The Effects of GAN Parameters on Generated Landscape Images with MLP Architecture

Jimi Michael, Luke Jacobson

***Abstract*—The creation of landscape photos using Generative Adversarial Networks, or GANs, is the subject of this research study. Our project is to take into account the more general research question of how GANs may be used to generate photos from a given dataset, with a focus on what settings and parameters can be adjusted to be able to output the most realistic-looking outcome possible, giving insight on great GAN balancing techniques. Using multi-layer perceptrons as generator and discriminator models, they consist of several different kinds layers, staying on the side of simplicity architecture-wise. This was so we could leverage a GAN with a generator, discriminator, and custom training loop to see how good we could get results from a simple architecture. In this configuration, the GAN was trained using a sizable dataset that included numerous landscape photos from Kaggle with our own manual pre-processing. After training, our model aims to provide the most high-quality landscape photographs possible given a simplistic MLP layer architecture. The several different output photos we were able to generate can give insight on how the many different parameters can affect the quality of the generated images, given realistic images for training data. Our research demonstrates the wide range of capabilities a GAN can have on image processing, and how viewing several different results from multiple parameters can give a higher-level idea of optimal ways to balance the GAN.**

Key Words

1) **Generative Adversarial Networks**
2) **Image Generation**
3) **Neural Networks**
4) **Computer vision**
5) **Multi Layer Perceptron / MLP**

Fig. 1. Favorable Image for Training [5]

Fig. 2. Non-Favorable Image for Training [5]

I. Introduction / Dataset / Architecture

THIS section will introduce and outline the several topics that will be explored in this paper. Our purpose is to dive deeper in the topic of Generative Adversarial Networks, also known as GANs, and give a more research-driven conclusion on the different ways to balance a GAN, and how different parameters one uses can affect what one's output images will be like, as well as how these parameters affect the trajectory, rate, and stability of your training. Ultimately, doing this type of research can give good insight about the effects of GAN parameters to people less experienced with GAN training. Using landscape images in particular, we believe the choosing in this dataset provides for a challenging problem, yet will have easily identifiable characteristics that has the possibility to yield more versatile results, ultimately providing a more broad insight into the true effects the parameters have.

To achieve these results and to give significant insight, we will talk about the dataset we use, highlight a simple MLP architecture that can be used, discuss how the GAN algorithm (given one is using MLPs) works and the mathematics behind it, experiment with 3 different crucial GAN parameters and given the expectations of how they should affect the images, compare the resulting image and the convergence of the loss between each experiment, and then make conclusions based on these different experiments.

To start, we will talk about the dataset that we used, as well as talk about model architecture that we used and will remain static throughout this experimentation. (for the purpose of our research) If we were to try other architectures, there would be much too many combinations of architectures, parameters, and metrics to look into. Therefore, for the purpose of experimentation, we have 1 simple architecture for each model (generator and discriminator) to look at, and will focus on the effects of the changing and the combination of several different parameters on the development of the photos as well as the

```python
# GAN Components
class Generator(nn.Module):
    def __init__(self, latent_dim, img_shape):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.LeakyReLU(0.2),
            nn.BatchNorm1d(256, 0.8),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2),
            nn.BatchNorm1d(512, 0.8),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2),
            nn.BatchNorm1d(1024, 0.8),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh(),
            nn.Unflatten(1, img_shape)
        )

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

class Discriminator(nn.Module):
    def __init__(self, img_shape):
        super().__init__()
        self.model = nn.Sequential(
            nn.Flatten(),
            nn.Linear(int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

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

Fig. 3. Generator / Discriminator Architecture

what the photos look like at the end.

The dataset that we used for this experiment has been derived from a Kaggle dataset called ”Landscape Images” by Arnaud Rougetet. [5]

The dataset has a wide variety of different images, including mountain ranges, beaches, deserts, city buildings, close-ups of wildlife, etc. For the data to be as consistent as possible, we concluded that we need images that only consist of nature, and are actually views of landscapes. (Rather than close-ups of wildlife) An example of an image we want can be seen in Fig. 1, while an image we did believe would work well in our GAN training can be found in Fig. 2. The set contained 4319 files, however we went through manually deleted lots of files that we deemed non-favorable, now with 3393 files remaining.

Our generator is not anything special, as I mentioned before, we are keeping it simplistic for the purpose of the experiment. Our generator code can be viewed in Fig. 3.

As one can see in Fig. 3, these are basic implementations of multi-layer perceptrons, with the purpose of acting as the generator and discriminator for the GAN. The adversarial modeling framework is most straightforward to apply when the models are both multilayer perceptrons.[2] It is important to give the layers in detail for the purpose of reproduction of this experiment.

$$\mathrm{Tanh}(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

Fig. 4. Tanh Function[1]

$$\mathrm{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$$

Fig. 5. Sigmoid Function[1]

The generator is initialized to take in itself (init), the latentdim (also known as Z), and the shape of the image (3 RGB channels, and 64x64 pixels for the image). It’s layers are mostly basic, including several linear layers, LeakyRelu, and batch normalizing layers with a final Tanh activation as well as an unflattening layer. The linear layers take in Z for input features and 256 as the output features to start, then 256 and 512 next, etc. all the way to input features of 1024 and the output features of the image shape. The following linear transformation is applied to the input data: y=xAT+b.[4] For the LeakyRelu function, it takes in 0.2 being the negative slope applied in the following: $LeakyReLU(x) = max(0, x) + negativeSlope * min(0, x)$ where x is the negative slope. [4] Then the Tanh activation function is used which is defined in Fig. 4. The unflatten layer is also applied, which essentially maps the tensor back to its original shape.

The discriminator also is initialized to take in itself and the shape of the image. It also uses, and starts out by flattening the tensor, then has a linear layer with input features of the product of the image shape and 512, a linear layer of the input feature 512, output 256, and a linear layer of input feature 256 and output feature of 1. This is moving in the opposite direction that the generator had it’s linear layers moving. LeakyReLU is also used here, again with a negative slope of 0.2. Finally, the activation function being used is Sigmoid, or Logistic Sigmoid. It can be represented as seen in Fig. 5.

For Sigmoid activation function, the output gets squashed between [0,1]. For Tanh activation, the inputs get squashed as well, and between [-1,1]. These can be viewed in Fig. 4. These are very basic activation functions for a MLP, and they also both suffer from vanishing gradient and computational complexity problems.[1] This would not normally be optimal, and much more complex architectures could be created by adding convolutional layers for example. The choosing of this architecture of layers and activations did not have any necessary reasoning, other than being simplistic, and being a multi layer perceptron. As mentioned before, the architecture is not the focus in this experiment, and this simplicity could possibly be more beneficial when observing the results of the effects of different parameters. These parameters will be revealed within the Experiments section.

Fig. 6 is a helpful visual for being able to view the distributions of these linear, sigmoid, and tanh functions, and how they all look compared to each other.

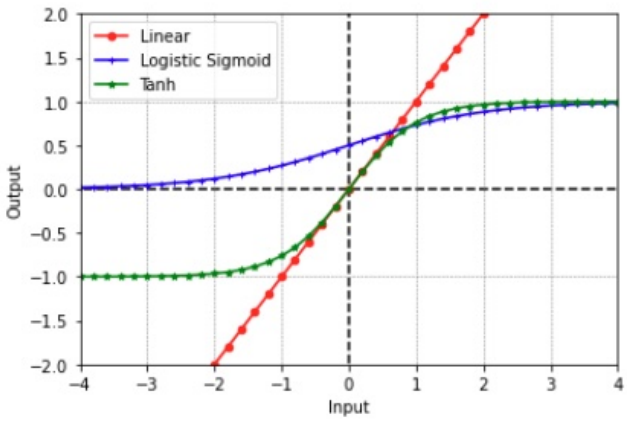


Fig. 6. Activation Function Distributions[1]

```python
from torch.utils.data import Dataset
from torchvision.datasets.folder import default_loader
from torchvision import transforms
from PIL import Image
import os

#Custom dataset class, in order to seamlessly get the next image from the set, and make sure the data is an image file
class CustomDataset(Dataset):
    def __init__(self, root, transform=None):
        self.root = root
        self.transform = transform
        self.images = [os.path.join(root, img) for img in os.listdir(root) if img.endswith(('.jpg', '.jpeg', '.png'))]

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        img_path = self.images[idx]
        image = default_loader(img_path)

        if self.transform:
            image = self.transform(image)

        return image
```

Fig. 7. CustomDatset Implementation

II. Methods / The Algorithm

Before addressing the parameters that we will be experimenting with, it is important to go over the algorithm of the GAN that we are implementing to our generator and discriminator.

Firstly, it is important to see how the data is derived for each batch size. For this we created a CustomDataset class which can be viewed in Fig. 7. The class takes in dataset, and contains a transform which is a normalization technique to rezise the image, (64,64) convert it to a tensor which is a PyTorch type, and normalize the image. Our implementation of the data transform can be viewed at Fig. 8. The CustomDataset has two functions that get the length, as well as get the next item in the dataset.

When dealing with the first MLP (generator), learning the generator's distribution pg over some data x, is defined, then input noise variables are defined pz (z), represent the mapping

```python
#Define the transformation for the data for normalization
data_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

Fig. 8. Data Transformatioin for Normalization

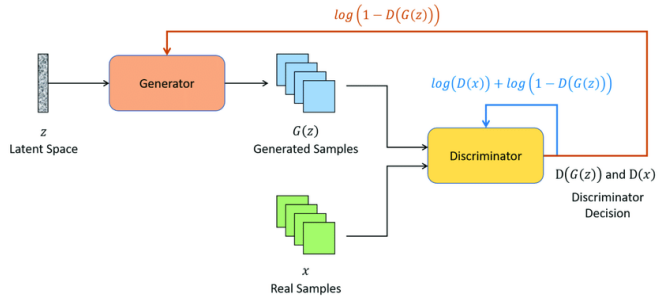


Fig. 9. GAN Training Overview [6]

of data to a space as

$$G(z;\theta g)$$

where G is a differentiable function represented by a MLP with parameters θg.[2]

Then, when dealing with the second MLP (discriminator), this is defined as

$$D(x;\theta g)$$

that outputs a single scalar. Here, D(x) is the probability of providing the correct label to the training sample, as well as the sample given from G (generator). While this is happening, G is trained to minimize the function:

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

An interesting perspective for the dynamic between the discriminator and generator, is that they are playing a competitive two-player game, with the value function V(G,D):

$$minGmaxD, V(D,G) =$$

$$Expdata(x)[logD(x)] + Ezpz(z)[log(1D(G(z)))$$

[2]

A great example of how GAN architecture is used with the data samples as well as the latent space can be seen in Fig. 9.

Now, to talk more about the general algorithm of the GAN training loop. Fig. 11 can be seen to see for a more detailed description of the algorithm.

The algorithm can be described as the following: for the amount of epochs or training loops, and for the amount of k iterations, set a batch of noise samples are initialized pg(z) of batch size length, and a batch of real images from the data also of the batch size length. The discriminator gets updated from ascending the stochastic gradient, and a reminder that the discriminator gets updated like this k times. [2] Back propagation is then performed for the discriminator, and the corresponding optimizer takes a step.

The k steps ends, and then the generator is trained by getting another batch of noise samples of batch size length, and then the generator gets updated by descending the stochastic gradient. [2] Back propagation is then performed on the generator, and the corresponding optimizer takes a step.

The optimizers contain very important information and parameters, and the algorithm for the Adam optimizer has a lot of power on how the images will turn out. The Adam optimizer through PyTorch is essentially used for the generator

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Fig. 10. (Adam Optimizer Algorithm [3]

for number of training iterations **do**
for k steps **do**
• Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
• Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
• Update the discriminator by ascending its stochastic gradient:

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

end for
• Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
• Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Fig. 11. GAN Training Algorithm [2]

and discriminator gradient descent/ascent calculation. The optimizers take in the step size, (learning rate) beta1 and beta2 which are exponential decay rates, and the parameters of the generator or discriminator model. [3] A more detailed algorithm can be seen in Fig. 10.

To briefly describe the k steps, it is essentially the amount of times the discriminator is updated per generator update. In our code, this is a parameter in our train gan function as discriminator step, which can be seen in Fig. 13. The reasoning behind this, is because getting D to be as optimal as possible from only performing one step per discriminator and generator can prove to take longer for convergence, be more demanding computation power-wise, and finite datasets can generalize/overfit easier. [2] This can be solved by alternating between k discriminator steps and 1 generator step. This results in D being maintained closer to its optimal solution D*G(x), which can be described as in Fig. 14.

Our basic implementation of the base training loop can be seen in Fig. 12.

III. Experiments

As mentioned before, we will be experimenting with 3 different and crucial GAN parameters. These parameters include the learning rates of the generator and discriminator, the loss function used, and the amount of k steps. (Discriminator iterations per generator iteration) This experimentation can give a much greater insight into the effects of these parameters

```
for real_batch in real_images:
    real_batch = real_batch.to(device)


    #Update the discriminator several times (more than the generator), this can be changed f
    for _ in range(discriminator_steps):
        #Generate the fake images, normal distribution noise
        noise = torch.randn(batch_size, latent_dim, device=device)
        generated_images = generator(noise)

        #Label real and fake images
        labels_real = torch.ones(batch_size, 1, device=device)
        labels_fake = torch.zeros(batch_size, 1, device=device)

        d_loss_real = nn.BCELoss()(discriminator(real_batch), labels_real) #Binary cross-ent
        d_loss_fake = nn.BCELoss()(discriminator(generated_images.detach()), labels_fake)
        d_loss = 0.5 * (d_loss_real + d_loss_fake)

        #Perform Backpropagation
        discriminator.zero_grad()
        d_loss.backward()
        discriminator_optimizer.step()

    #Train the generator after mltiple discriminator updates
    noise = torch.randn(batch_size, latent_dim, device=device)
    labels_gan = torch.ones(batch_size, 1, device=device)
    g_loss = nn.BCELoss()(discriminator(generator(noise).view(-1, 64 * 64 * 3)), labels_gan)

    #Backpropagation with zero gradient
    generator.zero_grad()
    g_loss.backward()
    generator_optimizer.step()
    #Keeps track of losses
    gen_loss.append(g_loss.item())
    disc_loss.append(d_loss.item())

print(f"Epoch {epoch + 1}/{epochs}, [D loss: {d_loss.item()}, [G loss: {g_loss.item()}]")
```

Fig. 12. Our Implementation of the Training Loop

```
#Function to train the GAN
def train_gan(generator, discriminator, epochs, latent_dim, data_dir, discriminator_steps = 1):
```

Fig. 13. GAN Inputs, where the discriminator steps equal k

on the generated images, and can be very helpful to those needing this insight to balance their GAN.

Now would also be a good time to introduce the loss functions we will be experimenting with, which are PyTorch implementations of binary cross-entropy as well as a version of binary cross entropy combined with a sigmoid layer in the class. In PyTorch, these are called BCELoss() and BCEWithLogitsLoss() respectively. It is claimed that BCEWithLogitsLoss() is more numerically stable than BCELoss(). [4] Both of these functions take in the model images and labels. (either real or fake)

For consistency, Table I can be viewed as every single parameter and value for our base model. This base model is important, because all of the next experiments will added on from this base model. Note that other than the learning rates, loss functions, and number of k-steps, the rest of the parameters and their values from the base model stay the same. Following this experiment section, the generated images, training stability, and loss will all be monitored and compared in the Results section.

For tables II, III, IV, V, VI, VII, and VIII, these begin our

$$D_G^*(\boldsymbol{x}) = \frac{p_{data}(\boldsymbol{x})}{p_{data}(\boldsymbol{x}) + p_g(\boldsymbol{x})}$$

Fig. 14. Optimal D Calculation [2]

experiments where we take our base model, only change 1 of the 3 parameters, and monitor the results. This includes testing a higher generator learning rate, then a higher discriminator learning rate, having both of them higher, using a different loss function called BCEWithLogitsLoss, (across the generator and discriminator) test 3 k-steps, and then finally test 5 k-steps. These experiments are very crucial to begin to monitor how to images change, and will act as a solid foundation for gaining more insight in later experiments.

Next, for tables IX, X, XI, XII, and XIII, these are all experiments combining the different change in the parameters to see how each is affected. Within these experiments, we combined previous experiments of changing generator learning rate with a different loss function, changing discriminator learning rate with a different loss function, change both of the learning rates with a different loss function, a different loss function with 3 k-steps, and a different loss function with 5 k-step. This is potentially one of the most interesting parts of the experimentation, because of what changes may come by combining more of these parameter changes. Will the image look the same, better, or worse? Will the training become unstable? How will it effect the loss?

Finally, For table XIV, all of the possible changes are implemented, which one would expect to provide the most different results from our base model. This experiment will have a high learning rate for the generator and discriminator, have the different loss function,(BCEWithLogitsLoss) and have 5 k-steps.

Parameter	**Value**
Dataset Source	Kaggle Landscape Images
Total Number of Images	3393
Image Preprocessing	Normalization, Resizing to 64x64
Generator Architecture	Linear Layers
Discriminator Architecture	Linear Layers
Activation Functions	Sigmoid/Tanh
Learning Rate	G=0.0002, D=0.0002
Batch Size	64
Number of Epochs	50
Optimizer	Adam
Regularization Techniques	Dropout, Batch Normalization
Loss Function Used	[e.g., Binary Cross-Entropy]
Hardware and Software	PyTorch, [specific GPU/CPU]
Performance Metrics	[e.g., Accuracy, Loss, FID score]
Model Evaluation Criteria	[Criteria used for assessing image quality]

TABLE I
BASE MODEL: ALL PARAMETERS AND VALUES

Our experiments aimed to adjust these parameters to gain better insight on their effects.

Parameter	**Value**
Learning Rates	G = 0.0002, D = 0.0002
Loss Function	Binary Cross-Entropy Loss
K Steps	1

TABLE II
EXPERIMENT 1: BASE MODEL

Fig. 15. Experiment 1: graph results

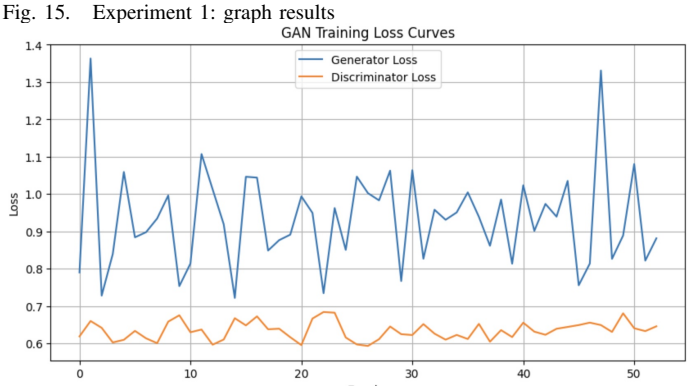


Fig. 16. Experiment 1: Picture Results

IV. RESULTS

For the results section, we will be analyzing our results by inspecting the quality of the images, as well as viewing the loss of the generator and loss of the discriminator across 50 epochs on a graph. Note that with more training epochs, results can always be better, however viewing these images along with their corresponding loss graphs at the start of training can still give great insight on the trajectory of how successful and how stable the model will be.

Right off the bat, looking at experiment 1 versus experiment 2, experiment two has a higher generator learning rate. The results show that the training seemed more stable for experiment two, and the images were more smooth but also more blurry. Comparing this to experiment 3 with the higher discriminator learning rate, the images looked more like experiment 2's, however the training was very unstable, and the generator loss started out very low.

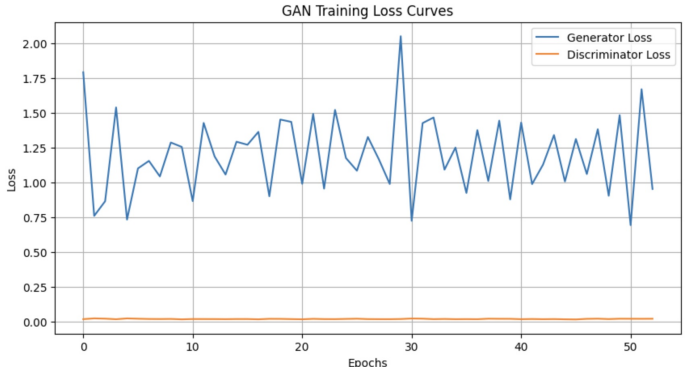
Experiment 4 had a higher loss for the generator and discriminator. Compared to experiment 1, (who's learning rates were also the same) the images look very similar. The training is also very stable, other than the 1 large jump in the generator loss, which can be indicative of mode collapse. This is essentially where the generator can get updated too much without updating the discriminator, causing G to collapse too many values of z to the same value x to have enough diversity to the model.[2] This can be problematic in the long run of training.

Now, checking out experiment 5 using the new loss function BCEWithLogitsLoss, the claims made in the PyTorch documentation do seem correct, it seem that numerically the training is much more stable compared to experiment 1's BCELoss. However, the images also become much more contrasted. The lighter spots turned lighter and the darker spots turned darker. This gives good insight in general for those who need to balance their GAN.

Parameter	Value
Learning Rates	G = 0.001, D = 0.0002
Loss Function	Binary Cross-Entropy Loss
K Steps	1

TABLE III
EXPERIMENT 2: HIGHER G LEARNING RATE

Fig. 17. Experiment 2: graph results



Experiment 6 uses the same loss as experiment 5, however changes the k-steps to 3. The graph looked the same with very stable training, and the images did not also change noticeably.

It is clear there are already some trending results. Higher generator loss can possibly stabilize training, and produce smoother images, but a higher discriminator loss is not stable. Higher learning rates in general though can cause more contrasting pictures color-wise. BCEWithLogitsLoss can clearly prove to show a smoother image with less noise, but can also be more blurry in general.

Experiment 7 lowers loss for both models, uses BCELoss, and uses 5 k-steps. This will directly show how k-steps change the original model. The k-steps alone seem to keep the model stable (although the G loss is high) and produce smooth, yet decently clear, non-noisy images.

Experiment 8 as a higher G learning rate, BCEWithLogitsLoss, and 1 k-step. The training here is extremely stable, but the images are extremely contrasted with little to no clarity. Perhaps with more epochs this could become better.

Experiment 9 is like experiment 8, only with higher D loss rather than G. The pictures are again extremely contrasted, but the training in the discriminator takes a large jump. This does not appear to be optimal.

Experiment 10 has a high G and D learning rate, BCEWithLogits loss, and 1 k-step. For an unknown reason, the G and D loss get flipped from where they have normally been. The images are also extremely noisy and do not look how you would want it to.

Experiment 11 has high G and D learning rates, BCELoss, and 3 k-steps. The training isn't the most stable, but the G loss moves close to the D loss, possibly converging later in training. The images though, are smooth, but extremely contrasted color-wise.

Experiment 12 is like 11, but with 5 k-steps instead of 3. The results are nearly the same, but the images look smoother and more contrasted.

Fig. 18. Experiment 2: Picture Results

Parameter	Value
Learning Rates	G = 0.0002, D = 0.001
Loss Function	Binary Cross-Entropy Loss
K Steps	1

TABLE IV
EXPERIMENT 3: HIGHER D LEARNING RATE

Experiment 13 is like 12, but with BCEWithLogitsLoss rather than BCELoss. The training is very unstable, and the images are extremely contrasted but also noisy. Surprisingly, the model converges, but much earlier than what would be desired. This model has proven to be the least optimal of the bunch, which really goes to show how GAN training requires very particular balancing.

To summarize, using k-steps proves to be a powerful tool for stabilizing a GAN as well give smoother images. A higher generator learning rate appears to also stabilize training, but does not change the image much. An equal learning also provides less stable training, and also worse quality pictures, proving the power of the k-step. BCELoss compared to BCEWithLogitsLoss is much less stable, but also seemed to provide much smoother images in most situations. Training GANs is truly a balancing act, and one thing can be clear, that too much or too little of given parameters can result in bad training and bad results.

One important note and reminder, is we could improve the results by training the model for a longer period of time, this would allow us to achieve better results even in only 50 epochs. The early periods are crucial to understanding the course of the model training. Our study demonstrates how small adjustments to loss functions, k-steps and learning rates have a big impact on the stability and general performance of our model. This research opens the door for better understanding optimization in subsequent research and analysis.

V. CONCLUSION

This study used a Multi-layer Perceptron architecture to investigate the multiple effects of different Generative Adversarial Network parameters. The question we pondered in our experiments is how tweaking certain parameters effect the output of the image being generated by the GAN model. We made many updates to our parameters throughout an enumeration of experiments, by changing the learning rates, loss functions, and k-steps. According to our results from our experiments highlighted in our research, the parameters used in our code do in play a significant role on the output and trajectory of training. In order to produce the most accurate possible output image after GAN training all parameters need

Fig. 19. Experiment 3: graph results

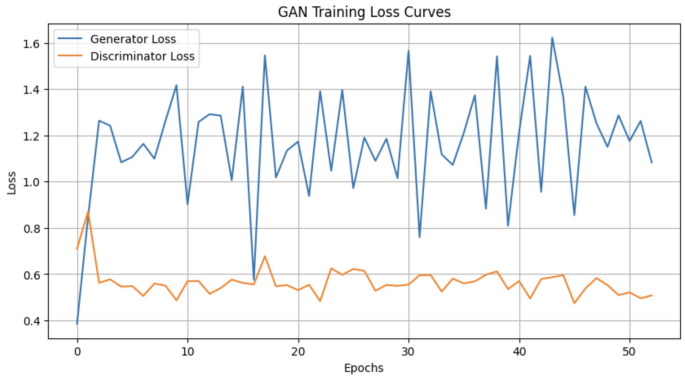
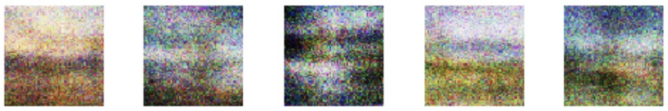


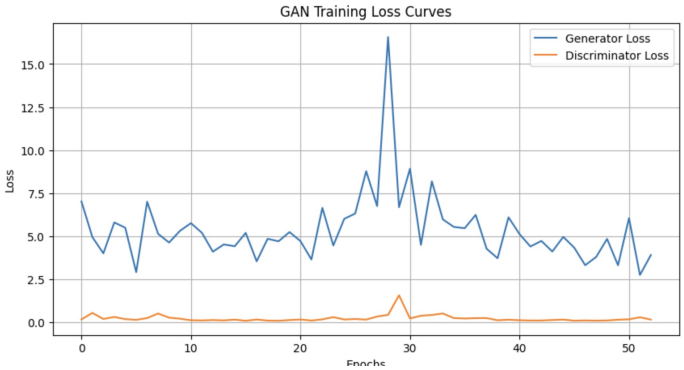
Fig. 20. Experiment 3: Picture Results



Parameter	Value
Learning Rates	G = 0.001, D = 0.001
Loss Function	Binary Cross-Entropy Loss
K Steps	1

TABLE V
EXPERIMENT 4: HIGHER G AND D LEARNING RATE

Fig. 21. Experiment 4: graph results



to be thought through and contain a fine balance. Changing each parameter yields different results, as you can see in our research. Moreover, the research we have conducted highlights GAN's potential for image production, even with just 2 simple Neural Network Models.

Although in our findings we are able to highlight the power of GAN's for image generation using simple neural network architecture, there are some drawbacks to point out. Our experiment's scope was limited in terms of the image types it takes in as input, which only used landscape images to train the model, though they were very versatile in nature. The scope of the model was also limited to a small number of parameters. The following limitations may not fully show the capabilities of a more advanced GAN model. Our research could potentially be improved by further investigation into a more intricate structured GAN, with a large amount of factors in consideration. This further development would not only increase the accuracy of the model but would further our research and in-depth understanding of GAN models.

For more information and a better understanding of the mathematics and algorithm of GANs as well as relevant implementation information, we highly reading our given references which provide great insight to this research topic.

REFERENCES

[1] S. Dubey, et al. Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark. *arXiv preprint arXiv:2109.14545*, Computer Vision and Biometrics Laboratory, Indian Institute of Information Technology, Kolkata, India and Indian Statistical Institute, June 28, 2022. https://arxiv.org/abs/2109.14545

[2] I. J. Goodfellow, et al. Generative Adversarial Networks. *arXiv preprint arXiv:1406.2661*, Département d'informatique et de recherche opérationnelle, Université de Montréal, June 10, 2014. https://arxiv.org/abs/1406.2661

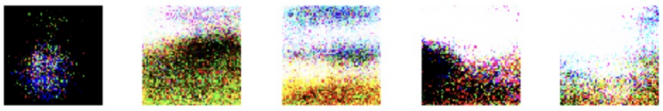
[3] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, ICLR 2015, January 30, 2017. https://arxiv.org/abs/1412.6980

[4] A. Paszke, et al. PYTORCH Documentation. *The Linux Foundation*, 2023. Accessed December 12, 2023. https://pytorch.org/docs/stable/index.html

[5] A. Rougetet. Kaggle Landscape Pictures, 2019. https://www.kaggle.com/rougetet/landscape-pictures

[6] D. Vint, et al. Automatic Target Recognition for Low Resolution Foliage Penetrating SAR Images Using CNNs and GANs. *ResearchGate*, 2023. Accessed December 13, 2023. https://www.researchgate.net/publication/349182009_Automatic_Target_Recognition_for_Low_Resolution_Foliage_Penetrating_SAR_Images_Using_CNNs_and_GANs

Fig. 22. Experiment 4: Picture Results



Parameter	Value
Learning Rates	G = 0.0002, D = 0.0002
Loss Function	Binary Cross-Entropy with Sigmoid Loss
K Steps	1

TABLE VI
EXPERIMENT 5: BCEWITHLOGITSLOSS

Fig. 23. Experiment 5: graph results

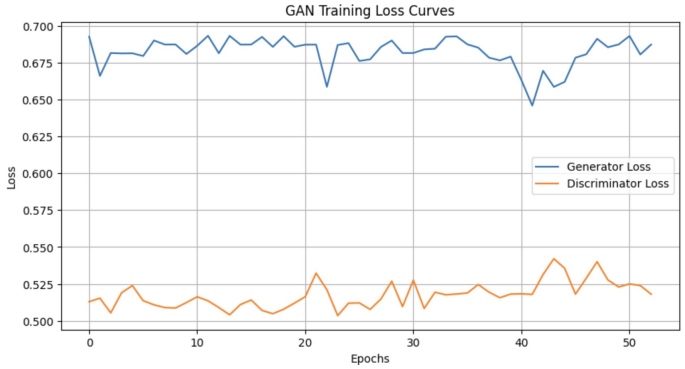
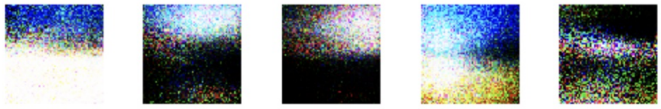


Fig. 24. Experiment 5: Picture Results



Parameter	Value
Learning Rates	G = 0.0002, D = 0.0002
Loss Function	Binary Cross-Entropy with Sigmoid Loss
K Steps	3

TABLE VII
EXPERIMENT 6: 3 K-STEPS

Fig. 25. Experiment 6: graph results

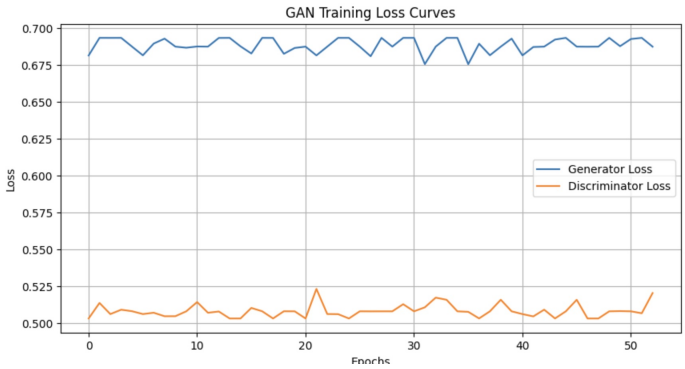
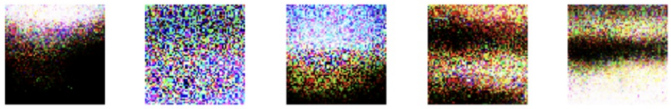


Fig. 26. Experiment 6: Picture Results



Parameter	Value
Learning Rates	G = 0.0002, D = 0.0002
Loss Function	Binary Cross-Entropy
K Steps	5

TABLE VIII
EXPERIMENT 7: 5 K-STEPS

Fig. 27. Experiment 7: graph results

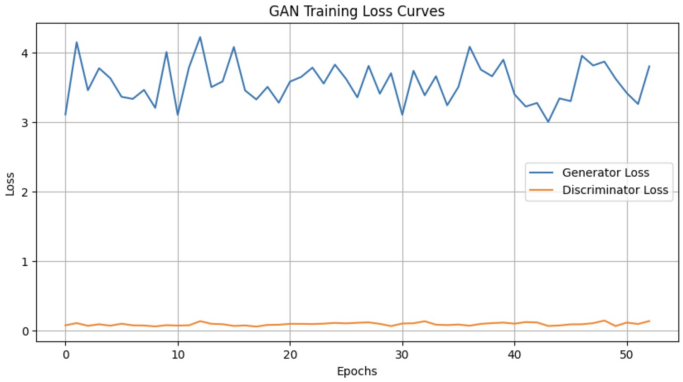


Fig. 28. Experiment 7: Picture Results

Parameter	Value
Learning Rates	G = 0.001, D = 0.0002
Loss Function	Binary Cross-Entropy with Sigmoid Loss
K Steps	1

TABLE IX
EXPERIMENT 8: BCEWITHLOGITSLOSS, AND HIGH G LR

Fig. 29. Experiment 8: graph results

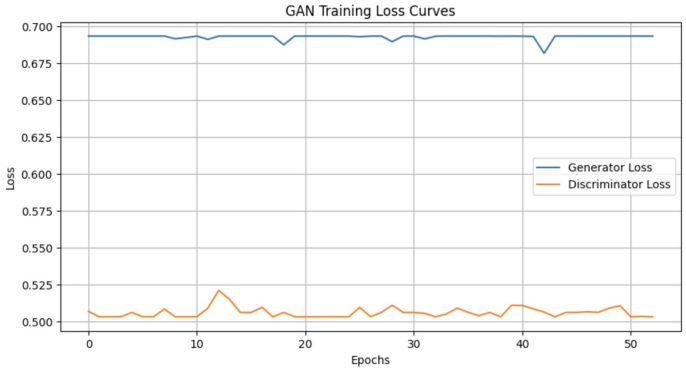
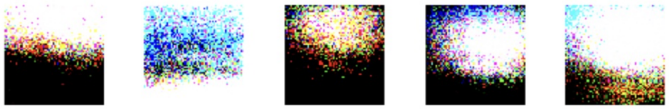


Fig. 30. Experiment 8: Picture Results



Parameter	Value
Learning Rates	G = 0.0002, D = 0.001
Loss Function	Binary Cross-Entropy with Sigmoid Loss
K Steps	1

TABLE X
EXPERIMENT 9: BCEWITHLOGITSLOSS, AND HIGH D LR

Fig. 31. Experiment 9: graph results

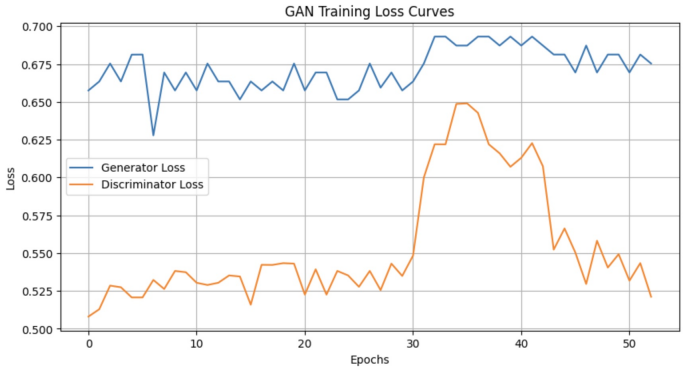
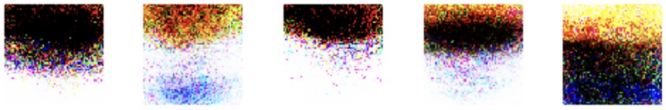


Fig. 32. Experiment 9: Picture Results



Parameter	Value
Learning Rates	G = 0.001, D = 0.001
Loss Function	Binary Cross-Entropy with Sigmoid Loss
K Steps	1

TABLE XI
EXPERIMENT 10: BCEWITHLOGITSLOSS, AND HIGH D AND G LR

Fig. 33. Experiment 10: graph results

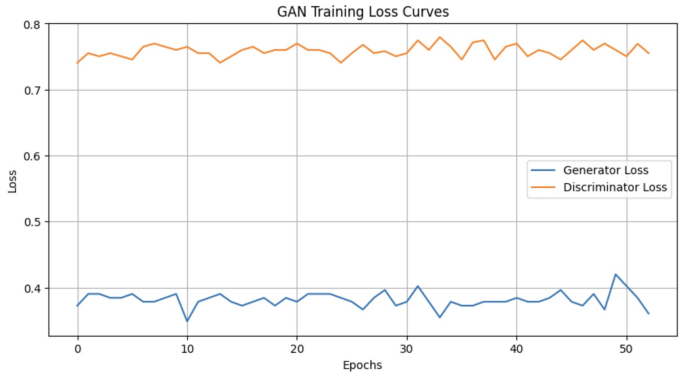
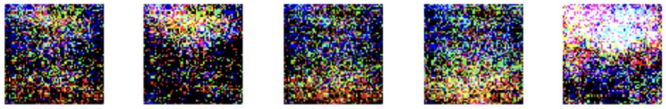


Fig. 34. Experiment 10: Picture Results



Parameter	Value
Learning Rates	G = 0.001, D = 0.001
Loss Function	Binary Cross-Entropy Loss
K Steps	3

TABLE XII
EXPERIMENT 11: HIGH D AND G LR, AND 3 K-STEPS

Fig. 35. Experiment 11: graph results

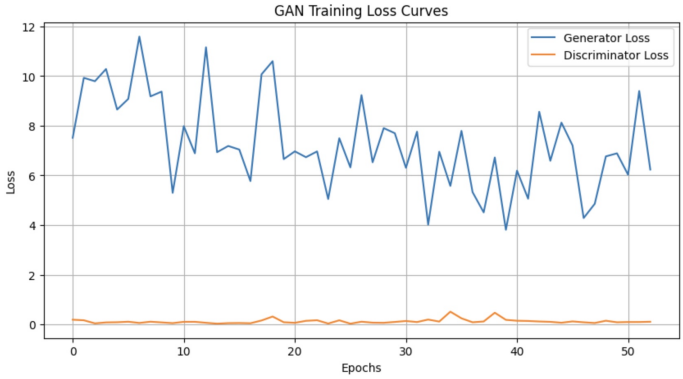
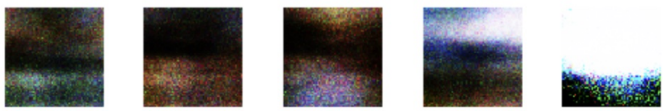


Fig. 36. Experiment 11: Picture Results



Parameter	Value
Learning Rates	G = 0.001, D = 0.001
Loss Function	Binary Cross-Entropy Loss
K Steps	5

TABLE XIII
EXPERIMENT 12: HIGH D AND G LR, AND 5 K-STEPS

Fig. 37. Experiment 12: graph results

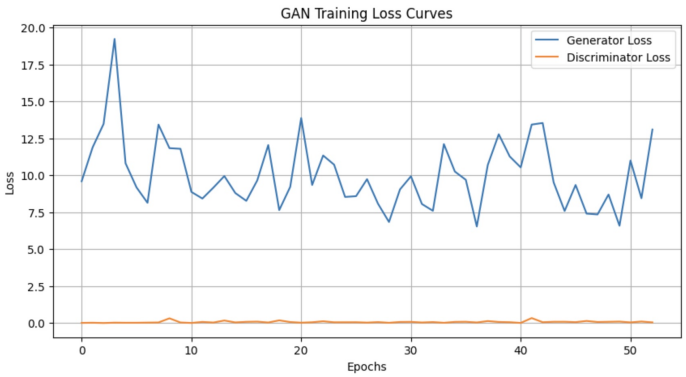


Fig. 38. Experiment 12: Picture Results

Parameter	Value
Learning Rates	G = 0.001, D = 0.001
Loss Function	Binary Cross-Entropy With Sigmoid Loss
K Steps	5

TABLE XIV
EXPERIMENT 13: HIGH D AND G LR, BCEWITHLOGITSLOSS, AND 5 K-STEPS

Fig. 39. Experiment 13: graph results

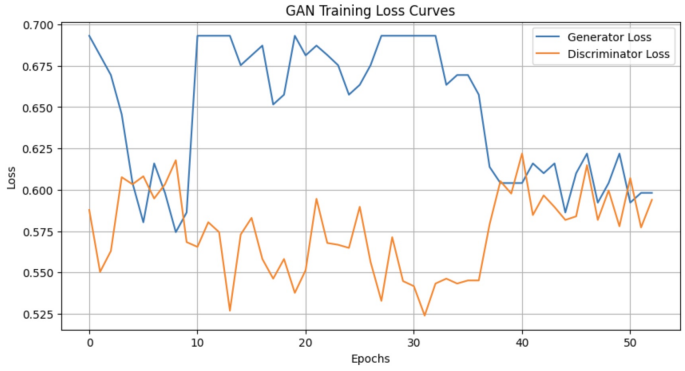


Fig. 40. Experiment 13: Picture Results