Probabilistic generative models

Image generation

Variational Autoencoders (VAEs),

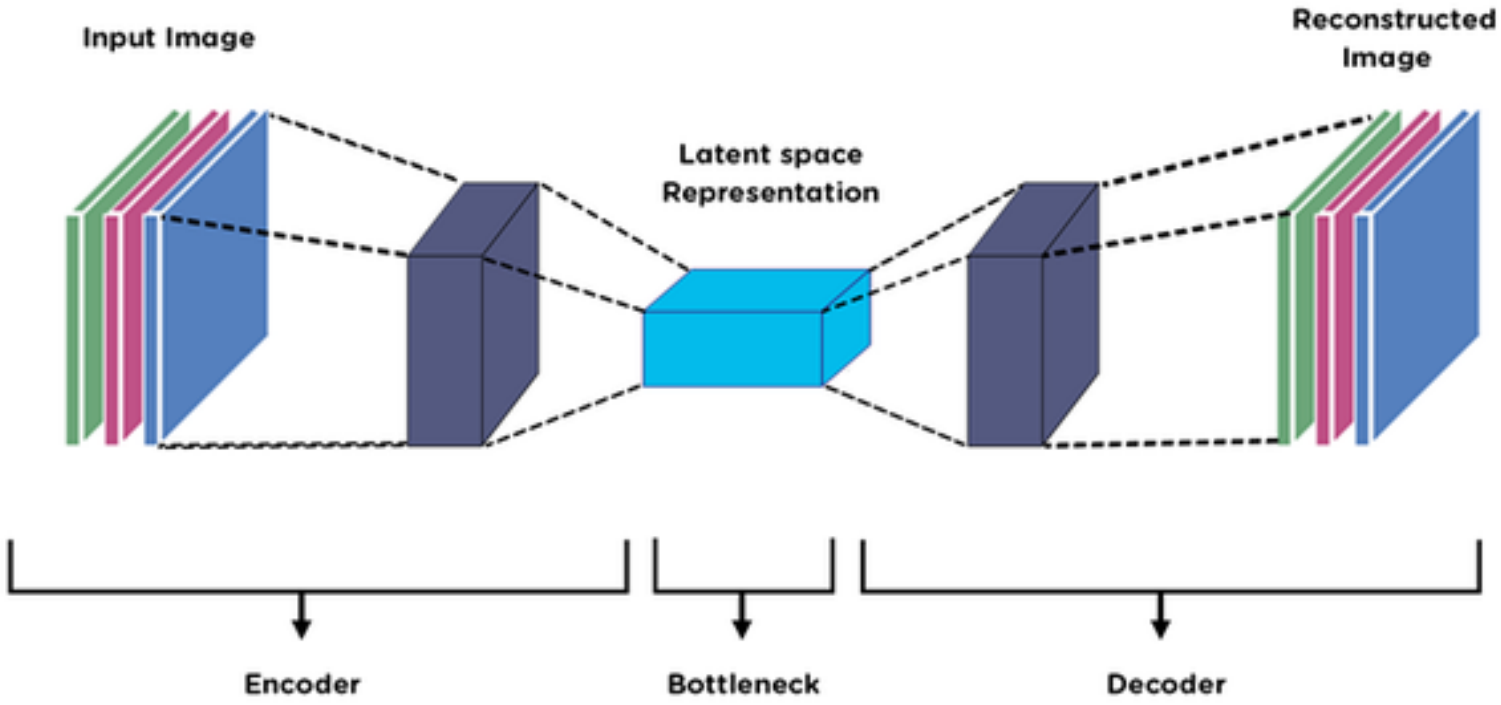
Generative Adversarial Networks (GANs),

and Diffusion Model (DMs)

https://tinyurl.com/43bar97u

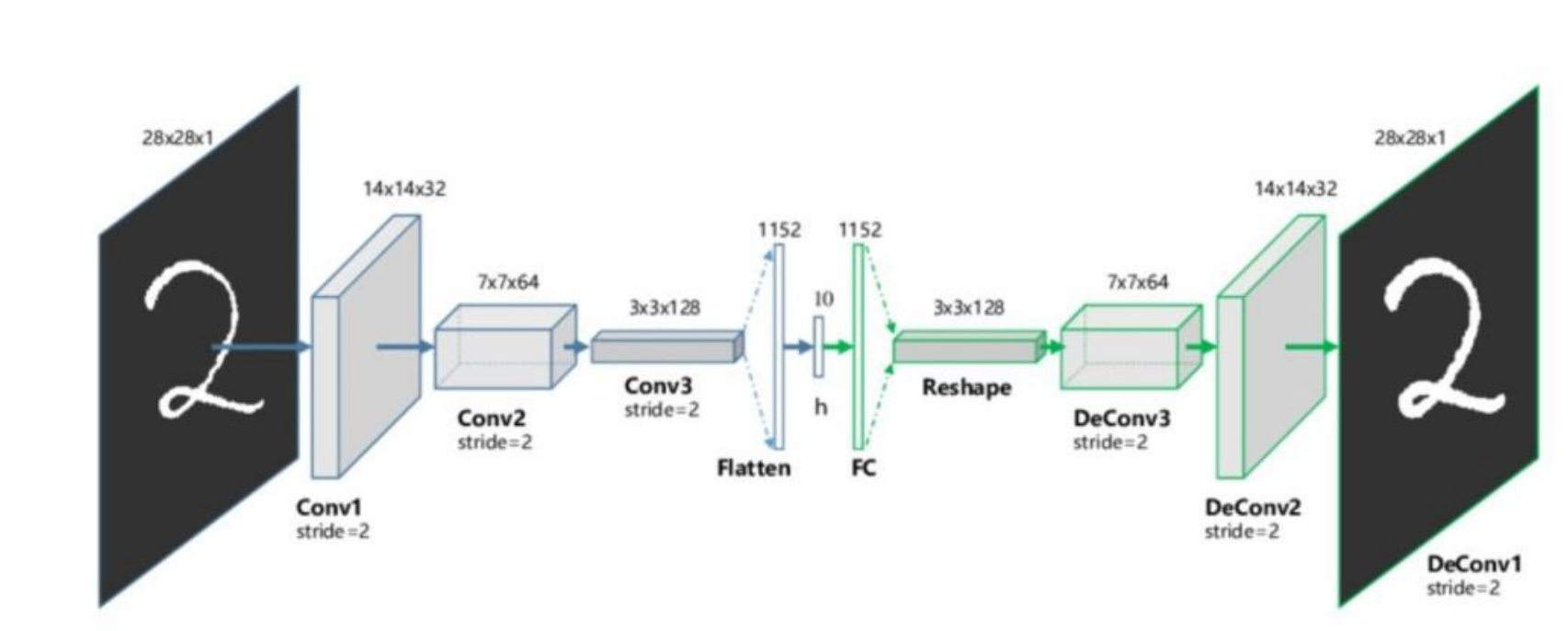
Introduction to Autoencoders and VAEs - Basic Autoencoders

- **Principle of Autoencoders:**
  - Autoencoders are a type of neural network designed specifically to learn an efficient representation of data, typically for the purpose of dimensionality reduction or feature learning.
  - These networks seek to transform input data into a different form in the hidden layer (often termed as the "latent" space) and then retransform it back to its original form at the output layer.



Introduction to Autoencoders and VAEs - Basic Autoencoders

- **Autoencoders consist of two primary components:**
  - **Encoder:** Compresses the input into a latent code (a bottleneck layer) which attempts to capture the significant features of the input data in a lower-dimensional space.
  - **Decoder:** Reconstructs the input data from this compressed latent code.



Introduction to Autoencoders and VAEs - Basic Autoencoders

- **Encoder:**
  - Maps input data *x* to a latent representation *z*.
  - The encoder can be represented as a function $z = f(x)$, where *h* is the latent representation.
- **Decoder:**
  - Maps the latent representation *z* back to the reconstructed data $\hat{x}$.
  - The decoder can be represented as a function $\hat{x} = g(z)$.

```
class Encoder(nn.Module):
        def __init__(self, input_dim, hidden_dim):
                super(Encoder, self).__init__()
                self.fc = nn.Linear(input_dim, hidden_dim)
        def forward(self, x):
                return F.relu(self.fc(x))

class Decoder(nn.Module):
        def __init__(self, hidden_dim, output_dim):
                super(Decoder, self).__init__()
                self.fc = nn.Linear(hidden_dim, output_dim)
        def forward(self, z):
                return torch.sigmoid(self.fc(z))
```

- **Application to Dimensionality Reduction:**
  - Autoencoders are often used as a non-linear alternative to Principal Component Analysis (PCA) for dimensionality reduction.
  - They can capture more complex relationships and structures in the data because of their non-linear nature.
  - Applications include noise reduction, data compression, and data visualization in lower dimensions.

```
class Autoencoder(nn.Module):
        def __init__(self, input_dim, hidden_dim):
                super(Autoencoder, self).__init__()
                self.encoder = Encoder(input_dim, hidden_dim)
                self.decoder = Decoder(hidden_dim, input_dim)
        def forward(self, x):
                latent_code = self.encoder(x)
                reconstructed = self.decoder(latent_code)
                return reconstructed
```

Introduction to Autoencoders and VAEs - Probabilistic Formalism and Introduction to VAEs

Limitations of Classical Autoencoders:

- They are typically deterministic and do not model the underlying distribution of the data.
- Because of their limited variability, classical autoencoders might not be suitable for generating new, realistic instances of data.

VAEs (Variational Autoencoders):

- VAEs introduce a probabilistic approach to autoencoders, allowing generation of new data by modeling uncertainty and variability.
- They aim to learn a probability distribution over the latent variables instead of fixed encodings.
- **Objective:** The goal is to learn the parameters of the latent space such that new samples can be generated from this space.

KINGMA, Diederik P. et WELLING, Max. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

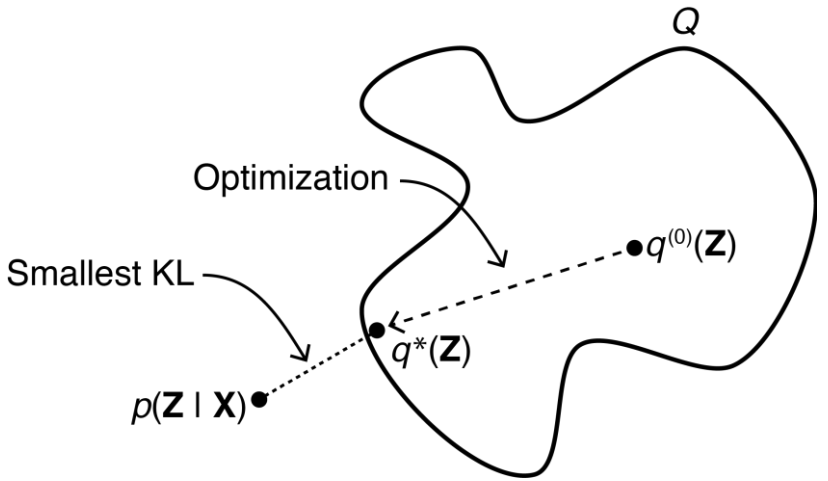
Introduction to Autoencoders and VAEs - Probabilistic Formalism and Introduction to VAEs

ELBO (Evidence Lower Bound) Objective:

- To train a VAE, we maximize a variation of the likelihood of the data which includes both the reconstruction error and the KL divergence between the learned latent variable distribution and a prior distribution.
- The ELBO loss function *L* includes:
  - A reconstruction term $L_{reconstruction}$ which ensures the generated samples are like the actual data.
  - A regularization term $L_{regularization}$ (KL divergence) which ensures the latent space distribution is similar to the prior (usually a Gaussian). Measures how much the learned latent distribution deviates from the prior distribution.

$$\text{ELBO} = \underbrace{\mathbb{E}q(z|x)[\log p(x|z)]}_{L_{reconstruction}} - \underbrace{DKL(q(z|x) \parallel p(z))}_{L_{regularization}}$$

$$D_{KL}(q(z|x)||p(z)) = \frac{1}{2}\sum_{i=1}^{L}\left(1 + \log(\sigma_i^2) - \mu_i^2 - \sigma_i^2\right)$$



Introduction to Autoencoders and VAEs - Probabilistic Formalism and Introduction to VAEs

Optimization and Learning:

- **VAE Loss Function:** The total loss is a combination of the reconstruction loss and the KL divergence.
  - **Reconstruction Loss:** Measures how well the decoder can reconstruct the input from the latent representations.
  - **KL Divergence:** Measures how much the learned latent distribution deviates from the prior distribution.

```
def vae_loss(reconstructed_x, x, mu, logvar):
        BCE = F.binary_cross_entropy(reconstructed_x, x, reduction='sum’)
        KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
        return BCE + KLD
```

- Common optimization techniques such as SGD or Adam are used for training VAEs.

Reparameterization Trick:

- Allows the gradient to propagate through the sampling process during backpropagation.
- This trick involves parameterizing the sampling step in terms of a deterministic function of the network output and some independent random noise.
- $z = \mu + \sigma . \epsilon$ where ϵ is drawn from a standard normal distribution.
- By utilizing the reparameterization trick, VAEs facilitate the learning of a smooth and well-structured latent space, thereby enhancing their ability to generate new, realistic data points.

Introduction to Autoencoders and VAEs - Network Architecture

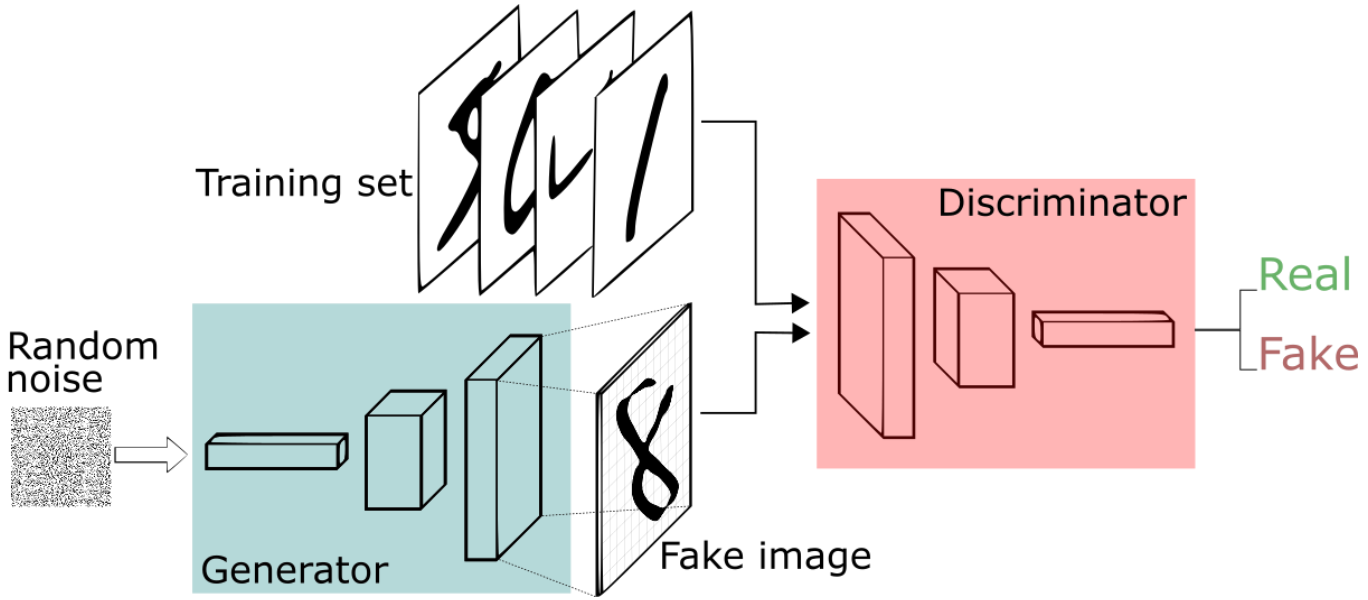
- **Encoder:**
  - Transforms the input data into a distribution in the latent space, usually represented as a Gaussian distribution with mean μ and standard deviation σ.
- **Decoder:**
  - Samples from the latent distribution and generates data from these samples.
- **VAEs can employ various neural network architectures based on the task and data type, e.g.,**
  - Deep Neural Networks (DNNs): For general tabular data.
  - Convolutional Neural Networks (CNNs): For image data.
  - Recurrent Neural Networks (RNNs): For sequential data.

```
class VAE(nn.Module):
        def __init__(self, input_dim, hidden_dim, latent_dim):
                super(VAE, self).__init__()
                self.encoder = Encoder(input_dim, hidden_dim)
                self.fc_mu = nn.Linear(hidden_dim, latent_dim)
                self.fc_logvar = nn.Linear(hidden_dim, latent_dim)
                self.decoder = Decoder(latent_dim, hidden_dim)
                self.final_layer = nn.Linear(hidden_dim, input_dim)

        def encode(self, x):
                h = self.encoder(x)
                mu = self.fc_mu(h)
                logvar = self.fc_logvar(h)
                return mu, logvar

        def reparameterize(self, mu, logvar):
                std = torch.exp(0.5 * logvar)
                eps = torch.randn_like(std)
                return mu + eps * std

        def decode(self, z):
                h = self.decoder(z)
                return torch.sigmoid(self.final_layer(h))

        def forward(self, x):
                mu, logvar = self.encode(x)
                z = self.reparameterize(mu, logvar)
                return self.decode(z), mu, logvar
```

Introduction to GANs- Principle of GANs

Concept and Motivation:

- **Two Competing Neural Networks:**
  - GANs (Generative Adversarial Networks) consist of two neural networks that are trained simultaneously by an adversarial process.
  - **Generator (G):** Attempts to generate realistic synthetic data.
  - **Discriminator (D):** Attempts to distinguish between real data and synthetic data produced by the generator.

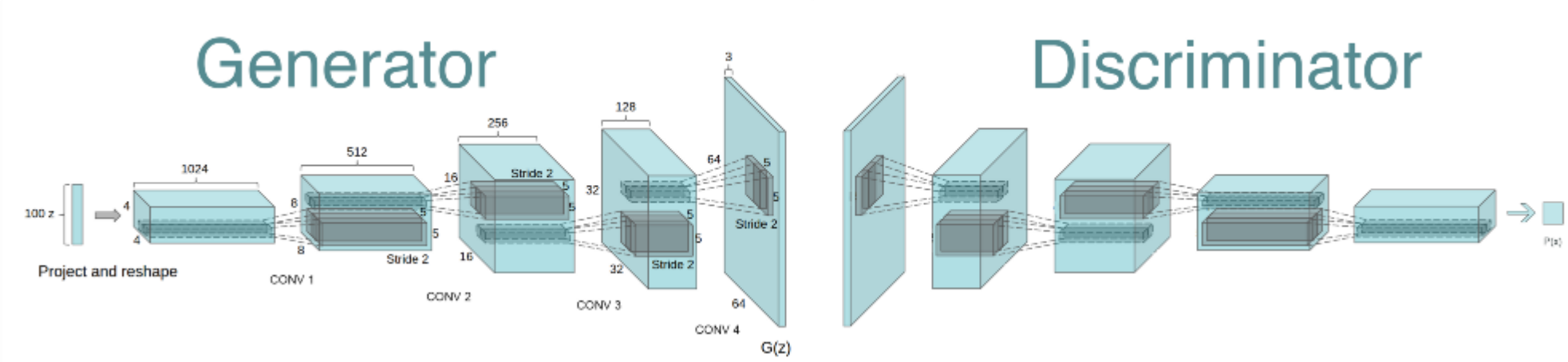


Introduction to GANs- Principle of GANs

General Functioning:

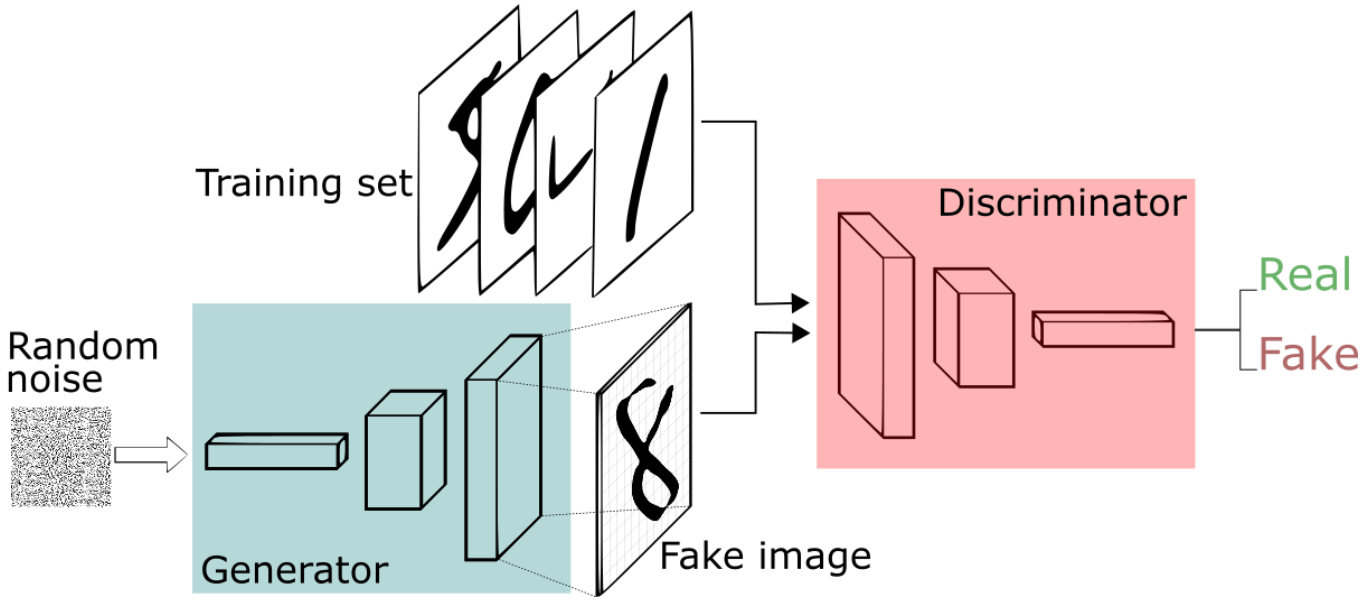
The training process of GANs is a zero-sum game where the success of one network corresponds to the failure of the other.

- **Generator (G):**
  - The generator receives a random latent vector (often sampled from a normal or uniform distribution).
  - It transforms this latent vector into synthetic data that resembles the real data.
- **Discriminator (D):**
  - The discriminator receives data (both real and generated) and predicts whether the data is real or fake.
  - Its goal is to correctly classify real data from the training set and fake data from the generator.



Introduction to GANs- Principle of GANs

- **Discriminator Training:**
  - Given a batch of real data examples from the training set and a batch of generated samples from the generator:
    - The discriminator trains to maximize the probability of assigning the correct label to both real and fake instances.
- **Generator Training:**
  - The generator receives feedback from the discriminator about the generated samples.
  - It updates its weights to maximize the probability of the discriminator classifying these fake samples as real.



Introduction to GANs- Formal Formulation

The game played between the generator and discriminator can be formally described using a minimax formulation:

$$\min_G \max_D \mathbb{E}x \sim p\text{data}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))]$$

- **Objective of the Generator (G):** To minimize the probability that the discriminator correctly identifies the fake samples.

$$\min_G \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))]$$

- **Objective of the Discriminator (D):** To maximize the probability of correctly distinguishing real data from fake data.

$$\max_D \mathbb{E}x \sim p\text{data}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))]$$

Introduction to GANs- Architecture

Generator (G):

- **Latent Vector:** The generator receives a random vector *z* from a predefined latent space (e.g., a normal or uniform distribution).
- **Synthetic Data Generation:** Through a series of transformations (often implemented as layers in a neural network), the generator converts this latent vector into synthetic data.

Discriminator (D):

- **Input:** The discriminator receives either real data from the dataset or synthetic data generated by the generator.
- **Classification:** It processes the input through a series of transformations (layers) and outputs a single value (typically through a sigmoid function) representing the probability that the input data is real.

```
class Generator(nn.Module):
        def __init__(self, latent_dim, data_dim):
                super(Generator, self).__init__()
                self.model = nn.Sequential( nn.Linear(latent_dim, 128),
                nn.ReLU(True),
                nn.Linear(128, 256),
                nn.ReLU(True),
                nn.Linear(256, 512),
                nn.ReLU(True),
                nn.Linear(512, data_dim),
                nn.Tanh() )

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

```
class Discriminator(nn.Module):
        def __init__(self, data_dim):
                super(Discriminator, self).__init__()
                self.model = nn.Sequential( nn.Linear(data_dim, 512),
                nn.LeakyReLU(0.2),
                nn.Linear(512, 256),
                nn.LeakyReLU(0.2),
                nn.Linear(256, 128),
                nn.LeakyReLU(0.2),
                nn.Linear(128, 1),
                nn.Sigmoid() )

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

Introduction to GANs- Cost Function, Convergence

- The cost function for training GANs is based on binary cross-entropy.
  - **Discriminator Loss:**
    - For real data: $\log D(x)$
    - For fake data: $\log(1 - D(G(z)))$
  - **Generator Loss:**

    $$\log(1 - D(G(z)))$$

```
def discriminator_loss(real_output, fake_output):
                real_loss = nn.BCELoss()(real_output, torch.ones_like(real_output))
                fake_loss = nn.BCELoss()(fake_output, torch.zeros_like(fake_output))
                total_loss = real_loss + fake_loss return total_loss

def generator_loss(fake_output):
                return nn.BCELoss()(fake_output, torch.ones_like(fake_output))
```

- **Stabilization of the Minimax Game:**
  - Training GANs can be challenging because it involves finding a delicate balance between the generator and the discriminator.
  - If one network becomes too strong, it can overpower the other, leading to poor training dynamics.
  - Proper convergence requires careful tuning of the learning rates, network architectures, and training schedules.

Introduction to GANs- Common Problems

Mode Collapse:

- In mode collapse, the generator produces a limited variety of outputs, typically sticking to a small subset of modes in the real data distribution.
- This can occur because the generator finds it easier to trick the discriminator with few modes, rather than covering the full distribution.

```
import matplotlib.pyplot as plt

# Visualizing samples from the generator
def visualize_generator_samples(generator, latent_dim, num_samples=100):
                with torch.no_grad():
                                latent_vectors = torch.randn(num_samples, latent_dim).to(device)
                                generated_samples = generator(latent_vectors).cpu().numpy()
                                for i in range(min(num_samples, 20)):
                                                plt.imshow(generated_samples[i].reshape(28, 28),
                                                cmap='gray’)
                                                plt.show()
```

Introduction to GANs- Common Problems

Training Instability:

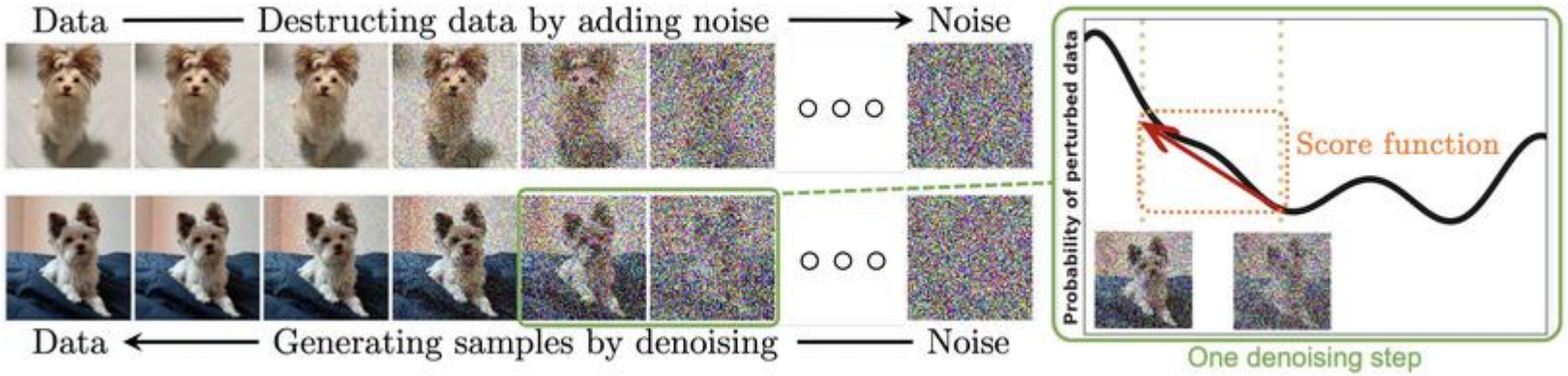
- GAN training can be unstable, leading to oscillations in the loss values and difficulty in achieving a good equilibrium.
- Various techniques, such as learning rate adjustments, batch normalization, label smoothing, and alternative loss functions (e.g., Wasserstein loss), can be employed to stabilize training.

```
# Initialize models and optimizers
generator = Generator(latent_dim=100, data_dim=784).to(device)
discriminator = Discriminator(data_dim=784).to(device)
optim_G = torch.optim.Adam(generator.parameters(), lr=0.0002)
optim_D = torch.optim.Adam(discriminator.parameters(), lr=0.0002)

# Training loop
num_epochs = 100
for epoch in range(num_epochs):
        for real_data, _ in dataloader:
                batch_size = real_data.size(0)
                real_data = real_data.view(batch_size, -1).to(device)
                # Train Discriminator
                noise = torch.randn(batch_size, 100).to(device)
                fake_data = generator(noise)
                real_output = discriminator(real_data)
                fake_output = discriminator(fake_data.detach())
                d_loss = discriminator_loss(real_output, fake_output)
                optim_D.zero_grad()
                d_loss.backward()
                optim_D.step()
                # Train Generator
                fake_output = discriminator(fake_data)
                g_loss = generator_loss(fake_output)
                optim_G.zero_grad()
                g_loss.backward()
                optim_G.step()
        print(f'Epoch {epoch} - Loss D: {d_loss.item()}, Loss G: {g_loss.item()}')
```

Introduction to Probabilistic Diffusion Models - Concept and Motivation

Definition and Overview:

- **Probabilistic Diffusion Models** are a class of generative models that generate images by simulating an inverse diffusion process. These models leverage a sequence of transformations to gradually refine random noise into coherent images.
- The core idea involves two main phases:
  - **Diffusion (Degradation):** Adding Gaussian noise incrementally to the data at each step, which makes the data progressively more noisy until it becomes pure noise.
  - **Generation (Restoration):** The reverse process where noise is incrementally reduced (denoised) to restore the original data.



Introduction to Probabilistic Diffusion Models - Concept and Motivation

Two Main Phases:

1.Diffusion (Degradation):

- This phase involves transforming clean data into noisy data through a series of steps.
- At each step (t), a small amount of Gaussian noise is added to the data.
- The diffusion process can be mathematically described by:

$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t\mathbb{I})$ where β_t denotes a time-dependent noise variance term.

2.Generation (Restoration):

- This phase is the reverse of the diffusion phase, where the model starts from pure noise and applies a series of transformations to produce clean data.
- The inverse process aims to estimate $p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \sigma_\theta(t)^2\mathbb{I})$

Introduction to Probabilistic Diffusion Models - Concept and Motivation

Motivation:

- **Quality of Generated Images:**
  - Probabilistic diffusion models are capable of generating very high-quality images by capturing complex distributions. They can model intricate data patterns, which is beneficial for tasks requiring high fidelity and diversity in the generated samples.
- **Robustness and Stability:**
  - These models tend to be more stable during training compared to some adversarial approaches like GANs, especially under complex data distributions.

Introduction to Probabilistic Diffusion Models - Architecture

Diffusion Phase:

- **Diffusion Process:**
  - Incrementally adds Gaussian noise to the data. Each step in the process makes the data slightly more noisy until it becomes indistinguishable from pure Gaussian noise.
  - Mathematical description:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t\mathbb{I})$$

Generation Phase:

- **Inverse Generation Process:**
  - Begins with a sample from a pure noise distribution and applies transformations to denoise the sample incrementally.
  - Mathematical description:

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \sigma_\theta(t)^2\mathbb{I})$$

Training:

- **Training Objective:**
  - The model is trained to approximate the reverse process using a neural network to estimate the parameters of the denoising steps.
  - The training loss includes the sum of KL divergence terms over the diffusion/denoising steps:

$$L = \sum_{t=1}^{T} D_{KL}(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t))$$

Introduction to Probabilistic Diffusion Models - Architecture

Example Implementation in PyTorch (Simplified):

```
import torch
import torch.nn as nn
import torch.optim as optim

# Simplified diffusion model
class DiffusionModel(nn.Module):
        def __init__(self, beta_schedule):
                super(DiffusionModel, self).__init__()
                self.betas = beta_schedule
        def forward(self, x_t, t):
                return x_t - self.betas[t] * torch.randn_like(x_t)

# Hyperparameters and models
timesteps = 1000

# Number of diffusion steps
beta_schedule = torch.linspace(0.0001, 0.02, timesteps)
diffusion_model = DiffusionModel(beta_schedule)

# Simplified training
optimizer = optim.Adam(diffusion_model.parameters(), lr=1e-4)
for epoch in range(num_epochs):
        for batch in dataloader:
                x0 = batch.to(device)
                t = torch.randint(0, timesteps, (x0.size(0),), device=x0.device).long()
                noise = torch.randn_like(x0)
                xt = diffusion_model(x0, t)
                loss = ((x0 - xt) ** 2).mean()
                optimizer.zero_grad()
                loss.backward() optimizer.step()
        print(f"Epoch {epoch} - Loss: {loss.item()}")
```

Latent Diffusion Models – Concept and motivation

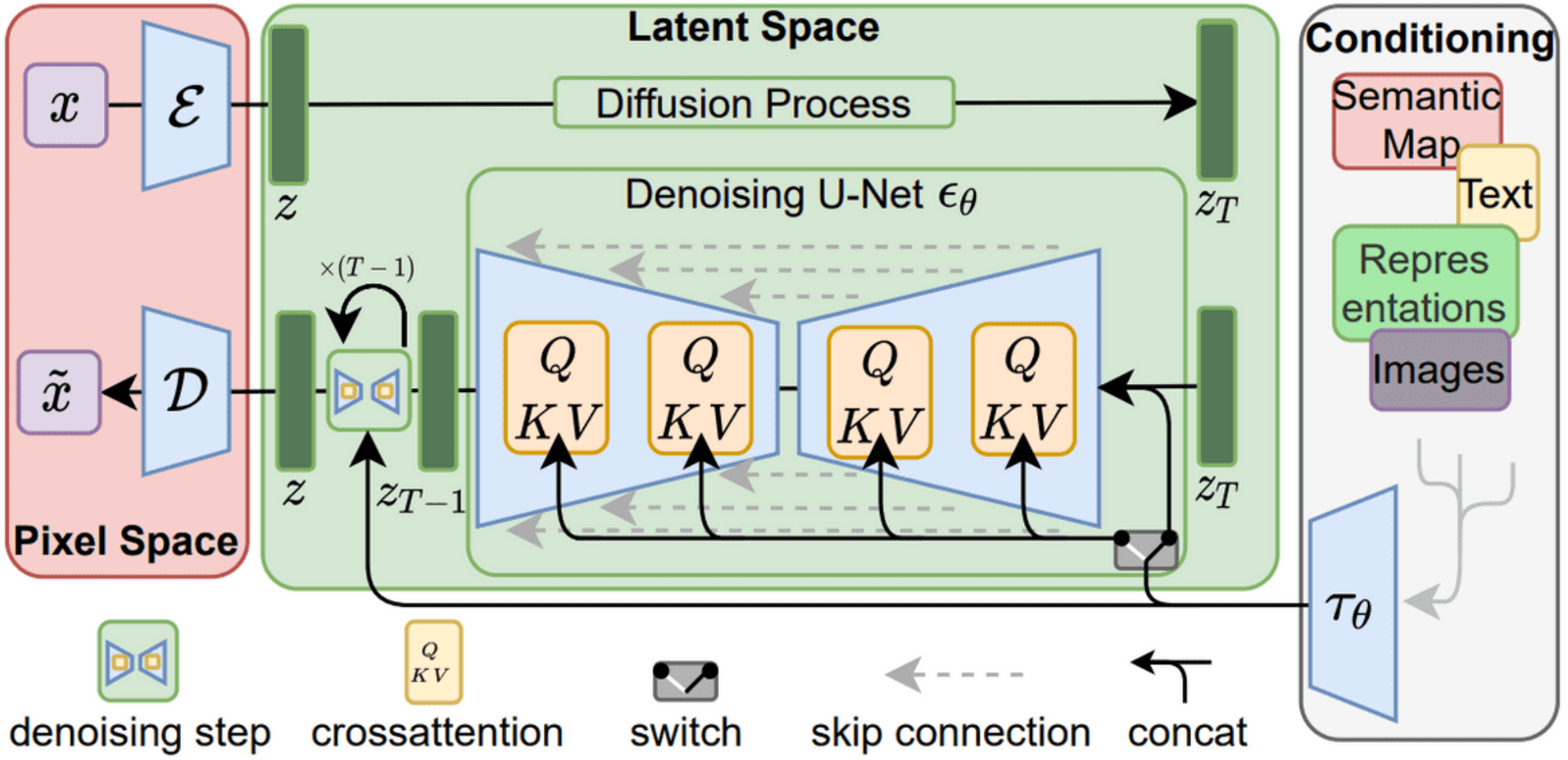
Definition and Overview:

- **Latent Diffusion Models** build upon the principles of probabilistic diffusion models by applying the diffusion process within a latent space rather than directly over the data space. This makes the generation process more efficient and potentially more effective.
- **Combination of VAEs and Diffusion Models:**
  - These models leverage the strengths of Variational Autoencoders (VAEs) to encode data into a compact latent space and diffusion models to refine latent representations.

Motivation:

- **Efficiency:**
  - Operating in a lower-dimensional latent space reduces the computational complexity and enhances the data generation efficiency.
- **Quality:**
  - The hybrid approach maintains high image quality by leveraging the accurate latent representations and effective denoising steps.

Latent Diffusion Models – Concept and motivation



Latent Diffusion Models – Architecture

Encoding and Latent Space:

- **Encoder:**
  - Transforms high-dimensional data into a lower-dimensional continuous latent space.
- **Latent Diffusion Process:**
  - The diffusion process is applied within this latent space rather than directly on the raw data

Decoding:

- **Inverse Diffusion Phase:**
  - The noisy latent representations are incrementally refined through the inverse diffusion process.
- **Two-Step Generation Process:**
  - Sampling new instances from the latent space followed by generating high-quality data from these latent representations using a decoder.

Latent Diffusion Model:

- Combines encoding, diffusion in the latent space, and decoding into a complete model.

```
class Encoder(nn.Module):
        def __init__(self, input_dim, latent_dim):
                super(Encoder, self).__init__()
                self.fc = nn.Linear(input_dim, latent_dim)

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

```
class Decoder(nn.Module):
        def __init__(self, latent_dim, output_dim):
                super(Decoder, self).__init__()
                self.fc = nn.Linear(latent_dim, output_dim)

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

```
class LatentDiffusionModel(nn.Module):
        def __init__(self, beta_schedule, latent_dim):
                super(LatentDiffusionModel, self).__init__()
                self.betas = beta_schedule
                self.encoder = Encoder(latent_dim, latent_dim)
                self.decoder = Decoder(latent_dim, latent_dim)

        def forward(self, x_t, t):
                z = self.encoder(x_t)
                latent_diffused = z - self.betas[t] * torch.randn_like(z)
                return self.decoder(latent_diffused)
```

Latent Diffusion Models – Architecture

Training:

- Similar to probabilistic diffusion models but applied to the latent space. The training objective is to minimize the discrepancy between the reconstructed clean data and the noisy data at each step.

```
# Hyperparameters and models
input_dim = 784

# Example for MNIST
latent_dim = 20
timesteps = 1000
beta_schedule = torch.linspace(0.0001, 0.02, timesteps)
latent_diffusion_model = LatentDiffusionModel(beta_schedule, latent_dim)

# Simplified training
optimizer = optim.Adam(latent_diffusion_model.parameters(), lr=1e-4)
for epoch in range(num_epochs):
                for batch in dataloader:
                                x0 = batch.view(-1, input_dim).to(device)
                                t = torch.randint(0, timesteps, (x0.size(0),), device=x0.device).long()
                                xt = latent_diffusion_model(x0, t)
                                loss = ((x0 - xt) ** 2).mean()
                                optimizer.zero_grad()
                                loss.backward()
                                optimizer.step()
                print(f"Epoch {epoch} - Loss: {loss.item()}")
```

Implementation of a Mini-Project with VAEs, GANs, Diffusion Models, or Latent Diffusion Models

Students have the flexibility to select a type of generative model to implement based on their interests and the availability of data. The options include:

1. **Variational Autoencoders (VAEs):** For those interested in probabilistic models and latent space learning.
2. **Generative Adversarial Networks (GANs):** Suitable for students keen on adversarial training and competitive neural network dynamics.
3. **Diffusion Models:** Ideal for students who want to explore iterative denoising processes.
4. **Latent Diffusion Models:** Combines elements of VAEs and Diffusion Models, suitable for those looking to implement advanced generative techniques.

Mini project - Data Preparation

Download and Pre-process a Dataset:

- Commonly used datasets include MNIST (handwritten digits), CIFAR-10 (image classification), and CelebA (celebrity faces).

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define transformations for MNIST
transform = transforms.Compose([ transforms.ToTensor(),
                transforms.Normalize((0.5,), (0.5,)) ])

# Download and load MNIST dataset
dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)
```

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define transformations for CIFAR-10
transform_cifar10 = transforms.Compose([ transforms.ToTensor(),
                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) ])

# Download and load CIFAR-10 dataset
dataset_cifar10 = datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform_cifar10)
dataloader_cifar10 = DataLoader(dataset_cifar10, batch_size=64, shuffle=True)
```

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define transformations for CelebA
transform_celeba = transforms.Compose([ transforms.Resize((64, 64)), # Resizing
images to 64x64
                transforms.ToTensor(),
                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) ])

# Download and load CelebA dataset
dataset_celeba = datasets.CelebA(root='./data', split='train', download=True,
transform=transform_celeba)
dataloader_celeba = DataLoader(dataset_celeba, batch_size=64, shuffle=True)
```

Mini project - Model Construction

For VAEs:

- **Define the encoder and decoder architecture.**

```
class Encoder(nn.Module):
        def __init__(self, input_dim, latent_dim):
                super(Encoder, self).__init__() self.fc = nn.Linear(input_dim, latent_dim)
        def forward(self, x):
                return self.fc(x)

class Decoder(nn.Module):
        def __init__(self, latent_dim, output_dim):
                super(Decoder, self).__init__() self.fc = nn.Linear(latent_dim, output_dim)
        def forward(self, z):
                return self.fc(z)
```

For GANs:

- **Define the generator and discriminator architecture.**

```
class Generator(nn.Module):
        def __init__(self):
                super(Generator, self).__init__() self.main =
                nn.Sequential( nn.Linear(100, 256), nn.ReLU(True), nn.Linear(256, 512),
                nn.ReLU(True), nn.Linear(512, 1024), nn.ReLU(True), nn.Linear(1024, 784),
                nn.Tanh() )
        def forward(self, x):
                return self.main(x)

class Discriminator(nn.Module):
        def __init__(self):
                super(Discriminator, self).__init__() self.main =
                nn.Sequential( nn.Linear(784, 1024), nn.ReLU(True), nn.Dropout(0.3),
                nn.Linear(1024, 512), nn.ReLU(True), nn.Dropout(0.3), nn.Linear(512, 256),
                nn.ReLU(True), nn.Dropout(0.3), nn.Linear(256, 1), nn.Sigmoid() )
        def forward(self, x):
                return self.main(x)
```

Mini project - Model Construction

For Diffusion Models:

- **Define the iterative denoising steps and the noise schedule.**

```
class DiffusionModel(nn.Module):
        def __init__(self, beta_schedule):
                super(DiffusionModel, self).__init__()
                self.betas = beta_schedule

        def forward(self, x_t, t):
                return x_t - self.betas[t] * torch.randn_like(x_t)
```

For Latent Diffusion Models:

- **Define the encoder, iterative refinement process in latent space, and decoder.**

```
class LatentDiffusionModel(nn.Module):
        def __init__(self, beta_schedule, latent_dim):
                super(LatentDiffusionModel, self).__init__()
                self.betas = beta_schedule
                self.encoder = Encoder(latent_dim, latent_dim)
                self.decoder = Decoder(latent_dim, latent_dim)

        def forward(self, x_t, t):
                z = self.encoder(x_t)
                latent_diffused = z - self.betas[t] * torch.randn_like(z)
                return self.decoder(latent_diffused)
```

Mini project - Model Training

Use deep learning frameworks like TensorFlow or PyTorch.

Monitor Convergence and Adjust Hyperparameters:

- Ensure that the training process is stable and the model is improving over epochs.

```
optimizer = optim.Adam(model.parameters(), lr=1e-4)
for epoch in range(num_epochs):
        for batch in dataloader:
                x0 = batch.view(-1, input_dim).to(device)
                t = torch.randint(0, timesteps, (x0.size(0),), device=x0.device).long()
                if model_type == 'vae':
                        reconstructed, mu, logvar = model(x0)
                        loss = vae_loss(reconstructed, x0, mu, logvar)
                elif model_type == 'gan':
                        # Train Discriminator
                        real_output = discriminator(x0)
                        noise = torch.randn(x0.size(0), 100).to(device)
                        fake_data = generator(noise)
                        fake_output = discriminator(fake_data.detach())
                        d_loss = discriminator_loss(real_output, fake_output)
                        optimizerD.zero_grad()
                        d_loss.backward()
                        optimizerD.step()
                        # Train Generator
                        fake_output = discriminator(fake_data)
                        g_loss = generator_loss(fake_output)
                        optimizerG.zero_grad()
                        g_loss.backward()
                        optimizerG.step()
                        loss = d_loss + g_loss
                elif model_type == 'diffusion':
                        xt = diffusion_model(x0, t)
                        loss = diffusion_loss(x0, xt) optimizer.zero_grad()
                        loss.backward()
                        optimizer.step()
        print(f"Epoch {epoch} - Loss: {loss.item()}")
```

Mini project - Evaluation and Visualization

- **Generate New Samples:**
  - After training, generate samples from the trained model to evaluate its performance.
- **Visualize the Results:**
  - Use Matplotlib or other visualization tools to display the generated samples.

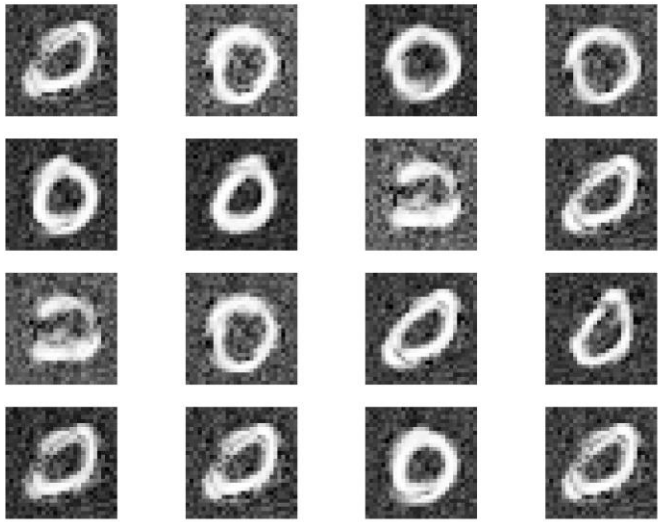
```
import matplotlib.pyplot as plt

# Sampling and Visualization
with torch.no_grad():
        if model_type == 'gan':
                noise = torch.randn(16, 100).to(device)
                samples = generator(noise).cpu().numpy()
        elif model_type == 'vae':
                noise = torch.randn(16, latent_dim).to(device)
                samples = model.decoder(noise).cpu().numpy()
        elif model_type == 'diffusion':
                noise = torch.randn(16, input_dim).to(device)
                samples = model(noise, torch.tensor(
                [timesteps-1]*16).to(device)).cpu().numpy()

        for i in range(16):
                plt.subplot(4, 4, i+1)
                plt.imshow(samples[i].reshape(28, 28), cmap='gray')
                plt.axis('off')
        plt.show()
```

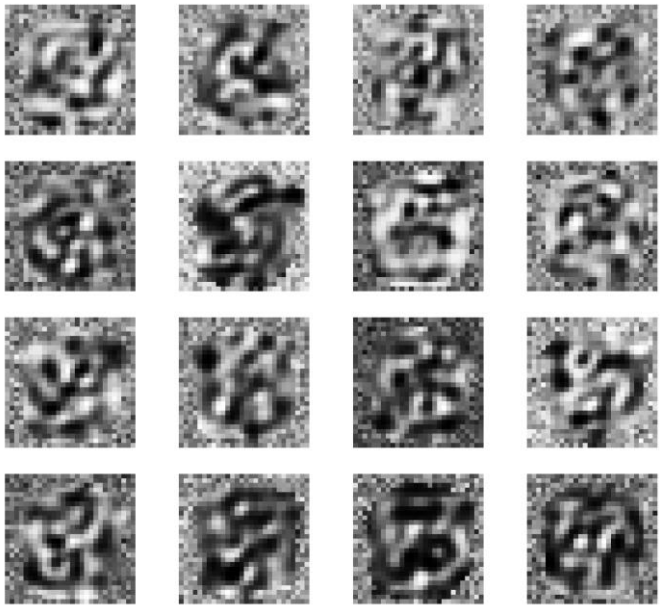
Mini project – Example code for GAN

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Definition of the Generator and Discriminator
class Generator(nn.Module):
        def __init__(self):
                super(Generator, self).__init__()
                self.main = nn.Sequential( nn.Linear(100, 256), nn.ReLU(True),
                nn.Linear(256, 512), nn.ReLU(True), nn.Linear(512, 1024),
                nn.ReLU(True), nn.Linear(1024, 784), nn.Tanh() )
        def forward(self, x):
                return self.main(x)

class Discriminator(nn.Module):
        def __init__(self):
                super(Discriminator, self).__init__()
                self.main = nn.Sequential( nn.Linear(784, 1024), nn.ReLU(True),
                        nn.Dropout(0.3), nn.Linear(1024, 512), nn.ReLU(True),
                        nn.Dropout(0.3), nn.Linear(512, 256), nn.ReLU(True),
                        nn.Dropout(0.3), nn.Linear(256, 1), nn.Sigmoid() )
        def forward(self, x):
                return self.main(x)

# Instantiate networks and define optimizers
G = Generator()
D = Discriminator()
criterion = nn.BCELoss()
optimizerD = optim.Adam(D.parameters(), lr=0.0002)
optimizerG = optim.Adam(G.parameters(), lr=0.0002)
```

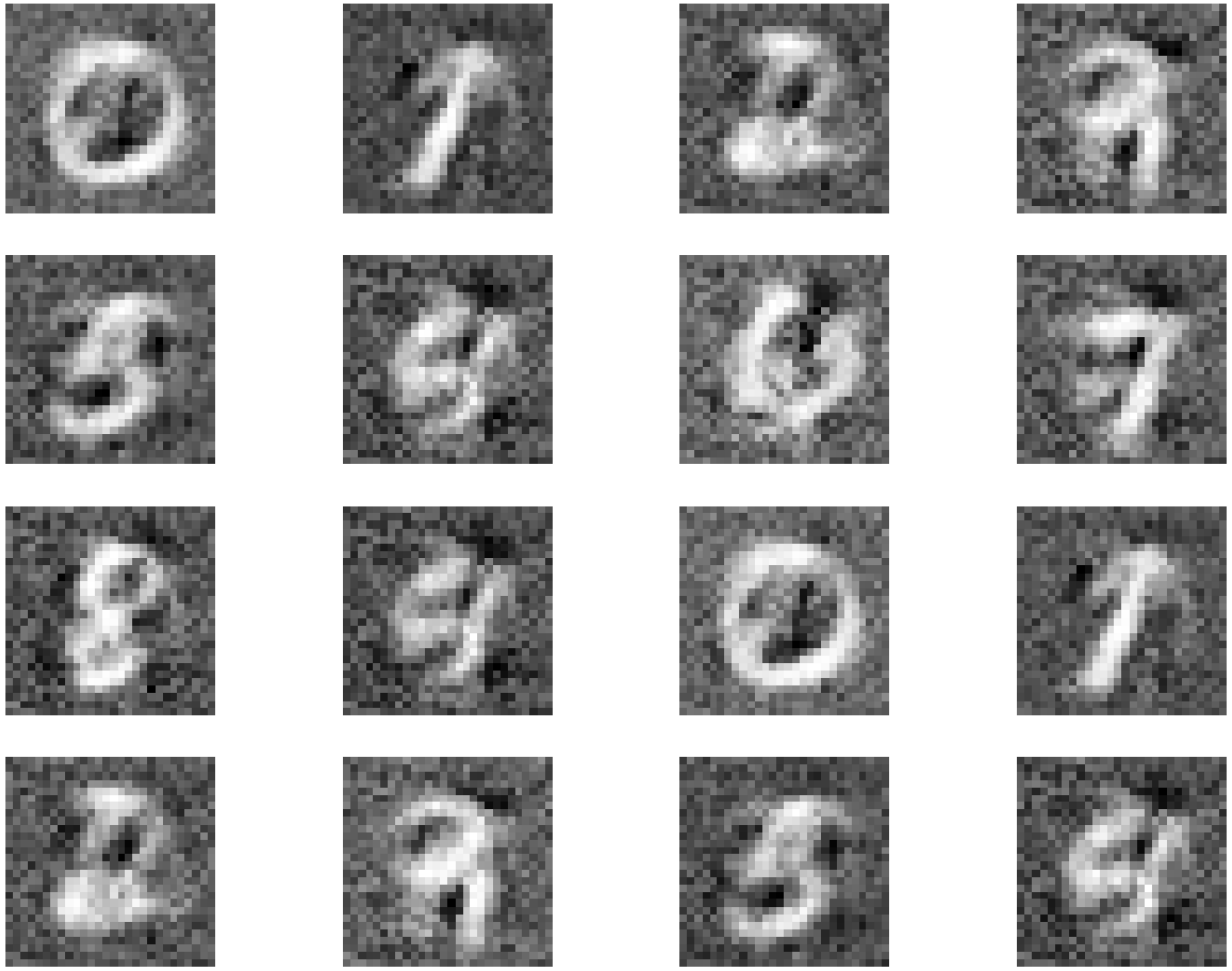
```
# Simplified training  loop
num_epochs = 100
dataloader = DataLoader(datasets.MNIST('./data', train=True, download=True,
transform=transforms.ToTensor()), batch_size=64, shuffle=True)

for epoch in range(num_epochs):
        for i, (real, _) in enumerate(dataloader):
                # Train Discriminator
                D.zero_grad()
                input = real.view(real.size(0), -1)
                output = D(input)
                label = torch.ones(input.size(0))
                errD_real = criterion(output, label)
                noise = torch.randn(input.size(0), 100)
                fake = G(noise)
                output = D(fake.detach())
                label = torch.zeros(input.size(0))
                errD_fake = criterion(output, label)
                errD = errD_real + errD_fake
                errD.backward()
                optimizerD.step()

                # Train Generator
                G.zero_grad()
                output = D(fake)
                label = torch.ones(input.size(0))
                errG = criterion(output, label)
                errG.backward()
                optimizerG.step()
        print(f'Epoch {epoch}/{num_epochs} - Loss D: {errD.item()} Loss G:
{errG.item()}')
```

Mini project – results for VAE & GAN & Diffusion





Mini project – results for CGAN



Mini project - Analyzing Results

- **Evaluation of Generation for GANs:**
  - Assess the quality of the samples generated by the GAN. This can be done by visual inspection or quantitative metrics like the Inception Score or Fréchet Inception Distance (FID).

- **Analysis of VAEs:**
  - Evaluate the quality of the reconstructions and inspect the learned latent space. Visualizations of interpolations in the latent space can be informative.

- **Analysis of Diffusion Models and Latent Diffusion Models:**
  - Consider the quality and fidelity of progressively refined samples. Assess whether the model can effectively denoise input noise to produce coherent outputs.