
Implementing GAN on MNIST and SVHN

Shengjie Sun

ss5593

Department of Statistics
Columbia University
New York, NY 10027
ss5593@columbia.edu

Zeyu Yang

zy2327

Department of Statistics
Columbia University
New York, NY 10027
zy2327@columbia.edu

Abstract

We implement DCGAN on MNIST and SVHN dataset. The generated samples for both datasets are great although it takes quite some time to train the model. To help the model converge faster, we implement WGAN as well. The result **TBD**.

1 Introduction

Generative Adversarial Nets (GAN), first introduced by Ian Goodfellow in 2014[1], is a model that can be used to generate new images. It has been a hot topic since then.

The core idea of GAN is to create a two-player game. Build a generator that creates fake images while the discriminator tells whether the image is true or fake.

We train D to maximize the probability of assigning the correct label to both training examples and samples from G . We simultaneously train G to minimize $\log(1 - D(G(z)))$.

The optimum case is that the generate can fully recover the distribution of the data and the discriminator cannot tell whether the image is fake or not i.e. the output of discriminator is $1/2$.

2 Implementation on MNIST

MNIST dataset is a dataset of handwritten images, including 60,000 training images and 10,000 testing images. All digits have been size-normalized and centered in a 28×28 fixed-size image[2].

The dataset has been generally used to conduct pattern recognition. In this project, however, we will use the training image to train our GAN model and generate similar images.

2.1 Architecture

We used Deep Convolutional GAN (DCGAN)[4] as our structure. It has plenty convolutional layers and does not have max pooling layer. We mainly use transpose convolutional layer to perform upsampling in our model.

2.1.1 Generator

The input of the first dense layer the 100×1 random noise vector. We have 12,544 neurons, apply batch normalization and activation function Leaky Relu, and reshape it to $7 \times 7 \times 256$.

The second Convolutional Transpose layer outputs $7 \times 7 \times 128$.

The third and fourth Convolutional layers have stride 2 and padding *same*, they alter the first two dimensions of the input and make the input from 7×7 to 14×14 to 28×28 . The filter makes the third dimension from 128 to 64 to 1.

Thus the final output is a $28 \times 28 \times 1$ black and white image.

- First layer: Dense
 - Input: 100×1 vector
 - Units: 12,544
 - Batch normalization
 - Activation: leaky relu
 - Reshape 12,544 to $7 \times 7 \times 256$
- Second layer: Conv2DTranspose
 - Input $7 \times 7 \times 256$
 - Filter: 128
 - Kernel size: 5
 - Stride: 1
 - Padding: same
 - Batch normalization
 - Activation: leaky relu
- Third layer: Conv2DTranspose
 - Input $7 \times 7 \times 128$
 - Filter: 64
 - Kernel size: 5
 - Stride: 2
 - Padding: same
 - Batch normalization
 - Activation: leaky relu
- Fourth layer: Conv2DTranspose
 - Input $14 \times 14 \times 64$
 - Filter: 1
 - Kernel size: 5
 - Stride: 2
 - Padding: same
 - Activation: tanh
 - Output $28 \times 28 \times 1$

2.1.2 Discriminator

The discriminator takes in a $28 \times 28 \times 1$ array. After three convolutional layers and one dense layer, the output will be a scalar. This scalar will be applied to a logistics function in future steps to be a probability that lies between 0 and 1. It is the probability of the input being a true image or a fake image.

- First layer: Conv2D
 - Input: $28 \times 28 \times 1$ array
 - Filter: 64
 - Kernel size: 5
 - Stride: 2
 - Padding: same
 - Activation: leaky relu
 - Dropout: 0.3
- Third layer: Conv2D
 - Filter: 128
 - Kernel size: 5

- Stride: 2
- Padding: same
- Activation: leaky relu
- Dropout: 0.3
- Fourth layer: Conv2D
 - Filter: 256
 - Kernel size: 5
 - Stride: 2
 - Padding: same
 - Activation: leaky relu
 - Dropout: 0.3
- Fifth layer: Flatten
- Sixth layer: Dense with output 1

2.1.3 Hyperparameters

- Epoch: 100
- Batch: 256
- Learning rate
 - Generator: $1e-3$
 - Discriminator: $1e-4$

We have experimented on different batches and epochs and we believe a batch of 256 samples and 100 epochs already produced satisfying result. In fact, the quality of the image does not seem to have significant improve after 50 epochs.

The reason why we choose the learning rate for the discriminator 10 times smaller than the learning rate for generator is because we noticed the loss of discriminator is much smaller than the loss of generator. To avoid the gradient jumping back and forth around the optimum spot, we lower the learning rate for discriminator. The training result shows that the loss for both of them are at the same magnitude after the modification.

2.2 Result

2.2.1 Tensorboard

As we can observe from Figure 1, training loss of both generator and discriminator is fluctuating. And generally when the loss of generator decreases, the loss of discriminator increases. This is the representation of the two player game i.e. both players want to minimize their own loss.

Although the loss remains around the same level and does not continue decrease, the quality of the images are in fact improving as we can see in the following section.

2.2.2 Generated samples

From Figure 2, we can see the generated samples has a good approximation after 30 epochs.

The samples from the 10th are quite blurry and they cannot have a good representation of number 2, 5, 8, etc.

The latter samples are clearer and the edges of the numbers are smoother. Number 0, 1, 3, 5, 7, 9 are quite ideal.

Although not all samples are of high quality, there are a proportion of the images that have the quality of the samples presented in paper[1] 2(a).

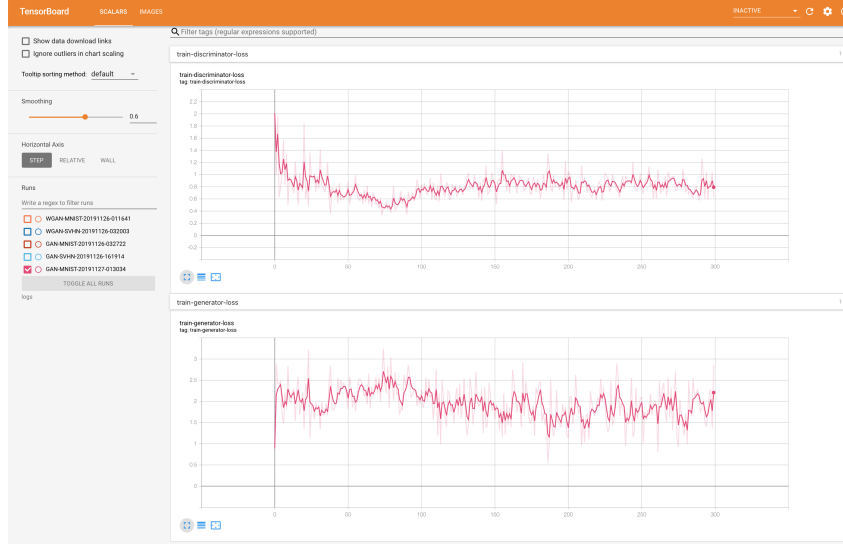
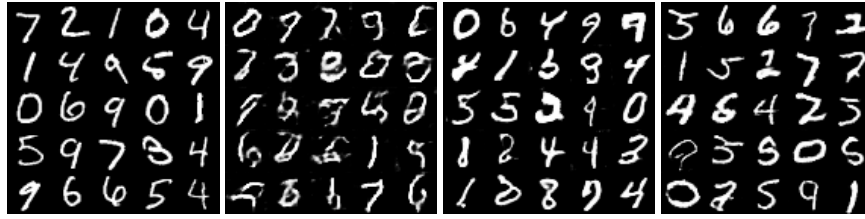


Figure 1: Tensorboard GAN MNSIT



(a) MNIST data (b) Epoch 10/100 (c) Epoch 30/100 (d) Epoch 100/100

Figure 2: Comparison of MNIST data and generated samples from DCGAN

3 Implementation on SVHN

SVHN dataset is a dataset similar to MNIST, containing real-world digit images that obtained from house numbers in Google Street View images. The images have been cropped into 32×32 fixed-size array with three channels.

In this project, we will again use the training image to train our GAN model and generate similar images.

3.1 Architecture

3.1.1 Generator

The generator is slightly modified from the generator for MNIST. The output of the first dense layer is changed to $8 \times 8 \times 256$ and the filter of the fourth layer (Conv2DTranspose) has changed from 1 to 3 so that the output of the whole generator would be a $32 \times 32 \times 3$ image.

3.1.2 Discriminator

Discriminator is exactly the same as the previous one.

3.1.3 Hyperparameters

- Epoch: 200
- Batch: 256

- Learning rate
 - Generator: $1e-3$
 - Discriminator: $1e-4$

Since the SVHN dataset is more complicated, we train the model with 200 epoches.

3.2 Result

3.2.1 Tensorboard

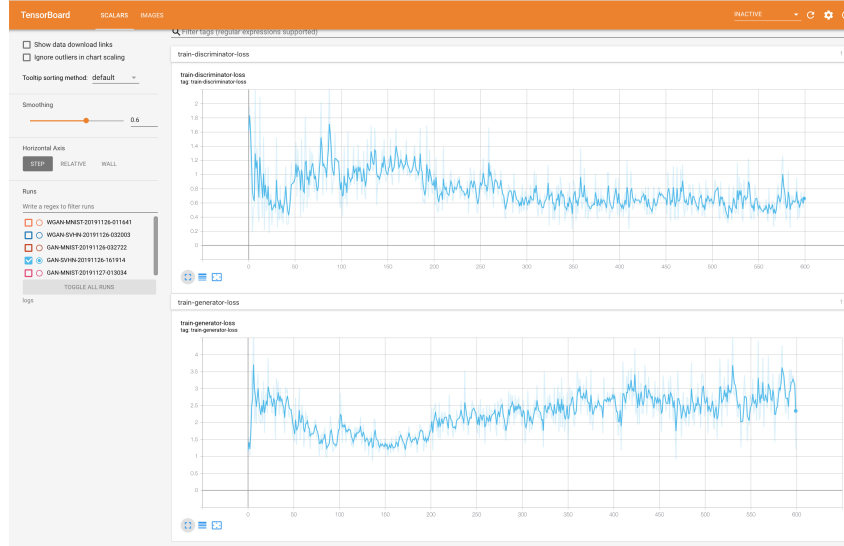
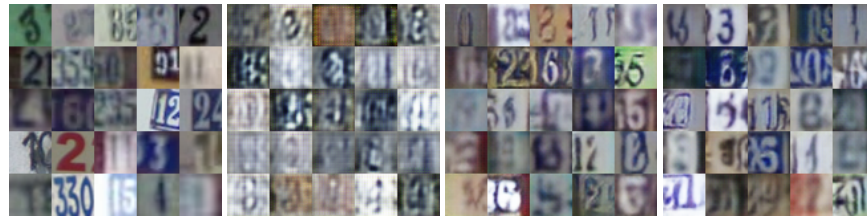


Figure 3: Tensorboard GAN SVHN

3.2.2 Generated samples



(a) SVHN data (b) Epoch 30/200 (c) Epoch 150/200 (d) Epoch 200/200

Figure 4: Comparison of SVHN data and generated samples from DCGAN

The quality of the generated samples are not as good as the MNIST samples since the SVHN dataset is more complicated. This complexity makes the model harder to converge i.e. uses more epoches to get good generated samples.

Although complicated as the dataset is, the generated samples demonstrate some qualities such as the variety of the samples. There are different combinations of green, white, and red background with white, blue, and red fonts.

4 WGAN

Wasserstein GAN is a modified GAN introduced by Arjovsky, Martin, et al.(2017)[5]. It proposed Wasserstein that has a better property than Jensen-Shannon divergence.

Down to the implementation, the main difference is the loss function. We modified the loss function from *BinaryCrossentropy* to a customized loss function.

The generator and discriminator are exactly the same as the GAN.

The hyperparameters are the same except this time we only train 50 epoches for each model.

4.1 WGAN on MNIST

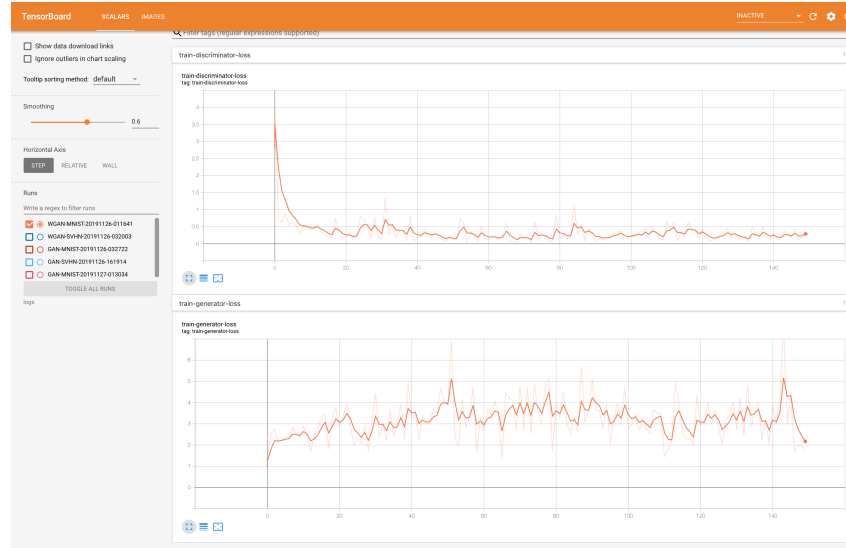


Figure 5: Tensorboard WGAN MNSIT

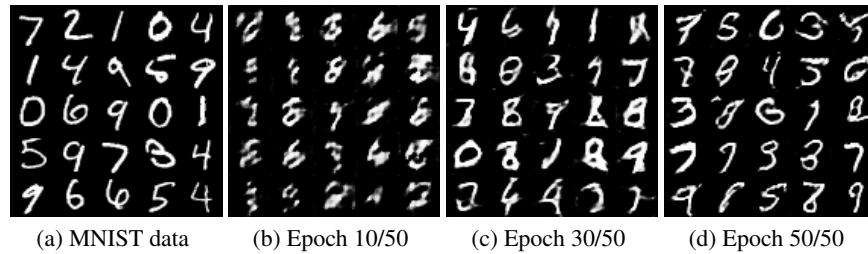


Figure 6: Comparison of MNIST data and generated samples from WGAN

4.2 WGAN on SVHN

5 Summary

xxx

5.1 Future steps

Since plenty new models have been published other than WGAN, we can try other models such as triple-GAN, Self-attention neural network, etc..

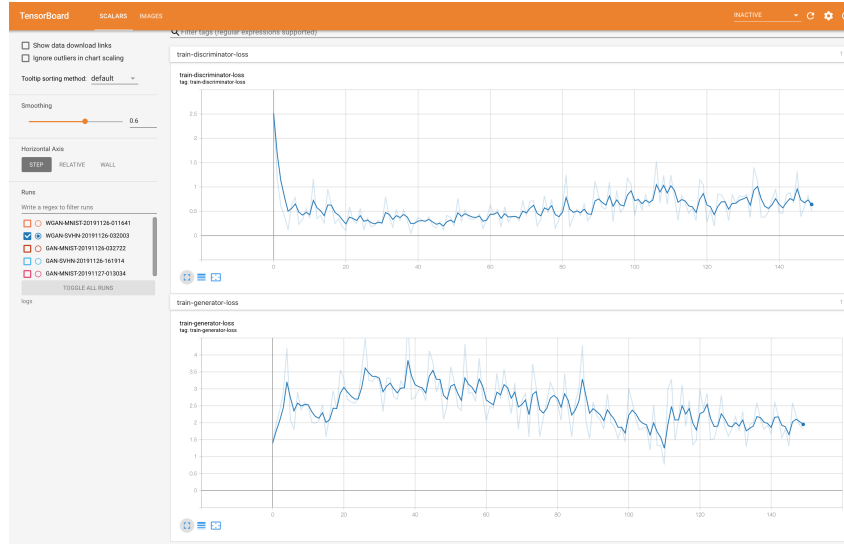


Figure 7: Tensorboard WGAN SVHN

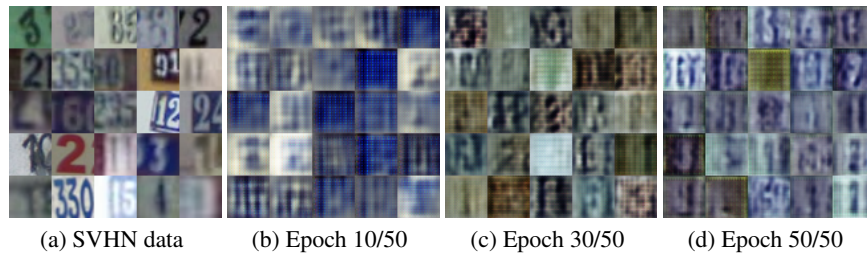


Figure 8: Comparison of SVHN data and generated samples from WGAN

References

- [1] Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems. 2014.
- [2] LeCun, Yann, et al. "THE MNIST DATABASE." MNIST Handwritten Digit Database, <http://yann.lecun.com/exdb/mnist/>.
- [3] Netzer, Yuval, et al. "Reading digits in natural images with unsupervised feature learning." (2011).
- [4] Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015).
- [5] Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein gan." arXiv preprint arXiv:1701.07875 (2017).