Índice del capítulo

Capítulo 9

Autoencoders clásicos e inferencia variacional

En este capítulo se revisarán los fundamentos teóricos de una familia importante de modelos generativos llamados autoencoders variacionales. Si bien esta clase de modelos fue propuesta el año 2013, hoy en día estos modelos se siguen utilizando de forma activa como parte de modelos más complejos. Para complementar la formulación teórica, se realizarán dos implementaciones minimales de un autoencoder variacional, y se revisará la idea de interpolación en el espacio latente.

Un **autoencoder variacional** (VAE) es, al igual que una GAN, una red bayesiana de variable latente que busca aprender una distribución de probabilidad desconocida, $p_{\text{data}}(x)$, a partir de un conjunto de muestras i.i.d., $\mathcal{D} = \{x^n\}_{n=1}^N \subset \mathbb{R}^D$, generadas a partir de $p_{\text{data}}(x)$. Como es usual, x representará siempre la variable aleatoria de los datos en $\mathbb{R}^D$ (en el código, D será representado mediante `data_dim`), mientras que z será la variable aleatoria latente en $\mathbb{R}^L$ (en el código, L será representado por `latent_dim`). Además, en línea con la manifold hypothesis, es común considerar $L \ll D$, aunque no es un requisito estricto para el desarrollo teórico y, de hecho, se mencionarán casos donde es útil considerar $D < L$.

Al igual que en otros modelos de variable latente, los términos de la descomposición $p_\theta(x, z) = p_\theta(z)p_\theta(x \mid z)$ son los siguientes:

- $p_\theta(z)$ se interpreta como una distribución a priori sobre la variable latente $z \in \mathbb{R}^L$.
- $p_\theta(x \mid z)$ es la distribución desde la que se generarán nuevas muestras $x \in \mathbb{R}^D$ a partir de un valor dado de la variable latente $z \in \mathbb{R}^L$.

Por lo tanto, una vez el VAE esté entrenado, se podrá generar una nueva muestra desde $p_\theta(x)$ utilizando ancestral sampling, donde se comienza generando una muestra de la variable latente $z \sim p_\theta(z)$ y luego se genera otra desde el modelo condicional $x \sim p_\theta(x \mid z)$.

9.1 Autoencoders clásicos

Antes de comenzar el estudio de los autoencoders variacionales se repasarán algunos conceptos asociados a los autoencoders clásicos (no variacionales), donde la principal diferencia

entre ambos enfoques es que los AEs clásicos son de naturaleza determinista, mientras que los AEs variacionales son de naturaleza probabilística. En particular, los autoencoders clásicos no pueden ser vistos como modelos generativos, lo cual es esperable considerando que este tipo de modelos surgió hace más de 30 años, donde el foco estaba puesto en capacidades de representación compresión, y no de generación.

Un autoencoder es, en principio, una red neuronal utilizada para aprender representaciones eficientes de un conjunto de datos. Estas representaciones son aprendidas de manera autosupervisada mediante la minimización de alguna función objetivo.

Dada una muestra $x \in \mathbb{R}^D$, un autoencoder estándar busca aprender una representación compacta $E_\phi(x) \in \mathbb{R}^L$ (con $E_\phi : \mathbb{R}^D \to \mathbb{R}^L$ una red neuronal llamada **encoder**), de tal modo que otro modelo neuronal $D_\theta : \mathbb{R}^L \to \mathbb{R}^D$ (llamado **decoder**) sea capaz de reconstruir con bastante precisión la muestra original $x \in \mathbb{R}^D$ a partir de esta representación compacta, es decir, $D_\theta\big(E_\phi(x)\big) \approx x$. Para esto, las redes neuronales E_ϕ y D_θ son entrenadas de forma conjunta utilizando cómo función de pérdida alguna métrica de discrepancia, siendo la distancia euclidiana la función de pérdida usual[1]. Más precisamente, si $\mathcal{D} = \{x^n\}_{n=1}^N \subset \mathbb{R}^D$ es el conjunto de entrenamiento, las redes neuronales son optimizadas minimizando

$$\mathcal{L}_{\text{MSE}}(\phi, \theta) = \frac{1}{N} \sum_{n=1}^{N} \left\| x^n - \big(D_\theta \circ E_\phi\big)(x^n) \right\|^2,$$

donde $\big(D_\theta \circ E_\phi\big)(x^n) = D_\theta\big(E_\phi(x^n)\big)$ es la composición del encoder con el decoder aplicado sobre la muestra $x^n \in \mathbb{R}^D$. Esta función de pérdida es natural ya que busca precisamente que la reconstrucción sea lo más cercana posible a la muestra original. Además, notar que si $L \geq D$, entonces la reconstrucción se puede realizar de forma exacta (e.g. se puede considerar $E_\phi(x) = \left(x_1, \cdots, x_D, \underbrace{0 \cdots 0}_{L-D \text{ veces}}\right)^\top \in \mathbb{R}^L$ y luego $D_\theta(z) = (z_1, \cdots, z_D)^\top \in \mathbb{R}^D$), por lo que este problema solo es no trivial cuando $L < D$.

A modo de ejemplo se implementará un autoencoder clásico de forma minimal usando el mismo dataset de juguete 2D usado en la introducción de GANs:

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.datasets import make_swiss_roll

def get_batch(batch_size=1000, noise=0.1):
```

[1]Notar que en este tipo de modelos no tienen sentido conceptos como la verosimilitud debido a la falta de una estructura probabilística sobre el modelo y las muestras.

```
    x, _ = make_swiss_roll(batch_size, noise=noise)
    x = x[:, [0, 2]]
    x = (x - x.mean()) / x.std()
    return torch.tensor(x).float()

# Ejemplo:
samples = get_batch()
plt.figure(figsize=(3, 3))
plt.scatter(samples[:, 0], samples[:, 1], s=1)
plt.show()
```

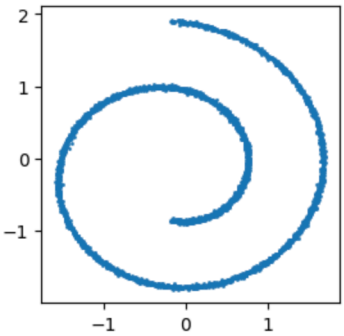

Figura 1: Muestras del dataset de juguete 2D.

Dada la baja complejidad de los datos, será suficiente considerar redes neuronales fully connected. Para el encoder $E_\phi : \mathbb{R}^D \to \mathbb{R}^L$ se utilizará la siguiente red neuronal:

```
class Encoder(nn.Module):

    def __init__(self, data_dim, latent_dim):
        super().__init__()

        self.encoder = nn.Sequential(
            nn.Linear(data_dim, 128), nn.ReLU(),
            nn.Linear(128, 64), nn.ReLU(),
            nn.Linear(64, 32), nn.ReLU(),
            nn.Linear(32, latent_dim)
        )

    def forward(self, x):
        return self.encoder(x)
```

Mientras que para el decoder $D_\theta : \mathbb{R}^L \to \mathbb{R}^D$ se utilizará la siguiente red neuronal:

```
class Decoder(nn.Module):
```

```
    def __init__(self, data_dim, latent_dim):
        super().__init__()

        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 32), nn.ReLU(),
            nn.Linear(32, 64), nn.ReLU(),
            nn.Linear(64, 128), nn.ReLU(),
            nn.Linear(128, data_dim)
        )

    def forward(self, z):
        return self.decoder(z)
```

La siguiente clase `Autoencoder` implementa el loop de entrenamiento y reconstrucción de un AE clásico con error cuadrático medio como función de costo:

```
class Autoencoder:

    def __init__(self, data_dim, latent_dim):

        self.latent_dim = latent_dim

        self.encoder = Encoder(data_dim, latent_dim)
        self.decoder = Decoder(data_dim, latent_dim)
        self.encoder_optimizer = optim.AdamW(self.encoder.parameters())
        self.decoder_optimizer = optim.AdamW(self.decoder.parameters())

    def train(self, iters):

        loss_fn = nn.MSELoss()

        for _ in range(iters):

            # MSE:
            x = get_batch()
            z = self.encoder(x)
            x_dec = self.decoder(z)
            loss = loss_fn(x_dec, x)

            # Optimización:
            self.encoder_optimizer.zero_grad()
            self.decoder_optimizer.zero_grad()
            loss.backward()
            self.encoder_optimizer.step()
```

```
            self.decoder_optimizer.step()

    def reconstruct(self, x):
        z = self.encoder(x)
        x_dec = self.decoder(z)
        return x_dec
```

Entrenando el autoencoder definido anteriormente, se puede observar que la reconstrucción de un autoencoder es casi exacta:

```
# Entrenamiento:
autoencoder = Autoencoder(data_dim=2, latent_dim=16)
autoencoder.train(iters=5000)

# Reconstrucción:
x = get_batch()
reconstruction = autoencoder.reconstruct(x).detach()

plt.figure(figsize=(3, 3))
plt.scatter(x[:, 0], x[:, 1], s=1, label='Original')
plt.scatter(reconstruction[:, 0], reconstruction[:, 1], s=1,
label='Reconstrucción')
plt.legend()
plt.show()
```

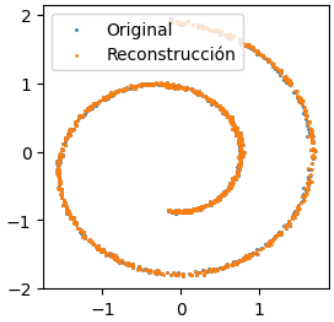

Figura 2: Reconstrucción realizada por un autoencoder clásico.

Que la reconstrucción sea prácticamente exacta es esperable ya que esa es precisamente la función objetivo sobre la que se entrenó el autoencoder. Sin embargo, se verá que esta función de pérdida no es suficiente para darle capacidades generativas a un autoencoder, lo cual se debe, esencialmente, a una falta de noción probabilística sobre la formulación del modelo.

9.1.1 Aplicaciones

Reducción de dimensionalidad

La aplicación natural de un autoencoder es la reducción de dimensionalidad, donde una muestra original (usualmente de alta dimensión) es representada mediante un vector de menor dimensión, lo cual es útil, por ejemplo, para almacenar información usando menos memoria (similar a como ocurre con la compresión JPEG), para implementar sistemas de information retrieval de imágenes (e.g. RAG), o para entrenar otros modelos de machine learning utilizando representaciones compactas de la data en vez de usar los datos originales. Esto último es el uso más común hoy en día para los VAEs, donde un modelo más grande (e.g. un modelos de difusión para imágenes o videos) es entrenado en el espacio latente de un VAE, volviendo mucho más eficiente y escalable el entrenamiento.

Detección de anomalías

Durante el proceso de aprendizaje de un AE, es esperable que el encoder E_ϕ aprenda a omitir la información común entre las instancias de entrenamiento $\mathcal{D} = \{x^n\}_{n=1}^N \subset \mathbb{R}^D$ para solo enfocarse en codificar las características distintivas de cada instancia. Al mismo tiempo, el decoder D_θ debe ser capaz de "memorizar" esta información común para incluirla durante la reconstrucción, y solo variar los detalles propios de la muestra utilizando la representación compacta dada por el encoder.

Debido a este patrón de omitir la información común entre las instancias, es esperable que la codificación y reconstrucción de instancias anómalas sea de menor calidad que la reconstrucción obtenida para las muestras durante el entrenamiento. De esta forma, el error de reconstrucción de un AE puede ser utilizado como un sistema de detección de anomalías mediante el filtrado de instancias que se alejan más de un cierto umbral del error medio de entrenamiento.

Representation learning

Pendiente.

9.1.2 Propiedades de los autoencoders

En esta subsección se revisarán, por completitud, algunas propiedades comúnmente mencionadas sobre los autoencoders clásicos.

Propiedad de Johnson-Lindenstrauss

El siguiente resultado indica que existe una función encoder que preserva, aproximadamente, la distancia euclidiana entre las muestras originales:

Teorema 1 (Johnson-Lindenstrauss). *Dado un conjunto de puntos* $\{x^1, \cdots, x^N\} \subset \mathbb{R}^D$ *y un escalar* $\epsilon \in (0, 1)$*, entonces, para todo* $L \geq \mathcal{O}\left(\frac{\log N}{\epsilon^2}\right)$*, existe un mapa* $f : \mathbb{R}^D \to \mathbb{R}^L$ *tal que*

$$(1-\epsilon)\left\|x^i - x^j\right\|^2 \leq \left\|f(x^i) - f(x^j)\right\|^2 \leq (1+\epsilon)\left\|x_i - x_j\right\|^2,$$

para todo $i, j \in \{1, \cdots, N\}$*. Más aún, esta cota no se puede mejorar. Es decir, existe un conjunto* $\{x^1, \cdots, x^N\} \subset \mathbb{R}^D$ *donde se alcanza la cota.*

Este resultado indica que es posible encontrar una función de codificación tal que la distancia relativa entre las muestras originales no se distorsiona demasiado cuando son transformadas a su representación compacta de menor dimensión. Más aún, se puede probar que el problema de encontrar el mapa f está en la clase de complejidad BPP.

Relación con PCA

Pendiente.

Uso como modelos generativos

Dada la similitud de un autoencoder con un modelo de variable latente, se podría intentar utilizar el decoder como un modelo generativo de forma similar a como ocurre en una GAN. Sin embargo, dado que no hay un prior $p(z)$ para la variable latente, no es obvio qué variable latente elegir para realizar la generación. A modo de ejemplo, se intentará generar un conjunto de muestras a partir de un conjunto variables latentes gaussianas:

```
def generate_samples(autoencoder, n_samples):
    z = torch.randn(n_samples, autoencoder.latent_dim)
    x_dec = autoencoder.decoder(z)
    return x_dec

# Generación:
samples = generate_samples(autoencoder, 5000).detach()
plt.figure(figsize=(5, 5))
plt.scatter(samples[:, 0], samples[:, 1], s=1)
plt.show()
```

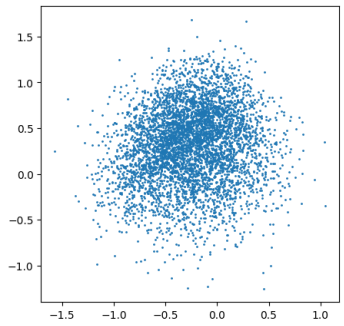

Figura 3: Muestras generadas por un autoencoder clásico.

Se observa que las muestras decodificadas a partir de muestras latentes $z \sim \mathcal{N}(0, \mathrm{I}_L)$ no corresponden a muestras similares a las observadas en el conjunto de entrenamiento `make_swiss_roll`. Esto se debe, principalmente, a que no se definió una distribución prior $p(z)$ durante el entrenamiento, por lo que la elección $z \sim \mathcal{N}(0, \mathrm{I}_L)$ es realmente arbitraria y no justificada (lo mismo ocurriría si se elige, por ejemplo, una distribución uniforme).

Una solución trivial a este problema es considerar como prior a la distribución empírica de las representaciones latentes de las muestras en $\mathcal{D}$, $p_{\mathcal{D}}(z) = \frac{1}{N} \sum_{n=1}^{N} \delta_{E_\phi(x^n)}(z)$. Sin embargo, este prior solo permitirá generar las mismas muestras usadas durante el entrenamiento, por lo que no habría variabilidad. Más aún, este prior no tiene soporte conexo, por lo que las interpolaciones lineales de variables latentes no necesariamente tendrán sentido como sí ocurre, por ejemplo, en una GAN.

La siguiente función muestra que las interpolaciones lineales en el espacio latente de un AE clásico no necesariamente tienen sentido semántico:

```
def latent_interpolation(autoencoder, x0, x1, t):

    z_0 = autoencoder.encoder(x0.unsqueeze(0))
    z_1 = autoencoder.encoder(x1.unsqueeze(0))

    z_t = (1 - t) * z_0 + t * z_1
    x_t = autoencoder.decoder(z_t).squeeze(0)

    return x_t

# Interpolación:
batch_x = get_batch()
x_0, x_1 = batch_x[0], batch_x[1]

plt.figure(figsize=(3, 3))
plt.scatter(batch_x[:, 0], batch_x[:, 1], s=1, alpha=0.05)
```

```
n_steps = 500
for t in torch.linspace(0, 1, n_steps):
    x_t = latent_interpolation(autoencoder, x_0, x_1, t).detach()
    plt.scatter(*x_t, s=1, color='k')

plt.scatter(*x_0, s=100, color='r', label='$x_0$')
plt.scatter(*x_1, s=100, color='g', label='$x_1$')

plt.legend()
plt.show()
```

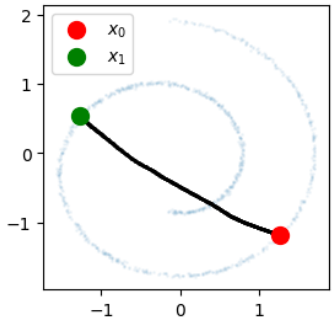

Figura 4: Interpolaciones lineales en el espacio latente de un AE clásico.

Se observa que las interpolaciones en el espacio latente no producen muestras semánticamente significativas debido a la falta de conexidad y convexidad del espacio latente. Por otro lado, si bien se pueden utilizar heurísticas para inducir variabilidad (e.g. perturbar la distribución empírica $p_{\mathcal{D}}(z) = \frac{1}{N}\sum_{n=1}^{N} \delta_{E_\phi(x^n)}(z)$ con kernels gaussianos), esta es una limitación intrínseca de los AEs debido a su naturaleza determinística. En un VAE, en cambio, estos problemas no ocurrirán debido a que el prior será definido desde un comienzo como una distribución gaussiana, la cual tiene soporte denso. Sin embargo, como se verá en la implementación, imponer esta condición generará un trade-off entre la calidad de reconstrucción y la capacidad generativa.

9.1.3 Otros tipos de autoencoders

Si bien la función objetivo $\mathcal{L}_{\text{MSE}}(\phi, \theta)$ es natural, existen algunas variantes que dotan al autoencoder de otras propiedades útiles en el campo de representation learning. En esta subsección se revisarán algunas técnicas clásicas de regularización de autoencoders.

Sparse autoencoder

Como se comentó anteriormente, si el autoencoder es overcomplete ($L > D$), entonces la tarea de reconstrucción es trivial ya que los modelos tienen suficiente capacidad para construir la función identidad en la composición $D_\theta \circ E_\phi : \mathbb{R}^D \to \mathbb{R}^D$. Sin embargo, hay casos donde sí es útil considerar el caso overcomplete.

Un ejemplo usual consiste en el problema de desacoplar características de los datos, donde el objetivo es que cada coordenada $E_\phi(x)_l \in \mathbb{R}$, $l \in \{1, ..., L\}$, represente una característica específica de la muestra $x \in \mathbb{R}^D$ en vez de un conjunto de características mezcladas. Por ejemplo, en el caso de imágenes, una coordenada del encoding $E_\phi(x) \in \mathbb{R}^L$ puede representar únicamente el color de pelo de una persona, mientras que otra coordenada representa únicamente el color de ojos. En el caso no desacoplado, una misma coordenada puede representar varias features al mismo tiempo, limitando la interpretabilidad de la representación latente.

De forma similar, las coordenadas de la codificación dada por el encoder pueden indicar la presencia o ausencia de ciertas features relevantes aprendidas durante el entrenamiento, lo cual puede ser muy útil para tareas de clasificación. En estos casos, y en línea con la *sparse coding hypothesis* formulada en neurociencia, la representación $E_\phi(x) \in \mathbb{R}^L$ suele ser un vector sparse, es decir, un vector donde la mayoría de sus coordenadas son nulas.

Un ***K*-sparse autoencoder** (K-SAE) es un AE con $L > D$ entrenado para que la codificación aprendida tenga a lo más $K \leq L$ coordenadas no nulas. Si bien esto se podría conseguir dejando solo las K dimensiones con mayor valor absoluto y cambiando el resto de coordenadas por 0, una opción más regular es relajar esta condición y agregar un regularizador a la función de costo que penalice la norma de las salidas de encoder. Por ejemplo, se podría considerar:

$$\mathcal{L}_{\text{SAE}}(\phi, \theta) = \mathcal{L}_{\text{MSE}}(\phi, \theta) + \frac{\alpha}{N} \sum_{n=1}^{N} \left\| E_\phi(x^n) \right\|_p^p,$$

donde $\alpha > 0$ es un ponderador que indica el grado de regularización y $p \in \mathbb{N}$ define la métrica de penalización. Si bien esto no garantiza que a lo más K coordenadas sean no nulas, al momento de la inferencia (i.e., con el SAE entrenado) se pueden apagar las $L - K$ coordenadas de menor valor para forzar esta condición.

En el próximo capítulo se revisarán los β-VAEs, los cuales sesgan al modelo a aprender representaciones latentes desacopladas.

Contractive autoencoder

Una técnica de regularización usual en los AE consiste en sesgar al modelo a preferir representaciones $E_\phi(x^1), E_\phi(x^2) \in \mathbb{R}^L$ cercanas (en algún sentido) para muestras $x^1, x^2 \in \mathbb{R}^D$ cercanas entre sí. Dado que la derivada mide la tasa de cambio de una función, acotar superiormente la derivada del encoder permite acotar superiormente su variación entre dos puntos de su dominio. Más precisamente, asumiendo un encoder $E_\phi : \mathbb{R}^D \to \mathbb{R}^L$ lo suficientemente regular, se puede probar la siguiente condición suficiente de lipschitzianidad:

$$\text{si} \quad \sup_{x \in \mathbb{R}^D} \left\| \mathrm{D}_x E_\phi(x) \right\|_F \leq M, \quad \text{entonces} \quad \left\| E_\phi(x^2) - E_\phi(x^1) \right\| \leq M \left\| x^2 - x^1 \right\|,$$

para todo $x^1, x^2 \in \mathbb{R}^D$. Aquí, $\mathrm{D}_x E_\phi(x) \in \mathcal{M}_{L,D}(\mathbb{R})$ es la matriz jacobiana[2] de E_ϕ (evaluada en $x \in \mathbb{R}^D$) y $\left\| \mathrm{D}_x E_\phi(x) \right\|_F^2 = \sum_{i=1}^L \sum_{j=1}^D \left(\mathrm{D}_x E_\phi(x) \right)_{ij}^2$ es su norma Frobenius (al cuadrado). En particular, la propiedad anterior indica que si la cota global $M > 0$ es pequeña, las codificaciones de dos muestras cercanas se mantendrán cercanas en el sentido de la norma euclidiana. Equivalentemente, pequeñas variaciones en una muestra produce pequeñas variaciones en sus representaciones compactas.

Motivado por la propiedad anterior, un **autoencoder contractivo** (CAE, [1]) penaliza la magnitud de las derivadas del encoder para inducir representaciones cercanas para muestras cercanas:

$$\mathcal{L}_{\mathrm{CAE}}(\phi, \theta) = \mathcal{L}_{\mathrm{MSE}}(\phi, \theta) + \frac{\alpha}{N} \sum_{n=1}^N \left\| \mathrm{D}_x E_\phi(x^n) \right\|_F^2$$

Denoising autoencoder

Otra variante típica del autoencoder usual consiste en corromper levemente los datos de entrada $x \in \mathbb{R}^D$ para dificultad aún más su reconstrucción. Para esto, se pueden utilizar distintos enfoques de corrupción:

- **Ruido gaussiano:** a la muestra original se le suma un ruido gaussiano $\epsilon \sim \mathcal{N}(0\mathrm{I}_D)$. Útil para datos de naturaleza continua y no acotada. Este es el tipo de ruido que se utilizará en los modelos de difusión.
- **Masking:** se fijan algunas coordenadas al azar (e.g. el 10%) a 0 con el fin de "ocultar" la información contenida por esa coordenada, de forma análoga al masking realizado en BERT.

[2]Recordar que para un campo vectorial $F : \mathbb{R}^M \to \mathbb{R}^N$, la matriz jacobiana $\mathrm{D}_x F(x) \in \mathcal{M}_{N,M}(\mathbb{R})$ está definida como $\left(\mathrm{D}_x F(x) \right)_{ij} = \frac{\partial F_i}{\partial x_j}(x)$. Equivalentemente, $\mathrm{D}_x F(x) = \begin{pmatrix} \nabla_x F_1(x)^\top \\ \vdots \\ \nabla_x F_N(x)^\top \end{pmatrix}$.

- **Sal y pimienta:** se eligen algunas coordenadas al azar y se fijan sus valores al valor máximo o mínimo (decidido al azar) que puede tomar la coordenada.

Si $T : \mathbb{R}^D \to \mathbb{R}^D$ es la función de corrupción, un **denoising autoencoder** (DAE, [2]) es entrenado de la misma forma que un AE normal, solo que la entrada es pasada por T antes de entrar al autoencoder. Es decir, un DAE es entrenado minimizando la función objetivo

$$\mathcal{L}_{\text{DAE}}(\phi, \theta) = \frac{1}{N} \sum_{n=1}^{N} \left\| x^n - \big(D_\theta \circ E_\phi \circ T\big)(x^n) \right\|^2,$$

donde $\big(D_\theta \circ E_\phi \circ T\big)(x) = D_\theta\big(E_\phi(T(x))\big)$ puede ser visto como la composición de una técnica de data augmentation (input corruption) con un AE clásico. Esta técnica de data augmentation busca mejorar la capacidad de generalización del modelo durante su entrenamiento, por lo que usualmente no se aplica la transformación T cuando se utiliza el modelo ya entrenado. Sin embargo, también es posible utilizar el DAE como un modelo de denoising en sí, donde se cuenta con una imagen corrupta y el DAE permite obtener una reconstrucción limpia de la imagen.

A continuación se implementará un DAE simple sobre el dataset Fashion MNIST:

```
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torchvision.utils import make_grid
from torch.utils.data import DataLoader
from torch.optim import Adam
import matplotlib.pyplot as plt
import random
import tqdm

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

Para mostrar una variación de implementación, se utilizará una única red neuronal que contenga tanto el encoder como el decoder de forma unificada (i.e., la red neuronal modelará directamente $D_\theta \circ E_\phi : \mathbb{R}^D \to \mathbb{R}^D$). Esto es una práctica usual cuando el autoencoder va a ser utilizado de manera completa y no solo una de sus partes (e.g. solo el encoder o solo el decoder).

Para el entrenamiento se considerará a la transformación $T : \mathbb{R}^D \to \mathbb{R}^D$ como la corrupción de sal y pimienta, la cual se puede implementar usando variables aleatorias binarias sobre cada pixel de la imagen, donde el parámetro de cada distribución Bernoulli asociada se elige uniformemente dentro de un rango de valores $[p_{\text{mín}}, p_{\text{máx}}] \subset (0, 1)$:

```
class DAE(nn.Module):

    def __init__(self, img_shape):
        super().__init__()

        self.img_shape = img_shape
        n_features = img_shape[0] * img_shape[1] * img_shape[2]

        self.autoencoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(n_features, 256), nn.ReLU(),
            nn.Linear(256, 128), nn.ReLU(),
            nn.Linear(128, 64), nn.ReLU(),
            nn.Linear(64, 128), nn.ReLU(),
            nn.Linear(128, 256), nn.ReLU(),
            nn.Linear(256, n_features), nn.Sigmoid(),
            nn.Unflatten(1, img_shape)
        )

    def forward(self, x):
        return self.autoencoder(x)

    def train(self, dataloader, optimizer, epochs, p_range):

        self.to(DEVICE)

        try:
            loss_fn = nn.MSELoss()

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

                for x, _ in dataloader:
                    x = x.to(DEVICE)

                    # Inyección de ruido:
                    p = random.uniform(*p_range)
                    B, C, H, W = x.size()
                    noise = torch.empty([B, 1, H, W],
device=DEVICE).bernoulli_(p)
                    noisy_x = x * (1 - noise)

                    # Denoising:
```

```
                    output = self(noisy_x)

                    # Entrenamiento:
                    loss = loss_fn(output, x)
                    optimizer.zero_grad()
                    loss.backward()
                    optimizer.step()

            except KeyboardInterrupt:
                print('Entrenamiento interrumpido.')
```

Se entrenará la red neuronal anterior sobre el dataset FashionMNIST durante 5 épocas:

```
# Datos de entrenamiento:
dataset = datasets.FashionMNIST('data', transform=transforms.ToTensor(),
download=True)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True,
drop_last=True)

# Red neuronal:
img_shape = dataset[0][0].shape
autoencoder = DAE(img_shape)

# Entrenamiento:
optimizer = Adam(autoencoder.parameters())
DAE.train(dataloader, optimizer, epochs=5, p_range=[0.3, 0.7])
```

Con el DAE entrenado, este puede ser utilizado como cualquier otro AE clásico. Como se comentó anteriormente, también es posible utilizarlo como un modelo de denoising (propiedad que no tiene un AE estándar):

```
autoencoder.eval()
autoencoder.cpu()

for _ in range(5):

    # Imagen real:
    idx = random.randint(0, len(dataset))
    true_img, _ = dataset[idx]

    # Imagen ruidosa:
    p = random.uniform(0.3, 0.7)
    noise = torch.empty(true_img.size()[1:]).bernoulli_(p)
    noisy_img = true_img * (1 - noise)
```

```
# Imagen reconstruida:
output = autoencoder(noisy_img.unsqueeze(0))
reconstructed_img = output[0].detach().cpu()

img_grid = make_grid([true_img, noisy_img, reconstructed_img])
plt.figure(figsize=(3, 1))
plt.imshow(1 - img_grid.permute(1, 2, 0))
plt.axis('off')
plt.show()
```

Figura 5: Denoising del DAE entrenado. A la izquierda de cada grupo se muestra la imagen original, en el centro la imagen ruidosa, y a la derecha la imagen reconstruida por el DAE.

Se observa que un DAE es capaz de reconstruir las partes faltantes de la imagen dada como entrada. Esta metodología de entrenamiento (reconstruir imagen original a partir de una versión ruidosa) tiene cierta similitud con la metodología usada para entrenar un modelo de difusión (en particular, con la reparametrización x_0-prediction que se estudiará en el respectivo capítulo). Más aún, un modelo de difusión puede verse como un DAE "mejorado", donde las diferencias son análogas a las diferencias que se observan entre un AE clásico y un AE variacional, aunque los modelos de difusión también incluyen una componente temporal que no se observa en un DAE estándar. De hecho, la función objetivo de un modelo de difusión será la extensión natural de la función objetivo de un VAE estándar.

9.2 Formulación de un VAE

Si bien un autoencoder clásico no tiene una formulación probabilística (en particular, no es una red bayesiana), la parte reconstructiva del decoder $z \mapsto x = D_\theta(z)$ es muy similar

en concepto a la parte generativa $p_\theta(x \mid z)$ de un modelo de variable latente $p_\theta(x, z) = p_\theta(z)p_\theta(x \mid z)$ (aunque en un AE clásico, el prior $p_\theta(z)$ no estaría definido y la parte generadora $p_\theta(x \mid z) \sim \delta_{D_\theta(z)}(z)$ sería determinista, de forma similar a como ocurre en una GAN estándar). Más aún, el encoder $x \mapsto z = E_\phi(x)$ permite tener una cantidad análoga a lo que sería la distribución posterior $p_\theta(z \mid x)$, la cual suele ser una cantidad intratable. Dados estos beneficios y limitaciones de un AE, un autoencoder variacional [3] introduce una formulación probabilística para estas cantidades, permitiendo obtener un nuevo paradigma generativo.

Dado que los VAEs son modelos de variable latente, se debe buscar un enfoque de entrenamiento alternativo a la maximización de la verosimilitud ya que la distribución marginal $p_\theta(x) = \int_{R^L} p_\theta(x, z)\,\mathrm{d}z$ suele ser intratable, por lo que no resulta posible entrenar este tipo de modelos usando el enfoque de máxima verosimilitud. Mientras las GANs evitan este problema entrenando el modelo generativo con un enfoque adversativo, los VAEs utilizan un enfoque basado en inferencia variacional. Para motivar este enfoque, se expresará la cantidad intratable $p_\theta(x)$ de forma conveniente. Descomponiendo la distribución conjunta $p_\theta(x, z)$ en orden contrario al natural (para poder tener la marginal $p_\theta(x)$):

$$p_\theta(x, z) = p_\theta(z \mid x)p_\theta(x) \Longrightarrow p_\theta(x) = \frac{p_\theta(x, z)}{p_\theta(z \mid x)},$$

por lo que podría utilizarse la expresión del lado derecho para el cálculo de la log-verosimilitud. Sin embargo, si bien el numerador es computable directamente (se puede calcular usando la factorización natural del modelo, $p_\theta(x, z) = p_\theta(z)p_\theta(x \mid z)$), el denominador no lo es. En efecto, la distribución posterior $p_\theta(z \mid x)$ debe ser calculada mediante la fórmula de Bayes, la cual requiere conocer la verosimilitud $p_\theta(x)$, que es precisamente lo que se busca calcular.

9.2.1 Inferencia variacional

Del análisis anterior, dado que la distribución posterior $p_\theta(z \mid x)$ no es tratable, no es posible computar eficientemente $p_\theta(x) = \frac{p_\theta(x,z)}{p_\theta(z \mid x)}$ para el entrenamiento por máxima verosimilitud. La solución que proponen los VAEs es estimar la posterior intratable, $p_\theta(z \mid x)$, mediante otro modelo neuronal, $q_\phi(z \mid x)$ (i.e., una red neuronal con parámetros ϕ aprende los parámetros de una nueva distribución $q_\phi(z \mid x)$ que busca parecerse a $p_\theta(z \mid x)$). De esta forma, si $q_\phi(z \mid x) \approx p_\theta(z \mid x)$, entonces se podría estimar la verosimilitud $p_\theta(x)$ mediante $\frac{p_\theta(x,z)}{q_\phi(z \mid x)}$.

Así, un VAE está compuesto por dos modelos probabilísticos, donde los parámetros de cada uno son aprendidos por una red neuronal diferente. Para definir una función objetivo para el entrenamiento de estas redes neuronales, se deben incluir las dos condiciones pedidas:

- $p_\theta(x, z)$ debe aproximar la distribución desconocida, $p_{\text{data}}(x)$, mediante la marginal $p_\theta(x)$. Esto se puede inducir pidiendo una log-verosimilitud alta.
- $q_\phi(z \mid x)$ debe aproximar la posterior desconocida $p_\theta(z \mid x)$. Esto se puede inducir pidiendo una baja divergencia de Kullback-Leibler entre ambas distribuciones.

Combinando ambos objetivos se obtiene la **ELBO** (también conocida como *variational lower bound*), la cual es la función objetivo utilizada en los VAEs, y posteriormente en los modelos de difusión. Esta función objetivo resultará ser tratable, lo que permitirá entrenar este tipo de modelos de forma eficiente. Además, sus distintas descomposiciones permitirán darle distintas interpretaciones y modificaciones, las cuales influirán directamente en los modelos entrenados.

Definición 2 (ELBO). Sea $p_\theta(x, z) = p_\theta(z)p_\theta(x \mid z)$ un modelo de variable latente y $q_\phi(z \mid x)$ otro modelo que busca aproximar $p_\theta(z \mid x)$. Para una muestra $x \in \mathbb{R}^D$, se define

$$\text{ELBO}(x) \coloneqq \log p_\theta(x) - \text{D}_{\text{KL}}\big(q_\phi(z \mid x) \,\|\, p_\theta(z \mid x)\big)$$

Notar que la desigualdad de Gibbs garantiza que $\text{D}_{\text{KL}}\big(q_\phi(z \mid x) \,\|\, p_\theta(z \mid x)\big) \geq 0$, por lo que $\text{ELBO}(x) \leq \log p_\theta(x)$, justificando así el nombre de la ELBO como una cota inferior de la log-verosimilitud (*Evidence Lower BOund*). Por otro lado, notar que la ELBO también depende de los parámetros neuronales θ y ϕ, pero estos se omiten en la notación por simplicidad.

Con esto, la función objetivo que se optimiza al entrenar un VAE es la ELBO esperada sobre la distribución de los datos $p_{\text{data}}(x)$:

$$\max_{\theta,\phi} \mathbb{E}_{p_{\text{data}}(x)}[\text{ELBO}(x)],$$

donde la esperanza es aproximada, como es usual, con una estimación de Monte Carlo utilizando un conjunto de entrenamiento $\mathcal{D} = \{x^1, \cdots, x^N\} \subset \mathbb{R}^D$ generado desde $p_{\text{data}}(x)$:

$$\mathbb{E}_{p_{\text{data}}(x)}[\text{ELBO}(x)] \approx \frac{1}{N} \sum_{n=1}^{N} \text{ELBO}(x^n)$$

Es importante destacar que, si bien la log-verosimilitud $\log p_\theta(x)$ por sí sola no es tratable, cuando se combina con el objetivo de inferencia variacional, $\text{D}_{\text{KL}}\big(q_\phi(z \mid x) \,\|\, p_\theta(z \mid x)\big)$ (el cual tampoco es tratable) se obtiene una función objetivo que sí es tratable, lo que vuelve a la ELBO la función objetivo por defecto para entrenar un VAE. En efecto, recordando que $\texttt{kl}\ (q, p) = \mathbb{E}_q\left[\log \frac{q}{p}\right]$:

$$
\begin{aligned}
\mathrm{ELBO}(x) &= \log p_\theta(x) + \mathbb{E}_{q_\phi(z\,|\,x)}\left[\log\left(\frac{p_\theta(z\,|\,x)}{q_\phi(z\,|\,x)}\right)\right] \\
&= \mathbb{E}_{q_\phi(z\,|\,x)}\left[\log\left(\frac{p_\theta(x,z)}{q_\phi(z\,|\,x)}\right)\right] \\
&= \mathbb{E}_{q_\phi(z\,|\,x)}[\log p_\theta(x\,|\,z)] + \mathbb{E}_{q_\phi(z\,|\,x)}\left[\log\left(\frac{p_\theta(z)}{q_\phi(z\,|\,x)}\right)\right] \\
&= \underbrace{\mathbb{E}_{q_\phi(z\,|\,x)}[\log p_\theta(x\,|\,z)]}_{\text{término de reconstrucción}} - \underbrace{\mathrm{D}_{\mathrm{KL}}\big(q_\phi(z\,|\,x)\,\|\,p_\theta(z)\big)}_{\text{prior matching}}
\end{aligned}
$$

Notar que todos los términos en la última igualdad se pueden computar, lo que permite entrenar ambos modelos de manera conjunta. En particular, la esperanza puede ser aproximada usando muestras generadas desde $q_\phi(z\,|\,x)$, mientras que la divergencia de Kullback-Leibler puede ser calculada en forma cerrada si se eligen modelos $q_\phi(z\,|\,x)$ y $p_\theta(z)$ convenientes.

9.2.2 Modelos paramétricos

El desarrollo hecho hasta el momento es agnóstico a las distribuciones que se elijan para $p_\theta(z)$, $p_\theta(x\,|\,z)$ y $q_\phi(z\,|\,x)$. Estas distribuciones se eligen de manera conveniente para poder evaluar eficientemente la ELBO usando la descomposición anterior, lo cual requiere que se cumplan las siguientes condiciones:

- Debe ser fácil de generar muestras desde $q_\phi(z\,|\,x)$ para poder aproximar la esperanza en el término de reconstrucción.
- $q_\phi(z\,|\,x)$ y $p_\theta(z)$ deben pertenecer a una buena familia de distribuciones (e.g. la familia exponencial) para que el término de prior matching se pueda obtener de forma cerrada.

Para la variable latente incondicional, z, es usual considerar una distribución gaussiana estándar:

$$
p_\theta(z) \sim \mathcal{N}(0, \mathrm{I}_L)
$$

Con esta elección, el término de prior matching en la ELBO se puede obtener de forma cerrada si luego se elige $q_\phi(z\,|\,x)$ también gaussiana. Como es usual, se escribirá $p(z)$ en vez de $p_\theta(z)$ para indicar que la distribución está fija y no posee parámetros entrenables.

Encoder

De acuerdo a la elección del prior $p(z)$, es conveniente elegir la distribución posterior aproximada, $p_\phi(z \mid x)$, también como una distribución gaussiana con el fin de poder calcular el término prior matching de forma cerrada. Más aún, la manifold hypothesis motiva a utilizar una matriz de covarianza diagonal si se asume que las características *esenciales* de una muestra (variables latentes) son elegidas de manera independiente. Por lo tanto, un VAE considera la siguiente distribución para el modelo de inferencia aproximada:

$$q_\phi(z \mid x) \sim \mathcal{N}\Big(\mu_\phi(x), \operatorname{diag}\Big(\sigma^2_\phi(x)\Big)\Big)$$

donde $\mu_\phi : \mathbb{R}^D \to \mathbb{R}^L$ y $\sigma_\phi : \mathbb{R}^D \to \mathbb{R}^L_{++}$ son redes neuronales que aprenden el vector de medias y el vector de varianzas de la distribución gaussiana $q_\phi(z \mid x)$[3].

Es importante notar que la distribución posterior real $p_\theta(z \mid x)$ que busca aproximar este modelo no tiene por qué ser gaussiana (en general no lo es) pero se elige este modelo por conveniencia en la función objetivo. Además, el uso de una matriz de covarianzas diagonal es una práctica usual ya que su determinante es fácil de calcular, el cual será necesario en el cálculo del término prior matching, $\mathrm{D}_{\mathrm{KL}}\big(q_\phi(z \mid x) \,\|\, p_\theta(z)\big)$. Por último, esta elección de $q_\phi(z \mid x)$ permitirá aplicar el *truco de la reparametrización*, el cual resulta ser un requisito esencial para poder entrenar un VAE con algoritmos de gradiente.

Decoder

La distribución $p_\theta(x \mid z)$ dependerá de la naturaleza de la distribución que se busca aprender, $p_{\text{data}}(x)$. Aquí se considerará tanto el caso continuo (usando el dataset de juguete 2D) como el discreto (usado en la generación de imágenes). Notar que esta distribución solo aparece en el término de reconstrucción de la ELBO, por lo la única diferencia entre el entrenamiento de un VAE para datos continuos y un VAE para imágenes estará en la expresión utilizada para este término ya que el término de prior matching será el mismo en ambos casos.

Si las muestras $x \in \mathbb{R}^D$ son vectores cuyas coordenadas pueden tomar cualquier valor en el intervalo $(-\infty, +\infty)$, es usual considerar una distribución gaussiana,

[3]Recordar que para $a \in \mathbb{R}^L$

$$\operatorname{diag}(a) \coloneqq \begin{pmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_L \end{pmatrix}$$

$$p_\theta(x \mid z) \sim \mathcal{N}(\mu_\theta(z), \sigma_0^2 \mathrm{I}_D)$$

donde $\mu_\theta : \mathbb{R}^L \to \mathbb{R}^D$ es una red neuronal que aprende el vector de medias de la distribución $p_\theta(x \mid z)$, mientras que $\sigma_0^2 > 0$ es un parámetro fijo (usualmente pequeño). Si bien es posible aprender también la matriz de covarianza de $p_\theta(x \mid z)$, aquí se está considerando fija e isotrópica ya que simplifica la función de costo y, en consecuencia, la implementación. Por otra parte, es importante indicar que considerar una distribución gaussiana no limita la capacidad de generación del modelo ya que la verdadera complejidad viene codificada en el vector de medias, $\mu_\theta(z) \in \mathbb{R}^D$, el cual es aprendido por la red neuronal $\mu_\theta : \mathbb{R}^L \to \mathbb{R}^D$.

Por otro lado, cuando las muestras $x \in \mathbb{R}^D$ son imágenes (con $D = \text{alto} \cdot \text{ancho}$, y asumiendo un único canal de color por simplicidad), es usual modelar cada pixel $x_d \in [0, 1]$, $d \in \{1, ..., D\}$ como una distribución Bernoulli. En particular, dado el valor de la variable latente $z \in \mathbb{R}^L$, cada pixel se considera independiente del resto de pixeles. Es decir, se utiliza la siguiente distribución para el decoder cuando se trabaja con imágenes:

$$p_\theta(x \mid z) = \prod_{d=1}^{D} p_\theta(x_d \mid z), \quad \text{donde} \quad p_\theta(x_d \mid z) \sim \text{Bernoulli}(r_\theta(z)_d)$$

Aquí, $r_\theta : \mathbb{R}^L \to [0, 1]^D$ es una red neuronal que aprende los parámetros $r_\theta(z)_d \in [0, 1]$ de la distribución Bernoulli de cada pixel $x_d \in [0, 1]$ de la imagen.

9.2.3 ELBO en un VAE

Teniendo definido cada uno de los modelos paramétricos usados en un VAE, es posible desarrollar los términos de reconstrucción y prior matching de la ELBO para obtener las expresiones que se utilizan en las implementaciones. Se partirá desarrollando el término de prior matching y luego el término de reconstrucción.

Prior matching

Dado que las distribuciones $q_\phi(z \mid x)$ y $p_\theta(z)$ son ambas gaussianas, el término de prior matching, $\mathrm{D}_{\mathrm{KL}}\big(q_\phi(z \mid x) \,\|\, p_\theta(z)\big)$, puede ser calculado en forma cerrada. Para eso, se utilizará el siguiente resultado clásico:

Teorema 3. *Dadas dos distribuciones gaussianas en $\mathbb{R}^L$, su divergencia de Kullback-Leibler tiene forma cerrada[4]:*

[4]Para $A \in \mathcal{M}_{m,n}(\mathbb{R})$, $\det(A) \in \mathbb{R}$ es el determinante de A y corresponde al producto de sus valores propios. Del mismo modo, $\mathrm{tr}(A) \in \mathbb{R}$ es la traza de A y corresponde a la suma de sus valores propios, lo cual resulta ser equivalente a la suma de los elementos de su diagonal.

$$
\begin{aligned}
&\mathrm{D}_{\mathrm{KL}}(\mathcal{N}(\mu_1, \Sigma_1) \,\|\, \mathcal{N}(\mu_2, \Sigma_2)) \\
&= \frac{1}{2}\Big[(\mu_1 - \mu_2)^\top \Sigma_2^{-1}(\mu_1 - \mu_2) + \operatorname{tr}(\Sigma_1 \Sigma_2^{-1}) - L - \log(|\det(\Sigma_1 \Sigma_2^{-1})|)\Big]
\end{aligned}
$$

Aplicando este resultado a la posterior aproximada $q_\phi(z \,|\, x) \sim \mathcal{N}\Big(\mu_\phi(x), \operatorname{diag}\Big(\sigma_\phi^2(x)\Big)\Big)$ y al prior latente $p_\theta(z) \sim \mathcal{N}(0, \mathrm{I}_L)$ se obtiene una expresión cerrada para el término de prior matching:

$$
\begin{aligned}
\mathrm{D}_{\mathrm{KL}}\big(q_\phi(z \,|\, x) \,\|\, p_\theta(z)\big) &= \frac{1}{2}\left[\mu_\phi(x)^\top \mu_\phi(x) + \sum_{l=1}^{L} \sigma_\phi^2(x) - L - \log\left(\prod_{l=1}^{L} \sigma_\phi^2(x)_l\right)\right] \\
&= \frac{1}{2}\Big(\big\|\mu_\phi(x)\big\|^2 + \big\|\sigma_\phi(x)\big\|^2\Big) - \sum_{l=1}^{L} \log \sigma_\phi(x)_l + \text{constante}
\end{aligned}
$$

En la primera igualdad se usó que el determinante de una matriz diagonal es el producto de su diagonal, mientras que en la segunda igualdad se identificaron dos formas equivalentes de la norma de un vector[5].

Término de reconstrucción

Antes de desarrollar el término de reconstrucción $\mathbb{E}_{q_\phi(z \,|\, x)}[\log p_\theta(x \,|\, z)]$ sustituyendo el respectivo modelo $p_\theta(x \,|\, z)$ en la esperanza, se reescribirá la distribución $q_\phi(z \,|\, x)$ de forma conveniente para que la variable aleatoria $q_\phi(z \,|\, x)$ sobre la que se calcula la esperanza $\mathbb{E}_{q_\phi(z \,|\, x)}[\log p_\theta(x \,|\, z)]$ no dependa de ϕ. Esto permitirá, como se verá en el próximo capítulo, no tener problemas al momento de entrenar la red neuronal.

Considerando que $q_\phi(z \,|\, x) \sim \mathcal{N}\Big(\mu_\phi(x), \operatorname{diag}\Big(\sigma_\phi^2(x)\Big)\Big)$, esta variable aleatoria se puede reparametrizar esta usando el cambio de variable $z = \mu_\phi(x) + \sigma_\phi \odot \epsilon$, donde $\epsilon \sim \mathcal{N}(0, \mathrm{I}_L)$ es una nueva variable aleatoria independiente de ϕ y $\odot$ es el producto de Hadamard. Con esta sustitución, el término de reconstrucción se puede escribir como

$$
\mathbb{E}_{q_\phi(z \,|\, x)}[\log p_\theta(x \,|\, z)] = \mathbb{E}_{\epsilon \sim \mathcal{N}(0, \mathrm{I}_L)}\big[\log p_\theta\big(x \,\big|\, \mu_\phi(x) + \sigma_\phi \odot \epsilon\big)\big]
$$

Además, como es usual, la esperanza anterior se puede estimar utilizando una aproximación de Monte Carlo:

$$
\mathbb{E}_{q_\phi(z \,|\, x)}[\log p_\theta(x \,|\, z)] \approx \frac{1}{K} \sum_{k=1}^{K} \log p_\theta\big(x \,\big|\, \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon_k\big),
$$

[5]Recordar que, para $z \in \mathbb{R}^L$, $\|z\|^2 = \sum_{l=1}^{L} z_l^2 = \langle z, z \rangle = z^\top z$.

donde $\epsilon_k \sim \mathcal{N}(0, \mathrm{I}_L)$ para $k \in \{1, ..., K\}$. Usualmente es suficiente considerar $K = 1$, aproximando el término de reconstrucción con una única muestra de la variable latente $z \sim q_\phi(z \mid x)$.

Para concluir con la formulación de un VAE y pasar a la implementación, solo falta desarrollar el término $\log p_\theta(x \mid z)$ dentro de la esperanza, el cual dependerá si se está trabajando con datos continuos o con imágenes.

Como se mencionó anteriormente, si $p_{\text{data}}(x)$ es una distribución continua, se suele considerar un decoder con distribución $p_\theta(x \mid z) \sim \mathcal{N}(\mu_\theta(z), \sigma_0^2 \mathrm{I}_D)$, con $\sigma_0^2 > 0$ un hiperparámetro, luego:

$$p_\theta(x \mid z) = \frac{1}{\sqrt{(2\pi\sigma_0^2)^D}} \exp\left(-\frac{1}{2\sigma_0^2}\|x - \mu_\theta(z)\|^2\right).$$

Por lo tanto, el término de reconstrucción se reduce, salvo constante aditiva, a una diferencia de cuadrados, siendo similar a la función objetivo que se utiliza en un autoencoder clásico:

$$\mathbb{E}_{q_\phi(z \mid x)}[\log p_\theta(x \mid z)] = -\frac{1}{2\sigma_0^2}\mathbb{E}_{q_\phi(z \mid x)}\left[\|x - \mu_\theta(z)\|^2\right] + \text{constante}$$

Notar que si la varianza $\sigma_0^2 > 0$ también fuera un parámetro entrenable (i.e., $\sigma_0 = \sigma_\theta(z)$), el término $-\frac{D}{2}\log\big(2\pi\sigma_\theta^2(z)\big)$ en $\log p_\theta(x \mid z)$ ya no sería constante, por lo que se debería incluir en la implementación. En este caso, quedó como un hiperparámetro que balancea la importancia del término de reconstrucción con respecto al término de prior matching.

Por otro lado, si $p_{\text{data}}(x)$ es una distribución definida sobre imágenes, cada pixel $x_d \in [0, 1]$ (condicionado a z) se suele modelar como una distribución Bernoulli con parámetro $r_d = r_\theta(z)_d$, por lo que la función de masa para dicho pixel es $p_\theta(x_d \mid z) = r_d^{x_d} \cdot (1 - r_d)^{1 - x_d}$. Luego:

$$\mathbb{E}_{q_\phi(z \mid x)}[\log p_\theta(x \mid z)] = \mathbb{E}_{q_\phi(z \mid x)}\left[\sum_{d=1}^{D}(x_d \cdot \log r_d + (1 - x_d) \cdot \log(1 - r_d))\right]$$

En particular, esta función objetivo resulta ser equivalente a la entropía cruzada binaria (salvo factor de ponderación negativo).

9.3 Implementación de un VAE

En esta sección se implementarán los dos tipos de VAEs vistos.

9.3.1 VAE gaussiano

Como ejemplo de data continua se utilizará el mismo dataset de juguete 2D usado anteriormente al implementar un AE clásico.

Redes neuronales

Dado que el prior $p(z) \sim \mathcal{N}(0, \mathrm{I}_L)$ está fijo, solo es necesario entrenar redes neuronales que aprendan los parámetros del encoder $q_\phi(z \mid x)$ y del decoder $p_\theta(x \mid z)$. Considerando que se está trabajando con un dataset simple, es suficiente utilizar redes neuronales fully connected de pocas capas. Además, como es usual, el encoder tendrá una cantidad descendente de neuronas, mientras que el decoder tendrá una cantidad ascendente, replicando el cuello de botella de un autoencoder estándar.

Para el encoder $q_\phi(z \mid x) \sim \mathcal{N}\Big(\mu_\phi(x), \mathrm{diag}\Big(\sigma^2_\phi(x)\Big)\Big)$ es necesario aprender un vector de medias $\mu_\phi(x) \in \mathbb{R}^L$ y un vector de desviacions estándar $\sigma_\phi(x) \in \mathbb{R}^L_{++}$. Sin embargo, para evitar la restricción de positividad de $\sigma_\phi(x)$, en la práctica se suele aprender el vector irrestricto $\log \sigma_\phi(x) = \big(\log \sigma_\phi(x)_1, \cdots, \log \sigma_\phi(x)_L\big) \in \mathbb{R}^L$ en vez de aprender directamente $\sigma_\phi(x) \in \mathbb{R}^L_{++}$. Por otra parte, se utilizará una única red neuronal $\mathrm{Encoder}_\phi : \mathbb{R}^D \to \mathbb{R}^L \times \mathbb{R}^L$ que aprenda los dos parámetros del encoder, $\big(\mu_\phi(x), \sigma_\phi(x)\big) \in \mathbb{R}^L \times \mathbb{R}^L_{++}$, al mismo tiempo, lo cual es una práctica usual en la implementación de VAEs.

```python
class Encoder(nn.Module):

    def __init__(self, data_dim, latent_dim):
        super().__init__()

        self.mlp = nn.Sequential(
            nn.Linear(data_dim, 128), nn.ReLU(),
            nn.Linear(128, 64), nn.ReLU(),
            nn.Linear(64, 32), nn.ReLU(),
        )
        self.mean = nn.Linear(32, latent_dim)
        self.log_std = nn.Linear(32, latent_dim)

    def forward(self, x):
        x = self.mlp(x)
        mean = self.mean(x)
        logstd = self.log_std(x)
        return mean, logstd.exp()
```

Con respecto a la red neuronal asociada al decoder $p_\theta(x \mid z)$, esta dependerá de la naturaleza de $x \in \mathbb{R}^D$. Dado que en este caso se están considerando datos continuos, $p_\theta(x \mid$

$z) \sim \mathcal{N}(\mu_\theta(z), \sigma_0^2 I)$, por lo que solo es necesario aprender el vector de medias $\mu_\theta(z) \in \mathbb{R}^D$. Además, dado que este vector buscado no tiene restricciones, no hace falta agregar ninguna una función de activación en la salida de la red neuronal.

```
class Decoder(nn.Module):

    def __init__(self, data_dim, latent_dim):
        super().__init__()

        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 32), nn.ReLU(),
            nn.Linear(32, 64), nn.ReLU(),
            nn.Linear(64, 128), nn.ReLU(),
            nn.Linear(128, data_dim)
        )

    def forward(self, z):
        mean = self.decoder(z)
        return mean
```

Clase para el VAE

Teniendo las redes neuronales para el encoder y el decoder, se implementará una clase `VAE` que contenga los métodos para el entrenamiento de las redes neuronales y para la generación de nuevas muestras. En este caso, $D = 2$ y se considerará una dimensión latente $L = 16$. Además, por elección empírica se considerará $\sigma_0 = 0.1$.

```
class VAE:

    def __init__(self, data_dim, latent_dim):

        self.latent_dim = latent_dim

        self.encoder = Encoder(data_dim, latent_dim)
        self.decoder = Decoder(data_dim, latent_dim)
        self.encoder_optimizer = optim.AdamW(self.encoder.parameters())
        self.decoder_optimizer = optim.AdamW(self.decoder.parameters())

        self.decoder_std = 0.1

    def train(self, iters):

        for _ in range(iters):
```

```
            x = get_batch()

            # Prior matching:
            encoder_mean, encoder_std = self.encoder(x)
            prior_matching = 1/2 * (encoder_mean.norm(dim=-1) ** 2 +
encoder_std.norm(dim=-1) ** 2) - encoder_std.log().sum(dim=-1)

            # Reconstruction term:
            z = encoder_mean + encoder_std * torch.randn_like(encoder_mean)
            decoder_mean = self.decoder(z)
            reconstruction_term = - 1 / (2 * self.decoder_std ** 2) * (x -
decoder_mean).norm(dim=-1) ** 2

            # ELBO:
            elbo = reconstruction_term - prior_matching
            loss = - elbo.mean()

            # Optimización:
            self.encoder_optimizer.zero_grad()
            self.decoder_optimizer.zero_grad()
            loss.backward()
            self.encoder_optimizer.step()
            self.decoder_optimizer.step()

    def generate_samples(self, n_samples):
        z = torch.randn(n_samples, self.latent_dim)
        x_mean = self.decoder(z)
        x_dec = x_mean# + self.decoder_std * torch.randn_like(x_mean)
        return x_dec
```

Entrenamiento y generación

Con la clase anterior definida, se puede entrenar un VAE sobre el dataser de juguete:

```
vae = VAE()
vae.train(iters=5000)
```

Una vez entrenado el modelo se pueden generar nuevas muestras usando `generate_samples`. Se observa que, a diferencia de un AE clásico, el VAE implementado es capaz de generar nuevas muestras luego del entrenamiento. Notar también que las muestras generadas son más dispersas y no se concentran alrededor de la distribución original de los datos. En el caso de imágenes, esta dispersión en la distribución aprendida se refleja en las imágenes generadas, las cuales suelen ser más borrosas que las imágenes generadas por una GAN.

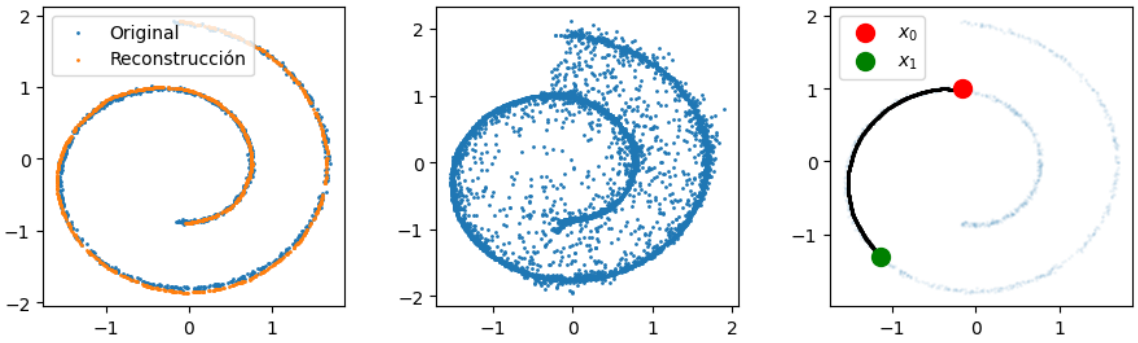

Figura 6: Reconstrucción de muestras del dataset (izquierda), muestras generadas (centro) e interpolación latente (derecha) usando un VAE entrenado sobre el dataset de juguete 2D.

Por otro lado, es posible evaluar la capacidad de reconstrucción de un VAE, la cual resulta ser siempre de menor calidad que en un AE clásico (donde el objetivo es precisamente reconstruir). Esto se debe principalmente a la presencia del término prior matching en la ELBO, el cual obliga al modelo a ceder un poco en la calidad de la reconstrucción a cambio de un espacio latente mejor estructurado (producto del regularizador de prior matching).

En este caso, la reconstrucción se puede implementar con el siguiente método adicional sobre la clase `VAE`:

```
def reconstruct(self, x):
    z_mean, z_std = self.encoder(x)
    z = z_mean# + z_std * torch.randn_like(z_mean)
    x_mean = self.decoder(z)
    x_dec = x_mean# + self.decoder_std * torch.randn_like(x_mean)
    return x_dec
```

Por otro lado, dada la estructura gaussiana inducida sobre el espacio latente, la interpolación latente en un VAE, a diferencia de un AE clásico, sí resulta en una interpolación con sentido semántico. En este caso, la interpolación latente se puede realizar con el siguiente método adicional sobre la clase `VAE`:

```
def latent_interpolation(self, x0, x1, t):

        # Latentes:
        z_0, _ = self.encoder(x0.unsqueeze(0))
        z_1, _ = self.encoder(x1.unsqueeze(0))

        # Interpolación:
        z_t = (1 - t) * z_0 + t * z_1
        x_t = self.decoder(z_t).squeeze(0)
```

```
        return x_t
```

9.3.2 VAE para imágenes

Ahora se implementará un VAE para trabajar con imágenes. Notar que, en este caso, el cálculo de la ELBO cambia ya que el término de reconstrucción $\mathbb{E}_{q_\phi(z\,|\,x)}[\log p_\theta(x\,|\,z)]$ ahora considera una distribución $p_\theta(x\,|\,z)$ discreta. Además, para agregar más flexibilidad, se implementará un VAE condicional[6] para poder indicar la clase a la que debe pertenecer la imagen que se generará.

Las bibliotecas adicionales que se utilizarán son las siguientes:

```
import torch.nn.functional as F
from torch.utils.data import Data**Loader
from torchvision import datasets, transforms, utils
import tqdm

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

Como datos de entrenamiento, se utilizará el dataset Fashion MNIST escalado a 32×32 para poder usar redes convolucionales más profundas en el encoder y decoder.

```
transf = transforms.Compose([transforms.Resize((32, 32)),
transforms.ToTensor()])
dataset = datasets.FashionMNIST('data', train=True, transform=transf,
download=True)
dataloader = DataLoader(dataset, batch_size=256, shuffle=True,
drop_last=True)

def show_batch(images):
    grid_tensor = utils.make_grid(images, nrow=10)
    plt.axis('off')
    plt.imshow(1-grid_tensor.permute(1, 2, 0))
    plt.tight_layout()
    plt.show()

# Ejemplo:
batch_x, batch_y = next(iter(dataloader))
show_batch(batch_x[:50])
```

[6]Recordar que toda red bayesiana $p(x_1, ..., x_N) = \prod_{n=1}^{N} p(x_n\,|\,\mathrm{Pa}(x_n))$ puede ser extendida a su forma condicional, $p(x_1, ..., x_N\,|\,y) = \prod_{n=1}^{N} p(x_n\,|\,\mathrm{Pa}(x_n), y)$, extendiendo las redes neuronales que aprenden los parámetros de $p(x_n\,|\,\mathrm{Pa}(x_n))$ para que ahora también reciban la condición y.

Figura 7: Muestras del dataset Fashion MNIST.

Redes neuronales

El encoder consistirá en una red convolucional estándar, la cual pasará por una transformación lineal en la última capa para obtener los parámetros de media y varianza de $q_\phi(z \mid x)$. Por otro lado, la condición de clase será codificada como un vector one-hot, el cual será expandido y concatenado en la dimensión de los canales de la imagen (i.e., cada pixel aumentará su cantidad de canales en 10, donde 10 es la cantidad de clases distintas del dataset MNIST):

```
class ImageEncoder(nn.Module):

    def __init__(self, image_size, latent_dim, n_classes):
        super().__init__()

        self.n_classes = n_classes

        conv = lambda in_ch, out_ch: nn.Conv2d(in_ch, out_ch,
kernel_size=3, stride=2, padding=1)
        c, h, w = image_size
        flatten_dim = 64 * (h // 8) * (w // 8)

        self.encoder = nn.Sequential(
            conv(c + n_classes, 16), nn.ReLU(),
            conv(16, 32), nn.ReLU(),
            conv(32, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(flatten_dim, 2 * latent_dim)
        )

    def forward(self, x, y):
```

```
        batch_size, c, h, w = x.shape
        y_emb = F.one_hot(y, self.n_classes)  # [batch_size, n_classes].
        y_emb = y_emb[:, :, None, None].expand(batch_size, self.n_classes,
h, w)  # [batch_size, n_classes, h, w].

        x_cond = torch.cat([x, y_emb], dim=1)  # [batch_size, c +
n_classes, h, w].

        mean, logstd = self.encoder(x_cond).chunk(2, dim=-1)
        std = logstd.exp()
        return mean, std
```

Para el decoder, se utilizará una red convolucional simétrica a la usada en el encoder:

```
class ImageDecoder(nn.Module):

    def __init__(self, image_size, latent_dim, n_classes):
        super().__init__()

        c, h, w = image_size
        flatten_dim = 64 * (h // 8) * (w // 8)
        self.n_classes = n_classes

        deconv = lambda in_ch, out_ch: nn.ConvTranspose2d(in_ch, out_ch, 3,
2, 1, 1)

        self.decoder = nn.Sequential(
            nn.Linear(latent_dim + n_classes, flatten_dim),
            nn.ReLU(),
            nn.Unflatten(1, (64, h // 8, w // 8)),
            deconv(64, 32), nn.ReLU(),
            deconv(32, 16), nn.ReLU(),
            deconv(16, c), nn.Sigmoid()
        )

    def forward(self, z, y):
        y_onehot = F.one_hot(y, self.n_classes)  # [batch_size, n_classes].
        z_cond = torch.cat([z, y_onehot], dim=-1)  # [batch_size,
latent_dim + n_classes].
        x_hat = self.decoder(z_cond)
        return x_hat
```

Clase `ImageVAE` para imágenes

Ahora, se implementará una clase `ImageVAE` análoga a la anterior. El cálculo de la función de pérdida se hará en el método `_calc_loss` para no hacer tan extenso el método `train`.

```
class ImageVAE:

    def __init__(self, image_size=(1, 32, 32), latent_dim=128,
n_classes=10):

        self.image_size = image_size
        self.latent_dim = latent_dim

        self.encoder = ImageEncoder(image_size, latent_dim, n_classes)
        self.decoder = ImageDecoder(image_size, latent_dim, n_classes)

        self.encoder_optimizer = optim.AdamW(self.encoder.parameters())
        self.decoder_optimizer = optim.AdamW(self.decoder.parameters())

        self.encoder.to(DEVICE)
        self.decoder.to(DEVICE)

    def train(self, dataloader, epochs):

        self.encoder.train()
        self.decoder.train()

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

                for x, y in dataloader:
                    x, y = x.to(DEVICE), y.to(DEVICE)
                    loss = self._calc_loss(x, y)

                    # Optimización:
                    self.encoder_optimizer.zero_grad()
                    self.decoder_optimizer.zero_grad()
                    loss.backward()
                    self.encoder_optimizer.step()
                    self.decoder_optimizer.step()

        except KeyboardInterrupt:
            print('Entrenamiento interrumpido.')
```

```
    def _calc_loss(self, x, y):

        # Prior matching:
        encoder_mean, encoder_std = self.encoder(x, y)
        prior_matching = 1/2 * (encoder_mean.norm(dim=-1) ** 2 +
encoder_std.norm(dim=-1) ** 2) - encoder_std.log().sum(dim=-1)

        # Término de reconstrucción:
        z = encoder_mean + encoder_std * torch.randn_like(encoder_mean)
        x_hat = self.decoder(z, y)
        x = x.flatten(start_dim=1)  # [batch_size, c * h * w]
        x_hat = x_hat.flatten(start_dim=1)  # [batch_size, c * h * w]
        reconstruction_term = (x * x_hat.log() + (1 - x) * (1 -
x_hat).log()).sum(dim=-1)

        elbo = reconstruction_term - prior_matching
        return - elbo.mean()

    def generate_samples(self, y, n_samples):

        self.decoder.eval()

        with torch.no_grad():
            y = torch.tensor(y, device=DEVICE).expand(n_samples)
            z = torch.randn(n_samples, self.latent_dim, device=DEVICE)
            x_hat = self.decoder(z, y)

        return x_hat
```

Entrenamiento y generación

Con la clase anterior implementada, se entrenará el VAE condicional durante 50 épocas:

```
vae = ImageVAE()
vae.train(dataloader, epochs=50)
```

Con el modelo entrenado, se generarán 10 muestras para cada etiqueta de clase $y \in \{0, \cdots, 9\}$:

```
samples = [vae.generate_samples(y, n_samples=10) for y in range(10)]
samples = torch.cat(samples, dim=0)
show_batch(samples.cpu())
```

Figura 8: Generación condicional usando el VAE entrenado sobre Fashion MNIST.

Con los modelos encoder y decoder entrenados, es posible revisar la reconstrucción realizada por el VAE. Para esto, se elige un par de muestras originales, se pasan por el encoder para obtener sus respectivas representaciones latentes y luego, dichas representaciones latentes se pasan por el decoder para generar nuevas muestras. Si el VAE está bien entrenado, las muestras generadas deben ser similares a las muestras originales:

```
vae.encoder.eval()

batch_x, batch_y = next(iter(dataloader))
x = batch_x[:10].to(DEVICE)
y = batch_y[:10].to(DEVICE)

encoder_mean, encoder_std = vae.encoder(x, y)
z = encoder_mean + encoder_std * torch.randn_like(encoder_mean)
x_hat = vae.decoder(z, y)
```

```
imgs = torch.cat([x, x_hat], dim=0)
show_batch(imgs.cpu())
```

Figura 9: Reconstrucción de muestras usando el VAE entrenado sobre Fashion MNIST.

Se observa que las muestras reconstruidas (abajo) son muy similares a las muestras originales (arriba), lo que indica que tanto el modelo generador (encoder) como el modelo codificador (encoder) están bien entrenados.

En el siguiente capítulo se aprovechará la existencia del encoder para poder realizar modificación de atributos de forma fácil. Además, se revisarán algunas propiedades y variantes del VAE clásico revisado en este capítulo.

Referencias

[1] S. Rifai, P. Vincent, X. Muller, X. Glorot, y Y. Bengio, «Contractive Auto-Encoders: Explicit Invariance During Feature Extraction», en *Proceedings of the 28th International Conference on Machine Learning (ICML)*, 2011. [En línea]. Disponible en: https://icml.cc/2011/papers/455_icmlpaper.pdf

[2] P. Vincent, H. Larochelle, Y. Bengio, y P.-A. Manzagol, «Extracting and Composing Robust Features with Denoising Autoencoders», *Proceedings of the 25th International Conference on Machine Learning (ICML)*, 2008, [En línea]. Disponible en: https://www.cs.toronto.edu/~larocheh/publications/icml-2008-denoising-autoencoders.pdf

[3] 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