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 Implmentation on MNIST

2.1 Architecture

We used Deep Convoltional GAN (DCGAN)[2] as our structure.

2.1.1 Generator

- First layer: Dense
  - Input: 100×1 vector
  - Output $7 \times 7 \times 256$
  - Batch normalization
  - Activation: leaky relu
- Second layer: Conv2DTranspose
  - Filter: 128
  - Kernel size: 5
  - Stride: 1
  - Padding: same
  - Batch normalization
  - Activation: leaky relu

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

- Third layer: Conv2DTranspose
  - Filter: 64
  - Kernel size: 5
  - Stride: 2
  - Padding: same
  - Batch normalization
  - Activation: leaky relu
- Fourth layer: Conv2DTranspose
  - Filter: 1
  - Kernel size: 5
  - Stride: 2
  - Padding: same
  - Activation: tanh

2.1.2 Discriminator

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

We have experimented on different bacthes and epoches and we believe a batch of 128 samples and 50 epoches already produced satisfying result. The quality of the image does not seem to have significant improve after 50 epoches.

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.

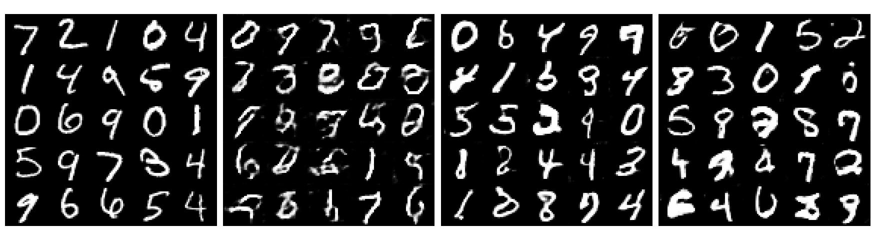
2.2 Result

Training process in tensorboard

(a) MNIST data

Figure 1: Tensorboard GAN MNSIT

the generated samples compared with training dataset and figure 2a in paper



(a) MNIST data (b) Epoch 10/50 (c) Epoch 30/50 (d) Epoch 50/50

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

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

The samples from the 10th are quite blury 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.

3 Implmentation on SVHN

3.1 Architecture

3.1.1 Generator

The generator is slightly modifed 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

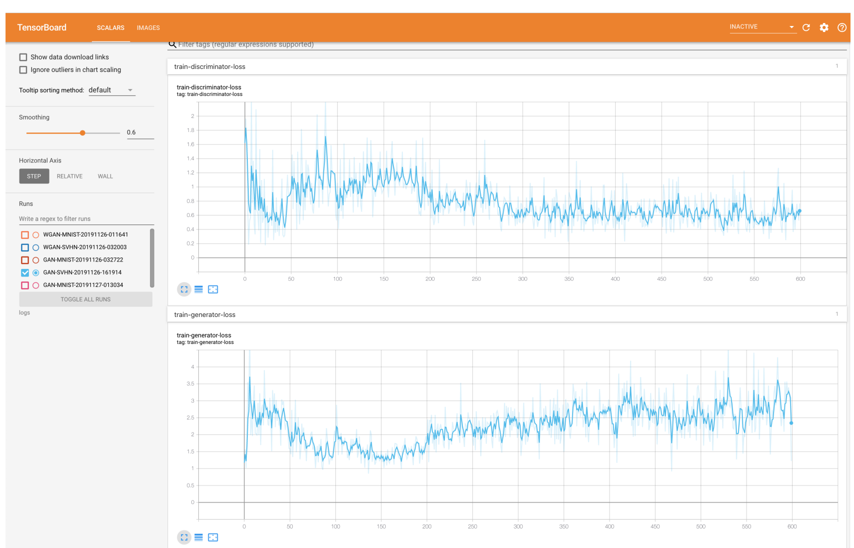
Discriminator is exactly the same as the previous one.

3.1.3 Hyperparameters

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

3.2 Result

Training process in tensorboard



(a) MNIST data

Figure 3: Tensorboard GAN SVHN

the generated samples compared with training dataset

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 gree, white, and red background with white, blue, and red fonts.

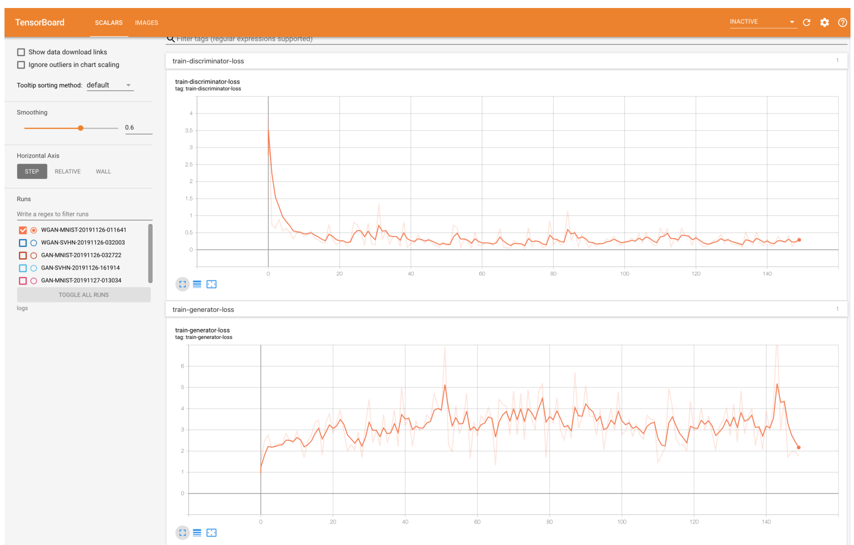
(a) SVHN data (b) Epoch xx/xx (c) Epoch xx/xx (d) Epoch xx/xx

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

4 WGAN

Wasserstein GAN is a modified GAN introduced by Arjovsky, Martin, et al.(2017)[3]. It proposed Wasserstein that has a better property than Jensen-Shannon divergence.

4.1 WGAN on MNIST



(a) MNIST data

Figure 5: Tensorboard WGAN MNSIT

4.2 WGAN on SVHN

5 Summary

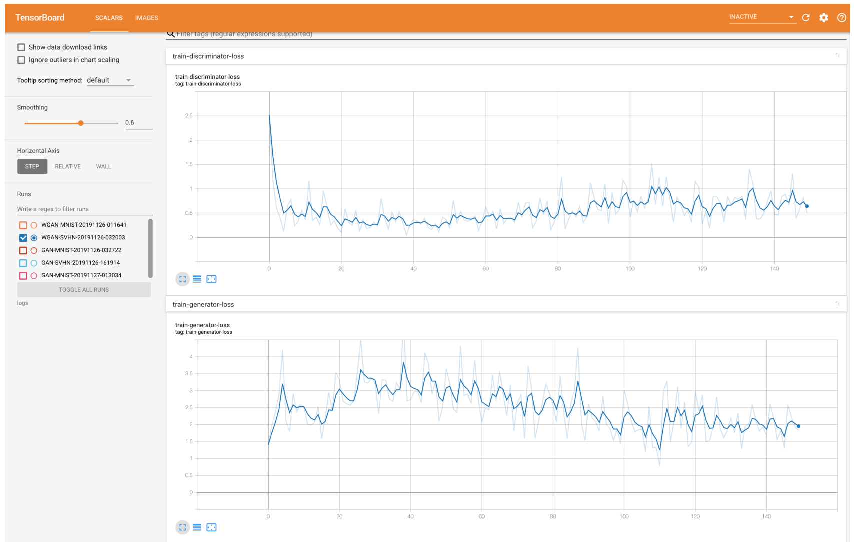
xxx

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..

References

[1] Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems. 2014.



(a) SVHN data

Figure 6: Tensorboard WGAN SVHN

[2] Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015).

[3] Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein gan." arXiv preprint arXiv:1701.07875 (2017).