Final Project: Implementation of Generative Adversarial Networks (GANs)

Jiarui Yang

Chengming Xie

Xinyi Wang

Abstract

This is the final report of Advanced Statistical Machine Learning course presented by authors. The report summarizes the generative adversarial networks (GANs). The report describes: (1) What problem could GAN solve, (2) What are the concept and cost function of GAN, (3) Architecture of our GAN with hidden layers, and implementation on MNIST, SVHN and CelebA data, (4) Result of our GAN model, (5) Implementations using WGAN Model on MNIST and SVHN data, (6) Discussion of important takeaways and potential improvements.

1 Introduction

1.1 Problem Initiation

Generative modeling is an unsupervised machine learning algorithm that could automatically learn patterns of the input data and then generate new data points. This technique could solve the problem when we want to train a classifier but do not have enough data. In some cases, we could create data that improve classifier accuracy using generative models. Moverover, it is also useful for increasing image resolution without introducing artifacts. However, given some high dimensional data such as images or audio, understanding its probability distribution and generating samples following that distribution are not trivial. Generative Adversarial Networks (GAN) provides a possible solution of understanding the probability distribution of high dimensional data by constructing two networks, a Generator G and a Discriminator D.

1.2 Concept and Cost Function of GAN

Paper Source The paper we read is the original paper about GAN algorithm: *Generative Adversarial Nets*, which can be found at

`https://papers.nips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf`

We also follow the Tensorflow tutorial on GAN, which can be found at:

`https://www.tensorflow.org/tutorials/generative/dcgan`

According to the original paper about GAN algorithm, the Generator G is a differentiable function used to understand the probability distribution p_g over data x, by taking the noise variables $p_z(z)$ as an input and $G(z;\theta_g)$, a "fake" sample as a output. The Discriminator D is a traditional supervised learning method that could classify inputs into two classes (whether an input is from "fake" sample or "real" data), represented as $D(x;\theta_d)$. In general, The Generator is trained to "fool" the Discriminator,

Preprint. Under review.

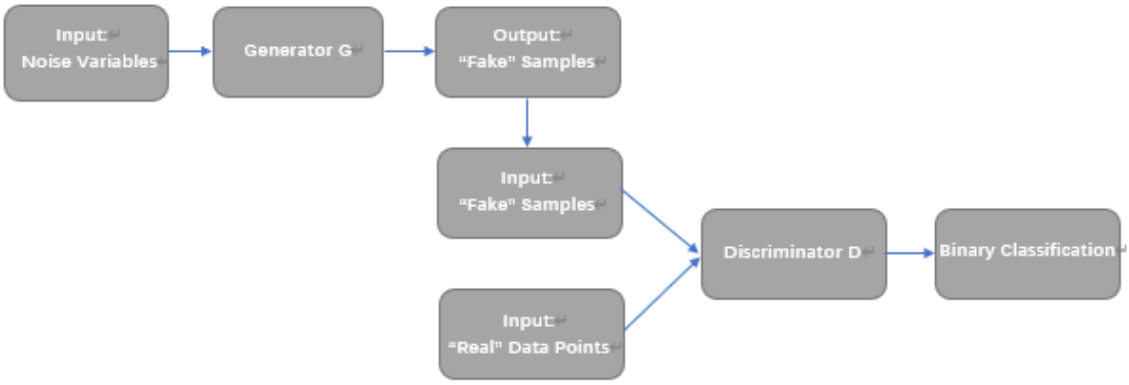


Figure 1: Sample training process of one iteration

and the Discriminator is trained to "uncover" samples from the Generator. Figure 1 shows the general relationship between Generator and Discriminator.

Cost Function In the GAN model, we use log-loss error function for both the Generator and the Discriminator. Specifically, we want to maximize the probability of correct classifications of Discriminator, and simultaneously want to minimize $log(1 - prediction)$, in order to optimize the ability of Generator to "fool" the Discriminator. Therefore the objective function will be

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

Noticed that in the objective function, the first part of the equation does not depend on the parameters of G, which consists of the objective function of the Generator: $log(1 - D(G(z)))$.

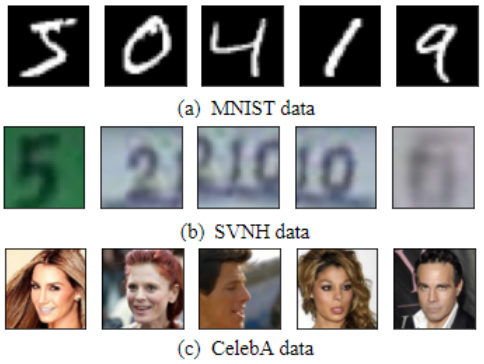
According to Algorithm 1 of the original paper, we first sample a batch of random noise vectors and a batch of vectors from the real data. We then use the above objective function of GAN in Eq. 1 to update the parameters of Discriminator by gradient ascent. Notice that when we update the Discriminator network, the parameters of the Generator remain fixed. After we finish updating the Discriminator network by certain iterations, we freeze parameters of Discriminator and then train the Generator. Based on the training process discussed above and code implementation in Tensorflow tutorial, we can implement our own GAN architecture on MNIST, SVNH and CelebA datasets.

1.3 Data Set Description

The first dataset we want to implement GAN algorithm on is typical MNIST data, which is a hand-written digits dataset. The dimension of single MNIST data is 28 * 28 *1 and we load our training dataset using *tf.keras.datasets.mnist.load_data()*. The training dataset includes 60000 images and the size of the minibatch we are feeding into each step is 32.

The second dataset we implement GAN on is following the project guideline - SVNH dataset, a collection of color images of house numbers. Although it is a different real-world dataset with more channels, the general algorithms stays the same with Algorithm 1. The image size is 32*32*3 and the mini-batch size of real images we feed into the Discriminator with the fake image is 64.

The third dataset we implement GAN on is CelebA dataset. CelebFaces Attributes Dataset (CelebA) is a large-scale colored face attributes dataset with 100K celebrity images. For better running time, we use the first 10K images as the training set. The image size is 64*64*3 and the mini-batch size we choose is also 64. We implement our model to this dataset because we want to analyze the performance of basic GAN for complex image processing and generation. Figure 2 shows sample training images of MNIST, SVHN and CelebA data.



(a) MNIST data

(b) SVNH data

(c) CelebA data

Figure 2: Sample images of different datasets

2 Methods

2.1 Strating Point

Our first step is to implement our custom DCGAN architecture on the MNIST data. Regarding the methods to train the GAN, we are following Algorithm 1 of the original paper *Generative Adversarial Nets*.

For the SVHN data, since the most common form of a ConvNet architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats it until it has a better result. Our starting model is:

$$INPUT -> [CONV -> RELU -> POOL] -> FC -> RELU -> FC \quad (2)$$

The starting model only contains one convolutional layer. The input volume is a 32 x 32 x 3 array of pixel values. The number of output filters in the convolution is 32, which means that the dimension of the output space is 32. Since the kernel size is 3*3, the output will have 3*3*3*32 = 864 parameters. We use 'same' padding so that the output has the same height/width dimension as the input. The layer did not use a bias vector, since the starting model is a relatively small network. Then we choose a RELU layer and follow with a pooling layer to reduce the spatial size. The pool layer has pool size (2, 2), which will take the max value over a 2x2 pooling window. After the pooling layer, the output has shape (16,16,32), then a dropout layer has applied. Since the output is multidimensional and we want to pass it onto the Dense layer, we flattened it and moved to Dense with 64 units. The last Dense layer after another dropout has 10 units and Softmax activation which converts a real vector to a vector of categorical probabilities. Then we train the model using batch size 32 with 5 epochs, and repeat this procedure as:

$$INPUT -> [CONV -> RELU -> POOL] * 2 -> FC -> RELU -> FC \quad (3)$$

Moreover, after several attempts, we see that the model with a more convolutional layer has less loss. From this perspective, we construct our own GAN model on SVHN data that include 6 convolutional layers with Leaky-Relu activation.

2.2 Architectures of the GAN Model

For MNIST data:

We first train the Discriminator network corresponding to the algorithm by compiling the TensorFlow model using binary cross-entropy loss function since we simply need Dscrimator to correctly identify the true image of MNIST and the fake image from Generator. Our first Discriminator simply includes regular Convnet to do the image classification task by classifying the real and fake images. It includes 5 Conv2D layers with Leaky-Relu activation and batch normalization layer on the hidden states, and we then Flatten the tensors and feed into a FFN with output unit of 1.

Then we train the Generator network by generating new batch of random noises and label the output fake image as 1. By freezing the gradients of Discriminator, the adversarial network allows the GAN to reach a zero-sum game while still letting the gradients to backpropagate from the last layer of

Discriminator to the first layer of Generator. Our first trial of Generator network includes the first FFN layer that maps random noise vectors into three dimensions with filter size of 128. Then we use two Transpose Convnet with Strides = 2 to double the size of the feature maps and use two more Convnet with Strides = 1 and Padding = 'Same' to reach the final dimension of 28*28. The filter size of the final Transpose Conv2D is 1. In the middle of each building block, we use Batch Normalization to expedite the convergence of the loss function and add some regularization effect. The activation of Transpose Convolution layer is Relu or Leaky-Relu and the kernel size we chose are 5*5 or 3*3 to compare the performance.

After comparing the performance of GAN between GAN with sigmoid activation function on Generator's last output layer and tanh activation on Generator's last output layer, we find that tanh activation gives a better performance regarding the qualities of fake images.

For SVNH data:

The algorithms stay the same. Our architecture of Generator consists of 1 FFN layer with 4*4*512 units, two Transpose Conv2D layers with kernel size 5*5, filter size with [256, 128, 3], strides = 2 and padding = "same" to double the size of the feature maps. The filter size of the final Transpose Conv2D layer is 3 which is the channel size of the SVHN dataset. The batch normalization is applied to normalize the hidden states of the intermediate feature maps. The architecture of the Discriminator consists of 3 Conv2D layers with Leaky-Relu activation and Batch Normalization layer on the intermediate states, which is then followed by Flatten layer as well as a FFN layer with sigmoid activation function. The training process of the GAN stays the same with MNIST implementation.

For CelebA data:

The algorithms stay the same. Comparing to the Generator for SVNH data, the architecture stays the same except one more Transpose Conv2D layer with kernel size 5*5, filter size with 64, in order to generate 64*64*3 images.

2.3 Implementations using WGAN Model

We built our own WGAN network on MNIST and SVNH datasets by following the differences between Regular GAN and not revising the codes extensively. The training process is based on the paper of *Wasserstein GAN (2017)*, which can be found at

`https://arxiv.org/pdf/1701.07875.pdf`

We first introduced *n_critic* hyperparameter to our WGAN network training process, which means that we are training n_critic times of mini-batches on Discriminator before training one mini-batch on Generator. Also, when training Discriminator, we applied gradient clipping to Discriminator between -0.01 and 0.01 to satisfy Lipschitz constraint. In WGAN, we are instead using a linear activation function in the output layer of Discriminator and Wasserstein loss function for Discriminator.

Accordingly, when training the discriminator network, we label the real image as 1 and fake image as -1. We train 1 batch of real data first, then 1 batch of fake images instead of 1 combined batch of real and fake images. This tweak, according to sources online, prevents the gradient from vanishing due to opposite signs of real and fake data labels (i.e. +1 and -1) and small magnitude of weights due to clipping.

The overall architectures of WGAN stay the same with our GAN network on MNIST and SVNH datasets.

3 Results

For GAN:

Figure 3a shows our result of generating fake images on MNIST data after 50 epochs. For each iteration, we transform 64 random noise vectors into fake images and randomly sample 64 mini-batch samples from MNIST, which are altogether fed into the Discriminator network. Then we train the adversarial network by freezing the weights of Discriminator. Moreover, Figure 5 shows the plot of Generator and Discriminator losses on MNIST data.

(a) MINST data (b) SVNH data (c) CelebA data

Figure 3: Results of "fake" images of different datasets using GAN

(a) MINST data (b) SVNH data

Figure 4: Results of "fake" images of different datasets using WGAN

Figure 3b shows the result of generating fake images after the 120th epoch using SVHN data. The total epoch number is 120, and for each epoch, we iterate every image in the training dataset and generate random seed as input. There are 64 noises we generated in the training loop and transferred into fake images. Then the loss is calculated for each of these models, and update the generator and discriminator using gradients.

Figure 3c shows fake human faces images generated by our model using CelebA data. The total epoch number is 200. There are 100 noises we generated in the training loop and transferred into fake images. The generation procedure stays the same as for SVHN data. Noticed that although the outline of the face can be seen from the picture, it is very blurry and not close to the real training data.

For WGAN:

Figure 4a and 4b shows the WGAN results on MNIST and SVHN data after training 50 epochs. The training process is the same as what we state in the method section. The training time of each epoch of WGAN is much longer than that of GAN, however, we could not see any clear improvement in terms of image quality.

4 Discussion

For now, our WGAN results are not promising enough to beat our GAN implementation. After the semester ends, we may need to delve deeper into the architecture choice of WGAN which might lead to better convergence of the zero-sum game. Training WGAN for longer time might be a good choice since based on the WGAN paper, the learning rate for the RMSprop optimizer is lower than the optimizer for GAN network.

For CelebA data, we could only draw an outline by implementing the basic DCGAN from the original paper (2014), but the picture is not clear even though we run a significant amount of epoch. Therefore, for complex images such as human faces, more advanced evolutions or algorithms are needed, for example, NVIDIA's Hyperrealistic Face Generator. Moreover, there are many impressive applications of GAN except the MNIST and SVNH such as CycleGAN which translates photographs into artistics style photos. Such evolutions of the basic GAN architecture to various advanced applications are fascinating and worth for future exploration.

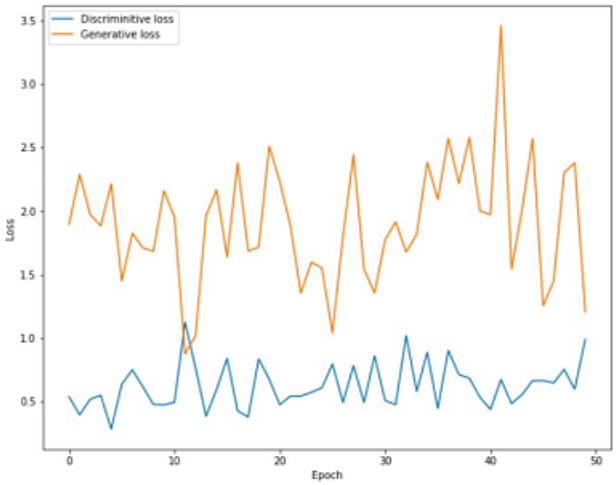


Figure 5: Loss plot

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. (2014). Generative Adversarial Nets. In NIPS'2014 .

[2] Deep Convolutional Generative Adversarial Network: TensorFlow Core. (n.d.). Retrieved December 13, 2020, from https://www.tensorflow.org/tutorials/generative/dcgan

[3] The Street View House Numbers (SVHN) Dataset: Retrieved December 19, 2020, from http://ufldl.stanford.edu/housenumbers/

[4] Large-scale CelebFaces Attributes (CelebA) Dataset: Retrieved December 19, 2020, from http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

[5] Martin Arjovsky, Soumith Chintala, and L´eon Bottou. (2017). Wasserstein GAN.