Part 2 - Implementation

1. Background and Data Selection

In part 1, I have guided you to tour through the fundamental ideas and principles of the new and promising generative model in deep learning - Generative Adversarial Networks (GANs). In this section, we are going to implement this new approach and apply it on a data set named "notMNIST" to create new fronts. For ages, font designers have been trying to create new fonts of all kinds of styles for aesthetic purposes or to serve different applications. Now using GAN, we can let our deep learning models to help create new fonts with little effort!

Of course, the GAN model cannot really create something out of empty. We must provide some training data to our model for it to "learn" what to generate. Luckily, there is a dataset of weird fonts created by Yaroslav Bulatov. The set of images of characters were taken from publicly available fonts and were generated by extracting glyphs from them. They were designed to look similar to the classic MNIST dataset, but more distorted and with more variations. As can be seen from the examples (Figure 1), there are quite a few empty images, black-out images, or ones with unrecognizable characters. The data is a lot less 'clean' than the MNIST. One would expect this to be a harder task.

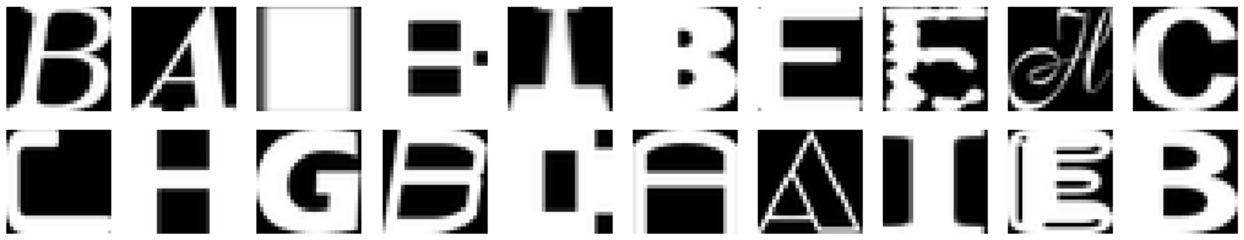
Figure 1: Sample images from notMNIST

In this blog, we will challenge our GAN model for generating characters/fonts from the notMNIST dataset. Sample codes are provided in the attached Jupyter Notebook. The codes were inspired by the one introduced by Oshea's for the MNIST dataset, but with structure modifications, fine tuning and parameter adjustments.

To start with, we want to download the dataset. There are two datasets available for downloading: one large, and one small (of reduced size). Both consist of characters rendered in a variety of fonts ranging from 'A' through 'J' (10 classes in total, including both uppercase and lowercase letters). The large set is a bit too big for our demo case, and is slow to train, so we are going to use the smaller set, which is about 8.4 MB in size and contains 18k images. It would be of a reasonable size for training.

2. Build Up the GAN Model

Now the data has been downloaded, we want to convert each image into the dimension of (1, 28, 28) and normalize the values to be within the range of [0, 1]. Then, before diving into training the GAN models, we start with building two neural networks.

- **Generator (G)** – the generator takes random noise as input, and tries to generate a front sample from the input.
- **Discriminator (D)** – the discriminator takes a font sample, and tries to tell if it's from the original set or is fake.

In our implementation, we use Keras with Tensorflow as backend to build the GAN model. The generator (G) is constructed from convolutional network with interlooping convolution and upsampling layers with batch normalization. The discriminator (D) is constructed from similar convolutional network with LeakyReLU and drop out to improve the performance. The key idea of this method is to first train discriminator to be both smart, and then combine the generator and discriminator to form a complete generative adversarial network. Then we fix the discriminator weights, and train the weights of generative network to make random noisy inputs to approach the real example outputs of the adversarial half.

Now let's break down the different components:

- **Generator Module**

1. For this network, we first take random noise of size 128 as the inputs, and gradually map them to the dimension of (1, 28, 28) to match the real samples.

2. Then we include a fully connected layer followed by neural networks with convolutional layers with ReLU as activation followed by batch normalization except for the output layer. We use batch normalization to speed up training and improve overall accuracy. This is

because normalization can generally make data comparable across features. As the data flows through a deep network, the data might become too big/small as a result of adjusting the weights and parameters - typically named "internal covariate shift" by many papers. By normalizing the data in mini-batches, this problem is largely avoided.

3. At the output layer, we use sigmoid to 'push' grey values (in-between value in range of [0, 1]) towards either 0 or 1. We use ADAM as the optimizer for the generator as suggested by Radford et. al. 2015 and some other reference, and apply a learning rate of 0.0001 to the generator. We also use a binary cross-entropy loss function.

- **Discriminator Module**

  1. The discriminator takes the input image, runs through a network, and determines if it is fake or not.

  2. For this network, we build it with a set of convolutional layers with two fully connected layers at the output. We also included dropouts in between. As literature suggests, drop-outs are applied at test time for pix2pix GANs to add noise. They are useful since you can't just add a latent noise vector like in normal GANs. (induced noises can help you acheive better results, otherwise you might always get identical outputs.) We use a drop-out rate of 0.25 for the discriminator network. Again, we use ADAM as the optimizer, and apply the same learning rate of 0.0001.

  2. Finally, we use softmax as the activation function at the output layer which encodes [0,1] as fake and [1,0] as real.

- **GAN**

  1. Now we have both the G model and D model, we are ready to combine them into a complete Generative Adversarial Network (GAN). Before really stacking them together and run the GAN, we train the discriminator for one epoch. This approach has been taken in Oshea's model. In his method, random images were generated by the untrained generative model, then the noisy images were concatenated with an equal number of real images, and were labeled appropriately. Then the generative model was used for fitting until relatively stable loss values can be achieved. I found this trick very useful in training my model.

2. At combining both G and D models, We want to set trainable flag to False for elements in the adversarial half of the network such that we can freeze the weights of this part during back-propagation of the joint model as stated in the previous texts.

3. Results and Discussions

Some techniques and tricks used to accelerate our model and improve accuracy and performance have been discussed in Section 2. Overall, our GAN model is constructed without any complex structure. However, after tuning, as a matter of fact, this model runs quite fast, and it generates relatively decent results for an unclean dataset like notMNIST. The model has been run for 12,000 epochs. Both discriminative and generative losses have been plotted as a function of time in Figure 2. With my budget GeForce GTX GPU, it took about 23 min to run 12,000 epochs, and the plots converged quite fast. The model generated discernible characters (letters) after about 4,000 epochs. When the run proceeded to about 6,000 epochs (~12min later), most output characters could be identified. The quality of the generated images improved a little bit from 6,000-8,000 epochs. After that, I haven't seen much improvement between from 8,000 till the end of 12,000 epochs.

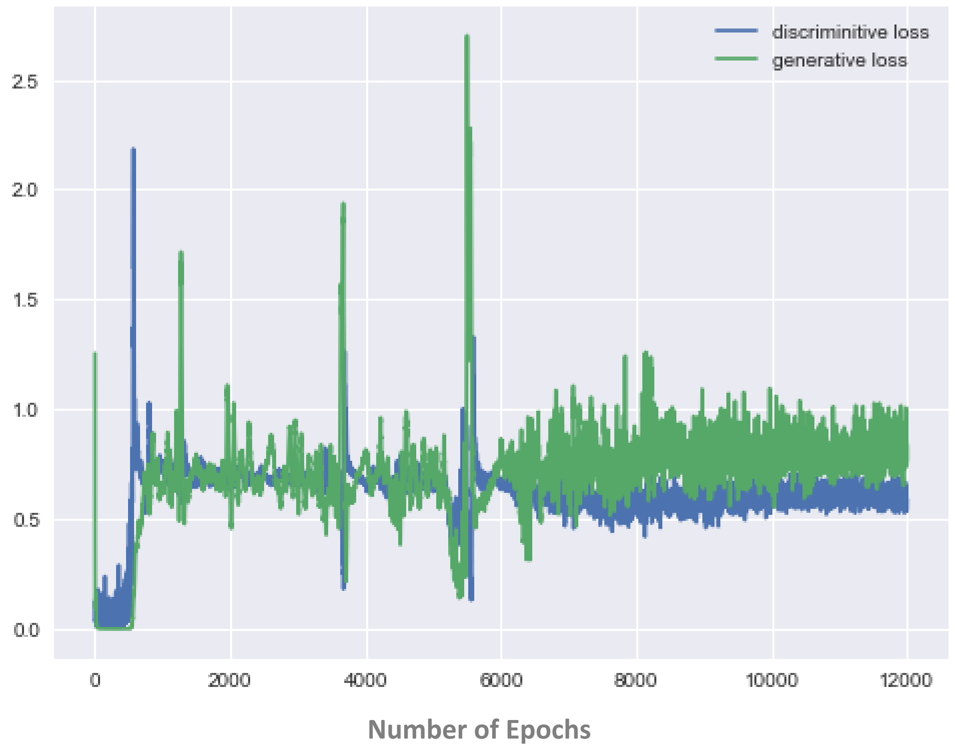

Figure 2. Discriminative and generative losses as a function of epoch numbers

Meanwhile, the adversarial behaviors of the generator and discriminator can also be seen from the oscillation of the two loss plots. The generator and discriminator act against each other, and reached dynamic balance after certain epochs (about 6-7,000).

Some sample output images are shown in Figure 3. Note that our original dataset for training only includes letters from ‘A’ to ‘J’.

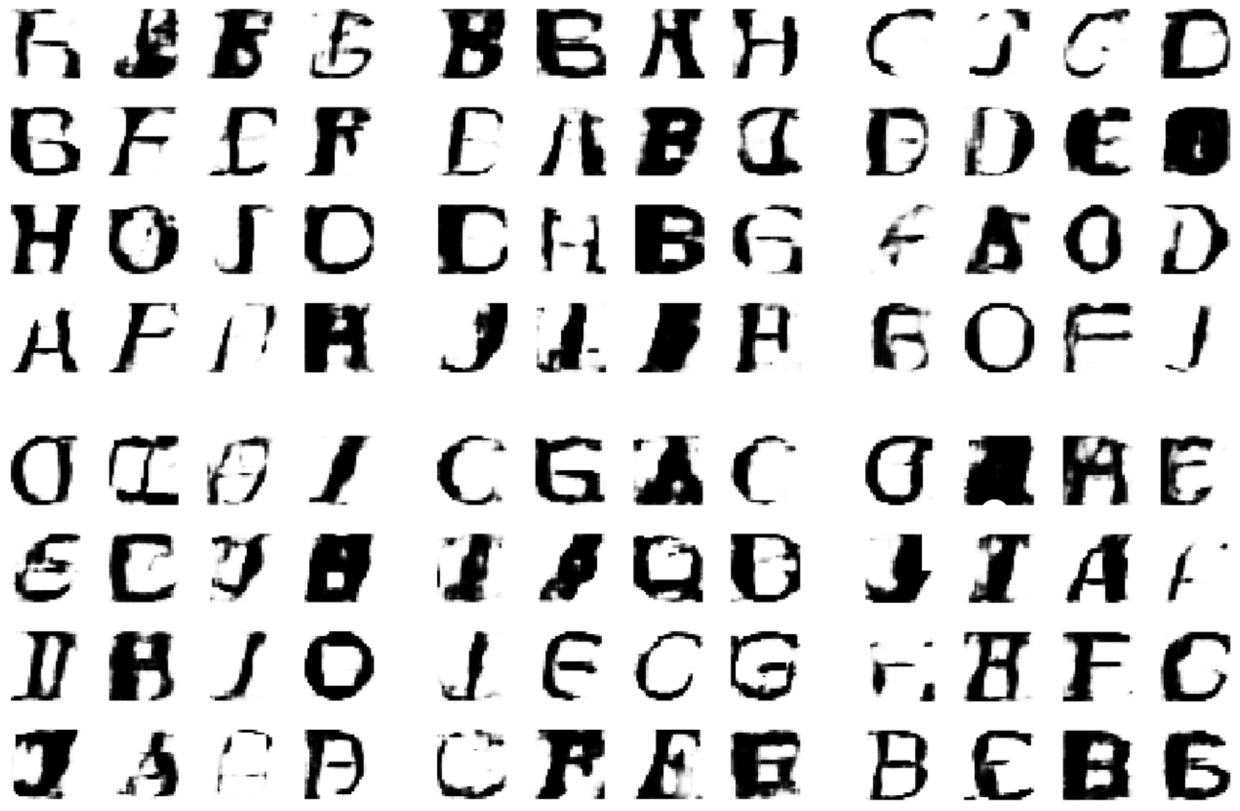

Figure 3: Generated letters from our GAN model

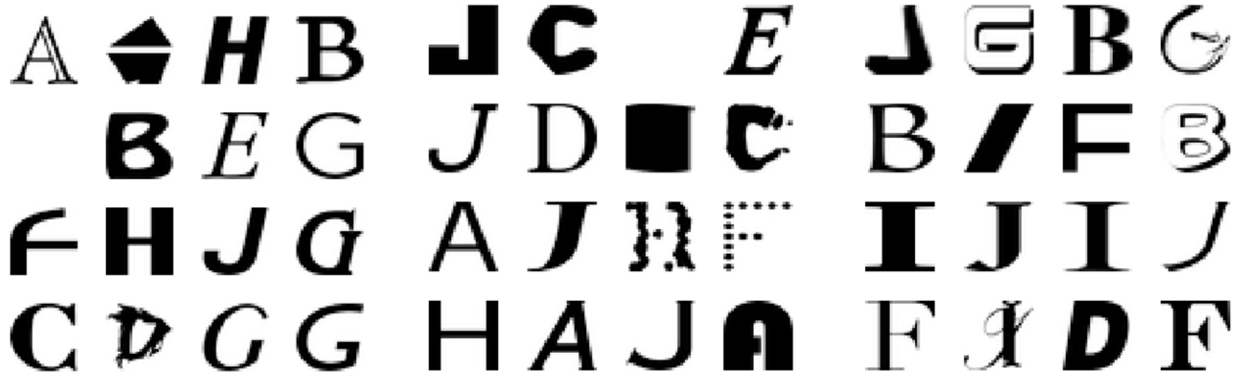

Figure 4. Letters/Fonts from the original nonMNIST dataset

Figure 4 displays some samples from the original notMNIST dataset for comparison. As you can see from the images, the GAN model is able to grasp the key features of the input image, and learn to produce new images based on the input. The original dataset contains letters from ‘A’ to ‘J’. Our model is able to reproduce all these letters. Another interesting fact you may notice is that the produced characters do look like they have the same fonts. In other words, the GAN model is capable of finding the common features of the input data, and throw away a lot of variations and details. This may not be a useful feature in our case which generates fonts from a whole bunch of random inputs. But think about it, if you want to combine two classes of inputs/features to generate a hybrid feature, this could be helpful. In other cases, if you feed the model with inputs from a single source, say, a lot of handwritings or paintings from a single person, the model could be able to learn the pattern of this person’s handwritings, and generate writing samples. This could be used in some cases for aesthetic purposes, or in some other cases to recognize a person’s writing/painting style, and thus used to identify if a new sample is the product of the same person. Also, if a GAN model is refined enough, its capability of ‘finding the common features among large noisy inputs’ may have an application of helping us identify the common information hidden in a sea of random and noisy data. There are still a lot of hidden gems to be explored in GANs, both in theory and in application. Hope this blog post could help you get start with a simple but interesting implementation of CNN-GAN model. Of course, this is a whole lot of room for improvement in our model. Feel free to try it out yourself by playing around with WGAN, ACGAN, LS-GAN, etc., and see what differences/improvements you can make. Hope you have fun playing with GANs!

Reference

[1] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *arXiv preprint arXiv:1502.03167* (2015).

[2] He, Kaiming, et al. "Identity mappings in deep residual networks." *European Conference on Computer Vision*. Springer International Publishing, 2016.

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

[4] Isola, Phillip, et al. "Image-to-image translation with conditional adversarial networks." *arXiv preprint arXiv:1611.07004* (2016).

[5] Ledig, Christian, et al. "Photo-realistic single image super-resolution using a generative adversarial network." *arXiv preprint arXiv:1609.04802* (2016).

Other Resources and References

How to Train a GAN? Tips and tricks to make GANs work:https://github.com/soumith/ganhacks

KerasGAN Implementation: https://github.com/osh/KerasGAN

Data Source

notMNIST (http://yaroslavvb.blogspot.ru/2011/09/notmnist-dataset.html)

Direct link: http://yaroslavvb.com/upload/notMNIST/