DS675 Project - Milestone 2

Generative Adversarial Networks (GANs)

Team Members

1. **Shamael Mariam Saigal - ms3238@njit.edu**
2. **Bhavya Podapati - bp536@njit.edu**
3. **Naga Anjaneyulu Karumuri - nk643@njit.edu**

Link to drive folder:
https://drive.google.com/drive/folders/14G_jHwu8NKzvyQE6ZO_nQt51IfNKINNN?usp=sharing

It contains all files - videos, notebooks, etc.

Introduction to GANs
Generative Adversarial Networks[1] (GANs) represent a groundbreaking approach in generative modeling, leveraging deep learning methods like convolutional neural networks. This technique, which falls under unsupervised learning, focuses on understanding and discovering patterns in data, allowing models to generate new examples that could be part of the original dataset.

GANs are uniquely structured, comprising two sub-models: the generator and the discriminator. These models are trained together in a zero-sum, adversarial game. The ultimate goal is to refine the generator to the point where the discriminator is fooled approximately 50% of the time, indicating the generator's proficiency in creating plausible data instances.

GAN Loss Functions
The GAN architecture can use various loss functions, but the most common are the minimax GAN loss and the non-saturating loss. We also considered alternate loss functions like the least squares and Wasserstein [2] loss functions. However, our extensive evaluations suggested minor differences in these approaches when other factors like computational budget and hyperparameters were constant.

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right]$$

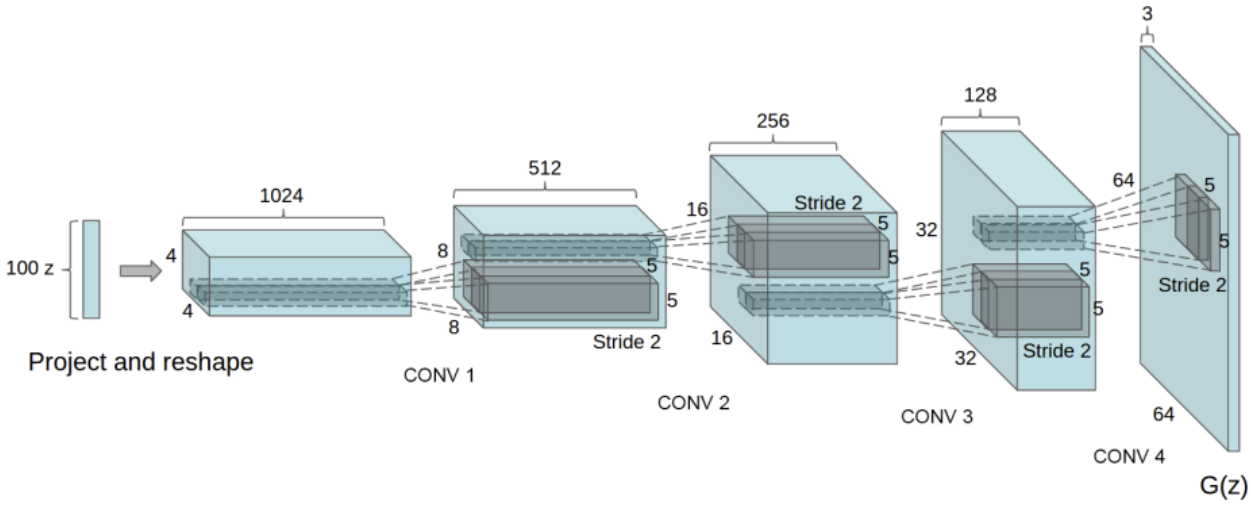
Challenges with GAN Loss

GANs can be difficult to comprehend, especially regarding their loss functions. The discriminator directly learns from real and generated images, classifying them as real or fake. The generator, however, learns indirectly through the discriminator's feedback.

Standard GAN Loss Functions
We explored both the Minimax GAN Loss and the Non-Saturating GAN Loss. The former involves a min-max strategy, optimizing the generator and discriminator's losses simultaneously. The latter addresses the saturation problem by adjusting the generator's loss function.

Alternate GAN Loss Functions
Our research also delved into least squares and Wasserstein loss[2] functions. We noticed that binary cross-entropy loss could lead to vanishing gradients, particularly when generated images starkly differed from real images. The Wasserstein loss, based on minimizing the distance between actual and predicted probability distributions, offered an alternative perspective.

DCGAN - Deep Convolutional GAN
DCGAN[3], a variant focusing on deep convolutional networks for both generator and discriminator, proved effective in our experiments. Key features included convolutional layers for capturing spatial patterns, batch normalization for stability, and the absence of fully connected layers to better capture spatial dependencies.



The Fashion MNIST Dataset
Fashion MNIST [4], an alternative to traditional MNIST, provided a more challenging benchmark with its grayscale images of various fashion items. It served as an excellent testbed for our GAN experiments, especially for tasks like image classification.

Models

The models described here are central components of a Generative Adversarial Network (GAN), specifically the Discriminator and Generator.

Discriminator Model
The Discriminator is a neural network that classifies input data as real or fake. This particular model is structured as follows:

Input Layer: It takes an input of size 784. This likely corresponds to flattened images (28x28 pixels).
Hidden Layers: It consists of two hidden layers with 512 and 256 units, respectively. Each layer uses a LeakyReLU activation function with a negative slope of 0.2. This choice of activation function helps prevent the problem of vanishing gradients during training.
Output Layer: The final layer is a single unit with a Sigmoid activation function. This structure is typical for binary classification tasks, outputting a probability indicating whether the input data is real or generated (fake).
Generator Model
The Generator is a network that creates new data instances:

Input Layer: It takes a 100-dimensional input, typically a random noise vector, which acts as a seed for data generation.
Hidden Layers: Similar to the Discriminator, it contains three hidden layers with 256, 512, and 1024 units. These layers also use the LeakyReLU activation function with a negative slope of 0.2, aiding in effective gradient propagation.
Output Layer: The final output is a layer with 784 units followed by a Tanh activation function. The Tanh function outputs values between -1 and 1, consistent with the pre-processed input data of the Discriminator. The output is reshaped into a 28x28 image, representing a generated (fake) image.
Both models play a pivotal role in the GAN framework, where the Generator learns to produce increasingly realistic data. At the same time, the Discriminator becomes better at distinguishing real data from generated data. This adversarial process continues until the Generator produces highly realistic data.

```
generator
✓ 0.0s
```

```
Generator(
  (model): Sequential(
    (0): Linear(in_features=100, out_features=256, bias=True)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Linear(in_features=256, out_features=512, bias=True)
    (3): LeakyReLU(negative_segment=0.2)
    (4): Linear(in_features=512, out_features=1024, bias=True)
    (5): LeakyReLU(negative_slope=0.2)
    (6): Linear(in_features=1024, out_features=784, bias=True)
    (7): Tanh()
  )
)
```

```
discriminator
✓ 0.0s
```

```
Discriminator(
  (model): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Linear(in_features=512, out_features=256, bias=True)
    (3): LeakyReLU(negative_slope=0.2)
    (4): Linear(in_features=256, out_features=1, bias=True)
    (5): Sigmoid()
  )
)
```

DCGAN

Discriminator

Layer Composition: Comprises convolutional layers, batch normalization, and LeakyReLU activations.
Convolutional layers are used to process image data effectively.
Batch normalization stabilizes training.
LeakyReLU prevents vanishing gradients.
Output Layer: Ends with a linear layer and a sigmoid activation, suitable for binary classification (real vs. fake images).

Generator

Input: Takes a 100-dimensional noise vector.
Layer Composition: This consists of linear layers, batch normalization, ReLU activations, and convolutional transpose layers.

- Linear layers and unflatten operations transform the noise vector into a suitable format for convolutional processing.
- Convolutional transpose layers (deconvolutional layers) are used to upscale the data back to image dimensions.
- Tanh activation in the output layer normalizes the output to the range [-1, 1], aligning with the pre-processed input images.\

Observations from Training Logs

Variation in Loss: Both the discriminator and generator exhibit fluctuations in loss across epochs, which is typical in GAN training due to the adversarial nature of the process.
Accuracy Trends: The real and fake accuracies for the discriminator show how its performance in distinguishing real from fake images evolves. The goal is typically to maintain these accuracies around 0.5, indicating that the discriminator is equally likely to classify real and fake images correctly. However, varying accuracies across epochs are common as the network learns.

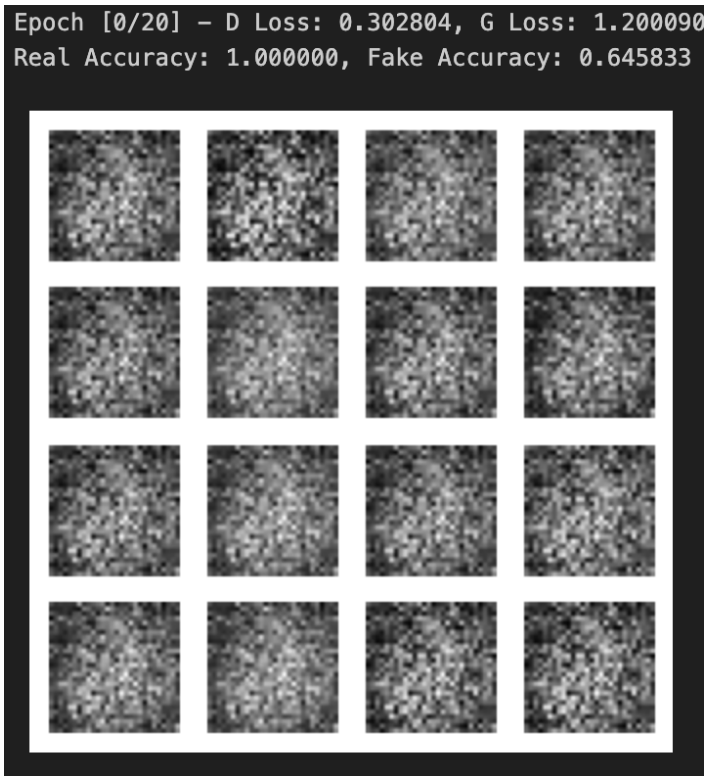
Conclusion
The described DCGAN architecture effectively uses convolutional approaches in both the generator and discriminator, aligning with contemporary practices in deep learning for image data.
The training process indicates active learning, with the generator improving over time in creating convincing images. The discriminator's fluctuating accuracies suggest it's continually adapting to the improving generator.

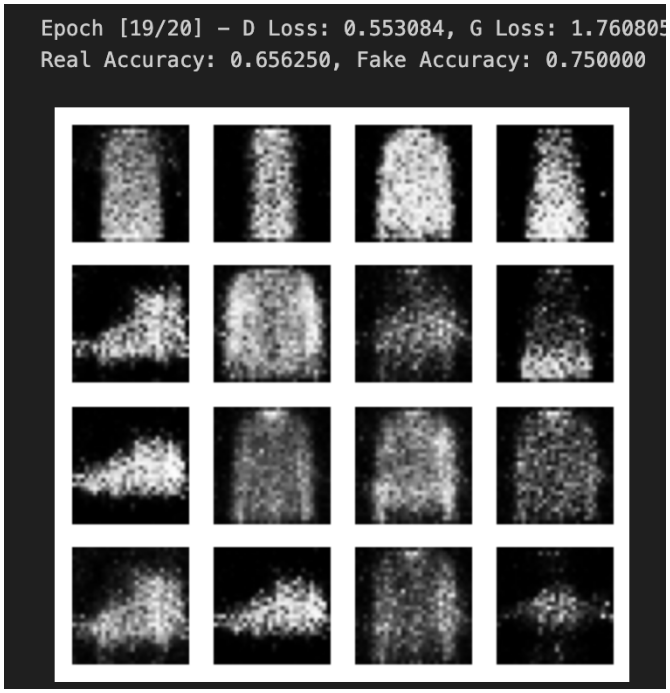
```
Discriminator(
  (layer1): Sequential(
    (0): Conv2d(1, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2)
  )
  (layer2): Sequential(
    (0): Conv2d(64, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2)
  )
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (dropout): Dropout(p=0.3, inplace=False)
  (final_layer): Sequential(
    (0): Linear(in_features=6272, out_features=1, bias=True)
    (1): Sigmoid()
  )
)
```

```
Generator(
  (model): Sequential(
    (0): Linear(in_features=100, out_features=12544, bias=True)
    (1): BatchNorm1d(12544, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Unflatten(dim=1, unflattened_size=(256, 7, 7))
    (4): ConvTranspose2d(256, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): ConvTranspose2d(128, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1))
    (8): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(64, 1, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (11): Tanh()
  )
)
```

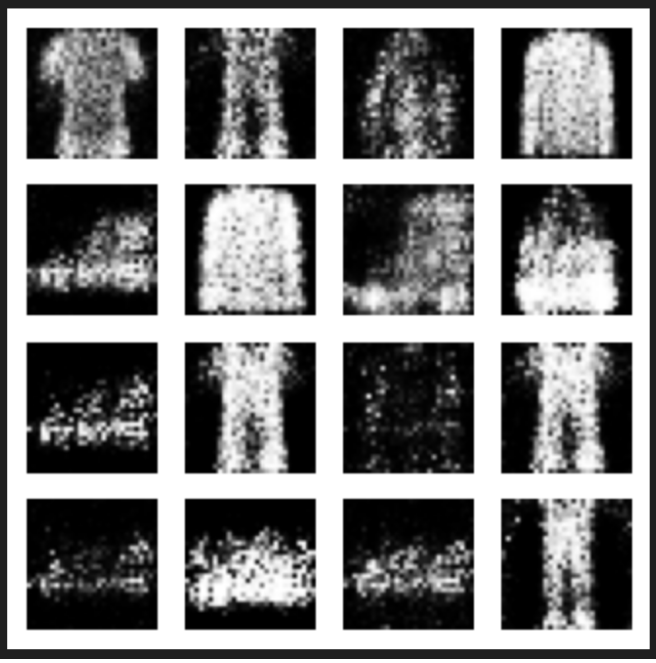
Results

Normal Gan



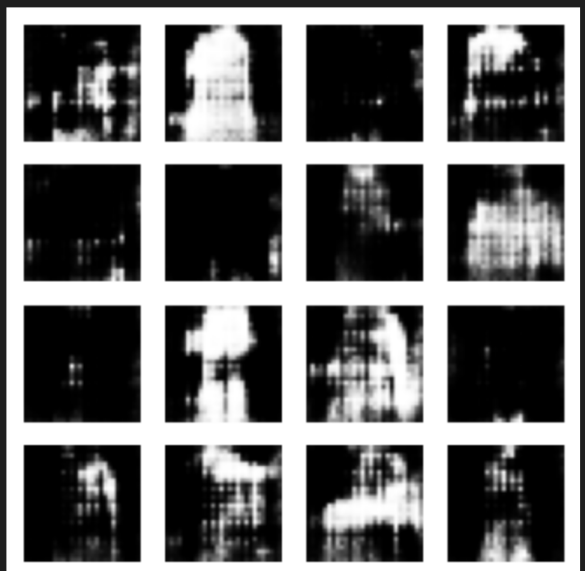


```
Epoch [39/40] - D Loss: 0.539806, G Loss: 1.150565
Real Accuracy: 0.760417, Fake Accuracy: 0.645833
```



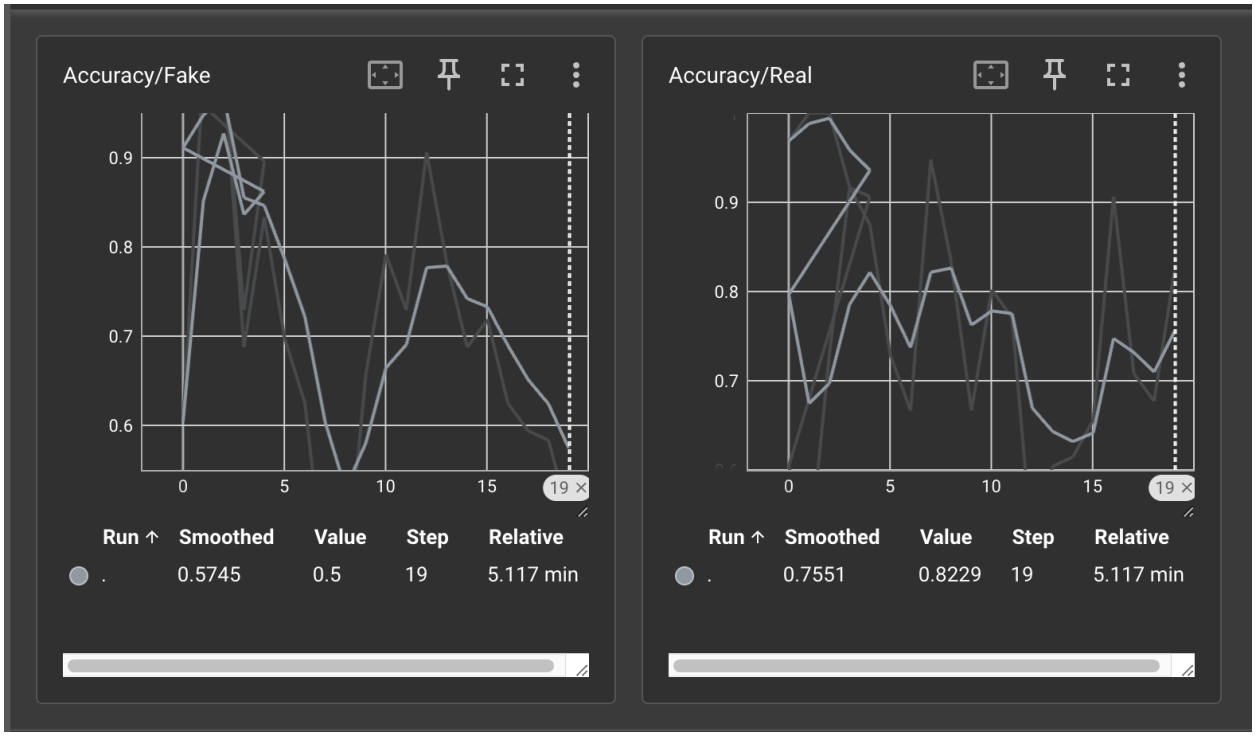
DCGAN

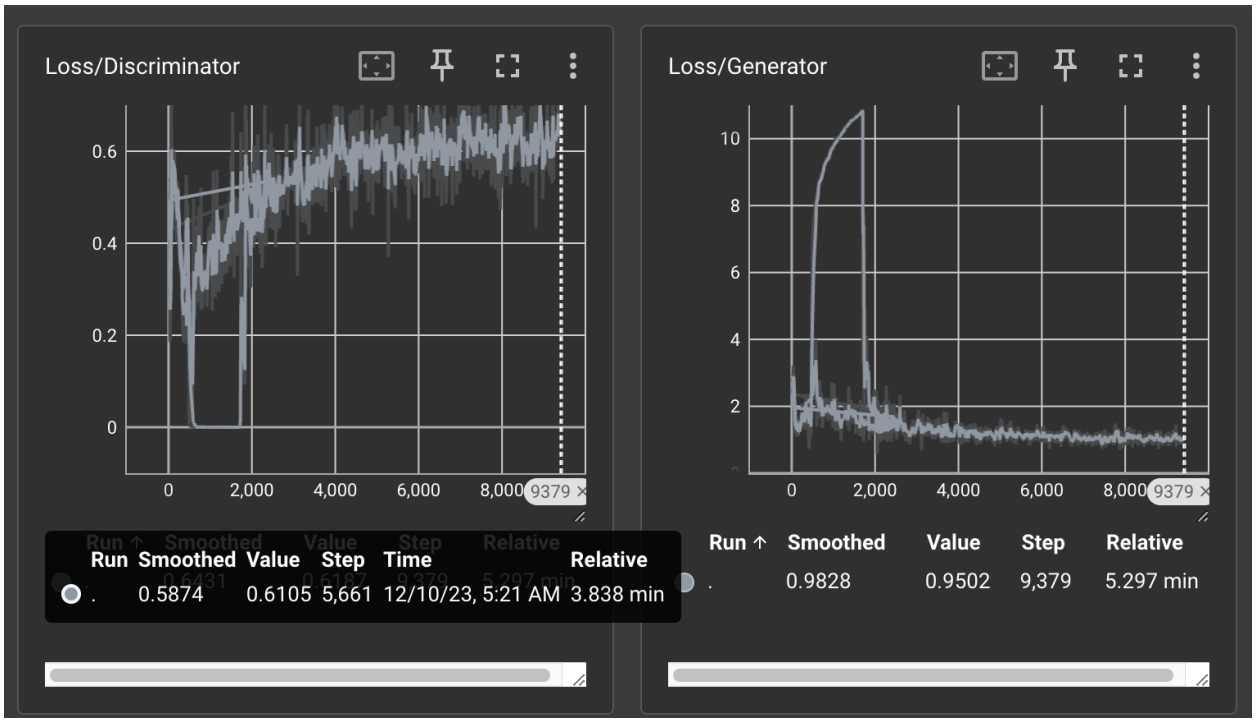
```
Epoch [0/20] - D Loss: 0.469851, G Loss: 2.354015
Real Accuracy: 0.604167, Fake Accuracy: 0.979167
```



```
Epoch [5/20] - D Loss: 0.513121, G Loss: 1.118069
Real Accuracy: 0.729167, Fake Accuracy: 0.697917
```

```
Epoch [19/20] - D Loss: 0.618739, G Loss: 0.950217
Real Accuracy: 0.822917, Fake Accuracy: 0.500000
```


Accuracy/Fake
0.9
0.8
0.7
0.6
0
5
10
15
19
Run ↑ Smoothed Value Step Relative
. 0.5745 0.5 19 5.117 min
Accuracy/Real
0.9
0.8
0.7
0
5
10
15
19
Run ↑ Smoothed Value Step Relative
. 0.7551 0.8229 19 5.117 min


Loss/Discriminator
0.6
0.4
0.2
0
0
2,000
4,000
6,000
8,000
9379
Run Smoothed Value Step Time Relative
. 0.5874 0.6105 5,661 12/10/23, 5:21 AM 3.838 min
Loss/Generator
10
8
6
4
2
0
2,000
4,000
6,000
8,000
9379
Run ↑ Smoothed Value Step Relative
. 0.9828 0.9502 9,379 5.297 min

Conclusion

Our project demonstrated GANs' remarkable ability to generate realistic examples in various domains, particularly in image synthesis. The adversarial training approach, coupled with thoughtful choices in loss functions and network architectures, played a crucial role in achieving impressive results. The Fashion MNIST dataset, in particular, provided a challenging yet accessible platform for benchmarking our models. This project not only underscored the potential of GANs in generative modeling but also highlighted the importance of selecting appropriate datasets, architectures, and loss functions for successful model training and evaluation.

References

[1] I. J. Goodfellow *et al.*, “Generative Adversarial Networks.” arXiv, Jun. 10, 2014. doi: 10.48550/arXiv.1406.2661.

[2] M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein GAN.” arXiv, Dec. 06, 2017. doi: 10.48550/arXiv.1701.07875.

[3] A. Radford, L. Metz, and S. Chintala, “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.” arXiv, Jan. 07, 2016. doi: 10.48550/arXiv.1511.06434.

[4] “Fashion-MNIST.” Zalando Research, Dec. 11, 2023. Accessed: Dec. 10, 2023. [Online]. Available: https://github.com/zalandoresearch/fashion-mnist

Other References.

1.https://developers.google.com/machine-learning/gan/applications#image-to-image-translation

2. https://jonathan-hui.medium.com/gan-some-cool-applications-of-gans-4c9ecca35900

3. https://machinelearningmastery.com/generative-adversarial-network-loss-functions/