Índice del capítulo

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].

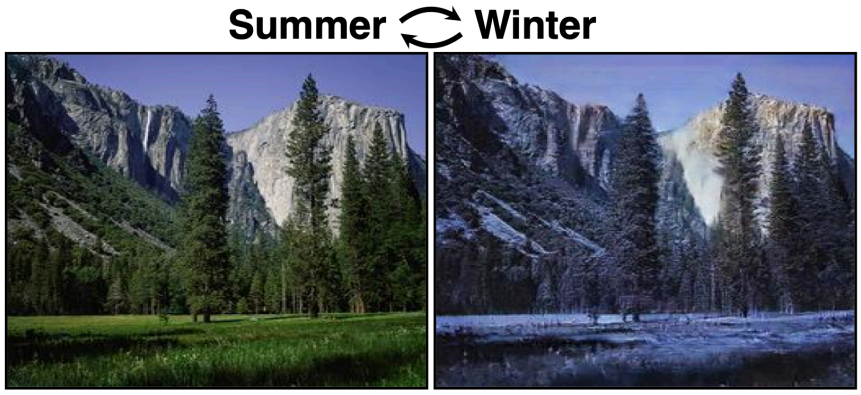

Figura 2: Tarea de transferencia de estilo. Imagen obtenida desde [4].

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

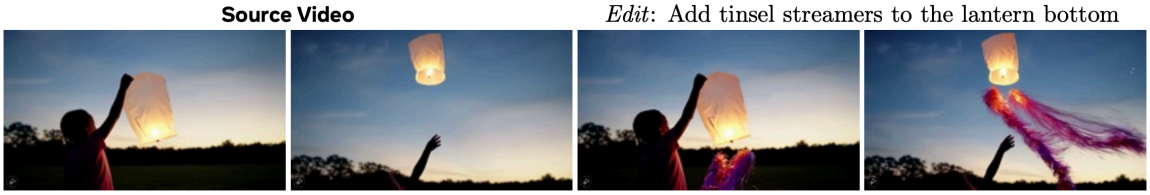

Figura 3: Edición de video mediante un prompt. Imagen obtenida desde [5].

- **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.

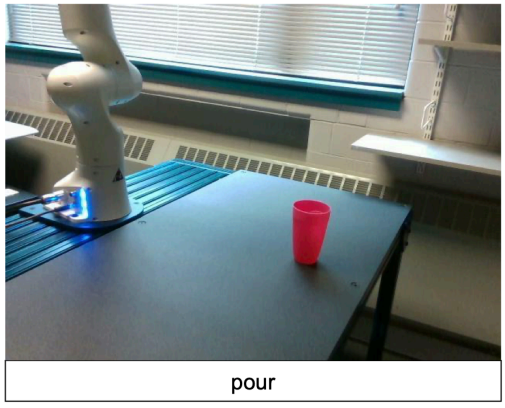

Figura 5: Robot usando un LLM para resolver una tarea. Imagen obtenida desde [8].

- **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.

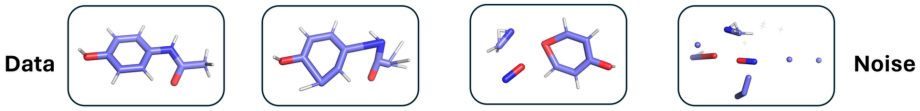

Figura 6: Modelo de difusión utilizado para la generación molecular. Imagen obtenida desde [10].

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

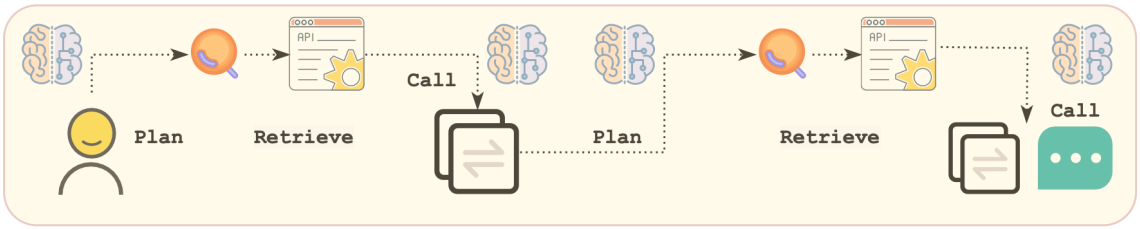

Figura 7: Planificación y uso de herramientas para completar una tarea. Imagen obtenida desde [11].

- **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.

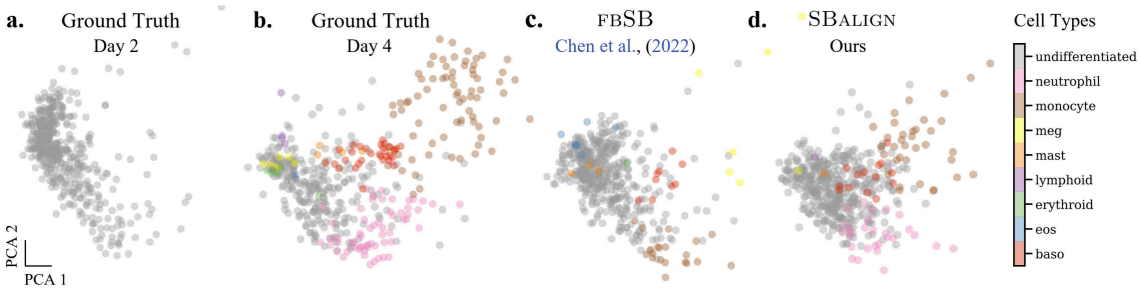

Figura 8: Predicción de diferenciación celular usando puentes de Schrödinger. Imagen obtenida desde [12].

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].

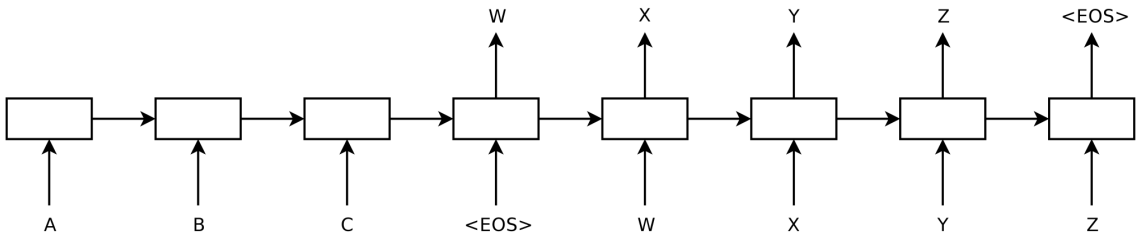

Figura 9: Modelo seq2seq, donde la entrada inicial (prompt) es ABC y la secuencia generada de forma autorregresiva es WXYZ. Imagen obtenida desde [2].

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.

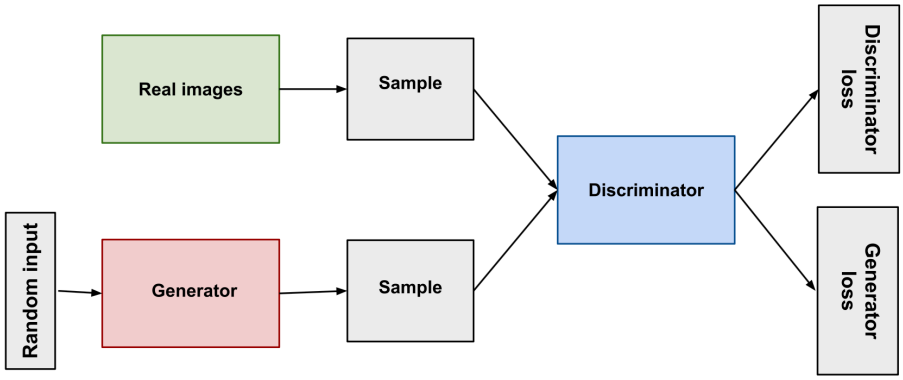

Figura 10: Un modelo discriminativo (clasificador) aprende a diferenciar muestras reales de muestras generadas aleatoriamente. Imagen obtenida desde [17].

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 $z \mapsto x$.

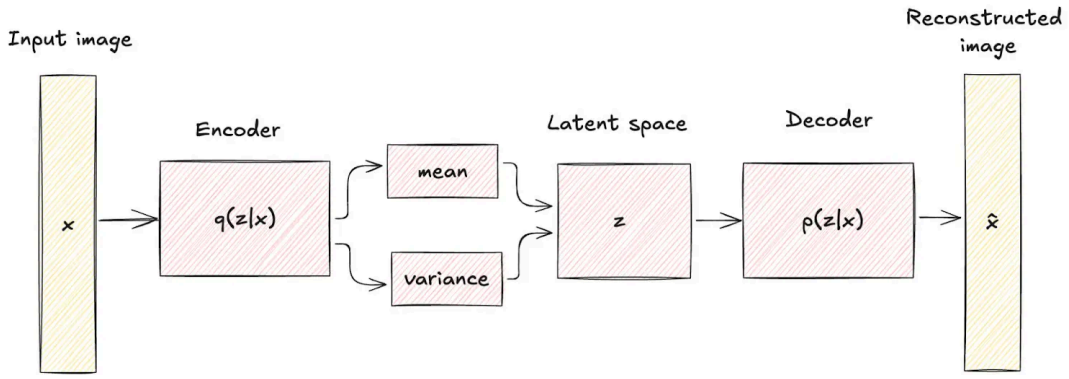

Figura 11: El encoder de un VAE predice la media y varianza de una variable latente z gaussiana. Por otro lado, el decoder aprende a reconstruir la imagen a partir de la variable latente que generó el encoder. Imagen obtenida desde [18].

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]).

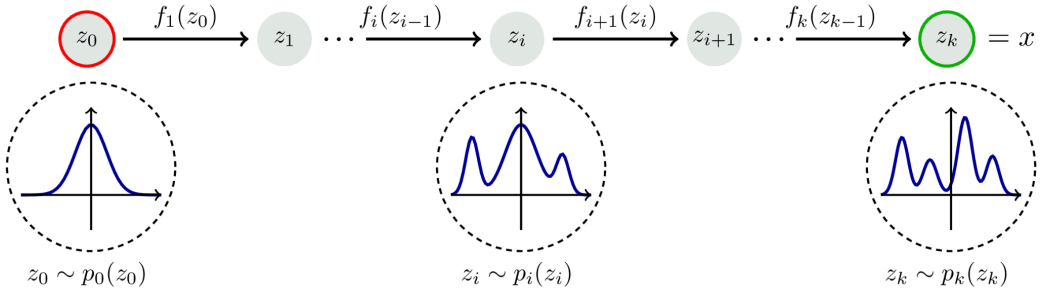

Figura 12: Transformación de un ruido gaussiano inicial en una muestra de la distribución que se busca aprender. Imagen obtenida desde [22].

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.

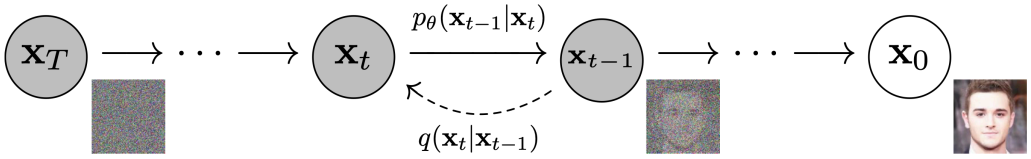

Figura 13: Proceso de denoising de un modelo de difusión. Imagen obtenida desde [23].

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 $r \in \Delta^3$:

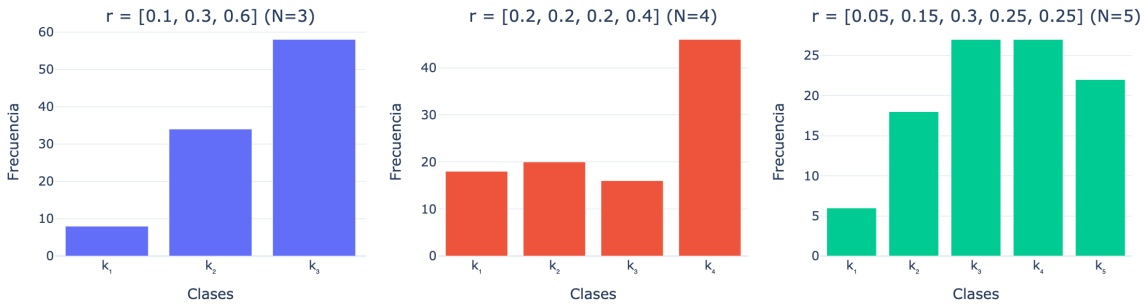

Figura 14: 100 muestras generadas para tres distribuciones categóricas con distintos vectores de probabilidad $r \in \Delta^3$.

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 $[0, 1]$:

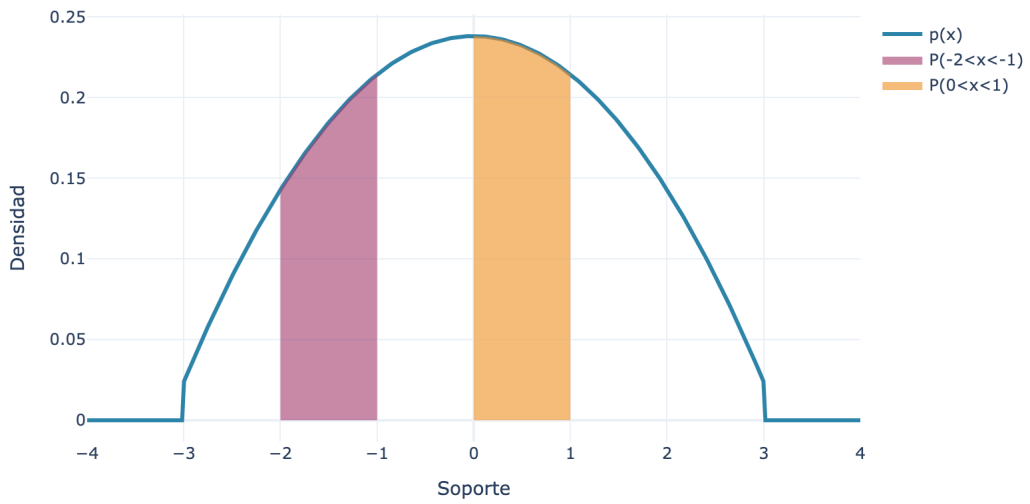

Figura 15: Función de densidad $p(x) = (10 - x^2)/42$ sobre $[-3, 3]$, con las áreas correspondientes a $\mathbb{P}(-2 \leq x \leq -1)$ y $\mathbb{P}(0 \leq x \leq 1)$.

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.

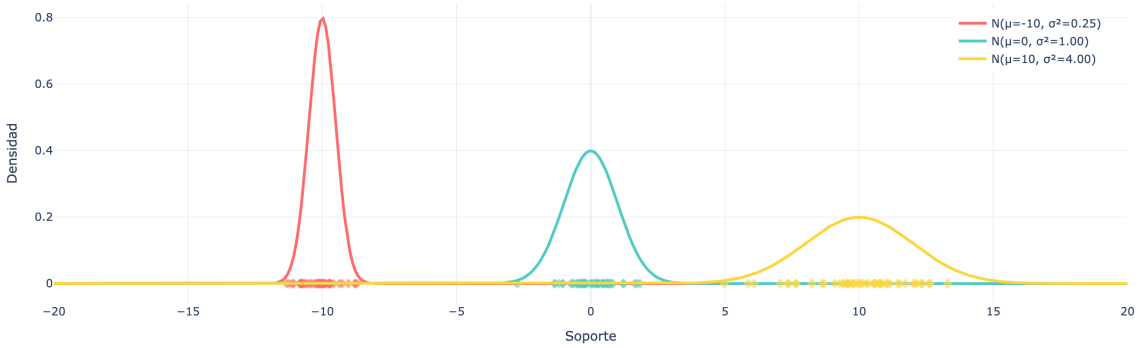

Figura 16: 50 muestras generadas desde 3 distribuciones gaussianas con distintos valores de σ^2.

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.

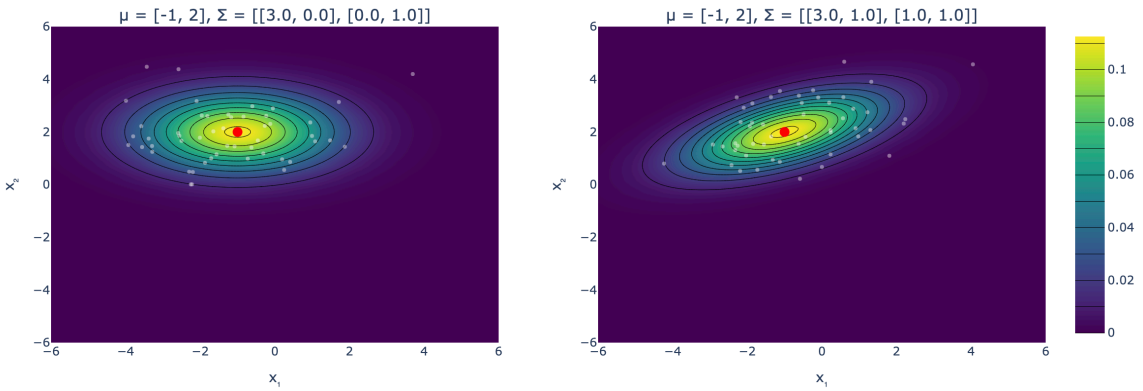

Figura 17: Funciones de densidad y muestras para dos gaussianas bidimensionales: a la izquierda, con matriz de covarianza diagonal; a la derecha, con componentes dependientes.

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:

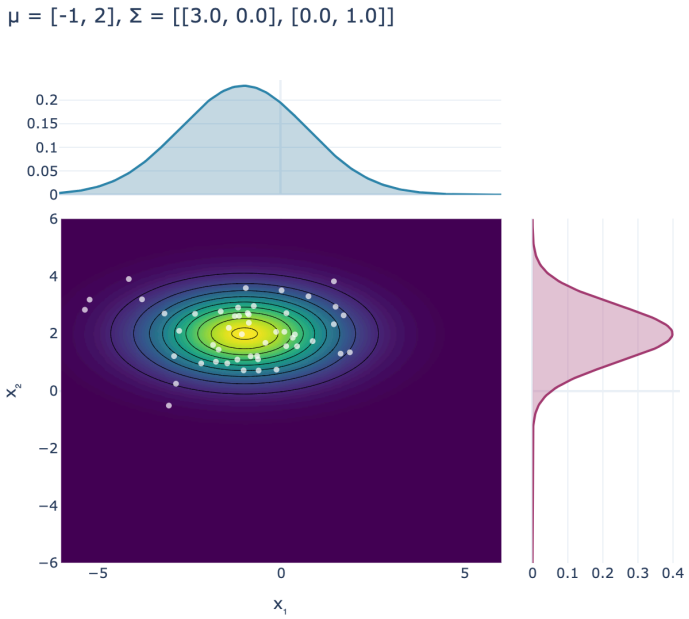

Figura 18: Gaussiana bidimensional con Σ diagonal y sus distribuciones marginales.

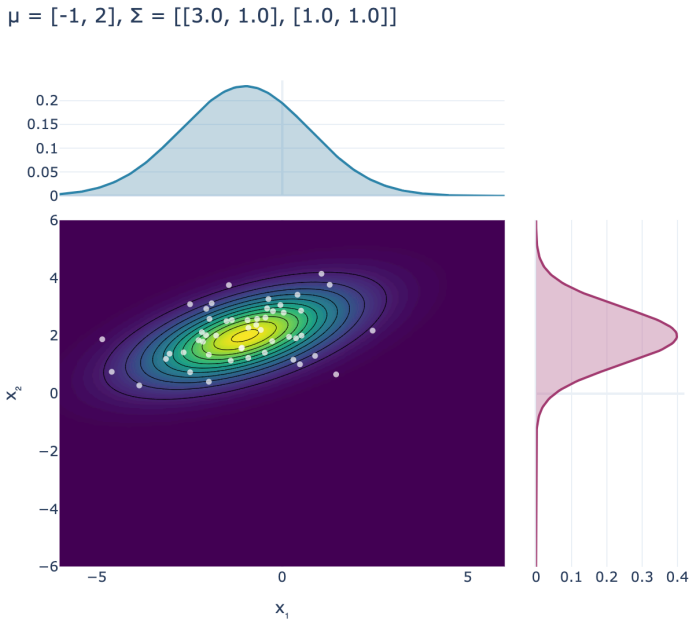

Figura 19: Gaussiana bidimensional con componentes dependientes y sus distribuciones marginales.

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:

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

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

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

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:

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

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, $\mathrm{Cov}(x, x) = \mathbb{V}(x) \geq 0$.

Por otra parte, notar que $\mathrm{Cov}(x, y) = 0$ 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, $\mathrm{Cov}(x, y) = 0$ 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 $\mathrm{Cov}(x) \in \mathcal{M}_{D,D}(\mathbb{R})$ cuyas entradas son $\mathrm{Cov}(x)_{ij} \coloneqq \mathrm{Cov}\big(x_i, x_j\big)$. En particular, la diagonal de esta matriz contiene las varianzas de cada componente de x. Además, notando que $\mathrm{Cov}\big(x_i, x_j\big) = \mathrm{Cov}\big(x_j, x_i\big)$, se tiene que la matriz de covarianzas siempre es simétrica.

Por otro lado, la magnitud de $\mathrm{Cov}(x, y)$ 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:

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

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
```

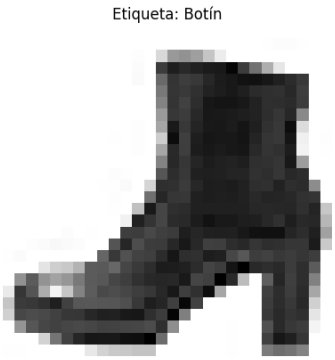

Figura 20: Ejemplo de muestra del dataset Fashion MNIST.

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)
```

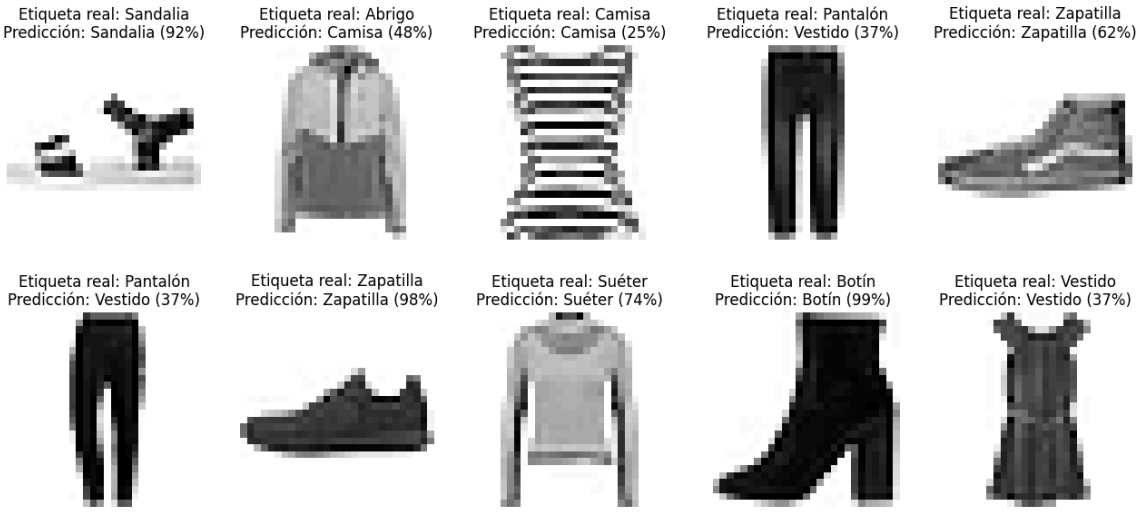

Figura 22: Predicciones realizadas por la red neuronal para 10 imágenes del dataset Fashion MNIST.

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.

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