Lior Ben Ishay - 313135261 Keren Druker - 208678847

Deep Learning, TAU, 0510-7255

EX 3 – VAE, GAN, WGAN

Please read the submission guidelines before you start.

Theory

1. Consider the following 3 distance measures between the distributions p and q:

   - KL (Kullback–Leibler Divergence):

     $$D_{KL}(p||q) = \int_{x \in X} p(x) log \frac{p(x)}{q(x)} dx$$

     Note that it is always positive, and achieves 0 iff $\forall x \in X, p(x) = q(x)$.

     One of its disadvatages is that it is asymmetric.

   - JS (Jensen-Shannon Divergence) – a symmetric (and smoother) version of the KL divergence:

     $$D_{JS}(p||q) = \frac{1}{2} D_{KL}\left(p||\frac{p+q}{2}\right) + \frac{1}{2} D_{KL}\left(q||\frac{p+q}{2}\right)$$

   - Wasserstein Distance (also known as "Earth Mover distance), defined for continuous domain:

     $$W(p,q) = \inf_{\gamma \sim \prod(p,q)} \sum_{x,y} \gamma(x,y)\|x-y\| = \inf_{\gamma \sim \prod(p,q)} E_{(x,y)\sim\gamma}[\|x-y\|]$$

     where $\prod(p,q)$ is the set of all possible joint distributions of p and q, γ describes a specific one, and x and y are values drawn from p and q respectively. Intuitively, it measures the minimal cost of "moving" p to become q.

   Suppose we have two 2-dimensional probability distributions, P and Q:

   - $\forall (x,y) \in P, x = 0 \; and \; y \sim U(0,1)$
   - $\forall (x,y) \in Q, x = \theta, 0 \leq \theta \leq 1, and \; y \sim U(0,1)$

   θ is a given constant.

   Obviously, when $\theta \neq 0$, there is no overlap between P and Q.

   a. For the case where $\theta \neq 0$, calculate the distance between P and Q by each of the three measurements.

   b. Repeat section a for the case where $\theta = 0$.

   c. Following your answers, what is the advantage of the Wasserstein Distance over the previous two?

Lior Ben Ishay - 313135261
Keren Druker - 208678847

Ex 3 - Theory

1.a. In case of $\theta \neq 0$:

In this case we can conclude that $p(x)$ and $q(x)$ are disjoint.

Since they both have $y \sim U(0,1)$, as well as $\theta = 0$, we know that:

$x = 0$: $p(x) = 1$, $q(x) = 0$

KL Divergence:

$$D_{KL}(P \| Q) = \int_{x=0,\, y\sim U(0,1)} p(x) \cdot \log\left(\frac{p(x)}{q(x)}\right) dx =$$

$$= \int_{x=0,\, y\sim U(0,1)} 1 \cdot \log\left(\frac{1}{0}\right) = \boxed{\infty}$$

JS Divergence:

Same as above, we get:

$$D_{KL}\left(p \,\|\, \frac{p+q}{2}\right) = \int_{x=0,\, y\sim U(0,1)} p(x) \cdot \log\left(\frac{p(x)}{\frac{p(x)+q(x)}{2}}\right) dx =$$

$$= \int_{x=0,\, y\sim U(0,1)} 1 \cdot \log\left(\frac{1}{\frac{1+0}{2}}\right) dx = \log(2)$$

$$D_{KL}\left(q \,\|\, \frac{p+q}{2}\right) = \int_{x=0,\, y\sim U(0,1)} q(x)\cdot \log\left(\frac{q(x)}{\frac{p(x)+q(x)}{2}}\right) dx =$$

$$= \int_{x=0,\, y\sim U(0,1)} 1 \cdot \log\left(\frac{1}{\frac{1+0}{2}}\right) = \log(2)$$

Therefore:

$$\boxed{D_{JS}(p\|q) = \frac{1}{2}\cdot\log(2) + \frac{1}{2}\cdot\log(2) = \log(2)}$$

Wasserstein Distance:

As learnt in class, wasserstein Distance is actually the "Earth mover distance". Since P and Q are seperated by θ along the x-axis, the minimum cost of moving P to Q is θ. Therefore:

$$\boxed{W(p,q) = |\theta|}$$

1.b. In case of $\theta = 0$:

In this case $p(x)$ and $q(x)$ have exactly the same distribution: $x = \theta = 0$, $y \sim U(0,1)$, there for $p(x) = q(x)$.

KL Divergence:

$$D_{KL}(P\|Q) = \int_{x=0,\, y\sim U(0,1)} p(x)\cdot \log\left(\frac{P(x)}{q(x)}\right) dx =$$

$$= \int_{x=0,\, y\sim U(0,1)} P(x) \cdot \log\left(\frac{1}{}\right) = \boxed{0}$$

JS Divergence:

Same as above, we get:

$$D_{KL}\left(p \,\|\, \frac{p+q}{2}\right) = \int_{x=0,\, y\sim U(0,1)} p(x) \cdot \log\left(\frac{p(x)}{\frac{p(x)+q(x)}{2}}\right) dx =$$

$$= \int_{x=0,\, y\sim U(0,1)} p(x) \cdot \log\left(1\right) dx = 0$$

$$D_{KL}\left(q \,\|\, \frac{p+q}{2}\right) = \int_{x=0,\, y\sim U(0,1)} q(x) \cdot \log\left(\frac{q(x)}{\frac{p(x)+q(x)}{2}}\right) dx =$$

$$= \int_{x=0,\, y\sim U(0,1)} 1 \cdot \log(1) = 0$$

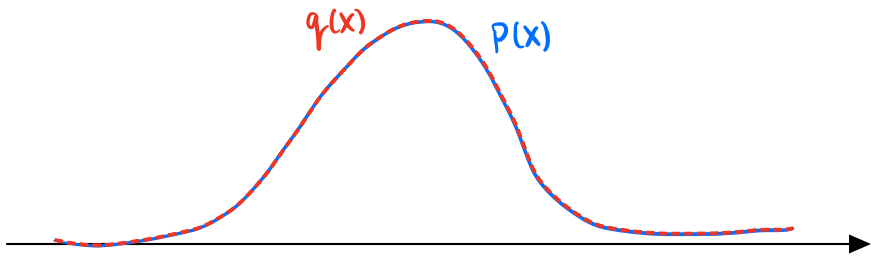
Therefore:

$$\boxed{D_{JS}(p \,\|\, q) = 0}$$

Wasserstein Distance:

As stated before, P and Q have the same distribution therefore in term of "earth mover" the distance between them is zero.

Visualy we can see that:



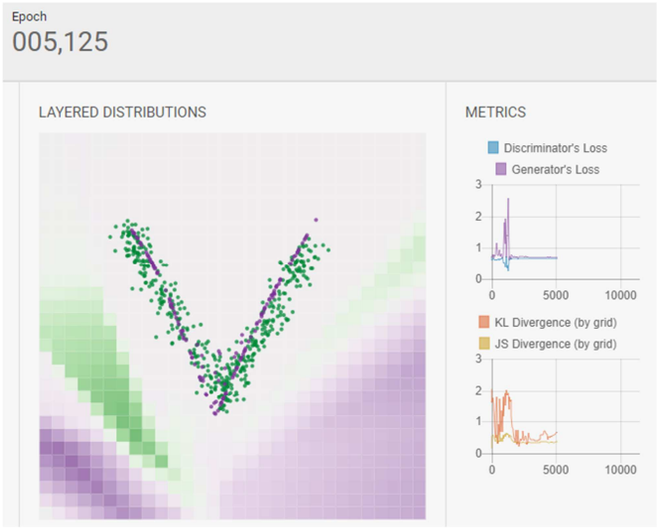
So: $\boxed{W(p, q) = 0}$

1.C. Following the previous answers, we can indicate the advantages of Wasserstein Distance over KL and JS.

As seen above, the Wasserstein Distance is a robust and useful even when two distribution are completely different (disjoint). It is very useful when the distributions don't have any overlap, common case in the real world. Thanks to the fact that this metric is non-negative as well as symmetric, it is useful in cases of optimization and analyses like gradiant decents in our scope.

Practical

2. To gain Intuition (and have fun), go to https://poloclub.github.io/ganlab/. Read the instructions, and then train a GAN by yourself. Draw a distribution and train the GAN on it. Submit in the PDF a screen shot of the result along with the epoch number and the graphs on the right. For example:



   Comments:

   - The model should be trained until convergence, and at least 5,000 epochs. Convergence can be seen visually and by the graphs.
   - Do not remove the gradients in the picture. In the example, they just nullified after convergence.
   - Make sure to explain your results!

3. Here we address semi-supervised learning via a variational autoencoder. You will be implementing a part of the paper "Semi-supervised Learning with Deep Generative Models", by Kingsma et al.

   Read the paper, and implement the M1 scheme, as described in Algorithm1, and detailed throughout the paper. It is based on a VAE for feature extraction, and then a (transductive) SVM for classification.

   Implement the network suggested for MNIST, and apply it on the Fashion MNIST data set. Present the results for 100, 600, 1000 and 3000 labels, as they are presented in Table 1 in the paper.

   For simplicity, you may use an regular SVM (with a kernel of your choice). In addition, no need for "self training". Simply take the latent representation of the labeled data as training, and then test it on the test set.

Comments:

- Please mention in the pdf which kernel did you use.
- Describe in the readme file how to train and test your model.
- Make sure to save the SVM model as well as the NN weights.
- For each amount of labels, make sure you have an equal amount of examples from each class. In addition, use a fixed seed value so the results would be consistent.
- Compare your results with table 1 in the paper (you might want to also train with MNIST so as to get a better insight into your results).

4. Here you will get to play a bit with GANs and WGANs, by implementing a part of the paper "Improved Training of Wasserstein GANs", by Gulrajani et al.

   Read the paper, and implement by yourself the architecture designed for CIFAR10 (The simpler one, without residual layers, section F). Make appropriate modifications for MNIST. Then apply it on the Fashion MNIST data set.

   a. Plot the loss function as a function of the iterations for training with the DCGAN and the WGAN.

   b. Select two "DCGAN" generated images, and two "WGAN" generated images. If possible, from the same label (or two labels). In addition, add two real images (from the corresponding label).

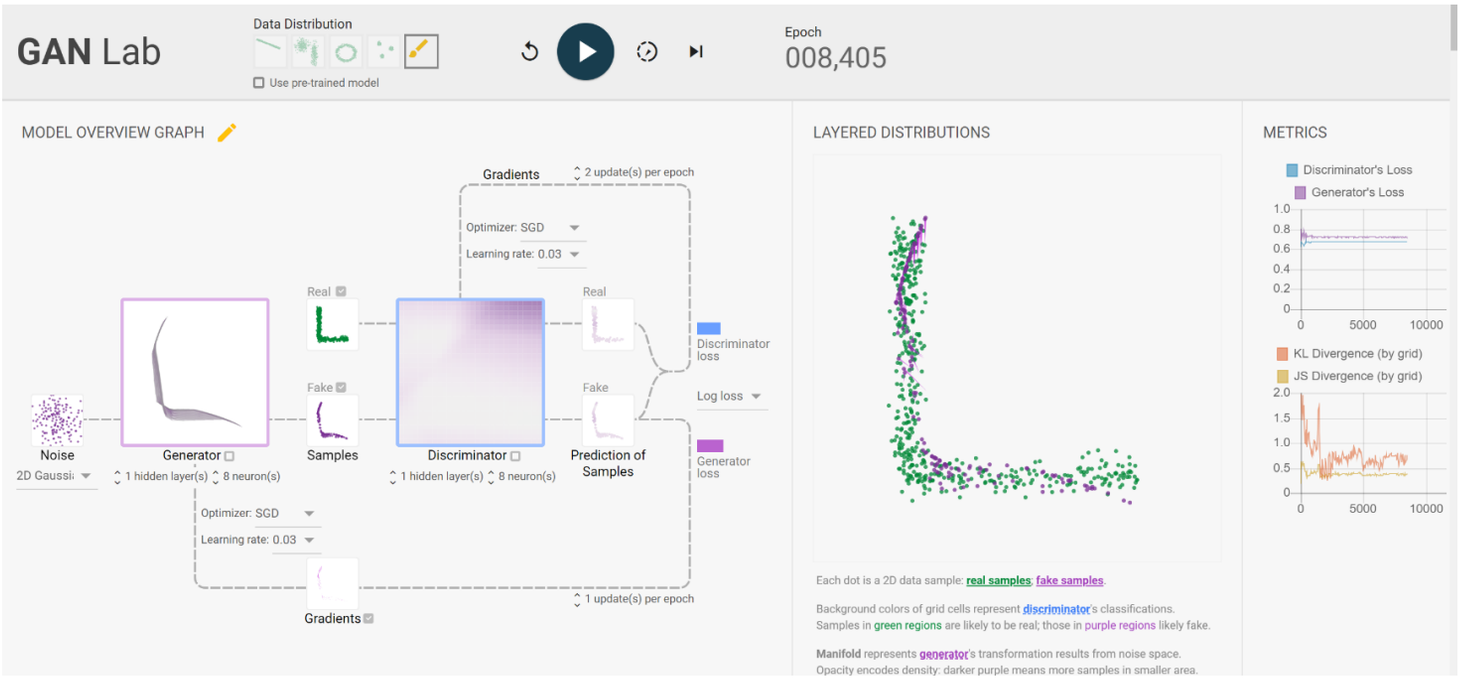
Comments:

- Describe in the readme file how to train the model, and how to generate (with the trained weights) a new picture from a trained model with DCGAN, and how to do it with WGAN.
- If you decide to use "Gradient Penalty", "Weight Clipping", or the architecture with the residual layers, you may do so, but note your decision in the PDF.
- Write the amount of failed convergences of your implementation before having a successful one (both for DCGAN and WGAN).
- Make sure the code and the readme file support easy generation of new images.

Good Luck!

Question 2:

Using the GAN Lab platform, we received the following results:



The **generated samples** (shown in purple) are gradually aligning with the **real samples** (in green). This indicates that the Generative Adversarial Network (GAN) is learning to approximate the target distribution. However, there are still some discrepancies: certain areas have too many similar generated samples, while others lack representation.
In general convergence can be seen visually on the graphs as well as in the losses graphs.

We can see that both the discriminator's loss (blue) and generator's loss (purple) are still fluctuating, indicating that the GAN is still learning and has not fully converged yet.
The continued oscillations suggest that the generator and discriminator are engaged in a continuous adversarial game, where each network is trying to outperform the other.
The losses are still fluctuating, and the divergence metrics (KL and JS) suggest that the generated distribution is not yet an exact match to the target distribution.

In summary, the GAN is improving, but additional training might be necessary to achieve a more accurate match between the real and generated data.

Question 3:

In this part, we have implemented the M1 scheme from the paper "Semi-supervised Learning with Deep Generative Models" by Kingma et al., which uses a Variational Autoencoder for feature extraction and a transductive SVM for classification. The network architecture suggested for MNIST is applied to the Fashion MNIST dataset, with results presented for 100, 600, 1000, and 3000 labelled samples.
To implement the required task, we defined a variational autoencoder (VAE) model architecture, containing encoder and decoder layers [1]. An SVM classifier with an RBF kernel is trained on the latent representations of the labeled data. We chose to work with "RBF" kernel since this kernel is the most effective when applying image classification. [2]
To make it easy to train, we have defined some hyperparameters, such as the batch size, learning rate, and number of epochs at the beginning of the code. We define both 'FashionMNIST' and 'MNIST' datasets in order to evaluate the network's performance. Both datasets are split into labeled and unlabeled sets for various numbers of labels (100, 600, 1000, and 3000). The labeled sets are balanced across classes.
The loss of VAE was calculated using binary cross-entropy for the reconstruction loss and KL divergence loss, as shown in class. The trained SVM is evaluated on the test set, and the accuracy is reported and the results are plotted.

How to train and test the model:

1. Run the provided code.
2. The model will be trained automatically, for both datasets.
3. The VAE weights and SVM model will be saved
4. The accuracies for different numbers of labels will be printed, as well as comparison between real and reconstructed images.

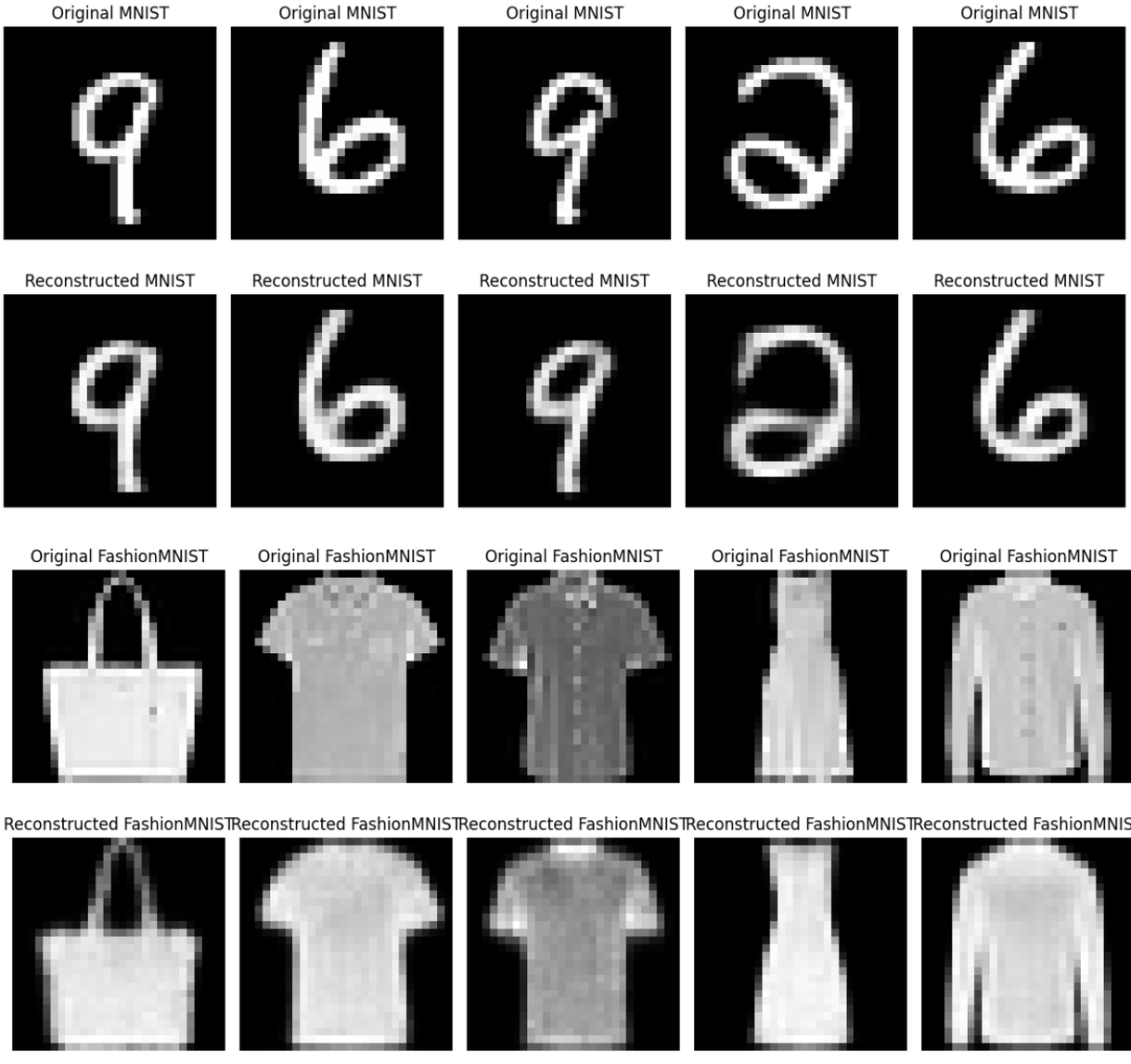
Note: The code assumes that the Fashion-MNIST / MNIST datasets will be downloaded and stored in the `data` directory. You may need to adjust the path if your dataset is located elsewhere.

After training and evaluating the model, we got the following results:

MNIST Accuracies: **[0.8285, 0.9298, 0.9402, 0.9631]**

FashionMNIST Accuracies: **[0.7025, 0.759, 0.7903, 0.8138]**

As can be seen, the model is more accurate for the MNIST dataset than the FashionMNIST.
This trend is also noticeable in the visual results, as they shown below, where the reconstructed images of the MNIST dataset are much more close to the real ones related to those of FashionMNIST:



In comparison to table 1 from the paper described above, trained on MNIST dataset, we can see that:

N	NN	CNN	TSVM	CAE	MTC	AtlasRBF	M1+TSVM	M2	M1+M2	*ours*
100	25.81	22.98	16.81	13.47	12.03	8.10 (± 0.95)	11.82 (± 0.25)	11.97 (± 1.71)	**3.33** (± 0.14)	**17.15**
600	11.44	7.68	6.16	6.3	5.13	–	5.72 (± 0.049)	4.94 (± 0.13)	**2.59** (± 0.05)	**7.02**
1000	10.7	6.45	5.38	4.77	3.64	3.68 (± 0.12)	4.24 (± 0.07)	3.60 (± 0.56)	**2.40** (± 0.02)	**5.98**
3000	6.04	3.35	3.45	3.22	2.57	–	3.49 (± 0.04)	3.92 (± 0.63)	**2.18** (± 0.04)	**3.69**

As can be seen, our results are pretty like those shown in the paper, which is reasonable as we have used M1 algorithm. We can see that for 3000 labels the results are pretty accurate (~96%). Note that those results are for MNIST, where for FashionMNIST we got less accurate results (~80% for 3000 labels).

Question 4:

Our implementation of Deep Convolutional Generative Adversarial Networks (DCGAN) and Wasserstein Generative Adversarial Networks (WGAN) leverages advanced techniques to generate high-quality images, addressing common challenges such as training stability and mode collapse. Both architectures employ a shared generator model designed to produce images, yet they differ in their discriminator models, loss functions, and training strategies, reflecting their unique approaches to overcoming GAN-related challenges.

Discriminator and Generator Models:

In DCGAN, the discriminator employs a sigmoid activation layer to classify images as real or fake, outputting probabilities. This setup follows traditional GAN architecture, focusing on binary classification.

WGAN's discriminator, termed as a critic, foregoes the sigmoid layer, instead directly outputting a scalar score that estimates the Earth Mover's distance between the real and generated image distributions. This fundamental difference allows WGAN to provide smoother gradients to the generator, facilitating more stable training.

Loss Functions:

DCGAN utilizes a binary cross-entropy loss, measuring the discriminator's ability to distinguish real from fake images and the generator's capacity to fool the discriminator.

WGAN introduces the Wasserstein loss, which offers a more stable training experience by providing meaningful gradients even when the discriminator performs optimally. To further enhance WGAN's performance, we integrated Gradient Penalty (GP) as in the relevant paper recommendation. This involves interpolating between real and fake images, calculating the gradient of the critic's scores with respect to these interpolations, and applying a penalty based on the deviation of these gradients. Adopting a lambda value of 10 ensures the effective enforcement of the Lipschitz constraint, enhancing image quality without resorting to weight clipping.

Training Process and Stability:

DCGAN's training involves alternating updates between the discriminator and generator, adhering to architectural guidelines such as stride convolutions and batch normalization to mitigate instability and improve image quality.

WGAN's training distinguishes itself by updating the critic multiple times (typically five) for every generator update. This frequent critic updating ensures a more accurate estimation of the Wasserstein distance, offering the generator more stable and accurate gradients. Such a strategy, coupled with the GP method for enforcing the Lipschitz constraint, significantly reduces the risk of training collapse and mode collapse, common issues in traditional GAN training.

Outcomes:

While DCGAN marked a significant improvement in utilizing convolutional networks for GANs, enhancing stability and image quality, WGAN has propelled the field further by introducing a novel loss function that markedly improves training stability, mitigates mode collapse, and yields higher quality and more diverse images. Additionally, WGAN's approach to training the

critic more extensively than the generator has proven critical in providing stable and meaningful gradients, further contributing to the robustness and efficacy of GAN training.

In conclusion, our implementation of DCGAN and WGAN represents a comprehensive approach to leveraging the strengths of both architectures. By carefully designing their components and tailoring the training processes, we address key challenges in GAN training, leading to the generation of high-quality, diverse images while ensuring training stability and consistency.

For generating new images:

We used trained Generator models from both DCGAN and WGAN to create new, synthetic images. First, we prepared a noise vector as an input to both DCGAN and WGAN generations.

The noise vector is essentially a batch of random numbers, which the Generators will use to produce images. The randomness ensures that each generated image is unique.

noise = torch.randn(2, 100, 1, 1, device=device)

generate 2 vectors each.

The noise vectors are passed through the DCGAN and WGAN Generator models, respectively. Each model uses its learned parameters to transform the input noise vectors into data that resembles the training dataset.

Despite both models receiving the same noise input, the images they generate will differ due to the unique weights and different architectures.

dcgan_imgs = dcgan_generator(noise)

wgan_imgs = wgan_generator(noise)

Finally, a comparison between the generated and real images was shown, in order to evaluate the success of the training.

How to train and test the model:

1. Run the provided code.
2. The model will be trained automatically, for both DCGAN and WGAN.

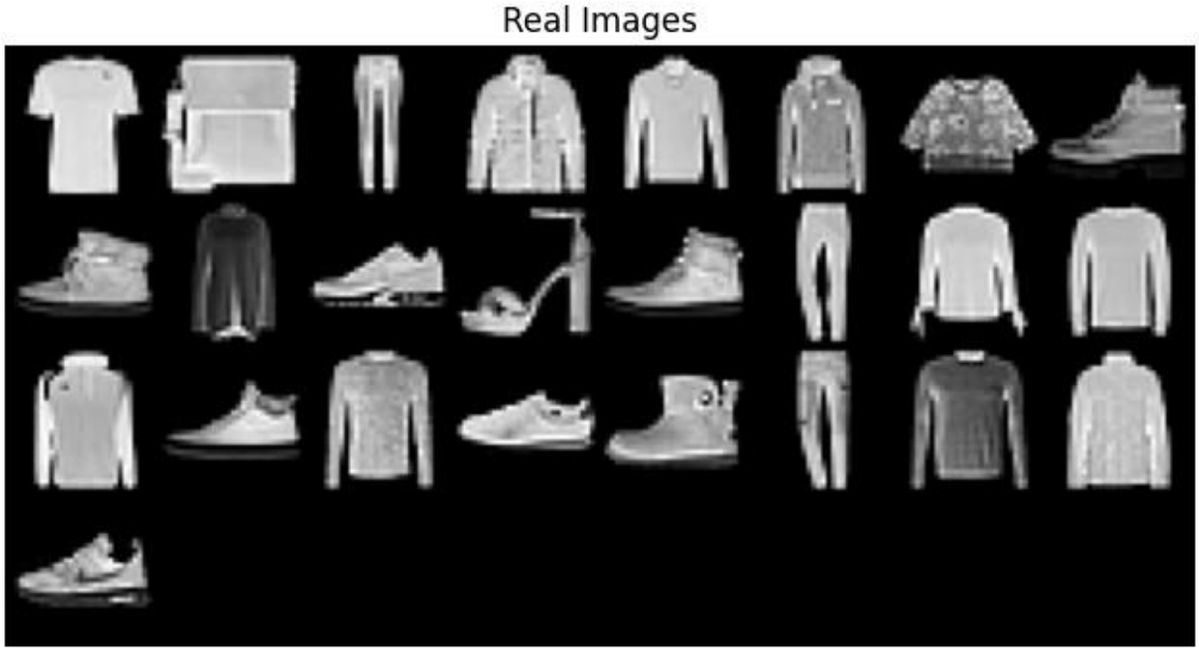
Load and generate new picture from a trained model:

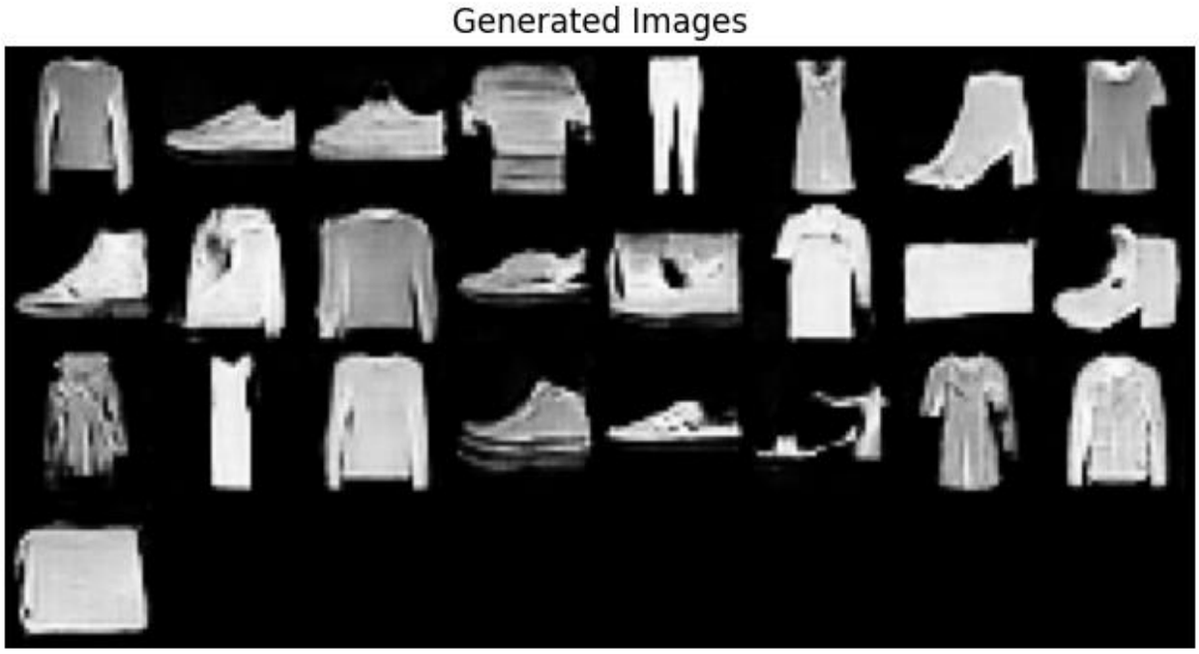
If a pre-trained model exists, the last 3 cells in our code are meant to load the model and generate new images as required – DCGAN and WGAN.

Make sure to load the model with the right name and inside the same folder as the notebook, as shown at “Load weights” part. Alternatively, change the load path to the desired location.

Results:
Both DCGAN and WGAN was trained over 80 epochs, resulting in the following results:

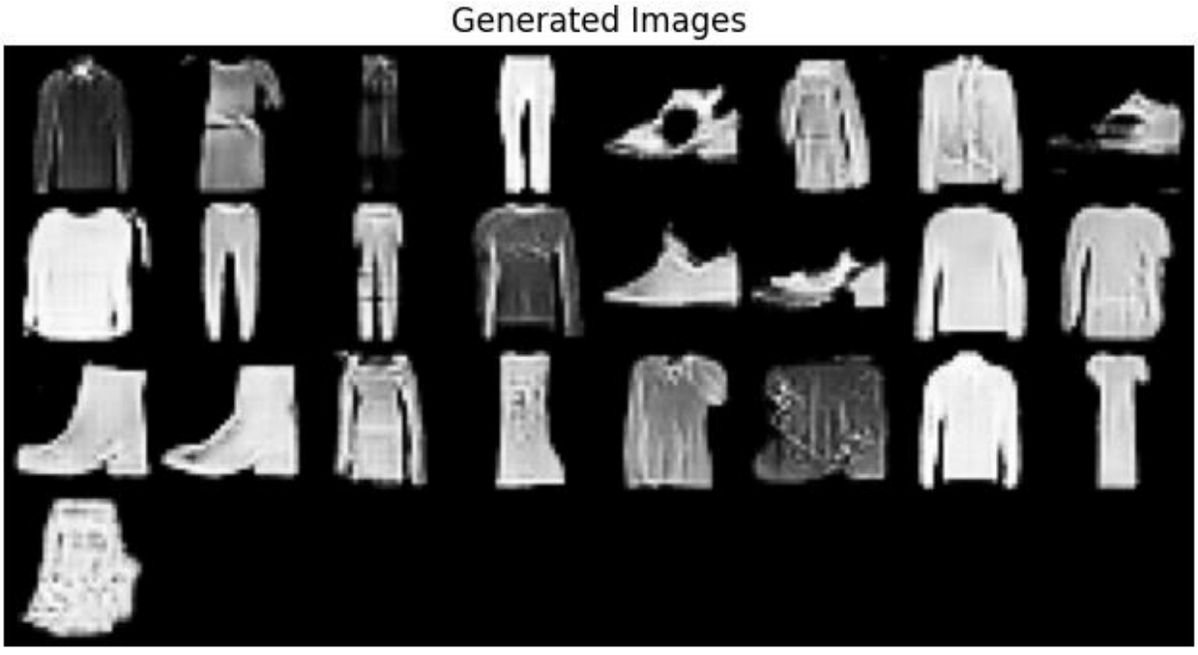
DCGAN Reconstruction Images





In our implementation, we had 2 failed convergences before having a successful one as shown above.

WGAN Reconstruction Images



In our implementation, we had 1 failed convergence before having a successful one as shown above.

For both cases we can observe a pretty accurate results, with generated images that look very close to the original dataset images. Due to limitation of GPU power, we ran only 80 epochs, and we assume that more epochs could lead to better results even further. In our experience, WGAN implementation was easier to converge related to DCGAN where we had to try few times before achieving convergence.

Bibliography:

[1] https://www.youtube.com/watch?v=9zKuYvjFFS8

[2] A. Patle and D. S. Chouhan, "SVM kernel functions for classification," *2013 International Conference on Advances in Technology and Engineering (ICATE)*, Mumbai, India, 2013, pp. 1-9, doi: 10.1109/ICAdTE.2013.6524743.

Appendix:

Q3 code:

```
# **Import Libraries**
"""

import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence
from collections import Counter
import os
from tqdm import tqdm
from torchvision import datasets, transforms
from sklearn import svm
import random

try:
    import dill as pickle
except ImportError:
    import pickle

"""# **Model Structure**

###**Args:**

*   **input_dim (int):** Dimensionality of the input data (e.g., image size).
*   **latent_dim (int):** Dimensionality of the latent space.

###**Attributes:**

*   **fc1 (nn.Linear):** First fully connected layer for encoding.
*   **fc2_mean (nn.Linear):** Layer for mean of latent space.
*   **fc2_logvar (nn.Linear):** Layer for log variance of latent space.
*   **fc3 (nn.Linear):** First fully connected layer for decoding.
*   **fc4 (nn.Linear):** Final layer for reconstructing input data.
```

```
###**Methods:**

*   **encode(x):** Encodes input data to mean and log variance of latent space.
*   **reparameterize(mean, logvar):** Samples from the latent space using reparameterization trick, enabling backpropagation
*   **decode(z):** Decodes latent representation to reconstruct input data.
*   **forward(x):** Computes reconstructed data, mean, and log variance.
"""

# Define the VAE model
class VAE(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(VAE, self).__init__()
        self.input_dim = input_dim
        self.latent_dim = latent_dim

        # Encoder layers
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 256),
            nn.ReLU(),
            nn.Linear(256, 2 * latent_dim)  # Output mean and log variance
        )

        # Decoder layers
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 64 * 7 * 7),
            nn.ReLU(),
            nn.Unflatten(1, (64, 7, 7)),
            nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 1, kernel_size=4, stride=2, padding=1),
            nn.Sigmoid()  # Use sigmoid for Bernoulli distribution
        )

    def encode(self, x):
        h = self.encoder(x)
        mean, logvar = h[:, :self.latent_dim], h[:, self.latent_dim:]
        return mean, logvar

    def reparameterize(self, mean, logvar):
```

```
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        z = mean + eps * std
        return z

    def decode(self, z):
        x = self.decoder(z)
        return x

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

"""# **Params**"""

# Hyperparameters
batchSize = 512
learningRate = 1e-3
epochs = 100
labelsNum = [100, 600, 1000, 3000]
accuracies_mnist = []
accuracies_fashion_mnist = []
inputDim = 784 # MNIST image size (28x28)
latentDim = 50

# Set Models
vaeMNIST = VAE(inputDim, latentDim)
vaeFashionMNIST = VAE(inputDim, latentDim)

optimizer_mnist = optim.Adam(vaeMNIST.parameters(), lr = learningRate)
optimizer_fashion_mnist = optim.Adam(vaeFashionMNIST.parameters(), lr =
learningRate)

# Check if GPU is available or not
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")

"""# **Fashion MNIST dataset Loading**

Load both FashionMNIST and MNIST datasets
"""

transform = transforms.Compose([transforms.ToTensor()])

# Define the data sets
```

```
train_dataset_mnist = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
test_dataset_mnist = datasets.MNIST(root='./data', train=False, download=True,
transform=transform)

train_dataset_fashion_mnist = datasets.FashionMNIST(root='./data', train=True,
download=True, transform=transform)
test_dataset_fashion_mnist = datasets.FashionMNIST(root='./data', train=False,
download=True, transform=transform)

# Define the data loaders
test_loader_mnist = DataLoader(test_dataset_mnist, batch_size=batchSize)
test_loader_fashion_mnist = DataLoader(test_dataset_fashion_mnist,
batch_size=batchSize)

"""# **Functions**"""

def vaeLoss(recon_data, data, mu, logvar):

  # Reconstruction Loss
  recon_loss = F.binary_cross_entropy(recon_data, data, reduction = 'sum')

  # KL Divergence Loss
  kl_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - torch.exp(logvar))

  #  Total Loss
  loss = recon_loss + kl_loss

  return loss


def train_vae_svm(vae, train_dataset, test_dataset, test_loader, device,
epochs, batchSize, labelsNum, accuracies, optimizer):

    for n in labelsNum:
        # Select an equal number of labled examples from each class
        labeled = []
        for i in range(10):
            label_idx = np.where(train_dataset.targets == i)[0]
            labeled.extend(list(np.random.choice(label_idx, n//10,
replace=False)))

        # Select an equal number of unlabled examples from each class
        unlabeled = list(set(range(len(train_dataset.data))) - set(labeled))

        # Create labeled and unlabeled data loaders
        labeled_loader = DataLoader(train_dataset, batch_size=batchSize,
sampler=torch.utils.data.SubsetRandomSampler(labeled))
```

```
        unlabeled_loader = DataLoader(train_dataset, batch_size=batchSize,
sampler=torch.utils.data.SubsetRandomSampler(unlabeled))

        # VAE Training
        for epoch in range(epochs):
            vae.train()
            vae.to(device)
            trainLoss = 0
            for batch_i, (data, _) in enumerate(labeled_loader):
                data = data.to(device)
                data = data.reshape(-1, 1, 28, 28)
                optimizer.zero_grad()
                recon_data, mu, logvar = vae(data)
                loss = vaeLoss(recon_data, data, mu, logvar)
                loss.backward()
                trainLoss += loss.item()
                optimizer.step()

            print(f'Training Epoch {epoch+1}/{epochs}, Loss:
{trainLoss/(batch_i+1):.4f}')

        # Get latent representation
        vae.eval()

        testLoss = 0
        latentLabled = []
        labledTarget = []
        latentTest = []
        testTarget = []

        for inputs, targets in tqdm(labeled_loader):
            with torch.no_grad():
                inputs, targets = inputs.to(device), targets.to(device)
                # Get the latent representation
                inputs = inputs.reshape(-1, 1, 28, 28)
                latentData, _ = vae.encode(inputs)
                latentLabled.append(latentData)
                labledTarget.append(targets)
        latentLabled = torch.cat(latentLabled, dim=0)
        labledTarget = torch.cat(labledTarget)

        for data, targets in tqdm(test_loader):
            with torch.no_grad():
                data, targets = data.to(device), targets.to(device)
                # Get the latent representation
                data = data.reshape(-1, 1, 28, 28)
                latentData, _ = vae.encode(data)
                latentTest.append(latentData)
```

```
                testTarget.append(targets)
        latentTest = torch.cat(latentTest, dim=0)
        testTarget = torch.cat(testTarget)

        # Train the SVM classifier (using RBF kernel)
        clf = svm.SVC(kernel='rbf')
        clf.fit(latentLabled.cpu().numpy(), labledTarget.cpu().numpy())

        # Evaluate on the test set
        test_acc = clf.score(latentTest.cpu().numpy(), testTarget.cpu())
        accuracies.append(test_acc)

        vae.train()
        print(f'Number of labels: {n}, Test accuracy: {test_acc:.4f}')

    print(f'Accuracies: {accuracies}')
    return clf

"""# **Train & Save models**"""

# Train and test MNIST
MNIST_clf = train_vae_svm(vaeMNIST, train_dataset_mnist, test_dataset_mnist,
test_loader_mnist, device, epochs, batchSize, labelsNum, accuracies_mnist,
optimizer_mnist)

# Save the SVM model and VAE weights
torch.save(vaeMNIST.state_dict(), 'vae_MNIST_weights.pth')
with open('svm_MNIST_model.pkl', 'wb') as f:
    pickle.dump(MNIST_clf, f)

# Train and test FashionMNIST
Fashion_clf = train_vae_svm(vaeFashionMNIST, train_dataset_fashion_mnist,
test_dataset_fashion_mnist, test_loader_fashion_mnist, device, epochs,
batchSize, labelsNum, accuracies_fashion_mnist, optimizer_fashion_mnist)

torch.save(vaeFashionMNIST.state_dict(), 'vae_fashionMNIST_weights.pth')
with open('svm_fashionMNIST_model.pkl', 'wb') as f:
    pickle.dump(Fashion_clf, f)

"""#**Plot comparison graph**"""

print('**********************************************************************
****')
print(f'MNIST Accuracies: {accuracies_mnist}')
print(f'FashionMNIST Accuracies: {accuracies_fashion_mnist}')
print('**********************************************************************
****')
```

```
# Plot the before and after reconstructed images for MNIST
vaeMNIST.eval()
with torch.no_grad():
    # Get a random batch of test data
    random_idx = random.randint(0, len(test_loader_mnist) - 1)
    for i, (sample_data, _) in enumerate(test_loader_mnist):
        if i == random_idx:
            sample_data = sample_data.to(device)
            sample_data = sample_data.reshape(-1, 1, 28, 28)
            reconstructed_data, _, _ = vaeMNIST(sample_data)
            break

# Plot the original and reconstructed MNIST images
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(12, 6))

for i, ax in enumerate(axes.flatten()):
    if i < 5:
        ax.imshow(sample_data[i, 0].cpu().numpy(), cmap='gray')
        ax.set_title('Original MNIST')
        ax.axis('off')
    else:
        ax.imshow(reconstructed_data[i - 5, 0].cpu().detach().numpy(),
cmap='gray')
        ax.set_title('Reconstructed MNIST')
        ax.axis('off')

plt.tight_layout()
plt.show()

# Plot the before and after reconstructed images for FashionMNIST
vaeFashionMNIST.eval()
with torch.no_grad():
    # Get a random batch of test data
    random_idx = random.randint(0, len(test_loader_fashion_mnist) - 1)
    for i, (sample_data, _) in enumerate(test_loader_fashion_mnist):
        if i == random_idx:
            sample_data = sample_data.to(device)
            sample_data = sample_data.reshape(-1, 1, 28, 28)
            reconstructed_data, _, _ = vaeFashionMNIST(sample_data)
            break

# Plot the original and reconstructed FashionMNIST images
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(12, 6))

for i, ax in enumerate(axes.flatten()):
    if i < 5:
        ax.imshow(sample_data[i, 0].cpu().numpy(), cmap='gray')
        ax.set_title('Original FashionMNIST')
```

```
        ax.axis('off')
    else:
        ax.imshow(reconstructed_data[i - 5, 0].cpu().detach().numpy(),
cmap='gray')
        ax.set_title('Reconstructed FashionMNIST')
        ax.axis('off')

plt.tight_layout()
plt.show()
```

Q4 Code:

```
import torch
from torch import nn, optim
from torchvision import datasets, transforms, utils
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
import torchvision.utils as vutils
import numpy as np
from torchvision.datasets import FashionMNIST
import matplotlib.pyplot as plt
import torchvision

"""Set up data loading for MNIST"""

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

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

"""define generator for WGAN - The generator needs to output 28x28 images."""

class WGAN_Generator(nn.Module):
    def __init__(self):
        super(WGAN_Generator, self).__init__()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(100, 256, 7, 1, 0, bias=False),  # Output: 7x7
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),  # Output:
14x14
            nn.BatchNorm2d(128),
```

```
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),  # Output: 28x28
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 1, 3, 1, 1, bias=False),  # Keeps output:
28x28
            nn.Tanh()  # Output in [-1, 1] to match the normalized dataset
        )

    def forward(self, input):
        output = self.model(input)
        return output

"""define generator for DCGAN - The generator needs to output 28x28 images."""

class DCGAN_Generator(nn.Module):
    def __init__(self):
        super(DCGAN_Generator, self).__init__()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(100, 256, 7, 1, 0, bias=False),  # Output: 7x7
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),  # Output:
14x14
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),  # Output: 28x28
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 1, 3, 1, 1, bias=False),  # Keeps output:
28x28
            nn.Tanh()  # Output in [-1, 1] to match the normalized dataset
        )

    def forward(self, input):
        output = self.model(input)
        return output

"""define discriminator for WGAN for MNIST 28x28"""

class WGAN_Discriminator(nn.Module):
    def __init__(self):
        super(WGAN_Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 64, 4, 2, 1, bias=False),  # Input: 28x28 -> Output:
14x14
            nn.LeakyReLU(0.2, inplace=True),
```

```
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),  # Input: 14x14 ->
Output: 7x7
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, 3, 2, 1, bias=False),  # Input: 7x7 -> Output:
4x4
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 1, 4, 1, 0, bias=False),  # Input: 4x4 -> Output:
1x1
        )

    def forward(self, input):
        output = self.model(input)
        return output.view(-1, 1).squeeze(1)

"""define discriminator for DCGAN for MNIST 28x28"""

class DCGAN_Discriminator(nn.Module):
    def __init__(self):
        super(DCGAN_Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 64, 4, 2, 1, bias=False),  # Input: 28x28 -> Output:
14x14
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),  # Input: 14x14 ->
Output: 7x7
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, 3, 2, 1, bias=False),  # Input: 7x7 -> Output:
4x4
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 1, 4, 1, 0, bias=False),  # Input: 4x4 -> Output:
1x1
            nn.Sigmoid()
        )

    def forward(self, input):
        output = self.model(input)
        return output.view(-1, 1).squeeze(1)

"""Define Gradient Penalty Calculation"""

def compute_gradient_penalty(D, real_samples, fake_samples):
    """Calculates the gradient penalty loss for WGAN GP"""
    # Random weight term for interpolation between real and fake samples
    alpha = torch.rand((real_samples.size(0), 1, 1, 1), device=device)
```

```
    # Get random interpolation between real and fake samples
    interpolates = (alpha * real_samples + ((1 - alpha) *
fake_samples)).requires_grad_(True)
    d_interpolates = D(interpolates)
    fake = torch.ones(d_interpolates.size(), requires_grad=False,
device=device)

    # Get gradient w.r.t. interpolates
    gradients = torch.autograd.grad(
        outputs=d_interpolates,
        inputs=interpolates,
        grad_outputs=fake,
        create_graph=True,
        retain_graph=True,
        only_inputs=True,
    )[0]

    gradients = gradients.view(gradients.size(0), -1)
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean()
    return gradient_penalty

"""show the images"""

def show_generated_imgs(imgs):
    plt.figure(figsize=(8,8))
    plt.axis("off")
    plt.title("Generated Images")
    plt.imshow(np.transpose(vutils.make_grid(imgs[:25], padding=2,
normalize=True).cpu(),(1,2,0)))
    plt.show()

def show_real_imgs(imgs):
  plt.figure(figsize=(8,8))
  plt.axis("off")
  plt.title("Real Images")
  plt.imshow(np.transpose(vutils.make_grid(imgs[:25], padding=2,
normalize=True).cpu(),(1,2,0)))
  plt.show()

"""claculate the discrimanator accuracy.
how well the discriminator distinguishes between real and fake images. This is
done by measuring the percentage of real images correctly identified as real
and fake images correctly identified as fake.
"""

def discriminator_accuracy(real_output, fake_output):
    real_correct = torch.sum(real_output > 0.5).item()
    fake_correct = torch.sum(fake_output < 0.5).item()
```

```
    total = real_output.size(0) + fake_output.size(0)
    return (real_correct + fake_correct) / total

device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")
dcgan_generator = DCGAN_Generator().to(device)
dcgan_discriminator = DCGAN_Discriminator().to(device)
wgan_generator = WGAN_Generator().to(device)
wgan_discriminator = WGAN_Discriminator().to(device)

"""WGAN-GP Training Function"""

def train_wgan_gp(device, train_loader, generator, discriminator, epochs=15,
lr=0.001, lambda_gp=10, n_critic=5):

    optim_d = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.9))
    optim_g = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.9))

    for epoch in range(epochs):
        for i, (imgs, _) in enumerate(train_loader):
            real_imgs = imgs.to(device)
            z = torch.randn(imgs.size(0), 100, 1, 1, device=device)
            fake_imgs = generator(z)

            # Train Discriminator
            optim_d.zero_grad()
            real_validity = discriminator(real_imgs)
            fake_validity = discriminator(fake_imgs)
            gradient_penalty = compute_gradient_penalty(discriminator,
real_imgs.data, fake_imgs.data)
            d_loss = -torch.mean(real_validity) + torch.mean(fake_validity) +
lambda_gp * gradient_penalty
            d_loss.backward()
            optim_d.step()

            # Train Generator
            if i % n_critic == 0:
                optim_g.zero_grad()
                fake_imgs = generator(z)
                fake_validity = discriminator(fake_imgs)
                g_loss = -torch.mean(fake_validity)
                g_loss.backward()
                optim_g.step()

            if i % 100 == 0:
                acc = discriminator_accuracy(real_validity, fake_validity)
```

```
                print(f"[Epoch {epoch}/{epochs}] [Batch
{i}/{len(train_loader)}] [D loss: {d_loss.item()}] [G loss: {g_loss.item()}]
[D Acc: {acc}]")
            show_real_imgs(real_imgs)
            show_generated_imgs(fake_imgs)
        torch.save(generator.state_dict(), 'wgan_generator_model_weights.pth')
        torch.save(discriminator.state_dict(),
'wgan_discriminator_model_weights.pth')

"""DCGAN Training Function (with Gradient Penalty for Illustration)"""

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

def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

def train_dcgan_gp(device, train_loader, generator, discriminator, epochs=80,
lr=0.0001, lambda_gp=10):
    # Initialize weights
    generator.apply(weights_init)
    discriminator.apply(weights_init)

    optim_d = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.9))
    optim_g = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.9))

    for epoch in range(epochs):
        for i, (imgs, _) in enumerate(train_loader):
            real_imgs = imgs.to(device)
            z = torch.randn(imgs.size(0), 100, 1, 1, device=device)
            fake_imgs = generator(z)

            # Train Discriminator
            optim_d.zero_grad()
            real_validity = discriminator(real_imgs)
            fake_validity = discriminator(fake_imgs.detach())
            real_loss =
torch.nn.functional.binary_cross_entropy_with_logits(real_validity,
torch.ones_like(real_validity))
            fake_loss =
torch.nn.functional.binary_cross_entropy_with_logits(fake_validity,
torch.zeros_like(fake_validity))
```

```
            d_loss = real_loss + fake_loss

            if lambda_gp > 0:  # Optionally add gradient penalty
                gradient_penalty = compute_gradient_penalty(discriminator,
real_imgs.data, fake_imgs.data)
                d_loss += lambda_gp * gradient_penalty

            d_loss.backward()
            optim_d.step()

            # Train Generator
            optim_g.zero_grad()
            fake_validity = discriminator(fake_imgs)
            g_loss =
torch.nn.functional.binary_cross_entropy_with_logits(fake_validity,
torch.ones_like(fake_validity))
            g_loss.backward()
            optim_g.step()

            if i % 100 == 0:
                acc = discriminator_accuracy(real_validity,
fake_validity.detach())
                print(f"[Epoch {epoch}/{epochs}] [Batch
{i}/{len(train_loader)}] [D loss: {d_loss.item()}] [G loss: {g_loss.item()}]
[D Acc: {acc}]")

        show_real_imgs(real_imgs)
        show_generated_imgs(fake_imgs)
    torch.save(generator.state_dict(), 'dcgan_generator_model_weights.pth')
    torch.save(discriminator.state_dict(),
'dcgan_discriminator_model_weights.pth')

dcgan_generator.apply(weights_init)
dcgan_discriminator.apply(weights_init)
train_dcgan_gp(device, train_loader, dcgan_generator, dcgan_discriminator,
epochs=80, lr=0.0001, lambda_gp=10)

train_wgan_gp(device, train_loader, wgan_generator, wgan_discriminator,
epochs=80, lr=0.001, lambda_gp=10)

"""Load Trained Models"""

device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")
dcgan_generator = DCGAN_Generator().to(device)
dcgan_discriminator = DCGAN_Discriminator().to(device)
wgan_generator = WGAN_Generator().to(device)
wgan_discriminator = WGAN_Discriminator().to(device)
```

```
# Load weights
wgan_generator.load_state_dict(torch.load('wgan_generator_model_weights.pth',
map_location=device))
wgan_discriminator.load_state_dict(torch.load('wgan_discriminator_model_weight
s.pth', map_location=device))
dcgan_generator.load_state_dict(torch.load('dcgan_generator_model_weights.pth'
, map_location=device))
dcgan_discriminator.load_state_dict(torch.load('dcgan_discriminator_model_weig
hts.pth', map_location=device))

dcgan_generator.eval()
wgan_generator.eval()

"""Generate Images"""

with torch.no_grad():
    noise = torch.randn(2, 100, 1, 1, device=device)  # Generate 2 images
    dcgan_imgs = dcgan_generator(noise)
    wgan_imgs = wgan_generator(noise)

"""Select Real Images for Comparison"""

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
# Loading the dataset
dataset = FashionMNIST(root='./data', train=False, download=True,
transform=transform)

# DataLoader with batch_size=1 to iterate images one by one
loader = DataLoader(dataset, batch_size=1, shuffle=True)

# Initialize a dictionary to store images for each label
label_images = {i: [] for i in range(10)}  # 10 labels in FashionMNIST

# Number of images to collect for each label
images_per_label = 2

for image, label in loader:
    label = label.item()  # Convert label tensor to int
    if len(label_images[label]) < images_per_label:
        label_images[label].append(image)

    # Check if we have collected enough images for each label
    if all(len(images) == images_per_label for images in
label_images.values()):
```

```
        break

with torch.no_grad():
    noise = torch.randn(2, 100, 1, 1, device=device)  # Generate 2 images each
    dcgan_imgs = dcgan_generator(noise)
    wgan_imgs = wgan_generator(noise)

# Display function
def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis('off')
    plt.show()

# Display Generated Images by DCGAN and WGAN
print("Generated by DCGAN:")
imshow(torchvision.utils.make_grid(dcgan_imgs.cpu(), nrow=2))
print("Generated by WGAN:")
imshow(torchvision.utils.make_grid(wgan_imgs.cpu(), nrow=2))


for label, images in label_images.items():
    print(f"Real Images, Label: {label}")
    imshow(torchvision.utils.make_grid(torch.cat(images, dim=0),
nrow=images_per_label))
```