GAN Implementation and Improvement

Jia Li
jl5520@columbia.edu

Lining Zhang
lz2697@columbia.edu

Shuyuan Wang
sw3449@columbia.edu

Yuwei Tong
yt2713@columbia.edu

Abstract

We explore Generative Adversarial Network (GAN) in this project. We focus on two applications of GAN: Deep Convolutional GAN (DCGAN) and Wasserstein GAN (WGAN). The GANs are implemented on MNIST and SVHN dataset. We notice that both DCGAN and WGAN are capable of generating good-quality images, but WGAN mitigates the problem of image collapsing and provides loss and indicator to evaluate the training process.

1 Introduction

In this project, our task is to build Generative Adversarial Network (GAN) models on two datasets and find ways to improve them. We will introduce the papers we refer to, the basic idea behind GAN, explain the methods we use to construct our models as well as show the performance of our models on two datasets.

Initially, a DCGAN model is constructed as our own model, which adds a deep convolutional network to the generator and discriminator. Starting from TensorFlow tutorial and guided by related research papers, we use some tricks to fine-tune our network's hyper-parameters and architecture. Beyond that, we also analyze the problem lying in our DCGAN, and further improve it by using Wasserstein GAN (WGAN). We explore the differences between WGAN and DCGAN, and how WGAN can theoretically improve the performance.

2 Related Work

We understand the core ideas of GAN from the paper of Goodfellow et al. (2014). We start our project by an example of GAN on MNIST[1].

Several papers talking about improving stability of training and quality of GAN generated samples are rather helpful. We develop our DCGAN model based on methods proposed by Radford et al. (2016); Salimans et al. (2016); Isola et al. (2018), as discussed in the following sections.

As for the exploration of improved version of GAN, WGAN, we gain insights and guidance from paper of Arjovsky and Bottou (2017); Arjovsky et al. (2017) and blog of Weng (2017).

[1] Available at `https://www.tensorflow.org/tutorials/generative/dcgan`

Preprint. Under review.

3 Generative Adversarial Network

3.1 From a Simple Example

Before introducing theory behind GAN with bundles of confusing notations, let's start with a simple case.

The simplest way to start making counterfeit money is to learn from the real one. At the beginning, the counterfeit fails to be deposited since it would be recognized as fake. After studying the techniques of making realistic money, the counterfeit finally successfully tricks the ATM machine to be deposited, although the ATM machine also improves on distinguishing between real and fake money.

3.2 What is GAN

GAN is exactly the same as the procedure above. Generally it could be divided into two parts: Generator and Discriminator. We grab random data from a data distribution, like Gaussian distribution, put it into Generator to produce a series of data. Then we put the generated data into Discriminator to see if the generated distribution is "true" enough to cheat the Discriminator. We repeat this procedure, during which both Generator and Discriminator improve adversarially.

3.3 Mathematical Explanation

From the paper of Goodfellow et al. (2014), the objective function of DCGAN is given by

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

Notation	Meaning
x	real data
$p_{data}(x)$	distribution of real data
z	noise sample
$p_z(z)$	distribution of the noise
G	Generator
D	Discriminator
$V(G,D)$	loss function

$D(\cdot)$ represents the probability that input comes from real data rather than generated distribution. Since we want $G(z)$ to get close to real data distribution, equivalently we want $D(G(z)) \to 1$ and $log[1-D(G(z))] \to -\infty$, thus minimizing $V(D,G)$.

On the contrary, we want D to distinguish between real and generated data, equivalently we want $D(x) \to 1$, and $D(G(z)) \to 0$, thus maximizing $V(G,D)$.

We realize the above idea using Algorithm 1 of Goodfellow et al. (2014). See algorithm details in appendix 7.

4 DCGAN Implementation

4.1 Datasets

Two datasets are involved in our project, namely MNIST (Handwritten Digits) and SVHN (Street View House Numbers). We load MNIST dataset by applying `tf.keras.datasets.mnist()` and load SVHN from its original website [2]. We only use training set of these two datasets. MNIST has 60,000 training images, with each image of size 28 x 28 x 1, while SVHN has 73,257 training images, with each image of size 32 x 32 x 3.

4.2 Starting Point

After understanding the core ideas of GAN from the paper (Goodfellow et al., 2014), we start to solve the problem by looking at TensorFlow tutorial[3] on DCGAN implemented on MNIST dataset. In the tutorial, the generator starts with a `Dense` layer that takes a random noise as input, then uses `Conv2DTranspose` to upsample three times to reach the desired image size. All layers use batch

[2] Available at `http://ufldl.stanford.edu/housenumbers`

[3] Available at `https://www.tensorflow.org/tutorials/generative/dcgan`

normalization and use `LeakyReLU` as activation function, except the output layer uses `Tanh`. The discriminator is a three-layer CNN image classifier, with `LeakyReLU` as activation function and one `Dropout` layer. We believe the tutorial provides a solid base for GAN, however leaving a space for us to improve.

Observing the images generated by the existing DCGAN (see Figure 2), we notice that it takes 50 epochs to produce realistic handwritten digit images which is rather slow. Although some of the images produced at epoch 200 are indistinguishable, many are not in good shapes. The plot of the training process (see Figure 1) indicates the generator increasingly becomes good at producing images so that the discriminator cannot correctly classify, leading to higher error. As the image quality is not satisfactory, we start to dive deeper into the world of GAN, exploring relevant papers to try different measures to improve the existing DCGAN.

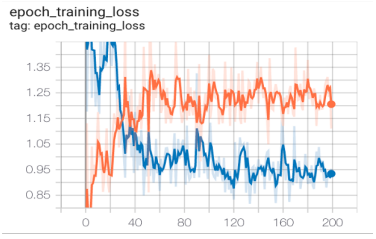
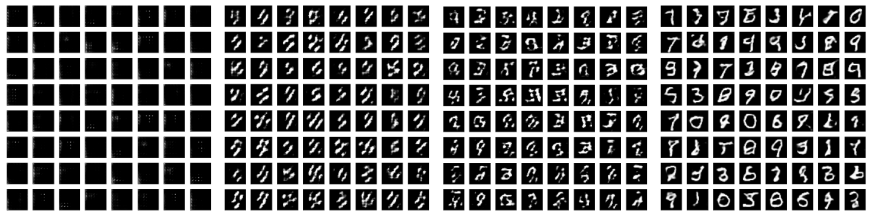
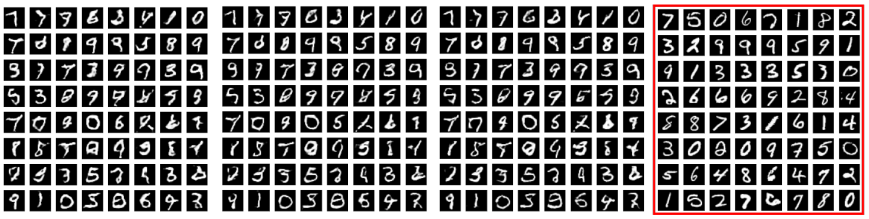


Figure 1: Training process of DCGAN at starting point.



(a) Epoch 1 (b) Epoch 5 (c) Epoch 10 (d) Epoch 50



(e) Epoch 100 (f) Epoch 150 (g) Epoch 200 (h) Real image

Figure 2: Generated images of DCGAN at starting point and real image.

4.3 Network Architecture

4.3.1 MNIST

The following approaches are adopted to modify the existing DCGAN on MNIST dataset.

The first is Batch Normalization. Batchnorm stabilizes learning by normalizing the input to each unit to have zero mean and unit variance, which contributes to avoiding problems arising due to poor initialization and helps gradient flow in deeper models (Radford et al., 2016). In the existing DCGAN, no batch normalization is applied to the discriminator. This could be a potential reason why the image quality is not satisfactory. Guided by the work of Radford et al. (2016), we decide to add batchnorm to all layers, except the generator output layer and the discriminator input layer to avoid sample oscillation and model instability.

The second is label smoothing. In order not to let the discriminator loss reach 0 at an early stage, we apply soft labels instead of hard labels when training the discriminator. Advised by the paper of Salimans et al. (2016), we end up smoothing only the positive labels to 0.9 and keeping the negative labels 0.

The third is Dropout. In the existing DCGAN, the generator does not have `Dropout` layer. Inspired by the paper of Isola et al. (2018), we add a `Dropout` layer to the generator so as to provide certain level of noise.

The fourth is activation function. Following the work of Radford et al. (2016), we use `ReLU` activation in generator for all layers except for the output, which uses `Tanh`, and use `LeakyReLU` activation in the discriminator for all layers.

```python
# define a function to build generator
def build_generator():
  model = tf.keras.Sequential([
          layers.Dense(128, use_bias=False, input_shape=(100,)),
          layers.BatchNormalization(),
          layers.LeakyReLU(0),

          layers.Dropout(0.2),

          layers.Dense(7*7*256, use_bias=False),
          layers.BatchNormalization(),
          layers.LeakyReLU(0),
          layers.Reshape((7,7,256)),

          layers.Conv2DTranspose(128, (5,5), strides=1, padding='same', use_bias=False), # 7,7,128
          layers.BatchNormalization(),
          layers.LeakyReLU(0),

          layers.Conv2DTranspose(64, (5,5), strides=2, padding='same', use_bias=False), #14,14,64
          layers.BatchNormalization(),
          layers.LeakyReLU(0),

          layers.Conv2DTranspose(1, (5,5), strides=2, padding='same', use_bias=False), #28,28,1
          layers.Activation('tanh')
  ])

  return model
```

(a) Generator

```python
# define a function to build discriminator
def build_discriminator():
  model = tf.keras.Sequential([
            layers.Conv2D(128, (5,5), strides=2, padding='same',
                          input_shape=(28,28,1)), #14,14,64
            layers.LeakyReLU(0.2),

            layers.Conv2D(64, (5,5), strides=2, padding='same'), #7,7,128
            layers.BatchNormalization(),
            layers.LeakyReLU(0.2),
            layers.Dropout(0.2),

            layers.Flatten(),
            layers.Dense(128, activation='relu'),
            layers.Dropout(0.2),
            layers.Dense(1, activation='sigmoid')
  ])
  return model
```

(b) Discriminator

Figure 3: Network Structure of GAN on MNIST.

4.3.2 SVHN

Then we apply the advanced DCGAN model to SVHN dataset with adjustments for the different image size.

5 DCGAN Performance

5.1 MNIST

The training process (Figure 5) and the generated images (Figure 6) of the DCGAN with stated modifications are shown below. We believe the training is expected as is shown in the plot of training process from Tensorboard. The discriminator loss is in orange and generator loss in blue. Since we have a binary cross-entropy loss for discriminator `D`, and another binary cross-entropy loss for `GAN` which is concatenated by `G` and `D`. So, the generator loss in blue is the discriminator's loss when

```python
def build_generator(input_size, leaky_alpha=0.2):
    '''generates images in (32,32,3)
    with up-sampling technique'''

    model = tf.keras.Sequential([
        layers.Dense(4*4*512, input_shape=(input_size,)),
        layers.Reshape(target_shape=(4, 4, 512)),                                  # 4,4,512
        layers.BatchNormalization(),
        layers.LeakyReLU(alpha=leaky_alpha),
        layers.Dropout(0.2),
        layers.Conv2DTranspose(256, kernel_size=5, strides=2, padding='same', use_bias=False), # 8,8,256
        layers.BatchNormalization(),
        layers.LeakyReLU(alpha=leaky_alpha),
        layers.Conv2DTranspose(128, kernel_size=5, strides=2, padding='same', use_bias=False), # 16,16,128
        layers.BatchNormalization(),
        layers.LeakyReLU(alpha=leaky_alpha),
        layers.Conv2DTranspose(3, kernel_size=5, strides=2, padding='same', use_bias=False),   # 32,32,3
        layers.Activation('tanh')
    ])

    return model
```

(a) Generator

```python
def build_discriminator(leaky_alpha=0.2):
  model = tf.keras.Sequential([
            layers.Conv2D(64, (5,5), strides=2, padding='same', input_shape=(32,32,3)), # 16,16,64
            layers.LeakyReLU(leaky_alpha),
            layers.Dropout(0.2),
            layers.Conv2D(128, (5,5), strides=2, padding='same'), # 8,8,128
            layers.BatchNormalization(),
            layers.LeakyReLU(leaky_alpha),
            layers.Dropout(0.2),
            layers.Conv2D(256, (5,5), strides=2, padding='same'), # 4,4,256
            layers.BatchNormalization(),
            layers.LeakyReLU(leaky_alpha),
            layers.Flatten(),
            layers.Dense(1, activation='sigmoid')
  ])
  return model
```

(b) Discriminator

Figure 4: Network Structure of GAN on SVHN.

dealing with generated images. The loss is expected to be larger as training goes by, meaning that the GAN model generates images that discriminator fails to distinguish.

An initial shape of digits is generated at epoch 5 and the shape becomes more realistic at epoch 10, compared with the images produced by starting point DCGAN, the improved DCGAN displays a faster speed of convergence. Moreover, comparing the images produced at epoch 200, we believe the improved DCGAN generates images of higher quality, given that the digits are easier to identify and there are fewer failed samples.

Furthermore, when comparing the generated images with the real image (placed next to the generated ones), we think they are indistinguishable in most cases. We also examine the generated images with the Figure 2a7 of the paper (Goodfellow et al., 2014). Digits 8, 6, 9, 3, 1, 2, 0 which are shown in the original paper also have good shapes in our generated images at Epoch 200.

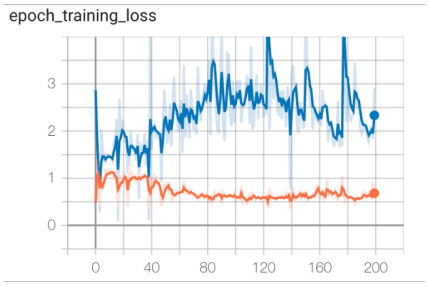
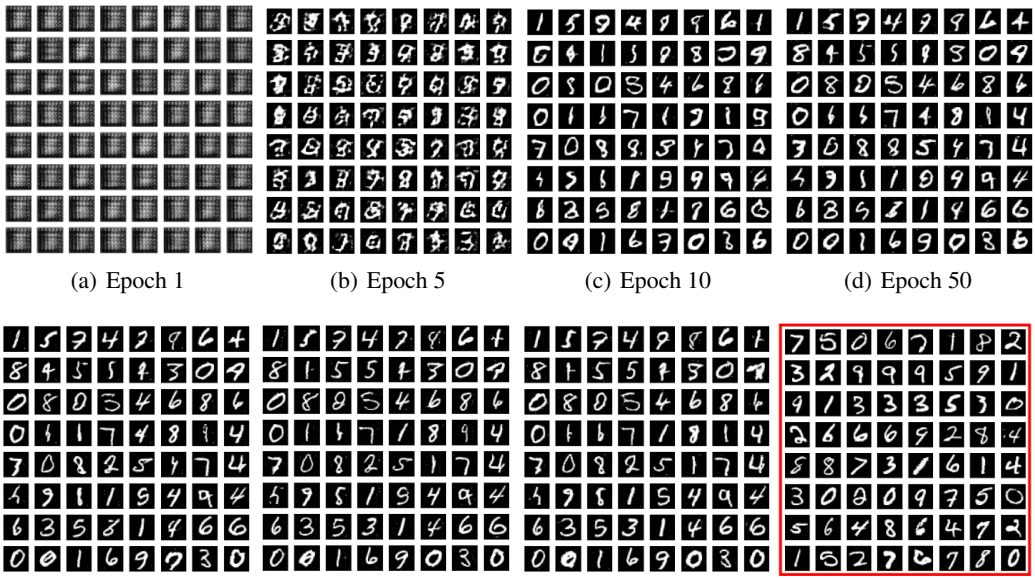


Figure 5: Training process of improved DCGAN on MNIST.



(a) Epoch 1 (b) Epoch 5 (c) Epoch 10 (d) Epoch 50

(e) Epoch 100 (f) Epoch 150 (g) Epoch 200 (h) Real image

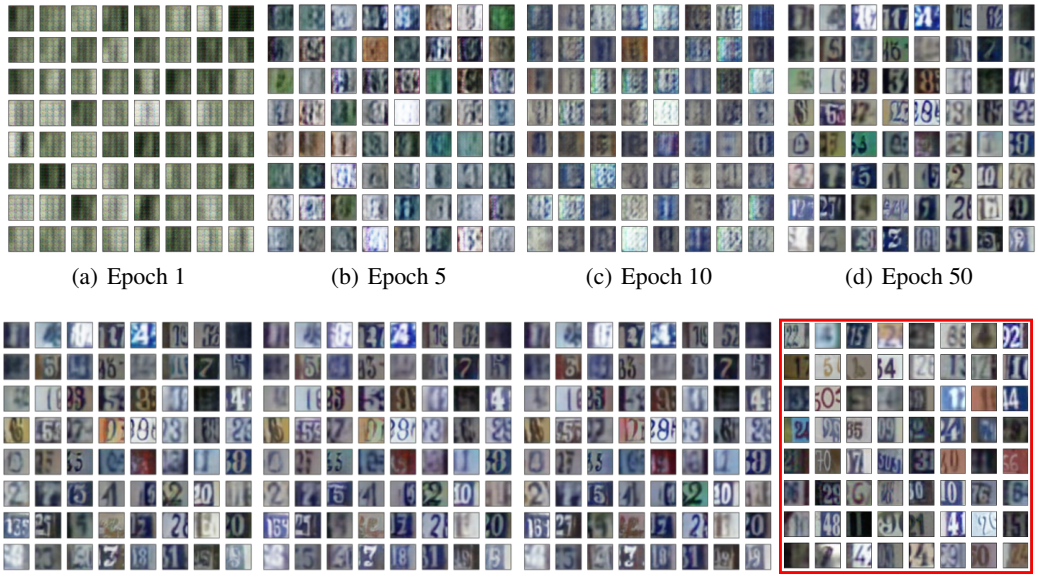
Figure 6: Generated images of improved DCGAN and real image.

Figure 7: Figure 2a from Goodfellow et al.

5.2 SVHN

Similar to the results on MNIST data, an initial color and shape of digits are generated at epoch 5 and become more and more clear. After around 50 epochs, the images generated are quite stable, with only small adjustments. Finally at epoch 200, the generated images look quite similar with the real images (see Figure 9).

Figure 8: Training process of DCGAN on SVHN.



(a) Epoch 1 (b) Epoch 5 (c) Epoch 10 (d) Epoch 50

(e) Epoch 100 (f) Epoch 150 (g) Epoch 200 (h) Real image

Figure 9: Generated images of DCGAN on SVHN and real image.

However, comparing with the results on MNIST data, the image quality of outputs on SVHN data seems to be a little lower. The generated images here are more blurred, and there is a larger proportion of numbers that are difficult to identify. But that's reasonable because the image quality of the training data from SVHN dataset is not as good as those from MNIST dataset. Also, SVHN images have more features. There are various fonts, text sizes, text colors, and background colors, so it is much more difficult to train those data.

Considering the above reasons, the results of our improved DCGAN models on SVHN data seem to be quite good, but there is still something that we'd like to enhance. Sometimes the results are not stable enough. For example, it's not sure that we will get satisfactory outputs every time, and the images at some epoch may be worse than the previous ones. In that case we'd like to try an advanced version of GAN, Wasserstein GAN (WGAN), to see whether the problem can be solved.

6 Wasserstein GAN

Wasserstein GAN(Arjovsky et al., 2017) and *Towards Principled Methods For Training Generative Adversarial Networks*(Arjovsky and Bottou, 2017) provide us with important insights and improvements. Some of the materials in these two papers are well beyond our knowledge, so we have to make some simplifications for our understanding. The key idea is that the current loss function has some drawbacks and we should find a more efficient way to measure difference between real image distribution and generated image distribution.

In our previous built DCGAN, we use the $-log(D)$ alternative trick. The loss function for generator is:

$$E_{x\sim P_z}[-log(D(g_\theta(z)))] \tag{1}$$

Given a perfect discriminator $D^* = \frac{P_g(x)}{P_{data}(x)+P_g(x)}$, loss function(1) becomes:

$$KL(P_{g_\theta}||P_r) - 2 * JSD(P_{g_\theta}||P_r) \tag{2}$$

Problems related to loss function(2) are:

- First, the signs for KL divergence and JS divergence are opposite. In order to minimize the loss, we need to minimize KL divergence, and maximize JS divergence at the same time. However, the ultimate goal is to let the distribution of generated images approach the

distribution of real images as much as possible. Maximizing JS divergence contradicts to our goal and would cause instability.

- Second, KL divergence is not symmetric. It will give very high penalty on outputing fake images outside the real image distribution, which is the case $P_r \to 0, P_{g_\theta} \to 1$. But it gives very low penalty on not outputing images in real image distribution, $P_r \to 1, P_{g_\theta} \to 0$. As a result, image collapsing will occur, which means the output images don't have enough diversity.

This is what happened in our previous built DCGAN. We have tried various architectures, and the loss for the discriminator always fluctuates at a low level, which implies it has been trained quite well. Consequently, we encounter repeated patterns for certain digits and unstable gradient for generator and output quality.

To overcome this problem, Arjovsky et al. (2017) provides us with a new loss function, called Wasserstein distance or Earth-Mover(EM) distance. EM distance has some nice properties such as smoothness, and can provide meaningful gradients. Besides, the paper also shows that EM distance is highly related to the quality of output image. Thus, it can be used as an indicator of the performance of the current model. Like DCGAN, discriminator tries to increase this distance between the real distribution and the generated distribution of images, while generator is going to reduce this distance.

The original expression for EM distance is complicated and impossible to solve:

$$W(P_r, P_\theta) = Sup_{||f||_L <= 1}(E_x \sim P_r[f(x)] - E_x \sim P_\theta[f(x)]) \tag{3}$$

However, we can approximate it using the following formula(4) by adding value clipping to the discriminator's parameters:

$$L = E_{x \sim P_r}[f_w(x)] - E_{x \sim P_\theta}[f_w(x)] \tag{4}$$

This approach is applied on original GAN model with the main structure unchanged despite the following four updates:

- We don't add sigmoid as activation function in discriminator.
- We don't apply log-loss on generator and discriminator.
- We clip the absolute values of the parameters of discriminator between 0 and constant c.
- We choose `RMSProp` as our optimizer instead of `Adam` to make training more stable.

The task for the discriminator now is not to make a binary classification, but to try fitting the EM distance like a regression problem. So we drop the sigmoid activation here.

We use Algorithm 1 in *Wasserstein GAN*(Arjovsky et al., 2017) as template and update the stated changes to our previous code. The following are the outputs for images generated by WGAN on MNIST11 and SVHN:

6.1 MNIST

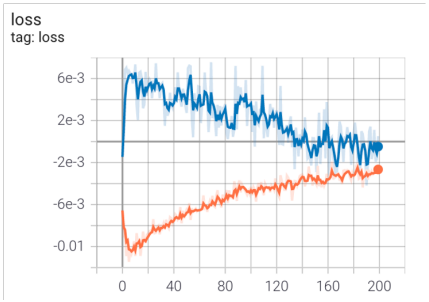
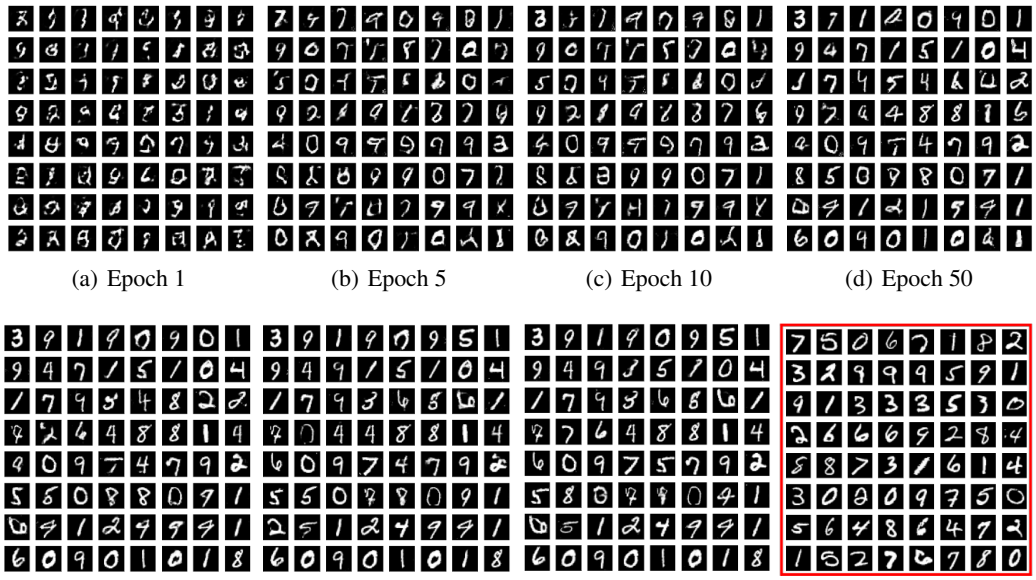


Figure 10: Training process of WGAN on MNIST.



(a) Epoch 1 (b) Epoch 5 (c) Epoch 10 (d) Epoch 50

(e) Epoch 100 (f) Epoch 150 (g) Epoch 200 (h) Real image

Figure 11: Generated images of WGAN on MNIST and real image.

6.2 SVHN

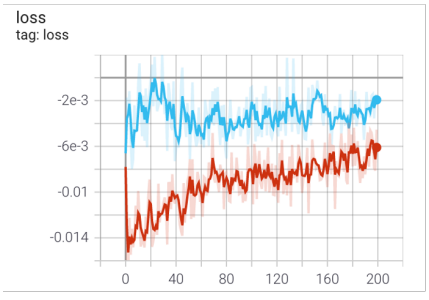
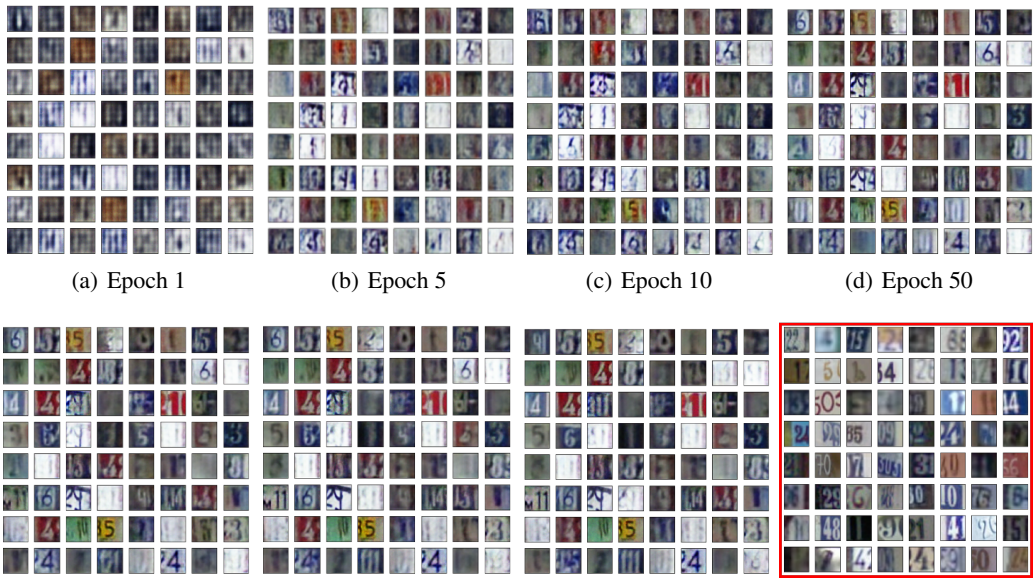


Figure 12: Training process of WGAN on SVHN.



(a) Epoch 1 (b) Epoch 5 (c) Epoch 10 (d) Epoch 50

(e) Epoch 100 (f) Epoch 150 (g) Epoch 200 (h) Real image

Figure 13: Generated images of WGAN on SVHN and real image.

On both datasets, the images are in good shape at first few epochs, but the quality improves slowly after 50 epochs. WGAN and GAN have similar output when they use deep convolution layers. The clarity of pictures using WGAN is even worse on SVHN. However, the images have slightly more diversified background colors and patterns. The negative of discriminator loss, which is the estimate of EM distance, has an overall decreasing trend over epochs. Since EM distance is correlated to the quality of output, we can use the distance or discriminator loss as an indicator to guide training process. This is one significant advantage of WGAN.

We conjecture WGAN needs more time to converge, because the estimate distance is still decreasing. We can try using more epochs and set `n_critic` to a larger number to improve the output.

7 Conclusion and Further Discussion

In Conclusion, both DCGAN and WGAN produce good-quality images on MNIST and SVHN dataset. Although we barely see quality improvement by WGAN, it helps mitigate the problem of model collapsing and more importantly, provide us with an indicator to evaluate the progress of training beyond exhaustively looking at the output images.

To estimate the EM distance, WGAN uses weight clipping on discriminator to satisfy the Lipschitz constraint. This method can lead to slow convergence and samples with low quality. These are exactly what we need to improve on. Fortunately, in the paper *Improved Training of Wasserstein GANs*(Gulrajani et al., 2017), gradient penalty is introduced. Penalty on the norm of discriminator's gradient is added to its loss function to improve the performance of WGAN. This model is called WGAN-gp, we will work on this method in our future study or research.

References

Arjovsky, M. and L. Bottou (2017). Towards principled methods for training generative adversarial networks.

Arjovsky, M., S. Chintala, and L. Bottou (2017). Wasserstein gan.

Goodfellow, I. J., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio (2014). Generative adversarial networks.

Gulrajani, I., F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville (2017). Improved training of wasserstein gans.

Isola, P., J.-Y. Zhu, T. Zhou, and A. A. Efros (2018). Image-to-image translation with conditional adversarial networks.

Radford, A., L. Metz, and S. Chintala (2016). Unsupervised representation learning with deep convolutional generative adversarial networks.

Salimans, T., I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen (2016). Improved techniques for training gans.

Weng, L. (2017). From gan to wgan.

Appendix

GAN Algorithm

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $z^{(1)}, \ldots, z^{(m)}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $x^{(1)}, \ldots, x^{(m)}$ from data generating distribution $p_{data}(x)$.
- Update the discriminator by ascending its stochastic gradient:

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

end for

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

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

end for