SMRTTECH 4AI3

Photograph Generation with Generative Adversarial Networks and CIFAR-10 Dataset

Final Project Report

By Nicholas Grzelak (grzelakn), Thomas French (frenct3)

GitHub: https://github.com/NicholasGrzelak/SMRTTECH_4AI3_Project

Contents

Introduction (Report 1)

Background

Machine Learning models are generally split into two categories: discriminative models and generative models. Discriminative models learn the decision boundary between two or more classes and can classify a given input based on this learning. Generative models learn the patterns and distribution in each dataset and learn how to recreate data that closely resembles the data that it was trained on.

Originally proposed in 2014 by Goodfellow et. al. [1], generative adversarial networks, or GANs, are made up of two sub-models: a generative model and a discriminative model. The generator is trained to create new data that "mimics" the dataset it was trained with, based on an input prompt. The discriminator is trained to classify a given input as belonging to the original data set or being AI-generated. The two models then compete against each other: the generator tries to "fool" the discriminator, and the discriminator tries to correctly identify AI-generated data. This "competition" between the models is why the GANs are termed "adversarial". Once the generator can fool the discriminator with sufficient frequency, the model is considered trained.

Problem Statement

This project will focus on the creation of a generative adversarial network (GAN), that will be capable of generating an original image based on a user prompt. This is similar to existing models such as DALL-E and Midjourney, but the scope of this project is limited to images that are 32x32 pixels in size, and the available user prompts will be limited.

The model will be trained using the CIFAR-10 dataset, which comprises 60,000 coloured images that fall into 10 classes: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships and trucks. The trained model will be capable of generating a brand-new image that falls into one of the classes, as prompted by the user.

The first step is preprocessing all 60,000 images in the dataset. The images currently have pixel intensities of 0 to 255 in each of their red, green, and blue states. For better convergence of the model and more stability, values should be normalized between 0 and 1. The next step involves selection of the best GAN for our needs: each model presents its unique value proposition. More accurate models tend to have heavier computational requirements and trade-offs must be made. Once a GAN is selected, the third step will be to train the model and tune it for the specifications of the dataset. Finally, the last step in the process is to validate the trained model and use it to create unique images not found in the original dataset.

The model will be coded in Python using the TensorFlow Keras library, as well as the NumPy, Pandas, Scikit-learn and Matplotlib libraries.

The successful completion of this project requires a strong understanding of deep learning, image processing, and GANs. Since some of these topics will not be covered until the end of the course, it is the group's responsibility to develop the necessary knowledge and skills ahead of time in order to complete the project.

Data Preprocessing

The CIFAR-10 dataset contains 60,000, each of which is represented by a tensor that is 32x32x3 in size. The first two dimensions represent the image size (32x32 pixels) and do not need any additional processing. The final dimension is the size of the vector that specifies the pixel colour intensities: red, green, and blue. RGB values are typically integers from 0 to 255. To improve model performance and facilitate training, these values are normalized to floats between 0 and 1.

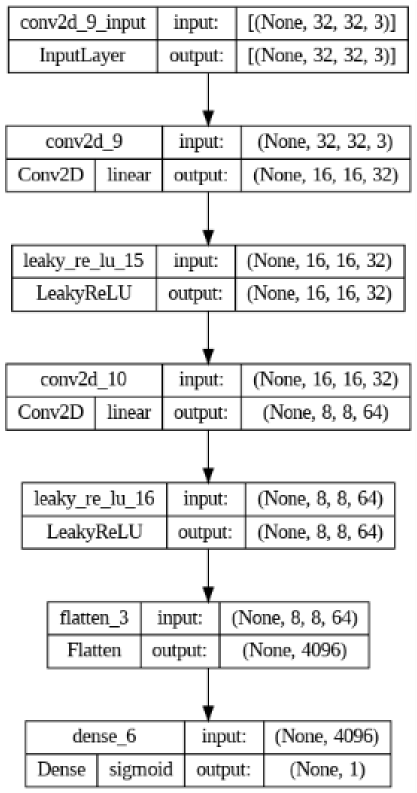
The images were split into training and testing cases, with a test weighting of 20% (the splitting process occurs automatically when the data is loaded). The discriminator is trained using the training dataset and compares them with those produced by the generator to determine which ones were AI generated. The testing set of images is used to empirically compare those with the generator after the generator has been trained, as well as to evaluate the discriminator performance once training is complete.

Since the CIFAR-10 dataset is designed for use with machine learning models, data preprocessing in this project is very limited.

Model Structure

Research of GANs on other datasets inspired a structure which could be adapted to the CIFAR-10 dataset. Previous exposure to creating a GAN for the black-and-white fashion MNIST dataset during Lab 9 demonstrated a structured template to follow. Another GAN which the group took inspiration from was one created for the MNIST Handwritten Digit Dataset [2]. The structures were altered to match the coloured CIFAR-10 dataset.

The discriminator must take in a full-sized image (32x32x3) and then use 2D convolutional layers to down sample the image. The fist convolutional layer used 32 filters and a stride length of (2,2) and same padding. The layer turns the (32x32x3) image into a smaller (16x16x32) image. It is then applied with a leaky rectified linear unit (Leaky ReLU) to introduce nonlinearity to the system. The second layer is very similar to the first except 64 filters are used with size (3,3). The image is now (8x8x64). Once the image is sufficiently small it can then be flattened into a one-dimensional vector of size 4096 before reaching a single neuron. This neuron takes input from the previous layer to decide if the image is AI generated, 0 or from the dataset, 1. For this classification task, it uses binary cross entropy as the loss function and accuracy as the metric. This full process is visualized below in Figure 1.



conv2d_9_input | input: | [(None, 32, 32, 3)]
InputLayer | output: | [(None, 32, 32, 3)]
conv2d_9 | input: | (None, 32, 32, 3)
Conv2D | linear | output: | (None, 16, 16, 32)
leaky_re_lu_15 | input: | (None, 16, 16, 32)
LeakyReLU | output: | (None, 16, 16, 32)
conv2d_10 | input: | (None, 16, 16, 32)
Conv2D | linear | output: | (None, 8, 8, 64)
leaky_re_lu_16 | input: | (None, 8, 8, 64)
LeakyReLU | output: | (None, 8, 8, 64)
flatten_3 | input: | (None, 8, 8, 64)
Flatten | output: | (None, 4096)
dense_6 | input: | (None, 4096)
Dense | sigmoid | output: | (None, 1)

Figure 1: Discriminator Model Structure Diagram

The generator receives a standard normal distribution of noise with specified latent dimensions. It then sends that to a dense layer with (8x8x128) or 8192 neurons. It is then passed through a Leaky ReLU to add in non-linearity before being reshaped into a vector of dimensions (8x8x128). Using a 2D transpose with

64 filters which have a size of (3,3), stride of (2,2) and same padding, the image is upscaled to (16x16x64). Leaky ReLU is then applied again to maintain nonlinearity. Then the second layer applies this again except with a filter of 32 to get a size of (32x32x32). This process is similar to what occurs in the discriminator but instead acts in reverse to upscale the image. After the third Leaky ReLU layer, a 2D convolution layer with 3 filters is used to get the required RGB colour mapping and shape (32x32x3). This layer uses an activation function of tanh to get values between 0 and 1 as suggested by Radford et al [3]. The process can be seen in Figure 2: Generator Model Structure.

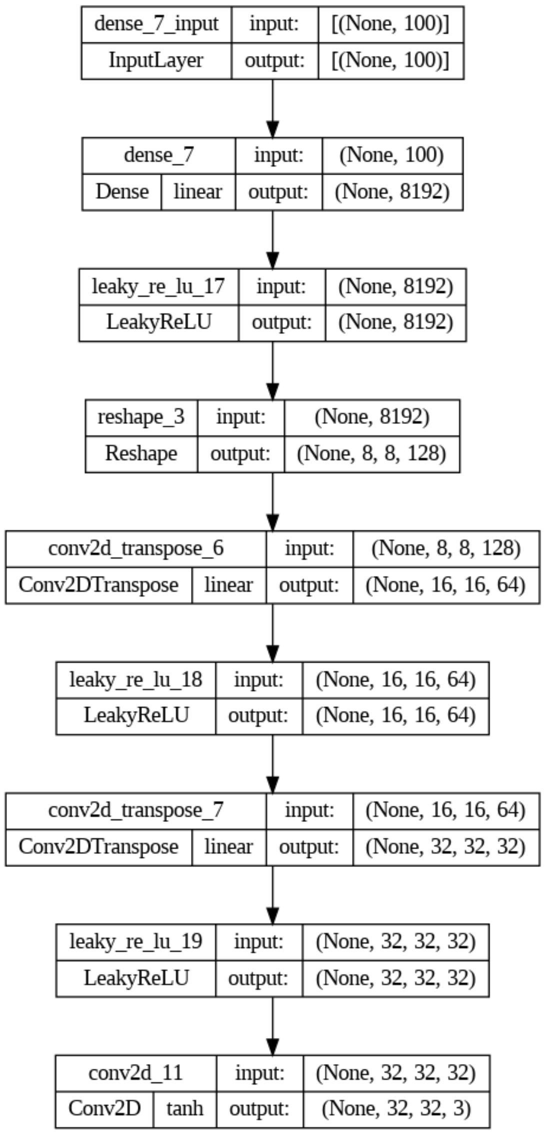

dense_7_input | input: | [(None, 100)]
InputLayer | output: | [(None, 100)]
dense_7 | input: | (None, 100)
Dense | linear | output: | (None, 8192)
leaky_re_lu_17 | input: | (None, 8192)
LeakyReLU | output: | (None, 8192)
reshape_3 | input: | (None, 8192)
Reshape | output: | (None, 8, 8, 128)
conv2d_transpose_6 | input: | (None, 8, 8, 128)
Conv2DTranspose | linear | output: | (None, 16, 16, 64)
leaky_re_lu_18 | input: | (None, 16, 16, 64)
LeakyReLU | output: | (None, 16, 16, 64)
conv2d_transpose_7 | input: | (None, 16, 16, 64)
Conv2DTranspose | linear | output: | (None, 32, 32, 32)
leaky_re_lu_19 | input: | (None, 32, 32, 32)
LeakyReLU | output: | (None, 32, 32, 32)
conv2d_11 | input: | (None, 32, 32, 32)
Conv2D | tanh | output: | (None, 32, 32, 3)

Figure 2: Generator Model Structure Diagram

Finally, the GAN is created by connecting the Generator and Discriminator models in that order. This structure can be seen below in

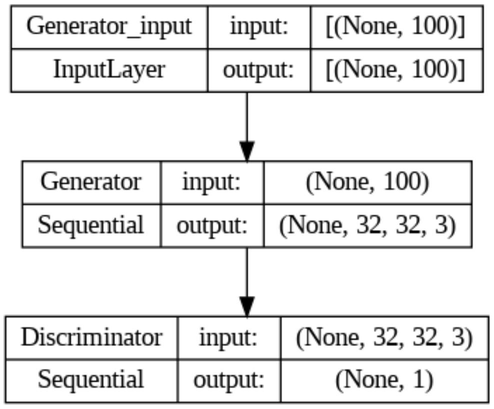


Figure 3: GAN Structure.

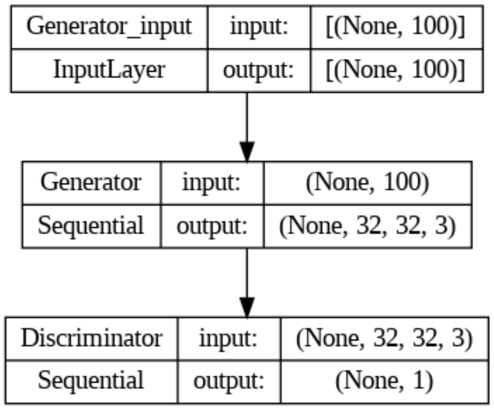


Figure 3: GAN Structure Diagram

Base Case for Hyper-parameter Tuning

The model was used as a starting point for the tuning process. This allows for certain parameters to be changed while having a comparison to measure the changes against. The main comparisons were between the generated images, discriminator loss, generator loss and the training time. Occasionally, the discriminator accuracy is also recorded. Since GAN's have no real measure of success, all these parameters were needed to effectively evaluate each hyper-parameter change. Table 1: Base Case Hyper-Parameter, shows the base parameters that were used along with the previously described model. In the hyper-parameter tuning process, these parameters, along with certain aspects of the model structure, will be varied in order to find their optimal values.

Table 1: Base Case Hyper-Parameter Values

Parameter	Value
Epochs	30
Batch Size	256

Latent Space	100
Learning Rate	0.0002
Activation Functions	Leaky ReLU

The results of this model were promising with a GAN that had steady discriminator loss values from 0.6890 to 0.6978 but slight increase in generator loss values from 0.6567 to 0.6992. These values were calculated as an average over the entire first and last epoch. The full tables of parameters can be seen in Table 2: Base Case Training Results.

Table 2: Base Case Training Results

Initial Discriminator loss	0.6890
Final Discriminator loss	0.6978
Initial Generator loss	0.6567
Final Generator loss	0.6992
Training Time	707.23s

The base model generated the following grid of 100 images below. These images seem promising as they vary in colour with whites, blues, oranges, and blacks. The images also have a distinct variety in structure, although there is some overlap. In general, it seems that the base model does not suffer from mode collapse, which occurs when all generated images appear very similar [4]. With a good starting model, improvements were sought to increase the quality of the images.

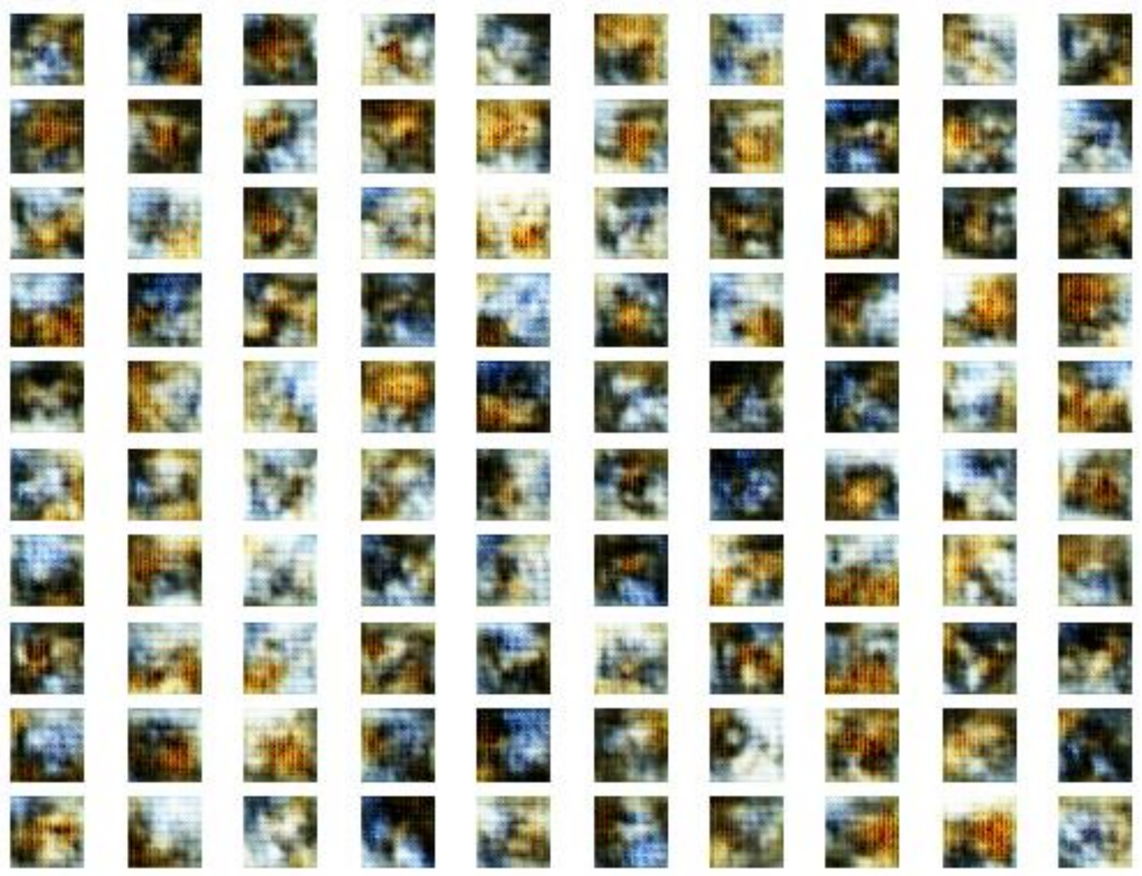


Figure 4: Images Generated by the Base Model

Hyper-parameter Tuning

To improve the performance of the models, several hyper-parameters and training characteristics were tuned. The model learning rate, the sample latent space, the activation functions, and certain layers of the discriminator and generator models, as well as the number of epochs and the batch size in the training process, were varied to determine the optimal values for the final model. In each of the following sections, one hyper-parameter is changed while the rest are held constant, and a new model is trained. For each model, a sample of 100 generated images in presented, as well as a summary of the training process.

Learning Rate

The learning rate for both the discriminator and GAN models was tested at 0.002 and 0.00002 (10-fold increase and decrease respectively).

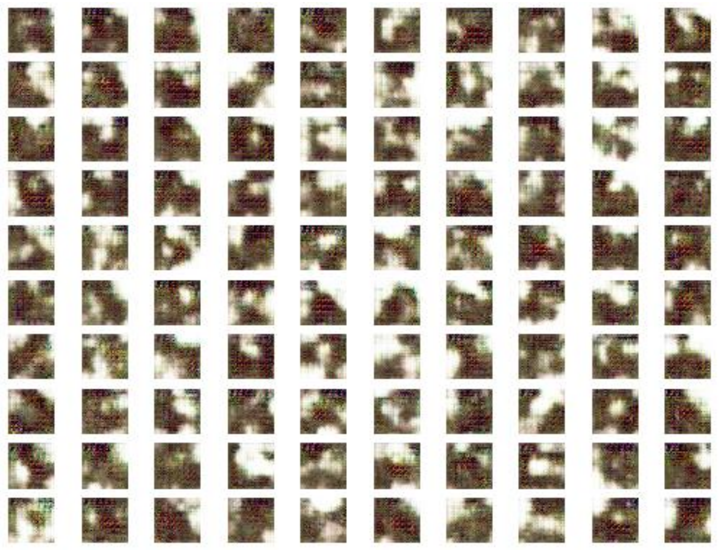


Figure 5: Images Generated by the Model with High Learning Rate (0.002)

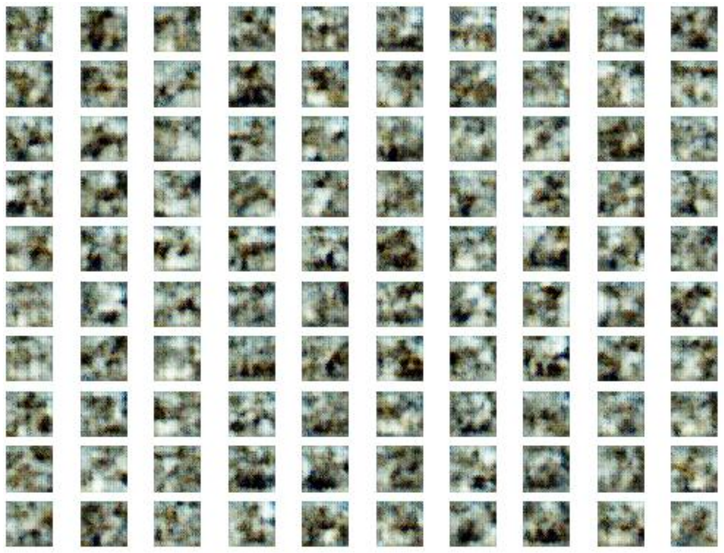


Figure 6: Images Generated by the Model with Low Learning Rate (0.00002)

There is very little colour variety in both sets of images. This suggests that for the learning rate, the base case value of 0.0002 is optimal. It is possible that the first model suffers from mode collapse, since the images appear very similar (Figure 5).

Table 3: Training Results for Learning Rate Tuning

	Higher Learning Rate (0.002)	Lower Learning Rate (0.00002)
Initial Discriminator loss	0.7149	0.6531
Final Discriminator loss	0.6991	0.7014
Initial Generator loss	0.8411	0.6344
Final Generator loss	0.7051	0.6926
Training Time	696.15s	712.07s
Discriminator Accuracy	Real Samples: 93% Fake Samples: 0%	Real Samples: 35% Fake Samples: 68%

The effect of changing the learning rate had minimal effects on the model loss and the training time, however increasing the learning rate significantly changed the accuracy of the discriminator model. For the higher learning rate, the discriminator identified real samples with 93% accuracy, and fake samples with 0% accuracy. This indicates that the discriminator was not effective, since the ideal accuracy is around 50% (the discriminator is correct half the time and is fooled half the time). This performance is reflected in the poor quality and diversity of the generated images.

Model Architecture

Slight changes were made to the layers of the model: in one case, dropout layers were added into the discriminator after each Leaky ReLU activation layer. These were theorized to prevent the discriminator from overpowering the generator. The training time increased from 707.23 seconds to 752.36 seconds. The results of the parameter tuning can be seen in Table 4. Looking at Figure 7 below of the generated images, it seems that by adding the dropout layers there is more variety in image types but not necessarily of colour. With many epochs this could be beneficial to the GAN as it would add more stability and decrease the chances of mode collapse.

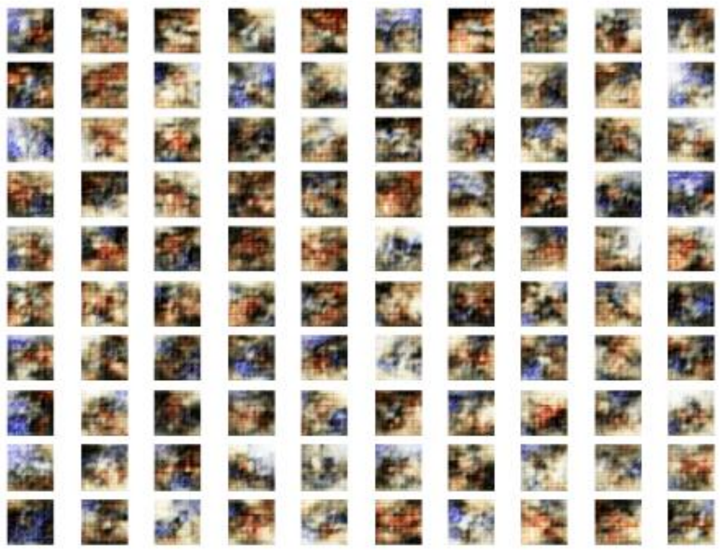


Figure 7: Images Generated by Model with Dropout Layers

In the second case, batch normalization layers were tested in both the generator and discriminator. They were added after every Leaky ReLU in the generator and discriminator as is best practice [3]. This paper noted that adding batch normalization is meant to add stability and prevent mode collapse. The results of batch normalization seemed to increase the training time by roughly 30 seconds, but it had a much larger effect on the losses. The initial discriminator loss started at 0.0394 and rose to 0.2408. The generator loss climbed from 0.0843 to 0.2262. The low initial losses are concerning, but the images seem to show promising results. The images seem to have a good variety of colour with whites, greens, blues, reds and blacks. By adding batch normalization, it seems like the generator was able to train faster and produce a variety of results.

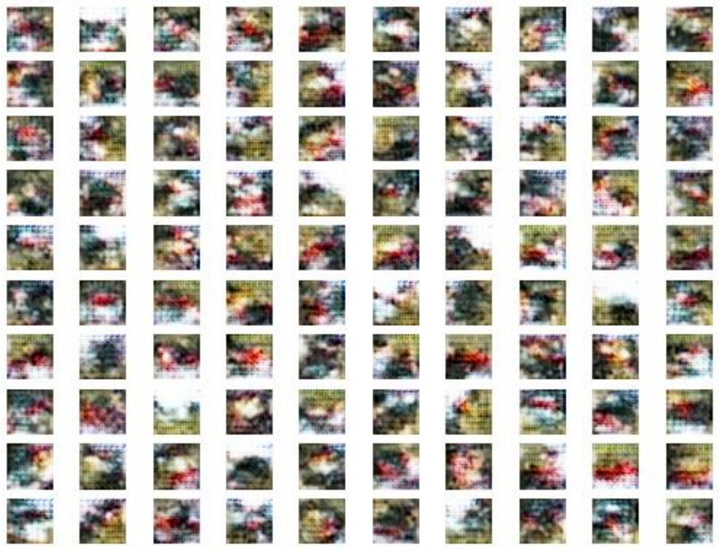


Figure 8: Images Generated by Model with Batch Normalization Layers

Table 4: Training Results for Model Architecture Tuning

	Batch Normalization	Dropout Rate of 40%
Initial Discriminator loss	0.0394	0.6932
Final Discriminator loss	0.2408	0.6979
Initial Generator loss	0.0843	0.6669
Final Generator loss	0.2262	0.7029
Training Time	735.38s	752.36s

Latent Space Dimension

The base model uses a latent space of 100 to generate images. This parameter signifies the amount of noise that is generated and then used to create an image. The group theorized that by changing the latent space, it would change the information given to the generator, in turn changing the amount of detail within the generated images. When testing the theory in the model it seems to be proven true. When latent space was 150 the images seemed to have a rich colour and structure variety. Changing the latent space to 50 seems to have dulled out the images reducing the colours and structures used. These image sets can be seen in Figure 9 and Figure 10 respectively. Surprisingly changing the latent space had very little impact on the results of the model with losses and training time being similar. The full results can be seen in Table 5.

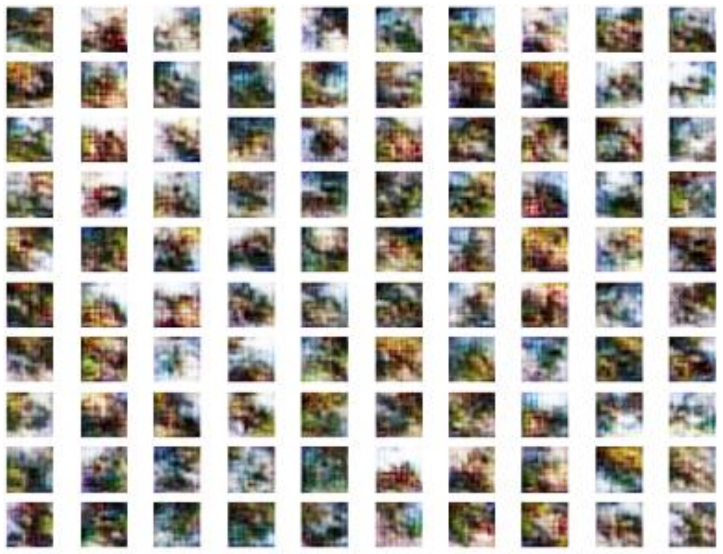


Figure 9: Images Generated by the Model with High Latent Space Dimension (150)

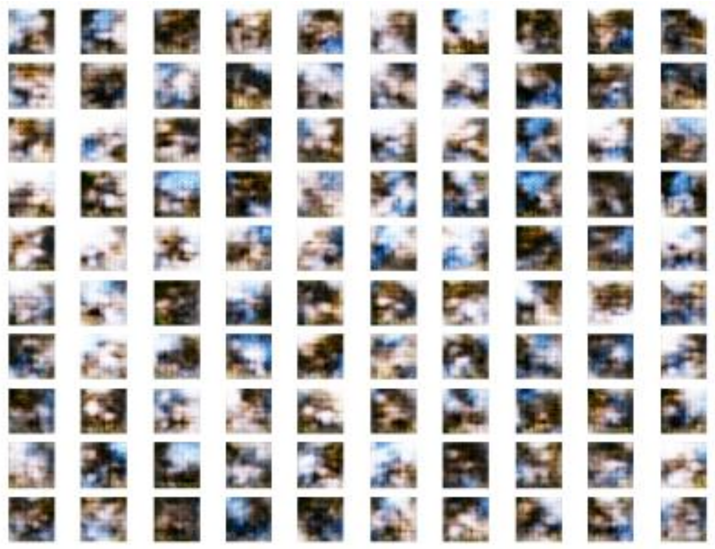


Figure 10: Images Generated by the Model with Low Latent Space Dimension (50)

Table 5: Training Results for Latent Space Dimension Tuning

	Latent Space: 50	Latent Space: 150
Initial Discriminator loss	0.6936	0.7016
Final Discriminator loss	0.6974	0.6970
Initial Generator loss	0.6493	0.6533
Final Generator loss	0.6992	0.7004
Training Time	721.06s	708.38s

Activation Functions

Proposed in the paper, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks" [3], ReLU is supposed to be used for the generator while Leaky ReLU is for the discriminator. The group sought to uncover the differences when using ReLU and when changing the final activation function to softmax for a better distribution. These models had to be trained using local hardware rather than Google Colab (as in the other models), so the training times are much longer. Using ReLU in the generator seemed to yield similar results in terms of loss to the base case of Leaky ReLU. The full results can be seen in Table 6. The images however seem to have less colour diversity with most being predominantly red or orange as seen in e..

Using ReLU and softmax as activation functions proved to have negative consequences on model performance. The images generated in Figure 12 seem to represent little more than coloured noise and have no real shape or structure.

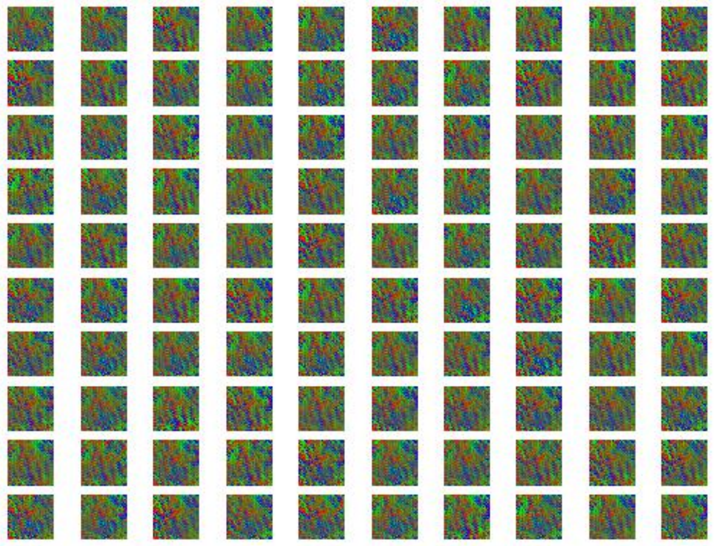


Figure 12: Images Generated by the Model with ReLU in Generator and Softmax in the GAN

Table 6: Training Results for Activation Function Tuning

	ReLU in Generator	ReLU and Softmax in Generator
Initial Discriminator loss	0.6977	0.5577
Final Discriminator loss	0.6987	0.0008
Initial Generator loss	0.6445	0.8822
Final Generator loss	0.7068	8.2294
Training Time	1697.82s	1596.30s

Number of Training Epochs

The number of epochs used in the model training process was tested at values of 10 and 50. It was expected that the model performance would increase as the number of epochs increases, however the group still chose to try reducing the number of epochs.

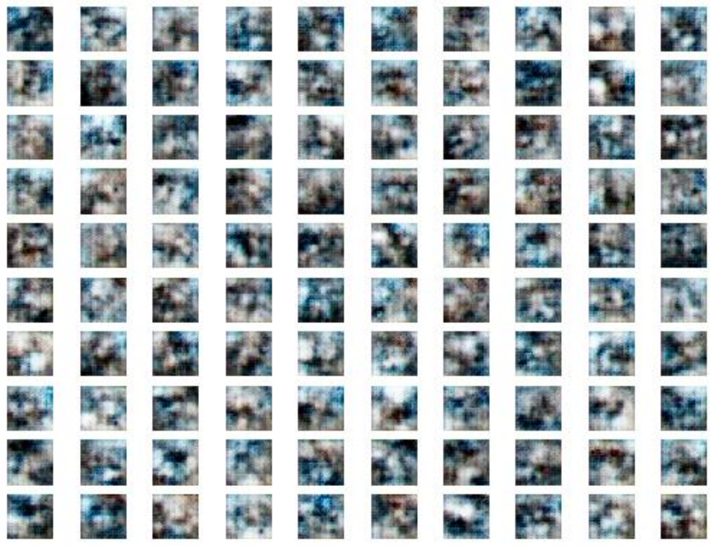


Figure 13: Images Generated by the Model with Fewer Training Epochs (10)

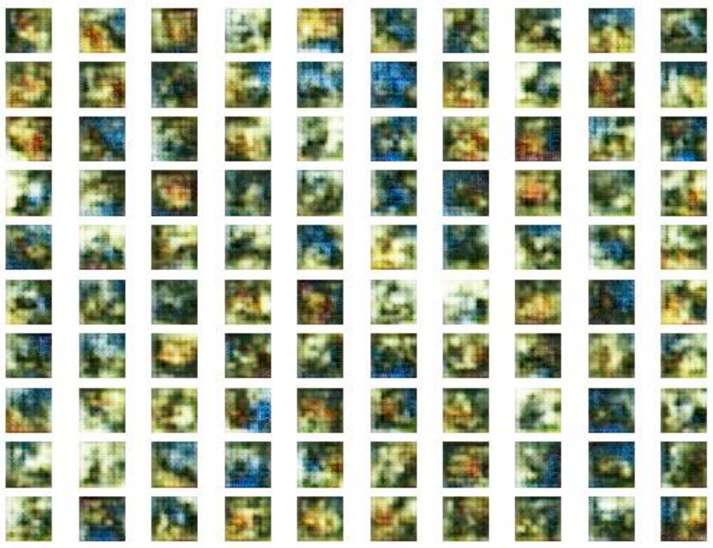


Figure 14: Images Generated by the Model with Fewer Training Epochs (50)

As expected, training the model for a larger number of epochs results in better quality images with more colour variety (Figure 13). However, the images generated after 50 epochs are still not realistic, and do not resemble any of the dataset classes (Figure 14): we can conclude that an even greater number of training epochs is required.

Table 7: Training Summary for Number of Epochs Tuning

	Fewer Epochs (10)	More Epochs (50)
Initial Discriminator loss	0.6889	0.6988
Final Discriminator loss	0.6965	0.6976
Initial Generator loss	0.6586	0.6461
Final Generator loss	0.7050	0.7015
Training Time	238.70s	1179.99s
Discriminator Accuracy	Real Samples: 35% Fake Samples: 66%	Real Samples: 87% Fake Samples: 35%

Logically, model training is faster with fewer epochs, and takes longer with more epochs. One interesting result from this experiment is that we can see the approximate training time for each epoch, which is about 24 seconds.

Batch Size

The effect of batch size on model performance was also tested. The model was trained using a batch size of 100 and 400 in addition to the base case.

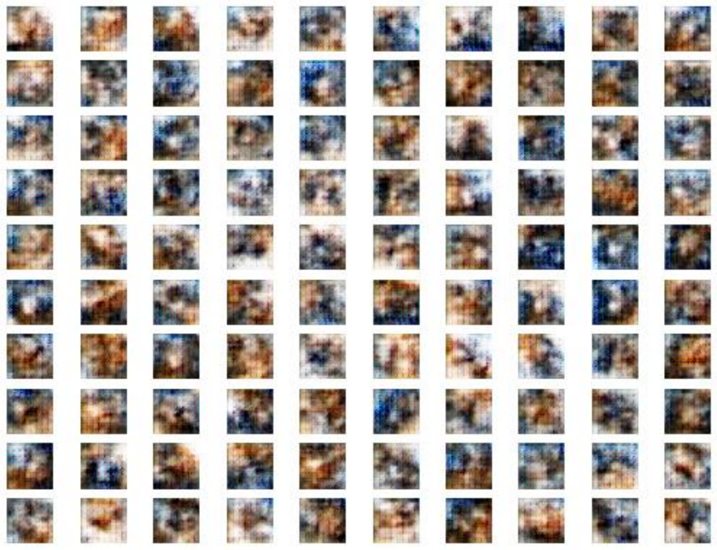


Figure 15: Images Generated by the Model with Smaller Batch Size (100)

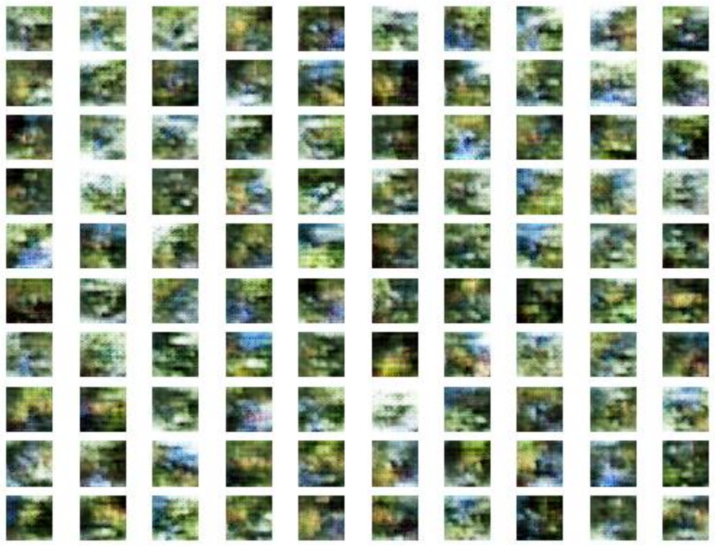


Figure 16: Images Generated by the Model with Larger Batch Size (400)

Training the model with a smaller batch size seems to have little effect on the resulting images (Figure 15): they are similar in colour and quality to those generated in the base case. Increasing the batch size changes the colour variety of the images, but the quality is similar: there are no realistic images (Figure 16).

Table 8: Training Summary for Batch Size Tuning

	Larger Batch Size (400)	Smaller Batch Size (100)
Beginning Discriminator loss	0.6476	0.7059
Ending Discriminator loss	0.6988	0.6963
Beginning Generator loss	0.6587	0.6786
Ending Generator loss	0.7064	0.7031
Training Time	309.25s	1771.07s
Discriminator Accuracy	Real Samples: 29% Fake Samples: 47%	Real Samples: 72% Fake Samples: 2%

Changing the batch size has a negligible effect on the loss values, however it significantly changes the training time. Increasing the batch sizes results in a much shorter training process, whereas decreasing the batch size has the opposite effect. The group suspects that this is because larger batch sizes result in few batches per epoch, and the increase in batch size has an insignificant effect on the training time per batch, so the overall training time is reduced. The discriminator accuracy for fake samples is only 2% for the smaller batch size model, which suggests that the generator consistently fools the discriminator, which is not desired.

Final Models

The results of the hyper-parameter tuning process were used to design two models that were trained over approximately twelve hours. The first model was designed to be more experimental, in order to better explore the interactions between the different hyper-parameters and model characteristics. The second model was chosen to be a "control" model, so it is very similar to the base model, with only slight changes. Both models are described below.

BNDRHL Model (Batch Normalization, Dropout, High Latent noise)

From the previous hyperparameter tuning, it was seen that using batch normalization, dropout rate and a higher latent noise was desirable. Using batch normalization, dropout rate and a higher latent noise the BNDRHL model was trained overnight. The model was trained overnight to reach the highest number of epochs it could. After 400 epochs of training, it was found together these parameters together proved the model unstable. The figure below shows some of the images generated by the fully trained generator. The images suffer from mode collapse as they all have a very similar colour pattern.

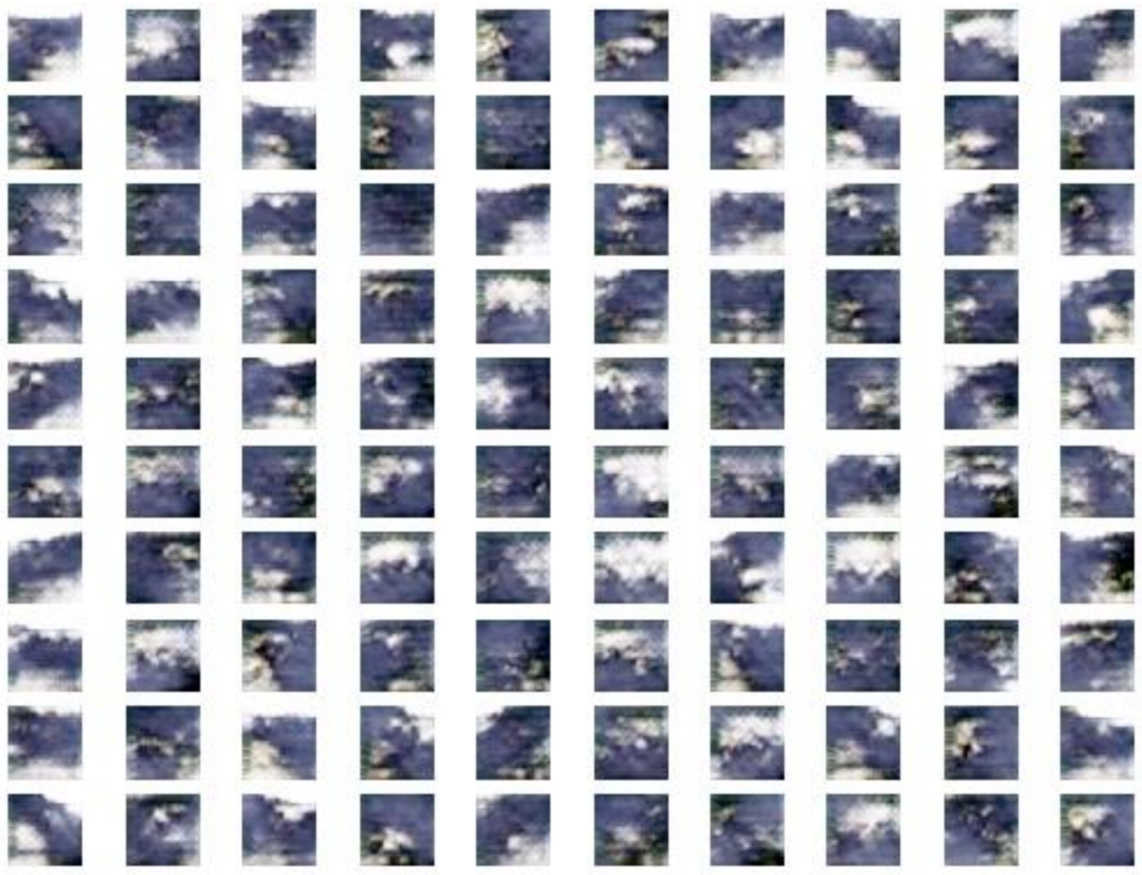


Figure 17: Images Generated by the BNDRHL Model, Trained for 400 Epochs

The average generator and discriminator losses per epoch also prove that the model was unstable. The discriminator loss rose quickly to a max of near 0.4 it then decreased before rising again at 30 epochs. It started a to decline after 30 epochs to a near 0 value by 400 epochs. This relation can be seen in

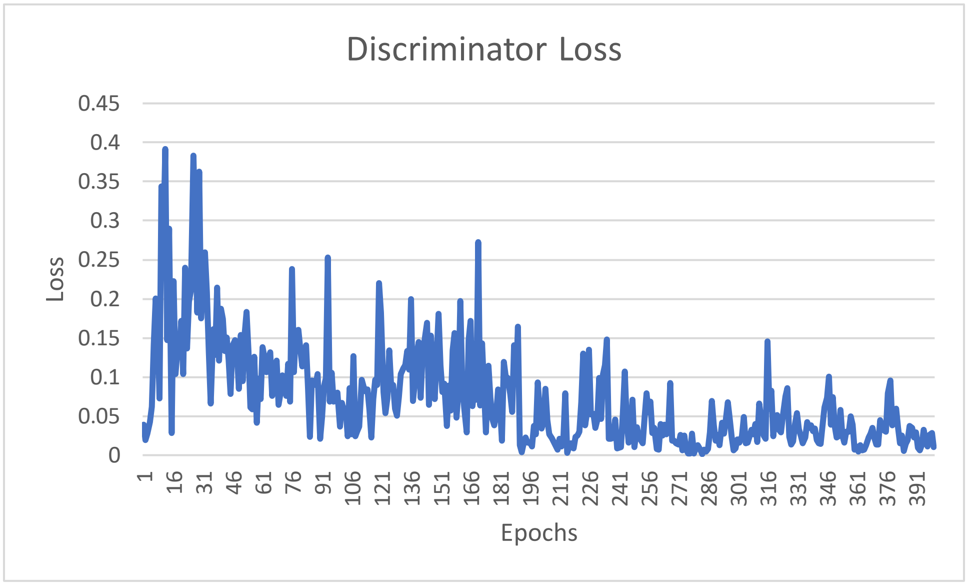


Figure 18: Discriminator Loss. The Generator loss was quite unstable reaching a maximum value of 6.49. Had the model been supervised while training, it would be clear that parameters would need to be changed.

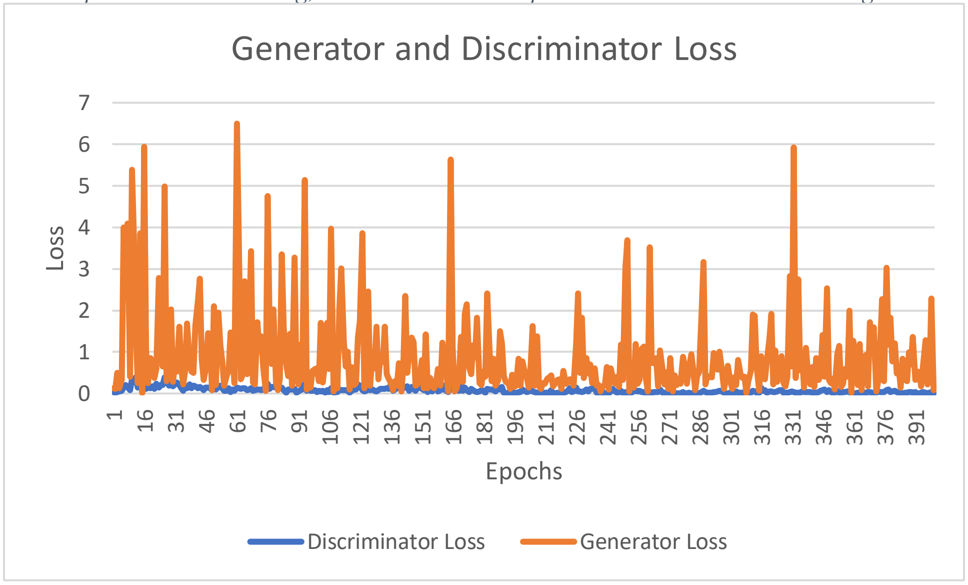


Figure 19: Generator and Discriminator Loss

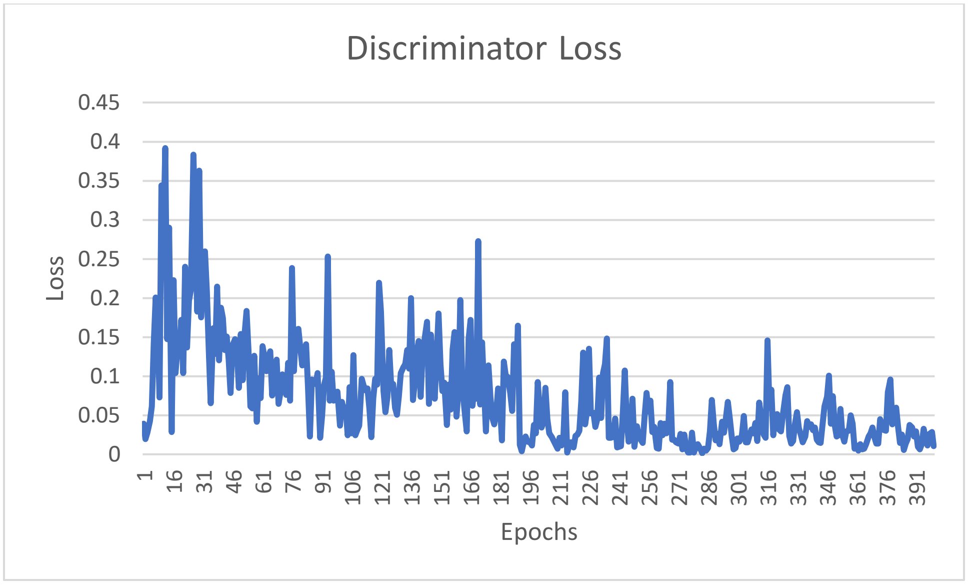


Figure 18: Discriminator Loss per Epoch for the BNDRHL Model

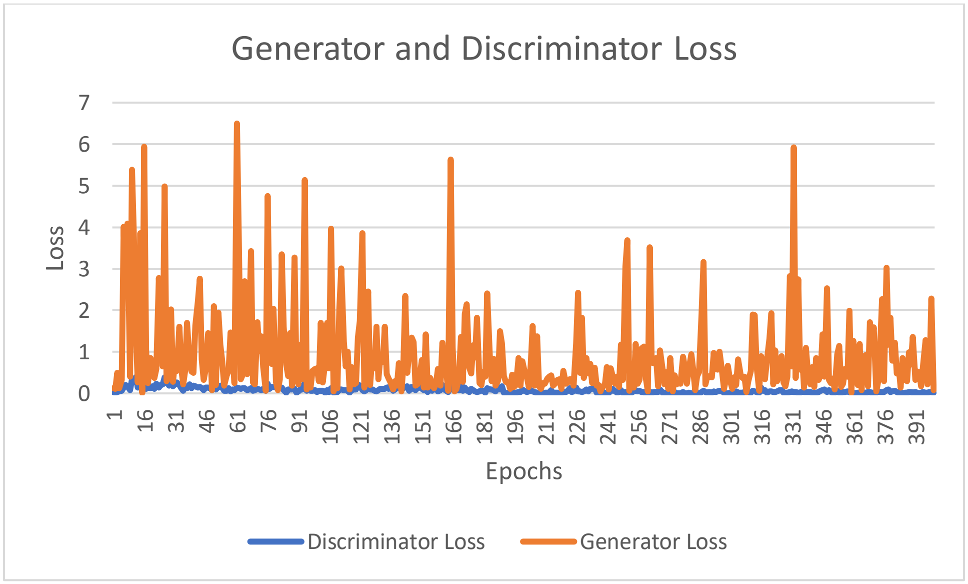


Figure 19: Generator and Discriminator Loss for the BNDRHL Model

Base Model with Larger Batch Size

A second model, identical in structure to the base model, was trained for 1000 epochs, with a batch size of 400. This model was much more successful than the BNDRHL model. Some sample images generated by this model are shown below.

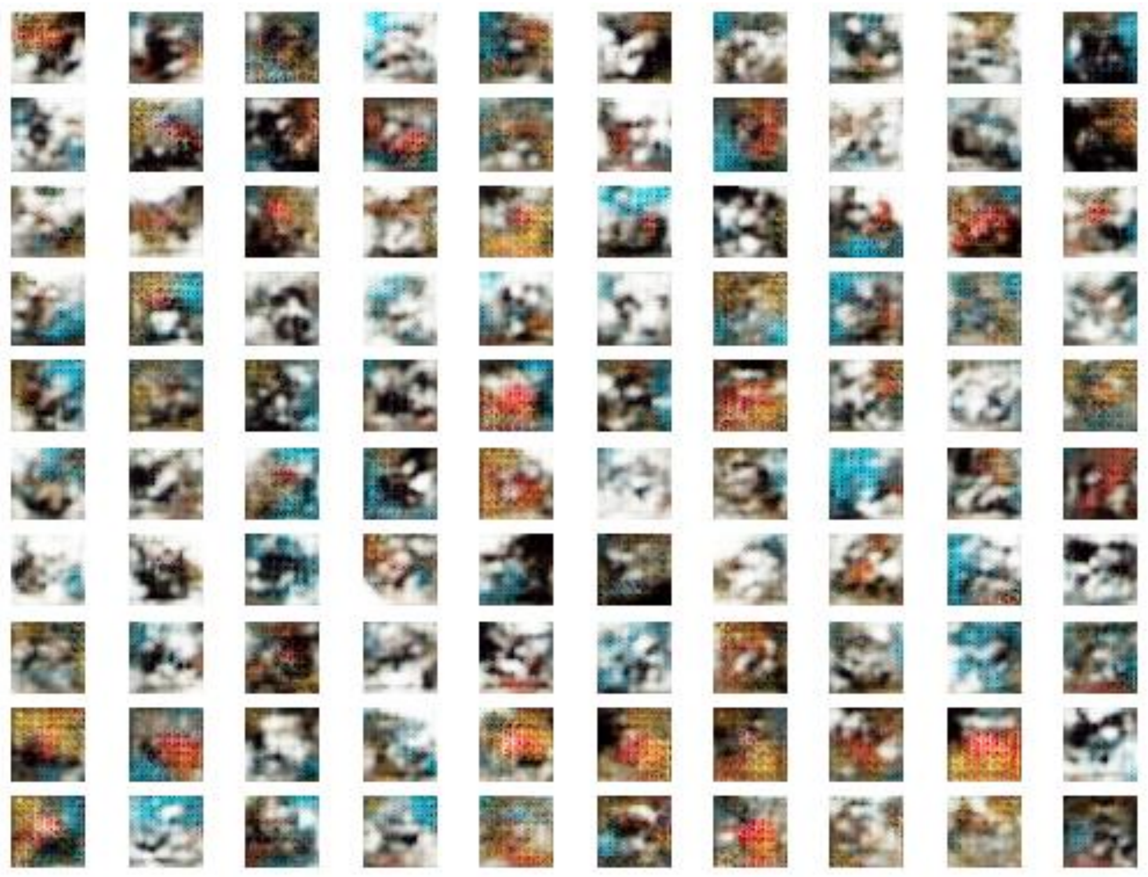


Figure 20: Images Generated by the Base Model with Larger Batch Size, Trained for 1000 Epochs

This model produces much more colour variety, however none of the images are easily recognizable as belonging to one of the ten dataset classes. Further training, or improvement of the model structure, is required to obtain clear, recognizable images.

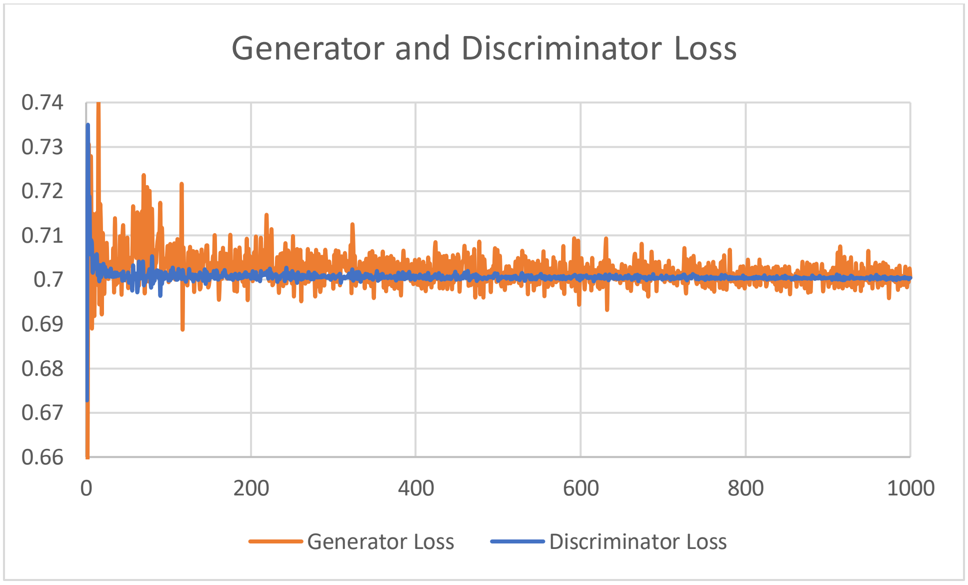


Figure 21: Generator and Discriminator Loss for the Base Model with Larger Batch Size, Trained for 1000 Epochs

The loss values for this model are very consistent around 0.7 for the duration of the training process. This indicates that both generator and discriminator model are learning in tandem, and the GAN training process is occurring smoothly and is stable.

Further Improvements and Next Steps

The results of the two final models are unsatisfactory: neither model is able to consistently produce realistic-looking images. There are several possible causes for this poor performance.

One possibility is that the hyper-parameter tuning was insufficient, and further optimization is required. Rather than continuing with manual tuning, we could implement automatic tuning using the Keras Tuner library to better optimize the model. We could also experiment further with the various activation functions and how they affect the performance of the models.

However, it is unlikely that hyper-parameter tuning is the sole reason for the models' failure. It is more likely that the model architecture is not complex enough to generate realistic images. To obtain a better model, we would have to experiment with different model structures, and include additional convolution, activation, dropout, and normalization layers in both the generator and discriminator models.

Another way to improve performance is to use a pretrained discriminator. This entails training the GAN with a discriminator that has already been exposed to real and artificial images, and so it is more effective at identifying fake samples. It is possible that this would allow the model to converge faster, and to generate more realistic images [5]. However, using a pretrained model could also lead to other problems with the model since the "competition" between the discriminator and the generator is no longer fair and the models no longer learn at the same pace.

To improve the realism of the generated images, we could consider introducing additional evaluation metrics for the generator, such as the Fréchet Inception Distance (FID). First proposed in 2017 by Heusel et al., FID is now the standard metric for assessing the realism of AI-generated images. FID works by evaluating the probability distribution of the datapoints in the image and comparing them to a set of real images [6].

Once the model is sufficiently improved using the methods outlined above, the final step in improving the model accuracy is to improve performance by training the model over a larger number of epochs. The optimal number of epochs cannot be known in advance, so one possibility is to let the model train for a very large number of epochs and observe the performance after every 10 iterations. Then, once the model converges satisfactorily, the model can be saved, and the training is stopped.

The group's ability to implement these solutions is limited by the available computing power. All models were trained on laptops or using Google Colab, so they had to be simple enough to train in a reasonable amount of time.

Conclusion

The original goal for the project was to create a model that outputs an image belonging to the class selected by the user. Unfortunately, the current models are not able to do this since they are not able to generate realistic images. If we were to implement this functionality, we could do so in several ways:

We could adapt our model into a Conditional GAN (CGAN). This type of model is like the regular GAN, except that the discriminator and generator also receive the image class as an input during the training process. This allows the model to adapt its generation/discrimination process to each class and generate specific images in response to user prompts. In other words, a CGAN is "aware" of the different classes in the dataset, whereas a regular GAN is not.

Rather than training a CGAN, we could instead train different models for each class in the dataset. This would involve splitting the dataset by class into ten subsets, and training ten different GANs. Then, based on the user's input, the corresponding generator model is called to generate an image.

Alternatively, we could train an additional discriminator to distinguish between the different classes in the dataset. When the user inputs a class, the generator repeatedly creates images until the class discriminator identifies one that belongs to the requested class, and the image is returned to the user.

Nevertheless, the project was still a success in that it provided exposure to the creation and optimization of machine learning models, and the group is happy with the results. The final model was able to generate diverse images, most of which do not appear realistic enough to belong to any of the dataset classes. However, certain images appear as if they could be real. Below are two examples of realistic looking images, and their interpretations by the group.

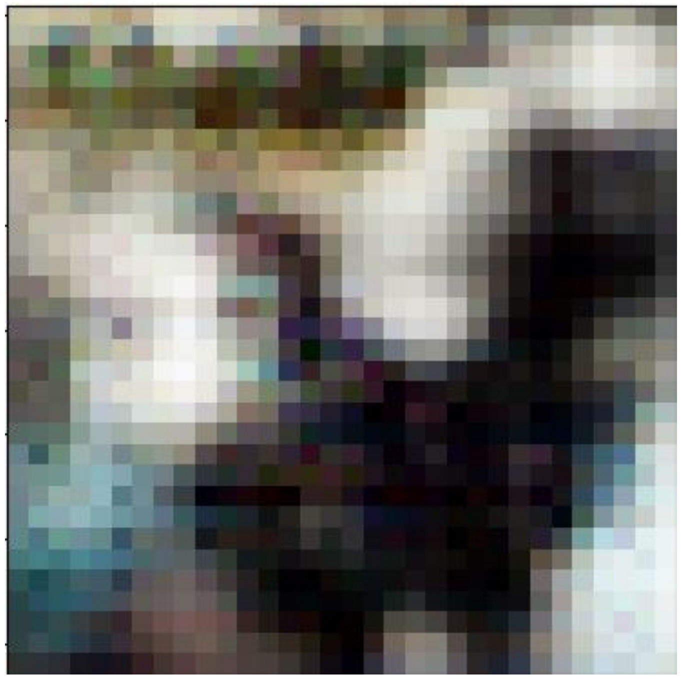


Figure 22: Cat in the Winter (Image Generated by the Base Model with Larger Batch Size)

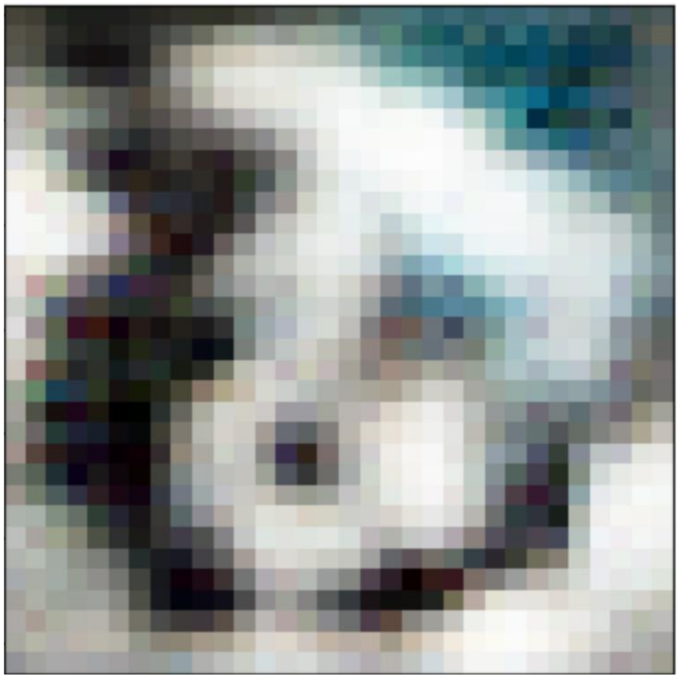


Figure 238: Head of a Small Dog (Image Generated by the Base Model with Larger Batch Size)

References

[1] I. J. Goodfellow et al., “Generative adversarial networks,” *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, Oct. 2020, doi: 10.1145/3422622.

[2] J. Brownlee, “How to Develop a GAN for Generating MNIST Handwritten Digits,” Machine Learning Mastery, Jun. 27, 2019. https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-keras/

[3] A. Radford, L. Metz, and S. Chintala, “Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks,” 2016. Available: https://arxiv.org/pdf/1511.06434.pdf

[4] “Common Problems | Generative Adversarial Networks,” Google Developers. https://developers.google.com/machine-learning/gan/problems

[5] Y. Wang, C. Wu, L. Herranz, J. Van De Weijer, A. González-García, and B. Raducanu, “Transferring GANs: generating images from limited data,” arXiv (Cornell University), May 2018, doi: 10.48550/arxiv.1805.01677.

[6] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, “GANs trained by a two Time-Scale update rule converge to a local Nash equilibrium,” arXiv (Cornell University), Jun. 2017, doi: 10.48550/arxiv.1706.08500.