Forecasting Stock Prices with Generative Adversarial Networks and Reinforcement Learning

The Analytics Club, Center for Innovation, IIT Madras

April 6, 2020

Task report by **Nishant Prabhu**

Contents

1 Task Overview

We were asked to implement and train a Deep Convolutional Generative Adversarial Network (DCGAN) to generate new samples of MNIST handwritten digits. MNIST handwritten digits images were taken from the Keras datasets repository and reshaped to standard $(28 \times 28 \times 1)$ grayscale form. In the sections below, the overall working of DCGAN and the implementation details have been described.

2 Working Principle

In this section, the dataflow, construction and caveats related to the implementation of DCGAN have been described. The system as a whole behaves very stochastically, and the desired results have been obtained after multiple runs of the system.

2.1 GAN Overview

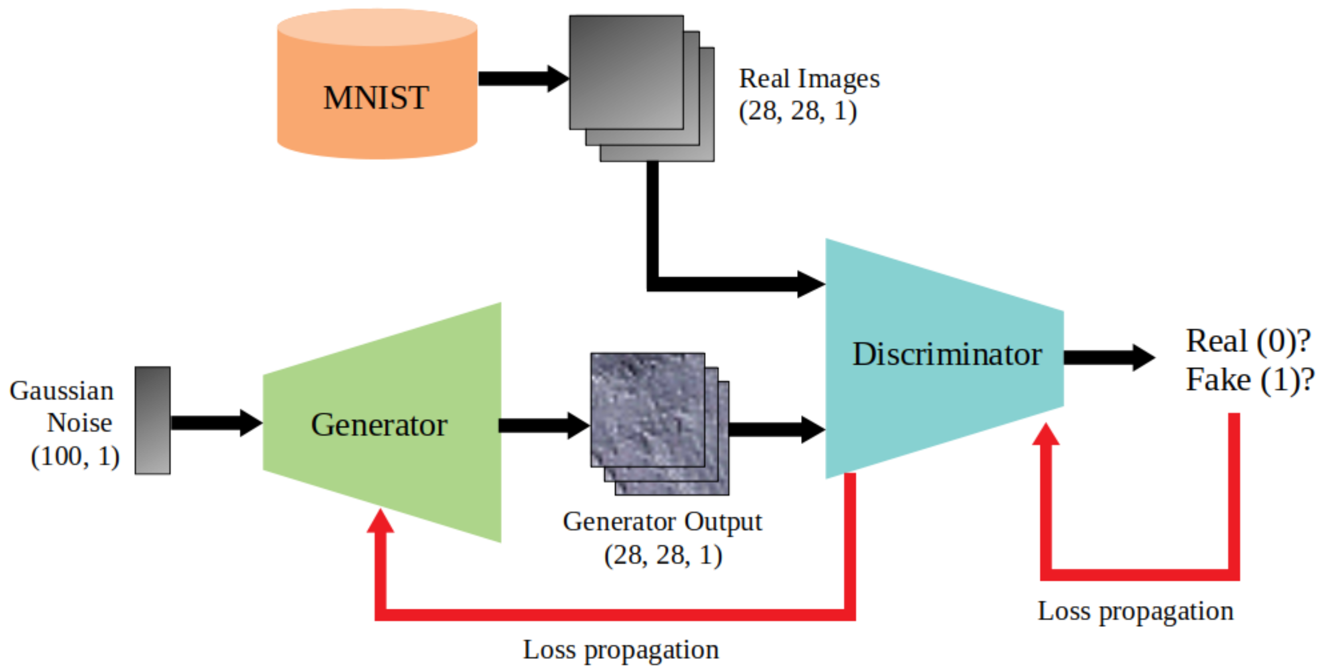


Figure 1: Simplified flowchart of DCGAN

DCGAN consists of two neural networks playing a minimax game. The first one, which we will call a **Discriminator** (**D**), is a classifier which tells whether an image given to it is fake (class label 1) or real (class label 0). The discriminator is fed with a concatenation of two sets of images (each of dimensions $28 \times 28 \times 1$): one containing `int(batch_size / 2)` number of images uniformly randomly sampled from the MNIST dataset of handwritten digits, and the other containing `int(batch_size / 2)` number of outputs from the second network, called the **Generator** (**G**).

The generator takes in an array of random numbers sampled from a "latent" distribution (standard Gaussian in our case) and outputs a bitmap of the same size as that of an MNIST digit image. The goal of this network pair is to modify the parameters of **G** such that its outputs are indistinguishable from real MNIST bitmaps. This measure of "closeness" is indirectly quanitified through classification error of **D**.

Discriminator **D** is pretrained to classify real and fake images prior to training DCGAN. Since **G** is initially very bad at producing real images, **D** trains very quickly (within 2 epochs). The GAN model is initialized by combining **G** and **D** in a `Sequential` model. Now, the following loop is executed for specified number of epochs:

1. A batch of data (`int(batch_size / 2)` real images and `int(batch_size / 2)` fake images generated by **G**, along with class labels) are generated and **D** is trained on the them.
2. All layers of **D** are rendered untrainable.
3. Another batch of `batch_size` number of fake images from **G** with inverted labels (all with label 0) are passed through the GAN. This is done so that pretrained **D** classifies all as fake (label 1), resulting in a large crossentropy loss (and thus large gradient). However we do not want **D** to update its weights to fit to these fake labels, so we render it untrainable beforehand. The loss is propagated through the GAN and only **G**'s weights get updated.
4. Every 10 epochs, the model configuration is saved.

Esentially, **G** lies to **D** with fake images calling them real, and based on the feedback from **D**, tunes it's own parameters to make its fake images look more and more real. These updates are very unstable in general, so training the model for long doesn't ensure better generator performance. The loss function which the network optimizes is given below.

$$L = \min_{\theta_g} \max_{\theta_d} \left\{ \log D_{\theta_d}(x) + \log D_{\theta_d}\left(G_{\theta_g}(z)\right) \right\}$$

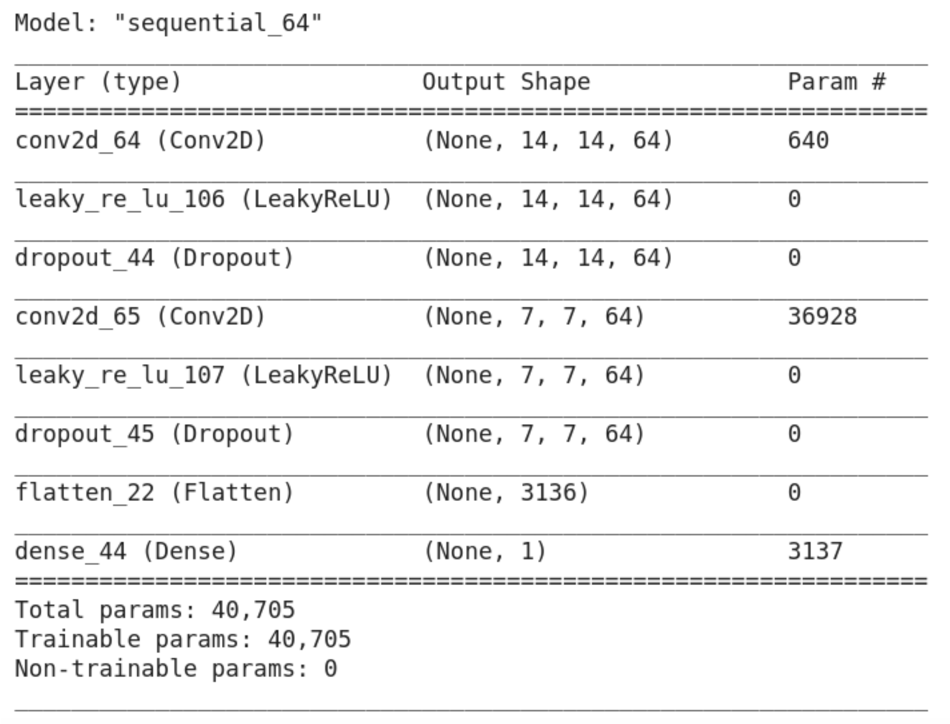
where $\mathbf{x}$ is the real sample and $\mathbf{z}$ is the noise drawn from the latent distribution. L is seen to remain more or less constant in a stable training process.

2.2 Discriminator Architecture

D is a convolutional classifier network which takes in inputs with shape (28, 28, 1) and outputs a class label, 0 or 1, determined by sigmoid activation. The configuration of the discriminator used for this experiment is shown in **Figure 2**.

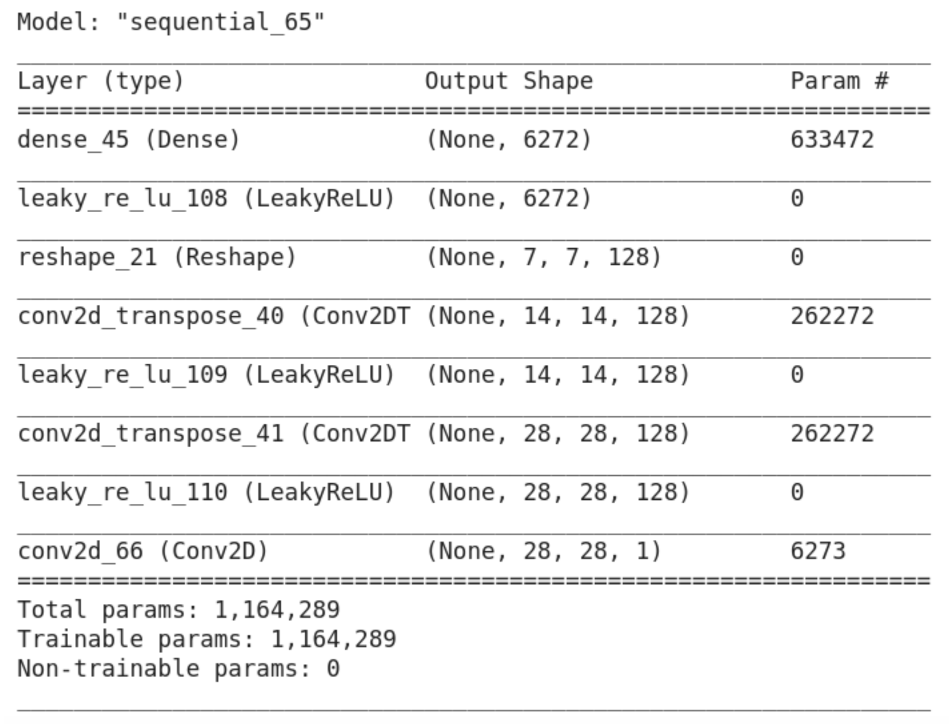
2.3 Generator Architecture

G takes in an array random samples from a latent distribution (standard Gaussian in our case) and upscales it into a bitmap of shape (28, 28, 1). While there are several upscaling methods, the model in consideration uses **Transpose Convolutional** layers (also known as "deconvolution" layers) with strides of 2 to double the dimensions of the bitmap at each layer. The configuration of the generator used for this experiment is shown in **Figure 3**.



```
Model: "sequential_64"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_64 (Conv2D)           (None, 14, 14, 64)        640
_________________________________________________________________
leaky_re_lu_106 (LeakyReLU)  (None, 14, 14, 64)        0
_________________________________________________________________
dropout_44 (Dropout)         (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_65 (Conv2D)           (None, 7, 7, 64)          36928
_________________________________________________________________
leaky_re_lu_107 (LeakyReLU)  (None, 7, 7, 64)          0
_________________________________________________________________
dropout_45 (Dropout)         (None, 7, 7, 64)          0
_________________________________________________________________
flatten_22 (Flatten)         (None, 3136)              0
_________________________________________________________________
dense_44 (Dense)             (None, 1)                 3137
=================================================================
Total params: 40,705
Trainable params: 40,705
Non-trainable params: 0
_________________________________________________________________
```

Figure 2: Discriminator model summary, built with Keras



```
Model: "sequential_65"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_45 (Dense)             (None, 6272)              633472
_________________________________________________________________
leaky_re_lu_108 (LeakyReLU)  (None, 6272)              0
_________________________________________________________________
reshape_21 (Reshape)         (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_transpose_40 (Conv2DT (None, 14, 14, 128)       262272
_________________________________________________________________
leaky_re_lu_109 (LeakyReLU)  (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_transpose_41 (Conv2DT (None, 28, 28, 128)       262272
_________________________________________________________________
leaky_re_lu_110 (LeakyReLU)  (None, 28, 28, 128)       0
_________________________________________________________________
conv2d_66 (Conv2D)           (None, 28, 28, 1)         6273
=================================================================
Total params: 1,164,289
Trainable params: 1,164,289
Non-trainable params: 0
_________________________________________________________________
```

Figure 3: Generator model summary, built with Keras

2.4 Caveats

Large parts of the model architecture were borrowed from examples of GAN available online (references provided in the end). Features worth noting are mentioned below.

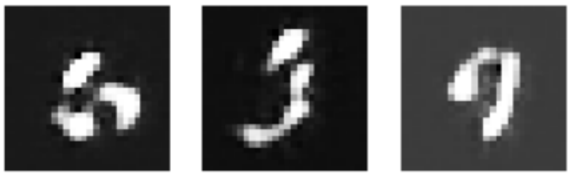
- `LeakyReLU` activation between convolutional layers with a slope of 0.2 has been reported to result in stable training most of the times.
- Transpose Convolution layers with strides of 2 is a better option than pooling layers for upsampling in generator.
- Sigmoidal activation in the last layer of the generator (`sigmoid` or `tanh`) so that outputs are scaled between -1 and $+1$. This also requires than bitmap intensity values are normalized to lie between 0 and 1.

3 Results

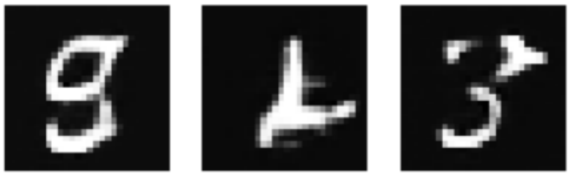
The network was initialized and trained with the following hyperparameters. Images generated by the generator after some epochs have been shown.

Learning rate	0.0002
Epochs	100
Batch size	128
Latent distribution dimension	100

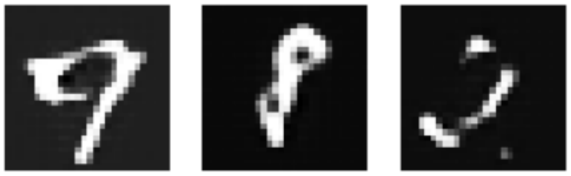
Epoch 5



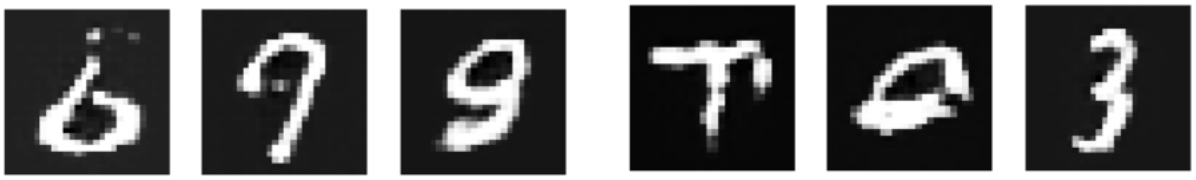
Epoch 25



Epoch 50



Epoch 90 and above



Here are my speculations on the "bonus" questions.

4 Latent Space Analogies

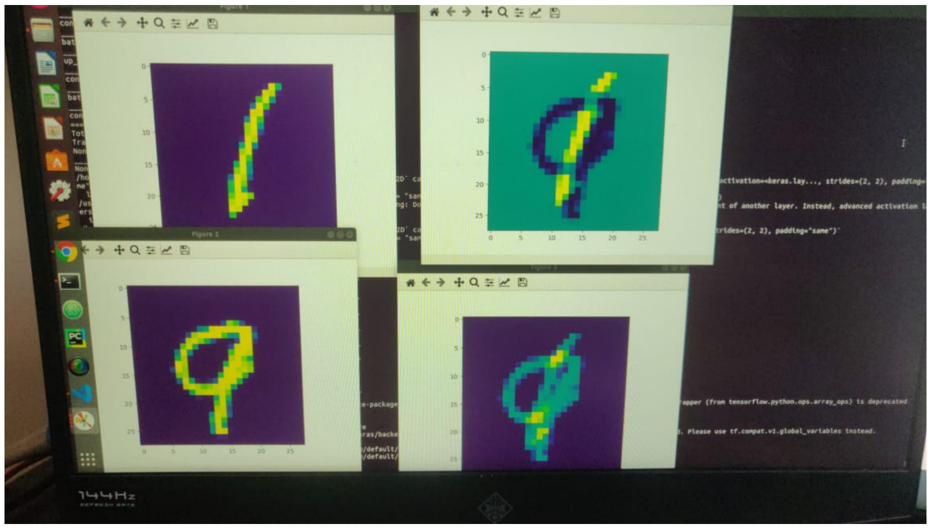


Figure 4: Adding and subtracting images to get hybrids

The latent space from which the generator produces an image acts like a representation of that image in reduced dimensions. Typically we would expect this representation to capture key features present in the image, like the presence of low intensity regions and high intensity regions in specific spatial orientations with respect to each other. We can retrieve the latent representation of an image by reverse passing them through a generator. Mathematical operations in this latent space will then affect the intensity with which a *feature* appears in a reconstructed image. It is perhaps due to this effect that we see the results shown in **Figure 4**.

5 Wasserstein GANs

The current form of GANs relies on the minimax loss to ensure convergence. However, this model can be annoyingly unstable and it is up to a human to stare at the outputs and verdict if they are plausible. Wasserstein GANs introduce a new loss function which has been empirically proven to be much more stable and informative.

Wasserstein Loss. Instead of having a discriminator to predict class probabilities (real or fake), WGANs use a committee of "critics" who output the **Earthmover (EM) distance** of a given image from a real image. Real images will have a smaller EM distance while fake images will have a relatively larger EM distance. For every batch of training samples provided, each critic in the committe outputs a score, whose aggregation (usually the mean) is considered the score for that batch. A "good" critic should have sufficient separation between its real and fake EM scores.

Implementation. We assign class labels of -1 for real images and $+1$ for fake images, with no loss of generality. The loss for the critics and the generator are defined as follows:

$$\text{Critic's Loss} = \mathit{mean}(\text{Critic scores on real images}) - \mathit{mean}(\text{Critic scores on fake images})$$

$$\text{Generator's Loss} = -\,\mathit{mean}(\text{Critic scores on fake images})$$

For a given batch of images, this loss can be condensed to this expression:

$$\text{Wasserstein loss for fake images} = -1 * \mathit{mean}(\text{Predicted scores})$$

$$\text{Wasserstein loss for real images} = 1 * \mathit{mean}(\text{Predicted scores})$$

```
def wasserstein_loss(y_true, y_pred):
    return mean(y_true * y_pred)
```

6 References

1. Ian Goodfellow, Generative Adversarial Networks (NIPS 2016 Tutorial): **YouTube**
2. Generative Models: **Stanford School of Engineering, YouTube**
3. How to Develop a GAN for Generating MNIST Handwritten Digits: **Machine Learning Mastery**
4. How to Implement Wasserstein Loss for Generative Adversarial Networks: **Machine Learning Mastery**