Implementation of GANs for Generating Digit Images

Chirong Zhang, Zhichao Liu, Yunxiao Zhao, Yusang Mao [*]
{cz2533, zl2686, yz3380, ym2694}@columbia.edu

December 16, 2019

Abstract

Generative Adversarial Networks (GANs) are among the most popular topics in Deep Learning nowadays. There has been a tremendous increase in the number of papers being published on GANs over the last two years. Despite GANs have achieved major successes in a great variety of problems, the most appropriate choice of a GAN architecture is still not well understood. In this project, we explore various GANs to generate number digit images using training samples from two well-known datasets. We discover that GANs, with different architecture, are not easy to train and evaluate even though they can produce satisfactory results. Certain hyperparameters play an important role in helping GANs learn effectively, not only for our own DCGAN model developed from adding up simple functional layers but also models found in recent papers with sophisticated designs. The results are demonstrated in threefold: quality of generated images, loss during the training process, and the Inception Score.

1 Introduction

1.1 Problem Description

In machine learning, the Generative Model is a powerful way of learning any kind of data distribution using unsupervised learning approach and it has achieved tremendous success in just a few years. All types of generative models aim at learning the true data distribution of the training set so as to generate new data points with some variations. However, it is not always efficient and even possible to learn the exact distribution of our data either implicitly or explicitly and so we try to model a distribution that is as similar as possible to the true data distribution. For this, we can leverage the power of deep neural networks to learn a function that can help approximate the model distribution to the true distribution better.

[*]Project Github Repository: `https://github.com/Trccc/5242Project`.

Two of the most commonly used and efficient approaches nowadays are Variational Autoencoder (VAE) and Generative Adversarial Networks (GANs). VAE aims at maximizing the lower bound of the data log-likelihood and GANs aim at achieving an equilibrium between Generator and Discriminator. In this project, we would use GANs and its derivatives to generate new grey-scale/RGB images given a simple noise under specific distribution (e.g. Gaussian).

The training datasets we use in this project are MNIST and SVHN. MNIST is a dataset of grey-scale handwritten digits (0-9) which has a training set of 60,000 examples.

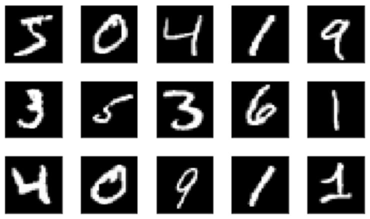

Figure 1.1: Training Samples from MNIST

The Street View House Numbers (SVHN) [10] is a dataset of real-world digits and numbers, which is obtained from house numbers in Google Street View images. There are 73,257 training samples in total.

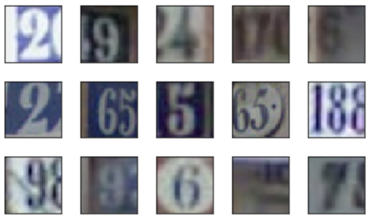

Figure 1.2: Training Samples from SVHN

1.2 Findings from Existing Papers

Since GANs were first raised by Goodfellow in 2014 [1], they have been one of the most popular topics in deep learning. GANs are famous for their generative ability to produce fake pictures which are close to the real ones. However, GANs also suffer from difficulty in training and parameter tuning, as well as other problems like mode collapse. There have been different modifications made on naive GANs in the architecture of both the generator and discriminator [2]. For example, DCGAN replaces fully connected layers with convolutional layers and introduces batch normalization, which sharply increases the quality of the pictures generated. WGAN [3] improves performance by changing the loss function and objective function to increase the stability of the training process.

Inspired by WGAN, WGAN with gradient penalty (WGAN-GP) [4] and GANs with spectral normalization [5] try to meet the 1-Lipschitz continuity in the discriminator.

1.3 Challenges

GANs are known for their incredibly train-difficulty, which can be summarized specifically for our project as the following:

Model architecture architecture influences the performance a lot, the same architecture performs well in MNIST may not be suitable for SVHN.

Optimizer Adam [10], the state of the art optimizer that does a great job in many GANs problems, is not performing well in WGAN-CLIP. In order to achieve acceptable performance, we may need to turn to Rmsprop.

Hyperparameters It is the first time when we need to consider the parameters of Adam. During the training process of DCGAN, we discover that setting $\beta_1 = 0.5$ and the learning rate to 1e-4 is of great significance for our models to learn the data effectively.

2 Methods

2.1 Model Architecture

In our project, we explore various models and test its performance on the aforementioned two datasets. All the model architecture can be found in the Appendix part and detailed findings are presented in the experiments and results part.

2.1.1 DCGAN

Deep Convolutional Generative Adversarial Networks, or DCGANs, are models scaling GANs with CNN architecture which is commonly used in the supervised literature. It introduces stride convolutions as downsampling and uses batch normalization to stabilize learning, helping gradient flow in deeper models [2]. DCGANs do not modify the loss in the model, which means it has the same objective function as plain GANs. Our project starts with a conventional DCGAN model and tests it on MNIST data first.

2.1.2 FCC-GAN

Besides DCGAN, we also explore a structured GAN with fully connected and convolutional net, or FCC-GAN [9]. The key architecture in FCC-GAN are multiple deep fully connected layers both before convolution layers in the generator, to convert the low-dimensional noise vector to a high-dimensional representation of image features, and before classification in the discriminator, to map the high-dimensional features extracted by convolution layers to a lower-dimensional space. Also, it adds average pooling layers with unit-stride convolution in the discriminator, instead of the conventional choice of 2 by 2 stride convolution. In our experiments with MNIST and SVHN datasets, adding FCC features helps our DCGAN model learn both faster and also generate a higher quality of samples.

2.1.3 WGAN-CLIP

The reason why GANs are hard to train is that they approximate the LS divergence which is the same for all pairs of distributions without any overlap. In order to overcome this problem, (Martin et al, 2017) introduced Wasserstein GAN, which measures the distance between two distribution by Earth Mover's distance. The corresponding optimization problem becomes:

$$V(G, D) = \max_{D \in 1-Lipschitz} \{E_{x \sim P_{data}}[D(x)] - E_{x \sim P_G}[D(x)]\}.$$

where V is now called the Wasserstein distance. However, the function D in the new form of Wasserstein metric is epected to satisfy $||f||_L \leq K$, meaning it should be K-Lipschitz continuous. Here, we set $K = 1$. In order to approximate this constraint, there are two ways, clipping discriminators weights to (-0.01,0.01) and adding gradient penalty to the discriminator. We do experiments on both of them named as WGAN-CLIP and WGAN-GP.

2.1.4 WGAN-GP

WGAN-CLIP optimization process is difficult because of interactions between the weight constraints and the cost function, which leads to either vanishing or exploding gradients without careful tuning of the clipping threshold c.

In order to overcome this problem, (Ishaan et al, 2017) proposed an alternative way to enforce the Lipschitz constraint. Since a differential function is 1-Lipschtiz if and only if it has gradients with norm at most 1 everywhere, they consider directly constraining the gradient norm of the discriminator's output with respect to its input. As a result, they finally enforce a soft version of the constraint with a penalty on the gradient norm for random samples $\hat{x} \sim P_{\hat{x}}$. Now the new objective(loss) of discriminator is:

$$L = E_{\tilde{x} \sim P_G}[D(\tilde{x})] - E_{x \sim P_{data}}[D(x)] + \lambda E_{\hat{x} \sim P_{\hat{x}}}[(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2].$$

where λ is the gradient penalty coefficient($\lambda = 10$ by default) and $\hat{x} \sim P_{\hat{x}}$ is sampled uniformly along straight lines between pairs of points x and $\tilde{x}$ sampled from the data distribution P_{data} and the generator distribution P_G.

2.1.5 SNGAN

SNGAN uses another method, which is spectral normalization, to make the discriminator 1-Lipschitz continuous and constrain the degree of variation. As long as each layer is made to satisfy the condition, the whole discriminator will be 1-Lipschitz continuous. We constrain the maximum singular value of the weight matrix in each layer as 1, which can be achieved by dividing the weights by its maximum singular value. To find the singular value, power iteration is used to find its approximation, since singular value decomposition for weights of high dimensions requires high time and computation cost. Besides, activation functions such as ReLU and Leaky ReLU is 1-Lipschitz, only convolution and dense layers needed to be normalized. Batch normalization should be dropped as it does not satisfy Lipschitz continuity.

2.2 Training Process

The training process of GANs is just like playing a minimax game. In this game, the generator is trying to maximize its probability of having its outputs recognized as real, while the discriminator is trying to minimize this same value. The objective function for the game is:

$$\min_G \max_D V(D,G) = \{E_{x\sim P_{data}}[logD(x)] + E_{z\sim P_z(z)}[log(1-D(G(z)))]\}.$$

Since both the generator and discriminator are being modeled with neural networks, we use a gradient-based optimization algorithm to train our GANs. The general idea is to sample a noise set and a real dataset, each with same mini-batch size first. Next is to train the discriminator and generator on this data and update them and then to repeat from the previous step. The algorithm is described as follows:

Algorithm 1 GAN. G means generator and D means discriminator.

```
For the number of epochs do:
    1. Sample minibatch of m noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from poise prior $z$.
    2. Sample minibatch of m examples $\{x^{(1)}, ..., x^{(m)}\}$ from training data.
    3. Update the discriminator by descending its stochastic gradient:
           $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [logD(x^{(i)}) + log(1-D(G(z^{(i)})))]$.
    4. Update the generator by descending its stochastic gradient:
           $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} log(1-D(G_z^{(i)})))$.
End for
```

For DCGANs, we also tweak the loss function in order to achieve better results. Specifically, we add smoothing labels when calculating binary cross-entropy and add noises to labels by flipping the real output and fake output at a small probability. We end up with using 0.1 as smoothing label and setting a 5% chance to flip the labels.

2.3 Model Evaluation: Inception Score

One problem with deep generative models is that there is no objective way to evaluate the quality of the generated images. Many attempts have been made to establish an objective measure of generated image quality. One of them is the Inception Score or IS [6]. It involves using a pre-trained deep learning neural network model for image classification to classify the generated images. Specifically, the Inception v3 model [7].

$$IS = exp(\sum_y \sum_x P(y|x) logP(y|x) - \sum_y P(y) logP(y)).$$

The Inception Score seeks to capture two properties of a collection of generated images: image quality and image diversity. In the formula above, we have the first part, the negative entropy of P(y|x), measures how good images look like specific objects, and the second part, the entropy of $P(y)$, measures range of generated images. The score has a minimum value of 1 and a maximum value of the number of classes supported by the classification model (1000 for Inception v3). Higher value in IS stands for a better result.

However, there are problems with the Inception Score that make it not a perfect metric for the evaluation and comparison of generative models [8]. For example, it is sensitive to small changes in network weights that do not affect the final classification accuracy of the network. In our experiments, the Inception Score are not always consistent with the subjective evaluation for the quality of generated images. Thus, to evaluate our models in a more empirical way, we also monitor and record loss and output images during the training process. All these methods together lead to more consistent and reasonable conclusions of our experiments.

3 Experiments & Results

Besides conventional CNN architecture, we add different features to DCGAN model including fully connected layers, average pooling layers, and spectral normalization. We evaluate the quality of generated images as well as their effectiveness on two benchmark image datasets: MNIST and SVHN. We also train our models on the Wasserstein distance objective function from WGAN [12]. Codes for all implementation can be found at `https://github.com/Trccc/5242Project`.

3.1 DCGAN

We start our experiment from a simple DCGAN, adding Batch Normalization, ReLU to the generator and Batch Normalization, LeakyReLU (0.2), Dropout (0.3) to the discriminator. We train the model on MNIST first. The loss function is adjusted with smoothing label and noise as mentioned before. We use Adam optimizer with β_1 set to 0.5 and the learning rate set to 1e-4. The results are shown in figure 3.1.

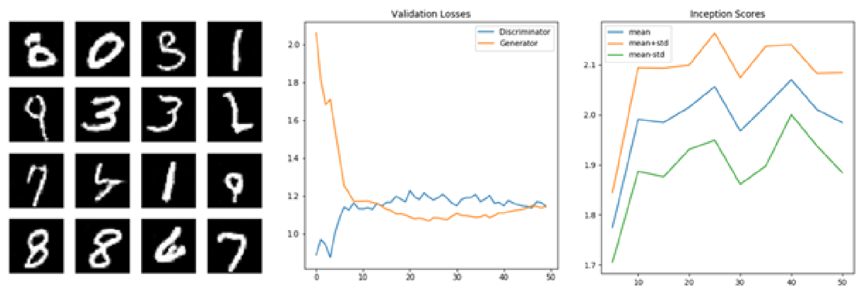

Figure 3.1: Plain DCGAN on MINIST, 50 epochs

Next, we reduce the number of layers in CONV but add more fully connected layers to both generator and discriminator to make it an FCC-GAN model. We keep the hyperparameters the same and plot the results in figure 3.2.

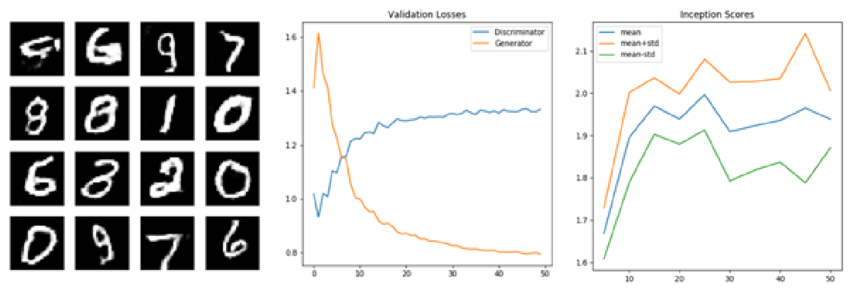

Figure 3.2: FCC-GAN on MNIST, 50 epochs

The results from FCC-GAN do not improve compared to DCGAN, though the training time in each epoch is significantly reduced from 10 seconds to 3 seconds. From the loss plot, we see that the generator performs better than DCGAN while the discriminator is not. Thus, we modify our model by only adding fully connected layers to the generator. This time the quality of generated images and Inception Score seems slightly improved. See figure 3.3 for results.

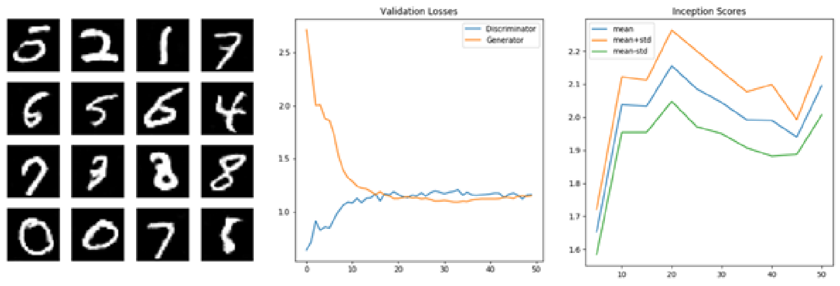

Figure 3.3: FCC-generator GAN on MNIST, 50 epochs

For SVHN dataset, we change the input size of the generator model and add average pooling layers before LeakyReLU to the discriminator. All hyperparameters and adjusted loss functions are the same as for MNIST, except for β_2 in Adam is set to 0.99999, which as a result makes the loss for generator converges instead of continuously moving upward after 10 epochs.

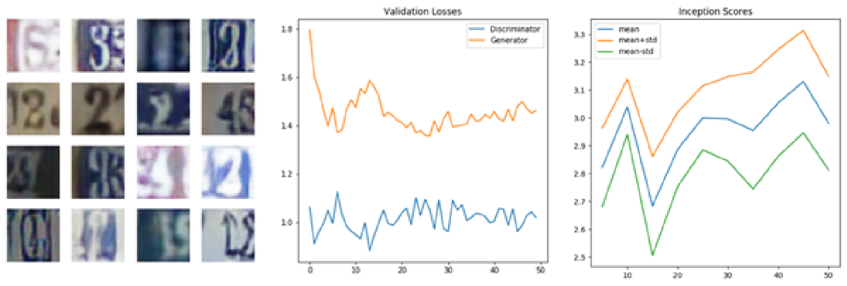

Figure 3.4: DCGAN with pooling layers on SVHN, 50 epochs

We then modify the FCC-GAN model, as suggested in the FCC-GAN paper, by removing the Dropout layers in the discriminator and replace the padding method in the generator by adding a Cropping2D layer after the CONV2DTranspose. The result improved significantly from plain DCGAN model, both in quality of generated images, loss and Inception Scores.

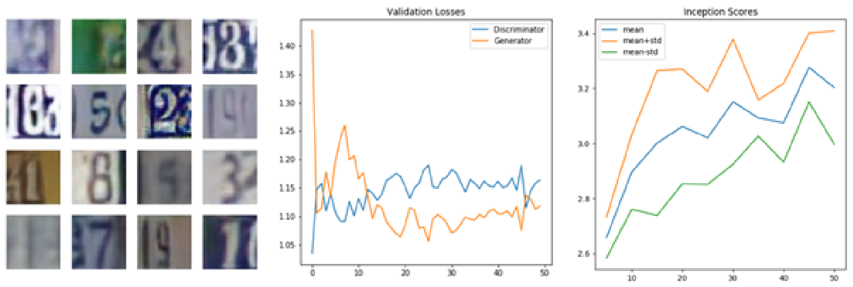

Figure 3.5: FCC-GAN on SVHN, 50 epochs

3.2 WGAN-CLIP

WGAN-CLIP is a way to constrain the discriminator to be in 1-Lipschitz, which, however, is not a good approach. It is hard for the generator to generate figures if the hyperparameters are not set properly. We try imbalanced learning rates(1e-4 for generator, 4e-4 for discriminator), different optimizers(Adam, Rmsprop), various none-linearity functions(ReLU, Leaky ReLU), and different architecture(dropout, kernel size, layers). We end up with a setting similar to the original paper and use the same architecture on both datasets. To be more specific, we use RMSprop(5e-5) for both generator and discriminator and balanced training(generator and discriminator are trained one by one).

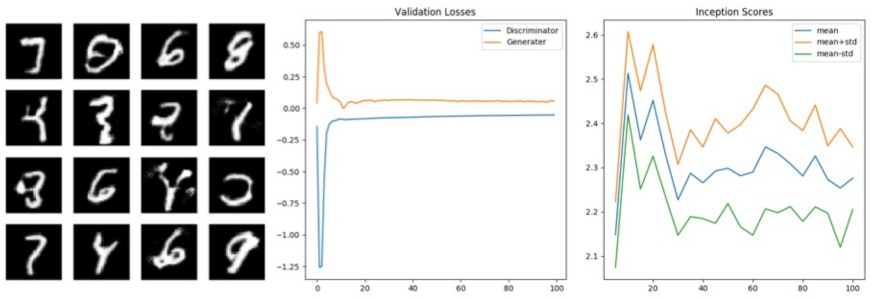

Figure 3.6: WGAN-CLIP on MNIST, 100 epochs

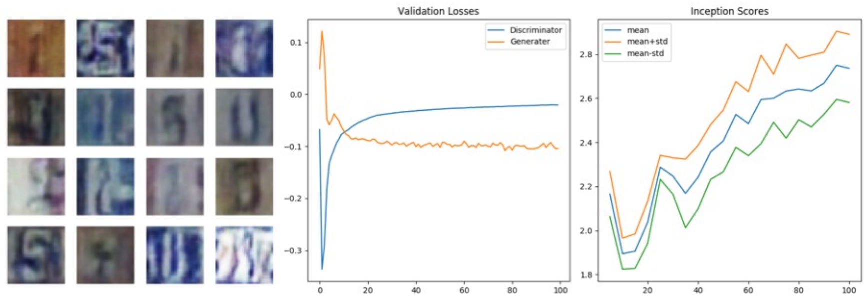

Figure 3.7: WGAN-CLIP on SVHN, 100 epochs

3.3 WGAN-GP

WGAN-GP adds a gradient penalty to the previous discriminator loss of WGAN-CLIP, so it seems to be much easier to take control of the discriminator loss in the training steps. Actually, it does. Firstly, we start our experiment from adjusted FCC-GAN architecture for both generator and discriminator. We keep the hyperparameters as same as the one WGAN-GP paper sets, i.e., Adam Optimizer with learning rate=0.0001, $\beta_1 = 0$, $\beta_2 = 0.9$ for both G and D. The results are shown in figure 3.8.

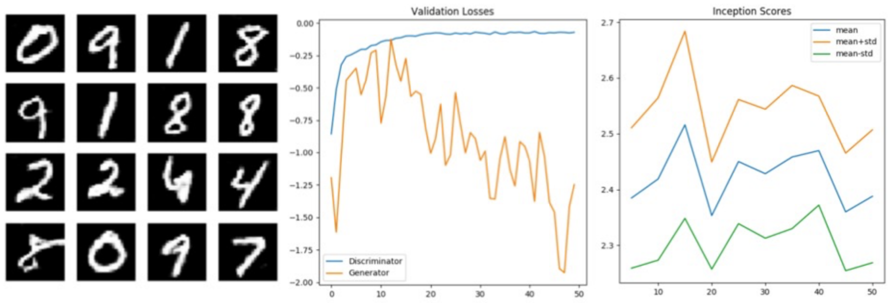

Figure 3.8: WGAN-GP on MNIST, 50 epochs

From the results above, although the output image and Inception Score are both good compared to the previous models, the generator loss explodes gradually after 10 epochs. We believe that the reason behind is that WGAN-GP leaves a penalty term in the discriminator loss, leading to discriminator overpowers generator in the training process. So for MNIST, we change the exponential decay rate for the 2nd moment estimates(i.e. β_2) of Adam optimizer for generator from 0.9 to 0.99999. We think that by doing so, the generator's performance could be reinforced during training. The results are shown in figure 3.9.

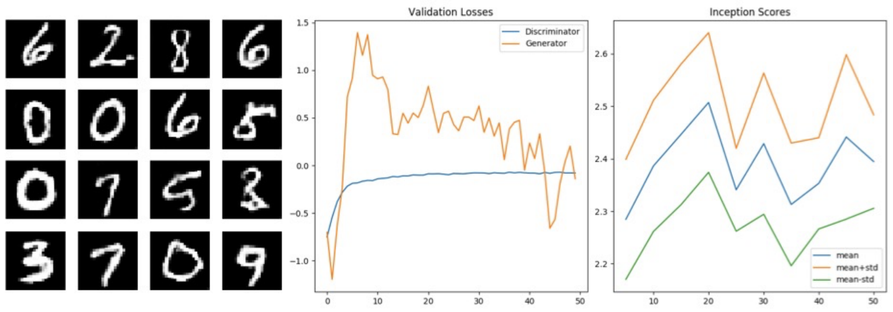

Figure 3.9: Adjusted WGAN-GP on MNIST, 50 epochs

The validation Losses seems to be convergent, and the quality of output images and Inception Scores are not bad as well.

Next, we move on to SVHN dataset. Here we use the same Neural Network structure of the model and optimizer for discriminator as default, but change the hyperparameter β_2 of Adam optimizer for generator from 0.9 to 0.99. We try different values, and believe this little change can make our results become much better. The results are shown in figure 3.10.

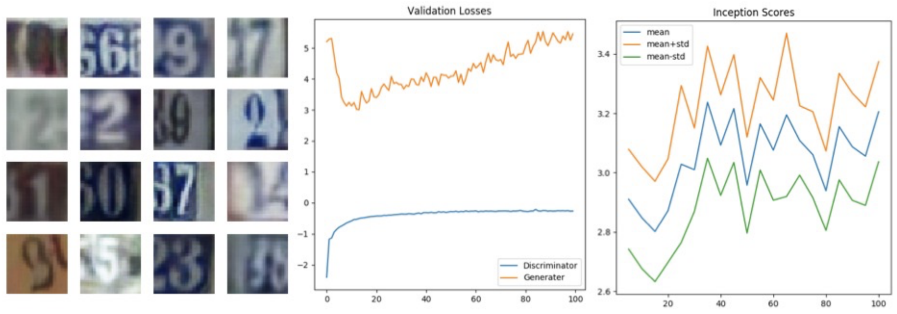

Figure 3.10: WGAN-GP on SVHN, 100 epochs

Similarly, we use an adjusted WGAN-GP on SVHN to make validation losses converge during the training process. To be more detailed, we change β_2 of Adam optimizer for generator to 0.99999 and add decay equals to 1e-5. The results are shown in figure 3.11.

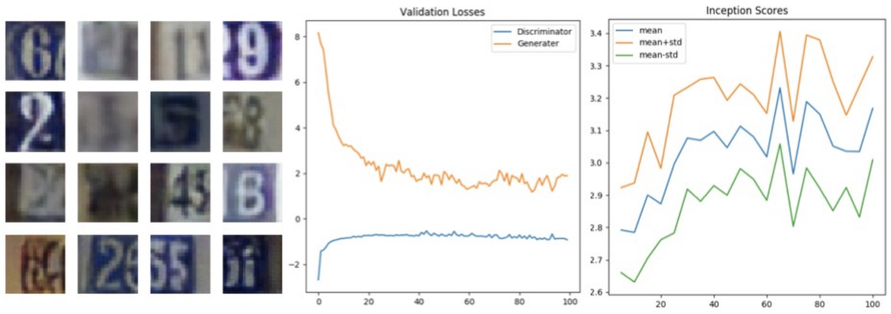

Figure 3.11: Adjusted WGAN-GP on SVHN, 100 epochs

The results are also convincing.

3.4 SNGAN

Similar to WGAN-CLIP, we looped in each layer of the discriminator and normalized the weights if it was a convolutional layer. As a result, the time cost for SNGAN increased a bit compared with the vanilla GAN. Convolutional layers without batch normalization are added in the discriminator. The loss function used is binary cross-entropy with label smoothing, and the optimizer is Adam with learning rate=1e-4, $\beta_1 = 0.5$ and $\beta_2 = 0.9999$.

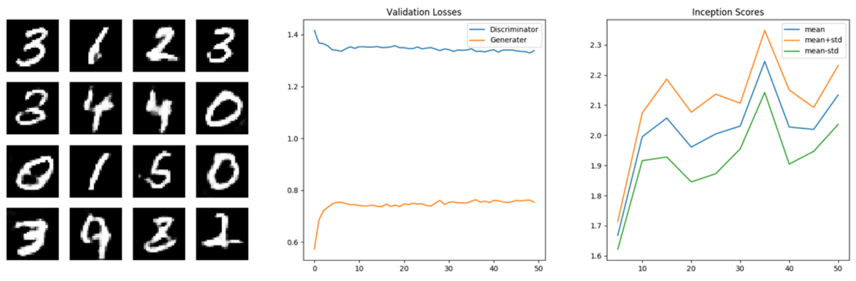

Figure 3.12: SNGAN on MNIST, 50 epochs

For SVHN data, models are of different structures but loss functions and hyperparameters are the same. That is to say, we use binary cross-entropy with label smoothing, and Adam optimizer learning rate=1e-4, $\beta_1 = 0.5$ and $\beta_2 = 0.9999$.

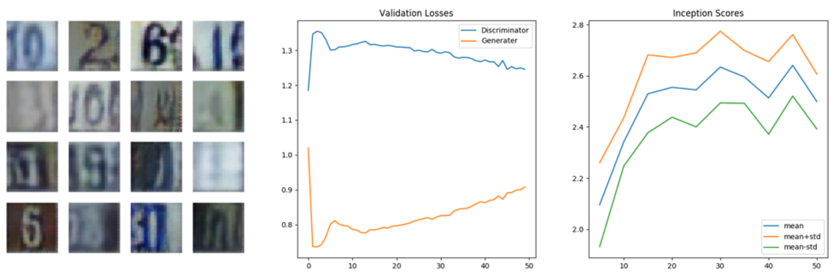

Figure 3.13: SNGAN on SVHN, 50 epochs

From the loss plots, we can see that although the pictures generated seem good, both of the generator and discriminator loss do not have obvious improvement after several epochs of training.

When training on SVHN, the generator loss does not converge and keeps increasing, which is a potential problem. However, if we check the pictures generated, there is no obvious decay in the picture quality.

4 Conclusion

In this project, we define our research problem to be image generation using GANs, and successfully apply what we have learned in class to work out this problem. We try various kinds of GANS on two number digits datasets and evaluate our models by visualization of different metrics. During extensive and detailed experiments, a number of interesting findings are proposed, aiming to summarize our results in the analysis.

First, almost all of the GANs in this project are able to produce satisfactory results (except WGAN-CLIP) on both datasets with fine-tuned hyperparameters. Meanwhile, the training process tells us that, GANs, though a powerful idea, are not easy to train or to evaluate objectively. Slightly different choices in hyperparameters and architecture will cast a huge impact on the generated images, which requires us a quick iteration and effective training strategy. Also, Inception Scores have only limited use in model assessment since it may not always consistent with the quality of the generated images. Some tricks we find from recent research papers do help to ease the painful training process. They are label smoothing, adding noise to label, and hyperparameters in Adam including learning rate, β_1 and β_2. Finally, there is no global choice for best models. Optimal results depend on both the nature of data and model architecture.

References

[1] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *NIPS*, pp. 2672–2680, 2014.

[2] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *ICLR*, 2016.

[3] Martin Arjovsky and L'eon Bottou. Towards principled methods for training generative adversarial networks. In *ICLR*, 2017.

[4] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein GANs. arXiv preprint arXiv:1704.00028, 2017.

[5] Miyato, Takeru, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. arXiv preprint arXiv:1802.05957, 2018.

[6] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training GANs. In *NIPS*, pp. 2234–2242, 2016.

[7] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. arXiv preprint arXiv:1512.00567, 2015.

[8] Shane Barratt, Rishi Sharma. A Note on the Inception Score. arXiv:1801.01973, 2018.

[9] Sukarna Barua, Sarah Monazam Erfani, James Bailey. FCC-GAN: A fully connected and convolutional net architecture for GANs. arXiv:1905.02417, 2019.

[10] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. 2011. Reading digits in natural images with unsupervised feature learning. In *NIPS* (workshop track), Vol. 2011.5.

[11] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[12] Martin Arjovsky, Soumith Chintala, and L'eon Bottou. Wasserstein GAN. arXiv preprint arXiv:1701.07875 (2017).

A Model Architecture

A.1 DCGAN

Table A.1: Plain DCGAN Architecture for MNIST

Generator	Discriminator
Input: Z (100)	Input: (28, 28, 1)
FC (12544), BN, ReLU	CONV (128, 3, 2), BN, LReLU, Dropout
Reshape (7, 7, 256)	CONV (256, 3, 2), BN, LReLU, Dropout
CONVT (128, 3, 1), BN, ReLU	CONV (512, 3, 2), BN, LReLU, Dropout
CONVT (64, 3, 2), BN, ReLU	Flatten (3072)
CONVT (1, 3, 2), Tanh	FC (1), Sigmoid
Output: (28, 28, 1)	Output: (1)

Table A.2: FCC-GAN Architecture for MNIST

Generator	Discriminator
Input: Z (100)	Input: (28, 28, 1)
FC (64), ReLU	CONV (32, 3, 2), BN, LReLU, Dropout
FC (512), ReLU	CONV (64, 3, 2), BN, LReLU, Dropout
FC (1152), BN	CONV (128, 3, 2), BN, LReLU, Dropout
Reshape (3, 3, 128)	Flatten (1152)
CONVT (64, 3, 2), BN, ReLU	FC (512), LReLU
CONVT (32, 3, 2), BN, ReLU	FC (64), LReLU
CONVT (1, 3, 2), Tanh	FC (16), LReLU
Output: (28, 28, 1)	FC (1), Sigmoid
	Output: (1)

Table A.3: FCC Generator GAN Architecture for MNIST

Generator	Discriminator
Input: Z (100)	Input: (28, 28, 1)
FC (64), ReLU	CONV (128, 3, 2), BN, LReLU, Dropout
FC (512), ReLU	CONV (256, 3, 2), BN, LReLU, Dropout
FC (1152), BN	CONV (512, 3, 2), BN, LReLU, Dropout
Reshape (3, 3, 128)	Flatten (8192)
CONVT (64, 3, 2), BN, ReLU	FC (1), Sigmoid
CONVT (32, 3, 2), BN, ReLU	Output: (1)
CONVT (1, 3, 2), Tanh	
Output: (28, 28, 1)	

Table A.4: DCGAN Architecture for SVHN

Generator	Discriminator
Input: Z (100)	Input: (32, 32, 3)
FC (4096), BN, ReLU	CONV (128, 4, 1), BN, AvgPool, LReLU, Dropout
Reshape (4, 4, 256)	CONV (256, 4, 1), BN, AvgPool, LReLU, Dropout
CONVT (128, 4, 2), BN, ReLU	CONV (512, 4, 1), BN, AvgPool, LReLU, Dropout
CONVT (64, 4, 2), BN, ReLU	Flatten (3072)
CONVT (3, 4, 2), Tanh	FC (1), Sigmoid
Output: (32, 32, 3)	Output: (1)

Table A.5: FCC-GAN Architecture for SVHN

Generator	Discriminator
Input: Z (100)	Input: (32, 32, 3)
FC (64), ReLU	CONV (64, 4, 2), BN, LReLU
FC (512), ReLU	CONV (128, 4, 2), BN, LReLU
FC (4096), BN	CONV (256, 4, 2), BN, LReLU
Reshape (4, 4, 256)	Flatten (4096)
CONVT (128, 4, 2), BN, ReLU	FC (512), LReLU
CONVT (64, 4, 2), BN, ReLU	FC (64), LReLU
CONVT (3, 4, 3), Tanh	FC (16), LReLU
Output: (32, 32, 3)	Output: (1)

A.2 WGAN-CLIP

Table A.6: WGAN-CLIP Architecture for MNIST

Generator	Discriminator
Input: Z (100)	Input: (32, 32, 1)
FC (64), ReLU	CONV (256, 4, 2), BN, LReLU
FC (512), ReLU	CONV (512, 4, 2), BN, LReLU
FC (1152), BN	CONV (1024, 4, 2), BN, LReLU
Reshape (3, 3, 128)	Flatten (4096)
CONVT (64, 3, 2), BN, ReLU	Output: (1)
CONVT (32, 3, 2), BN, ReLU	
CONVT (1, 4, 2), Tanh	
Output: (32, 32, 1)	

Table A.7: WGAN-CLIP Architecture for SVHN

Generator	Discriminator
Input: Z (100)	Input: (32, 32, 3)
FC (64), ReLU	CONV (256, 4, 2), BN, LReLU
FC (512), ReLU	CONV (512, 4, 2), BN, LReLU
FC (1152), BN	CONV (1024, 4, 2), BN, LReLU
Reshape (3, 3, 128)	Flatten (4096)
CONVT (64, 3, 2), BN, ReLU	Output: (1)
CONVT (32, 3, 2), BN, ReLU	
CONVT (1, 4, 2), Tanh	
Output: (32, 32, 3)	

A.3 WGAN-GP

Table A.8: WGAN-GP Architecture for MNIST

Generator	Discriminator
Input: Z (100)	Input: (28, 28, 1)
FC (64), ReLU	CONV (32, 3, 2), BN, LReLU, Dropout
FC (512), ReLU	CONV (64, 3, 2), BN, LReLU, Dropout
FC (1152), BN	CONV (128, 3, 2), BN, LReLU, Dropout
Reshape (3, 3, 128)	Flatten (1152)
CONVT (64, 3, 2), BN, ReLU	FC (512), LReLU
CONVT (32, 3, 2), BN, ReLU	FC (64), LReLU
CONVT (1, 3, 2), Tanh	FC (16), LReLU
Output: (28, 28, 1)	FC (1)
	Output: (1)

Table A.9: WGAN-GP Architecture for SVHN

Generator	Discriminator
Input: Z (100)	Input: (32, 32, 3)
FC (64), ReLU	CONV (64, 4, 1), BN, AvgPool, LReLU
FC (512), ReLU	CONV (128, 4, 1), BN, AvgPool, LReLU
FC (4096), BN	CONV (256, 4, 1), BN, AvgPool, LReLU
Reshape (4, 4, 256)	Flatten (4096)
CONVT (128, 4, 2), CROP(1), BN, ReLU	FC (512), LReLU
CONVT (64, 4, 2), CROP(1), BN, ReLU	FC (64), LReLU
CONVT (3, 4, 2), Tanh	FC (16), LReLU
Output: (32, 32, 3)	Output: (1)

A.4 SNGAN

Table A.10: SNGAN Architecture for MNIST

Generator	Discriminator
Input: Z (100)	Input: (28, 28, 1)
FC (12544), BN, LReLU	CONV (64, 5, 2), LReLU
Reshape (7, 7, 256)	CONV (128, 5, 2), LReLU
CONVT (128, 5, 1), BN, LReLU	Flatten (6272)
CONVT (64, 5, 2), BN, LReLU	FC (1)
CONVT (1, 5, 2)	Output: (1)
Output: (28, 28, 1)	

Table A.11: SNGAN Architecture for SVHN

Generator	Discriminator
Input: Z (100)	Input: (32, 32, 3)
FC (64)	CONV (64, 4, 2), LReLU
FC(512)	CONV (128, 4, 2), LReLU
FC(4096), BN	CONV(256, 4, 2), LReLU
Reshape (4, 4, 256)	Flatten (1024)
CONVT (128, 4, 2), BN, ReLU	FC(512), LReLU
CROP(1), BN, ReLU	FC(64), LReLU
CONVT (3, 4, 2)	FC(16), LReLU
CROP(1)	FC (1)
Output: (32, 32, 3)	Output: (1)

B Codes

B.1 WGAN-CLIP

```
## loss
def discriminator_loss(real_output, fake_output):

    real_loss = real_output
    fake_loss = fake_output
    total_loss = -tf.reduce_mean(real_loss) + tf.reduce_mean(fake_loss)
    return total_loss


def generator_loss(fake_output):
    return -tf.reduce_mean(fake_output)

## training steps

def train_step(images, showloss = False):
    noise = tf.random.normal([cfg.BATCH_SIZE, cfg.NOISE_DIM])
    g_loss = generator_loss
    d_loss = discriminator_loss

    with tf.GradientTape() as gen_tape,
         tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = g_loss(fake_output)
        disc_loss = d_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss,
                             generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
                                 discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
                                        generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_
                  discriminator, discriminator.trainable_variables))

    if cfg.WGAN_CLIP:
        for layer in discriminator.layers:
            weights = layer.get_weights()
            weights = [tf.clip_by_value(w, -cfg.CLIP, cfg.CLIP)
                       for w in weights]
            layer.set_weights(weights)

    return gen_loss, disc_loss
```

B.2 WGAN-GP

```
## loss
def gradient_penalty_loss(averaged_samples_output, averaged_samples,
                          gradient_penalty_weight):
    gradients = K.gradients(averaged_samples_output, averaged_samples)[0]
    gradients_sqr = K.square(gradients)
    gradients_sqr_sum = K.sum(gradients_sqr, axis=np.arange(1,
                              len(gradients_sqr.shape)))
    gradient_l2_norm = K.sqrt(gradients_sqr_sum)
    gradient_penalty = gradient_penalty_weight * K.square(1 -
                                         gradient_l2_norm)
    return K.mean(gradient_penalty)


def gp_discriminator_loss(real_output, fake_output,
    averaged_samples_output, averaged_samples, gradient_penalty_weight):
```

```
    loss = tf.reduce_mean(fake_output) - tf.reduce_mean(real_output) +
           gradient_penalty_loss(averaged_samples_output, averaged_samples,
                                 gradient_penalty_weight)
    return loss

def gp_generator_loss(fake_output):
    return -tf.reduce_mean(fake_output)

## train_step
def train_step(images):
    noise = tf.random.normal([cfg.BATCH_SIZE, cfg.NOISE_DIM])

    g_loss = gp_generator_loss
    d_loss = gp_discriminator_loss

    with tf.GradientTape() as gen_tape,
         tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        weights = K.random_uniform([cfg.BATCH_SIZE, 1, 1, 1])
        averaged_samples = generated_images + weights * (images -
                           generated_images)
        averaged_samples_output = discriminator(averaged_samples,
                                  training=True)

        gen_loss = g_loss(fake_output)
        disc_loss = d_loss(real_output, fake_output,
                    averaged_samples_output, averaged_samples,
                    cfg.GRADIENT_PENALTY_WEIGHT)

    gradients_of_generator = gen_tape.gradient(gen_loss,
                             generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
                                 discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
                                    generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_
             discriminator, discriminator.trainable_variables))

    return gen_loss, disc_loss
```

B.3 SNGAN

```
## train_step
def train_step(images, showloss = False):
```

```
noise = tf.random.normal([cfg.BATCH_SIZE, cfg.NOISE_DIM])

g_loss = generator_loss
d_loss = discriminator_loss

with tf.GradientTape() as gen_tape,
     tf.GradientTape() as disc_tape:
generated_images = generator(noise, training=True)

real_output = discriminator(images, training=True)
fake_output = discriminator(generated_images, training=True)

gen_loss = g_loss(fake_output)
disc_loss = d_loss(real_output, fake_output)

gradients_of_generator = gen_tape.gradient(gen_loss,
                         generator.trainable_variables)
gradients_of_discriminator = disc_tape.gradient(disc_loss,
                             discriminator.trainable_variables)

generator_optimizer.apply_gradients(zip(gradients_of_generator,
                       generator.trainable_variables))
discriminator_optimizer.apply_gradients(zip(gradients_of_
           discriminator, discriminator.trainable_variables))

if cfg.GAN_SN:
for layer in discriminator.layers:
        if black'blackconv2blackdblack' in layer.name:
        weights = layer.get_weights()
        weights = spectral_norm(weights)
        layer.set_weights(weights)
        if black'blackdenseblack' in layer.name:
        weights = layer.get_weights()
        if len(weights[0]) == 1:
                weights = [spectral_norm(weights).numpy()]
                layer.set_weights(weights)

return gen_loss, disc_loss
```