ECE 544 PATTERN RECOGNITION COURSE PROJECT
GENERATIVE ADVERSARIAL NETWORKS

Gizem Tabak
(tabak2)
December 7, 2016

1. BACKGROUND

Generative adversarial networks (GANs) are first proposed by Goodfellow et al. in 2014 [1]. Unlike Their working principle is often explained as a forgery-police game. They consist of two networks, generator and discriminator networks. Discriminator acts as the police that tries to identify if a bill is fake, i.e. generated by the forger, or real. Generator is the forger that generates fake bills and tries to trick the police. At each step, forger puts its fake bills into a stack with real bills, and the police tries to identify fake and real bills correctly.

In a more formal setting, both generator (G) and discriminator (D) are multi-layer neural networks. G is fed with random samples from a prior noise distribution $p_n(z)$, and then either the output of G or samples from an unknown data distribution $p_d(x)$ is fed to D. Both networks are simultaneously trained so that D will maximize the probability of assigning correct labels (i.e. real data or coming from G) to its input, and G will minimize the probability that D identifying output of G correctly.

In their paper, Goodfellow et al. [1] proposed an algorithm (Fig. 1) that utilizes standard gradient-based learning in order to train both networks properly. They provide a brief verbal proof of the convergence of this algorithm under some conditions, and applied this algorithm on MNIST, CIFAR-10 and Toronto Face Databases. They compared GANs with other generative methods, and their claim is although GANs do not necessarily generate better samples than others, they still generate competitive results with its own advantages and disadvantages. One disadvantage of GANs is G and D should be trained in a balance so that D is good enough to improve G with its predictions. Another disadvantage, and this is mostly common in other generative methods as well, is the evaluation of their generated distribution is hard since it can only be an approximation. On the other hand, networks are updated only with backpropagatin, so no inference and Markov chain-like complex methods is needed. Besides, they can generate sharper images compared to other methods that utilize Markov chains that mix different models resulting in blurry images.

In this report, first I will introduce their objective function and find global optimal discriminator for a given generator more in detail in Section 2. Then, in Section 3, I will demonstrate examples of GANs applied on (1) a Gaussian distribution, (2) the MNIST dataset with vector inputs and (3) the MNIST dataset with CNN.

2. METHOD

Notations in this section are as below:

- $p_n(z)$: Noise distribution on the input noise variables z
- $p_d(x)$: Data distribution where $x \in \mathcal{X}$
- $p_g(x)$: Generator distribution
- $G(z; \theta_g)$: Generator (neural network with parameters θ_g) that maps input noise samples z to $\mathcal{X}$
- $D(\hat{x}; \theta_d)$: Discriminator (neural network with parameters θ_d) that represents the probability of $\hat{x}$ coming from $p_d(x)$.
- $\hat{x} \leftarrow x$: $\hat{x}$ is coming from data, $p_d(x)$
- $\hat{x} \leftarrow G(z)$: $\hat{x}$ is coming from the generator

According to the problem defined in Section 1, the discriminator wants to maximize the (log) probability of assigning correct labels. Then, it should maximize D when its input comes from the data, and minimize when it comes from the generator. Hence, D should **maximize**

$$\mathbb{E}[\log(D(\hat{x} \leftarrow x))] + \mathbb{E}[\log(1 - D(\hat{x} \leftarrow G(z)))] \quad (1)$$

Simultaneously, G wants to maximize its probability of fooling D. Meaning, it should **maximize**

$$\mathbb{E}[\log(D(\hat{x} \leftarrow G(z)))] \quad (2)$$

In order to combine Eq. 1 and Eq. 2 into one single objective function, Eq. 2 can be rewritten as **minimizing**

$$\mathbb{E}[\log(1 - D(\hat{x} \leftarrow G(z)))] \quad (3)$$

and the objective function will be

$$\min_G \max_D \mathbb{E}_x[\log D(x)] + \mathbb{E}_z[\log(1 - D(G(z)))] \quad (4)$$

In this way, the problem can be viewed in a minmax problem perspective in theory. However, in the paper it is stated that the gradients from the objective function in (4) might be easier to saturate early in the training. In that case, it is better to use (2) to train G in practice.

For a given generator G, optimal discriminator is the one that maximizes (4). Then,

$$\begin{aligned}\frac{\partial V(G,D)}{\partial D} &= \frac{\partial}{\partial D}\mathbb{E}_x[\log D(x)] + \mathbb{E}_z[\log(1-D(G(z)))] \\ &= \frac{\partial}{\partial D}[\int_x p_d(x)\log(D(x))dx \\ &+ \int_z p_n(z)\log(1-D(G(z)))dz] \end{aligned} \tag{5}$$

Since $x = G(z)$ in the second part of the summation, if we use change of random variables

$$x = G(z) \rightarrow p_g(x) = \frac{p_n(z)}{G'(z)} \text{ and } dz = \frac{dx}{G'(z)} \tag{6}$$

and substituting (6) into the second part of Eq. 5,

$$\begin{aligned}\frac{\partial V(G,D)}{\partial D} &= \frac{\partial}{\partial D}[\int_x p_d(x)\log(D(x))dx \\ + \int_x p_g(x)\log(1-D(x))dx] \\ = \frac{\partial}{\partial D}\int_x p_d(x)\log(D(x)) + p_g(x)\log(1-D(x))dx \end{aligned} \tag{7}$$

Assuming the integration and derivation are interchangeable,

$$\frac{\partial V(G,D)}{\partial D} = \int_x \frac{p_d(x)}{D(x)} - \frac{p_g(x)}{1-D(x)}dx = 0 \tag{8}$$

and if the condition below holds,

$$\frac{p_d(x)}{D^*(x)} - \frac{p_g(x)}{1-D^*(x)} = 0 \rightarrow D^*(x) = \frac{p_d(x)}{p_g(x)+p_d(x)} \tag{9}$$

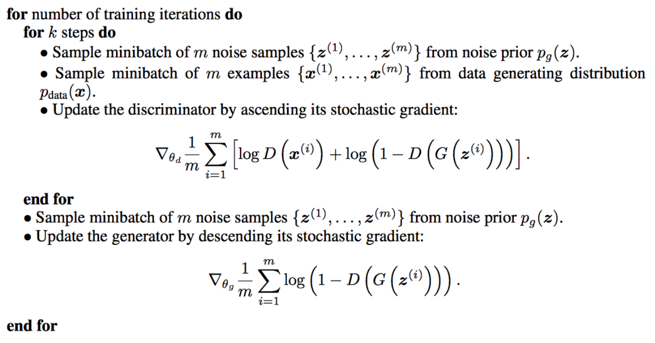
Eq. 8 holds as well. Hence,

$$D^*(x) = \frac{p_d(x)}{p_g(x)+p_d(x)} \tag{10}$$

is the optimal discriminator given generator G. This means that decision boundary at fully trained successful generator network should be at $\frac{1}{2}$. This intuitively makes sense since D outputing 0.5 means it cannot distinguish between real and generated data.

3. EXPERIMENTS

I performed three sets of experiments; (1) a toy example with a Gaussian data distribution, (2) MNIST dataset images represented as vectors and used as the samples drawn from the data distribution, and (3) again MNIST dataset, but this time with a Deep Convolutional Generative Adversarial Networks (DC-GAN) [2]. For the last set, since it involved advanced architectures beyond the scope of this project, such as deep deconvolutional networks, I used their open-sourced code and I used the last set only for the comparison purposes with the second one.



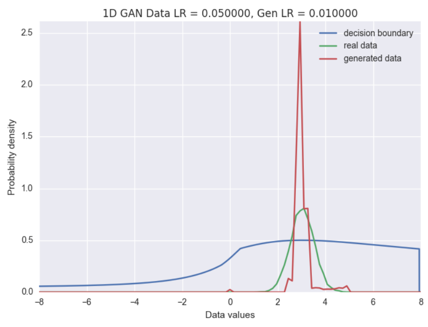
```
for number of training iterations do
  for k steps do
    • Sample minibatch of $m$ noise samples $\{z^{(1)},\dots,z^{(m)}\}$ from noise prior $p_g(z)$.
    • Sample minibatch of $m$ examples $\{x^{(1)},\dots,x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
    • Update the discriminator by ascending its stochastic gradient:
        $\nabla_{\theta_d}\frac{1}{m}\sum_{i=1}^{m}\left[\log D\left(x^{(i)}\right) + \log\left(1-D\left(G\left(z^{(i)}\right)\right)\right)\right].$
  end for
  • Sample minibatch of $m$ noise samples $\{z^{(1)},\dots,z^{(m)}\}$ from noise prior $p_g(z)$.
  • Update the generator by descending its stochastic gradient:
      $\nabla_{\theta_g}\frac{1}{m}\sum_{i=1}^{m}\log\left(1-D\left(G\left(z^{(i)}\right)\right)\right).$
end for
```

Fig. 1. DAN algorithm [1]

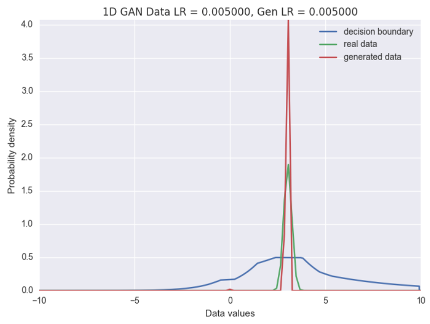
3.1. Gaussian Data

The algorithm for GAN (Fig. 1) is straightforward once determining which function to optimize. Once deciding on the objective function, i.e. loss, all we need is to apply backpropagation with the help of an optimizer. First, I worked on a toy example using one dimensional Gaussian distribution as the data distribution. I tried to generate Figure 1 in the paper [1]. I used 2-layer network for generator and 3-layer network for discriminator, with 11 hidden units at each layer for both G and D. I chose a more advanced network for discriminator in order to prevent discriminator not catching up with the generator and blowing up the system. I used uniform distributions $\mathcal{U}(-8,8)$ and $\mathcal{U}(-10,10)$ for noise distribution and generated 120 samples, each of which are 1-d numbers (z). I tried a few different real data distributions ($p_d(x) \sim \mathcal{N}(-2,0.5)$, $p_d(x) \sim \mathcal{N}(3,0.5)$, $p_d(x) \sim \mathcal{N}(3,0.2)$) and ran each of them several times. I trained the networks for 4000 steps and I had to tune learning rate, weight initializations and noise range of the uniform distribution $p_n(z)$ for different data distributions in order to get a plausible result. Yet, even though I used exactly the same parameters, results were different and somewhat less/more plausible at each run (Fig. 2 (b) and (c)).

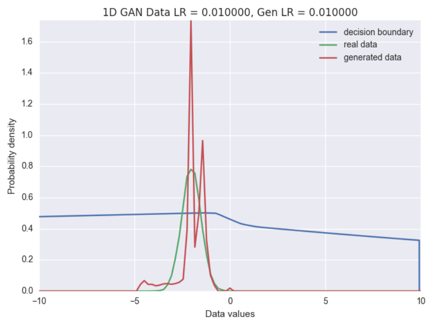
Some of the results are displayed in Fig. 2. As expected from Proposition 1, when the system is good enough, decision boundaries are very close/equal to 0.5. However, in none of the trainings, generated distributions were smooth enough to be indistinguishable from the data distribution. Apparently, this is one problem of generative adversarial networks, and it is addressed in [3]. I also noticed that when the variance of the real data Gaussian distribution is high, it is harder to tune the system to generate plausible distributions and it fits narrower distributions better (Fig. 2 (b)). These two findings together suggest that generative network tend to produce low-variance samples and might need some regularization tuning in its optimization. Additional results, distribution and training loss plots and training animations displaying distribution evolutions can be found in supplementary materials.



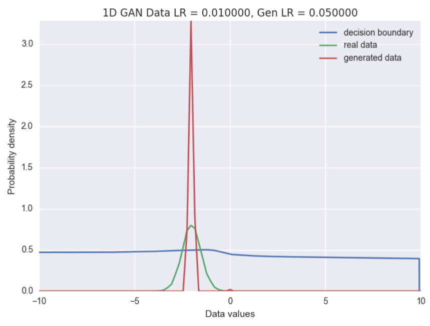
(a) $p_n(z) \sim \mathcal{U}(-8, 8)$, $p_d(x) \sim \mathcal{N}(3, 0.5)$



(b) $p_n(z) \sim \mathcal{U}(-10, 10)$, $p_d(x) \sim \mathcal{N}(3, 0.2)$

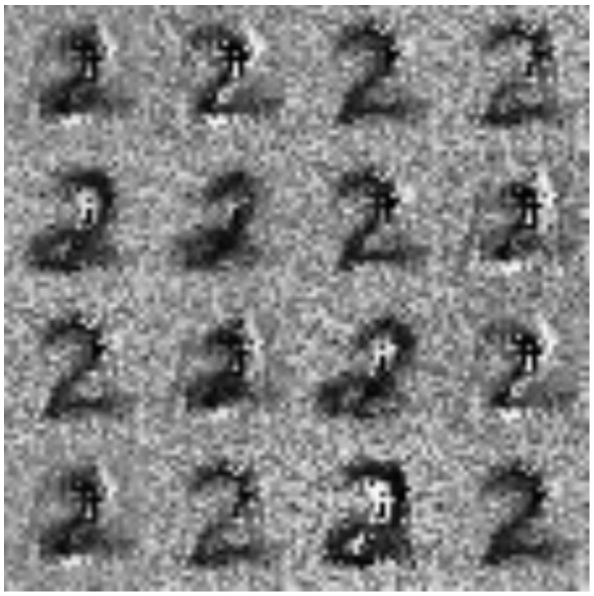


(c) $p_n(z) \sim \mathcal{U}(-10, 10)$, $p_d(x) \sim \mathcal{N}(-2, 0.5)$

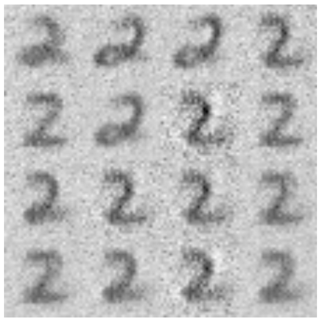


(d) $p_n(z) \sim \mathcal{U}(-10, 10)$, $p_d(x) \sim \mathcal{N}(-2, 0.5)$

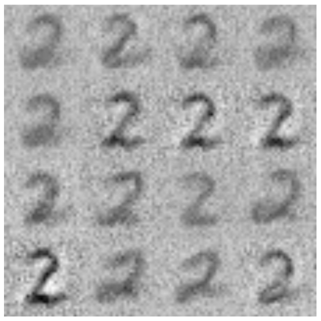
Fig. 2. Generative (red) and data (green) distributions with decision boundary (blue)



(a) 3 layer D, $tanh$ output



(b) 3 layer D, linear output



(c) 4 layer D, linear output

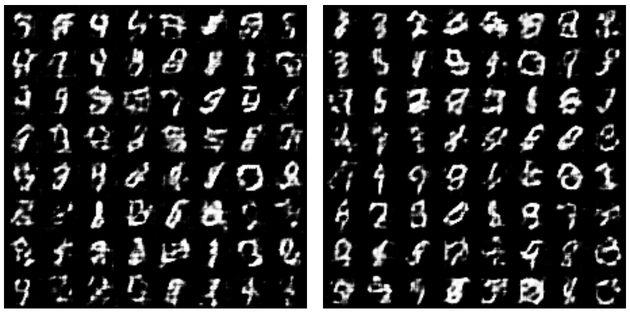
Fig. 3. GAN-generated results for the digit 2

3.2. MNIST Dataset

In this section, I used MNIST dataset [4] as real data distribution. So this time, input and output of the generator network were 784-d vectors. Again, I used 2-layer generator and 3-layer discriminator network at first, with 50 hidden nodes at each layer. Then, I changed the structure to 2-layer generator and 4-layer discriminator network. I initialized the noise samples the same way in the previous section with uniform distibution, but with noise range 1 this time. I ran with batch size 50 and 100 epochs.

Compared to previous section, tuning parameters and training networks were more easier. For most of the training parameters, generated data seemed plausible so that they were actually representing the input label (number to be generated).

In Fig. 3, 16 different randomly selected results generated by G is shown for digit 2. Results of other digits and loss functions can also be found in supplementary materials. In



(a) Trained with 2 epoch (b) Trained with 2 epochs

Fig. 4. Random examples of generated numbers with DC-GAN

Fig. 3 (b), it shows data generated by the 2-layer generative network without nonlinearity on its output layer. Although I could not quite get the intuition behind it, this was suggested in some blogs [1]. The output with $tanh$ has more contrast compared to the output of the network with linear final layer since $tanh$ tend to saturate larger (or smaller) values to 1 (or -1) and acts like a smoothed threshold, comparing Fig. 3 (a) and (b). Also, when I try to increase number of hidden layers in D from 3 to 4, its output were blurry compared to other ones (Fig. 3 (c) and (b)). This might be because it is a deeper network, it might take longer time for the system to train properly and this can be an early result during the process.

Compared to previous section with Gaussian data, the network is acting more consistently (in terms of plausibly mimicing the data distribution). Although they are not perfect, all 16 images in Fig. 3 looks like digit 2, and it is the same for other digits whose results (with $tanh$ output and 3-layer discriminator).

3.3. MNIST Data with DC-GAN

After implementing GAN on vectorized MNIST data, immediate following application I could think of was utilizing CNN instead of converting the image into 784-d vector. However, apparently, this was not an easy task because it requires implementing a ”deconvolutional” network to generate a 2-d image with G from the noise vector z. Since this was out of the scope of this project, in this section, I used the implementation of Salimans et al. named Deep Convolutional Generative Adversarial Networks (DC-GAN) in order to compare results with the previous part. The code of their implementation can be found in Github repository[2].

First, I trained the network on MNIST for 1 epoch. This took 55 mins as training with vector-type inputs took only 3 sec per epoch. This is because their code is designed for advanced tasks such as face generation and completion and hence utilizes an advanced convulutional and deconvolutional neural network. Results with 1 epoch can be seen in Fig. 4 (a). Since they were not plausible, I trained for one more epoch, and its results is in Fig. 4 (b). Although they are better than the one trained with 1 epoch and digits are more identifiable, they are still not as plausible as the ones generated in Section 3.2. It is for sure that results get better if I had the computational power to train it for more epochs. However, more than 1000 times computational time seemed like an overkill for this task with MNIST dataset.

4. CONCLUSION AND FUTURE WORK

Generative adversarial networks are fascinating models with their ability of mimicking a data distribution (and hence generating data) without need of complex structures like parametric models or Markov chains. They have their own drawbacks of keeping generative and discriminative network in balance during training, and hardship of quantitatively measuring their success assessing their performance. However, they generate more real-like images compared to other models.

In this report, first I applied GANs to a Gaussian distribution. It was hard to find right parameters and even then, results were not that plausible, especially when the real data had large standard deviation since generator network was producing ”narrow” distributions compared to the real data. This is a known problem of GANs and attempts to fix this problem, as well as some others, is explained in the work of Salimans et al. [3].

When I apply GANs on MNIST data, I obtained more plausible results, since almost all of the generated figures were obvious to identify as the intended digit. The results presented here are not obtained with networks trained for very long time due to computational power restrictions. Because of the lack of a numerical comparison of results, I was not able to fine-tune the parameters. As future work, increasing training epochs and fine-tuning would give better results.

Comparing inputs of vectorized images and DC-GAN model where the images are fed to a convolutional network, although the training time was ¿1000 times longer, results were worse and it was hard to identify generated digits. However, this is not a fair comparison since the advanced DC-GAN network did not have a chance to be fully trained with such less epochs and I could not afford to train them longer.

As future work in the literature of GANs in general, they need more convenient ways to be trained since it might be hard to find correct parameters for balanced networks and successful training. Besides, their theory lacks proper performance analysis. With the help of a proper analysis, these balance issues might be easier to resolve.

[1]http://blog.aylien.com/introduction-generative-adversarial-networks-code-tensorflow/

[2]https://github.com/carpedm20/DCGAN-tensorflow

5. CODE

I wrote the most of the code for Section 3.1 (`gan_gaussian.py`) and Section 3.2 (`gan_mnist.py`) myself in Tensorflow. I used animation generation and distribution plotting functions (`_samples, _plot_distribution, _save_animation`) from AYLIEN Github repository[3]. I defined 2 classes, one for neural networks and one for GAN. Parameters of both classes is arranged in `main` function, except data distribution which is defined in the initializer of GAN. Generative and distributive network structures are defined in NeuralNet class, with `initialize_variables` and `mlp` functions. GAN networks are generated using this class, and GAN model is constructed with `create_model` function. Losses and optimizers are defined in this function. Defining different losses and optimizers is one of the core ideas of GAN and it enables training of generator and discriminator together at each iteration. Then, model is trained with `train` function. At each iteration, batch of noise and real data is generated (or obtained from the dataset) and fed into the generated model in order to optimize losses for generator and discriminator at each iteration and update their weights accordingly. At the end, for plotting purposes, above mentioned functions are used.

The code is mostly the same for Section 3.2. I only needed to change the input structure so that instead of scalars, networks were able to receive and generate vectors, and also instead of generating Gaussian random data samples as the real data distribution, real data samples are obtained from MNIST dataset. In this part, in order to plot generated images in one tiled figure, I used `tile` function in this repository[4].

For Section 4, I did not write any code for DC-GAN except fixing a minor bug in the DC-GAN Github repository[5] for reasons explained in Section 4.

6. REFERENCES

[1] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, 2014, pp. 2672–2680.

[2] Alec Radford, Luke Metz, and Soumith Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *CoRR*, vol. abs/1511.06434, 2015.

[3] Tim Salimans, Ian J. Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen, "Improved techniques for training gans," *CoRR*, vol. abs/1606.03498, 2016.

[4] Yann LeCun, Corinna Cortes, and Christopher JC Burges, "The MNIST database of handwritten digits," 1998.

[3]https://github.com/AYLIEN/gan-intro

[4]https://github.com/PhoenixDai/improved-gan/blob/master/mnist_svhn_cifar10/plotting.py

[5]https://github.com/carpedm20/DCGAN-tensorflow