Final Project: Generative Adversarial Networks

Author
Wensong Qiao wq2144
Xinquan Wang xw2566
Yucheng Zhang yz3546

Abstract

In this project, we explore the topic of Generative Adversarial Networks. Like the name of this network, we simultaneously train two models: a "generative" model and it's "adversary", a discriminative model. The generative model G captures the data distribution and the corresponding discriminative model D try to estimate the probability that samples are generated from G rather than the real data. In addition, the generative model is trained to generate a better "fake artifact" that could cheat the discriminator and the discriminator is trained to discriminate samples from real data.

In the beginning, the discriminator is just a partially accurate classifier, which means the discriminator is also not a good classifier. Through the inner loop of the algorithm, the discriminator is trained to better discriminate the sample and the D(x) (the probability that a sample will be identified as a real sample) will converge to

$$\frac{P_{data(x)}}{P_{data(x)} + P_{g(x)}}$$

And after an update to the generator, the generator will perform better in cheating the discriminator. Within the two arbitrary functions of G and D, a unique solution exists with G recovering(almost perfectly) the real data distribution and D converges to 50% everywhere (which means although the discriminator is well trained, it could not tell the difference between a fake sample and a real one).

1 Introduction

In the adversarial network framework we introduced above, the generative model G is trained against an adversary: a discriminate model that learns to determine whether an observed sample is generated from the generator or from the real world data set. This process can be considered as artists who learn and try to paint the real world pictures and the art critic who determine whether the pictures are captured from the real world or just drawn by artists. In this case, the Generator model will represent the artists and the discriminator will play the role of art critic. As long as the artists are trying to learn how to make their paints more real, the art critics team is also learning to better identify the real and fake paint shown to them, Competitions in this "real and fake" paints will motivate both teams to sharpen their skills of both generating or identifying the pictures. And this competition will always end up with the result that the fake pictures are so real that even the professional art critics can not tell the difference between a fake image or a real image that they can only make a random guess.

In this project, we will first implement our own GAN with convolutional neural network layers on the famous MNIST data. We will describe the architecture of generator model and convolutional layer and many other hyper-parameters including drop-out in CNN classifier in discriminator. We will show a plot of training tensorboard to make sure that the training process is what we expected. After finishing the first GAN model on the MNIST handwritten digit data, we will present our results:

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

the generated handwritten images generated from the generator model. And we will compare the generated handwritten images with the real handwritten images to see if we can identify which image is the fake one.

For the second part of our project, we will implement a similar process on the SVHN, the street view house number images. We will build a generative model trained with real SVHN images and we will also build up a convolutional neural network classifier to discriminate the real world images and the fake images. We are interested in the final images generated from the well trained generator. We want to find out how is the quality of these images compared with the quality of the previous images generated from the GAN on MNIST handwritten images.

In addition, there are a lot of improved version of the traditional GAN method. One of the improved methods is Wassertain Generative Adversarial Networks (WGAN). We will apply this new method and see if this method generated a series of images with higher quality compared with the traditional GAN method.

2 Method

An Adversarial network is appropriate to apply when both models are multi-layer perceptrons. The generator model and convolutional neural networks both satisfies this condition so the adversarial network method is especially useful in this case. We first define a prior on input noise variables as $p_z(z)$to learn the generator's distribution pg. Then this $p_z(z)$ represent a mapping route to data space $G(z;\theta g)$. We also define another multi-layer perceptron $D(x;\theta d)$that output a single probability. D(x)represents the probability that a single sample comes from the real data rather than generator, the pg. As we introduced above, we train these two models to compete with each other and our purpose is to get a well trained generator which can generate samples that are good enough to cheat the discriminator or even the human eyes. We train D, the discriminator to maximize the probability to correctly discriminate the fake image generated from the real images without recognizing the real images into fake ones. At the same time, we train G, the generator to minimize the probability that the samples generated from G being discriminated by D. This probability can be written as log(1-D(G(z))) and the purpose of G is to minimize this term. Then, we can consider that G and D are playing with the min/max game with the key value function

$$V(G,D) : E_{x\ p_{data}(x)}[log(D(x))]] + E_{z\ p_z(z)}[log[(1 - D(G(z)))]$$

From the function listed above, we know that logarithm is a monotone function that the value increases as the value inside the log increases. The discriminator D wants to maximize this function as it wants to increase the value of expected log(D(x)) since x comes from the real world data and it also want to maximize the expected value when D is cheated by the fake image under the distribution of pz(z). Similarly, the generator G wants to maximize the expected value when the generator successfully cheat the discriminator.

The inner loop of algorithm we use can be described in the following way:

1) Generate m noise samples $\{z^1, z^2...z^m\}$ from our generator under distribution of p_g.

2) Choose m samples $\{x^1, x^2...x^m\}$ from the real world data set under distribution of p_{data}.

3)Update the discriminator by ascending its gradient:

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

4) Repeat steps 1-3 k times to update the discriminator.

5) Generate m noise samples$\{z^1, z^2...z^m\}$from our generator under distribution of p_g.

6) Update the generator by descending its stochastic gradient:

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

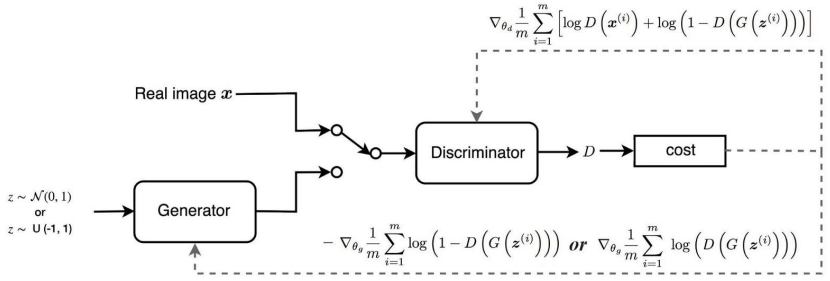

Figure 1: GAN

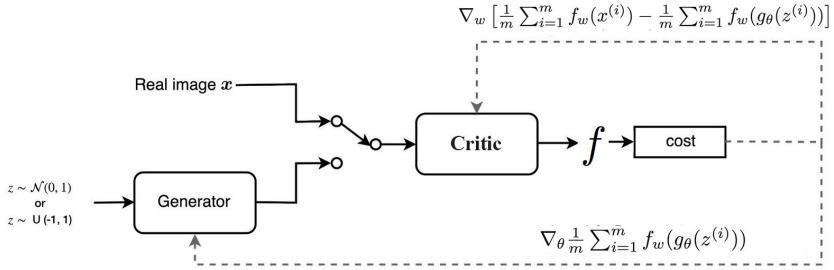

Figure 2: WGAN

7) Repeat steps 1-6 until the result converges to global optimal $p_g = p_{data}$.

Similar to GAN, Wasserstein GAN (WGAN) is a GAN variant which uses the 1-Wasserstein distance as loss function, rather than the JS-Divergence, to measure the difference between the model and target distributions. GAN:

The above two charts show the logic of GAN and WGAN

3 Implementation and Results

In our project, we have used two datasets——MNIST and SVHM. For each dataset, we have implemented three different architectures, they are baseline model, modified model, and WGAN model. The following are detailed architectures and loss plots for each architecture:

(1) MNIST Dataset MNIST dataset, a.k.a Modified NIST (National Institute of Standards and Technology) dataset, was developed by Dr. Yann LeCun, NYU, and Corinna Cortes Christopher Burges. For MNIST dataset, we used the tutorial's data preprocessing way which normalized the image to [-1,1]. For comparison, we put some original pictures here

3.1 Baseline model

For the MNIST dataset baseline model,we use the architectures in the tutorial. Therefore, just need to refer to https://www.tensorflow.org/tutorials/generative/dcgan. However, we set epoch = 100 instead of 50.For other parameters, se set binary cross entropy as our loss function,adam as our optimizer with a e-4 learning rate.

3.2 Modified model

For modified MNIST dataset, we keep the parameter/hyper-parameters as the baseline model. we just did a tiny adjustment for generator. We applied (3,3) filters instead of (5,5) filters. Because the original image is 28*28, we believe the smaller filters can get a more fine texture information.

Figure 3: Loss Comparison - MNIST

Figure 4: Comparison - MNIST

For discriminator, we add activation function——sigmoid. Because this is a binary classification, we just want to know whether the image is fake or not. Usually, sigmoid is good to transfer this kind of information.

3.3 WGAN model

For WGAN model, we keep parameter/hyperparameters as the baseline model.However, we define Wasserstein distance as our loss function here. For architecture, I just use the DCGAN architecture.

(2)SVHM Dataset

For SVHM dataset, we chose format 2: cropped digits. For getting a good performance, we have done some data preprocessing, (1) transferred the data into grayscale. It will reduce our training time and focus more on the image of numbers instead of colors. (2) normalization, it can make images of digits were normalized for the size intensity and centered. Usually, it brings us a better performance and accelerate the training speed. We will give some original images here for comparison.

3.4 Baseline model-SVHM

We set epoch = 100, binary cross entropy as our loss function,adam as our optimizer with a e-4 learning rate. For the generator, we used three inverse convolution layers with (5,5) filliters and batch normalization layers. We chose LeakyReLu as our activation functions to prevent gradient vanishing.

For discriminator, we add max pooling layers. Max pooling layers can extract more outlier/texture information. We put more dense layers as well, one dense layer just handle linear problems, however the multiple layer handle non-linear problems.

3.5 Modified model-SVHM

For modified model, we keep parameter/hyperparameters as the baseline model. We changed a smaller filter with (3,3) size. We tried different parameters, adding more layers, however, tiny wrong change will make the train fail.

For discriminator, as you can see, we used a very complicated architecture. Discriminator, in fact, is a kind of classification CNN. This kind of architecture of CNN can have a good accuracy rate to

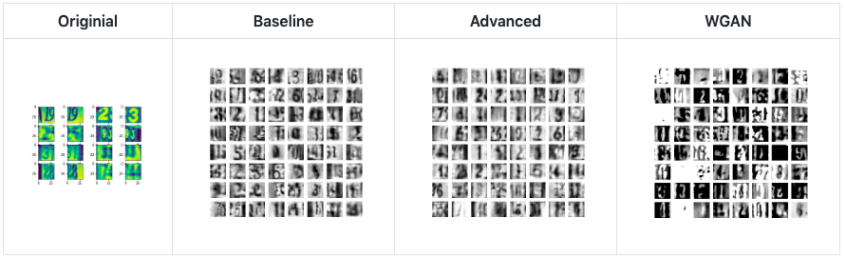

Figure 5: comparison - SVHM

Figure 6: Loss comparison - SVHM

classify the SVHN images. We add one more convolution layers and dense layers to extract higher dimension information.

3.6 WGAN model-SVHM

For WGAN model, we keep parameter/hyperparameters as the baseline model. For architecture, I just use the DCGAN architecture.

4 Insights And Conclusion

After analyzing the plots and comparing of images, we can see that 1) for images, all MNIST dataset models get good images, and we almost can't distinguish which one is better from eyes, but take further look, we can see that WGAN produce a higher granularity for the figures of number. For SVHN, all images are not very clear with a good resolution, only baseline model is slightly better. After we compared with original images, maybe it's because the quality of data set is not so good.

2) for loss function plots, we can know the loss function of discriminators and the loss function of generators has an inverse proportion relationship, because they are 'fighting' with each others. All loss functions converge, for GAN, it just start from a value, and vibrate to decrease loss a little bit. However, for WGAN, it starts at a very low value, and goes directly to a good performance.

3) After experiencing a heavy adjustment of parameters/hyperparameters, we found that GAN is not robuost and stable. Even, we just adjust one fittler size or add a layer of neural networks, the image may fail to produce.

4) GAN has a comparely high frequency to have gradient vanishing, for here, the result is to get a all-black image.

5)We compare GAN with WGAN. Although GAN is highly valued by lots of people and gain a good achievement as unsupervised generation model, we can still see from the result that GAN has lots of problems, like it is not robust, very depends on the architecture of discriminators and generators, patient of adjusting parameters, the comparily high frequency of Mode collapse, and etcs. For solving all these questions, we tried WGAN to improve the result. WGAN uses Wasserstein

distance (earth-mover distance) to measure the distance of two distributions. Therefore, we do not need to balance the capabilities of generator and discriminator, and increase the robustness, decrease the frequency of gradient vanishing. [4]

5 Limitations

Although we get some good results in this project and learn lots of new knowledge, there are some limitations we need to improve in the future.

1) We did not try to adjust every parameter/hyperparameter. For example, in neural networks, we just default to use adam as our optimizer because it is adaptive learning rate algorithm, however, SGD,Momentum, RMSprop maybe better.

2) We did learn lots mature architectures published by others. This time, we just tried WGAN, in the future, we need to try more.

3) The SVHN part problem. We did not get a very good result for SVHN dataset, when we look back to dataset, we found that the original images are blurry. Maybe we can find some image analysis method to improve resolution.

4) Lack of interpretability. When we adjust the architectures, we found even a tiny change, like add a layer or adjust filter size, we may fail to get images. We did not think the principles behind it. We may consider it in the future.

References

[1] Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems. 2014. http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[2] Deep Convolutional Generative Adversarial Network. https://www.tensorflow.org/tutorials/generative/dcgan

[3] Hui Jonathan. "GAN — Wasserstein GAN WGAN-GP" https://medium.com/@jonathan$_hui/gan - wasserstein - gan - wgan - gp - 6a1a2aa1b490$

[4] Wenyi Yu. "When we talk about Deep LearningGAN and WGAN" https://zhuanlan.zhihu.com/p/29394257

[5] Tim Sainburg. "Generative models in Tensorflow 2" https://github.com/timsainb/tensorflow2-generative-models

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 12544)             1254400
_________________________________________________________________
batch_normalization_3 (Batch (None, 12544)             50176
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 12544)             0
_________________________________________________________________
reshape_1 (Reshape)          (None, 7, 7, 256)         0
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 7, 7, 128)         294912
_________________________________________________________________
batch_normalization_4 (Batch (None, 7, 7, 128)         512
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 14, 14, 64)        73728
_________________________________________________________________
batch_normalization_5 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_transpose_5 (Conv2DTr (None, 28, 28, 1)         576
=================================================================
Total params: 1,674,560
Trainable params: 1,649,088
Non-trainable params: 25,472
_________________________________________________________________
```

Figure 7: Modified model generator

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 14, 14, 64)        1664
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 4, 4, 128)         204928
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 4, 4, 128)         0
_________________________________________________________________
dropout_3 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 2, 2, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 513
=================================================================
Total params: 207,105
Trainable params: 207,105
Non-trainable params: 0
_________________________________________________________________
```

Figure 8: Modified model discriminator

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16384)             1638400
_________________________________________________________________
batch_normalization (BatchNo (None, 16384)             65536
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 16384)             0
_________________________________________________________________
reshape (Reshape)            (None, 8, 8, 256)         0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 8, 8, 128)         819200
_________________________________________________________________
batch_normalization_1 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 16, 16, 64)        204800
_________________________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 32, 32, 1)         1600
=================================================================
Total params: 2,730,304
Trainable params: 2,697,152
Non-trainable params: 33,152
_________________________________________________________________
```

Figure 9: Baseline Generator - SVHM

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        1600
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 32, 32, 32)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
batch_normalization_3 (Batch (None, 16, 16, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 64)        51264
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 16, 16, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
batch_normalization_4 (Batch (None, 8, 8, 64)          256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 8, 8, 128)         0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              2098176
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               524800
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 513
=================================================================
Total params: 2,750,593
Trainable params: 2,750,401
Non-trainable params: 192
_________________________________________________________________
```

Figure 10: Baseline discriminator - SVHM

```
Model: "sequential_19"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_31 (Dense)             (None, 16384)             1638400
_________________________________________________________________
batch_normalization_43 (Batc (None, 16384)             65536
_________________________________________________________________
leaky_re_lu_64 (LeakyReLU)   (None, 16384)             0
_________________________________________________________________
reshape_10 (Reshape)         (None, 8, 8, 256)         0
_________________________________________________________________
conv2d_transpose_28 (Conv2DT (None, 8, 8, 128)         294912
_________________________________________________________________
batch_normalization_44 (Batc (None, 8, 8, 128)         512
_________________________________________________________________
leaky_re_lu_65 (LeakyReLU)   (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_transpose_29 (Conv2DT (None, 16, 16, 64)        73728
_________________________________________________________________
batch_normalization_45 (Batc (None, 16, 16, 64)        256
_________________________________________________________________
leaky_re_lu_66 (LeakyReLU)   (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_transpose_30 (Conv2DT (None, 32, 32, 1)         576
=================================================================
Total params: 2,073,920
Trainable params: 2,040,768
Non-trainable params: 33,152
_________________________________________________________________
```

Figure 11: Modified generator - SVHM

Layer (type)	Output Shape	Param #
conv2d_22 (Conv2D)	(None, 32, 32, 32)	1600
leaky_re_lu_67 (LeakyReLU)	(None, 32, 32, 32)	0
max_pooling2d_20 (MaxPooling	(None, 16, 16, 32)	0
batch_normalization_46 (Batc	(None, 16, 16, 32)	128
conv2d_23 (Conv2D)	(None, 16, 16, 64)	51264
leaky_re_lu_68 (LeakyReLU)	(None, 16, 16, 64)	0
max_pooling2d_21 (MaxPooling	(None, 8, 8, 64)	0
batch_normalization_47 (Batc	(None, 8, 8, 64)	256
conv2d_24 (Conv2D)	(None, 8, 8, 128)	73856
leaky_re_lu_69 (LeakyReLU)	(None, 8, 8, 128)	0
max_pooling2d_22 (MaxPooling	(None, 4, 4, 128)	0
batch_normalization_48 (Batc	(None, 4, 4, 128)	512
conv2d_25 (Conv2D)	(None, 4, 4, 64)	32832
leaky_re_lu_70 (LeakyReLU)	(None, 4, 4, 64)	0
max_pooling2d_23 (MaxPooling	(None, 2, 2, 64)	0
flatten_7 (Flatten)	(None, 256)	0
dense_32 (Dense)	(None, 1024)	263168
leaky_re_lu_71 (LeakyReLU)	(None, 1024)	0
dense_33 (Dense)	(None, 512)	524800
leaky_re_lu_72 (LeakyReLU)	(None, 512)	0
dense_34 (Dense)	(None, 216)	110808
leaky_re_lu_73 (LeakyReLU)	(None, 216)	0
dense_35 (Dense)	(None, 1)	217

Total params: 1,059,441
Trainable params: 1,058,993
Non-trainable params: 448

Figure 12: Modified discriminator - SVHM