría variabilidad. Más aún, este prior no tiene soporte conexo, por lo que las interpolaciones lineales de variables latentes no necesariamente tendrán sentido como sí ocurre, por ejemplo, en una GAN.

La siguiente función muestra que las interpolaciones lineales en el espacio latente de un AE clásico no necesariamente tienen sentido semántico:

```
def latent_interpolation(autoencoder, x0, x1, t):

    z_0 = autoencoder.encoder(x0.unsqueeze(0))
    z_1 = autoencoder.encoder(x1.unsqueeze(0))

    z_t = (1 - t) * z_0 + t * z_1
    x_t = autoencoder.decoder(z_t).squeeze(0)

    return x_t

# Interpolación:
batch_x = get_batch()
x_0, x_1 = batch_x[0], batch_x[1]

plt.figure(figsize=(3, 3))
plt.scatter(batch_x[:, 0], batch_x[:, 1], s=1, alpha=0.05)
```

```
n_steps = 500
for t in torch.linspace(0, 1, n_steps):
    x_t = latent_interpolation(autoencoder, x_0, x_1, t).detach()
    plt.scatter(*x_t, s=1, color='k')

plt.scatter(*x_0, s=100, color='r', label='$x_0$')
plt.scatter(*x_1, s=100, color='g', label='$x_1$')

plt.legend()
plt.show()
```

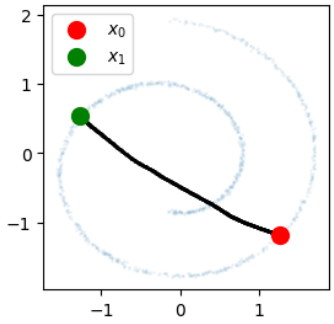

Figura 75: Interpolaciones lineales en el espacio latente de un AE clásico.

Se observa que las interpolaciones en el espacio latente no producen muestras semánticamente significativas debido a la falta de conexidad y convexidad del espacio latente. Por otro lado, si bien se pueden utilizar heurísticas para inducir variabilidad (e.g. perturbar la distribución empírica $p_{\mathcal{D}}(z) = \frac{1}{N}\sum_{n=1}^{N} \delta_{E_\phi(x^n)}(z)$ con kernels gaussianos), esta es una limitación intrínseca de los AEs debido a su naturaleza determinística. En un VAE, en cambio, estos problemas no ocurrirán debido a que el prior será definido desde un comienzo como una distribución gaussiana, la cual tiene soporte denso. Sin embargo, como se verá en la implementación, imponer esta condición generará un trade-off entre la calidad de reconstrucción y la capacidad generativa.

9.1.3 Otros tipos de autoencoders

Si bien la función objetivo $\mathcal{L}_{\text{MSE}}(\phi, \theta)$ es natural, existen algunas variantes que dotan al autoencoder de otras propiedades útiles en el campo de representation learning. En esta subsección se revisarán algunas técnicas clásicas de regularización de autoencoders.

Sparse autoencoder

Como se comentó anteriormente, si el autoencoder es overcomplete ($L > D$), entonces la tarea de reconstrucción es trivial ya que los modelos tienen suficiente capacidad para construir la función identidad en la composición $D_\theta \circ E_\phi : \mathbb{R}^D \to \mathbb{R}^D$. Sin embargo, hay casos donde sí es útil considerar el caso overcomplete.

Un ejemplo usual consiste en el problema de desacoplar características de los datos, donde el objetivo es que cada coordenada $E_\phi(x)_l \in \mathbb{R}$, $l \in \{1, ..., L\}$, represente una característica específica de la muestra $x \in \mathbb{R}^D$ en vez de un conjunto de características mezcladas. Por ejemplo, en el caso de imágenes, una coordenada del encoding $E_\phi(x) \in \mathbb{R}^L$ puede representar únicamente el color de pelo de una persona, mientras que otra coordenada representa únicamente el color de ojos. En el caso no desacoplado, una misma coordenada puede representar varias features al mismo tiempo, limitando la interpretabilidad de la representación latente.

De forma similar, las coordenadas de la codificación dada por el encoder pueden indicar la presencia o ausencia de ciertas features relevantes aprendidas durante el entrenamiento, lo cual puede ser muy útil para tareas de clasificación. En estos casos, y en línea con la *sparse coding hypothesis* formulada en neurociencia, la representación $E_\phi(x) \in \mathbb{R}^L$ suele ser un vector sparse, es decir, un vector donde la mayoría de sus coordenadas son nulas.

Un ***K*-sparse autoencoder** (K-SAE) es un AE con $L > D$ entrenado para que la codificación aprendida tenga a lo más $K \leq L$ coordenadas no nulas. Si bien esto se podría conseguir dejando solo las K dimensiones con mayor valor absoluto y cambiando el resto de coordenadas por 0, una opción más regular es relajar esta condición y agregar un regularizador a la función de costo que penalice la norma de las salidas de encoder. Por ejemplo, se podría considerar:

$$\mathcal{L}_{\text{SAE}}(\phi, \theta) = \mathcal{L}_{\text{MSE}}(\phi, \theta) + \frac{\alpha}{N} \sum_{n=1}^{N} \left\| E_\phi(x^n) \right\|_p^p,$$

donde $\alpha > 0$ es un ponderador que indica el grado de regularización y $p \in \mathbb{N}$ define la métrica de penalización. Si bien esto no garantiza que a lo más K coordenadas sean no nulas, al momento de la inferencia (i.e., con el SAE entrenado) se pueden apagar las $L - K$ coordenadas de menor valor para forzar esta condición.

En el próximo capítulo se revisarán los β-VAEs, los cuales sesgan al modelo a aprender representaciones latentes desacopladas.

Contractive autoencoder

Una técnica de regularización usual en los AE consiste en sesgar al modelo a preferir representaciones $E_\phi(x^1), E_\phi(x^2) \in \mathbb{R}^L$ cercanas (en algún sentido) para muestras $x^1, x^2 \in \mathbb{R}^D$ cercanas entre sí. Dado que la derivada mide la tasa de cambio de una función, acotar superiormente la derivada del encoder permite acotar superiormente su variación entre dos puntos de su dominio. Más precisamente, asumiendo un encoder $E_\phi : \mathbb{R}^D \to \mathbb{R}^L$ lo suficientemente regular, se puede probar la siguiente condición suficiente de lipschitzianidad:

$$\text{si} \quad \sup_{x \in \mathbb{R}^D} \left\| \mathrm{D}_x E_\phi(x) \right\|_F \leq M, \quad \text{entonces} \quad \left\| E_\phi(x^2) - E_\phi(x^1) \right\| \leq M \left\| x^2 - x^1 \right\|,$$

para todo $x^1, x^2 \in \mathbb{R}^D$. Aquí, $\mathrm{D}_x E_\phi(x) \in \mathcal{M}_{L,D}(\mathbb{R})$ es la matriz jacobiana[15] de E_ϕ (evaluada en $x \in \mathbb{R}^D$) y $\left\| \mathrm{D}_x E_\phi(x) \right\|_F^2 = \sum_{i=1}^{L} \sum_{j=1}^{D} \left(\mathrm{D}_x E_\phi(x) \right)_{ij}^2$ es su norma Frobenius (al cuadrado). En particular, la propiedad anterior indica que si la cota global $M > 0$ es pequeña, las codificaciones de dos muestras cercanas se mantendrán cercanas en el sentido de la norma euclidiana. Equivalentemente, pequeñas variaciones en una muestra produce pequeñas variaciones en sus representaciones compactas.

Motivado por la propiedad anterior, un **autoencoder contractivo** (CAE, [61]) penaliza la magnitud de las derivadas del encoder para inducir representaciones cercanas para muestras cercanas:

$$\mathcal{L}_{\text{CAE}}(\phi, \theta) = \mathcal{L}_{\text{MSE}}(\phi, \theta) + \frac{\alpha}{N} \sum_{n=1}^{N} \left\| \mathrm{D}_x E_\phi(x^n) \right\|_F^2$$

Denoising autoencoder

Otra variante típica del autoencoder usual consiste en corromper levemente los datos de entrada $x \in \mathbb{R}^D$ para dificultad aún más su reconstrucción. Para esto, se pueden utilizar distintos enfoques de corrupción:

- **Ruido gaussiano:** a la muestra original se le suma un ruido gaussiano $\epsilon \sim \mathcal{N}(0\mathrm{I}_D)$. Útil para datos de naturaleza continua y no acotada. Este es el tipo de ruido que se utilizará en los modelos de difusión.
- **Masking:** se fijan algunas coordenadas al azar (e.g. el 10%) a 0 con el fin de "ocultar" la información contenida por esa coordenada, de forma análoga al masking realizado en BERT.

[15] Recordar que para un campo vectorial $F : \mathbb{R}^M \to \mathbb{R}^N$, la matriz jacobiana $\mathrm{D}_x F(x) \in \mathcal{M}_{N,M}(\mathbb{R})$ está definida como $\left(\mathrm{D}_x F(x) \right)_{ij} = \frac{\partial F_i}{\partial x_j}(x)$. Equivalentemente, $\mathrm{D}_x F(x) = \begin{pmatrix} \nabla_x F_1(x)^\top \\ \vdots \\ \nabla_x F_N(x)^\top \end{pmatrix}$.

- **Sal y pimienta:** se eligen algunas coordenadas al azar y se fijan sus valores al valor máximo o mínimo (decidido al azar) que puede tomar la coordenada.

Si $T : \mathbb{R}^D \to \mathbb{R}^D$ es la función de corrupción, un **denoising autoencoder** (DAE, [62]) es entrenado de la misma forma que un AE normal, solo que la entrada es pasada por T antes de entrar al autoencoder. Es decir, un DAE es entrenado minimizando la función objetivo

$$\mathcal{L}_{\text{DAE}}(\phi, \theta) = \frac{1}{N} \sum_{n=1}^{N} \left\| x^n - \big(D_\theta \circ E_\phi \circ T\big)(x^n) \right\|^2,$$

donde $\big(D_\theta \circ E_\phi \circ T\big)(x) = D_\theta\big(E_\phi(T(x))\big)$ puede ser visto como la composición de una técnica de data augmentation (input corruption) con un AE clásico. Esta técnica de data augmentation busca mejorar la capacidad de generalización del modelo durante su entrenamiento, por lo que usualmente no se aplica la transformación T cuando se utiliza el modelo ya entrenado. Sin embargo, también es posible utilizar el DAE como un modelo de denoising en sí, donde se cuenta con una imagen corrupta y el DAE permite obtener una reconstrucción limpia de la imagen.

A continuación se implementará un DAE simple sobre el dataset Fashion MNIST:

```
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torchvision.utils import make_grid
from torch.utils.data import DataLoader
from torch.optim import Adam
import matplotlib.pyplot as plt
import random
import tqdm

DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Para mostrar una variación de implementación, se utilizará una única red neuronal que contenga tanto el encoder como el decoder de forma unificada (i.e., la red neuronal modelará directamente $D_\theta \circ E_\phi : \mathbb{R}^D \to \mathbb{R}^D$). Esto es una práctica usual cuando el autoencoder va a ser utilizado de manera completa y no solo una de sus partes (e.g. solo el encoder o solo el decoder).

Para el entrenamiento se considerará a la transformación $T : \mathbb{R}^D \to \mathbb{R}^D$ como la corrupción de sal y pimienta, la cual se puede implementar usando variables aleatorias binarias sobre cada pixel de la imagen, donde el parámetro de cada distribución Bernoulli asociada se elige uniformemente dentro de un rango de valores $[p_{\text{mín}}, p_{\text{máx}}] \subset (0, 1)$:

```
class DAE(nn.Module):

    def __init__(self, img_shape):
        super().__init__()

        self.img_shape = img_shape
        n_features = img_shape[0] * img_shape[1] * img_shape[2]

        self.autoencoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(n_features, 256), nn.ReLU(),
            nn.Linear(256, 128), nn.ReLU(),
            nn.Linear(128, 64), nn.ReLU(),
            nn.Linear(64, 128), nn.ReLU(),
            nn.Linear(128, 256), nn.ReLU(),
            nn.Linear(256, n_features), nn.Sigmoid(),
            nn.Unflatten(1, img_shape)
        )

    def forward(self, x):
        return self.autoencoder(x)

    def train(self, dataloader, optimizer, epochs, p_range):

        self.to(DEVICE)

        try:
            loss_fn = nn.MSELoss()

            progressbar = tqdm.trange(epochs)
            for epoch in progressbar:

                for x, _ in dataloader:
                    x = x.to(DEVICE)

                    # Inyección de ruido:
                    p = random.uniform(*p_range)
                    B, C, H, W = x.size()
                    noise = torch.empty([B, 1, H, W],
device=DEVICE).bernoulli_(p)
                    noisy_x = x * (1 - noise)

                    # Denoising:
```

```
                output = self(noisy_x)

                # Entrenamiento:
                loss = loss_fn(output, x)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

        except KeyboardInterrupt:
            print('Entrenamiento interrumpido.')
```

Se entrenará la red neuronal anterior sobre el dataset FashionMNIST durante 5 épocas:

```
# Datos de entrenamiento:
dataset = datasets.FashionMNIST('data', transform=transforms.ToTensor(),
download=True)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True,
drop_last=True)

# Red neuronal:
img_shape = dataset[0][0].shape
autoencoder = DAE(img_shape)

# Entrenamiento:
optimizer = Adam(autoencoder.parameters())
DAE.train(dataloader, optimizer, epochs=5, p_range=[0.3, 0.7])
```

Con el DAE entrenado, este puede ser utilizado como cualquier otro AE clásico. Como se comentó anteriormente, también es posible utilizarlo como un modelo de denoising (propiedad que no tiene un AE estándar):

```
autoencoder.eval()
autoencoder.cpu()

for _ in range(5):

    # Imagen real:
    idx = random.randint(0, len(dataset))
    true_img, _ = dataset[idx]

    # Imagen ruidosa:
    p = random.uniform(0.3, 0.7)
    noise = torch.empty(true_img.size()[1:]).bernoulli_(p)
    noisy_img = true_img * (1 - noise)
```

```
# Imagen reconstruida:
output = autoencoder(noisy_img.unsqueeze(0))
reconstructed_img = output[0].detach().cpu()

img_grid = make_grid([true_img, noisy_img, reconstructed_img])
plt.figure(figsize=(3, 1))
plt.imshow(1 - img_grid.permute(1, 2, 0))
plt.axis('off')
plt.show()
```

Figura 76: Denoising del DAE entrenado. A la izquierda de cada grupo se muestra la imagen original, en el centro la imagen ruidosa, y a la derecha la imagen reconstruida por el DAE.

Se observa que un DAE es capaz de reconstruir las partes faltantes de la imagen dada como entrada. Esta metodología de entrenamiento (reconstruir imagen original a partir de una versión ruidosa) tiene cierta similitud con la metodología usada para entrenar un modelo de difusión (en particular, con la reparametrización x_0-prediction que se estudiará en el respectivo capítulo). Más aún, un modelo de difusión puede verse como un DAE "mejorado", donde las diferencias son análogas a las diferencias que se observan entre un AE clásico y un AE variacional, aunque los modelos de difusión también incluyen una componente temporal que no se observa en un DAE estándar. De hecho, la función objetivo de un modelo de difusión será la extensión natural de la función objetivo de un VAE estándar.

9.2 Formulación de un VAE

Si bien un autoencoder clásico no tiene una formulación probabilística (en particular, no es una red bayesiana), la parte reconstructiva del decoder $z \mapsto x = D_\theta(z)$ es muy similar

en concepto a la parte generativa $p_\theta(x \mid z)$ de un modelo de variable latente $p_\theta(x, z) = p_\theta(z)p_\theta(x \mid z)$ (aunque en un AE clásico, el prior $p_\theta(z)$ no estaría definido y la parte generadora $p_\theta(x \mid z) \sim \delta_{D_\theta(z)}(z)$ sería determinista, de forma similar a como ocurre en una GAN estándar). Más aún, el encoder $x \mapsto z = E_\phi(x)$ permite tener una cantidad análoga a lo que sería la distribución posterior $p_\theta(z \mid x)$, la cual suele ser una cantidad intratable. Dados estos beneficios y limitaciones de un AE, un autoencoder variacional [1] introduce una formulación probabilística para estas cantidades, permitiendo obtener un nuevo paradigma generativo.

Dado que los VAEs son modelos de variable latente, se debe buscar un enfoque de entrenamiento alternativo a la maximización de la verosimilitud ya que la distribución marginal $p_\theta(x) = \int_{R^L} p_\theta(x, z)\,\mathrm{d}z$ suele ser intratable, por lo que no resulta posible entrenar este tipo de modelos usando el enfoque de máxima verosimilitud. Mientras las GANs evitan este problema entrenando el modelo generativo con un enfoque adversativo, los VAEs utilizan un enfoque basado en inferencia variacional. Para motivar este enfoque, se expresará la cantidad intratable $p_\theta(x)$ de forma conveniente. Descomponiendo la distribución conjunta $p_\theta(x, z)$ en orden contrario al natural (para poder tener la marginal $p_\theta(x)$):

$$p_\theta(x, z) = p_\theta(z \mid x)p_\theta(x) \Longrightarrow p_\theta(x) = \frac{p_\theta(x, z)}{p_\theta(z \mid x)},$$

por lo que podría utilizarse la expresión del lado derecho para el cálculo de la log-verosimilitud. Sin embargo, si bien el numerador es computable directamente (se puede calcular usando la factorización natural del modelo, $p_\theta(x, z) = p_\theta(z)p_\theta(x \mid z)$), el denominador no lo es. En efecto, la distribución posterior $p_\theta(z \mid x)$ debe ser calculada mediante la fórmula de Bayes, la cual requiere conocer la verosimilitud $p_\theta(x)$, que es precisamente lo que se busca calcular.

9.2.1 Inferencia variacional

Del análisis anterior, dado que la distribución posterior $p_\theta(z \mid x)$ no es tratable, no es posible computar eficientemente $p_\theta(x) = \frac{p_\theta(x,z)}{p_\theta(z \mid x)}$ para el entrenamiento por máxima verosimilitud. La solución que proponen los VAEs es estimar la posterior intratable, $p_\theta(z \mid x)$, mediante otro modelo neuronal, $q_\phi(z \mid x)$ (i.e., una red neuronal con parámetros ϕ aprende los parámetros de una nueva distribución $q_\phi(z \mid x)$ que busca parecerse a $p_\theta(z \mid x)$). De esta forma, si $q_\phi(z \mid x) \approx p_\theta(z \mid x)$, entonces se podría estimar la verosimilitud $p_\theta(x)$ mediante $\frac{p_\theta(x,z)}{q_\phi(z \mid x)}$.

Así, un VAE está compuesto por dos modelos probabilísticos, donde los parámetros de cada uno son aprendidos por una red neuronal diferente. Para definir una función objetivo para el entrenamiento de estas redes neuronales, se deben incluir las dos condiciones pedidas:

- $p_\theta(x, z)$ debe aproximar la distribución desconocida, $p_{\text{data}}(x)$, mediante la marginal $p_\theta(x)$. Esto se puede inducir pidiendo una log-verosimilitud alta.
- $q_\phi(z \mid x)$ debe aproximar la posterior desconocida $p_\theta(z \mid x)$. Esto se puede inducir pidiendo una baja divergencia de Kullback-Leibler entre ambas distribuciones.

Combinando ambos objetivos se obtiene la **ELBO** (también conocida como *variational lower bound*), la cual es la función objetivo utilizada en los VAEs, y posteriormente en los modelos de difusión. Esta función objetivo resultará ser tratable, lo que permitirá entrenar este tipo de modelos de forma eficiente. Además, sus distintas descomposiciones permitirán darle distintas interpretaciones y modificaciones, las cuales influirán directamente en los modelos entrenados.

Definición 4 (ELBO). Sea $p_\theta(x, z) = p_\theta(z)p_\theta(x \mid z)$ un modelo de variable latente y $q_\phi(z \mid x)$ otro modelo que busca aproximar $p_\theta(z \mid x)$. Para una muestra $x \in \mathbb{R}^D$, se define

$$\text{ELBO}(x) \coloneqq \log p_\theta(x) - \text{D}_{\text{KL}}\big(q_\phi(z \mid x) \,\|\, p_\theta(z \mid x)\big)$$

Notar que la desigualdad de Gibbs garantiza que $\text{D}_{\text{KL}}\big(q_\phi(z \mid x) \,\|\, p_\theta(z \mid x)\big) \geq 0$, por lo que $\text{ELBO}(x) \leq \log p_\theta(x)$, justificando así el nombre de la ELBO como una cota inferior de la log-verosimilitud (*Evidence Lower BOund*). Por otro lado, notar que la ELBO también depende de los parámetros neuronales θ y ϕ, pero estos se omiten en la notación por simplicidad.

Con esto, la función objetivo que se optimiza al entrenar un VAE es la ELBO esperada sobre la distribución de los datos $p_{\text{data}}(x)$:

$$\max_{\theta,\phi} \mathbb{E}_{p_{\text{data}}(x)}[\text{ELBO}(x)],$$

donde la esperanza es aproximada, como es usual, con una estimación de Monte Carlo utilizando un conjunto de entrenamiento $\mathcal{D} = \{x^1, \cdots, x^N\} \subset \mathbb{R}^D$ generado desde $p_{\text{data}}(x)$:

$$\mathbb{E}_{p_{\text{data}}(x)}[\text{ELBO}(x)] \approx \frac{1}{N} \sum_{n=1}^{N} \text{ELBO}(x^n)$$

Es importante destacar que, si bien la log-verosimilitud $\log p_\theta(x)$ por sí sola no es tratable, cuando se combina con el objetivo de inferencia variacional, $\text{D}_{\text{KL}}\big(q_\phi(z \mid x) \,\|\, p_\theta(z \mid x)\big)$ (el cual tampoco es tratable) se obtiene una función objetivo que sí es tratable, lo que vuelve a la ELBO la función objetivo por defecto para entrenar un VAE. En efecto, recordando que $\texttt{kl}\ (q, p) = \mathbb{E}_q\left[\log \frac{q}{p}\right]$:

$$\begin{aligned}
\mathrm{ELBO}(x) &= \log p_\theta(x) + \mathbb{E}_{q_\phi(z\,|\,x)}\left[\log\left(\frac{p_\theta(z\,|\,x)}{q_\phi(z\,|\,x)}\right)\right] \\
&= \mathbb{E}_{q_\phi(z\,|\,x)}\left[\log\left(\frac{p_\theta(x,z)}{q_\phi(z\,|\,x)}\right)\right] \\
&= \mathbb{E}_{q_\phi(z\,|\,x)}[\log p_\theta(x\,|\,z)] + \mathbb{E}_{q_\phi(z\,|\,x)}\left[\log\left(\frac{p_\theta(z)}{q_\phi(z\,|\,x)}\right)\right] \\
&= \underbrace{\mathbb{E}_{q_\phi(z\,|\,x)}[\log p_\theta(x\,|\,z)]}_{\text{término de reconstrucción}} - \underbrace{\mathrm{D}_{\mathrm{KL}}\big(q_\phi(z\,|\,x)\,\|\,p_\theta(z)\big)}_{\text{prior matching}}
\end{aligned}$$

Notar que todos los términos en la última igualdad se pueden computar, lo que permite entrenar ambos modelos de manera conjunta. En particular, la esperanza puede ser aproximada usando muestras generadas desde $q_\phi(z\,|\,x)$, mientras que la divergencia de Kullback-Leibler puede ser calculada en forma cerrada si se eligen modelos $q_\phi(z\,|\,x)$ y $p_\theta(z)$ convenientes.

9.2.2 Modelos paramétricos

El desarrollo hecho hasta el momento es agnóstico a las distribuciones que se elijan para $p_\theta(z)$, $p_\theta(x\,|\,z)$ y $q_\phi(z\,|\,x)$. Estas distribuciones se eligen de manera conveniente para poder evaluar eficientemente la ELBO usando la descomposición anterior, lo cual requiere que se cumplan las siguientes condiciones:

- Debe ser fácil de generar muestras desde $q_\phi(z\,|\,x)$ para poder aproximar la esperanza en el término de reconstrucción.
- $q_\phi(z\,|\,x)$ y $p_\theta(z)$ deben pertenecer a una buena familia de distribuciones (e.g. la familia exponencial) para que el término de prior matching se pueda obtener de forma cerrada.

Para la variable latente incondicional, z, es usual considerar una distribución gaussiana estándar:

$$p_\theta(z) \sim \mathcal{N}(0, \mathrm{I}_L)$$

Con esta elección, el término de prior matching en la ELBO se puede obtener de forma cerrada si luego se elige $q_\phi(z\,|\,x)$ también gaussiana. Como es usual, se escribirá $p(z)$ en vez de $p_\theta(z)$ para indicar que la distribución está fija y no posee parámetros entrenables.

Encoder

De acuerdo a la elección del prior $p(z)$, es conveniente elegir la distribución posterior aproximada, $p_\phi(z \mid x)$, también como una distribución gaussiana con el fin de poder calcular el término prior matching de forma cerrada. Más aún, la manifold hypothesis motiva a utilizar una matriz de covarianza diagonal si se asume que las características *esenciales* de una muestra (variables latentes) son elegidas de manera independiente. Por lo tanto, un VAE considera la siguiente distribución para el modelo de inferencia aproximada:

$$q_\phi(z \mid x) \sim \mathcal{N}\Big(\mu_\phi(x), \operatorname{diag}\Big(\sigma^2_\phi(x)\Big)\Big)$$

donde $\mu_\phi : \mathbb{R}^D \to \mathbb{R}^L$ y $\sigma_\phi : \mathbb{R}^D \to \mathbb{R}^L_{++}$ son redes neuronales que aprenden el vector de medias y el vector de varianzas de la distribución gaussiana $q_\phi(z \mid x)$[16].

Es importante notar que la distribución posterior real $p_\theta(z \mid x)$ que busca aproximar este modelo no tiene por qué ser gaussiana (en general no lo es) pero se elige este modelo por conveniencia en la función objetivo. Además, el uso de una matriz de covarianzas diagonal es una práctica usual ya que su determinante es fácil de calcular, el cual será necesario en el cálculo del término prior matching, $\mathrm{D}_{\mathrm{KL}}\big(q_\phi(z \mid x) \,\|\, p_\theta(z)\big)$. Por último, esta elección de $q_\phi(z \mid x)$ permitirá aplicar el *truco de la reparametrización*, el cual resulta ser un requisito esencial para poder entrenar un VAE con algoritmos de gradiente.

Decoder

La distribución $p_\theta(x \mid z)$ dependerá de la naturaleza de la distribución que se busca aprender, $p_{\text{data}}(x)$. Aquí se considerará tanto el caso continuo (usando el dataset de juguete 2D) como el discreto (usado en la generación de imágenes). Notar que esta distribución solo aparece en el término de reconstrucción de la ELBO, por lo la única diferencia entre el entrenamiento de un VAE para datos continuos y un VAE para imágenes estará en la expresión utilizada para este término ya que el término de prior matching será el mismo en ambos casos.

Si las muestras $x \in \mathbb{R}^D$ son vectores cuyas coordenadas pueden tomar cualquier valor en el intervalo $(-\infty, +\infty)$, es usual considerar una distribución gaussiana,

[16] Recordar que para $a \in \mathbb{R}^L$

$$\operatorname{diag}(a) \coloneqq \begin{pmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_L \end{pmatrix}$$

$$p_\theta(x \mid z) \sim \mathcal{N}(\mu_\theta(z), \sigma_0^2 \mathrm{I}_D)$$

donde $\mu_\theta : \mathbb{R}^L \to \mathbb{R}^D$ es una red neuronal que aprende el vector de medias de la distribución $p_\theta(x \mid z)$, mientras que $\sigma_0^2 > 0$ es un parámetro fijo (usualmente pequeño). Si bien es posible aprender también la matriz de covarianza de $p_\theta(x \mid z)$, aquí se está considerando fija e isotrópica ya que simplifica la función de costo y, en consecuencia, la implementación. Por otra parte, es importante indicar que considerar una distribución gaussiana no limita la capacidad de generación del modelo ya que la verdadera complejidad viene codificada en el vector de medias, $\mu_\theta(z) \in \mathbb{R}^D$, el cual es aprendido por la red neuronal $\mu_\theta : \mathbb{R}^L \to \mathbb{R}^D$.

Por otro lado, cuando las muestras $x \in \mathbb{R}^D$ son imágenes (con $D = \text{alto} \cdot \text{ancho}$, y asumiendo un único canal de color por simplicidad), es usual modelar cada pixel $x_d \in [0, 1]$, $d \in \{1, ..., D\}$ como una distribución Bernoulli. En particular, dado el valor de la variable latente $z \in \mathbb{R}^L$, cada pixel se considera independiente del resto de pixeles. Es decir, se utiliza la siguiente distribución para el decoder cuando se trabaja con imágenes:

$$p_\theta(x \mid z) = \prod_{d=1}^{D} p_\theta(x_d \mid z), \quad \text{donde} \quad p_\theta(x_d \mid z) \sim \text{Bernoulli}(r_\theta(z)_d)$$

Aquí, $r_\theta : \mathbb{R}^L \to [0, 1]^D$ es una red neuronal que aprende los parámetros $r_\theta(z)_d \in [0, 1]$ de la distribución Bernoulli de cada pixel $x_d \in [0, 1]$ de la imagen.

9.2.3 ELBO en un VAE

Teniendo definido cada uno de los modelos paramétricos usados en un VAE, es posible desarrollar los términos de reconstrucción y prior matching de la ELBO para obtener las expresiones que se utilizan en las implementaciones. Se partirá desarrollando el término de prior matching y luego el término de reconstrucción.

Prior matching

Dado que las distribuciones $q_\phi(z \mid x)$ y $p_\theta(z)$ son ambas gaussianas, el término de prior matching, $\mathrm{D}_{\mathrm{KL}}\big(q_\phi(z \mid x) \,\|\, p_\theta(z)\big)$, puede ser calculado en forma cerrada. Para eso, se utilizará el siguiente resultado clásico:

Teorema 5. *Dadas dos distribuciones gaussianas en $\mathbb{R}^L$, su divergencia de Kullback-Leibler tiene forma cerrada[17]:*

[17]Para $A \in \mathcal{M}_{m,n}(\mathbb{R})$, $\det(A) \in \mathbb{R}$ es el determinante de A y corresponde al producto de sus valores propios. Del mismo modo, $\mathrm{tr}(A) \in \mathbb{R}$ es la traza de A y corresponde a la suma de sus valores propios, lo cual resulta ser equivalente a la suma de los elementos de su diagonal.

$$\begin{aligned}&\mathrm{D}_{\mathrm{KL}}(\mathcal{N}(\mu_1,\Sigma_1)\,\|\,\mathcal{N}(\mu_2,\Sigma_2))\\&=\frac{1}{2}\Big[(\mu_1-\mu_2)^\top\Sigma_2^{-1}(\mu_1-\mu_2)+\mathrm{tr}(\Sigma_1\Sigma_2^{-1})-L-\log(|\det(\Sigma_1\Sigma_2^{-1})|)\Big]\end{aligned}$$

Aplicando este resultado a la posterior aproximada $q_\phi(z\,|\,x)\sim\mathcal{N}\big(\mu_\phi(x),\mathrm{diag}\big(\sigma_\phi^2(x)\big)\big)$ y al prior latente $p_\theta(z)\sim\mathcal{N}(0,\mathrm{I}_L)$ se obtiene una expresión cerrada para el término de prior matching:

$$\begin{aligned}\mathrm{D}_{\mathrm{KL}}\big(q_\phi(z\,|\,x)\,\|\,p_\theta(z)\big)&=\frac{1}{2}\left[\mu_\phi(x)^\top\mu_\phi(x)+\sum_{l=1}^{L}\sigma_\phi^2(x)-L-\log\left(\prod_{l=1}^{L}\sigma_\phi^2(x)_l\right)\right]\\&=\frac{1}{2}\Big(\big\|\mu_\phi(x)\big\|^2+\big\|\sigma_\phi(x)\big\|^2\Big)-\sum_{l=1}^{L}\log\sigma_\phi(x)_l+\text{constante}\end{aligned}$$

En la primera igualdad se usó que el determinante de una matriz diagonal es el producto de su diagonal, mientras que en la segunda igualdad se identificaron dos formas equivalentes de la norma de un vector[18].

Término de reconstrucción

Antes de desarrollar el término de reconstrucción $\mathbb{E}_{q_\phi(z\,|\,x)}[\log p_\theta(x\,|\,z)]$ sustituyendo el respectivo modelo $p_\theta(x\,|\,z)$ en la esperanza, se reescribirá la distribución $q_\phi(z\,|\,x)$ de forma conveniente para que la variable aleatoria $q_\phi(z\,|\,x)$ sobre la que se calcula la esperanza $\mathbb{E}_{q_\phi(z\,|\,x)}[\log p_\theta(x\,|\,z)]$ no dependa de ϕ. Esto permitirá, como se verá en el próximo capítulo, no tener problemas al momento de entrenar la red neuronal.

Considerando que $q_\phi(z\,|\,x)\sim\mathcal{N}\big(\mu_\phi(x),\mathrm{diag}\big(\sigma_\phi^2(x)\big)\big)$, esta variable aleatoria se puede reparametrizar esta usando el cambio de variable $z=\mu_\phi(x)+\sigma_\phi\odot\epsilon$, donde $\epsilon\sim\mathcal{N}(0,\mathrm{I}_L)$ es una nueva variable aleatoria independiente de ϕ y $\odot$ es el producto de Hadamard. Con esta sustitución, el término de reconstrucción se puede escribir como

$$\mathbb{E}_{q_\phi(z\,|\,x)}[\log p_\theta(x\,|\,z)]=\mathbb{E}_{\epsilon\sim\mathcal{N}(0,\mathrm{I}_L)}\big[\log p_\theta\big(x\,\big|\,\mu_\phi(x)+\sigma_\phi\odot\epsilon\big)\big]$$

Además, como es usual, la esperanza anterior se puede estimar utilizando una aproximación de Monte Carlo:

$$\mathbb{E}_{q_\phi(z\,|\,x)}[\log p_\theta(x\,|\,z)]\approx\frac{1}{K}\sum_{k=1}^{K}\log p_\theta\big(x\,\big|\,\mu_\phi(x)+\sigma_\phi(x)\odot\epsilon_k\big),$$

[18]Recordar que, para $z\in\mathbb{R}^L$, $\|z\|^2=\sum_{l=1}^{L}z_l^2=\langle z,z\rangle=z^\top z$.

donde $\epsilon_k \sim \mathcal{N}(0, \mathrm{I}_L)$ para $k \in \{1, ..., K\}$. Usualmente es suficiente considerar $K = 1$, aproximando el término de reconstrucción con una única muestra de la variable latente $z \sim q_\phi(z \mid x)$.

Para concluir con la formulación de un VAE y pasar a la implementación, solo falta desarrollar el término $\log p_\theta(x \mid z)$ dentro de la esperanza, el cual dependerá si se está trabajando con datos continuos o con imágenes.

Como se mencionó anteriormente, si $p_{\text{data}}(x)$ es una distribución continua, se suele considerar un decoder con distribución $p_\theta(x \mid z) \sim \mathcal{N}(\mu_\theta(z), \sigma_0^2 \mathrm{I}_D)$, con $\sigma_0^2 > 0$ un hiperparámetro, luego:

$$p_\theta(x \mid z) = \frac{1}{\sqrt{(2\pi\sigma_0^2)^D}} \exp\left(-\frac{1}{2\sigma_0^2}\|x - \mu_\theta(z)\|^2\right).$$

Por lo tanto, el término de reconstrucción se reduce, salvo constante aditiva, a una diferencia de cuadrados, siendo similar a la función objetivo que se utiliza en un autoencoder clásico:

$$\mathbb{E}_{q_\phi(z \mid x)}[\log p_\theta(x \mid z)] = -\frac{1}{2\sigma_0^2}\mathbb{E}_{q_\phi(z \mid x)}\left[\|x - \mu_\theta(z)\|^2\right] + \text{constante}$$

Notar que si la varianza $\sigma_0^2 > 0$ también fuera un parámetro entrenable (i.e., $\sigma_0 = \sigma_\theta(z)$), el término $-\frac{D}{2}\log\big(2\pi\sigma_\theta^2(z)\big)$ en $\log p_\theta(x \mid z)$ ya no sería constante, por lo que se debería incluir en la implementación. En este caso, quedó como un hiperparámetro que balancea la importancia del término de reconstrucción con respecto al término de prior matching.

Por otro lado, si $p_{\text{data}}(x)$ es una distribución definida sobre imágenes, cada pixel $x_d \in [0, 1]$ (condicionado a z) se suele modelar como una distribución Bernoulli con parámetro $r_d = r_\theta(z)_d$, por lo que la función de masa para dicho pixel es $p_\theta(x_d \mid z) = r_d^{x_d} \cdot (1 - r_d)^{1 - x_d}$. Luego:

$$\mathbb{E}_{q_\phi(z \mid x)}[\log p_\theta(x \mid z)] = \mathbb{E}_{q_\phi(z \mid x)}\left[\sum_{d=1}^{D}(x_d \cdot \log r_d + (1 - x_d) \cdot \log(1 - r_d))\right]$$

En particular, esta función objetivo resulta ser equivalente a la entropía cruzada binaria (salvo factor de ponderación negativo).

9.3 Implementación de un VAE

En esta sección se implementarán los dos tipos de VAEs vistos.

9.3.1 VAE gaussiano

Como ejemplo de data continua se utilizará el mismo dataset de juguete 2D usado anteriormente al implementar un AE clásico.

Redes neuronales

Dado que el prior $p(z) \sim \mathcal{N}(0, \mathrm{I}_L)$ está fijo, solo es necesario entrenar redes neuronales que aprendan los parámetros del encoder $q_\phi(z \mid x)$ y del decoder $p_\theta(x \mid z)$. Considerando que se está trabajando con un dataset simple, es suficiente utilizar redes neuronales fully connected de pocas capas. Además, como es usual, el encoder tendrá una cantidad descendente de neuronas, mientras que el decoder tendrá una cantidad ascendente, replicando el cuello de botella de un autoencoder estándar.

Para el encoder $q_\phi(z \mid x) \sim \mathcal{N}\Big(\mu_\phi(x), \mathrm{diag}\Big(\sigma^2_\phi(x)\Big)\Big)$ es necesario aprender un vector de medias $\mu_\phi(x) \in \mathbb{R}^L$ y un vector de desviacions estándar $\sigma_\phi(x) \in \mathbb{R}^L_{++}$. Sin embargo, para evitar la restricción de positividad de $\sigma_\phi(x)$, en la práctica se suele aprender el vector irrestricto $\log \sigma_\phi(x) = \big(\log \sigma_\phi(x)_1, \cdots, \log \sigma_\phi(x)_L\big) \in \mathbb{R}^L$ en vez de aprender directamente $\sigma_\phi(x) \in \mathbb{R}^L_{++}$. Por otra parte, se utilizará una única red neuronal $\text{Encoder}_\phi : \mathbb{R}^D \to \mathbb{R}^L \times \mathbb{R}^L$ que aprenda los dos parámetros del encoder, $\big(\mu_\phi(x), \sigma_\phi(x)\big) \in \mathbb{R}^L \times \mathbb{R}^L_{++}$, al mismo tiempo, lo cual es una práctica usual en la implementación de VAEs.

```
class Encoder(nn.Module):

    def __init__(self, data_dim, latent_dim):
        super().__init__()

        self.mlp = nn.Sequential(
            nn.Linear(data_dim, 128), nn.ReLU(),
            nn.Linear(128, 64), nn.ReLU(),
            nn.Linear(64, 32), nn.ReLU(),
        )
        self.mean = nn.Linear(32, latent_dim)
        self.log_std = nn.Linear(32, latent_dim)

    def forward(self, x):
        x = self.mlp(x)
        mean = self.mean(x)
        logstd = self.log_std(x)
        return mean, logstd.exp()
```

Con respecto a la red neuronal asociada al decoder $p_\theta(x \mid z)$, esta dependerá de la naturaleza de $x \in \mathbb{R}^D$. Dado que en este caso se están considerando datos continuos, $p_\theta(x \mid$

$z) \sim \mathcal{N}(\mu_\theta(z), \sigma_0^2 I)$, por lo que solo es necesario aprender el vector de medias $\mu_\theta(z) \in \mathbb{R}^D$. Además, dado que este vector buscado no tiene restricciones, no hace falta agregar ninguna una función de activación en la salida de la red neuronal.

```
class Decoder(nn.Module):

    def __init__(self, data_dim, latent_dim):
        super().__init__()

        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 32), nn.ReLU(),
            nn.Linear(32, 64), nn.ReLU(),
            nn.Linear(64, 128), nn.ReLU(),
            nn.Linear(128, data_dim)
        )

    def forward(self, z):
        mean = self.decoder(z)
        return mean
```

Clase para el VAE

Teniendo las redes neuronales para el encoder y el decoder, se implementará una clase `VAE` que contenga los métodos para el entrenamiento de las redes neuronales y para la generación de nuevas muestras. En este caso, $D = 2$ y se considerará una dimensión latente $L = 16$. Además, por elección empírica se considerará $\sigma_0 = 0.1$.

```
class VAE:

    def __init__(self, data_dim, latent_dim):

        self.latent_dim = latent_dim

        self.encoder = Encoder(data_dim, latent_dim)
        self.decoder = Decoder(data_dim, latent_dim)
        self.encoder_optimizer = optim.AdamW(self.encoder.parameters())
        self.decoder_optimizer = optim.AdamW(self.decoder.parameters())

        self.decoder_std = 0.1

    def train(self, iters):

        for _ in range(iters):
```

```
            x = get_batch()

            # Prior matching:
            encoder_mean, encoder_std = self.encoder(x)
            prior_matching = 1/2 * (encoder_mean.norm(dim=-1) ** 2 +
encoder_std.norm(dim=-1) ** 2) - encoder_std.log().sum(dim=-1)

            # Reconstruction term:
            z = encoder_mean + encoder_std * torch.randn_like(encoder_mean)
            decoder_mean = self.decoder(z)
            reconstruction_term = - 1 / (2 * self.decoder_std ** 2) * (x -
decoder_mean).norm(dim=-1) ** 2

            # ELBO:
            elbo = reconstruction_term - prior_matching
            loss = - elbo.mean()

            # Optimización:
            self.encoder_optimizer.zero_grad()
            self.decoder_optimizer.zero_grad()
            loss.backward()
            self.encoder_optimizer.step()
            self.decoder_optimizer.step()

    def generate_samples(self, n_samples):
        z = torch.randn(n_samples, self.latent_dim)
        x_mean = self.decoder(z)
        x_dec = x_mean# + self.decoder_std * torch.randn_like(x_mean)
        return x_dec
```

Entrenamiento y generación

Con la clase anterior definida, se puede entrenar un VAE sobre el dataser de juguete:

```
vae = VAE()
vae.train(iters=5000)
```

Una vez entrenado el modelo se pueden generar nuevas muestras usando `generate_samples`. Se observa que, a diferencia de un AE clásico, el VAE implementado es capaz de generar nuevas muestras luego del entrenamiento. Notar también que las muestras generadas son más dispersas y no se concentran alrededor de la distribución original de los datos. En el caso de imágenes, esta dispersión en la distribución aprendida se refleja en las imágenes generadas, las cuales suelen ser más borrosas que las imágenes generadas por una GAN.

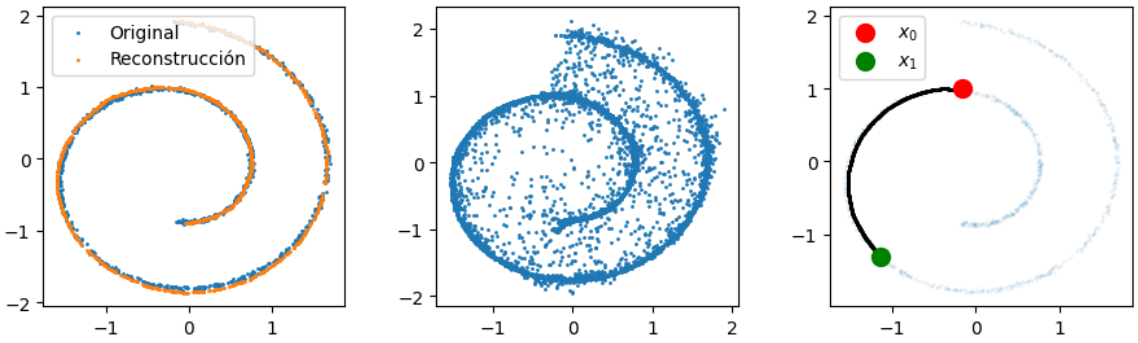

Figura 77: Reconstrucción de muestras del dataset (izquierda), muestras generadas (centro) e interpolación latente (derecha) usando un VAE entrenado sobre el dataset de juguete 2D.

Por otro lado, es posible evaluar la capacidad de reconstrucción de un VAE, la cual resulta ser siempre de menor calidad que en un AE clásico (donde el objetivo es precisamente reconstruir). Esto se debe principalmente a la presencia del término prior matching en la ELBO, el cual obliga al modelo a ceder un poco en la calidad de la reconstrucción a cambio de un espacio latente mejor estructurado (producto del regularizador de prior matching).

En este caso, la reconstrucción se puede implementar con el siguiente método adicional sobre la clase `VAE`:

```
def reconstruct(self, x):
    z_mean, z_std = self.encoder(x)
    z = z_mean# + z_std * torch.randn_like(z_mean)
    x_mean = self.decoder(z)
    x_dec = x_mean# + self.decoder_std * torch.randn_like(x_mean)
    return x_dec
```

Por otro lado, dada la estructura gaussiana inducida sobre el espacio latente, la interpolación latente en un VAE, a diferencia de un AE clásico, sí resulta en una interpolación con sentido semántico. En este caso, la interpolación latente se puede realizar con el siguiente método adicional sobre la clase `VAE`:

```
def latent_interpolation(self, x0, x1, t):

        # Latentes:
        z_0, _ = self.encoder(x0.unsqueeze(0))
        z_1, _ = self.encoder(x1.unsqueeze(0))

        # Interpolación:
        z_t = (1 - t) * z_0 + t * z_1
        x_t = self.decoder(z_t).squeeze(0)
```

```
        return x_t
```

9.3.2 VAE para imágenes

Ahora se implementará un VAE para trabajar con imágenes. Notar que, en este caso, el cálculo de la ELBO cambia ya que el término de reconstrucción $\mathbb{E}_{q_\phi(z\,|\,x)}[\log p_\theta(x\,|\,z)]$ ahora considera una distribución $p_\theta(x\,|\,z)$ discreta. Además, para agregar más flexibilidad, se implementará un VAE condicional[19] para poder indicar la clase a la que debe pertenecer la imagen que se generará.

Las bibliotecas adicionales que se utilizarán son las siguientes:

```
import torch.nn.functional as F
from torch.utils.data import Data**Loader
from torchvision import datasets, transforms, utils
import tqdm

DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Como datos de entrenamiento, se utilizará el dataset Fashion MNIST escalado a 32×32 para poder usar redes convolucionales más profundas en el encoder y decoder.

```
transf = transforms.Compose([transforms.Resize((32, 32)),
transforms.ToTensor()])
dataset = datasets.FashionMNIST('data', train=True, transform=transf,
download=True)
dataloader = DataLoader(dataset, batch_size=256, shuffle=True,
drop_last=True)

def show_batch(images):
    grid_tensor = utils.make_grid(images, nrow=10)
    plt.axis('off')
    plt.imshow(1-grid_tensor.permute(1, 2, 0))
    plt.tight_layout()
    plt.show()

# Ejemplo:
batch_x, batch_y = next(iter(dataloader))
show_batch(batch_x[:50])
```

[19] Recordar que toda red bayesiana $p(x_1, ..., x_N) = \prod_{n=1}^{N} p(x_n\,|\,\text{Pa}(x_n))$ puede ser extendida a su forma condicional, $p(x_1, ..., x_N\,|\,y) = \prod_{n=1}^{N} p(x_n\,|\,\text{Pa}(x_n), y)$, extendiendo las redes neuronales que aprenden los parámetros de $p(x_n\,|\,\text{Pa}(x_n))$ para que ahora también reciban la condición y.

Figura 78: Muestras del dataset Fashion MNIST.

Redes neuronales

El encoder consistirá en una red convolucional estándar, la cual pasará por una transformación lineal en la última capa para obtener los parámetros de media y varianza de $q_\phi(z \mid x)$. Por otro lado, la condición de clase será codificada como un vector one-hot, el cual será expandido y concatenado en la dimensión de los canales de la imagen (i.e., cada pixel aumentará su cantidad de canales en 10, donde 10 es la cantidad de clases distintas del dataset MNIST):

```
class ImageEncoder(nn.Module):

    def __init__(self, image_size, latent_dim, n_classes):
        super().__init__()

        self.n_classes = n_classes

        conv = lambda in_ch, out_ch: nn.Conv2d(in_ch, out_ch,
kernel_size=3, stride=2, padding=1)
        c, h, w = image_size
        flatten_dim = 64 * (h // 8) * (w // 8)

        self.encoder = nn.Sequential(
            conv(c + n_classes, 16), nn.ReLU(),
            conv(16, 32), nn.ReLU(),
            conv(32, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(flatten_dim, 2 * latent_dim)
        )

    def forward(self, x, y):
```

```
        batch_size, c, h, w = x.shape
        y_emb = F.one_hot(y, self.n_classes)  # [batch_size, n_classes].
        y_emb = y_emb[:, :, None, None].expand(batch_size, self.n_classes,
h, w)  # [batch_size, n_classes, h, w].

        x_cond = torch.cat([x, y_emb], dim=1)  # [batch_size, c +
n_classes, h, w].

        mean, logstd = self.encoder(x_cond).chunk(2, dim=-1)
        std = logstd.exp()
        return mean, std
```

Para el decoder, se utilizará una red convolucional simétrica a la usada en el encoder:

```
class ImageDecoder(nn.Module):

    def __init__(self, image_size, latent_dim, n_classes):
        super().__init__()

        c, h, w = image_size
        flatten_dim = 64 * (h // 8) * (w // 8)
        self.n_classes = n_classes

        deconv = lambda in_ch, out_ch: nn.ConvTranspose2d(in_ch, out_ch, 3,
2, 1, 1)

        self.decoder = nn.Sequential(
            nn.Linear(latent_dim + n_classes, flatten_dim),
            nn.ReLU(),
            nn.Unflatten(1, (64, h // 8, w // 8)),
            deconv(64, 32), nn.ReLU(),
            deconv(32, 16), nn.ReLU(),
            deconv(16, c), nn.Sigmoid()
        )

    def forward(self, z, y):
        y_onehot = F.one_hot(y, self.n_classes)  # [batch_size, n_classes].
        z_cond = torch.cat([z, y_onehot], dim=-1)  # [batch_size,
latent_dim + n_classes].
        x_hat = self.decoder(z_cond)
        return x_hat
```

Clase `ImageVAE` para imágenes

Ahora, se implementará una clase `ImageVAE` análoga a la anterior. El cálculo de la función de pérdida se hará en el método `_calc_loss` para no hacer tan extenso el método `train`.

```
class ImageVAE:

    def __init__(self, image_size=(1, 32, 32), latent_dim=128,
n_classes=10):

        self.image_size = image_size
        self.latent_dim = latent_dim

        self.encoder = ImageEncoder(image_size, latent_dim, n_classes)
        self.decoder = ImageDecoder(image_size, latent_dim, n_classes)

        self.encoder_optimizer = optim.AdamW(self.encoder.parameters())
        self.decoder_optimizer = optim.AdamW(self.decoder.parameters())

        self.encoder.to(DEVICE)
        self.decoder.to(DEVICE)

    def train(self, dataloader, epochs):

        self.encoder.train()
        self.decoder.train()

        try:
            for epoch in tqdm.trange(epochs):

                for x, y in dataloader:
                    x, y = x.to(DEVICE), y.to(DEVICE)
                    loss = self._calc_loss(x, y)

                    # Optimización:
                    self.encoder_optimizer.zero_grad()
                    self.decoder_optimizer.zero_grad()
                    loss.backward()
                    self.encoder_optimizer.step()
                    self.decoder_optimizer.step()

        except KeyboardInterrupt:
            print('Entrenamiento interrumpido.')
```

```
    def _calc_loss(self, x, y):

        # Prior matching:
        encoder_mean, encoder_std = self.encoder(x, y)
        prior_matching = 1/2 * (encoder_mean.norm(dim=-1) ** 2 +
encoder_std.norm(dim=-1) ** 2) - encoder_std.log().sum(dim=-1)

        # Término de reconstrucción:
        z = encoder_mean + encoder_std * torch.randn_like(encoder_mean)
        x_hat = self.decoder(z, y)
        x = x.flatten(start_dim=1)  # [batch_size, c * h * w]
        x_hat = x_hat.flatten(start_dim=1)  # [batch_size, c * h * w]
        reconstruction_term = (x * x_hat.log() + (1 - x) * (1 -
x_hat).log()).sum(dim=-1)

        elbo = reconstruction_term - prior_matching
        return - elbo.mean()

    def generate_samples(self, y, n_samples):

        self.decoder.eval()

        with torch.no_grad():
            y = torch.tensor(y, device=DEVICE).expand(n_samples)
            z = torch.randn(n_samples, self.latent_dim, device=DEVICE)
            x_hat = self.decoder(z, y)

        return x_hat
```

Entrenamiento y generación

Con la clase anterior implementada, se entrenará el VAE condicional durante 50 épocas:

```
vae = ImageVAE()
vae.train(dataloader, epochs=50)
```

Con el modelo entrenado, se generarán 10 muestras para cada etiqueta de clase $y \in \{0, \cdots, 9\}$:

```
samples = [vae.generate_samples(y, n_samples=10) for y in range(10)]
samples = torch.cat(samples, dim=0)
show_batch(samples.cpu())
```

Figura 79: Generación condicional usando el VAE entrenado sobre Fashion MNIST.

Con los modelos encoder y decoder entrenados, es posible revisar la reconstrucción realizada por el VAE. Para esto, se elige un par de muestras originales, se pasan por el encoder para obtener sus respectivas representaciones latentes y luego, dichas representaciones latentes se pasan por el decoder para generar nuevas muestras. Si el VAE está bien entrenado, las muestras generadas deben ser similares a las muestras originales:

```
vae.encoder.eval()

batch_x, batch_y = next(iter(dataloader))
x = batch_x[:10].to(DEVICE)
y = batch_y[:10].to(DEVICE)

encoder_mean, encoder_std = vae.encoder(x, y)
z = encoder_mean + encoder_std * torch.randn_like(encoder_mean)
x_hat = vae.decoder(z, y)
```

```
imgs = torch.cat([x, x_hat], dim=0)
show_batch(imgs.cpu())
```

Figura 80: Reconstrucción de muestras usando el VAE entrenado sobre Fashion MNIST.

Se observa que las muestras reconstruidas (abajo) son muy similares a las muestras originales (arriba), lo que indica que tanto el modelo generador (encoder) como el modelo codificador (encoder) están bien entrenados.

En el siguiente capítulo se aprovechará la existencia del encoder para poder realizar modificación de atributos de forma fácil. Además, se revisarán algunas propiedades y variantes del VAE clásico revisado en este capítulo.

Referencias

[1] D. P. Kingma y M. Welling, «Auto-Encoding Variational Bayes», *arXiv preprint arXiv:1312.6114*, 2013, [En línea]. Disponible en: https://arxiv.org/abs/1312.6114

[2] I. Sutskever, O. Vinyals, y Q. V. Le, «Sequence to Sequence Learning with Neural Networks», en *Advances in Neural Information Processing Systems (NeurIPS)*, 2014. [En línea]. Disponible en: https://arxiv.org/abs/1409.3215

[3] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, y B. Ommer, «High-Resolution Image Synthesis with Latent Diffusion Models», en *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. [En línea]. Disponible en: https://arxiv.org/abs/2112.10752

[4] J.-Y. Zhu, T. Park, P. Isola, y A. A. Efros, «Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks», en *IEEE International Conference on Computer Vision (ICCV)*, 2017. [En línea]. Disponible en: https://arxiv.org/abs/1703.10593

[5] A. Polyak y others, «Movie Gen: A Cast of Media Foundation Models», *arXiv preprint arXiv:2410.13720*, 2024, [En línea]. Disponible en: https://arxiv.org/abs/2410.13720

[6] D. Ha y J. Schmidhuber, «World Models», *arXiv preprint arXiv:1803.10122*, 2018, [En línea]. Disponible en: https://arxiv.org/abs/1803.10122

[7] D. Valevski, Y. Leviathan, M. Arar, y S. Fruchter, «Diffusion Models Are Real-Time Game Engines», *arXiv preprint arXiv:2408.14837*, 2024, [En línea]. Disponible en: https://arxiv.org/abs/2408.14837

[8] Wang y others, «Large Language Models for Robotics: A Survey», *arXiv preprint arXiv:2401.04334*, 2024, [En línea]. Disponible en: https://arxiv.org/abs/2401.04334

[9] J. Jumper y others, «Highly accurate protein structure prediction with AlphaFold», *Nature*, vol. 596, pp. 583-589, 2021, [En línea]. Disponible en: https://www.nature.com/articles/s41586-021-03819-2

[10] Alakhdar y others, «Diffusion Models in De Novo Drug Design», *arXiv preprint arXiv:2406.08511*, 2024, [En línea]. Disponible en: https://arxiv.org/abs/2406.08511

[11] Li y others, «API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs», *arXiv preprint arXiv:2304.08244*, 2023, [En línea]. Disponible en: https://arxiv.org/abs/2304.08244

[12] V. Ram Somnath y others, «Aligned Diffusion Schrödinger Bridges», *arXiv preprint arXiv:2302.11419*, 2023, [En línea]. Disponible en: https://arxiv.org/abs/2302.11419

[13] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, y M. Chen, «Hierarchical Text-Conditional Image Generation with CLIP Latents (DALL-E 2)», technical report, 2022. [En línea]. Disponible en: https://cdn.openai.com/papers/dall-e-2.pdf

[14] A. Vaswani *et al.*, «Attention Is All You Need», en *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. [En línea]. Disponible en: https://arxiv.org/abs/1706.03762

[15] K. Cho *et al.*, «Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation», *arXiv preprint arXiv:1406.1078*, 2014, [En línea]. Disponible en: https://arxiv.org/abs/1406.1078

[16] S. Hochreiter y J. Schmidhuber, «Long Short-Term Memory», *Neural Computation*, vol. 9, n.º 8, pp. 1735-1780, 1997, [En línea]. Disponible en: https://dl.acm.org/doi/10.1162/neco.1997.9.8.1735

[17] Google Developers, «Overview of GAN Structure». [En línea]. Disponible en: https://developers.google.com/machine-learning/gan/gan_structure

[18] DataCamp, «An Introduction to Variational Autoencoders (VAE)». [En línea]. Disponible en: https://www.datacamp.com/tutorial/variational-autoencoders

[19] Y. Lipman, R. T. Q. Chen, H. Ben-Hamu, M. Nickel, y M. Le, «Flow Matching for Generative Modeling», en *International Conference on Learning Representations (ICLR)*, 2023. [En línea]. Disponible en: https://arxiv.org/abs/2210.02747

[20] X. Liu, C. Gong, y Q. Liu, «Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow», en *International Conference on Learning Representations (ICLR)*, 2023. [En línea]. Disponible en: https://arxiv.org/abs/2209.03003

[21] Black Forest Labs, «FLUX.1 Kontext: Flow Matching for In-Context Image Generation», *arXiv preprint arXiv:2506.15742*, 2025, [En línea]. Disponible en: https://arxiv.org/abs/2506.15742

[22] TikZ.net, «Normalizing Flow Diagram». [En línea]. Disponible en: https://tikz.net/normalizing-flow/

[23] J. Ho, A. Jain, y P. Abbeel, «Denoising Diffusion Probabilistic Models», en *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. [En línea]. Disponible en: https://arxiv.org/abs/2006.11239

[24] A. Krizhevsky, I. Sutskever, y G. E. Hinton, «ImageNet Classification with Deep Convolutional Neural Networks», en *Advances in Neural Information Processing*

Systems (NeurIPS), 2012. [En línea]. Disponible en: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[25] I. J. Goodfellow *et al.*, «Generative Adversarial Nets», en *Advances in Neural Information Processing Systems (NeurIPS)*, 2014. [En línea]. Disponible en: https://arxiv.org/abs/1406.2661

[26] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, y R. Salakhutdinov, «Dropout: A Simple Way to Prevent Neural Networks from Overfitting», *Journal of Machine Learning Research*, vol. 15, pp. 1929-1958, 2014, [En línea]. Disponible en: https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

[27] D. P. Kingma y J. Ba, «Adam: A Method for Stochastic Optimization», en *International Conference on Learning Representations (ICLR)*, 2015. [En línea]. Disponible en: https://arxiv.org/abs/1412.6980

[28] K. Mishchenko y A. Defazio, «Prodigy: An Expeditiously Adaptive Parameter-Free Learner», *arXiv preprint arXiv:2306.06101*, 2023, [En línea]. Disponible en: https://arxiv.org/abs/2306.06101

[29] J. Chung, C. Gulcehre, K. Cho, y Y. Bengio, «Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling», *arXiv preprint arXiv:1412.3555*, 2014, [En línea]. Disponible en: https://arxiv.org/abs/1412.3555

[30] X. Glorot y Y. Bengio, «Understanding the difficulty of training deep feedforward neural networks», en *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010. [En línea]. Disponible en: https://proceedings.mlr.press/v9/glorot10a

[31] K. He, X. Zhang, S. Ren, y J. Sun, «Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification», en *IEEE International Conference on Computer Vision (ICCV)*, 2015. [En línea]. Disponible en: https://arxiv.org/abs/1502.01852

[32] PyTorch, «FashionMNIST — torchvision documentation». [En línea]. Disponible en: https://docs.pytorch.org/vision/stable/generated/torchvision.datasets.FashionMNIST.html

[33] PyTorch, «CrossEntropyLoss — PyTorch documentation». [En línea]. Disponible en: https://docs.pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

[34] PyTorch, «Adam — PyTorch documentation». [En línea]. Disponible en: https://docs.pytorch.org/docs/stable/generated/torch.optim.Adam.html

[35] A. Ramesh *et al.*, «Zero-Shot Text-to-Image Generation», en *International Conference on Machine Learning (ICML)*, 2021. [En línea]. Disponible en: https://arxiv.org/abs/2102.12092

[36] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, y M. Chen, «Hierarchical Text-Conditional Image Generation with CLIP Latents», *arXiv preprint arXiv:2204.06125*, 2022, [En línea]. Disponible en: https://arxiv.org/abs/2204.06125

[37] J. Betker y others, «Improving Image Generation with Better Captions (DALL-E 3)», technical report, 2023. [En línea]. Disponible en: https://cdn.openai.com/papers/dall-e-3.pdf

[38] P. Isola, J.-Y. Zhu, T. Zhou, y A. A. Efros, «Image-to-Image Translation with Conditional Adversarial Networks», en *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. [En línea]. Disponible en: https://arxiv.org/abs/1611.07004

[39] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, y I. Sutskever, «Robust Speech Recognition via Large-Scale Weak Supervision», en *International Conference on Machine Learning (ICML)*, 2023. [En línea]. Disponible en: https://arxiv.org/abs/2212.04356

[40] W. Peebles y S. Xie, «Scalable Diffusion Models with Transformers», en *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023. [En línea]. Disponible en: https://arxiv.org/abs/2212.09748

[41] E. Perez, F. Strub, H. de Vries, V. Dumoulin, y A. Courville, «FiLM: Visual Reasoning with a General Conditioning Layer», en *AAAI Conference on Artificial Intelligence*, 2018. [En línea]. Disponible en: https://arxiv.org/abs/1709.07871

[42] Z. Liu, P. Luo, X. Wang, y X. Tang, «Large-scale CelebFaces Attributes (CelebA) Dataset». [En línea]. Disponible en: https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

[43] scikit-learn developers, «sklearn.datasets.make_swiss_roll». [En línea]. Disponible en: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_swiss_roll.html

[44] A. Radford, L. Metz, y S. Chintala, «Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks», *arXiv preprint arXiv:1511.06434*, 2015, [En línea]. Disponible en: https://arxiv.org/abs/1511.06434

[45] T. White, «Sampling Generative Networks», *arXiv preprint arXiv:1609.04468*, 2016, [En línea]. Disponible en: https://arxiv.org/abs/1609.04468

[46] I. Goodfellow, Y. Bengio, y A. Courville, *Deep Learning*. MIT Press, 2016. [En línea]. Disponible en: https://www.deeplearningbook.org/

[47] A. Radford, K. Narasimhan, T. Salimans, y I. Sutskever, «Improving Language Understanding by Generative Pre-Training», technical report, 2018. [En línea]. Dispo-

nible en: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[48] J. L. Elman, «Finding Structure in Time», *Cognitive Science*, vol. 14, n.º 2, pp. 179-211, 1990, [En línea]. Disponible en: https://onlinelibrary.wiley.com/doi/10.1207/s15516709cog1402_1

[49] I. Loshchilov y F. Hutter, «Decoupled Weight Decay Regularization», en *International Conference on Learning Representations (ICLR)*, 2019. [En línea]. Disponible en: https://arxiv.org/abs/1711.05101

[50] S. Ioffe y C. Szegedy, «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift», en *International Conference on Machine Learning (ICML)*, 2015. [En línea]. Disponible en: https://arxiv.org/abs/1502.03167

[51] Y. Wu y K. He, «Group Normalization», en *European Conference on Computer Vision (ECCV)*, 2018. [En línea]. Disponible en: https://arxiv.org/abs/1803.08494

[52] J. L. Ba, J. R. Kiros, y G. E. Hinton, «Layer Normalization», *arXiv preprint arXiv:1607.06450*, 2016, [En línea]. Disponible en: https://arxiv.org/abs/1607.06450

[53] D. Bahdanau, K. Cho, y Y. Bengio, «Neural Machine Translation by Jointly Learning to Align and Translate», en *International Conference on Learning Representations (ICLR)*, 2015. [En línea]. Disponible en: https://arxiv.org/abs/1409.0473

[54] D. Hendrycks y K. Gimpel, «Gaussian Error Linear Units (GELUs)», *arXiv preprint arXiv:1606.08415*, 2016, [En línea]. Disponible en: https://arxiv.org/abs/1606.08415

[55] K. He, X. Zhang, S. Ren, y J. Sun, «Deep Residual Learning for Image Recognition», en *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. [En línea]. Disponible en: https://arxiv.org/abs/1512.03385

[56] S. Mohamed y B. Lakshminarayanan, «Learning in Implicit Generative Models», en *arXiv preprint arXiv:1610.03483*, 2016. [En línea]. Disponible en: https://arxiv.org/abs/1610.03483

[57] M. Mirza y S. Osindero, «Conditional Generative Adversarial Nets», en *arXiv preprint arXiv:1411.1784*, 2014. [En línea]. Disponible en: https://arxiv.org/abs/1411.1784

[58] H. Zhang, I. Goodfellow, D. Metaxas, y A. Odena, «Self-Attention Generative Adversarial Networks», en *International Conference on Machine Learning (ICML)*, 2019. [En línea]. Disponible en: https://arxiv.org/abs/1805.08318

[59] T. Karras, T. Aila, S. Laine, y J. Lehtinen, «Progressive Growing of GANs for Improved Quality, Stability, and Variation», en *International Conference on Learning Representations (ICLR)*, 2018. [En línea]. Disponible en: https://arxiv.org/abs/1710.10196

[60] O. Ronneberger, P. Fischer, y T. Brox, «U-Net: Convolutional Networks for Biomedical Image Segmentation», en *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2015. [En línea]. Disponible en: https://arxiv.org/abs/1505.04597

[61] S. Rifai, P. Vincent, X. Muller, X. Glorot, y Y. Bengio, «Contractive Auto-Encoders: Explicit Invariance During Feature Extraction», en *Proceedings of the 28th International Conference on Machine Learning (ICML)*, 2011. [En línea]. Disponible en: https://icml.cc/2011/papers/455_icmlpaper.pdf

[62] P. Vincent, H. Larochelle, Y. Bengio, y P.-A. Manzagol, «Extracting and Composing Robust Features with Denoising Autoencoders», *Proceedings of the 25th International Conference on Machine Learning (ICML)*, 2008, [En línea]. Disponible en: https://www.cs.toronto.edu/~larocheh/publications/icml-2008-denoising-autoencoders.pdf