Índice del capítulo

Capítulo 7

GANs y arquitecturas neuronales para imágenes

Las **redes generativas adversarias** (GANs) [1] son una familia de modelos generativos de variable latente, $p_\theta(x, z) = p(z)\, p_\theta(x \mid z)$, que constituyó el estado del arte en la generación de imágenes hasta la llegada de los modelos de difusión. A diferencia de otros paradigmas, las GANs no modelan explícitamente la densidad $p_\theta(x)$, sino que aprenden directamente un procedimiento para generar muestras desde $p_\theta(x)$ sin tener que evaluar dicha densidad. Esta característica clasifica a las GANs como **modelos generativos implícitos** [2], los cuales no pueden ser entrenados por enfoques que involucren la verosimilitud ya que no conocen $p_\theta(x)$. Si bien hay varios enfoques para evitar trabajar con la verosimilitud, las GANs utilizan un mecanismo de entrenamiento adversarial entre dos redes neuronales. En este capítulo se formulará e implementará una GAN básica, se introducirá la convolución transpuesta como bloque básico para el upsampling de los generadores, y se revisarán algunas de las arquitecturas neuronales más representativas de la tarea de generación de imágenes con GANs. Por otro lado, más allá del valor histórico de estos modelos, varias de las técnicas que se introdujeron aquí siguen siendo relevantes para el diseño de arquitecturas neuronales para otros paradigmas como los modelos de difusión.

7.1 Formulación de una GAN

7.1.1 Modelo generativo implícito

La red bayesiana asociada a una GAN es la red bayesiana de variable latente estándar, $p_\theta(x, z) = p(z)\, p_\theta(x \mid z)$, donde el prior latente se suele elegir como una distribución de la que es fácil generar muestras (e.g., $p(z) \sim \mathcal{N}(0, \mathrm{I}_L)$, aunque también se puede utilizar la distribución uniforme). En cambio, para la parte generadora $p_\theta(x \mid z)$, esta viene dada por una transformación determinista de z a través de una red neuronal $G_\theta : \mathbb{R}^L \to \mathbb{R}^D$ (denominada **generador**), es decir:

$$p_\theta(x \mid z) \sim \delta_{G_\theta(z)}(x).$$

De esta forma, $p_\theta(x \mid z)$ es una distribución que concentra toda su masa en el punto $G_\theta(z) \in \mathbb{R}^D$ (i.e., la única muestra que puede obtenerse desde $p_\theta(x \mid z)$ es $x = G_\theta(z) \in \mathbb{R}^D$). Esta es una primera diferencia con respecto a otros modelos generativos, donde lo usual es fijar distribuciones pertenecientes a alguna familia paramétrica (e.g., gaussianas o categóricas), y usar una red neuronal para aprender sus parámetros en vez de aprender directamente la transformación $z \mapsto x$.

7.1.2 Discriminador y entrenamiento adversarial

Para entrenar la red neuronal G_θ sin acceso a la función de verosimilitud, $p_\theta(x)$, una GAN introduce un clasificador binario auxiliar, $q_\phi(y \mid x)$, denominado **discriminador**, cuya tarea es distinguir si una muestra $x \in \mathbb{R}^D$ proviene de la distribución de datos $p_{\text{data}}(x)$ ($y = 1$, real) o de la distribución implícita $p_\theta(x)$ ($y = 0$, sintética). Naturalmente, el modelo discriminativo se elige como un clasificador binario estándar:

$$q_\phi(y \mid x) \sim \text{Bernoulli}\big(D_\phi(x)\big),$$

donde $D_\phi : \mathbb{R}^D \to [0, 1]$ es otra red neuronal independiente de $G_\theta : \mathbb{R}^L \to \mathbb{R}^D$. Mientras D_ϕ se entrena para clasificar correctamente, G_θ se entrena para engañarlo, lo que fuerza, como subproducto, a que $p_\theta(x)$ genere muestras similares a $p_{\text{data}}(x)$.

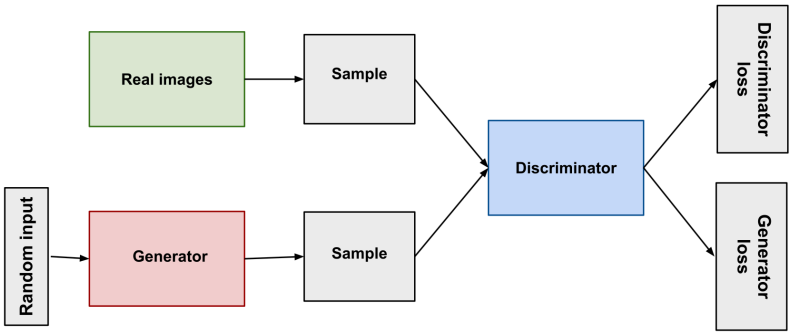

Figura 1: Diagrama de entrenamiento de una GAN. Imagen obtenida desde [3].

Es importante destacar que $q_\phi(y \mid x)$ no forma parte del modelo generativo; es solo un mecanismo de entrenamiento que se descarta una vez finalizado el ajuste del generador. La GAN, vista como modelo, sigue siendo una única red bayesiana $z \to x$.

Función objetivo

Dado que D_ϕ es un clasificador, su entrenamiento se puede realizar siguiendo el criterio de máxima verosimilitud sobre una distribución conjunta de pares (x, y) formados por imágenes reales (generadas desde $p_{\text{data}}(x)$; $y = 1$) e imágenes sintéticas (generadas desde

$p_\theta(x);\ y=0$). Si las muestras reales y sintéticas se utilizan de forma balanceada en el entrenamiento, esta distribución es

$$p_{\text{train}}(x,y) = \frac{1}{2}(p_{\text{data}}(x)\,\delta_1(y) + p_\theta(x)\,\delta_0(y)).$$

Desarrollando la log-verosimilitud del clasificador $q_\phi(y\,|\,x) \sim \text{Bernoulli}\big(D_\phi\big)$:

$$q_\phi(y \mid x) = \begin{cases} D_\phi(x) & \text{si } \ y=1 \\ 1-D_\phi(x) & \text{si } \ y=0 \end{cases} = D_\phi(x)^y\big(1-D_\phi(x)\big)^{1-y},$$

por lo que $\log q_\phi(y \mid x) = y\log D_\phi(x) + (1-y)\log\big(1-D_\phi(x)\big)$. En consecuncia, la función objetivo de una GAN es:

$$\begin{aligned}
\mathcal{L}_{\text{GAN}}(\theta,\phi) &:= \mathbb{E}_{p_{\text{train}}(x,y)}\big[\log q_\phi(y\,|\,x)\big] \\
&= \frac{1}{2}\mathbb{E}_{p_{\text{train}}(x,y)}\big[y\log D_\phi(x) + (1-y)\log\big(1-D_\phi(x)\big)\big] \\
&= \frac{1}{2}\Big(\mathbb{E}_{p_{\text{data}}(x)}\big[\log D_\phi(x)\big] + \mathbb{E}_{p_\theta(x)}\big[\log\big(1-D_\phi(x)\big)\big]\Big) \\
&= \frac{1}{2}\Big(\mathbb{E}_{p_{\text{data}}(x)}\big[\log D_\phi(x)\big] + \mathbb{E}_{p(z)}\big[\log\big(1-D_\phi(G_\theta(z))\big)\big]\Big)
\end{aligned}$$

Recordando que el discriminador (clasificador) busca maximizar su rendimiento y el generador busca que el discriminador que se equivoque, el entrenamiento de una GAN consiste en optimizar el siguiente juego minimax:

$$\min_\theta \max_\phi \mathcal{L}_{\text{GAN}}(\theta,\phi).$$

En la práctica, las esperanzas se aproximan mediante estimaciones de Monte Carlo. La primera se estima con un conjunto de entrenamiento $\mathcal{D} = \{x^1,...,x^N\} \subset \mathbb{R}^D$ de muestras i.i.d. desde $p_{\text{data}}(x)$, mientras que la segunda se estima generando muestras i.i.d. $\{z^1,...,z^N\} \subset \mathbb{R}^L$ desde $p(z) \sim \mathcal{N}(0,\mathrm{I}_L)$.

Por otro lado, es usual escribir la función objetivo de manera separada para cada red neuronal dado que muchas veces se suele modificar una (o ambas) funciones objetivo:

$$\begin{aligned}
\mathcal{L}_D(\phi) &= \mathbb{E}_{p_{\text{data}}(x)}\big[\log D_\phi(x)\big] - \mathbb{E}_{p(z)}\big[\log\big(1-D_\phi(G_\theta(z))\big)\big] \\
\mathcal{L}_G(\theta) &= \mathbb{E}_{p(z)}\big[\log\big(1-D_\phi(G_\theta(z))\big)\big],
\end{aligned}$$

donde se han omitido los factores $\frac{1}{2}$ y el término constante para θ en la verosimilitud.

Notar que esta función objetivo es **saturante** (en el sentido de que sufre de vanishing gradients), lo que es particularmente malo durante el comienzo del entrenamiento ya que en esta etapa el generador es muy débil, por lo que el discriminador puede diferenciar fácilmente una muestra real de una falsa. En consecuencia, al comienzo del entrenamiento, $D_\phi(G_\theta(z)) \approx 0$, por lo que el gradiente de $\log\big(1 - D_\phi(G_\theta(z))\big)$ respecto a θ es muy pequeño, lo que dificulta el aprendizaje.

Para resolver esto, en la práctica se entrena el generador minimizando una función de pérdida **no saturante**:

$$\mathcal{L}_G(\theta) = -\mathbb{E}_{p(z)}\left[\log D_\phi(G_\theta(z))\right].$$

Esta función de pérdida busca maximizar la probabilidad de que el discriminador clasifique las muestras falsas como reales, manteniendo el objetivo inicial del generador, pero otorgándole gradientes útiles a la red neuronal G_θ.

En la siguiente figura se puede ver la diferencia entre ambas curvas, en función de la salida del discriminador, $d = D_\phi(G_\theta(z))$:

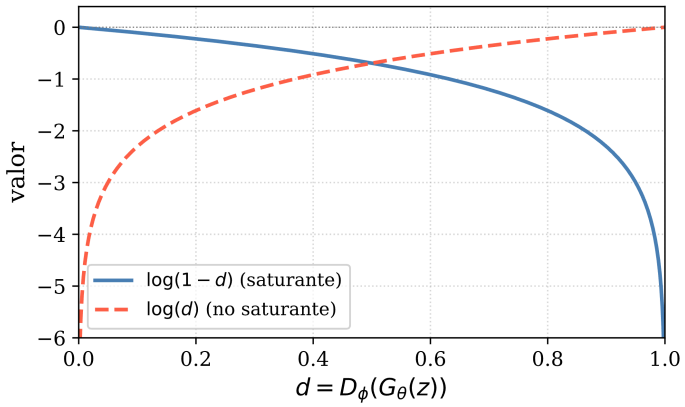

Figura 2: Función objetivo saturante y no saturante. Se observa que al comienzo del entrenamiento (cuando $d \approx 0$), la derivada de la función saturante es casi nula.

Respecto al entrenamiento basado en el juego minimax, en cada iteración se comienza optimizando los parámetros del discriminador D_ϕ varias veces (parte max), y luego se actualizan los parámetros del generador G_θ una vez (parte min). Esta estrategia, en la que el discriminador se entrena más que el generador, busca mantener al discriminador en un estado cercano al óptimo durante el entrenamiento, lo que proporciona señales de gradiente más confiables para el generador. Si el discriminador se debilita demasiado y pierde capacidad para distinguir entre datos reales y generados, el generador deja de recibir feedback útil durante el entrenamiento.

7.1.3 Implementación sobre datos bidimensionales

Para fijar la formulación se entrenará una GAN sobre un dataset bidimensional ($D = 2$). Las librerías que se utilizarán son las siguientes:

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

Como distribución de datos $p_{\text{data}}(x)$ se utiliza el dataset `make_swiss_roll` de scikit-learn [4], proyectado a $\mathbb{R}^2$, con ruido gaussiano de baja varianza para añadir variabilidad:

```
def get_batch(batch_size=1000, noise=0.1):
    x, _ = make_swiss_roll(batch_size, noise=noise)
    x = x[:, [0, 2]]
    x = (x - x.mean()) / x.std()
    return torch.tensor(x).float()

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

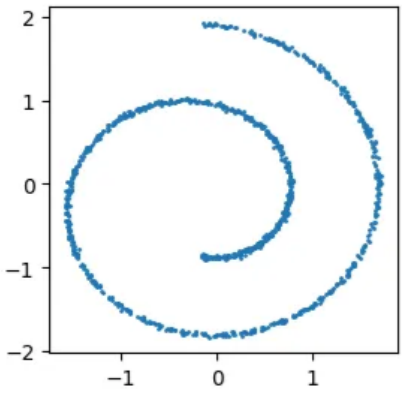

Figura 3: Muestras del dataset swiss roll proyectado a $\mathbb{R}^2$.

Para las redes neuronales $G_\theta : \mathbb{R}^L \to \mathbb{R}^D$ y $D_\phi : \mathbb{R}^D \to [0, 1]$ se utilizarán dos redes fully connected de cuatro capas. Como dimensión latente se considerará $L = 16$. Si bien lo usual es considerar $L \ll D$ (considerando la hipótesis de la variedad), en este ejemplo se considerará que $L > D$ para no colapsar la capacidad de la red neuronal del generador:

```
data_dim, latent_dim = 2, 16

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

discriminator = nn.Sequential(
    nn.Linear(data_dim, 128), nn.ReLU(),
    nn.Linear(128, 64), nn.ReLU(),
    nn.Linear(64, 32), nn.ReLU(),
    nn.Linear(32, 1), nn.Sigmoid()
)
```

Por la naturaleza competitiva entre G_θ y D_ϕ, las GANs suelen presentar una dinámica de entrenamiento inestable, lo que ha motivado el uso hiperparámetros que se sabe, principalmente por experiencia empírica, que funcionan bien. En este caso se utilizará Adam [5] con `betas=(0.5, 0.999)`, una elección común sugerida por DCGAN (revisada más abajo):

```
def train(generator, discriminator, latent_dim, iters=5000):

    # Optimizadores:
    generator_optimizer = optim.Adam(generator.parameters(), betas=(0.5,
0.999))
    discriminator_optimizer = optim.Adam(discriminator.parameters(),
betas=(0.5, 0.999))

    for iter in range(iters):

        # Datos de entrenamiento:
        x_true = get_batch()
        x_fake = generator(torch.randn([len(x_true), latent_dim]))

        # Entrenamiento discriminador:
        loss_y1 = torch.log(discriminator(x_true)).mean()
        loss_y0 = torch.log(1-discriminator(x_fake.detach())).mean()
        loss_discriminator = - 1 / 2 * (loss_y1 + loss_y0)
        discriminator_optimizer.zero_grad()
        loss_discriminator.backward()
        discriminator_optimizer.step()

        # Entrenamiento generador (versión saturante):
```

```
        loss_generator = 1 / 2 * torch.log(1-discriminator(x_fake)).mean()
        generator_optimizer.zero_grad()
        loss_generator.backward()
        generator_optimizer.step()
train(generator, discriminator, latent_dim)
```

Una vez entrenado el modelo, se pueden generar nuevas muestras mediante ancestral sampling sobre la red bayesiana $p(z)\,p_\theta(x \mid z)$ sampleando desde $z \sim p(z)$, y luego evaluando $x = G_\theta(z)$:

```
def generate_samples(n_samples=1000):
    z = torch.randn([n_samples, latent_dim])
    samples = generator(z).detach()
    return samples

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

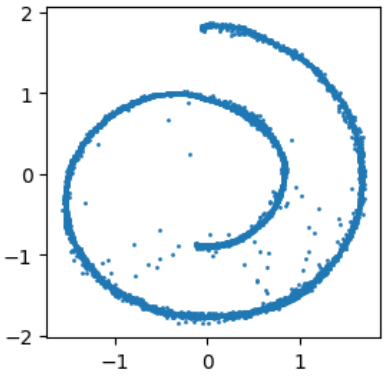

Figura 4: Muestras generadas por la GAN entrenada.

Se observa que el generador aprendió a reproducir, aproximadamente, la distribución $p_{\text{data}}(x)$ a pesar de no haber modelado explícitamente $p_\theta(x)$.

Observación 1. En la implementación anterior se incluyó una sigmoide en la salida del discriminador para que su rango sea $[0, 1]$. Una práctica más estable numéricamente es eliminar la sigmoide y trabajar directamente con los logits, usando `nn.BCEWithLogitsLoss` para el cálculo de la pérdida. La equivalencia con esta función de pérdida es directa al identificar la entropía cruzada (binaria) en la verosimilitud $\log q_\phi(y|x) = y \log D_{\phi(x)} + (1 - y) \log\big(1 - D_{\phi(x)}\big)$, por lo que:

$$\mathcal{L}_{\text{GAN}}(\theta, \phi) = -\mathbb{E}_x \left[\text{CE}\big(p_{\text{train}}(y|x), q_\phi(y|x)\big) \right]$$

Notar que esto es análogo a lo que se realiza con los modelos de lenguaje, donde usualmente no se incluye un módulo `nn.Softmax` en la salida del Transformer y se utiliza `nn.CrossEntropyLoss` para el equivalencia con la verosimilitud de una distribución categórica sobre el vocabulario.

7.1.4 Generación condicional

Al igual que toda red bayesiana, la formulación anterior se puede extender al caso condicional considerando una variable adicional $c \in \mathcal{C}$ que guíe la generación. La red bayesiana de una **GAN condicional** (CGAN) [6] se factoriza como $p_\theta(x, z \mid c) = p(z)\, p_\theta(x \mid c, z)$, donde implícitamente se ha asumido que $z \perp c$ (i.e., el prior latente es independiente del factor condicionante). Además, tanto el generador, $G_\theta : \mathcal{C} \times \mathbb{R}^L \to \mathbb{R}^D$, como el discriminador, $D_\phi : \mathcal{C} \times \mathbb{R}^D \to [0, 1]$, ahora reciben la condición $c \in \mathcal{C}$ como entrada adicional. Con estos cambios, la función objetivo de una CGAN es:

$$\mathcal{L}_{\text{CGAN}}(\theta, \phi) \coloneqq \mathbb{E}_{p_{\text{data}}(c)\, p_{\text{data}}(x \mid c)} \big[\log D_\phi(c, x)\big] + \mathbb{E}_{p(c)\, p(z)} \big[\log\big(1 - D_\phi(c, G_\theta(c, z))\big)\big],$$

donde $p(c)$ es una distribución sobre $\mathcal{C}$ que no es modelada ni aprendida (los valores de c son sampleados durante el entrenamiento). La forma en la que se inyecta c en las redes neuronales depende de su naturaleza:

- **Etiqueta de clase $c \in \{1, \ldots, K\}$.** Lo más simple es concatenar el valor de c a las entradas. Una opción más expresiva es utilizar una matriz de embedding $E \in \mathcal{M}_{K, D'}(\mathbb{R})$, de forma análoga a los modelos de lenguaje, donde la c-ésima fila de E es un vector aprendido asociado a la clase c. Para evitar parámetros adicionales se puede usar el vector one-hot de cada clase como embedding fijo.
- **Texto $c \in \mathcal{V}^*$.** Una opción simple y efectiva es pasar la secuencia de tokens por un modelo de embeddings preentrenado como CLIP, BERT o T5, y utilizar el vector resultante como entrada adicional al generador. Una alternativa más sofisticada, utilizada por modelos como Stable Diffusion [7], consiste en inyectar la secuencia de embeddings mediante atención cruzada en bloques específicos de la red.
- **Imagen $c \in \mathbb{R}^{C \times H \times W}$.** Para tareas image-to-image (e.g., colorización, super-resolución, inpainting) es usual utilizar arquitecturas especializadas como la U-Net (revisada más abajo). Además, para mezclar texto con imagen, se pueden usar arquitecturas más sofisticadas como la usada por FLUX (modelo de flow matching), donde se trabajan tokens de imagen de forma similar.

7.1.5 Implementación para MNIST

A modo de ejemplo, se implementará una CGAN sobre el dataset MNIST, donde la condición $c \in \{0, ..., 9\}$ será el dígito asociado a la imagen. Se comenzará instanciando el dataset con las imágenes normalizadas al intervalo $[-1, 1]$ (otra recomendación de DCGAN):

```
transform = transforms.Compose([
    transforms.ToTensor(),  # normalización [0, 1].
    transforms.Normalize((0.5,), (0.5,))  # normalización [-1, 1].
])

dataset = datasets.MNIST(root='data', train=True, download=True,
transform=transform)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True,
drop_last=True, num_workers=2)

# Ejemplo:
fig, axes = plt.subplots(1, 10, figsize=(10, 1.2))
for cond in range(10):
    idx = (dataset.targets == cond).nonzero()[0].item()
    x, cond_true = dataset[idx]
    img = x.permute(1, 2, 0) * 0.5 + 0.5  # desnormalización a [0, 1].
    axes[cond].imshow(img, cmap='gray_r')
    axes[cond].set_title(cond_true)
    axes[cond].axis('off')
plt.show()
```

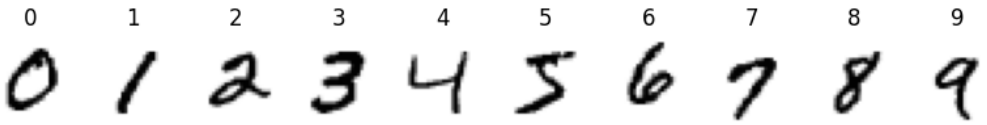

Figura 5: Muestras de MNIST con sus respectivas etiquetas.

Como redes neuronales se utilizarán MLPs simples, dejando el estudio de las arquitecturas convolucionales para la siguiente sección. Para el generador:

```
class Generator(nn.Module):

    def __init__(self, data_dim, n_classes, cond_embed_dim, latent_dim):
        super().__init__()
        self.cond_embed = nn.Embedding(n_classes, cond_embed_dim)
        self.net = nn.Sequential(
            nn.Linear(latent_dim + cond_embed_dim, 256),
nn.BatchNorm1d(256), nn.LeakyReLU(0.2),
```

```
            nn.Linear(256, 512), nn.BatchNorm1d(512), nn.LeakyReLU(0.2),
            nn.Linear(512, 1024), nn.BatchNorm1d(1024), nn.LeakyReLU(0.2),
            nn.Linear(1024, data_dim), nn.Tanh()
        )
        self.apply(self._init_weights)

    @staticmethod
    def _init_weights(m):
        if isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, 0.0, 0.02)
            nn.init.zeros_(m.bias)

    def forward(self, z, cond):
        cond = self.cond_embed(cond)
        input = torch.cat([z, cond], dim=1)
        return self.net(input)
```

Notar que se utilizaron heurísticas como inicialización de parámetros y uso de capas de normalización. Como se verá en las siguientes secciones, estas modificaciones permiten tener un entrenamiento más estable.

Para el discriminador se utilizará una red análoga, donde no se incluye `nn.Softmax` en la salida para trabajar con `nn.BCEWithLogitsLoss`:

```
class Discriminator(nn.Module):

    def __init__(self, data_dim, n_classes, cond_embed_dim):
        super().__init__()
        self.cond_embed = nn.Embedding(n_classes, cond_embed_dim)
        self.net = nn.Sequential(
            nn.Linear(data_dim + cond_embed_dim, 1024), nn.LeakyReLU(0.2),
nn.Dropout(0.3),
            nn.Linear(1024, 512), nn.LeakyReLU(0.2), nn.Dropout(0.3),
            nn.Linear(512, 256), nn.LeakyReLU(0.2), nn.Dropout(0.3),
            nn.Linear(256, 1)
        )
        self.apply(self._init_weights)

    @staticmethod
    def _init_weights(m):
        if isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, 0.0, 0.02)
            nn.init.zeros_(m.bias)
```

```
    def forward(self, x, cond):
        cond = self.cond_embed(cond)
        input = torch.cat([x.flatten(1), cond], dim=1)
        return self.net(input)
```

Para el entrenamiento, ahora se utilizará la función de pérdida no saturante (recomendación de la GAN original). Además, se utilizará label smoothing para el discriminador (similar a lo hecho en ARMs), lo que ayuda a evitar la saturación y mejorar la estabilidad del entrenamiento:

```
def train(generator, discriminator, latent_dim, epochs=50,
label_smoothing=0.9, g_steps=2):

    generator.to(DEVICE).train()
    discriminator.to(DEVICE).train()

    # Optimizadores:
    generator_optimizer = optim.Adam(generator.parameters(), lr=2e-4,
betas=(0.5, 0.999))
    discriminator_optimizer = optim.Adam(discriminator.parameters(),
lr=2e-4, betas=(0.5, 0.999))

    criterion = nn.BCEWithLogitsLoss()
    losses = {'generator': [], 'discriminator': []}

    try:
        for epoch in tqdm(range(epochs)):
            for x_true, cond_true in dataloader:
                batch_size = len(x_true)
                ones = torch.ones(batch_size, 1, device=DEVICE)
                zeros = torch.zeros(batch_size, 1, device=DEVICE)

                # Datos de entrenamiento:
                # - True:
                cond_true = cond_true.to(DEVICE)
                x_true = x_true.to(DEVICE)
                # - Fake:
                z = torch.randn(batch_size, latent_dim, device=DEVICE)
                x_fake = generator(z, cond_true)

                # Entrenamiento discriminador:
                # - Loss:
                loss_d_real = criterion(discriminator(x_true, cond_true),
ones * label_smoothing)
```

```
                loss_d_fake = criterion(discriminator(x_fake.detach(),
cond_true), zeros)
                loss_discriminator = 1/2 * (loss_d_real + loss_d_fake)
                # - Step:
                discriminator_optimizer.zero_grad()
                loss_discriminator.backward()
                discriminator_optimizer.step()

                # Entrenamiento generador (versión no saturante):
                for _ in range(g_steps):
                    z = torch.randn(batch_size, latent_dim, device=DEVICE)
                    x_fake = generator(z, cond_true)
                    loss_generator = criterion(discriminator(x_fake,
cond_true), ones)
                    generator_optimizer.zero_grad()
                    loss_generator.backward()
                    generator_optimizer.step()

                # Log:
                losses['generator'].append(loss_generator.item())
                losses['discriminator'].append(loss_discriminator.item())

    except KeyboardInterrupt:
        print('Entrenamiento interrumpido.')

    # Gráfico de entrenamiento:
    plt.plot(losses['generator'], label='Generador', alpha=0.7)
    plt.plot(losses['discriminator'], label='Discriminador', alpha=0.7)
    plt.xlabel('Step');
    plt.ylabel('Loss');
    plt.legend();
    plt.grid(alpha=0.3)
    plt.show()
```

Con esto, se pueden instanciar las redes neuronales y realizar el entrenamiento:

```
data_dim = dataset[0][0].numel()
n_classes = len(dataset.classes)
cond_embed_dim = 16
latent_dim = 64

generator = Generator(data_dim, n_classes, cond_embed_dim, latent_dim)
discriminator = Discriminator(data_dim, n_classes, cond_embed_dim)
```

```
train(generator, discriminator, latent_dim)
```

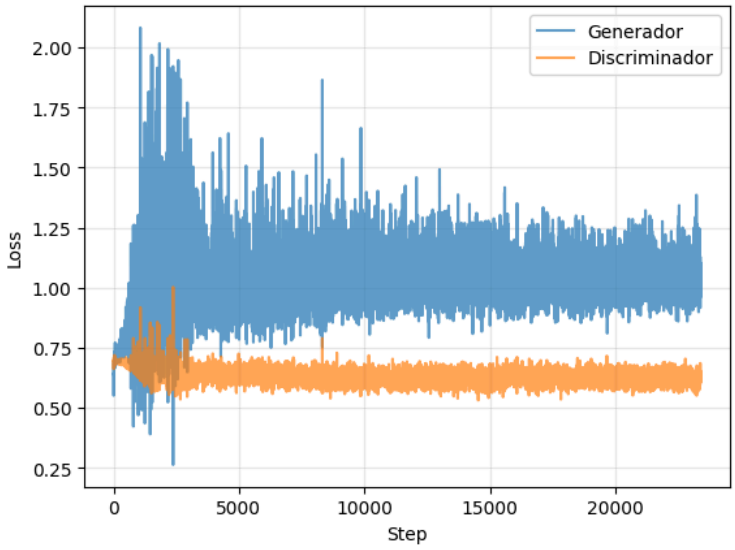

Figura 6: Dinámica de evolución de la CGAN.

Se observa que la dinámica de entrenamiento de la GAN es mucho más inestable e impredecible que la dinámica de, por ejemplo, los modelos autorregresivos. Más aún, la función de pérdida de ambos modelos no parece entregar ningún indicador obvio de cuándo detener el entrenamiento, o cómo detectar overfitting.

Para la generación, se utiliza la misma técnica de ancestral sampling usada en el caso incondicional, solo que además se debe incluir una condición $c \in \mathcal{C}$ en la red generadora:

```
@torch.no_grad()
def generate_samples(cond, n_samples):
    generator.eval()
    z = torch.randn(n_samples, latent_dim, device=DEVICE)
    cond = torch.full((n_samples,), cond, dtype=torch.long, device=DEVICE)
    samples = generator(z, cond)
    return samples.cpu()
```

Usando la función anterior, se generarán algunas muestras para cada etiqueta de clase:

```
n_samples = 15

fig, axes = plt.subplots(n_classes, n_samples, figsize=(n_samples,
n_classes))
for cond in range(n_classes):
    samples = generate_samples(cond=cond, n_samples=n_samples)
```

```
    samples = (samples * 0.5 + 0.5).clamp(0, 1).view(len(samples), 28, 28)
    for j in range(n_samples):
        axes[cond, j].imshow(samples[j], cmap='gray_r')
        axes[cond, j].axis('off')
    axes[cond, 0].set_ylabel(str(cond), fontsize=10)
plt.tight_layout()
plt.show()
```

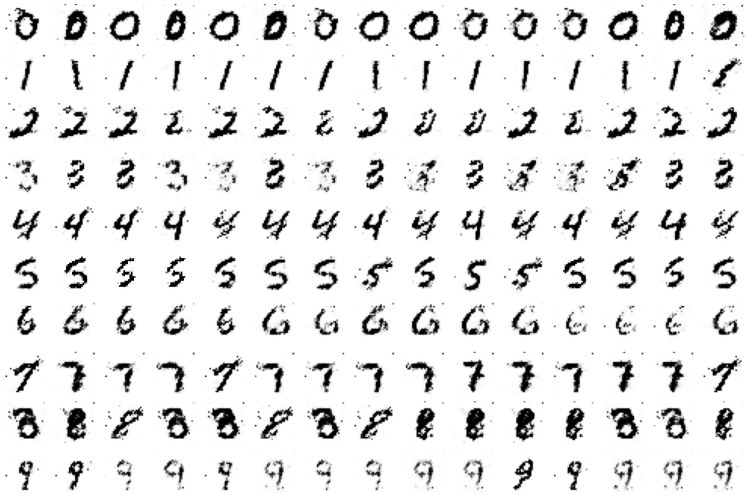

Figura 7: Muestras generadas por la CGAN para cada etiqueta de clase.

En la siguiente sección se estudiarán arquitecturas neuronales especializadas para imágenes, las cuales combinan distintas técnicas de diseño para inducir sesgos útiles en este tipo de datos. Como se verá, si bien las GANs tienen la capacidad de llegar a generar imágenes hiperrealistas, la principal limitación de este tipo de modelos es su inestabilidad durante el entrenamiento. Debido a esto, muchos de los trabajos en GANs consistieron principalmente en introducir modificaciones de arquitectura y heurísticas de optimización.

7.2 Redes convolucionales transpuestas

El proceso generativo de una GAN para imágenes consiste en transformar una muestra latente de baja dimensión, $z \in \mathbb{R}^L$, en una imagen de alta dimensión, $x = G_\theta(z) \in \mathbb{R}^D$, donde $D = C \times H \times W$ para una imagen de resolución $H \times W$ y C canales. Cuando $L \ll D$, este mapeo requiere un aumento progresivo de resolución a través de la red neuronal G_θ. Si bien esto se podría implementar con capas fully connected, este enfoque no escala bien a altas resoluciones ni aprovecha la estructura espacial de las imágenes. La operación natural que permite realizar este upsampling preservando la estructura espacial es la **convolución**

transpuesta, también llamada **convolución con stride fraccionario** (o mal llamada *deconvolución*).

7.2.1 Convolución como operador lineal

Antes de estudiar las convoluciones transpuestas, se recordará el concepto de convolución[1] usado en una red convolucional simple. Para esto, se considerará el caso simplificado unidimensional ($D = 1$) con un solo canal.

Sea $x = \left(x_1, ..., x_{D_{\text{in}}}\right) \in \mathbb{R}^{D_{\text{in}}}$ una entrada y $w = (w_1, ..., w_K) \in \mathbb{R}^K$ un **filtro convolucional** de tamaño $K \geq 1$. La convolución $\text{Conv} : \mathbb{R}^{D_{\text{in}}} \to \mathbb{R}^{D_{\text{out}}}$ con **stride** $S \geq 1$ y **padding** $P \geq 0$ produce un vector $y \in \mathbb{R}^{D_{\text{out}}}$ con componentes

$$y_j = \sum_{k=1}^{K} w_k \, x_{(j-1)S+k-P}, \quad j \in \{1, ..., D_{\text{out}}\},$$

donde $D_{\text{out}} = \left\lfloor \frac{D_{\text{in}}+2P-K}{S} \right\rfloor + 1$ y se considera $x_i = 0$ para $i \notin \{1, ..., D_{\text{in}}\}$ (zero-padding implícito). Se puede observar que la convolución es un operador lineal (i.e., $\text{Conv}(\alpha u + \beta v) = \alpha \, \text{Conv}(u) + \beta \, \text{Conv}(v)$), por lo que existe una matriz $A \in \mathcal{M}_{D_{\text{out}}, D_{\text{in}}}(\mathbb{R})$ tal que $y = Ax$. En el caso $S = 1$, $P = 0$, esta matriz tiene estructura de **Toeplitz**. Por ejemplo, para $D_{\text{in}} = 5$, $K = 3$, $S = 1$ y $P = 0$:

$$A = \begin{pmatrix} w_1 & w_2 & w_3 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 \end{pmatrix}.$$

En consecuencia, una red convolucional puede verse como una red fully connected con una matriz de pesos restringida a una estructura específica, lo que reduce drásticamente el número de parámetros (de $D_{\text{out}} D_{\text{in}}$ a K) y además induce sesgos útiles. En particular, dado que se comparten los pesos del kernel w a lo largo de todas las posiciones espaciales de la entrada, la convolución es **equivariante por traslación** (si un trozo de la entrada se traslada, la salida se traslada de la misma forma). Además, cuando la convolución se combina con operaciones de pooling, la red puede volverse **invariante por traslación local** (la salida no cambia si se traslada localmente la entrada). Estos sesgos inductivos son especialmente útiles en visión computacional, donde se asume que los patrones relevantes (bordes, texturas, objetos) aparecen en cualquier posición de la imagen, por lo que la red no debe depender de la ubicación absoluta de estos patrones. De forma similar, este tipo

[1] Más correctamente, esta operación corresponde a la **correlación cruzada**, ya que la convolución contiene el índice desfasado e invertido. En la práctica ambos conceptos son equivalentes a nivel de capacidad de aprendizaje, y el nombre de convolución es el más usual en deep learning.

de convoluciones 1D puede utilizarse para modelar el lenguaje. En este caso, el tamaño del kernel convolucional indica el tamaño de la ventana de contexto del modelo.

En el caso de más dimensión (e.g. imágenes 2D), la operación de convolución se aplica de forma independiente en cada uno de los ejes, por lo que la extensión es directa. Por otro lado, en general las convoluciones operan sobre múltiples canales de entrada (e.g. 3 canales de colores y un canal de transparencia) y producen múltiples canales de salida (cada canal un **feature map**). En estos casos, si la entrada tiene C_{in} canales de entrada, la convolución utiliza un kernel diferente por cada canal, y las contribuciones de todos los canales se suman (elemento a elemento) para producir un único canal de salida[2]. De forma dual, si se necesitan C_{out} canales de salida, se utiliza un bloque independiente de C_{in} kernels por cada canal de salida, y las salidas se concatenan a lo largo de la dimensión de canales. En consecuencia, una capa convolucional estándar necesita aprender $C_{\text{out}} \times C_{\text{in}} \times K$ parámetros (más parámetros de sesgo si se consideran).

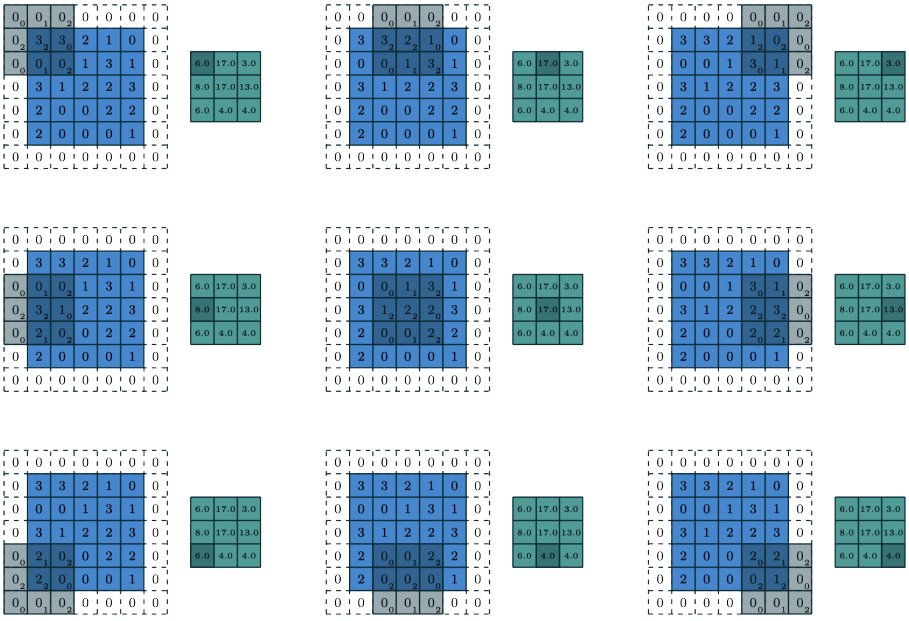

Figura 8: Convolución 2D con $K = 3$, $S = 2$ y $P = 1$ (mismo valor en ambas dimensiones).

7.2.2 Pooling

Las operaciones de **pooling** reducen la resolución espacial de un feature map sin cambiar el número de canales, aplicando una función de agregación sobre ventanas locales de la

[2]Equivalentemente, se puede considerar un único kernel $(D+1)$-dimensional que convolucione con los C_{in} canales de entrada.

entrada. Las dos operaciones más comunes son **max pooling** (retornar el valor máximo dentro de cada ventana) y **average pooling** (retornar el promedio de cada ventana). Por ejemplo, para un feature map de entrada de tamaño $C \times H \times W$, un max pooling 2×2 con $S = 2$ produce una salida de tamaño $C \times \frac{H}{2} \times \frac{W}{2}$, donde cada posición de la salida es el máximo de una ventana 2×2 de la entrada[3]. En arquitecturas típicas, el pooling se utiliza luego de convoluciones 3×3, las cuales no cambian la resolución (e.g., usando $P = 1$, $S = 1$) pero sí el número de canales.

7.2.3 Convolución transpuesta

Dada una convolución directa con matriz asociada $A \in \mathcal{M}_{D_{\text{out}}, D_{\text{in}}}(\mathbb{R})$, la **convolución transpuesta** es el operador lineal $A^\top : \mathbb{R}^{D_{\text{out}}} \to \mathbb{R}^{D_{\text{in}}}$ (i.e., $y = A^\top x$). Considerando que $D_{\text{in}} \geq D_{\text{out}}$ en una convolución tradicional, se observa que la convolución transpuesta es un operador que permite expandir la dimensión de la entrada, al mismo tiempo que se mantienen los sesgos inductivos que caracterizan a la convolución directa. En particular, este tipo de operador permite realizar un aumento de la resolución a medida que se avanza en una red neuronal generadora (como en una GAN o en un VAE).

Interpretando la convolución directa como un mecanismo que *agrega* información de la entrada (cada elemento de la salida recibe contribuciones de K elementos de la entrada), la convolución transpuesta puede interpretarse como un mecanismo que *distribuye* el valor de cada elemento de la entrada sobre K posiciones de la salida. Más aún, se puede ver que la convolución transpuesta con stride S es operacionalmente equivalente a insertar $S - 1$ ceros entre cada par de elementos consecutivos de la entrada y luego aplicar una convolución directa con stride 1 y padding $K - 1 - P$. Esta interpretación explica el nombre **stride fraccionario** (el filtro avanza efectivamente $\frac{1}{S}$ posiciones por cada elemento del input original).

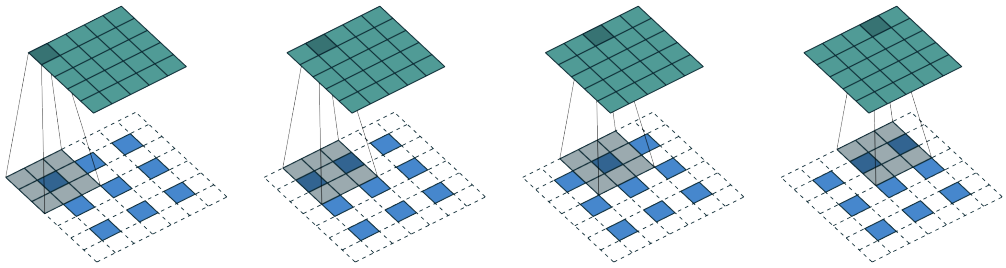

Figura 9: Convolución transpuesta 2D vista como una convolución con stride fraccionario.

[3]En particular, max pooling puede considerarse como un detector de características que se activa cuando al menos un elemento del campo receptivo posee la feature buscada (invarianza local).

Por otra parte, despejando D_{in} desde $D_{\text{out}} = \frac{D_{\text{in}}+2P-K}{S} + 1$ (asumiendo división exacta), es posible obtener la dimensión de salida que produce una convolución transpuesta con kernel de tamaño K, stride S y padding P sobre una entrada de largo D_{in}:

$$D_{\text{out}} = S(D_{\text{in}} - 1) + K - 2P.$$

En arquitecturas generativas las convolución con $K = 4$, $S = 2$, $P = 1$ y la convolución con $K = 2$, $S = 2$, $P = 0$ duplican la resolución espacial (i.e., $D_{\text{out}} = 2D_{\text{in}}$), por lo que son convoluciones transpuestas usuales en redes neuronales generativas.

Artefactos de tablero

Cuando K no es múltiplo de S, el filtro se superpone de forma desigual sobre la salida, lo que provoca que algunas posiciones reciban contribuciones de $\lceil \frac{K}{S} \rceil$ elementos de la entrada, mientras que otras reciben contribuciones de $\lfloor \frac{K}{S} \rfloor$ elementos.

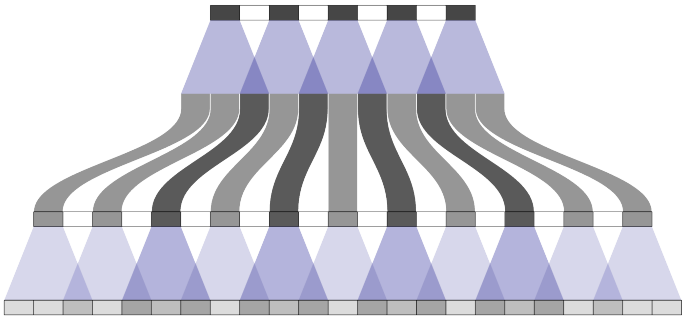

Figura 10: Periodicidad en la cantidad de contribuyentes para cada posición de salida.

Esta periodicidad se manifiesta como un patrón visual de cuadrícula en la imagen generada, conocido como **checkerboard artifact**.

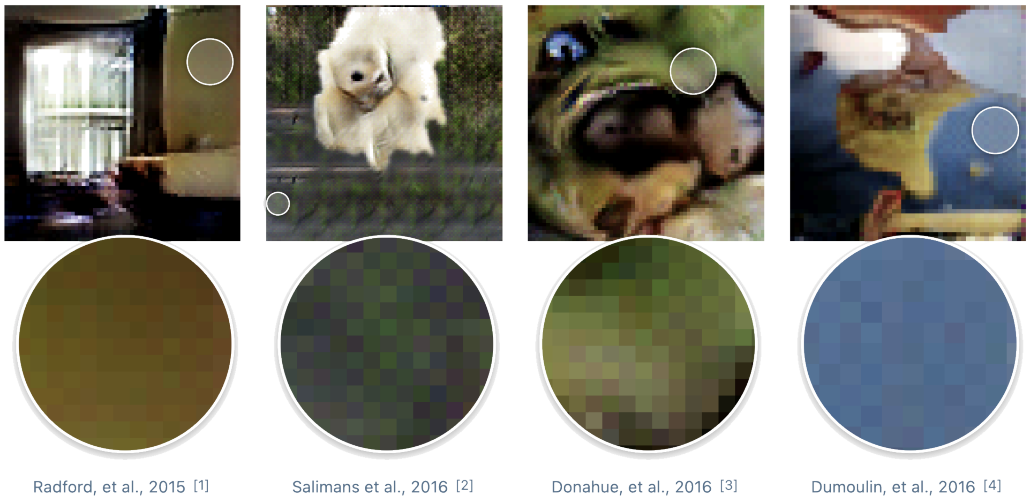

Figura 11: Ejemplos de checkboard artifacts en distintas imágenes generadas.

Existen dos estrategias usuales para evitar estos artefactos:

1. Elegir K múltiplo de S (típicamente $K = 2S$), de modo que todas las posiciones reciban el mismo número de contribuyentes.
2. Reemplazar la convolución transpuesta por una interpolación explícita (e.g., bilineal o nearest neighbor) para aumentar la resolución, seguida de una convolución que cambie la cantidad de canales pero no la resolución (e.g., $S = 1$, $K = 3$, $P = 1$).

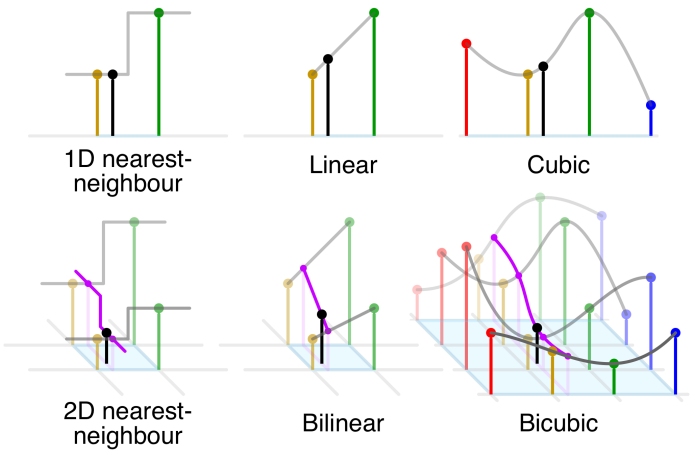

Figura 12: Comparación de interpolaciones en los casos $D = 1$ y $D = 2$.

7.3 Algunas GANs famosas

Las GANs constituyeron el estado del arte para la generación de imágenes entre los años 2014 y 2021, hasta la consolidación de los modelos de difusión. Como se mencionó, una parte importante de los trabajos sobre GANs estuvo enfocada en desarrollar mejores arquitecturas neuronales, con el objetivo de mitigar las inestabilidades del entrenamiento y propagar la información condicional de forma efectiva. Otra línea de investigación se dedicó a modificar la función de entrenamiento original, proponiendo formulaciones más estables o ligadas a divergencias con mejores propiedades geométricas (algunas divergencias serán estudiadas en el siguiente capítulo). Una tercera línea estuvo enfocada en obtener resultados teóricos sobre las GANs, como propiedades de convergencia y conexiones con teoría de juegos.

El siguiente diagrama muestra un orden temporal de los modelos tipo GAN más relevantes, junto con otros desarrollos contemporáneos como contexto histórico:

```
gantt
    dateFormat DD-MM-YYYY
    axisFormat %Y

    section Otros
        BatchNorm :milestone, 11-02-2015, 0d
        UNet :milestone, 18-05-2015, 0d
        ResNet :milestone, 10-12-2015, 0d
        LayerNorm :milestone, 21-06-2016, 0d
        GroupNorm :milestone, 22-03-2018, 0d

    section ARMs
        Seq2seq :milestone, 07-09-2014, 0d
        Transformer :milestone, 12-06-2017, 0d
        GPT :milestone, 15-05-2018, 0d
        GPT 2 :milestone, 14-02-2019, 0d

    section GANs
        GAN :milestone, 10-06-2014, 0d
        CGAN :milestone, 06-11-2014, 0d
        DCGAN :milestone, 19-11-2015, 0d
        WGAN :milestone, 26-01-2017, 0d
        Pix2Pix :milestone, 21-11-2016, 0d
        CycleGAN :milestone, 30-03-2017, 0d
        ProGAN :milestone, 27-10-2017, 0d
        SAGAN :milestone, 21-05-2018, 0d
        StyleGAN :milestone, 12-12-2018, 0d
```

En esta sección se revisarán algunas arquitecturas y modelos tipo GAN que se consideran relevantes en el campo. Si bien hay muchos otros trabajos importantes, la elección de estos modelos es principalmente debido a que algunas de sus propuestas fueron hereadadas a arquitecturas más modernas usadas en modelos de difusión (e.g. el uso de la U-Net con módulos de atención). Si bien no se seguirá el orden cronológico, cada arquitectura se motivará como respuesta a una limitación concreta de la anterior. En particular, DCGAN partirá dando reglas básicas para estabilizar el entrenamiento de GANs convolucionales, SAGAN incorporará dependencias globales para mejorar la coherencia geométrica, ProGAN permitirá escalar a mayores resoluciones mediante un entrenamiento progresivo, y StyleGAN reorganizará el generador para desacoplar factores de variación de alto y bajo nivel.

7.3.1 Deep convolutional GAN

La GAN original utilizaba redes fully connected, las cuales no aprovechan la estructura espacial de las imágenes y no escalan a resoluciones altas. **Deep convolutional GAN (DCGAN)** [8] reemplaza estas redes por arquitecturas convolucionales profundas y propone, mediante una extensa exploración empírica, un conjunto de restricciones arquitectónicas que estabilizan el entrenamiento. La contribución central de este trabajo es haber definido reglas generales para el diseño de arquitecturas para GANs. Adicionalmente, DCGAN mostró que tanto el generador como el discriminador aprenden representaciones semánticamente estructuradas, lo cual es destacable considerando que el entrenamiento es, al igual que en los modelos de lenguaje, autosupervisado (las etiquetas $y \in \{0, 1\}$ se obtienen automáticamente).

Restricciones arquitectónicas

Una técnica ampliamente utiliza en GANs es **batch normalization (BN)** [9]. Este módulo normaliza las pre-activaciones de cada neurona a media nula y varianza unitaria, y luego les aplica una transformación afín con parámetros aprendibles $\alpha_c, \beta_c \in \mathbb{R}$, con cada par de parámetros aprendido de forma independiente para cada canal.

Si $B = (x^1, ..., x^N) \in \mathbb{R}^{N \times C \times H \times W}$ es un batch de imágenes[4], durante el entrenamiento BN devuelve un tensor del mismo tamaño definido componente a componente por

$$\mathrm{bn}(B)^n_{c,h,w} := \alpha_c \frac{x^n_{c,h,w} - \mu_c}{\sqrt{\sigma_c^2 + \varepsilon}} + \beta_c,$$

[4]En general, un objeto tipo imagen es un tensor 3D formado por un conjunto de mapas de características 2D, donde cada mapa de características corresponde a un canal de la imagen. En particular, se le dirá imagen a la entrada y salida de cualquier bloque convolucional 2D.

para todo $(n, c, h, w) \in \{1, ..., N\} \times \{1, ..., C\} \times \{1, ..., H\} \times \{1, ..., W\}$, donde

$$\mu_c \coloneqq \frac{1}{NHW} \sum_{n=1}^{N} \sum_{h=1}^{H} \sum_{w=1}^{W} x^n_{c,h,w} \in \mathbb{R}, \qquad \sigma_c^2 \coloneqq \frac{1}{NHW} \sum_{n=1}^{N} \sum_{h=1}^{H} \sum_{w=1}^{W} \left(x^n_{c,h,w} - \mu_c\right)^2 \geq 0,$$

y $\varepsilon > 0$ es un hiperparámetro de estabilidad numérica. En inferencia, los estadísticos del batch, $(\mu_c, \sigma_c^2) \in \mathbb{R}^2$, $c \in \{1, ..., C\}$, se sustituyen por estimaciones globales $(\hat{\mu}_c, \hat{\sigma}_c^2) \in \mathbb{R}^2$ acumuladas durante el entrenamiento mediante una media móvil exponencial[5].

DCGAN propone construir una GAN para imágenes utilizando redes convolucionales con las siguientes cinco restricciones:

1. **Batch normalization en G_θ y D_ϕ:** se debe aplicar BN a todas las capas intermedias de G_θ y D_ϕ, excepto en la salida de G_θ (cuya distribución de píxeles no debe ser forzada a ser gaussiana) y en la entrada de D_ϕ (donde mezclar estadísticas de muestras reales y sintéticas en la primera capa degrada el clasificador).
2. **Sustitución de pooling por convoluciones con stride:** el downsampling en D_ϕ se debe realizar mediante convoluciones con $S = 2$ en vez de usar max-pooling, mientras que el upsampling en G_θ se debe realizar mediante convoluciones transpuestas con $S = 2$. En ambos casos, estos cambios le permite a la red aprender sus propias funciones de muestreo (kernels) en lugar de usar operadores fijos.
3. **Eliminación de capas fully connected ocultas:** en las primeras GANs, tanto G_θ como D_ϕ eran redes fully connected que trataban la imagen como un vector plano. En DCGAN el código latente $z \in \mathbb{R}^L$ se proyecta linealmente a un tensor en $\mathbb{R}^{C \times H \times W}$ mediante una única capa lineal seguida de un reshape, y el resto de la arquitectura opera exclusivamente con convoluciones.
4. **Activaciones en G_θ:** en el generador, todas las capas ocultas deben usar ReLU, excepto la capa de salida que debe usar tanh, cuyo rango $(-1, 1)$ es simétrico alrededor de cero[6]. Se prefiere tanh sobre la sigmoide estándar debido a que tanh tiene gradientes más fuertes cerca de los valores extremos, lo que facilita el aprendizaje de píxeles con intensidades altas o bajas. Además, el rango simétrico $(-1, 1)$ es más natural para representar imágenes normalizadas que el rango asimétrico $(0, 1)$ de la sigmoide.
5. **Activaciones en D_ϕ:** en el discriminador, todas las capas deben usar $\text{LeakyReLU}_\alpha(x) \coloneqq \begin{cases} x & \text{si } x \geq 0 \\ \alpha x & \text{si } x < 0 \end{cases}$ en lugar de ReLU (LeakyReLU con $\alpha = 0$). Esta elección garantiza gradientes no nulos en todo $\mathbb{R}$, lo que es crítico al inicio del entrenamiento, cuando G_θ produce imágenes de muy mala calidad.

[5]Tras cada batch de entrenamiento se actualiza $\hat{\mu}_c \leftarrow \lambda \hat{\mu}_c + (1 - \lambda)\mu_c$, donde $\lambda \in [0, 1]$ es un hiperparámetro de momentum. De forma análoga se actualiza $\hat{\sigma}_c^2$.

[6]Las imágenes de entrenamiento se deben normalizar al mismo rango.

Hiperparámetros de Adam

El algoritmo de optimización **Adam** [5] mantiene dos estimaciones acumuladas para cada parámetro $\theta \in \mathbb{R}^P$, las cuales corresponden al primer y segundo momento de los gradientes. Estas estimaciones se actualizan en cada step de entrenamiento mediante

$$m_t = \beta_1 \, m_{t-1} + (1 - \beta_1) \, g_t$$
$$v_t = \beta_2 \, v_{t-1} + (1 - \beta_2) \, g_t^2,$$

donde $g_t \in \mathbb{R}^P$ es el gradiente en el paso t y el cuadrado en $g_t^2 = g_t \odot g_t \in \mathbb{R}^P$ se realiza coordenada a coordenada. El parámetro $\beta_1 \in [0, 1)$ controla cuánta memoria mantiene Adam de la dirección media de los gradientes pasados (mayor β_1 implica más momentum), mientras que $\beta_2 \in [0, 1)$ controla la memoria para estimar la varianza. Tras aplicar corrección de sesgo, $\hat{m}_t = \frac{m_t}{1-\beta_1^t} \in \mathbb{R}^P$ y $\hat{v}_t = \frac{v_t}{1-\beta_2^t} \in \mathbb{R}^P$, la actualización de parámetros que realiza Adam es

$$\theta_t = \theta_{t-1} - \alpha \, \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

con las operaciones de raíz y división aplicada coordenada a coordenada. Notar que la corrección de sesgo es necesaria ya que, al inicio del entrenamiento (t pequeño), las estimaciones m_t y v_t están sesgadas hacia cero (ya que se inicializan en cero y los factores $\beta_1, \beta_2 \in [0, 1)$ son cercanos a 1). Dividir por $1 - \beta_i^t$ cancela este sesgo, ya que $(1 - \beta^t) \to 1$ cuando $t \to \infty$. La división $\frac{\widehat{m}_t}{\sqrt{\hat{v}_t}}$ normaliza cada componente del gradiente por su desviación estándar histórica, implementando una tasa de aprendizaje adaptativa por parámetro que acelera la convergencia en direcciones con gradientes consistentes, y frena el aprendizaje en direcciones ruidosas.

Si bien los valores por defecto de Adam son $(\beta_1, \beta_2) = (0.9, 0.999)$, DCGAN propone utilizar $\beta_1 = 0.5$. Durante el entrenamiento de una GAN, las redes neuronales G_θ y D_ϕ son adaptadas de manera adversativa, por lo que los gradientes del generador suelen cambiar de signo con frecuencia (ya que una dirección favorable en el paso t puede volverse perjudicial para G_θ en el paso $t + 1$ si el discriminador ha cambiado). Con $\beta_1 = 0.9$, Adam promedia gradientes a lo largo de aproximadamente 10 pasos[7] (ya que $\frac{1}{1-0.9} = 10$), por lo que, en una GAN, la actualización estándar de Adam muchas veces seguiría direcciones obsoletas, produciendo oscilaciones. DCGAN reduce β_1 a 0.5 ya que esta elección acorta

[7]La cantidad de pasos de memoria efectiva de una media móvil exponencial (EMA) con parámetro β se puede estimar como $\frac{1}{1-\beta}$. Esto proviene de observar que en la actualización $m_t = \beta m_{t-1} + (1 - \beta) g_t$, el peso del gradiente en el paso $t - k$ es proporcional a $\beta^k(1 - \beta)$, que decae exponencialmente. Luego, la suma acumulada de pesos, $\sum_{k=0}^{\infty} \beta^k = \frac{1}{1-\beta}$, entrega una medida del horizonte temporal efectivo de la EMA.

la memoria efectiva a aproximadamente $\frac{1}{1-0.5} = 2$ pasos, haciendo a Adam más reactivo a la información reciente de los gradientes.

Propiedades emergentes del espacio latente

El trabajo de DCGAN mostró que las representaciones aprendidas por una GAN, aun en un escenario autosupervisado, capturan propiedades semánticas no triviales. En particular, al utilizar las activaciones convolucionales intermedias del discriminador como entrada para un clasificador lineal entrenado sobre CIFAR-10 y SVHN, DCGAN logró obtener resultados competitivos respecto a métodos clásicos de aprendizaje no supervisado.

Más aún, en DCGAN observaron que el espacio latente exhibe una estructura algebraica con efectos semánticos. Más precisamente, si $z_0, z_1 \in \mathbb{R}^L$ son dos muestras del prior $p_\theta(z)$, la *curva de imágenes* dada por $t \mapsto x_t = G_\theta((1-t)z_0 + tz_1)$, $t \in [0,1]$, corresponde a una **interpolación semántica** entre las imágenes $G_\theta(z_0) \in \mathbb{R}^D$ y $G_\theta(z_1) \in \mathbb{R}^D$, en la cual los atributos relevantes de la imagen varían suavemente.

Figura 13: Interpolación latente en DCGAN. Imagen obtenida desde [8].

En particular, esta propiedad del espacio latente de una GAN permite realizar aritmética vectorial en el espacio latente con efectos semánticos interpretables, de forma similar a como se pueden operar los vectores de embedding de un vocabulario de texto. Por ejemplo, si $\hat{z}_{\text{hombre, con lentes}}$, $\hat{z}_{\text{hombre, sin lentes}}$ y $\hat{z}_{\text{mujer, sin lentes}}$ son promedios de vectores latentes que generan imágenes con los atributos correspondientes, entonces el vector latente

$$\hat{z}_{\text{mujer, con lentes}} = \hat{z}_{\text{hombre, con lentes}} - \hat{z}_{\text{hombre, sin lentes}} + \hat{z}_{\text{mujer, sin lentes}}$$

(y una vecindad suya) genera imágenes de mujeres con lentes, sugiriendo que G_θ aprende representaciones con cierto grado de **linealidad semántica**.

Este tipo de modificación de atributos será implementado al estudiar VAEs, donde además se verá como aumentar y disminuir la intensidad del atributo a modificar.

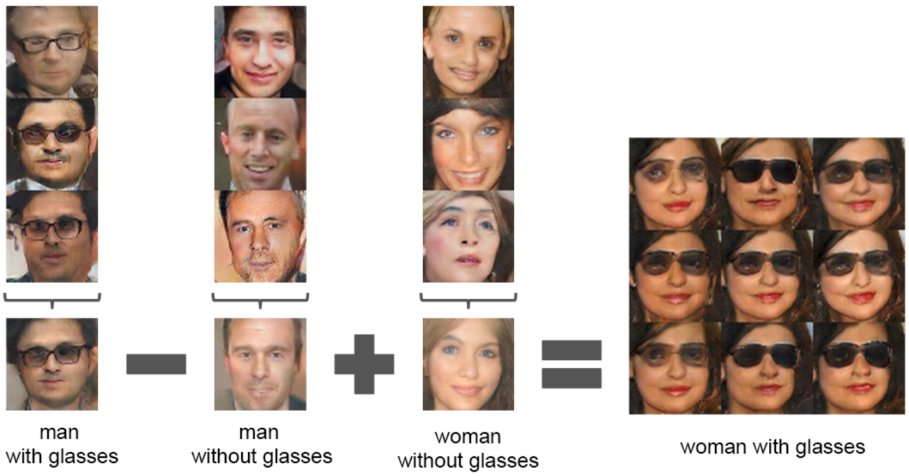

Figura 14: Aritmética vectorial en el espacio latente de DCGAN. Imagen obtenida desde [8].

7.3.2 Self-attention GAN

Si bien DCGAN produce resultados notables a 64×64, a mayor resolución presenta varias limitaciones que motivan las siguientes arquitecturas. En particular, las arquitecturas convolucionales clásicas presentan dificultades al modelar patrones con estructura geométrica compleja, donde las partes de un objeto deben mantener coherencia global (e.g., imágenes de animales). Esta limitación surge porque las convoluciones procesan información de manera local, restringiendo el modelado de dependencias de largo alcance y, si bien apilar varias capas convolucionales expande el campo receptivo, esto incrementa significativamente la profundidad y el costo computacional.

Self-attention GAN (SAGAN) [10] introduce un módulo de self-attention que captura dependencias entre cualquier par de posiciones del feature map con un único bloque, generando imágenes a 128×128 en la tarea de generación condicional (por clase) sobre ImageNet.

Módulo de self-attention

Para tratar un feature map convolucional como una secuencia, se puede considerar cada pixel como un token, donde la dimensión de embedding del token correspondería a la cantidad de canales. Si $x_1, ..., x_N \in \mathbb{R}^C$ son los $N = HW$ vectores asociados a las posiciones espaciales (con C canales), el módulo de self-attention en SAGAN parte realizando tres proyecciones lineales implementadas como convoluciones 1×1:

$$q_n = W_Q\, x_n, \quad k_n = W_K\, x_n, \quad v_n = W_V\, x_n, \quad \forall n \in \{1, ..., N\},$$

donde $W_Q, W_K, W_V \in \mathcal{M}_{\overline{C},C}(\mathbb{R})$ son matrices de parámetros y $\overline{C} \ll C$ es la dimensión del cabezal. La similitud entre las posiciones x_i y x_j se calcula mediante el producto interno $s_{ij} = \langle q_i, k_j \rangle$, y los pesos de atención se obtienen aplicando softmax por filas sobre la matriz de scores:

$$a_{ij} = \frac{e^{s_{ij}}}{\sum_{n=1}^{N} e^{s_{in}}}.$$

Luego vector de salida en la posición i agrega los valores ponderados y aplica una proyección de salida $W_O \in \mathcal{M}_{C,\overline{C}}(\mathbb{R})$ al único cabezal de atención para volver a la dimensión original:

$$o_i = W_O \sum_{j=1}^{N} a_{ij} v_j \in \mathbb{R}^C.$$

Finalmente, la salida se inyecta como una conexión residual con compuerta:

$$y_i = \gamma\, o_i + x_i.$$

El parámetro de compuerta, $\gamma \in \mathbb{R}$, es un escalar aprendible inicializado en cero. Con $\gamma = 0$, la salida del módulo de atención es $y_i = x_i$, por lo que la red se reduce a su contraparte puramente convolucional al inicio del entrenamiento. A medida que γ crece durante la optimización, el módulo incorpora información de largo alcance de forma gradual, evitando perturbaciones no locales que desestabilizarían el entrenamiento en las iteraciones iniciales.

Observación 2. A diferencia de la scaled dot-product attention de [11], SAGAN no divide los logits por $\sqrt{\overline{C}}$ (el control de magnitudes se delega a la normalización espectral[8]), y tampoco se usa layer normalization dentro del módulo.

Para elegir donde colocar los mecanismos de atención, notar que en resoluciones menores, el campo receptivo de la convolución ya cubre prácticamente todo el feature map, mientras que en resoluciones mayores el costo cuadrático de la atención se vuelve prohibitivo. Por este motivo, los autores de SAGAN sugieren insertar el módulo en el feature map de 64×64 ya que esta resolución ofrece la mejor relación costo-beneficio.

[8]Dada una capa lineal con matriz de pesos $W \in \mathcal{M}_{m,n}(\mathbb{R})$, la normalización espectral reemplaza W por $\frac{W}{\sigma_1(W)}$, donde $\sigma_1(W) = \max_{\|u\|=1} \|Wu\|$ es el mayor valor singular de W, el cual puede ser estimado eficientemente.

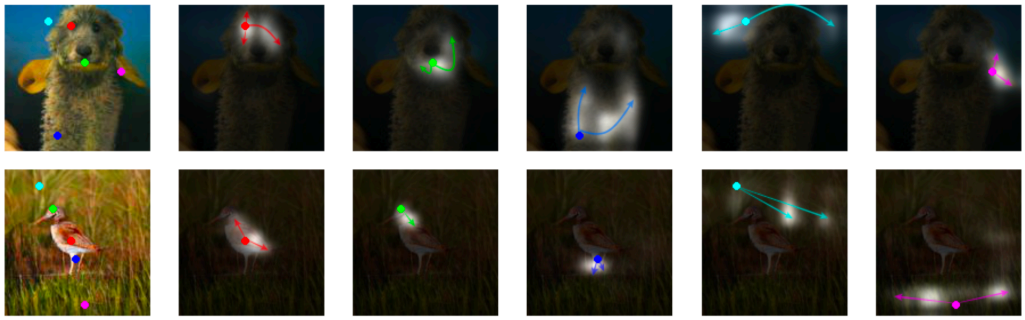

Figura 15: Visualización de los mapas de atención en SAGAN. Imagen obtenida desde [10].

Empíricamente, incorporar un mecanismo atención produce un aumento sustancial en la calidad de las imágenes generadas, especialmente en clases con patrones geométricos complejos. Debido a esto, el mecanismo de atención introducido por SAGAN ha influido notablemente en arquitecturas posteriores, como la U-Net usadas en modelos de difusión, donde se combinan bloques convolucionales con bloques de self-attention en las resoluciones intermedias siguiendo la idea de SAGAN.

Hinge aversarial loss

En la formulación estándar de una GAN, el discriminador produce una probabilidad $D_\phi(x) \in [0,1]$ y se entrena siguiendo el enfoque de máxima verosimilitud. En SAGAN, $D_\phi : \mathbb{R}^D \to \mathbb{R}$ produce un puntaje real sin restricción a $[0,1]$, y se entrena con la **hinge adversarial loss**:

$$\mathcal{L}_D(\phi) = -\mathbb{E}_{p_{\text{data}}(x)}\big[\min\big(0, -1 + D_\phi(x)\big)\big] - \mathbb{E}_{p(z)}\big[\min\big(0, -1 - D_\phi(G_\theta(z))\big)\big]$$
$$\mathcal{L}_G(\theta) = -\mathbb{E}_{p(z)}\big[D_\phi(G_\theta(z))\big].$$

Para muestras reales, el objetivo es $D_\phi(x) \geq 1$, y la penalización solo se activa cuando $D_\phi(x) < 1$. Para muestras sintéticas, el objetivo es $D_\phi(G_\theta(z)) \leq -1$, y la penalización solo se activa cuando $D_\phi(G_\theta(z)) > -1$. Notar que esta función de pérdida es análoga a la pérdida de margen de las SVMs, y evita que D_ϕ siga sobre-entrenando en ejemplos fáciles, lo que reduce la saturación de los gradientes que recibe G_θ.

7.3.3 Progressive GAN

SAGAN mejora la coherencia global de las muestras pero sigue limitado a 128×128, ya que el costo cuadrático de la atención y la dificultad de optimizar redes profundas a alta resolución impiden escalar la arquitectura directamente. **ProGAN** [12] fue el primer método en producir imágenes realistas a resolución 1024×1024. La idea central es un

curriculum learning[9] explícito, donde en lugar de entrenar los modelos directamente en la resolución final, G_θ y D_ϕ comienzan con arquitecturas simétricas que operan en resolución baja (4×4) y se van agregando pares de capas de manera progresiva hasta alcanzar la resolución objetivo.

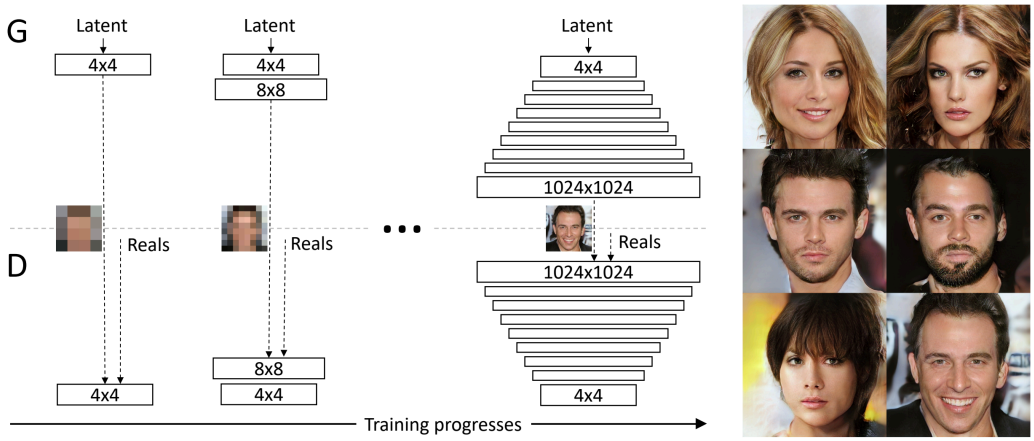

Figura 16: Arquitectura de ProGAN (izquierda) e imágenes generadas por el modelo (derecha).

Entrenamiento progresivo

Sean $G_\theta^{(r)}$ y $D_\phi^{(r)}$ las versiones de G_θ y D_ϕ a resolución $r \times r$. Al pasar de la resolución r a $2r$, ProGAN agrega un bloque convolucional en la parte superior de $G_\theta^{(r)}$ para realizar el upsampling $r \to 2r$ y, simétricamente, un bloque en la parte inferior de $D_\phi^{(r)}$ para realizar el downsampling $2r \to r$. Las imágenes reales de entrenamiento se redimensionan a la resolución actual y las nuevas capas se incorporan mediante un mecanismo de fade-in (explicado más abajo). Además, todas las capas existentes se siguen considerando entrenables al agregar las nuevas capas (esto permite interpretar el entrenamiento previo como una buena inicialización). Como resultado, los autores de ProGAN encontraron que, a igual presupuesto de cómputo, el entrenamiento progresivo es significativamente más rápido y estable que entrenar directamente en alta resolución.

Para cada nueva resolución $C_r \times r \times r$, al generador se le agrega una capa $\text{toRGB}_r : \mathbb{R}^{C_r \times r \times r} \to \mathbb{R}^{3 \times r \times r}$ al final, mientras que al discriminador se le agrega una capa $\text{fromRGB}_r : \mathbb{R}^{3 \times r \times r} \to \mathbb{R}^{C_r \times r \times r}$ al inicio. Estas capas cumplen la función de mapear entre el espacio de imágenes RGB y el espacio de features convolucionales, y viceversa. Ambas funciones son implementadas como convoluciones 1×1 que cambian la cantidad de canales sin alterar la resolución espacial.

[9] Esto es, un enfoque de aprendizaje progresivo donde la red se entrena primero en tareas más simples y luego se incrementa la dificultad.

Con respecto a la función objetivo, ProGAN es entrenado utilizando la pérdida WGAN-GP (estudiada en el próximo capítulo). También se utilizan otras heurísticas de entrenamiento y arquitectura, pero no se mencionarán ya que no son tan relevantes.

Fade-in de nuevas capas

Introducir nuevas capas de golpe altera considerablemente la función aprendida, lo que desestabiliza el entrenamiento. Para evitarlo, ProGAN utiliza una interpolación lineal controlada por un parámetro $\alpha \in [0, 1]$ que crece de 0 a 1 a lo largo de $N_{\text{fade-in}}$ steps de actualización. Más precisamente, durante la transición $r \to 2r$, la salida del generador es:

$$G_\theta^{(2r)}(z) = (1-\alpha)\,\text{toRGB}_{2r}(\text{Upsample}(h_r)) + \alpha\,\text{toRGB}_{2r}(\text{Conv}(\text{Upsample}(h_r))),$$

donde h_r es el último feature map de $G_\theta^{(r)}(z)$ a resolución r, Upsample es una interpolación determinista (e.g. nearest neighbor o bicúbica) y Conv es la nueva convolución que opera en resolución $2r \times 2r$. Simétricamente, durante la transición $2r \to r$, el discriminador se actualiza de la siguiente forma:

$$D_\phi^{(r)}(x_{2r}) = (1-\alpha)\,\text{fromRGB}_r(\text{Downsample}(x_{2r})) + \alpha\,\text{Conv}(\text{fromRGB}_{2r}(x_{2r})),$$

donde x_{2r} es la imagen real a resolución $2r$, Downsample es average pooling 2×2 y Conv es el nuevo bloque convolucional que pasa de resolución $2r \times 2r$ a $r \times r$. En $\alpha = 0$ se recuperan las arquitecturas de la resolución anterior y en $\alpha = 1$ las nuevas capas están completamente activas.

7.3.4 StyleGAN

Pendiente.

7.4 Arquitectura U-Net

Si bien las arquitecturas anteriores permiten generar imágenes en alta resolución, no es claro cómo utilizarlas de manera condicional cuando la condición c es una imagen (e.g., en tareas image-to-image). La arquitectura **U-Net** [13] es una red completamente convolucional con forma de **autoencoder**[10] que incluye **skip connections** entre los bloques del encoder y los bloques homólogos en el decoder. Por otro lado, si bien esta arquitectura fue propuesta originalmente para la tarea de **segmentación semántica** (donde cada píxel de una imagen

[10]Es decir, una red neuronal que primero comprime la información y luego la reconstruye. Estas redes serán estudiadas en más profundidad en el capítulo de autoencoders variacionales.

se clasifica individualmente), esta arquitectura es de gran importancia tanto en GANs como en modelos de difusión, donde la arquitectura debe recibir versiones más ruidosas de la imagen a generar, por lo que naturalmente se necesita una arquitectura tipo image-to--image.

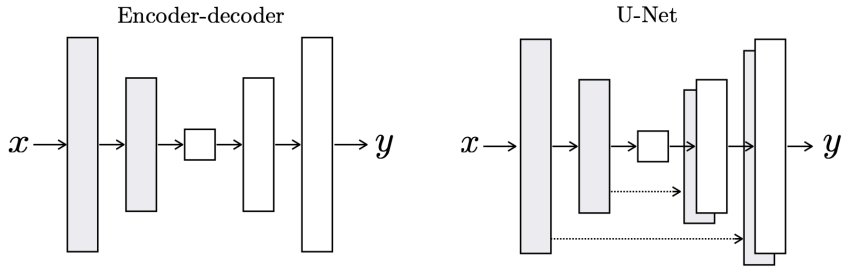

Figura 17: Comparación entre un autoencoder (izquierda) y la arquitectura U-Net (derecha). Imagen obtenida desde [14].

Por otro lado, hoy en día los modelos generativos de imágenes suelen utilizar arquitecturas tipo ViT (e.g. Diffusion Transformer), donde una imagen se *patchifica* para poder tratarla como una secuencia de tokens usando un Transformer.

El siguiente diagrama muestra una U-Net con dos bloques de bajada y dos bloques de subida. En cada bloque, la tupla (H, W, C) superior indica la resolución de entrada y la tupla inferior indica la resolución de salida del bloque:

```
graph LR

    Input["Input<br>(H, W, C)"] --> Phi0["Convolución<br>(H, W, C)<br>(H,
W, 64)"]

    Phi0 --> Down0["Downsample<br>(H, W, 64)<br>(H/2, W/2, 64)"]
    Down0 --> Phi1["Convolución<br>(H/2, W/2, 64)<br>(H/2, W/2, 128)"]
    Phi1 --> Down1["Downsample<br>(H/2, W/2, 128)<br>(H/4, W/4, 128)"]

    Down1 --> Phi2["Convolución<br>(H/4, W/4, 128)<br>(H/4, W/4, 256)"]

    Phi2 --> Up1["Upsample<br>(H/4, W/4, 256)<br>(H/2, W/2, 128)"]
    Up1 --> Psi1["Concat + Conv<br>(H/2, W/2, 256)<br>(H/2, W/2, 128)"]
    Psi1 --> Up0["Upsample<br>(H/2, W/2, 128)<br>(H, W, 64)"]
    Up0 --> Psi0["Concat + Conv<br>(H, W, 128)<br>(H, W, 64)"]

    Psi0 --> Out["Conv1×1<br>(H, W, 64)<br>(H, W, C)"]

    Phi0 -.- Psi0
```

```
    Phi1 -.- Psi1

    subgraph Encoder
        Phi0
        Down0
        Phi1
        Down1
    end

    subgraph Middle
        Phi2
    end

    subgraph Decoder
        Up1
        Psi1
        Up0
        Psi0
    end

    classDef default fill:#f9f9f9,stroke:#333,stroke-width:2px;
    classDef encoder fill:#fce4ec,stroke:#333,stroke-width:2px;
    classDef middle fill:#e3f2fd,stroke:#333,stroke-width:2px;
    classDef decoder fill:#fff8e1,stroke:#333,stroke-width:2px;

    class Phi0,Down0,Phi1,Down1 encoder;
    class Phi2 middle;
    class Up1,Psi1,Up0,Psi0 decoder;
    class Input,Out default;
```

Cada bloque de bajada del encoder se conecta a su bloque de subida homólogo en el decoder mediante una skip connection (concatenación en la dimensión de los canales). Esto facilita la transmisión directa de información de bajo nivel desde el encoder hacia el decoder, ayudando al modelo a conservar detalles espaciales finos. Las modificaciones modernas de esta arquitectura suelen incorporar bloques de self-attention en algunas resoluciones (similar a SAGAN) y mecanismos de atención cruzada para condicionamiento textual, lo cual será revisado al estudiar modelos de difusión.

7.4.1 Implementación

Dada la importancia de esta arquitectura neuronal para los modelos generativos, se entregará una implementación minimal de una U-Net, y luego se utilizará esta implementación

para entrenar un modelo de segmentación semántica sobre el dataset Oxford-IIIT Pet. Para simplificar la implementación, no se incluirán bloques de normalización ni dropout u otro tipo de regularizaciones. En el próximo capítulo se adaptará esta arquitectura para ser utiliza en modelo Pix2Pix (GAN para *traducción de imágenes*), y luego se adaptará con bloques de self-attention al implementar un modelo de difusión para imágenes.

La U-Net es una arquitectura completamente convolucional, por lo que se comenzará definiendo un módulo `ConvBlock`, formado por dos convoluciones 3×3 con $S = P = 1$ (i.e., no cambia la resolución). En caso de querer cambiar la cantidad de canales, esto se hará en la primera convolución[11].

```
class ConvBlock(nn.Module):

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(out_channels, out_channels, kernel_size=3,
padding=1),
            nn.ReLU(),
        )

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

Cada bloque de bajada del encoder, implementado en `DownBlock`, está compuesto por una `ConvBlock` (que duplica los canales sin cambiar la resolución) seguida de una operación de pooling (que divide en dos la resolución sin cambiar los canales). En consecuencia, cada `DownBlock` duplica la cantidad de canales y reduce la resolución a la mitad. Este bloque retorna, además, la salida de la convolución (antes del pooling) para que sea usada en la skip connection con el decoder en la resolución homóloga respectiva.

```
class DownBlock(nn.Module):

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = ConvBlock(in_channels, out_channels)
        self.down = nn.MaxPool2d(kernel_size=2)

    def forward(self, x):
```

[11]Esto es una decisión de diseño. De forma relacionada, el paper original de U-Net no incluye padding, por lo que las resoluciones van disminuyendo levemente. Aquí se seguirá la elección natural de usar $P = 1$.

```
        skip = self.conv(x)
        return self.down(skip), skip
```

Por otro lado, cada bloque de subida del decoder, implementado en `UpBlock`, parte duplicando la resolución y dividiendo los canales por dos usando una convolución transpuesta[12]. Luego, realiza la skip conection, concatenando lo anterior con el bloque auxiliar homólogo del encoder. Finalmente, se aplica una `ConvBlock` para ajustar a la cantidad de canales de salida deseada. En consecuencia, cada `UpBlock` reduce a la mitad la cantidad de canales y duplica la resolución.

```
class UpBlock(nn.Module):

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.up = nn.ConvTranspose2d(in_channels, in_channels // 2,
kernel_size=2, stride=2)
        self.conv = ConvBlock(in_channels, out_channels)

    def forward(self, x, skip):
        x = self.up(x)
        x = torch.cat([x, skip], dim=1)
        return self.conv(x)
```

Con estos bloques, la U-Net completa se construye de forma simétrica. Los bloques del decoder reciben las skip connections del encoder en orden inverso, y la salida del último bloque del decoder pasa por una convolución 1×1 para ajustar la cantidad de canales a la salida deseada (`n_classes` en la tarea de segmentación semántica, o 3 en la tarea de generación de imágenes).

```
class UNet(nn.Module):

    def __init__(self, in_channels, n_classes, base_ch=64):
        super().__init__()

        self.down1 = DownBlock(in_channels, base_ch)
        self.down2 = DownBlock(base_ch, base_ch * 2)
        self.down3 = DownBlock(base_ch * 2, base_ch * 4)
        self.down4 = DownBlock(base_ch * 4, base_ch * 8)

        self.bottleneck = ConvBlock(base_ch * 8, base_ch * 16)

        self.up4 = UpBlock(base_ch * 16, base_ch * 8)
```

[12]Esto se realiza para que, al concatenar con la skip connection del encoder (el cual tiene la misma resolución), la cantidad de canales coincida.

```
        self.up3 = UpBlock(base_ch * 8, base_ch * 4)
        self.up2 = UpBlock(base_ch * 4, base_ch * 2)
        self.up1 = UpBlock(base_ch * 2, base_ch)

        self.out = nn.Conv2d(base_ch, n_classes, kernel_size=1)

    def forward(self, x):
        x, skip1 = self.down1(x)
        x, skip2 = self.down2(x)
        x, skip3 = self.down3(x)
        x, skip4 = self.down4(x)

        x = self.bottleneck(x)

        x = self.up4(x, skip4)
        x = self.up3(x, skip3)
        x = self.up2(x, skip2)
        x = self.up1(x, skip1)

        return self.out(x)
```

7.4.2 Aplicación a segmentación semántica

Si bien esta arquitectura será utilizada en modelos generativos, U-Net fue propuesta originalmente para la tarea de segmentación semántica en imágenes médicas, donde se busca clasificar individualmente cada pixel de una imagen para construir un mapa de segmentación (máscara) y poder identificar diferentes estructuras o regiones de interés. Aquí se entrenará el modelo anterior sobre el dataset Oxford-IIIT Pet, el cual contiene imágenes de mascotas junto a sus mapas de segmentación (fondo, mascota y borde).

Para comenzar, todas las imágenes y máscaras se redimensionarán a resolución 256×256. Notar que las máscaras se deben interpolar utilizando métodos como `InterpolationMode.NEAREST` para evitar valores intermedios que no corresponderían a etiquetas válidas (fuera de $\{0, 1, 2\}$):

```
img_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
])

mask_transform = transforms.Compose([
    transforms.Resize((256, 256),
interpolation=transforms.InterpolationMode.NEAREST),
```

```
    transforms.PILToTensor(),
])

def transform(img, mask):
    img = img_transform(img)
    mask = mask_transform(mask).long().squeeze()
    return img, (mask - 1)

dataset = datasets.OxfordIIITPet(
    root='data', download=True, target_types='segmentation',
transforms=transform,
)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True,
drop_last=True)
```

Figura 18: Ejemplo de imagen y máscara de segmentación del dataset Oxford-IIIT Pet.

Para entrenar el modelo se seguirá el enfoque de máxima verosimilitud del clasificador minimizando la entropía cruzada, aplicada a nivel de pixel, donde la función de pérdida total es el promedio sobre todos los píxeles de las pérdidas individuales.

```
def train(model, optimizer, dataloader, n_epochs):

    model.to(DEVICE)
    model.train()

    loss_fn = nn.CrossEntropyLoss()
    losses = []

    try:
        for epoch in range(n_epochs):
            pbar = tqdm(dataloader, desc=f'Época {epoch+1}/{n_epochs}')
            for imgs, masks in pbar:
                imgs, masks = imgs.to(DEVICE), masks.to(DEVICE)
```

```
                logits = model(imgs)
                loss = loss_fn(logits, masks)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                pbar.set_postfix(loss=loss.item())
                losses.append(loss.item())

    except KeyboardInterrupt:
        print('Entrenamiento interrumpido.')
        training_log = {'model': model.state_dict(), 'losses': losses}
        torch.save(training_log, 'training.pt')
```

Finalmente, se entrenará el modelo y se graficará su dinámica de entrenamiento.

```
# Entrenamiento:
model = UNet(in_channels=3, n_classes=3)
optimizer = optim.Adam(model.parameters(), lr=1e-4)
train(model, optimizer, dataloader, n_epochs=50)

# Carga del modelo entrenado:
training_log = torch.load('training.pt', map_location=DEVICE,
weights_only=True)
model.load_state_dict(training_log['model'])
```

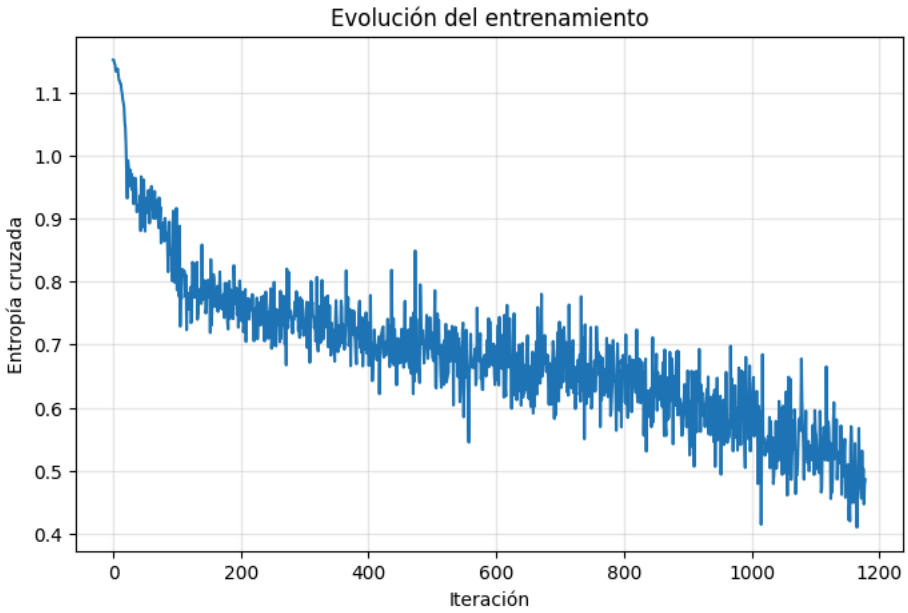

Figura 19: Curva de pérdida durante el entrenamiento de la U-Net para la tarea de segmentación.

Una vez entrenado el modelo, la predicción de la máscara para una imagen dada se obtiene aplicando arg max sobre los logits de salida en la dimensión de las clases.

```
def predict_mask(model, img):
    model.eval()
    with torch.no_grad():
        img = img.unsqueeze(0).to(DEVICE)
        logits = model(img)
        probs = torch.softmax(logits, dim=1)
        pred_mask = torch.argmax(probs, dim=1)
    return pred_mask.squeeze().cpu()
```

Figura 20: Imagen original, máscara real y predicción del modelo entrenado.

Referencias

[1] 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

[2] S. Mohamed y B. Lakshminarayanan, «Learning in Implicit Generative Models», en *arXiv preprint arXiv:1610.03483*, 2016. [En línea]. Disponible en: https://arxiv.org/abs/1610.03483

[3] Google Developers, «Overview of GAN Structure». [En línea]. Disponible en: https://developers.google.com/machine-learning/gan/gan_structure

[4] scikit-learn developers, «sklearn.datasets.make_swiss_roll». [En línea]. Disponible en: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_swiss_roll.html

[5] 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

[6] M. Mirza y S. Osindero, «Conditional Generative Adversarial Nets», en *arXiv preprint arXiv:1411.1784*, 2014. [En línea]. Disponible en: https://arxiv.org/abs/1411.1784

[7] 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

[8] A. Radford, L. Metz, y S. Chintala, «Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks», *arXiv preprint arXiv:1511.06434*, 2015, [En línea]. Disponible en: https://arxiv.org/abs/1511.06434

[9] S. Ioffe y C. Szegedy, «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift», en *International Conference on Machine Learning (ICML)*, 2015. [En línea]. Disponible en: https://arxiv.org/abs/1502.03167

[10] H. Zhang, I. Goodfellow, D. Metaxas, y A. Odena, «Self-Attention Generative Adversarial Networks», en *International Conference on Machine Learning (ICML)*, 2019. [En línea]. Disponible en: https://arxiv.org/abs/1805.08318

[11] 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

[12] T. Karras, T. Aila, S. Laine, y J. Lehtinen, «Progressive Growing of GANs for Improved Quality, Stability, and Variation», en *International Conference on Learning Representations (ICLR)*, 2018. [En línea]. Disponible en: https://arxiv.org/abs/1710.10196

[13] O. Ronneberger, P. Fischer, y T. Brox, «U-Net: Convolutional Networks for Biomedical Image Segmentation», en *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2015. [En línea]. Disponible en: https://arxiv.org/abs/1505.04597

[14] P. Isola, J.-Y. Zhu, T. Zhou, y A. A. Efros, «Image-to-Image Translation with Conditional Adversarial Networks», en *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. [En línea]. Disponible en: https://arxiv.org/abs/1611.07004