Extended Essay

Computer Science

Investigating the Impact of Reducing Dataset Sample Count on Generative Adversarial Network Performance

Research Question:
To what extent does limiting the size of a Generative Adversarial Network's training dataset impact the Fréchet Inception Distance fidelity score of its generated image samples?

Word Count: 3998 words

Contents

1 Introduction

Generative modelling has become a major field in machine learning, becoming particularly interesting with the introduction of the Generative Adversarial Network (GAN) by Ian Goodfellow in 2014. Since then, significant progress has been made with models such as Nvidia's StyleGAN2, OpenAI's DALL-E 2, and StabilityAI's Stable Diffusion, having all received significant media coverage. The applications of GANs range from generating realistic images of human faces, realistic sounding voices, generating an image given a text prompt, to style transfer (e.g., an image of a zebra is converted to an identical image of a horse instead). GANs are also the same technology powering deep fakes due to their high versatility.

From the impressive results achieved by the aforementioned GANs, it is evident that they are a tremendously powerful architecture, which Yann LeCunn, distinguished machine learning researcher and Chief AI Scientist at Meta has also commented on as being 'the most interesting idea in the last 10 years in Machine Learning'.

When applying GANs in situations involving computer vision such as generating images or video, it is important that the generated content is of high fidelity and of good visual quality, as GAN generated images can sometimes contain small artifacts or unusual patterns which are undesirable. This problem has been mitigated to some degree in recent GAN related papers, however they only address the architecture and propose improvements. As with all machine learning models, the more data it is exposed to the more realistic output a GAN can generate.

Situations may exist in which it is difficult to acquire large quantities of data to train a GAN, thus limiting its potential to produce high fidelity outputs. This paper seeks to investigate the extent to which the limitation in data impacts the fidelity of generated images (measured using the FID metric), answering the research question:

To what extent does limiting the size of a Generative Adversarial Network's training dataset impact the Fréchet Inception Distance fidelity score of its generated image samples?

2 Background Theory

2.1 Generative Models

The role of a generative machine learning model is, as the name implies, to generate data. This contrasts with the function of a classification model, where features x are taken in, to produce an output of y indicating the class prediction. Classification models are typically categorised as *supervised learning* models, as they are adjusted depending on how correct their prediction is. Generative models, however, take in parameter z, a vector of random values (noise), and outputs features x, which is effectively the opposite of a classification model.

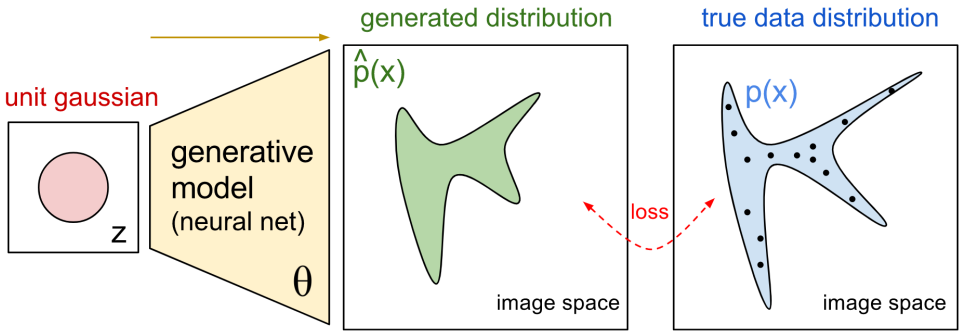


Figure 1: Illustration of generative model distributions (Karpathy et al. 2016)

Mathematically a dataset of samples $x_1, x_2, ..., x_n$ could be thought of as a distribution of *true* data $p(x)$ (blue in figure 1), where x_n is a unique sample of data (e.g. an image). A generative model takes in a vector of gaussian noise z, and passes it through a neural network with parameters θ, and outputs an approximation of the distribution of true samples $\hat{p}_\theta(x)$ (green in figure 1). During training the network parameters θ are adjusted using gradient descent (the method by which neural networks optimise parameters), in order to generate samples mimicking the real distribution (I. Goodfellow, Bengio, and Courville 2016).

2.2 Generative Adversarial Networks (GANs)

GANs are a type of generative model introduced in I. J. Goodfellow et al. 2014, which greatly improved upon previous generative models. A GAN consists of two neural networks, both tasked with opposing or antagonistic objectives, hence the *adversarial* aspect. It is analogous to two parties: a forensic expert (the discriminator), and a counterfeiter (the generator). The counterfeiter tries to make realistic counterfeit bills in an attempt to trick the expert, while the expert tries to discern counterfeit bills from real bills (I. J. Goodfellow et al. 2014).

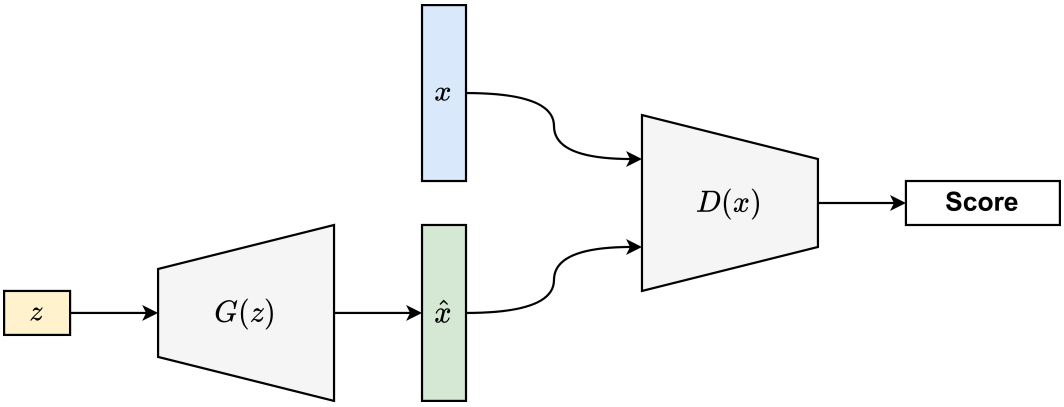


Figure 2: Illustration of the original GAN architecture

This process continues on for several iterations, where the generator generates better and better samples, while the discriminator also continually improves its ability to distinguish which samples are real. The generator's objective is to increase the loss (the metric measuring the error made in the discriminator's judgement), while the discriminator's objective is to decrease its loss. Since these two networks are pitted against each other, this is known as the *adversarial process.*

Referring to figure 2, the generator $G(z)$ first takes in the noise vector z (yellow), which is then passed through the generator, which generates a sample $\hat{x}$. Then the discriminator $D(x)$ is given two samples: a real sample x, and the generated sample $\hat{x}$, and returns a score between 0 and 1. The parameters of both networks are then updated accordingly using gradient descent through the adversarial process.

Once the discriminator is no longer able to distinguish between real and generated, the training process can be stopped, and the generator can be saved. It can then be used to generate unique new samples of data.

2.3 Limitations of GANs

It is important to understand that GANs are only able to generate data that is similar to the data that they are exposed to during training. For example, if a GAN is trained to generate images of dogs, and is trained only on images of non-spotted brown and grey dogs, it will not be able to generate any images of spotted dogs or white, despite being a generative model that is able to generate dog pictures. This is because GANs learn feature representations of the images, and then use them in new combinations to generate new images. Using the dog image example, a GAN could learn features such as ear length, fur style, colour on the provided training images, however it will only be able to identify features from those presented images, so if spotted or white dogs are not present in the training set, then the corresponding feature of 'spottiness' or 'fur whiteness' will not be learned. Consequently, this feature will not appear in any generated samples. It can be deduced that data generated by a GAN can only really be interpolations and combinations between existing features from the training dataset. This means that higher quality data can be generated with a GAN trained on a more extensive dataset containing a wide range of samples.

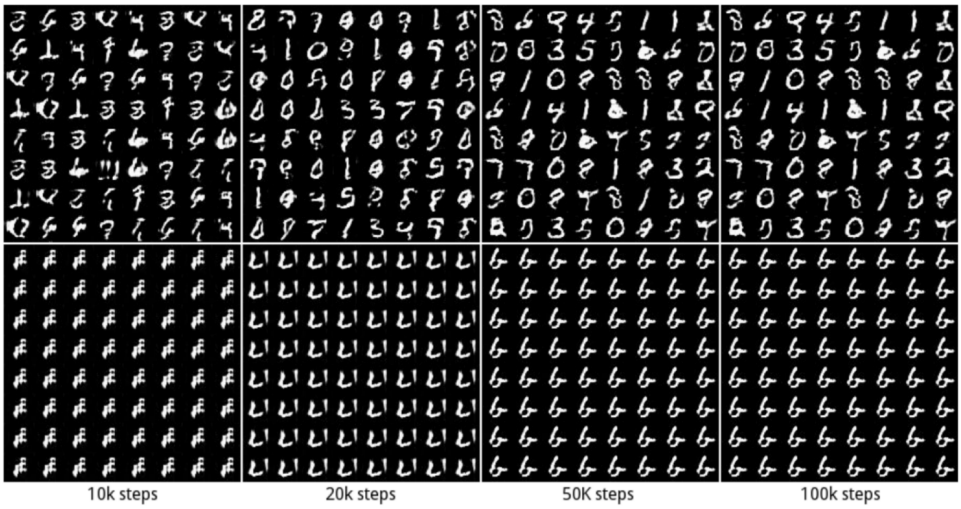


Figure 3: Modal collapse (top row is uncollapsed, bottom is collapsed) (Srihari n.d.)

GANs may also experience failure in the form of modal collapse. This occurs when the generator finds a *loophole* in the adversarial process and generates a small variety of very similar images. For example, only generating images of a single digit when the dataset contains digits 0-9 as shown in figure 3. In other terms, when the generator collapses, too many values of z correspond to identical samples $\hat{x}$ (the generated distribution), and are not able to fully model the real distribution (I. J. Goodfellow et al. 2014).

Another limitation of GANs is their training process can be rather unstable and may lead to situations in which the model is not able to converge, which can happen if the discriminator outperforms the generator too quickly in the process, not allowing the generator to improve its parameters. A similar situation can occur in reverse, where the generator outperforms the discriminator, although this is less likely. Onwards of 2014, papers have been published addressing these issues (discussed later).

2.4 Conditional GANs (cGANs)

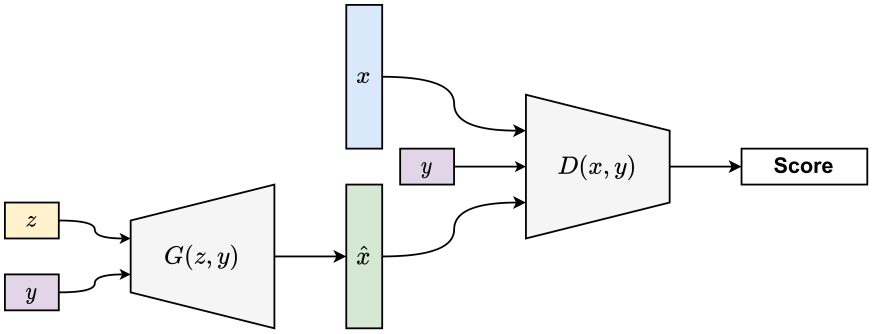


Figure 4: Conditional GAN illustration

Conditional GANs introduce a new parameter when training: the class labels y. This allows the GAN to generate images of a particular class, making the cGAN a type of *semi-supervised* model (Mirza and Osindero 2014). This modification is also effective at mitigating modal collapse as the model is forced to generate samples of specific classes which the discriminator is also expecting. This leads to the generator being penalised when generating images that do not correspond to the correct class. As a result, the generator is forced to generate a variety of classes, reducing the likelihood of modal collapse occurring since the generator will not be able to 'trick' the discriminator as easily with identical examples.

2.5 Deep Convolutional GANs (DCGANs)

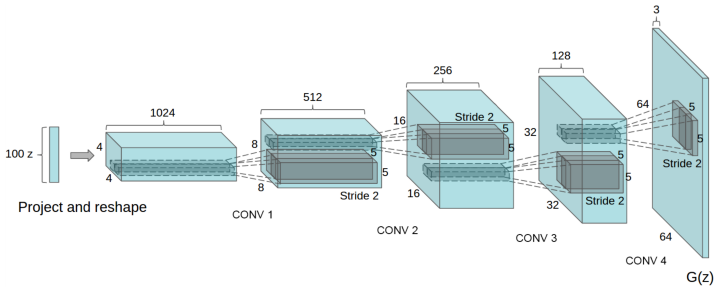
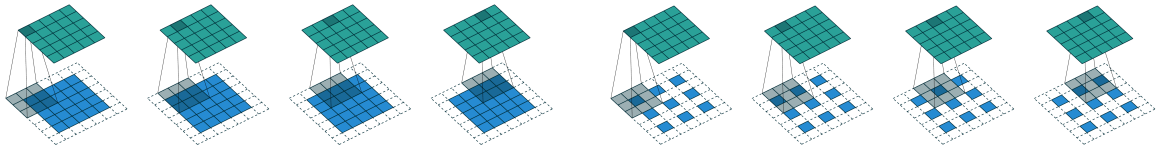


Figure 5: Deep Convolutional GAN generator architecture diagram (Radford, Metz, and Chintala 2015)

In 2015, the DCGAN was proposed for image generation. it improves upon the original GAN by replacing the multi-layer perceptron layers performing poorly for image generation, and instead uses strided convolutional layers (figure 6a) in the discriminator, and fractionally strided or transpose convolutional layers (figure 6b) in the generator. This alteration greatly improved generated image fidelity as 2-dimensional convolutional layers are specifically designed to for image processing.



(a) downscaling using strided convolutions (b) upscaling using transpose convolutions

Figure 6: Strided convolutions and transpose convolutions (Dumoulin and Visin 2016)

Figures 6a and 6b demonstrate the downscaling and upscaling capabilities of the two types of layers, eliminating the need for pooling layers which were specifically not included in the DCGAN. Additionally, these layers were chosen to allow the network to 'learn its own spatial down sampling' (Radford, Metz, and Chintala 2015). Due to these changes, there was a measurable improvement in generated image fidelity on the MNIST dataset, as well as the more complex LSUN dataset of indoor scene images.

2.6 Wasserstein GANs (WGANs)

The original GAN uses a loss function which quantifies the difference between the real and generated data distributions, but is prone to vanishing gradients. This means that the gradient of the loss function can initially be close to 0, which complicates the process of gradient descent because a small gradient can cause extremely small optimisation steps (Li 2022), leading to slower convergence or even non-convergence. This happens since the original GAN discriminator outputs values from 0-1, which can be very close to 0 when both distributions are far apart, as is typically the case during early iterations where very noisy samples are generated. This provides a weak gradient as shown in figure 7.

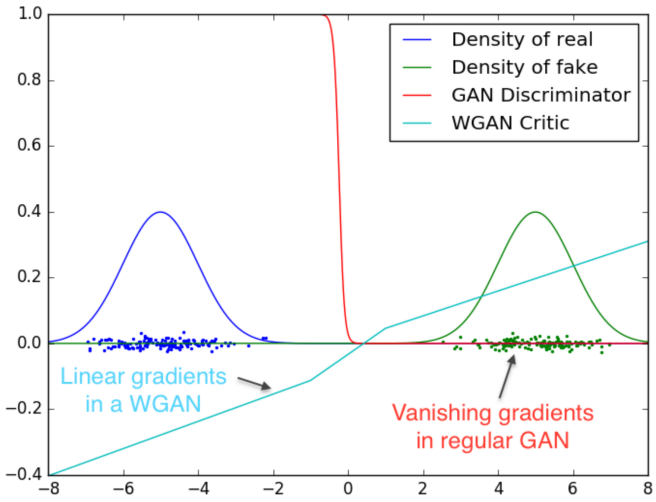


Figure 7: Comparison between GAN *discriminator* output and WGAN *critic* output (Arjovsky, Chintala, and Bottou 2017)

The WGAN proposes a new loss function which estimates the *Wasserstein Distance* or *Earth-mover distance* (EMD). To understand the Earth-mover distance, the two distributions of real and generated images can be imagined as two piles of dirt or *earth*. The metric would then approximate how much 'mass' needs to be moved so that both distributions would be identical. Effectively, the distance is the 'cost of the optimal transport plan' (Arjovsky, Chintala, and Bottou 2017).

This new function produces a more linear curve, easing the process of gradient descend due to stronger initial gradients. Additionally, the discriminator is referred to as the 'critic' in the WGAN, since it no longer discriminated between real and generated, but rather returns a score not limited between the range of 0 and 1 as was the case with the original GAN proposed by Goodfellow.

2.7 Measuring GAN Performance

Measuring the quality of a GAN's output is a relatively difficult task, as this is a qualitative observation, especially in the case of image data or audio data. While metrics do exist to quantify the fidelity of GAN generated images, they are not perfect, though they do provide a useful indication.

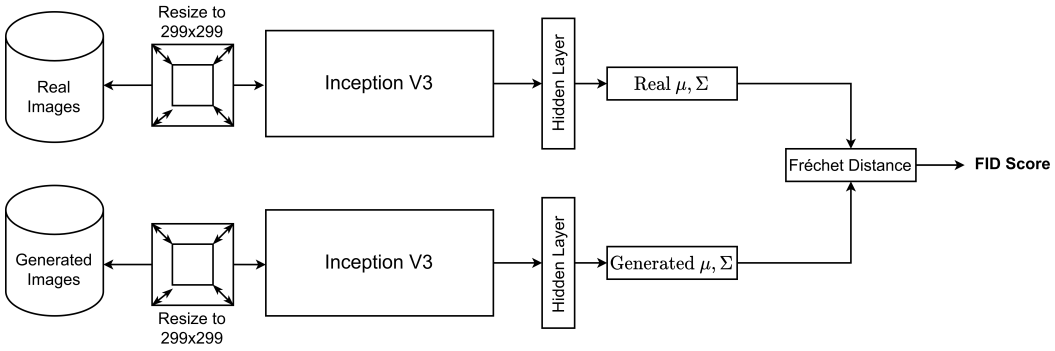


Figure 8: Illustration of FID metric calculation adapted from Parmar, Zhang, and Zhu 2021

Currently, the standard metric is the *Fréchet Inception Distance* score (FID), which is an improvement upon the *Inception Score* which used to be the 'de-facto' GAN evaluation metric. They both use the state-of-the-art Inception-v3 network. Specifically, the FID metric quantifies the difference between the real and generated distributions.

The Inception-v3 network serves as a feature extractor, identifying elements from the images (figure 8). The Fréchet Distance is then computed on the multivariate distributions of the real and generated image feature representations (which are vectors outputted by a hidden layer in the Inception network). This operation is described by the expression below, where $\mu_{\text{generated}}$ and $\Sigma_{\text{generated}}$ are the mean and covariance of the generated image, and μ_{real} and Σ_{real} are that of the real images.

$$\text{FID} = \underbrace{||\mu_{\text{real}} - \mu_{\text{generated}}||_2^2}_{\text{Distribution Means}} + \underbrace{\text{Tr}(\Sigma_{\text{real}} + \Sigma_{\text{generated}} - 2(\Sigma_{\text{real}}\Sigma_{\text{generated}})^{\frac{1}{2}})}_{\text{Distribution Covariance Matrices}} \quad \text{(Parmar, Zhang, and Zhu 2021)}$$

The lower the FID score, the more similarity there is between the two distributions, i.e. lower scores are desirable as they indicate good performance.

3 Methodology

3.1 Experimental Overview

The following experiment involves a custom implementation of a GAN in PyTorch (Python). This model was trained on varying dataset sizes, and the FID score was measured at each dataset size.

3.2 Dataset Selection

The Extended-MNIST (EMNIST) dataset was selected for use in this experiment which consists of grey-scale images of handwritten alphanumeric characters (A-Z, a-z, 0-9) (Cohen et al. 2017). This dataset is an extension of the MNIST dataset, which only consisted of 10 classes (handwritten digits 0-9). The EMNIST dataset was selected because of its use as a benchmark dataset, as well as being slightly less trivial than the original MNIST dataset due to its increased number of classes, and therefore increased number of features for a GAN to learn.

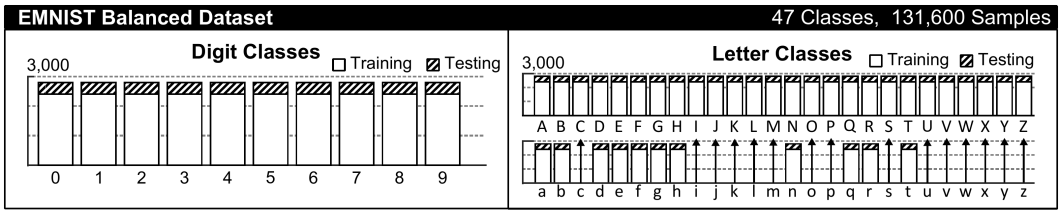


Figure 9: Quantities of samples per class: 'balanced' split (Cohen et al. 2017)

Specifically the 'balanced' split of the EMNIST dataset will be used due to its equal number of samples per class evident by the papers provided specifications in figure 9. Classes with upper and lowercase letters are also merged (for example, 's'→'S').

3.3 Experimental Variables

Independent Variable: The quantity of image samples from the EMNIST dataset supplied to the GAN during training. This was done by truncating the dataset to smaller and smaller sizes in increments of 10% (exact values shown in table 1).

Table 1: Independent variable increments

Proportions of the EMNIST Dataset									
10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
11,280	22,560	33,840	45,120	56,400	67,680	78,960	90,240	10,1520	112,800

The number of samples during training was done by removing references to files in the PyTorch `torchvision.datasets.EMNIST` object to achieve the desired number of references to files (number of samples). This process ensured that the distribution of classes is equal, since the dataset's references are not in order.

Dependent Variable: The FID score of the GAN after training for a dataset of a specific size, computed once 15 elapsed epochs of training.

This was measured using the `clean-fid` python library supported by PyTorch, which automatically scales the generated images from single-channel 28x28, to three-channel 299x299 images required for the Inception-v3 model used in the FID metric.

The library implementation was chosen for its improved image-processing methods, shown to impact the FID score (Parmar, Zhang, and Zhu 2021). Such considerations were not made in competing libraries. Additionally, the library facilitates pre-computation of statistics for a given dataset which saves unwelcome time and compute resources.

Generated image samples for each class were also be generated and saved for manual visual inspection and qualitative analysis.

Control Variable(s):

Number of Epochs: Each GAN configuration was trained for exactly 15 epochs, which provided a sufficient quantity of iterations for the model to converge. This meant that the model was exposed to every single unique sample in the dataset exactly 15 times. Keeping this constant was important as it made comparison between the different GAN configurations more accurate.

GAN Hyperparameters: Hyperparameters must be kept constant as they significantly impact the way a GAN will converge. This experiment in particular employed a WGAN due to its reduced sensitivity to hyperparameters. Since all GANs converged, it allowed for the comparison of FID scores to be more fair.

Architecture: Similar to hyperparameters, the architecture must also be kept constant, since it impacts convergence. The experiment's GAN architecture was chosen as it has shown favourable performance for handwritten digits.

3.4 Experimental Design

This GAN implementation uses convolutional layers (from the DCGAN) since it is designed to generate images. The Wasserstein loss function was also used, as it increased the likelihood of the model converging and reduced the chance of modal collapse.

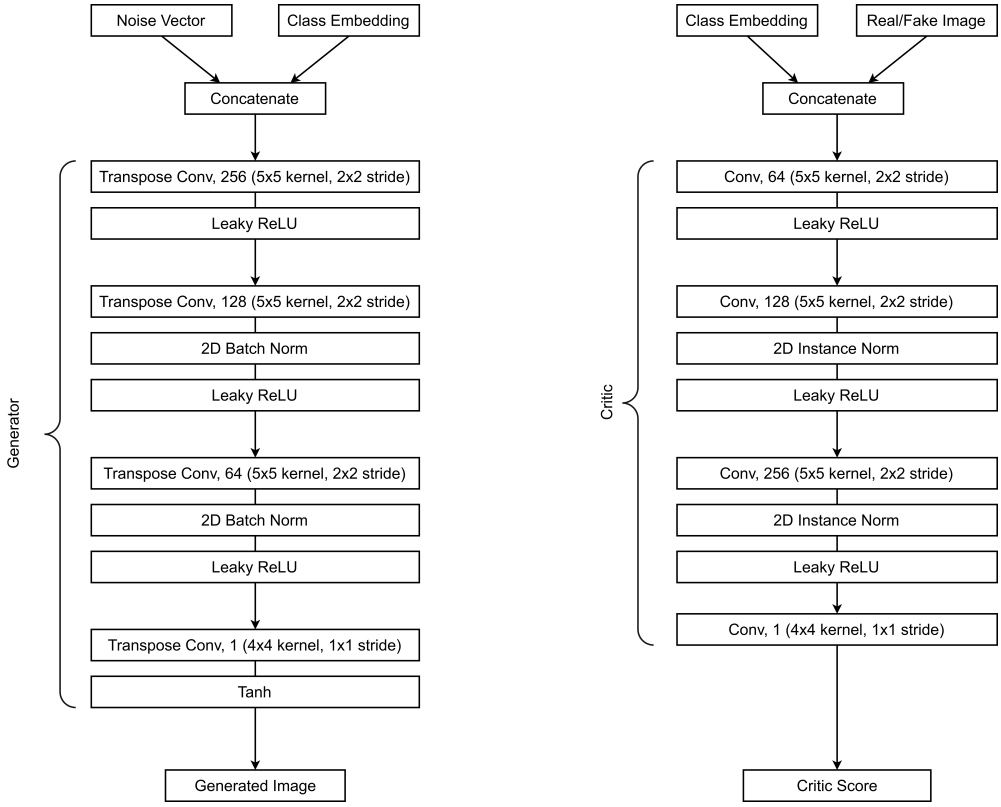


Figure 10: Architectural diagram of the custom GAN implementation

The architecture used was adapted from that found in Gulrajani et al., 2017, but was modified to use the EMNIST dataset as it was originally designed for the CIFAR-10 dataset using 32x32 colour images. Changes were primarily made to the first layers to accommodate the smaller images size and label encoding required for making a cGAN. Making the GAN conditional was chosen as it reduces modal collapse as described earlier.

For training, hyperparameters (shown in table 2) from Radford, Metz, and Chintala 2015 and Gulrajani et al. 2017 were taken as as they both showed favourable results.

Table 2: Hyperparameters used for training

$\boldsymbol{\lambda}$	$\boldsymbol{n_{\text{critic}}}$	$\boldsymbol{\alpha}$	$\boldsymbol{\beta_1}$	$\boldsymbol{\beta_2}$
10	5	1E-04	0.0	0.9
Gradient penalty coefficient	Ratio of critic to generator training iterations	learning rate of optimiser	Adam optimiser parameters	

Specifically, the GAN implementation employs the Wasserstein loss function, but with the *gradient penalty* proposed in Gulrajani et al. 2017, which was designed to replace the original WGAN papers weight clipping that was suboptimal as noted by the authors. The WGAN paper also proposed that the critic is to be trained for n_{critic} more times than the generator, to provide the generator with more accurate feedback. The value of n_{critic} was 5 in the paper, and was also used in this experiment. The following loss function was constructed by combining the techniques from each of the papers.

$$C_{\text{loss}} = \underbrace{C(G(z,y),y) - C(x,y)}_{\text{Wasserstein distance estimate}} + \underbrace{\lambda(\|\nabla C(\tilde{x},y)\|_2 - 1)^2}_{\text{Gradient penalty term}}$$

The 'Wasserstein distance estimate' remains the same from the original WGAN paper Arjovsky, Chintala, and Bottou 2017, however the generator $G(z,y)$ and critic $C(x,y)$, also take in the labels y, since the GAN is conditional. The next element term is the gradient penalty, which uses the norm of the gradients of the critic's weights, when inputting an interpolation of a real and generated image. The interpolation is done using a random number, $\epsilon \in [0,1]$, resulting in an interpolated image $\tilde{x} = \epsilon x + (1-\epsilon)\hat{x}$, where x and $\hat{x}$ are real and generated images.

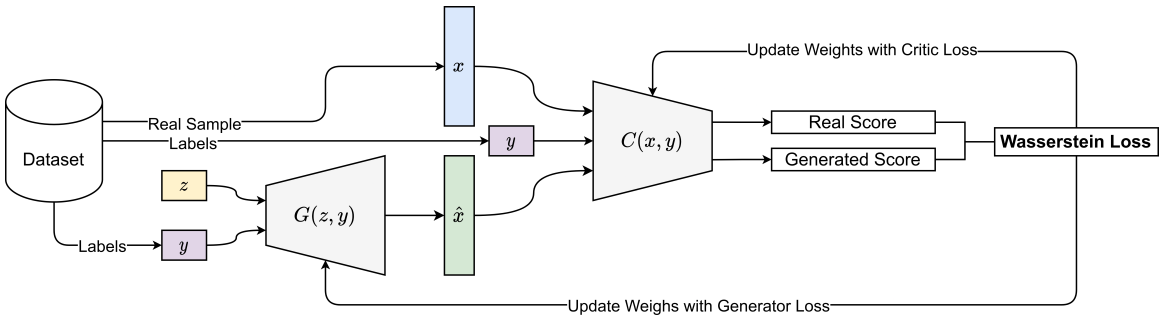


Figure 11: Training diagram of the custom GAN

Figure 11 shows the training cycle used to train the GAN which has been implemented in Python (refer to appendix). During the training of a batch, data the generator generates images for the corresponding labels for real samples in the batch. The critic then gives a 'score' to real and generated images taking class labels into consideration. Using these scores, the loss is calculated, and the critic and generator's network parameters are then optimised accordingly to the adversarial process.

3.5 Experimental Procedure

Part I: Training the GAN

1. Train the GAN on 112,800 samples for 15 epochs

   Note: Using Tensorboard, the Wasserstein distance was monitored for convergence

2. Once training has completed, save the generator state

   Note: Stored in the folder `checkpoints/dataset_size_112800/epoch_15.pth`

3. Repeat step 1-3 for all 10 dataset sizes (replace 112,800 with the appropriate value)

Part II: Gathering FID scores

1. Load the checkpoint of the GAN trained on 112,800 samples
2. Generate samples using the saved generator

   Note: The exact number of samples in the real dataset (112,800) were generated with equal class distributions

3. Calculate the FID score given the generated samples and real images from the dataset
4. Repeat for all saved generator states

Training this model required powerful hardware, so a cloud virtual machine was used with 8 Intel Xeon E5-2623 v4 cores, 30 GB of RAM, and an Nvidia Quadro P4000 with 8GB of vRAM. This was able to sufficiently train each model configuration, taking approximately 16 hours to do so including FID calculation.

4 Experimental Results

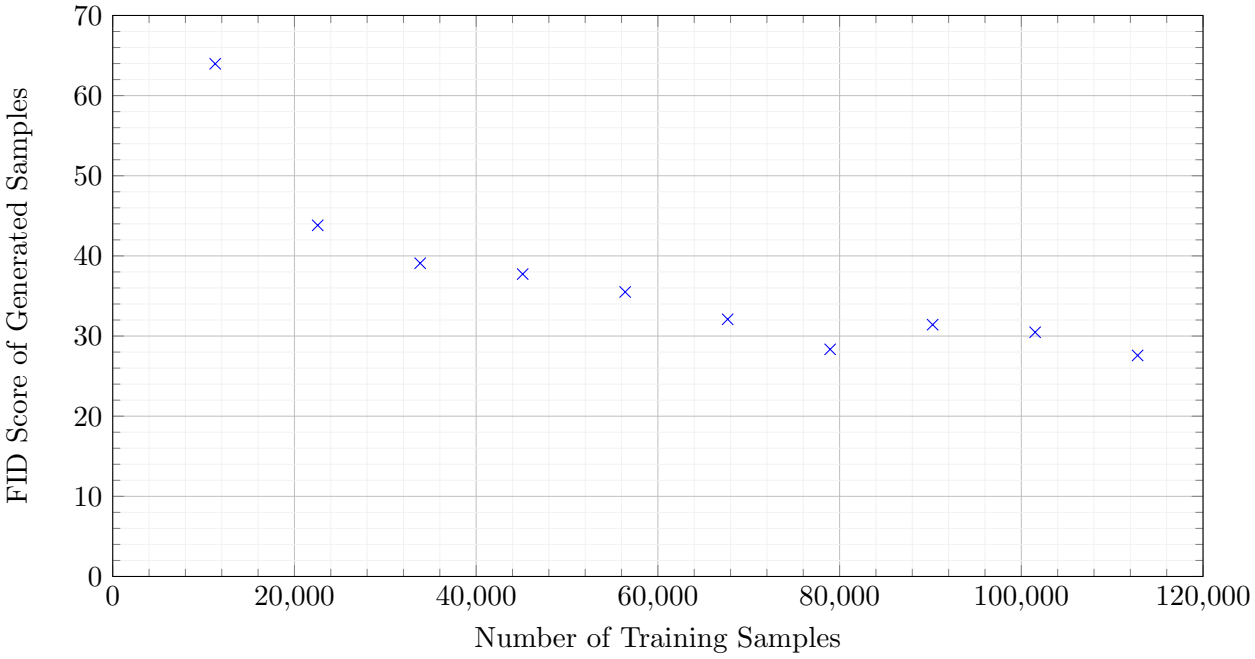
4.1 Quantitative Data

Table 3: Raw Experimental Data

Dataset Samples		FID Score	Δ FID score
Percentage of Total (%)	**Number of Samples**		
10	11,280	63.9696288	N/A
20	22,560	43.81612586	20.1535
30	33,840	39.08653845	4.729587
40	45,120	37.73255476	1.353984
50	56,400	35.50103584	2.231519
60	67,680	32.09195388	3.409082
70	78,960	28.34171738	3.750237
80	90,240	31.41788896	3.076172
90	101,520	30.47313068	0.944758
100	112,800	27.58175179	2.891379

Using the data from table 3, a graph can be plotted to visualise the relationship between the independent and dependent variables.

Graph 1: Plot of FID Score vs. Number of Training Samples

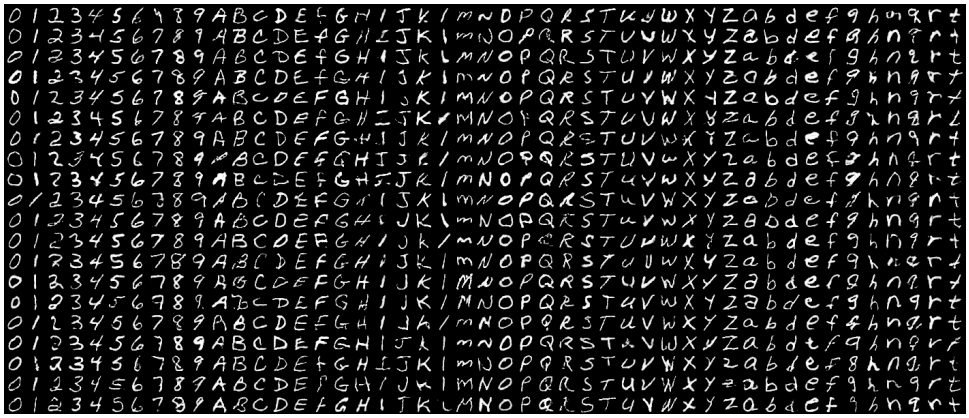


As is visible in the graph 1, the FID score clearly decreases as the number of training samples increases (indicating increased fidelity). It is also important to notice that the drop in FID is significantly higher at lower numbers of training samples, than at higher numbers (between 11,280 and 22,560 FID drops by 20.1 FID points).

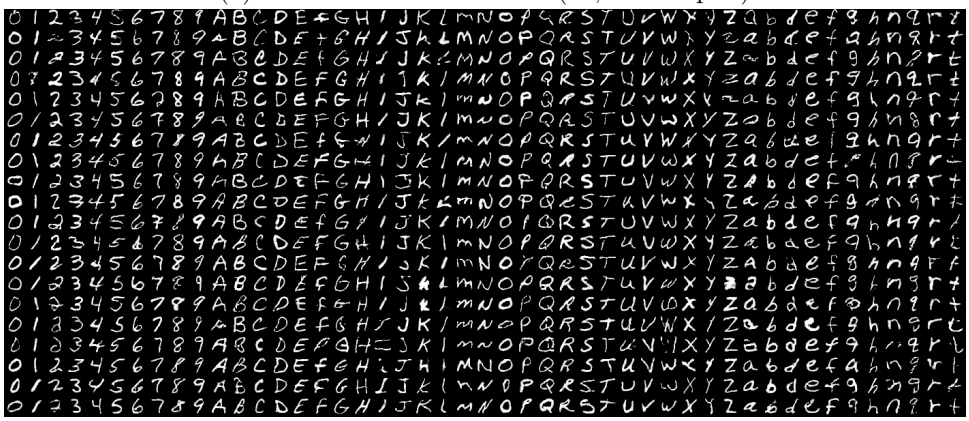
The reason for this may be that at a large enough quantity, the reduced number of samples is able to approximate the features found within the whole dataset, so the GAN is able to learn features of the dataset and is then able to generalise. The improvement in fidelity caused by an incremental increase in new samples will not be as profound at higher sample sizes, since the added samples will only marginally increase the features the GAN is able to learn, showing only a small increase in fidelity or decrease in FID score.

4.2 Qualitative Data

Since the FID metric only gives an indication of visual fidelity (in comparison to real data), the following generated samples were inspected manually to qualitatively evaluate the impact of reduced dataset size on image fidelity.



(a) 10% of EMNIST dataset (11,280 samples)



(b) 100% of EMNIST dataset (112,800 samples)

Figure 12: Generated samples after 15 epochs

Looking at figure 12, it can be noticed that the GAN specifically performed poorly at generating certain characters when the dataset was reduced to 10% of the original size, namely 'B'. 'J', and 'M', while instances of '1' do not visually vary as much between 12a and 12b.

5 Discussion

5.1 Conclusion & Evaluation

It can be concluded that the impact of reducing the number of samples a GAN is exposed to during training, has a significant impact on the fidelity of its generated samples, with a measured drop of 36.38 FID score points between a GAN trained on the full EMNIST dataset (112,800 samples) for 15 epochs, than on GANs trained on only 10% of that amount (11,280 samples). In other words, a 90% decrease in dataset samples, results in a sizeable 56.81% increase in FID, indicating a substantial loss in image fidelity, thus providing an answer to the original research question.

However, it is also important to understand the limitations of the experiment. Firstly, the FID score is not the most reliable metric for quantifying the fidelity of GAN generated image fidelity, not only because it is difficult to quantify fidelity (as mentioned in the Background Theory section), but because it employs the Inception-v3 network as a feature extractor, which was specifically designed for use with the ImageNet dataset – containing images of common day objects, and not handwritten characters. This means that its feature extraction capability was limited when using samples generated from the EMNIST dataset, as it contains images vastly different from those found in the ImageNet dataset. Despite this, the metric did provide an accurate indication of change in image fidelity, which was backed up by qualitative observations of the generated samples.

For example the FID score dropped by 57.14%, as a result of only using 10% of the total dataset, however this did not correspond to image fidelity dropping by 57.14% referring to the qualitative samples generated. Interestingly, it is difficult to qualitatively judge the degree to which the fidelity changed as a percentage, highlighting the key problem present with using a single numerical value to assess the quality of an image.

Table 4: Comparison of total iterations vs. total samples

Total Samples	**Batches (of 64)**	**Total Iterations**
11280	176	2644
22560	353	5288
33840	529	7931
45120	705	10575
56400	881	13219
67680	1058	15863
78960	1234	18506
90240	1410	21150
101520	1586	23794
112800	1763	26438

During the experiment the number of epochs the GAN was trained for was kept constant (at 15), but this did not ensure that the number of iterations or batches the GAN was trained on was also constant, since this quantity depends on the size of the dataset (the dependent variable) which was altered in the experiment. The number of iterations has a significant impact on final GAN performance, since the longer it is trained, the more samples the GAN is exposed to, the better samples it will be able to generate. Referring to table 4, there is an almost $\times 10$ reduction in total iterations when 10% of the dataset samples are used, which could have impacted the performance of that particular GAN configuration. It may be true if each GAN was trained on a constant number of iterations, rather than epochs, the performance of low-sample-count configurations may have improved, yielding lower FID scores.

Additionally, the EMNIST dataset used in the experiment is relatively simple, only containing low-resolution images, and is considered somewhat trivial, primarily being used for benchmarking purposes. In terms of the experimental findings, it is possible that they may not extrapolate as well to larger, more complex datasets used with GANs such as CIFAR-100 (low-resolution colour images of objects) or even FFHQ (high resolution human faces), since they contain many more features than the EMNIST dataset.

5.2 Improvements & Future Work

This paper focused on GAN applications in the field of computer vision, using GANs to generate image data, however it would also be possible to conduct a similar experiment on numerical data, audio data or even natural language data. To confirm whether or not the experimental findings of this paper extrapolate to larger datasets such as FFHQ or CelebA which are commonly used in contemporary GAN research. Such datasets were not investigated in this paper due to hardware limitations and constraints.

In addition, an investigation could also be conducted with a constant number of GAN training iterations rather than a constant number of epochs, which led to varied iteration counts between trials as discussed in the previous section. However, this would also increase the number of times the model is exposed to the same unique sample, meaning the frequency each unique sample is exposed to the GAN increases with smaller dataset sizes. This could potentially impact the results, through in a different way in comparison to keeping the number of epochs constant.

Future work can also be done in combination with the 'Few-shot GAN' introduced in Robb et al. 2020, which is particularly designed to function in situations where training data is scarce (100 to 1,000 samples), although it requires a GAN trained on another similar dataset. Carrying out a similar investigation with an FSGAN may lead to a different outcome, and may produce higher fidelity images in data-limited scenarios when compared to the more standard GAN architecture investigated in this paper.

Similarly, other network architecture configurations could be investigated using different activation functions and layer parameters, since many other GANs apart from the Wasserstein GAN have been proposed, such as the DRAGAN architecture (introduced in Kodali et al. 2017), and the Spectral Normalisation GAN (introduced in Miyato et al. 2018), which both attempt to achieve the same goal of reducing the hyperparameter sensitivity. The effects of alternative hyperparameters may also provide valuable insight into how they affect convergence on lower training sample sizes, as they can significantly impact the initial steps of the training process.

6 References

Arjovsky, Martin, Soumith Chintala, and Léon Bottou (2017). *Wasserstein GAN*. DOI: 10.48550/ARXIV.1701.07875. URL: https://arxiv.org/abs/1701.07875.

Cohen, Gregory et al. (2017). *EMNIST: an extension of MNIST to handwritten letters*. DOI: 10.48550/ARXIV.1702.05373. URL: https://arxiv.org/abs/1702.05373.

Dumoulin, Vincent and Francesco Visin (2016). *A guide to convolution arithmetic for deep learning*. DOI: 10.48550/ARXIV.1603.07285. URL: https://arxiv.org/abs/1603.07285.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Goodfellow, Ian J. et al. (2014). *Generative Adversarial Networks*. DOI: 10.48550/ARXIV.1406.2661. URL: https://arxiv.org/abs/1406.2661.

Gulrajani, Ishaan (June 2017). *Improved Training of Wasserstein GANs*. URL: https://github.com/igul222/improved_wgan_training.

Gulrajani, Ishaan et al. (2017). *Improved Training of Wasserstein GANs*. DOI: 10.48550/ARXIV.1704.00028. URL: https://arxiv.org/abs/1704.00028.

Karpathy, Andrej et al. (June 2016). *Generative Models*. URL: https://openai.com/blog/generative-models/.

Kodali, Naveen et al. (2017). *On Convergence and Stability of GANs*. DOI: 10.48550/ARXIV.1705.07215. URL: https://arxiv.org/abs/1705.07215.

Li, Katherine (July 2022). *Vanishing and exploding gradients [practical guide]*. URL: https://neptune.ai/blog/vanishing-and-exploding-gradients-debugging-monitoring-fixing.

Mirza, Mehdi and Simon Osindero (2014). *Conditional Generative Adversarial Nets*. DOI: 10.48550/ARXIV.1411.1784. URL: https://arxiv.org/abs/1411.1784.

Miyato, Takeru et al. (2018). *Spectral Normalization for Generative Adversarial Networks*. DOI: 10.48550/ARXIV.1802.05957. URL: https://arxiv.org/abs/1802.05957.

Parmar, Gaurav, Richard Zhang, and Jun-Yan Zhu (2021). *On Aliased Resizing and Surprising Subtleties in GAN Evaluation*. DOI: 10.48550/ARXIV.2104.11222. URL: https://arxiv.org/abs/2104.11222.

Persson, Aladdin (Jan. 2021). *Machine Learning Collection*. URL: https://github.com/aladdinpersson/Machine-Learning-Collection.

Radford, Alec, Luke Metz, and Soumith Chintala (2015). *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. DOI: 10.48550/ARXIV.1511.06434. URL: https://arxiv.org/abs/1511.06434.

Robb, Esther et al. (2020). *Few-Shot Adaptation of Generative Adversarial Networks*. DOI: 10.48550/ARXIV.2010.11943. URL: https://arxiv.org/abs/2010.11943.

Srihari, Sargur N. (n.d.). *GAN: Mode Collapse*. URL: https://cedar.buffalo.edu/~srihari/CSE676/22.3-GAN%20Mode%20Collapse.pdf.

Yang, Jinhyeok (Nov. 2019). *GANs Tutorial*. URL: https://github.com/Yangyangii/GAN-Tutorial.

7 Appendices

The architecture of the GAN used in this paper was based on that found in Gulrajani et al. 2017, which was implemented in Tensorflow in its respective git repository: Gulrajani 2017. The two repositories Persson 2021 and Yang 2019 were then referenced to implement the model in PyTorch.

7.1 main.py

```
import train
from tqdm import tqdm

EPOCHS = 15
DATASET_SIZES = [int((i + 1) * 11280) for i in range(10)]

for size in tqdm(
    DATASET_SIZES, total=len(DATASET_SIZES), desc=f"Configurations", unit="configs"
):
    train.training(size, EPOCHS, "./checkpoints")
```

7.2 train.py

```
import torch, torchvision, os
from models import Critic, Generator

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

from tqdm.auto import tqdm, trange
from torch.utils.tensorboard import SummaryWriter

torch.backends.cudnn.benchmark = True
NUM_WORKERS = 8
device = "cuda" if torch.cuda.is_available() else "cpu"
print("USING: " + device)


# HYPER PARAMETERS
BATCH_SIZE = 64
BETAS = (0.0, 0.9)
LR = 1e-4

Z_DIM = 100
EMBED_SIZE = 100
LAMBDA = 10
N_CRITIC = 5

NUM_CLASSES = 47


def training(dataset_size: int, epochs: int, checkpoint_dir: str):

    writer = SummaryWriter(f"logs/runs/emnist_{dataset_size}")

    fixed_noise = torch.randn((NUM_CLASSES**2, Z_DIM, 1, 1), device=device)
    fixed_labels = torch.zeros(
        (NUM_CLASSES, NUM_CLASSES), dtype=torch.int32, device=device
    )
    for y in range(fixed_labels.shape[0]):
```

```
        for x in range(fixed_labels.shape[1]):
            fixed_labels[y][x] = y
    fixed_labels = fixed_labels.view(-1)

    dataset = datasets.EMNIST(
        download=True,
        root="./data",
        split="balanced",
        transform=transforms.Compose(
            [
                transforms.Lambda(lambda x: transforms.functional.rotate(x, 270)),
                transforms.Lambda(lambda x: transforms.functional.hflip(x)),
                transforms.ToTensor(),
                transforms.Normalize((0.5,), (0.5,)),
            ]
        ),
    )

    dataset.data = dataset.data[:dataset_size]
    dataset.targets = dataset.targets[:dataset_size]

    dataloader = DataLoader(
        dataset=dataset,
        batch_size=BATCH_SIZE,
        shuffle=True,
        drop_last=True,
        num_workers=NUM_WORKERS,
        pin_memory=True,
    )

    G = Generator(z_dim=Z_DIM, embed_size=EMBED_SIZE, n_classes=NUM_CLASSES).to(device)
    C = Critic(n_classes=NUM_CLASSES).to(device)

    G_opt = torch.optim.Adam(G.parameters(), lr=LR, betas=BETAS)
    C_opt = torch.optim.Adam(C.parameters(), lr=LR, betas=BETAS)

    step = 0
    for epoch in range(epochs):
        loop = tqdm(
            enumerate(dataloader),
            desc=f"Epoch [{epoch+1}/{epochs}]",
            total=len(dataloader),
            unit="batch",
            leave=False,
        )
        for idx, (x, y) in loop:
            x = x.to(device)
            y = y.to(device)

            for _ in range(N_CRITIC):
                z = torch.randn((BATCH_SIZE, Z_DIM, 1, 1), device=device)
                x_ = G(z, y)
                C_x = C(x, y).view(-1)
                C_x_ = C(x_, y).view(-1)

                eps = torch.randn((BATCH_SIZE, 1, 1, 1), device=device).repeat(
                    1, 1, 28, 28
                )
                x_int = eps * x + (1 - eps) * x_

                C_x_int = C(x_int, y)
                gradient = torch.autograd.grad(
                    inputs=x_int,
```

```
                    outputs=C_x_int,
                    grad_outputs=torch.ones_like(C_x_int),
                    create_graph=True,
                    retain_graph=True,
                )[0]
                gradient = gradient.view(gradient.shape[0], -1)
                norm = gradient.norm(2, dim=1)
                gp = torch.mean((norm - 1) ** 2) * LAMBDA

                w_dist = torch.mean(C_x_) - torch.mean(C_x)
                C_loss = w_dist + gp

                C_opt.zero_grad()
                C_loss.backward(retain_graph=True)
                C_opt.step()

            C_G_z = C(x_, y).view(-1)
            G_loss = -torch.mean(C_G_z)

            G_opt.zero_grad()
            G_loss.backward()
            G_opt.step()

            if step % 10 == 0:
                loop.set_postfix(
                    C_l=f"{C_loss.item():.4f}",
                    G_l=f"{G_loss.item():.4f}",
                    W_l=f"{w_dist.item():.4f}",
                )
                writer.add_scalar("Critic Loss", C_loss.item(), global_step=step)
                writer.add_scalar("Generator Loss", G_loss.item(), global_step=step)
                writer.add_scalar(
                    "Wasserstein Distance Estimate", w_dist.item(), global_step=step
                )

            if step % 250 == 0:
                with torch.no_grad():
                    img = G(fixed_noise, fixed_labels)
                    grid = torchvision.utils.make_grid(img, nrow=NUM_CLASSES)
                    writer.add_image("Generated Images", grid, global_step=step)

            step += 1

        path = os.path.join(checkpoint_dir, f"emnist_{dataset_size}")
        if not os.path.exists(path):
            os.makedirs(path)
        torch.save(G.state_dict(), os.path.join(path, f"epoch_{epoch}.pth"))
```

7.3 models.py

```
import torch
from torch import nn


class Generator(nn.Module):
    def __init__(
        self, z_dim=100, img_size=28, n_channels=1, n_classes=10, embed_size=100
    ):
        super(Generator, self).__init__()
        self.img_size = img_size
```

```
        self.embed = nn.Embedding(n_classes, embed_size)
        self.gen = nn.Sequential(
            nn.ConvTranspose2d(z_dim + embed_size, 256, 4, 2, 0, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(256, 128, 5, 2, 2, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(128, 64, 5, 2, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(64, n_channels, 5, 2, 2, 1, bias=False),
            nn.Tanh(),
        )

    def forward(self, z, y):
        embedding = self.embed(y).unsqueeze(2).unsqueeze(3)
        z = torch.cat([z, embedding], dim=1)
        return self.gen(z)


class Critic(nn.Module):
    def __init__(self, n_channels=1, n_classes=10, img_size=28):
        super(Critic, self).__init__()
        self.img_size = img_size
        self.embed = nn.Embedding(n_classes, img_size * img_size)
        self.critic = nn.Sequential(
            nn.Conv2d(n_channels + 1, 64, 5, 2, 2, bias=False),
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, 128, 5, 2, 2, bias=False),
            nn.InstanceNorm2d(128, affine=True),
            nn.LeakyReLU(0.2),
            nn.Conv2d(128, 256, 5, 2, 2, bias=False),
            nn.InstanceNorm2d(256, affine=True),
            nn.LeakyReLU(0.2),
            nn.Conv2d(256, 1, 4, 1, 0, bias=False),
        )

    def forward(self, x, y):
        embedding = self.embed(y).view(y.shape[0], 1, self.img_size, self.img_size)
        x = torch.cat([x, embedding], dim=1)
        return self.critic(x)
```

7.4 fid.py

```
from cleanfid import fid
import torch, os, shutil, torch
import models

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

from tqdm import tqdm, trange

device = "cuda" if torch.cuda.is_available() else "cpu"


GEN_BATCH_SIZE = 11280
FID_BATCH_SIZE = 188
```

```
if not fid.test_stats_exists("emnist", mode="clean"):
    dataloader = DataLoader(
        dataset=datasets.EMNIST(
            download=True,
            root="./data",
            split="balanced",
            transform=transforms.Compose(
                [
                    transforms.Lambda(lambda x: transforms.functional.rotate(x, 270)),
                    transforms.Lambda(lambda x: transforms.functional.hflip(x)),
                    transforms.ToTensor(),
                ]
            ),
        ),
        batch_size=1000,
        shuffle=True,
        drop_last=True,
        num_workers=4,
        pin_memory=True,
    )

    img_id = 0
    if not os.path.exists("./temp"):
        os.makedirs("./temp")
    for batch, _ in tqdm(dataloader, desc=f"Processing Dataset", unit="img"):
        for img in batch:
            save_image(img, f"./temp/{img_id}.png")
            img_id += 1

    # precompute FID
    fid.make_custom_stats("emnist", "./temp", mode="clean", batch_size=FID_BATCH_SIZE)

    # Remove folder
    shutil.rmtree("./temp")


scores = []


paths = [
    [str(q.path) for q in sorted(os.scandir(str(p.path)), key=os.path.getmtime)]
    for p in sorted(os.scandir("./checkpoints"), key=os.path.getmtime)
]
ckpt_paths = [p[-1] for p in paths]


for path in tqdm(ckpt_paths, desc="Configs"):
    # generate in temp folder
    generator = models.Generator(z_dim=100, n_classes=47, embed_size=100).to(device)
    generator.load_state_dict(torch.load(path, map_location=device))
    generator.eval()

    if not os.path.exists("./temp"):
        os.makedirs("./temp")

    with torch.no_grad():
        img_id = 0
        for _ in trange(int(112800 / GEN_BATCH_SIZE), desc="Images", leave=False):
            z = torch.randn((GEN_BATCH_SIZE, 100, 1, 1), device=device)
            y = torch.randint(
                size=(GEN_BATCH_SIZE,), low=0, high=47, dtype=torch.int32, device=device
```

```
            )
            imgs = generator(z, y)
            del z, y

            for img in tqdm(imgs, total=len(imgs), desc="Saving", leave=False):
                save_image(img, f"./temp/{img_id}.png")
                img_id += 1
            del imgs

    # calculate fid from precompute
    score = fid.compute_fid(
        "./temp",
        dataset_name="emnist",
        mode="clean",
        dataset_split="custom",
        num_workers=8,
        batch_size=FID_BATCH_SIZE,
        device=device,
    )

    scores.append(score)
    print(score)

    shutil.rmtree("./temp")


print("SCORES:")
for idx, score in enumerate(scores):
    print(idx, score)
```