Exploration in Generative Adversarial Network

Feng Qiu, Shi Yin, Shengyang Zhang
Department of Statistics
Columbia University

Abstract

This report mainly contains the implementation of Generative Adversarial Networks and Wasserstein Generative Adversarial Networks. Also, MINST data set and SVHN data set are used for the method evaluation including loss, iteration efficiency and the final generated images. Eventually, WGAN have faster convergence rate and GAN have better performance.

1 Introduction

In this report, Generative Adversarial Networks will be introduced at the first place. The objective function of this method will shown and the target of two agents, Generator and Discriminator will be briefly explained. Followed by the detailed nets structure explanation, two image data sets will be talked about.

After showing the performance of GAN on both data sets, it is found that GAN has some mode collapse issue with SVHN data set, that is, the GAN may fail in some real-world complex situation and stuck in some spaces with low variety.

Therefore, an improved GAN method using Wasserstein distance as loss function will be introduced to see whether it have better performance on these two data sets.

2 Generative Adversarial Networks

The core idea of Generative Adversarial Networks is rather simple, to let two agents compete with each other during the training process at the same time. To be more specific, one of them known as Generator creates the fake picture as real as possible to fool the other agent known as Discriminator. While the Discriminator tries to distinguish between the counterfeit pictures and the real ones.

Given the formal objective function of GAN process,

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

In this function:

- **D(x)** is estimated probability of real data instance x is real of discriminator.
- **G(z)** is the output of generator given noise z.
- **D(G(z))** is estimated probability of fake instance is real of discriminator.

For the Generator, it is trained to increase the chance of Discriminator having high probability in counterfeit cases represented by term $D(G(x))$, thus to minimize $\mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z))]$.

As for the Discriminator, it is trained to ensure have accurate decisions over real data cases by maximizing $\mathbb{E}_{x \sim p_{data}}[logD(x)]$ and also trained to have low value, zero ideally, in fake cases $G(z)$ by maximizing $\mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z))]$.

Preprint. Under review.

3 GAN Implementation

3.1 Dataset

To see the performance of GAN, the well-known datasets MNIST[1] and SVHN[2] are introduced. The most significant difference is that MNIST only involve black and white (bilevel) images and the digits are size-normalized and centered. While SVHN contains colored, variable-resolution, real-world images.

3.2 Hyper-parameters

To balance the efficiency and the performance of GAN, some hyper-parameters are set as follows:

- **BATCH_SIZE** is set to 256 to accelerate the iteration.
- **EPOCHS** is set to 200 to produce better result of GAN.
- **noise_dim** represents the dimension of noise z which is set to 100.
- **num_examples_to_generate** is the number of images to generate which is set to 64.

3.3 Neural Network Structure

The GAN requires two neural nets:

- **Generator** is made up with two `Dense` sections followed by three `Conv2DTranspose` sections to generate the same image size as trained images. Except the output layer using `tanh`, all sections take `ReLU` as activation function.

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(128, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.ReLU())

    model.add(layers.Dense(7*7*256, use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.ReLU())
    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.ReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.ReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

Figure 1: Generator Net.

- **Discriminator** is a image classifier that based on convolutional neural works. Therefore, its output layer is a `Dense` layer with `sigmoid` activation function to indicate the decision of Discriminator on inputs. It starts with two 5×5 `Conv2D` filter layers taking `LeakyReLU` as activation function, followed by one `MaxPooling2D` layer, one `Dense` layer with `ReLU` activation function and a `Dropout` layer.

[1] MNIST dataset of handwritten digits, available at `http://yann.lecun.com/exdb/mnist/`

[2] Street View House Numbers dataset, available at `http://ufldl.stanford.edu/housenumbers/`

```
def make_discriminator_model():
        model = tf.keras.Sequential()
        model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                                                                input_shape=[28, 28, 1]))
        model.add(layers.LeakyReLU(alpha=0.2))

        model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
        model.add(layers.LeakyReLU(alpha=0.2))
        model.add(layers.MaxPooling2D())

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

        return model
```

Figure 2: Discriminator Net.

4 GAN Performance

4.1 MNIST

- **Loss**
  By tensor board, the loss for both Generator and Discriminator can be tracked. The tendency for Generator in blue and Discriminator both meet the expectation:

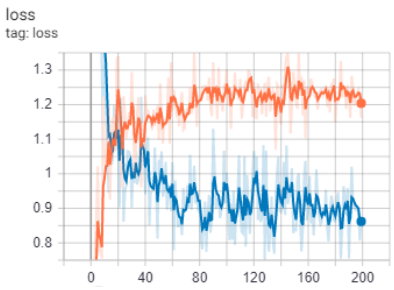

Figure 3: GAN Loss on MNIST

- **Iteration Efficiency**[3]
  For each epoch, the model inputs a image batch of size 256 randomly among the whole training data sets of size $60,000$. Each epoch takes approximately 13.4 seconds to finish.
- **Images**
  For standardized size bilevel number images, at the beginning of the training process, the generated images looks like random noise. However, with the training proceeding, the images becomes much more similar to the hand-written figures. After approximately 50 epochs, the number in generated images can be clearly told. With a few epochs iterated, the generated results seem to reach the convergence. The quality of the images is very closed to those in the paper.[4] The training process is shown in Figure 4.

4.2 SVHN

When dealing with more complicated data sets, for example, the SVHN data sets whose images are not standardized and contains multiple color channel, some adjustment on hyper-parameter and nets' structure is made to improve efficiency and performance of the model.

- **BATCH_SIZE** is set to be 128.

[3] Iteration time running on Google Colab with GPU acceleration.

[4] Figure 2a in *Generative Adversarial Nets*, available at `https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf`

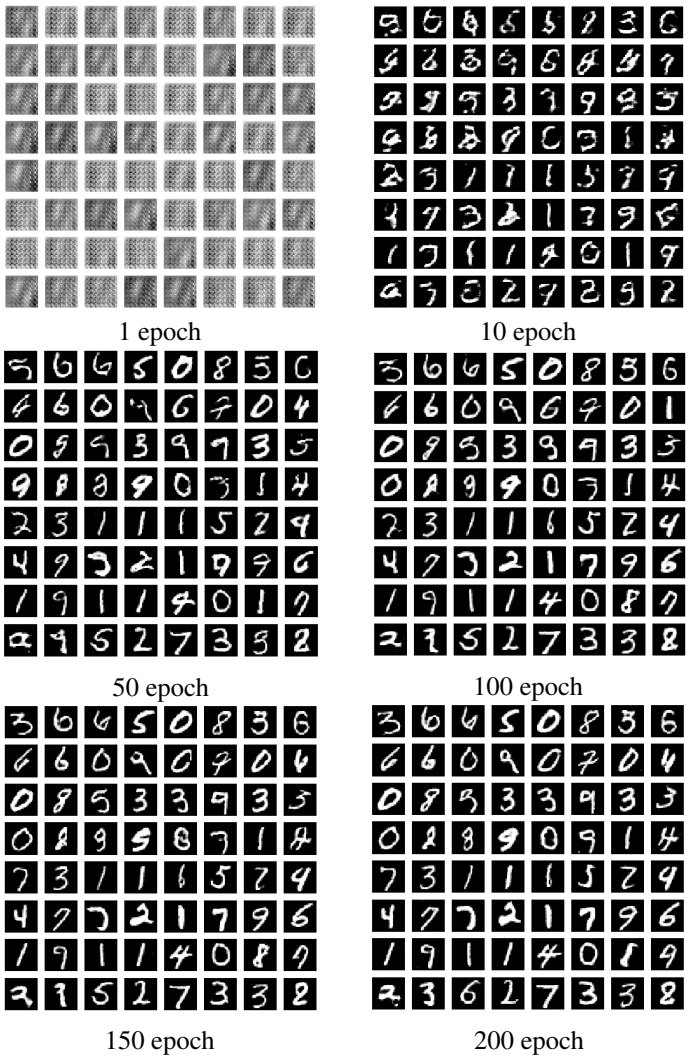

Figure 4: GAN on MNIST

- **Activation function** in Generator nets is replaced with `LeakyReLU` except the output layer remains `tanh`.

Even though with these adjustments listed above, the result of GAN model is still not as decent as the result of MNIST.

- **Loss**
  Unlike the Figure 3, the loss curve in Figure 7 of both agents goes oppositely to the expected direction. With the training process proceeding, the loss of Generator decrease while the loss of Discriminator increases. Such tendency verifies the performance of GAN on SVHN is not satisfying as expected.
- **Iteration Efficiency**
  For each epoch, the model inputs a batch of size 128 of images of size 32×32 with RGB color channel. Each epoch takes approximately 40.1 seconds to finish.[5]
- **Images**
  The outputs with few training steps looks like the random noise as the case in MNIST. After few more training steps, colors becomes more vivid and some numbers can be recognized.

[5]Iteration time running on Google Colab with GPU acceleration.

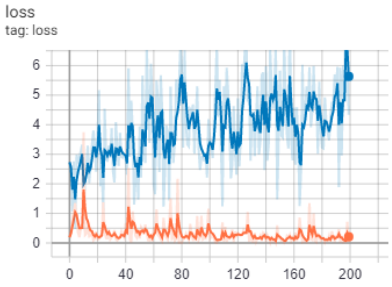

Figure 5: GAN Loss on SVHN

Though with the more and more epochs, the images with clear numbers can be generated, some results seems to have extreme similar pattern, most images generated contains letter 2 with blue and white colors. The generated images suffers the fading color issue as well. The training process is shown in Figure 6.

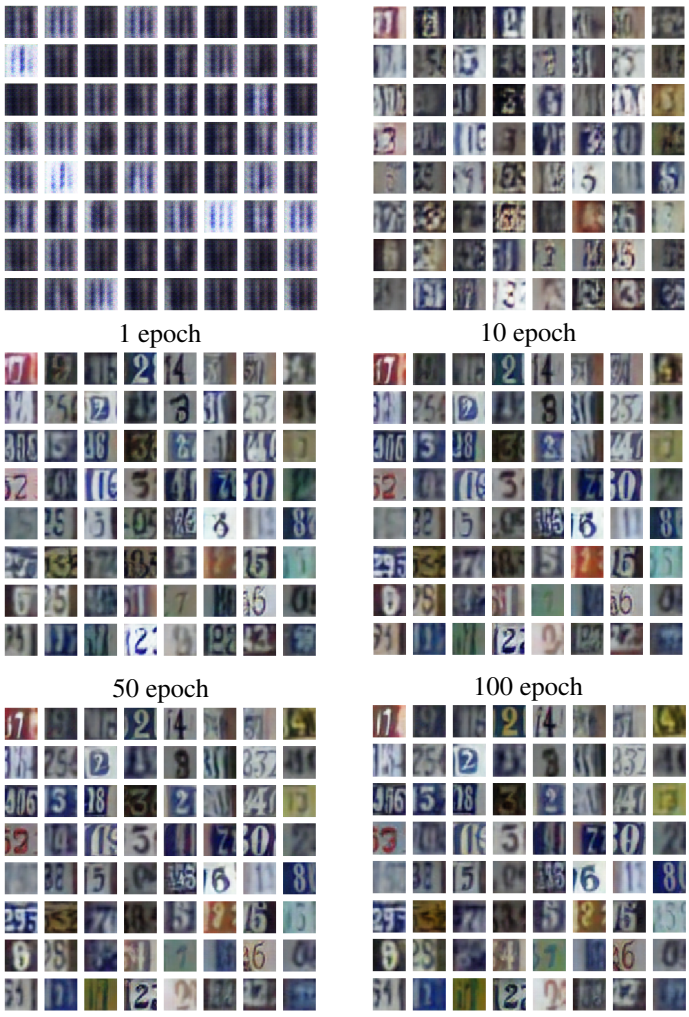

Figure 6: GAN on SVHN

5 Wasserstein GAN

Wassersatein GAN is one of the improved GAN method to help improve the performance of uniform GAN and solve the problem such as mode collapse observed in the case of GAN applied to SVHN data set.

There are two main difference between WGAN and GAN:

- **Objective function**
  Basically, Wassersatein distance measures the distance between two probability distributions p_r and p_z:
  $$W(p_r, p_z) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x \sim p_r}[f(x)] - \mathbb{E}_{x \sim p_z}[f(x)]$$
  where f is K-Lipschitz continuous.
  Therefore, the objective function of WGAN becomes:
  $$L = W(p_{data}, p_z) = \max_{w \in W} \mathbb{E}_{x \sim p_{data}}[f_w(x)] - \mathbb{E}_{x \sim p_z(z)}[f_w(x)]$$
  In WGAN, Discriminator is trained to lean a K-Lipschitz continuous function to compute Wasserstein distance. The Wasserstein distance decrease as the loss becomes smaller and the Generator can produce images more closed to the real data distribution.
- **Optimizer**
  WGAN takes `RMSProp` as the optimizer for empirical and efficiency reasons while GAN takes `Adam`.

6 WGAN Performance

In order to have reasonable comparison, the structure of nets and hyper-parameters are kept same as uniform GAN for each data set.

6.1 MNIST

- **Loss**

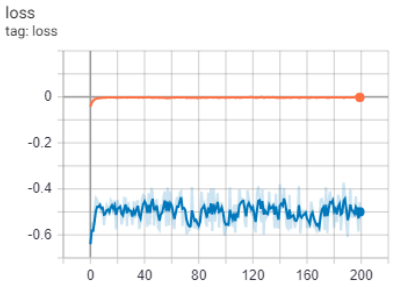

Figure 7: WGAN Loss on MNIST

- **Iteration Efficiency**
  Each epoch takes a image batch of size 256 randomly and takes approximately 22.3 seconds which a bit slower than GAN.
- **Images**
  At the very beginning of the training, the images generated by WGAN is much better than those by GAN. The background and the number area can be distinguished immediately, while for GAN images are just random noise. After approximately 10 training epoch, there are already some numbers are clearly shown. However, with even more epochs, there are still some numbers that cannot be distinguished. This indicates that WGAN has faster convergence rate than GAN but the quality may not exceed GAN.

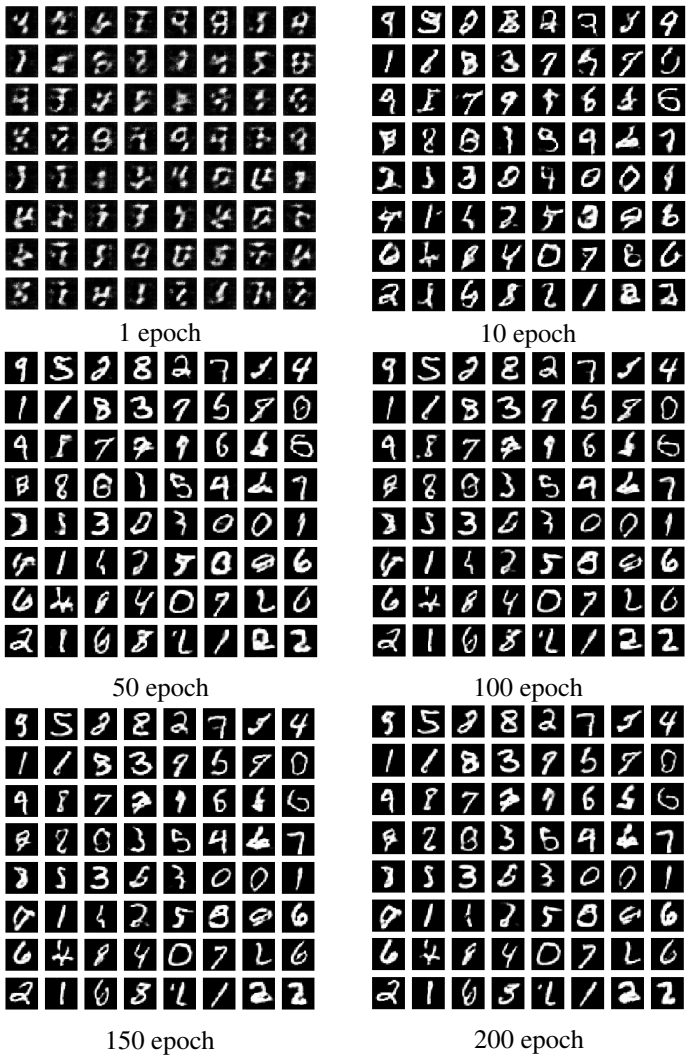

Figure 8: WGAN on MNIST

6.2 SVHN

- **Loss**

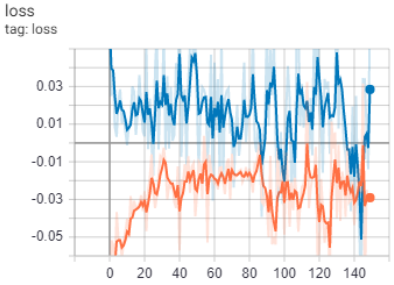

Figure 9: WGAN Loss on SVHN

- **Iteration Efficiency**
  Each epoch takes a batch of size 128 randomly and costs approximately 71.9 seconds. It is significantly slower than other cases.

- **Images** Similar to the cases of comparing two methods on MNIST, images generated after very few training steps by WGAN is much more colorful than images generated by GAN with the same number of training step. For SVHN data set, WGAN still beats GAN with respect to convergence rate but GAN have better quality of generated images. Additionally, WGAN model has no mode collapse issue but the fading color problem still exists.

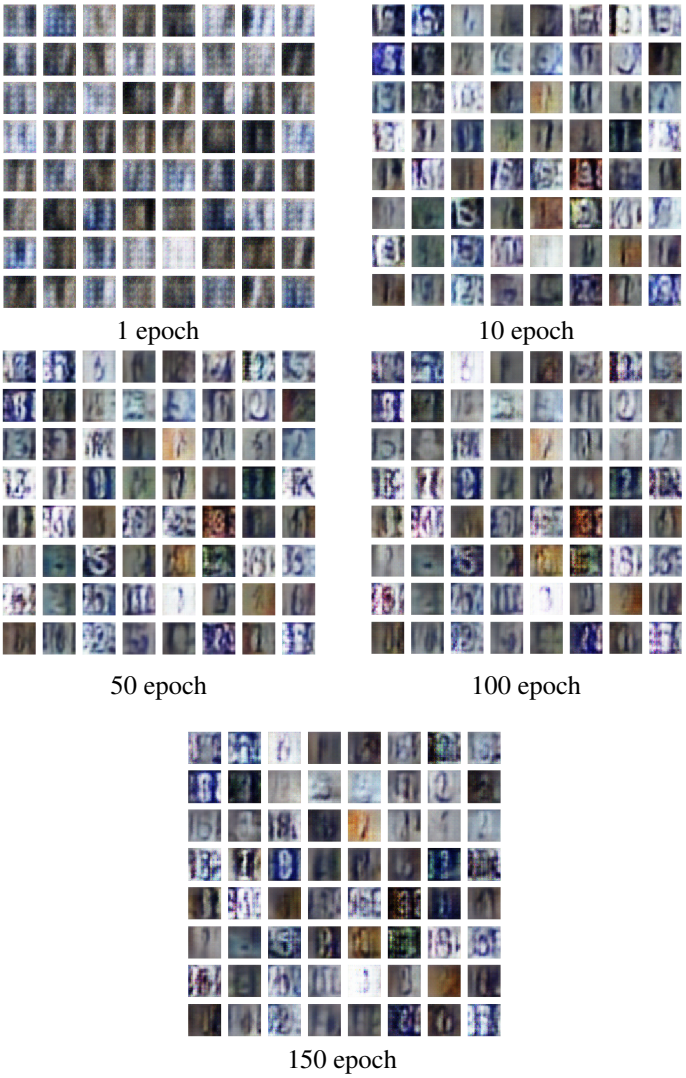

Figure 10: WGAN on SVHN

7 Conclusion

For MNIST and SVHN data set, with the same Generator and Discriminator net structure, GAN have better quality in generated images than Wasserstein GAN. However, WGAN is faster in convergence and it helps in solving mode collapse issue which occurs for GAN in complex training cases.

Both method have some fading color problems in SVHN data set, which might be caused by unclear training images in SVHN or it may also be related with the net structure. To solve this problem, further research is in need.

References

[1] Goodfellow, I.J. et al. (2014)*Generative Adversarial Nets*. Retrieved from https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[2] Arjovsky, M. & Chintala, S. & Boutou, L. (2017,Dec 6)*Wasserstein GAN*. Retrieved from https://arxiv.org/pdf/1701.07875.pdf

[3] Weng, L. (2017, Aug 20) *From GAN to WGAN*[Blog post]. Retrieved from https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html