Universidad Europea

Generative Adversarial Networks (GANs)

Lapland University of Applied Sciences

Dr. Manuel García Fernández
manuel.garcia2@universidadeuropea.es

Ve más allá

Index

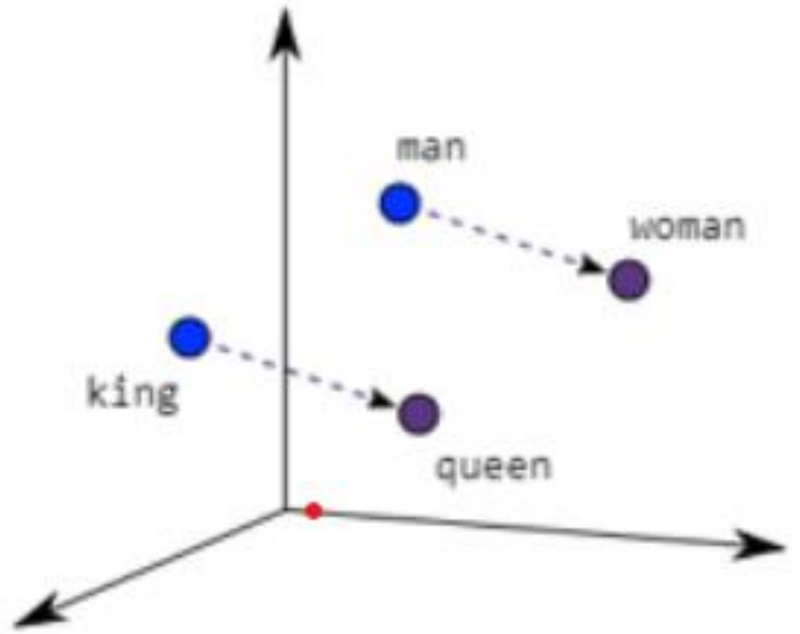
01

Fundamentals of Representation Learning

Fundamentals of Representation Learning

Representation learning is a foundational field within machine learning that focuses on automatically discovering and encoding meaningful features or abstractions from raw data to facilitate downstream tasks such as classification, prediction, or generation.

At the heart of this process lies the concept of a **latent space, which is a lower-dimensional, often continuous, embedding of the original high-dimensional data**. This latent space serves as the compressed form that captures the essential properties, underlying structure, and semantics of the data, making it central to how representation learning functions.



E.g.: King-queen are represented along the same axis. Man/King are represented at the same side as they refer both to the same gender

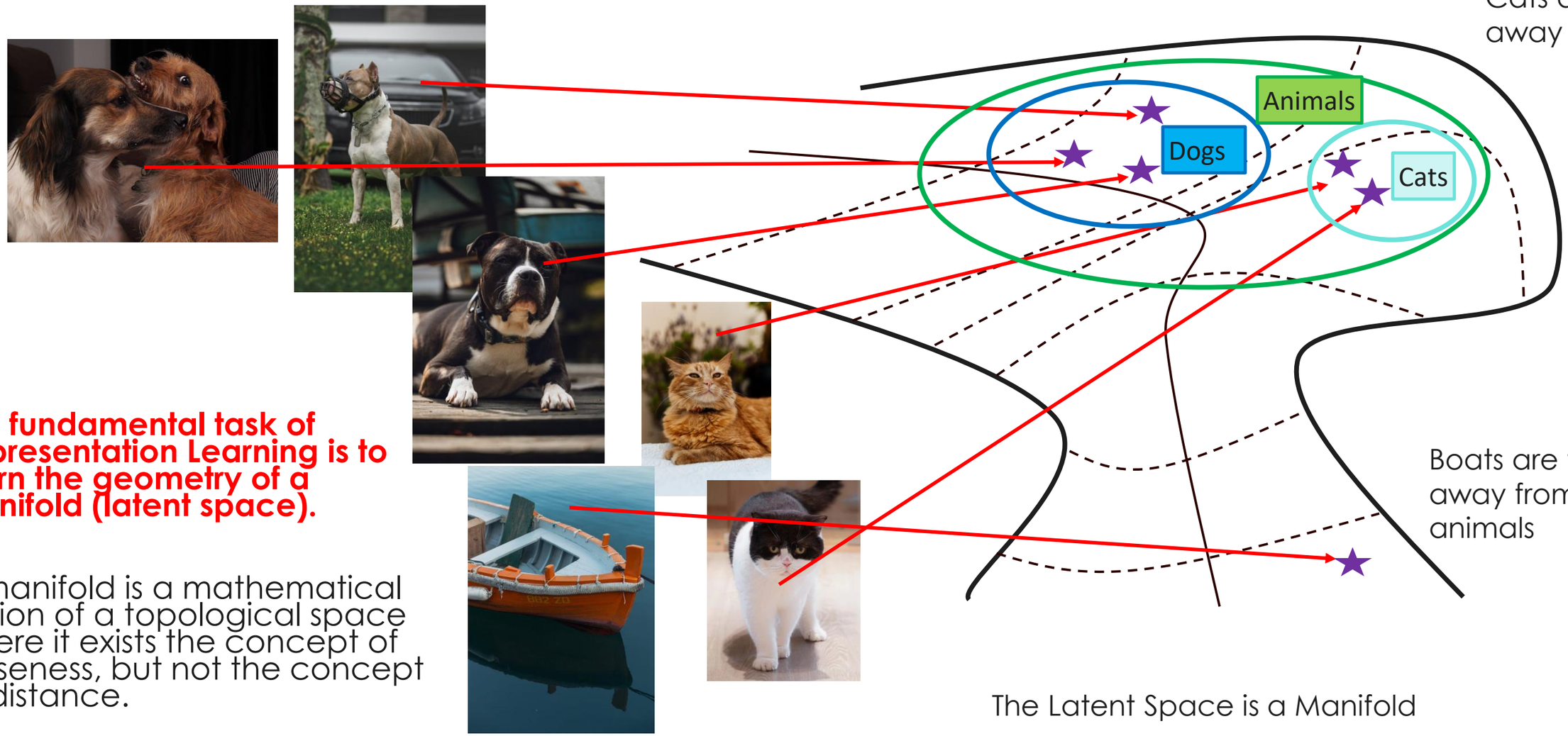
Fundamentals of Representation Learning

The primary objective of representation learning is to transform complex, high-dimensional data into a more compact, robust, and semantically meaningful representation. This transformation significantly reduces the input space's redundancy, benefiting various pattern recognition tasks. The latent space acts as this reduced-dimensionality vector space where the complex data is embedded, often capturing intricate semantic properties.

Latent spaces inherently perform dimensionality reduction by mapping high-dimensional input data into a lower-dimensional manifold. This process is crucial for managing computational complexity and focusing on the most salient features of the data. For instance, network representation learning embeds network vertices into a low-dimensional vector space, preserving network topology and other side information, thereby simplifying analysis. In complex scenarios like analyzing unprocessed images with thousands of pixels, latent variable models can effectively capture underlying structure. The representations learned in the latent space are designed to be more compact and robust, making them ideal for subsequent analysis.

A well-constructed latent space organizes data in a way that reflects meaningful attributes and preserves intrinsic geometric properties. It allows for the discovery of interpretable representations where different latent dimensions might correspond to distinct semantic features or generative factors

Fundamentals of Representation Learning



The fundamental task of Representation Learning is to learn the geometry of a manifold (latent space).

A manifold is a mathematical notion of a topological space where it exists the concept of closeness, but not the concept of distance.

Fundamentals of Representation Learning

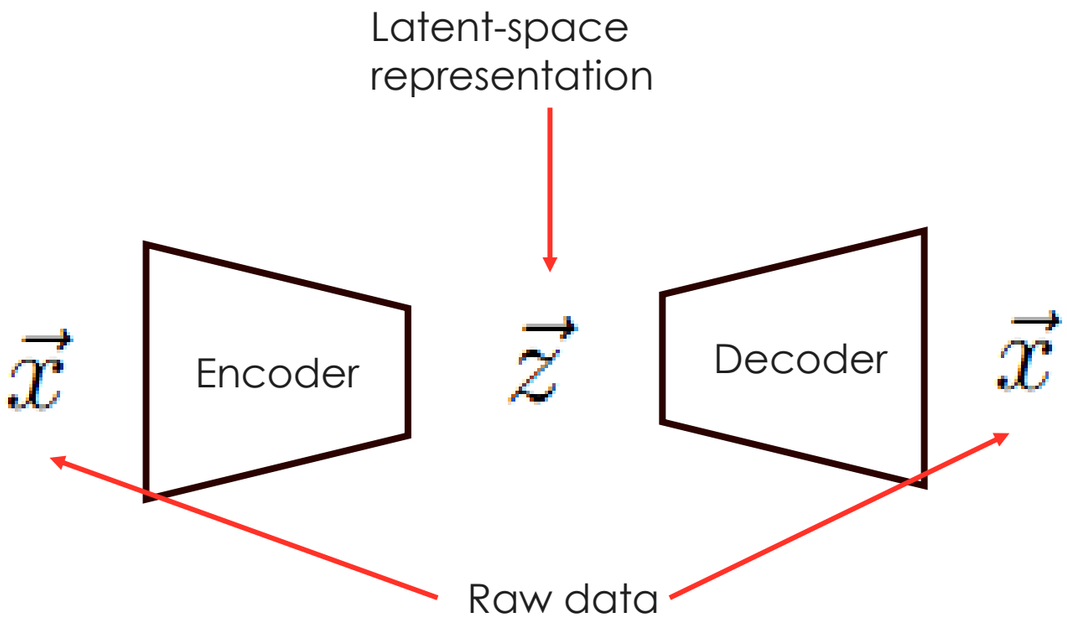
For leveraging Representation Learning and allowing a proper definition of the geometry of manifold, one has to learn the mapping between the raw high-dimensional space and the compressed latent-space.

Different approaches can be used. The simplest approach that uses Neural Networks is the Autoencoder.

An autoencoder is a set of two Neural Networks:

- An Encoder: which compresses data from an N-dimensional space to a M-dimensional space (N<M)
- A Decoder: which decompresses back from the M-dimensional space.

Typically, target goal of an Autoencoder seeks to minimize the reconstruction error, so Mean Squared Error (MSE) is used as loss function to be minimized.



$$\mathcal{L}_{AE} = \frac{1}{N}\sum_{i=1}^{N} ||\vec{x} - D(E(\vec{x}))||^2$$

Fundamentals of Representation Learning

On the Autoencoder, it was learned a invertible map from the data-space to the latent space.

- Encoder is a direct map from the real-space to the latent-space.
- Decoder is an invers map from the latent-space to the real-space.

Autoencoders are very useful for compressing data as they have are an invertible relationship. In addition, as the reconstruction error can be determined, this can be used for other applications, such as anomaly detection.

Autoencoders can be used as synthetic data generators by just by samling on the latent space with random noise and decode it with the Decoder.

Nevertheless, synthetic data generated by Autoencoders is not good, as the induced geometry of the manifold is not very accurate, as we are limited by the training data instances and interpolation does not make any sense, as close points not necessarily are similar.

As we only measure the geometry of the manifold at real training data instances, the sampling of the latent-space is very sparse, being bad at non-seen data instances.

Fundamentals of Representation Learning

Encoder and Decoder are Feedforward neural networks. They are symmetric and read MNIST images flattend (28x28=784)

```python
class Autoencoder(nn.Module):
    def __init__(self, latent_dim=128):
        super(Autoencoder, self).__init__()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Linear(784, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, latent_dim)
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 784),
            nn.Sigmoid()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

    def encode(self, x):
        return self.encoder(x)

def train_autoencoder():
    # Data loading
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,))
    ])

    train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
    train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)

    # Model setup
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = Autoencoder(latent_dim=16).to(device)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-4)

    # Training
    model.train()
    loss_history = []

    for epoch in range(300):
        total_loss = 0
        for data, _ in train_loader:
            data = data.view(-1, 784).to(device)

            optimizer.zero_grad()
            reconstructed = model(data)
            loss = criterion(reconstructed, data)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
        loss_history.append(avg_loss)
        print(f'Epoch {epoch+1}/300, Loss: {avg_loss:.4f}')

    return model, loss_history
```

The output of the Encoder is the input of the Decoder. The number of neurons of the output layer is the dimension number of the latent space

Loss function is Mean Square Error (MSE)

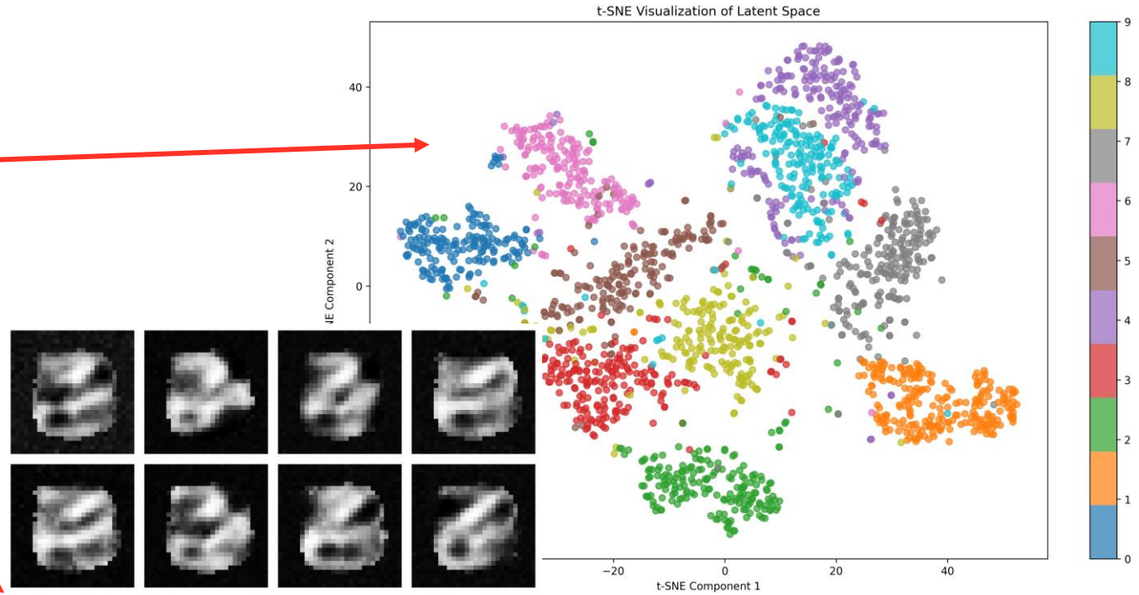
Fundamentals of Representation Learning

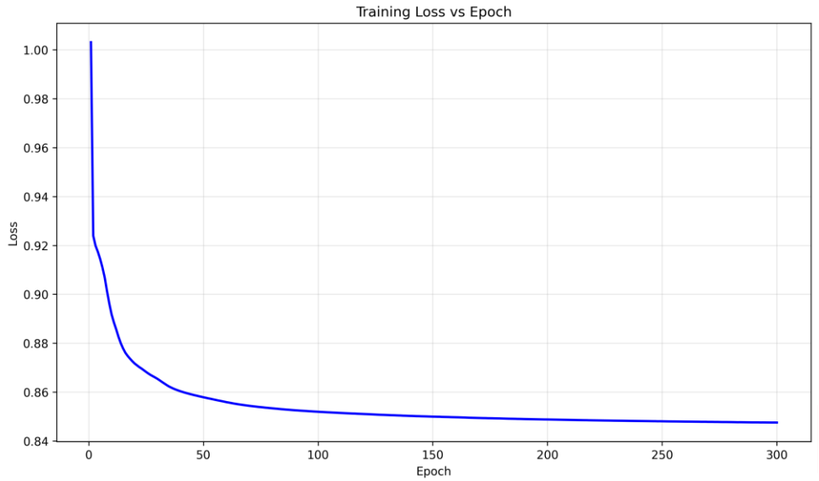
Although using a t-SNE visualization of the latent-space displays a fairly reasonable separation of digits for the training dataset.

The synthetic dataset displays poor images

The learned latent space by an Autoencoder has the following properties:

- Latent vectors are all deterministic (1-to-1 relationship).
- No guarantee of continuity or smoothnes.
- No guarantee simmilar points lie closer.





Fundamentals of Representation Learning

An improvement can be made at standard Autoencoder with the introduction of the Variational Autoencoder (VAE).

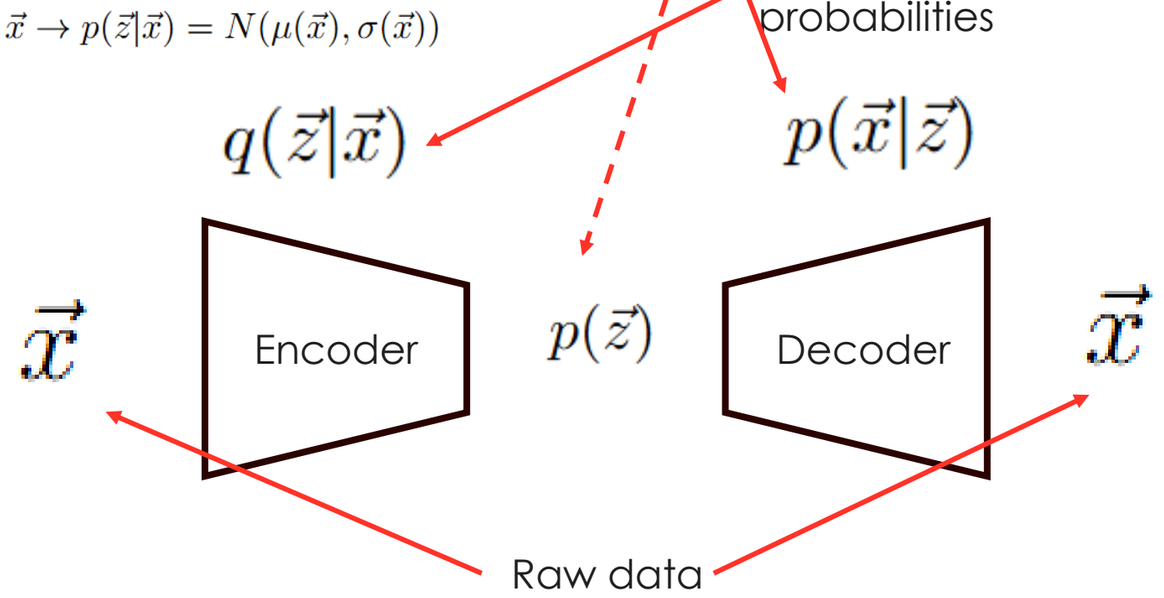
A Variational Autoencoder does not map raw point to latent-space points, but as probability distributions at the latent-space.

Probability distribution can be anything, but normally is assumed to be a Gaussian. Thus, we aim to learn the mean and variance of that Gaussian.

To enforce the learning of the probability distribution, we demand to minimize the reconstruction loss, but also that probability distributions ressemble each other (with KL-divergence).

Latent-space representation now induces a probability distribution.

Encoder and Decoder now sample conditional probabilities



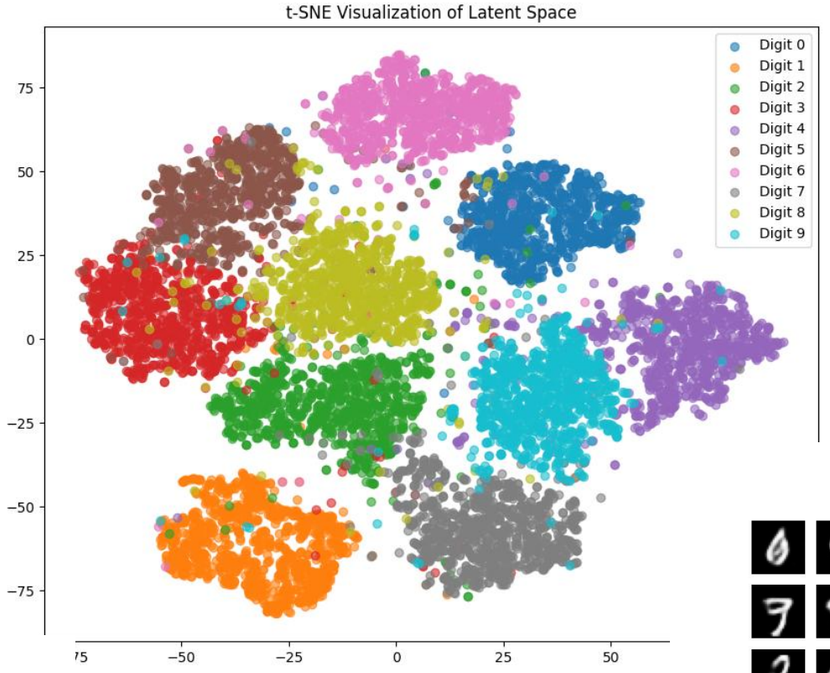
$$\mathcal{L}_{VAE} = \mathcal{L}_{AE} - D_{KL}(q(\vec{z}|\vec{x}), p(\vec{z})) = \frac{1}{N}\sum_{i=1}^{N} ||\vec{x} - D(E(\vec{x}))||^2 - \frac{1}{2}\sum_{i=1}^{N}(\sigma_i^2 + \mu_i^2 - 1 - \log \sigma_i^2)$$

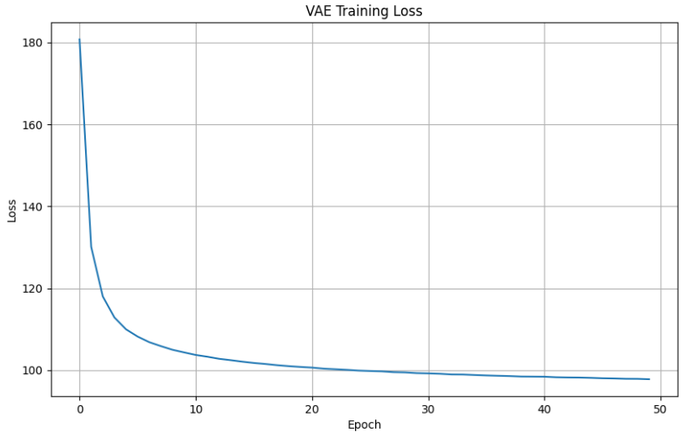
KL-divergence of two gaussians

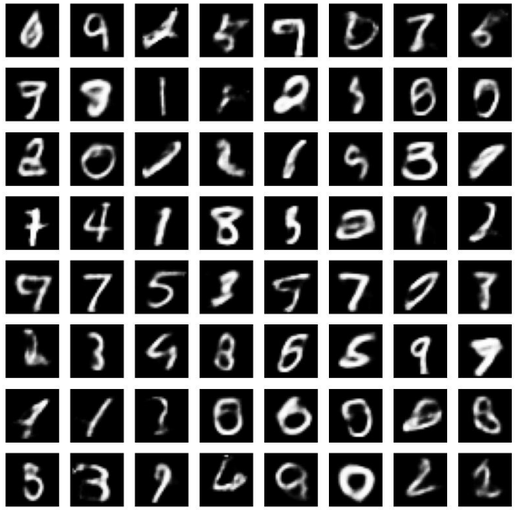
Fundamentals of Representation Learning

The learned manifold contrary to the Autoencoder has:

- Smoothnes. The transition between different datapoints is not abrupt as in AE.
- Closeness as a meaning (close points mean similar things).
- It is a continuous manifold (no holes).
- The mapping between real data and latent space is probabilistic.
- **Smoothnes ensures that interpolation has a meaning: this implies that unseen regions of the manifold mean something. Thus it is a truly generative model.**







02

Introduction to GANs

Introduction to GANs

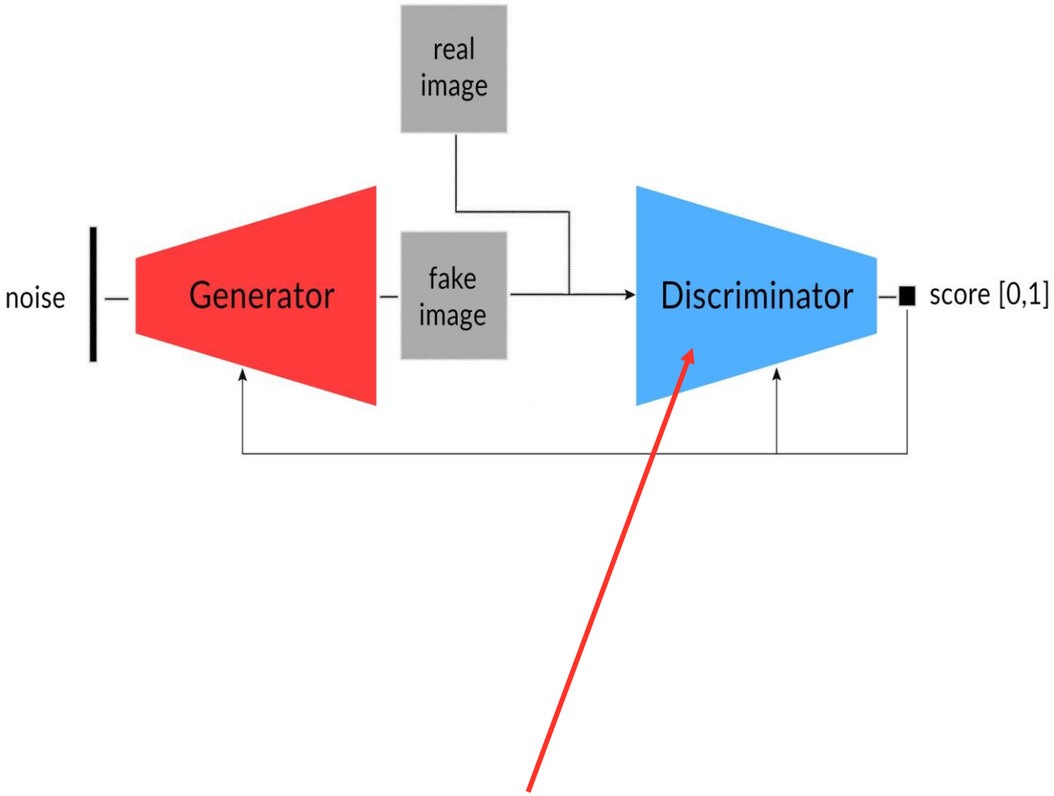
Generative Adversarial Networks (GANs) are set of two networks competing one with the other one.

It is composed by a Generator network and Discriminator network.

- Generator: it is a network that from pure random noise produces synthetic data instances.
- Discriminator: It is a network that tries to guess if a given data is real (from the training dataset) or is synthetic (generated by the Generator networ).

Traning process is such as Generator and Discriminator compete with each other in a competitive game, where we aim Generator win and generate so Good data instances that it fools the Discriminator.

Training process is such that only the Discriminator sees the real training data. Generator learns as the Discriminator "teaches" that network.



Discriminator sees synthetic and real data.
Discriminator can learn to differentiate both, as we at the training process know which are the real data and which from the Generator

Introduction to GANs

On the GAN, only the inverse map from the latent-space to the real-space is learned (which is the Generator).

The induced geometry of a GAN on the manifold, is much more accurate due to the sampling process. At the sampling of the latent-space, we generate random noise, each of which Will correspond to a point of the manifold. Thus, if we generate a large enough sample of random noise, we can reach virtually any desired accuracy on the geometry of the manifold.

Nevertheless, practical issues happen at training: as is a competitive game between two neural networks, convergence of the solution is not always stable and is very complex.

Introduction to GANs

In practice GANs produce better synthetic data than VAEs because:

- Generator is not rewarded to cover the full data distribution, rather for producing images that lie on the manifold of realistic images.
- Producing accurate data comes at expenses of ignoring very unlikely events.
- Contrary VAEs are trained to explain all posible variations of data. Thus, when multimodality happens, the safest bet is to average modes (e.g. a person with eyes looking at the left or looking at right it appear with blurry eyes, becase VAE is averaging left-looking and right-looking).
- VAEs are rewarded for covering all modes of the distribution, GANs are rewared for covering very well any of the modes (even if not covering some of them). **This is the problem of mode-collapse of GANs**.

Currently, state-of-the art of generative models for general image generation include Diffusion models, which combine advantages of GANs and VAEs: training stability, full probability density coverage while producing realistic data.

Introduction to GANs

The simplest GAN we can make is the Vanilla-GAN.

The Vanilla-GAN constitutes two feedforward networks: one for the Generator and another for the Discriminator.

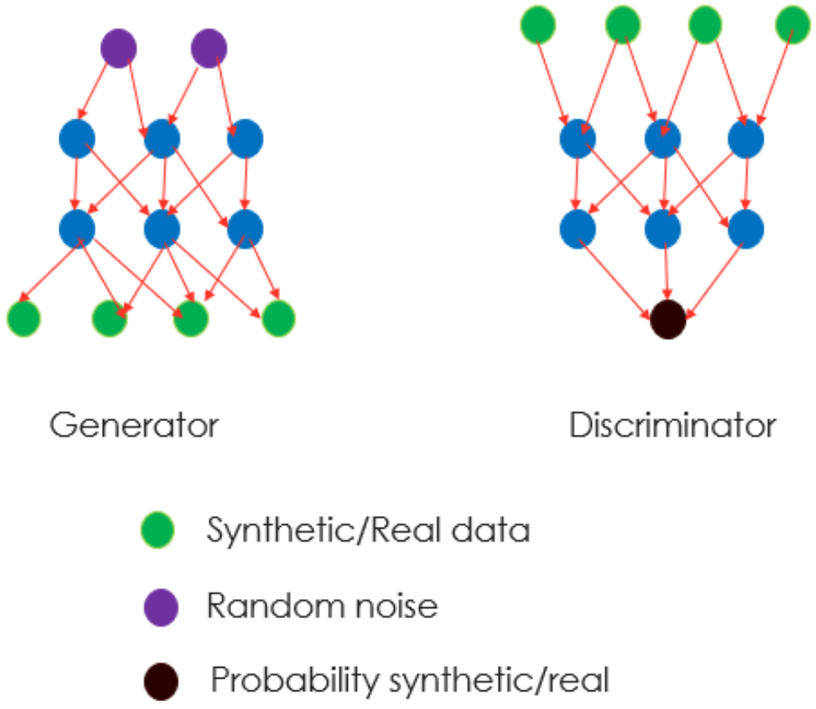
Each network Will have its own loss function.

Generator

$$\mathcal{L}_G = \frac{1}{m}\sum_{i=1}^{m}\log(1 - D(G(z_i)))$$

Discriminator

$$\mathcal{L}_D = \frac{1}{m}\sum_{i=1}^{m}[\log D(x_i)+\log(1-D(G(z_i)))]$$

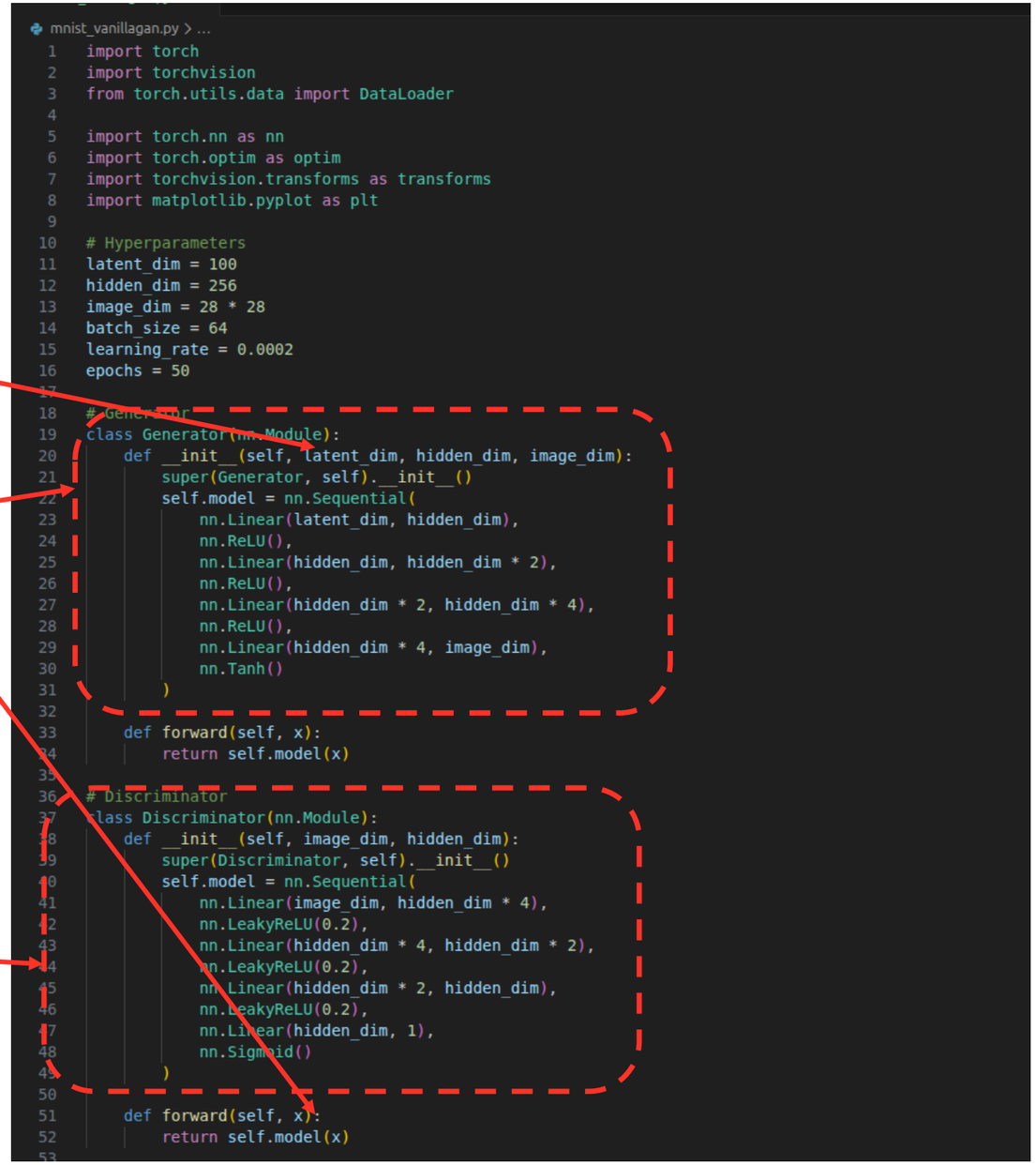


Introduction to GANs

Latent dimension is the number of neurons of the input layer of the Generator

Generator is a feedforward neural network with ReLU activation layers.

Discriminator is a feedforward neural network with LeakyReLU activation layers. LeakyReLU are needed because Generator learns through the Discriminator. Negative part of ReLU has no gradient, complicating the learning of the Generator



Introduction to GANs

Real and fake labels are generated as tensors with ones an zeros

We first show to the discriminator real data and compute the loss

Then we generate a fake image from randon noise

We show the fake images to the discriminator and compute the loss

Apply the backpropagation and advance the optimizer

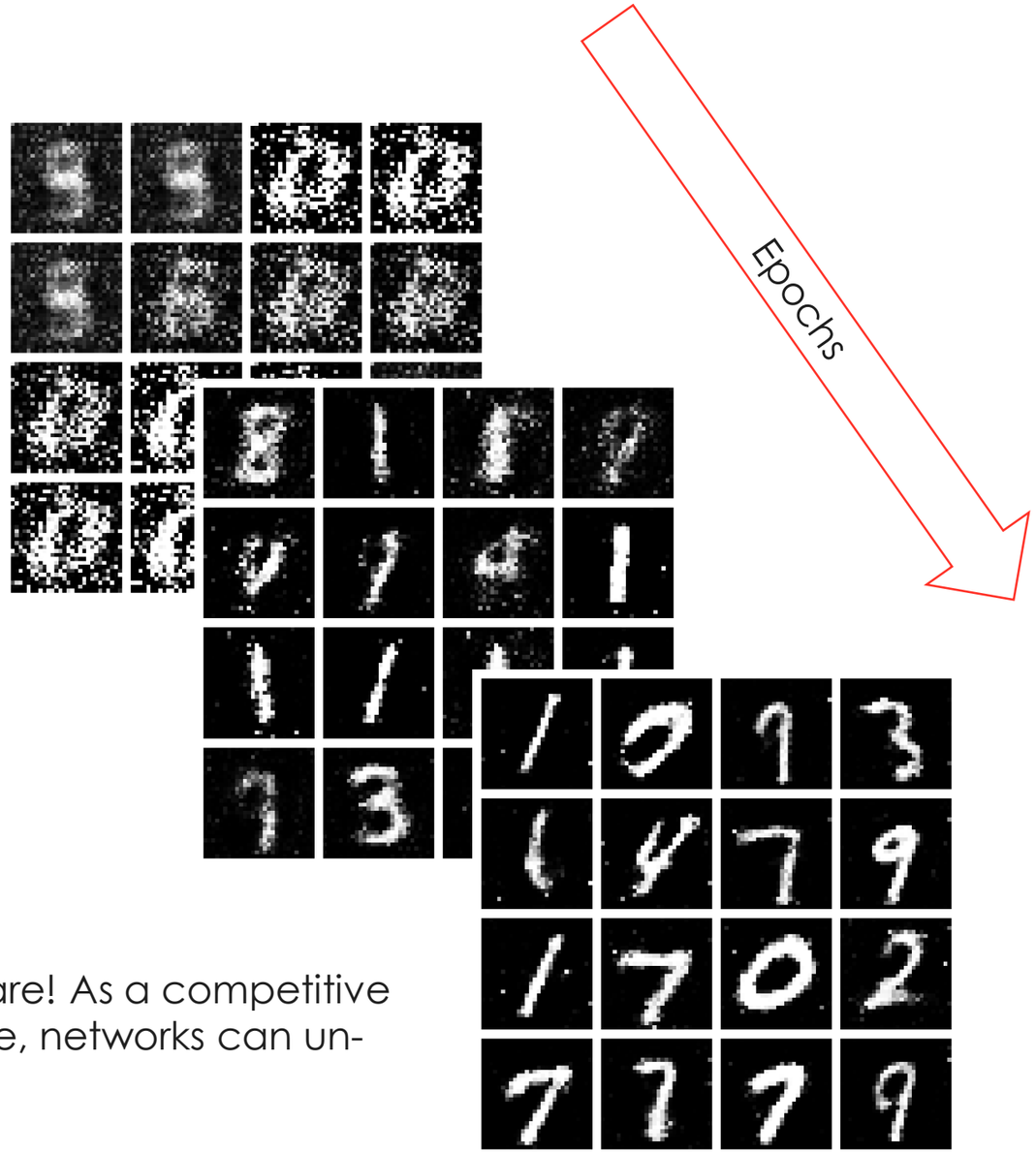
Now we generate a new fake image and compute the loss, but with “real” labels (fooling the discriminator).

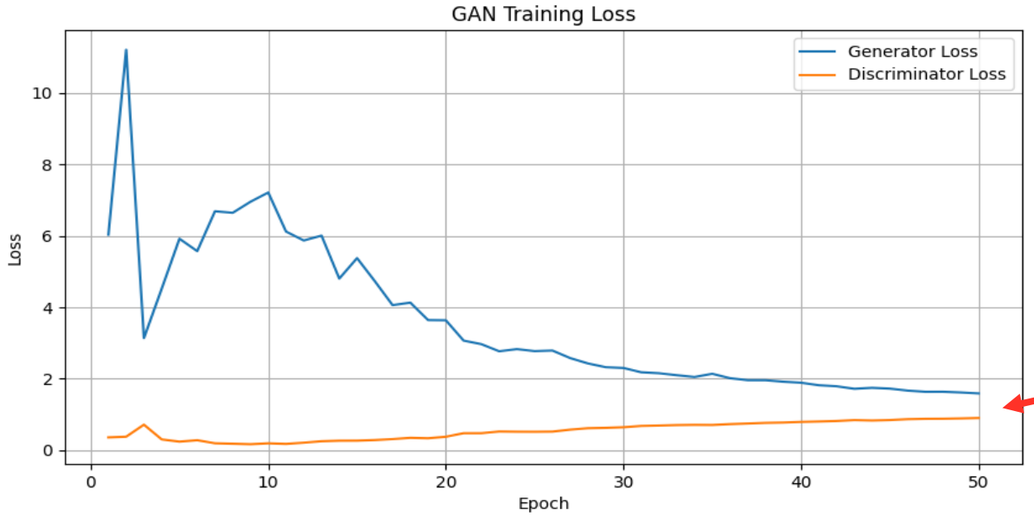
```python
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Initialize models
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
generator = Generator(latent_dim, hidden_dim, image_dim).to(device)
discriminator = Discriminator(image_dim, hidden_dim).to(device)

# Loss and optimizers
criterion = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=learning_rate)
optimizer_D = optim.Adam(discriminator.parameters(), lr=learning_rate)

# Lists to store losses
g_losses = []
d_losses = []

# Training loop
for epoch in range(epochs):
    epoch_g_loss = 0
    epoch_d_loss = 0
    num_batches = len(dataloader)

    for batch_idx, (real_images, _) in enumerate(dataloader):
        real_images = real_images.view(-1, image_dim).to(device)
        batch_size = real_images.size(0)

        # Train Discriminator
        real_labels = torch.ones(batch_size, 1).to(device)
        fake_labels = torch.zeros(batch_size, 1).to(device)

        # Real images
        outputs = discriminator(real_images)
        d_loss_real = criterion(outputs, real_labels)

        # Fake images
        z = torch.randn(batch_size, latent_dim).to(device)
        fake_images = generator(z)
        outputs = discriminator(fake_images.detach())
        d_loss_fake = criterion(outputs, fake_labels)

        d_loss = d_loss_real + d_loss_fake
        optimizer_D.zero_grad()
        d_loss.backward()
        optimizer_D.step()

        # Train Generator
        z = torch.randn(batch_size, latent_dim).to(device)
        fake_images = generator(z)
        outputs = discriminator(fake_images)
        g_loss = criterion(outputs, real_labels)

        optimizer_G.zero_grad()
        g_loss.backward()
        optimizer_G.step()

        epoch_g_loss += g_loss.item()
        epoch_d_loss += d_loss.item()

    # Calculate average losses for the epoch
    avg_g_loss = epoch_g_loss / num_batches
    avg_d_loss = epoch_d_loss / num_batches

    g_losses.append(avg_g_loss)
    d_losses.append(avg_d_loss)

    print(f'Epoch [{epoch+1}/{epochs}], d_loss: {avg_d_loss:.4f}, g_loss: {avg_g_loss:.4f}')
```

Introduction to GANs

Epoch by epoch generated images are more close to the real ones.

We do not have control over which number Will be generated





Beware! As a competitive game, networks can un-learn

03

Flavours of GANs

Flavours of GANs

But what is the target mathematical goal of a GAN?

We aim to follow the min-max $\min_{\theta_G} \max_{\theta_D} V(D,G)$ where V can be selected as the most general form as

$$V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_d(\mathbf{x})}[\mathbb{E}_{y\sim p_d(y)}[g_f(D(y|\mathbf{x}))] + \mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})}[-f^*(g_f(D(G(\mathbf{z}|\mathbf{x})|\mathbf{x})))]],$$

Where g and f are one the Frenchel conjugate of the other. They can be derived freely provided they are derived from a f-divergence. Thus, losses are:

$$\mathcal{L}_G(\theta_G) = \frac{-1}{N_{batch}} \sum_{i=1}^{N_{batch}} g_f(D(G(\mathbf{z}_i|\mathbf{x}_i;\theta_G)|\mathbf{x}_i;\theta_D)).$$

$$\mathcal{L}_D(\theta_D) = \frac{-1}{N_{batch}} \sum_{i=1}^{N_{batch}} [g_f(D(y_i|\mathbf{x}_i;\theta_D)) - f^*(g_f(D(G(\mathbf{z}_i|\mathbf{x}i);\theta_G)|\mathbf{x}i;\theta_D))],$$

Flavours of GANs

But what is a f-divergence?

An f-divergence is a concept of Information Theory and measures how different two probability distributions are. It actually measures how much difference of information there is.

The most remarkable f-divergence is the Kullback-Liebman divergence.

$$D_{KL}(P,Q) = \int P(\mathbf{x}) \ln \left[\frac{P(\mathbf{x})}{Q(\mathbf{x})} \right] d\mathbf{x}.$$

Where P and Q are probability distributions. **BEWARE of non-conmutativity!**

Giving result as

$$g_f(x) = x, \quad f^*(g_f(x)) = e^{x-1}.$$

Flavours of GANs

GANs can go beyond what has been stated before, just by changing the lossfunction.
Depending on the selected target function to be learned, a different property the GAN Will have

GAN type	Discriminator Loss	Generator Loss
Standard	$\mathcal{L}_D(\theta_D) = \frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} [\log D(\mathbf{x}_i, \theta_D) + \log(1 - D(G(z_i, \theta_G), \theta_D))]$	$\mathcal{L}_G(\theta_G) = \frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} \log(1 - D(G(\mathbf{z}_i, \theta_G), \theta_D))$
KL-GAN	$\mathcal{L}(\theta_D) = \frac{-1}{N_{batch}} \sum_{i=1}^{N_{batch}} \left[D(\mathbf{x}_i, \theta_D) - e^{D(G(\mathbf{z}_i, \theta_G), \theta_D))-1} \right]$	$\mathcal{L}(\theta_G) = \frac{-1}{N_{batch}} \sum_{i=1}^{N_{batch}} G(\mathbf{z}_i, \theta_G)$
WGAN	$\mathcal{L}_D = \frac{1}{N_{batch}} \left[\sum_{i=1}^{N_{batch}} D(G(\mathbf{z}_i, \theta_G), \theta_D) - \sum_{i=1}^{N_{batch}} D(\mathbf{x}_i, \theta_D) \right].$	$\mathcal{L}_G = -\frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} G(\mathbf{z}_i, \theta_G)$
KL-WGAN	$\mathcal{L}_D(\theta_D) = \frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} e^{r_i D(G(\mathbf{z}_i, \theta_G), \theta_D)} - \frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} e^{D(\mathbf{x}_i, \theta_D)}$	$\mathcal{L}_G(\theta_G) = \frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} r_i D(G(\mathbf{z}_i, \theta_G), \theta_D),$

Universidad Europea

Flavours of GANs

Based on the original formulation, the lossfuntions have evolved giving origin to different formulations.

- **Standard**: Original formulation by Goodfellow. Bad convergence. (https://arxiv.org/abs/1406.2661)
- **KL-GAN**: A generalization of the original. Better convergence but sitil hard. It has an added advantage, the output of the Generator fully matches a probability distribution. (https://arxiv.org/abs/1606.00709)
- **WGAN**: Alternative formulation. Faster and easier convergence, specially for images. But the output of the Generator can not be interpreted as a probability distribution (https://arxiv.org/abs/1701.07875)
- **KL-WGAN**: Generalizes both KL-GAN and WGAN. Gets the best of both worlds: improved convergence of the WGAN and the interpretability of a KL-GAN. (https://arxiv.org/pdf/1910.09779)

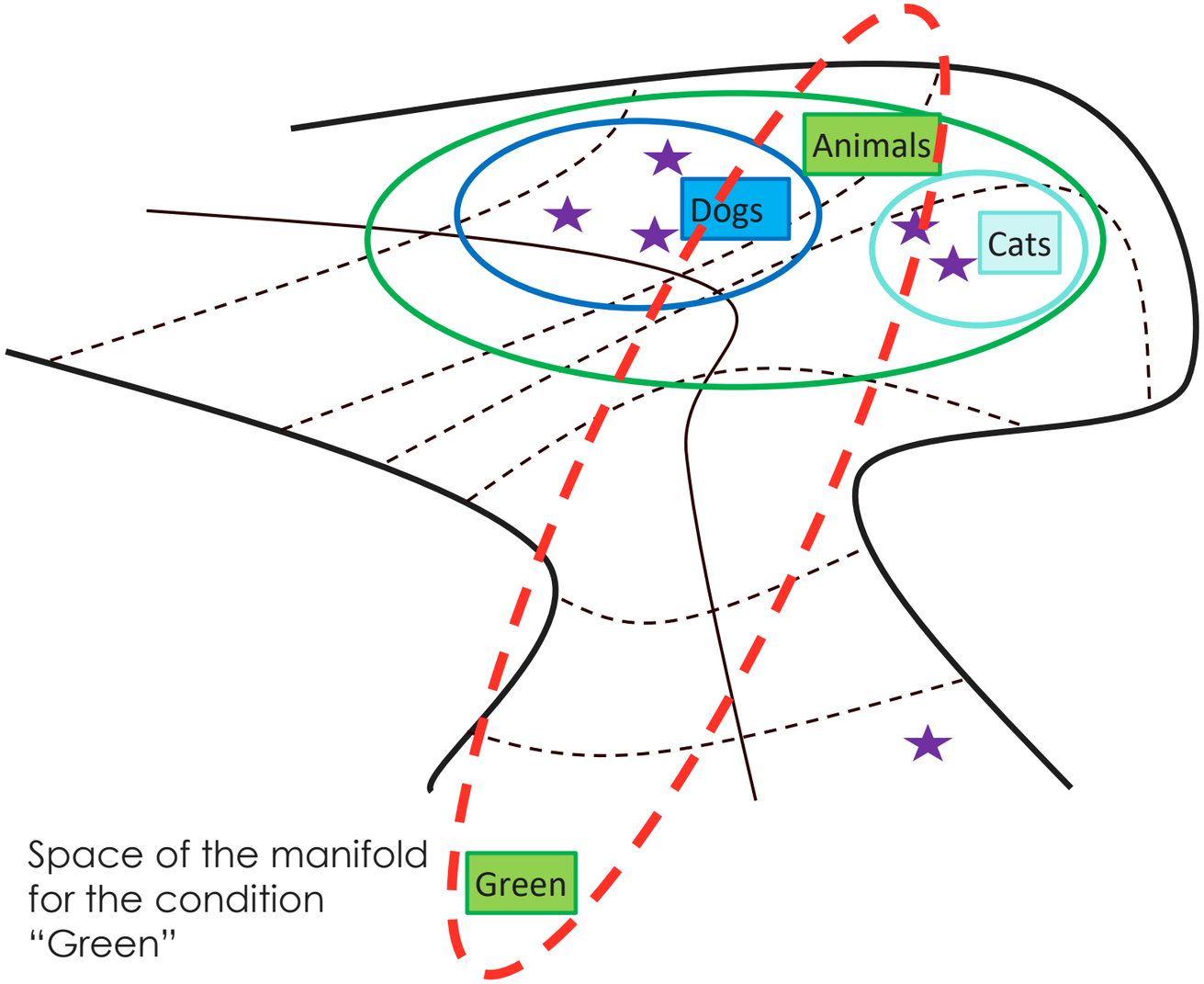
04

Conditional GANs

Conditional GANs

As seen previously, GANs will trace the underlying probability distribution of training data. The Conditional GANs (CGANs), trace the probability distribution given a condition. That is, they trace conditional probabilities.

At a conditional GAN, when sampling the latent-space, we will be only restricting ourselves to a specific region delimited by the condition.



Space of the manifold for the condition "Green"

Universidad Europea

Conditional GANs

CGANs have many different variants and applications in a wide range of different areas of application and different input type of data.

On one field CGANs have showcased extraordinary application is:

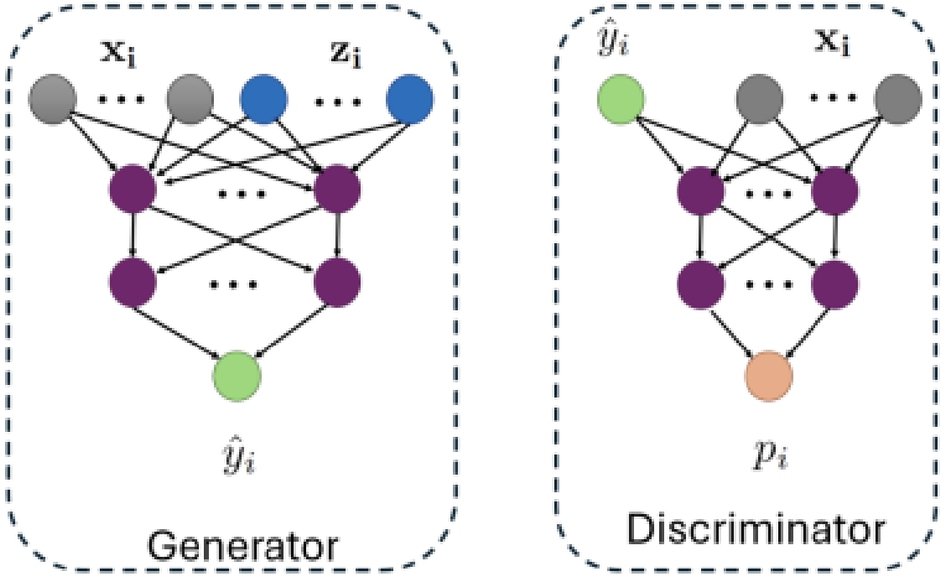
- Image synthesis and manipulation.
- Text-to-image generation.
- Image-to-mage translation.
  - Image segmentation
  - Style transfer
  - Super-resolution and image enhancement.
- Speech enhancement.
- Music generation.

Conditional GANs

At CGANs, the typical losses are the same as in the case of normal GANs.

CGANs, also have a Discriminator and a Generator. Nevertheless, a slight modification shall be made to trace the conditional probability.

Now, Generator network gets both random noise and the input condition. Discriminator takes both the generated data and the condition.



Conditional GANs

Generator and discriminator are both feedforward neural networks.

The difference with Vanilla GAN is that input neurons take the latent-space dimensión and the dimensions of the vector of conditions

Discriminator takes the image dimensions and the dimensions of the condition vector

```python
# Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(z_dim + num_classes, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, image_size),
            nn.Tanh()
        )

    def forward(self, noise, labels):
        label_onehot = torch.zeros(labels.size(0), num_classes).to(labels.device)
        label_onehot.scatter_(1, labels.unsqueeze(1), 1)
        input_tensor = torch.cat([noise, label_onehot], dim=1)
        return self.model(input_tensor)

# Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(image_size + num_classes, 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),
            nn.Sigmoid()
        )

    def forward(self, image, labels):
        label_onehot = torch.zeros(labels.size(0), num_classes).to(labels.device)
        label_onehot.scatter_(1, labels.unsqueeze(1), 1)
        input_tensor = torch.cat([image, label_onehot], dim=1)
        return self.model(input_tensor)
```

Universidad Europea

Conditional GANs

The only difference is that at CGAN, we now give the vector condition to the Discriminator when showing the real images

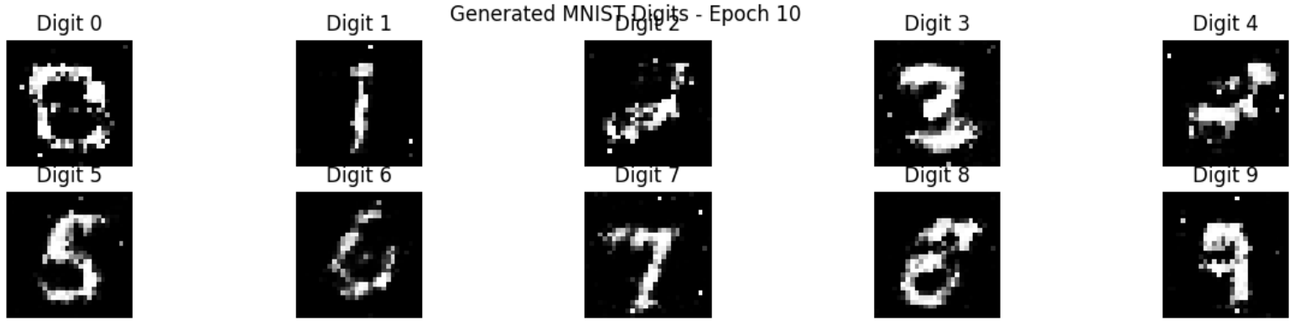
A random vector can be given as condition when showing the fake images. It is usually also given the real labels too.

```python
# Optimizers
optimizer_G = optim.Adam(generator.parameters(), lr=learning_rate, betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=learning_rate, betas=(0.5, 0.999))

# Loss function
criterion = nn.BCELoss()

# Lists to store losses
d_losses = []
g_losses = []

# Training
for epoch in range(num_epochs):
    for i, (real_images, labels) in enumerate(dataloader):
        real_images = real_images.view(-1, image_size).to(device)
        labels = labels.to(device)

        # Train Discriminator
        optimizer_D.zero_grad()

        # Real images
        real_labels = torch.ones(real_images.size(0), 1).to(device)
        real_output = discriminator(real_images, labels)
        d_loss_real = criterion(real_output, real_labels)

        # Fake images
        noise = torch.randn(real_images.size(0), z_dim).to(device)
        fake_labels = torch.randint(0, num_classes, (real_images.size(0),)).to(device)
        fake_images = generator(noise, fake_labels)
        fake_output = discriminator(fake_images.detach(), fake_labels)
        fake_targets = torch.zeros(real_images.size(0), 1).to(device)
        d_loss_fake = criterion(fake_output, fake_targets)

        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
        optimizer_D.step()

        # Train Generator
        optimizer_G.zero_grad()

        fake_output = discriminator(fake_images, fake_labels)
        g_loss = criterion(fake_output, real_labels)
        g_loss.backward()
        optimizer_G.step()

        # Store losses
        d_losses.append(d_loss.item())
        g_losses.append(g_loss.item())

        if i % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(dataloader)}], '
                  f'D Loss: {d_loss.item():.4f}, G Loss: {g_loss.item():.4f}')

```

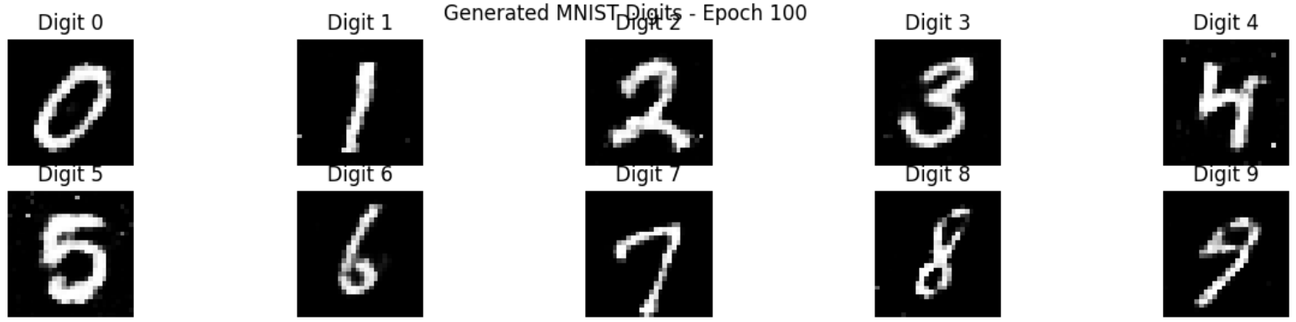
Conditional GANs

Example: MNIST image generation from input integer.

Now we have control over the generated number.



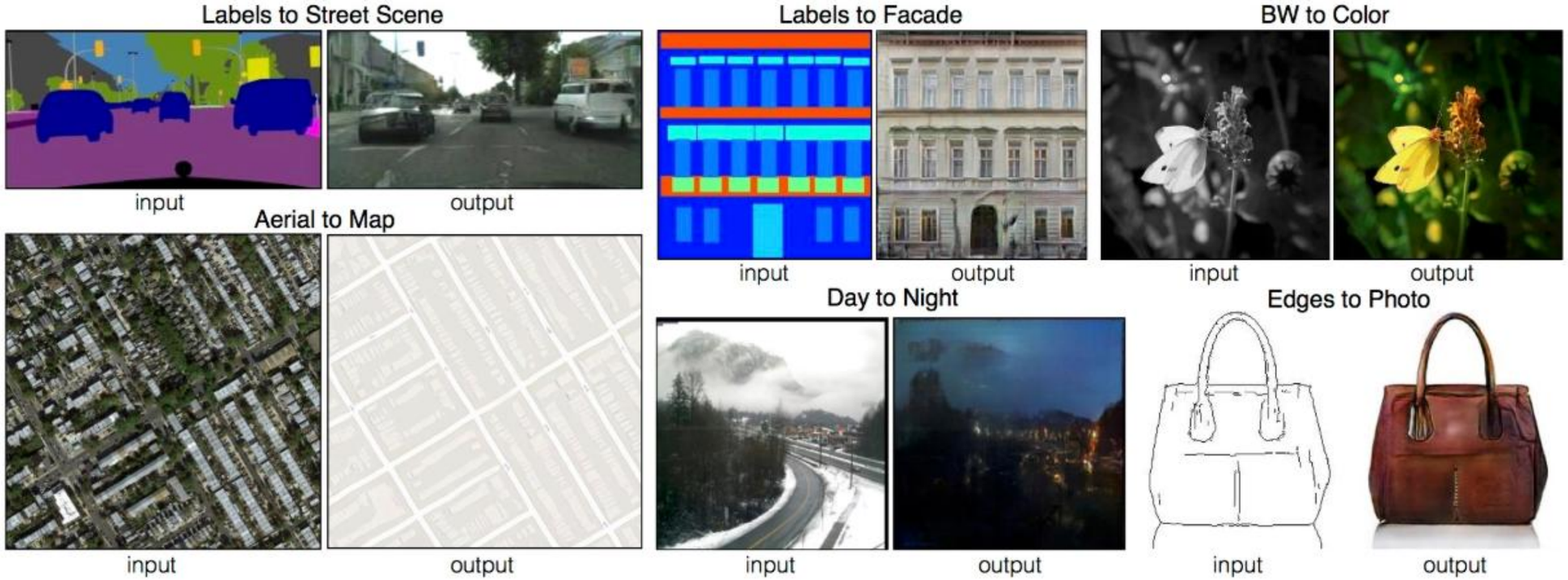
Epochs



Conditional GANs

One interesting application is image-to-image translation, which have many interesting algorithms. One of which is pix2pix.

Pix2pix is a CGAN that enable image-to-image translation for pairs of images. That is, every image, will have its own counterpart.



Conditional GANs

The most important consideration to take into account for training pix2pix is that images shall be placed in pairs.

It is important to remark that it is not strictly necessary that they are 1-to-1, but can be 1-to-many.

Beware! One night scene has many daylight couterparts.

Universidad Europea

Conditional GANs

Pix2pix is composed by:

- Generator: a Unet.
- Discriminator: a PatchGAN.

```python
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Conv2d(6, 64, 4, 2, 1),  # Input + Output channels
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(128, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(256, 512, 4, 1, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(512, 1, 4, 1, 1)
        )

    def forward(self, input_img, target_img):
        return self.model(torch.cat([input_img, target_img], 1))
```

```python
class UNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels, down=True, use_dropout=False):
        super().__init__()
        if down:
            self.conv = nn.Conv2d(in_channels, out_channels, 4, 2, 1, bias=False)
        else:
            self.conv = nn.ConvTranspose2d(in_channels, out_channels, 4, 2, 1, bias=False)

        self.norm = nn.BatchNorm2d(out_channels)
        self.relu = nn.LeakyReLU(0.2, inplace=True) if down else nn.ReLU(inplace=True)
        self.dropout = nn.Dropout(0.5) if use_dropout else None

    def forward(self, x):
        x = self.conv(x)
        x = self.norm(x)
        x = self.relu(x)
        if self.dropout:
            x = self.dropout(x)
        return x

class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        # Encoder
        self.down1 = nn.Conv2d(3, 64, 4, 2, 1)  # 128x128 -> 64x64
        self.down2 = UNetBlock(64, 128, down=True)  # 32x32
        self.down3 = UNetBlock(128, 256, down=True)  # 16x16
        self.down4 = UNetBlock(256, 512, down=True)  # 8x8
        self.down5 = UNetBlock(512, 512, down=True)  # 4x4
        self.down6 = UNetBlock(512, 512, down=True)  # 2x2
        self.down7 = UNetBlock(512, 512, down=True)  # 1x1

        # Decoder
        self.up1 = UNetBlock(512, 512, down=False, use_dropout=True)
        self.up2 = UNetBlock(1024, 512, down=False, use_dropout=True)
        self.up3 = UNetBlock(1024, 512, down=False, use_dropout=True)
        self.up4 = UNetBlock(1024, 256, down=False)
        self.up5 = UNetBlock(512, 128, down=False)
        self.up6 = UNetBlock(256, 64, down=False)
        self.final = nn.ConvTranspose2d(128, 3, 4, 2, 1)

    def forward(self, x):
        # Encoder
        d1 = self.down1(x)
        d2 = self.down2(d1)
        d3 = self.down3(d2)
        d4 = self.down4(d3)
        d5 = self.down5(d4)
        d6 = self.down6(d5)
        d7 = self.down7(d6)

        # Decoder with skip connections
        u1 = self.up1(d7)
        u2 = self.up2(torch.cat([u1, d6], 1))
        u3 = self.up3(torch.cat([u2, d5], 1))
        u4 = self.up4(torch.cat([u3, d4], 1))
        u5 = self.up5(torch.cat([u4, d3], 1))
        u6 = self.up6(torch.cat([u5, d2], 1))

        return torch.tanh(self.final(torch.cat([u6, d1], 1)))
```

Conditional GANs

Procedure is similar as in MNIST CGANT with integers as condition. Just now the condition is another image.

In this case, no random vector is used. UNET directly can map from from original image to the other image.

PatchGAN can take the two images (real and generated) and see if they match

```python
generator = Generator().to(device)
discriminator = Discriminator().to(device)

# Loss functions
criterion_GAN = nn.BCEWithLogitsLoss()
criterion_L1 = nn.L1Loss()

# Optimizers
optimizer_G = optim.Adam(generator.parameters(), lr=0.0002, betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))

# Training parameters
num_epochs = 100
lambda_L1 = 100

print("Starting training...")
print(f"Generator parameters: {sum(p.numel() for p in generator.parameters())}")
print(f"Discriminator parameters: {sum(p.numel() for p in discriminator.parameters())}")


loss_g = []
loss_d = []

# Training loop
for epoch in range(num_epochs):
    for i, (input_imgs, target_imgs) in enumerate(train_dataloader):
        input_imgs = input_imgs.to(device)
        target_imgs = target_imgs.to(device)

        # Train Generator
        optimizer_G.zero_grad()
        fake_imgs = generator(input_imgs)

        # GAN loss
        pred_fake = discriminator(input_imgs, fake_imgs)
        real_labels = torch.ones_like(pred_fake)
        loss_GAN = criterion_GAN(pred_fake, real_labels)

        # L1 loss
        loss_L1 = criterion_L1(fake_imgs, target_imgs)

        # Total generator loss
        loss_G = loss_GAN + lambda_L1 * loss_L1
        loss_G.backward()
        optimizer_G.step()

        # Train Discriminator
        optimizer_D.zero_grad()

        # Real images
        pred_real = discriminator(input_imgs, target_imgs)
        real_labels = torch.ones_like(pred_real)
        loss_real = criterion_GAN(pred_real, real_labels)

        # Fake images
        pred_fake = discriminator(input_imgs, fake_imgs.detach())
        fake_labels = torch.zeros_like(pred_fake)
        loss_fake = criterion_GAN(pred_fake, fake_labels)

        # Total discriminator loss
        loss_D = (loss_real + loss_fake) * 0.5
        loss_D.backward()
        optimizer_D.step()
```

Universidad Europea

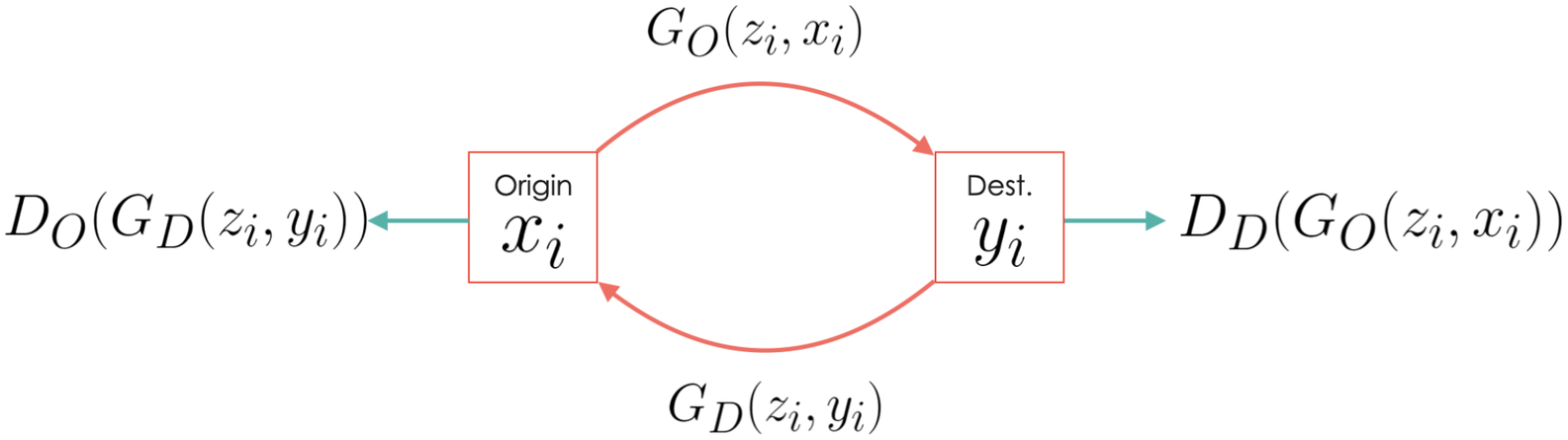
Conditional GANs

Nevertheless, there are cases where it is impossible to arrange images in image-pairs.

In this scenario, is where Cycle-GAN appears.

Cycle-GAN consist on 4 networks: 2 discriminators and 2 generators, all of them competing with each other.

In this case, images are arranged on just two sets: the origin set and the destination set.



Universidad Europea

Conditional GANs

While Diffusion Models have replaced largely GANs at general image generation because Diffusion models are much more stable and produce more realistic images, **CGAN still remain the state-of-the art for style transfer and image-to-image translation**.

This is because style-transfer is aligned better with overall CGAN objectives, because:

- Style-transfer is a conditional problem (like CGANs).
- PatchGAN stands for a top performer at Discriminator network because ensures both local and global realism.
- Most CGAN architectures were optimized for image-to-image translation, while diffusion architectures where optimized for text-to-image.
- Mode-collapse is what you aim when doing image-to-image translation (i.e. good realistic counterpart, rather the average between different modes).
- Image-to-image translation is just a guided transformation. Diffusion models are intended for image generation, they are an overkill for the tast.
- Cycle consistency of Cycle-GAN ensures visual coherence across images and invertibility across transformed spaces (which Diffusion models do not have).

Universidad Europea

Thanks

Dr. Manuel García Fernández
manuel.garcia2@universidadeuropea.es

Ve más allá