Conditional Generative Adversarial Networks with FashionMNIST

Burla Nur Korkmaz - Aras Umut Erarslan

I. Introduction

The aim of our project is to generate images using conditional generative adversarial networks(CGAN) to be able to generate labeled images. We choose to train our models on the Fashion MNIST dataset. We used different generator and discriminator models with different hyperparameters to test and compare the results to get the most realistic images. Evaluation of the results has been done by visualization of losses of models and the generated images.

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z}\sim p_{z}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}|\boldsymbol{y})))].$$

Fig.1. the cost function of CGAN

The reason we choose CGAN over GAN is there is no control over the data generation in GAN. CGAN has the same cost function as GAN(fig 1). By adding the label y as an additional parameter to the generator, the conditional GAN modifies this and expects that the corresponding images are generated(fig 2). To further differentiate real images, we also add labels to the discriminator input.

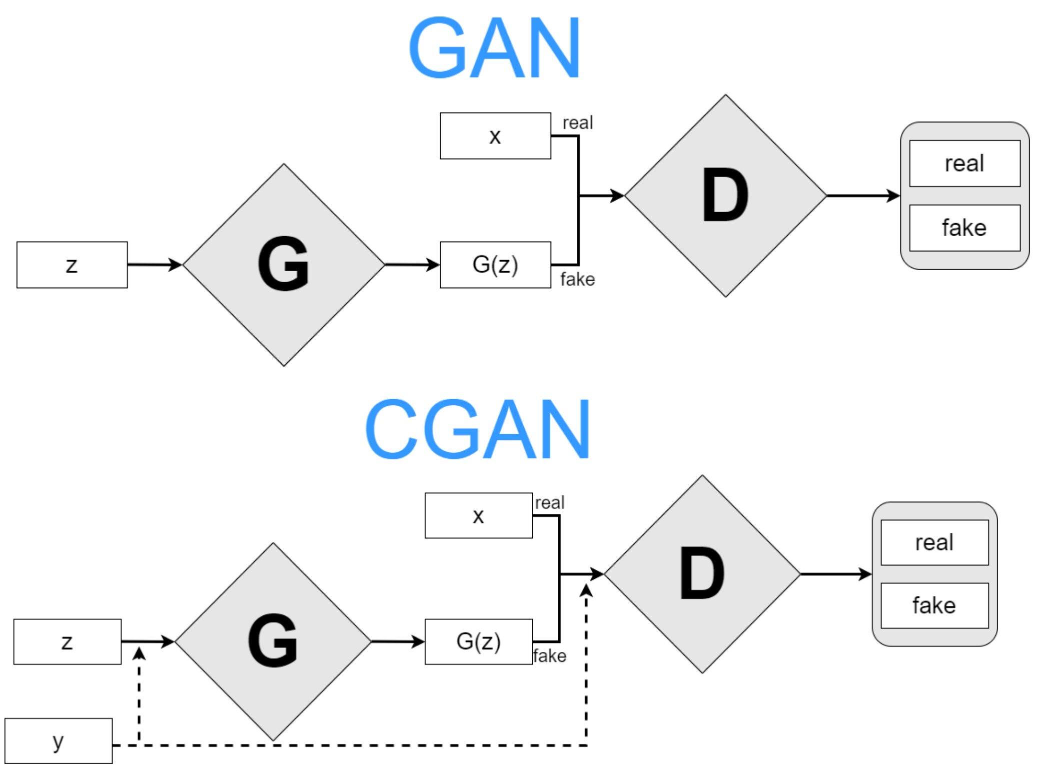


Fig.2. x symbolizes the real image, y the class label, z the noise vector, G the generator, and D the discriminator in GAN and conditional GAN (CGAN) architectures.

II. Implementation

While developing the application, we created the neural networks using the Keras library on GPU, and the plots were created with Matplotlib. train.py has been developed to train the models and save the trained models, and a file named generate.py has been prepared to load the saved models and generate new images with their labels as desired. While loading the models to generate new images in generate.py, we used the load_from_json function of keras. We would like to mention that the device we used for training has an RTX 2070S 8GB graphics driver.

The code can generate a new image with a generator at the desired interval and draw it as a plot and save this plot to the computer. This is a great way to observe how CGAN is doing and how it is improving. Again, it can save the generator and discriminator to the computer as “.h5” and “.json” in different ways at the desired interval.

II.I. Dataset

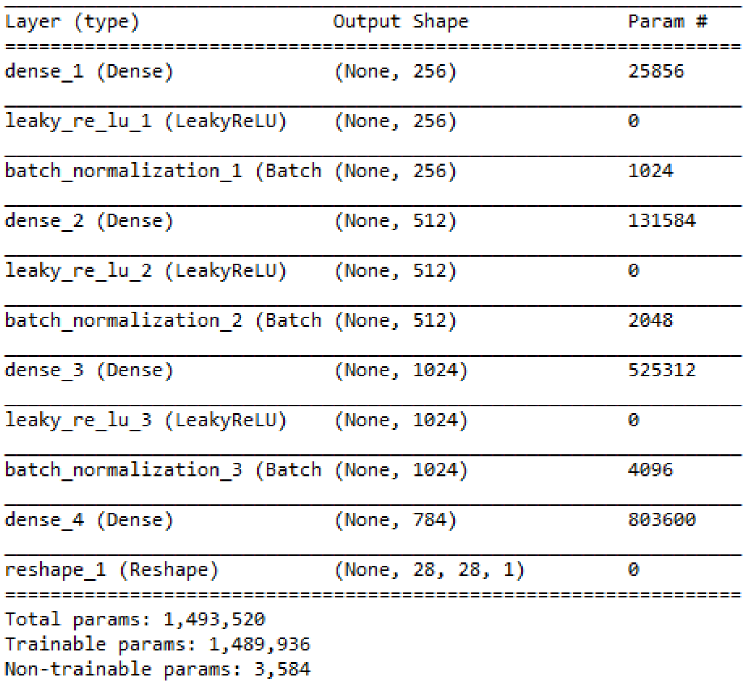
Dataset was downloaded from the official Fashion MNIST github page. Downloaded files’ extension is “.gz”. A “.gz” file is a compressed archive created using the GNU zip (gzip) compression technique. It's a compressed collection of one or more files that are often used for file compression on Unix operating systems. For this reason, we used gzip python library to load the dataset. We imported the NumPy library when assigning the data to arrays and for the operations to be used later. The dataset contains 10 different classes. Since we are using CGAN, we need a class-map when creating data with the generator. So we created an array holding class names. The downloaded dataset includes 28x28 images with color channel 1 (grayscale images). At the very beginning of our code, we created variables for all the parameters we will need later and assigned their values to them.

II.II. Generators

We tried over 10 generator models to use in CGAN and finally, we decided that 2 of them are suitable to use in this project. Among the 2 of them, there is a cost-quality trade. While we named the first generator model we used as “Generator X”, we named the second generator model we used as “Generator Y”. We will touch on the cost-quality relationship between them in detail later in the evaluation section. For both generators, we took input_dim as 100(num_features).

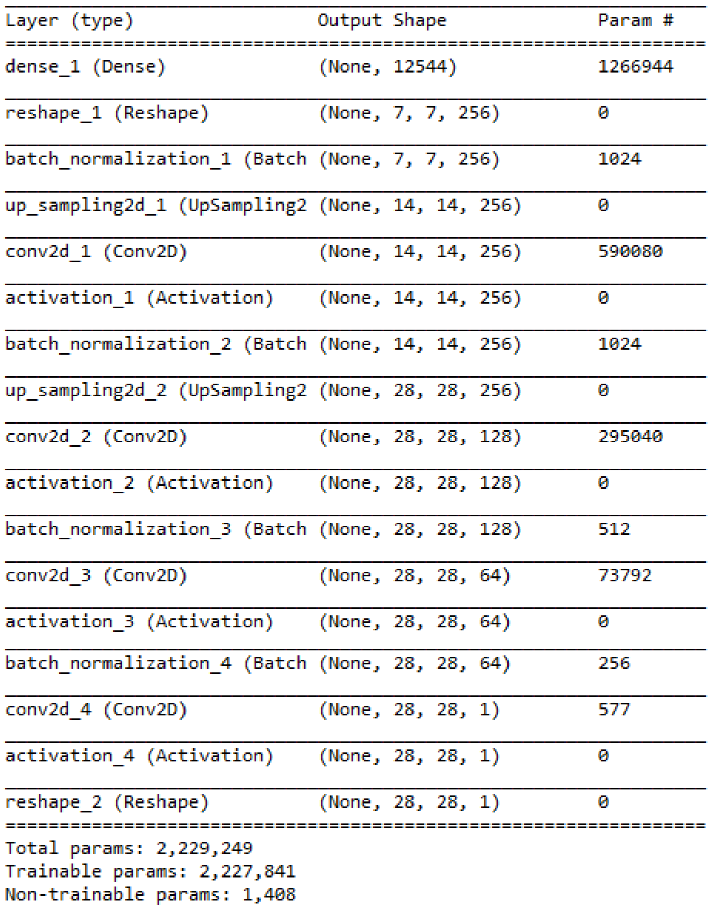
II.II.I Generator X

Although Generator X contains 4 dense layers, leaky relu and batch normalization are used between these layers. While the shape of the layers used is 256, 512, 1024, the last one that will output is 784 (fig 3). The reason for this is that the images will finally be reshaped into a 28x28 shape. Afterward, the activation function was set to “tanh”. After the model was created, we combined it with random noise and label to finalize it. Once and for all, Generator X has 1,489,936 trainable parameters.



```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 256)               25856
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 256)               0
_________________________________________________________________
batch_normalization_1 (Batch (None, 256)               1024
_________________________________________________________________
dense_2 (Dense)              (None, 512)               131584
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 512)               0
_________________________________________________________________
batch_normalization_2 (Batch (None, 512)               2048
_________________________________________________________________
dense_3 (Dense)              (None, 1024)              525312
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 1024)              0
_________________________________________________________________
batch_normalization_3 (Batch (None, 1024)              4096
_________________________________________________________________
dense_4 (Dense)              (None, 784)               803600
_________________________________________________________________
reshape_1 (Reshape)          (None, 28, 28, 1)         0
=================================================================
Total params: 1,493,520
Trainable params: 1,489,936
Non-trainable params: 3,584
```

Fig.3. the architecture of Generator X



```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 12544)             1266944
_________________________________________________________________
reshape_1 (Reshape)          (None, 7, 7, 256)         0
_________________________________________________________________
batch_normalization_1 (Batch (None, 7, 7, 256)         1024
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 14, 14, 256)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 256)       590080
_________________________________________________________________
activation_1 (Activation)    (None, 14, 14, 256)       0
_________________________________________________________________
batch_normalization_2 (Batch (None, 14, 14, 256)       1024
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 28, 28, 256)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 28, 28, 128)       295040
_________________________________________________________________
activation_2 (Activation)    (None, 28, 28, 128)       0
_________________________________________________________________
batch_normalization_3 (Batch (None, 28, 28, 128)       512
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 28, 28, 64)        73792
_________________________________________________________________
activation_3 (Activation)    (None, 28, 28, 64)        0
_________________________________________________________________
batch_normalization_4 (Batch (None, 28, 28, 64)        256
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 28, 28, 1)         577
_________________________________________________________________
activation_4 (Activation)    (None, 28, 28, 1)         0
_________________________________________________________________
reshape_2 (Reshape)          (None, 28, 28, 1)         0
=================================================================
Total params: 2,229,249
Trainable params: 2,227,841
Non-trainable params: 1,408
```

Fig.4. the architecture of Generator Y

II.II.II Generator Y

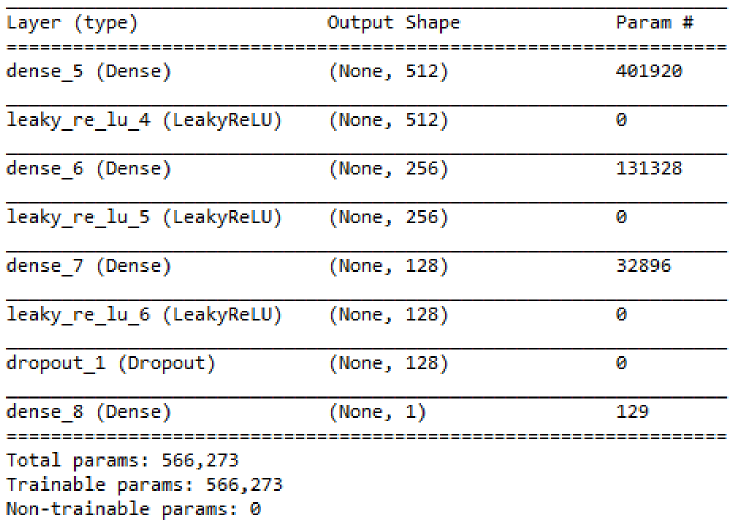
Dense layers are used in Generator X, while convulutional2d is used in Generator Y (fig 4). In addition, upsampling was applied between conv2ds. Since the output shape must be 28x28 and upsampling will be applied 2 times, the input shape must be 7x7. We also started with 256 of the first Conv2d. For this reason, the dense layer at the entrance should have a value of 7*7*256, that is 12544. After the first dense layer, 3 conv2d's with kernels_size =3 and padding "same", 256, 128, 64, were used. Conv2d with 1 neuron is used in the output layer. The reason for this is that the picture is requested to be 28x28. Then, as in Generator X, the activation function is set to "tanh". Compared to Generator X, Generator Y is 47.5% larger with 2,227,841 trainable parameters. Just like Generator X, it is finalized by combining the created random noise and label.

II.III Discriminators

While we were experimenting with different generator models, we tried different discriminators to be compatible with them. We have 2 discriminators for 2 generators. The name of the discriminator we use with Generator X will be “Discriminator X”, while the name of the Discriminator we use with Generator Y will be called “Discriminator Y”. The cost-quality balance will be explained in the evaluation section.

II.III.I Discriminator X

Discriminator X is used with Generator X, which is smaller than Generator Y. It has 3 dense layers of 512, 256, 128, and leaky relu layers between them (fig. 5). Output is obtained with a dense layer by applying a 0.5 dropout before the output dense layer. Next, the activation function is set to sigmoid. Finally, the model is finalized by combining a 28x28 symbolic keras.input function with a symbolic keras.input function created according to the number of classes and labels. While the optimizer of the model is set to Adam(0.0002, 0.5), its loss is loss='binary_crossentropy'. These created inputs were created to replace the pictures produced from the generator or from the dataset while the models are being trained. It has a total of 566,273 trainable parameters.

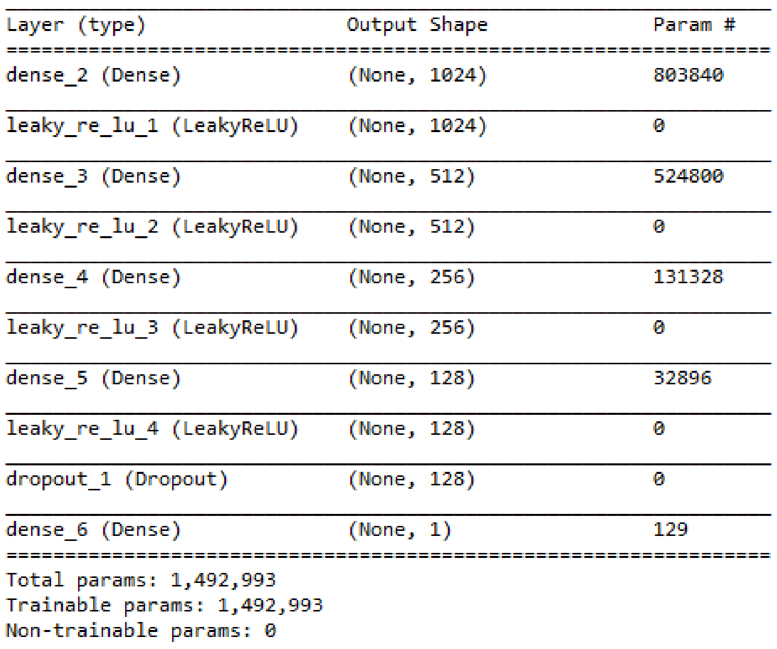


```
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 512)               401920
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 256)               131328
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 256)               0
_________________________________________________________________
dense_7 (Dense)              (None, 128)               32896
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 128)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 129
=================================================================
Total params: 566,273
Trainable params: 566,273
Non-trainable params: 0
```

Fig.5. the architecture of Discriminator X

II.III.I Discriminator Y

Discriminator Y is used with Generator Y. Just as Generator Y is larger than Generator X, Discriminator Y is larger than Discriminator X (fig 6). Thus, the balance between the generator and the discriminator is established. In addition to the Discriminator X model, a dense layer with 1024 neurons has been added to the first layer. With this new layer added, the number of trainable parameters of the discriminator has increased to 1,492,993.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_2 (Dense)              (None, 1024)              803840    
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 1024)              0         
_________________________________________________________________
dense_3 (Dense)              (None, 512)               524800    
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 512)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 256)               131328    
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 256)               0         
_________________________________________________________________
dense_5 (Dense)              (None, 128)               32896     
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 128)               0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 129       
=================================================================
Total params: 1,492,993
Trainable params: 1,492,993
Non-trainable params: 0
```

Fig.6. the architecture of Discriminator Y

II.IV Training

Before starting the training process, a new model named merged_CGAN was created, and the created generator and the appropriate discriminator were combined in this model. We are going to call the process we trained the Generator X and Discriminator X, the Training X, and the process we trained the Generator Y and Discriminator Y, the Training Y. For both training processes we used a batch size of 32 and we trained for 100k epochs. Previously, we said that we produced new images from a generator at certain intervals and showed them and recorded the model at certain intervals. Although we selected 1000 epochs as the image generating interval, we set the models saving interval to be 1000 epochs after the 10000th epoch. Thus, we were able to constantly control the whole training. Since images are created from 0 to 255, we divided all the pixels of images by 127.5 and extracted 1 from all the pixels to rescale all the pixels of the images between -1 and 1. We created the necessary variables to follow the loss of the discriminator with real and fake images and the loss of the generator in each epoch, and then we assigned the values to the arrays that should belong to each epoch in order to create a plot. For the training of the discriminator, we randomly took pictures from the data set and generated fake pictures from the generator. For the training of the generator, random noise, random labels, and data from the discriminator was used. In the training phase, in addition to generating and displaying a new image every 1000 epochs, the loss of the discriminator with real and fake images was printed to the console every 100 epochs.

II.V Plotting

During the training process, real and fake loss of discriminator and generator loss is saved into an array epoch by epoch. At the end of the training, all these 3 arrays are used to create a line plot with the Matplotlib library. Since we have many epochs(100k) before the plot is drawn, we eliminated every 99 value in every 100 value to create a better visualization of the data. That allows us to generally analyze and see what is going on with the models.

III. Evaluation

We would like to mention that, for both training processes, the batch size is taken as 32, and the epoch is taken as 100k.

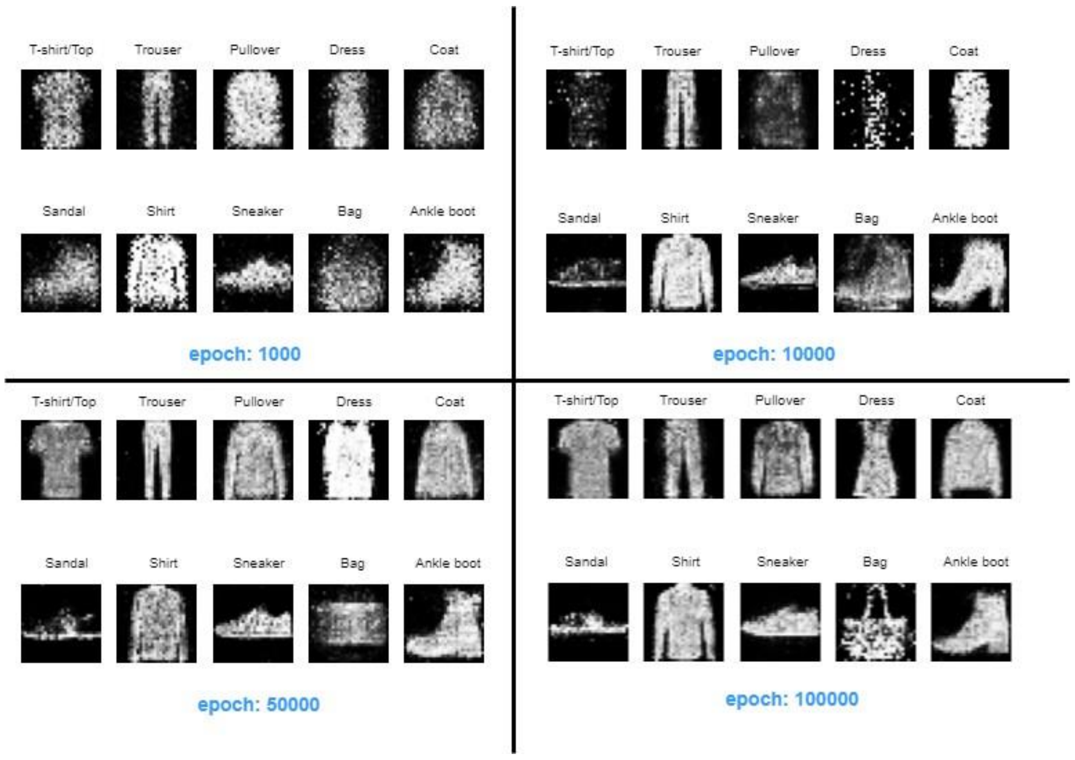


Fig.7. generated images by “Generator X” while during the training of “Training X”

III.I Training X

For Training X, we used very small models that are completed the 100k epoch in 1700 seconds with RTX 2070S 8GB graphics card including all the training processes we implemented like saving the models and plots. We started to get some scattered points that are shaped like the dataset after 1000 epochs(fig 7). After the 10k epochs, some instances are started to be created almost perfectly while some of them just point clouds. Around 50k epochs, the output of the generator became a very understandable shape. We were able to label them very easily with our hands. Only the bag class had small problems but it was still distinguishable. By the time we got to an epoch of around 100k, all the classes in all the created images were very easily distinguishable from one another. Although we occasionally saw individual points(pixels) in the picture, these points created almost no pollution. We've had pretty good results for small models that can be trained that fast.

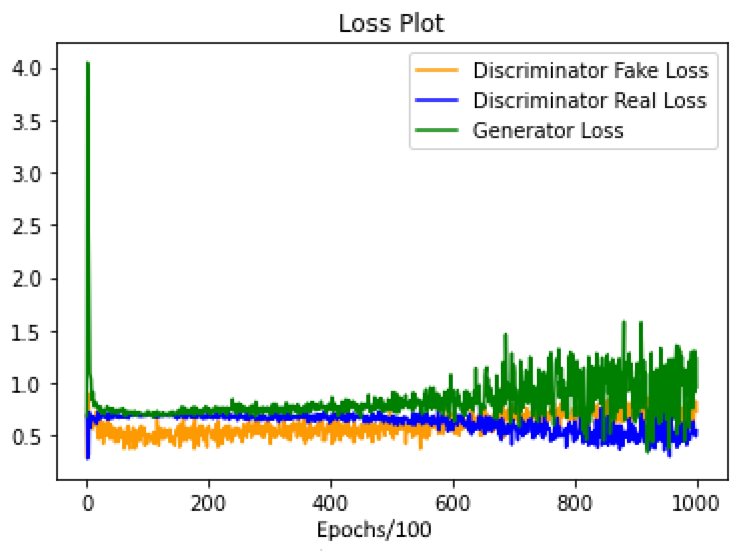


Fig.8. loss plot that is created by using the real and fake losses of Discriminator X and generator loss of Generator X. Every value in the x coordinate corresponds to 100. That means the total of the epoch is 100000

From what we've observed, the generator loss always experiences an absurd spike in the first epochs of training. As seen in Fig 8, we see the same situation in Training X. After about the 1000th epoch, the generator loss, and discriminator real loss stabilize and become regular (fig. 8). While the discriminator real loss is at its most stable in the 10000th epoch, it becomes more irregular as the epoch progresses. If we look at the generator loss, it started to become more and more irregular as the number of epochs increased, creating large fluctuations. If we talk about Discriminator fake loss, we see that it generally increases slightly as training continues. All these experiences are necessary for the models to be trained well, according to our calculations and observations. For example, if the discriminator accuracy was always 50%, it would mean that the generator could always be distinguished, and this would negatively affect the training of the models. Sitting around a certain value by experiencing such fluctuations is very important for the success of training.

III.II Training Y

For Training Y, we have used bigger generator and discriminator models. The models completed the 100k epoch in 3300 seconds with RTX 2070S 8GB graphics card including all others processes during our training section like saving the models and plots. If we compare the training time of Training X with Training Y, Training Y's training time almost doubles the training X's. We mentioned that the models we used in Training Y have much more training parameters to be trained so it is the reason for that much training time. As we observed from the images that Generator Y created during the training process, even at the 1000th epoch, smooth shapes began to appear. The points were not as messy as the Generator X produced and were much more similar to the images in the training data. When we look at the images produced in the 10000th epoch, we observed that some classes could be generated very successfully, but some were still incomprehensibly unsuccessful (Fig 9). When we looked at the generated images in the 50000th epoch, we observed that they were almost perfect. Compared to what Generator X produced, Generator Y was much more successful. In the 50000th epoch, Generator Y could very understandably generate classes that Generator X could not successfully generate. In the 100000th epoch, The model was able to generate objects with different characteristics such as bags and high heels in a very understandable way. Training Y was a much more successful training than Training X, but took much longer in return. In addition, since the models used in Training Y are larger in structure, the space they took up in the folder when saved was almost double that of the Training X models.

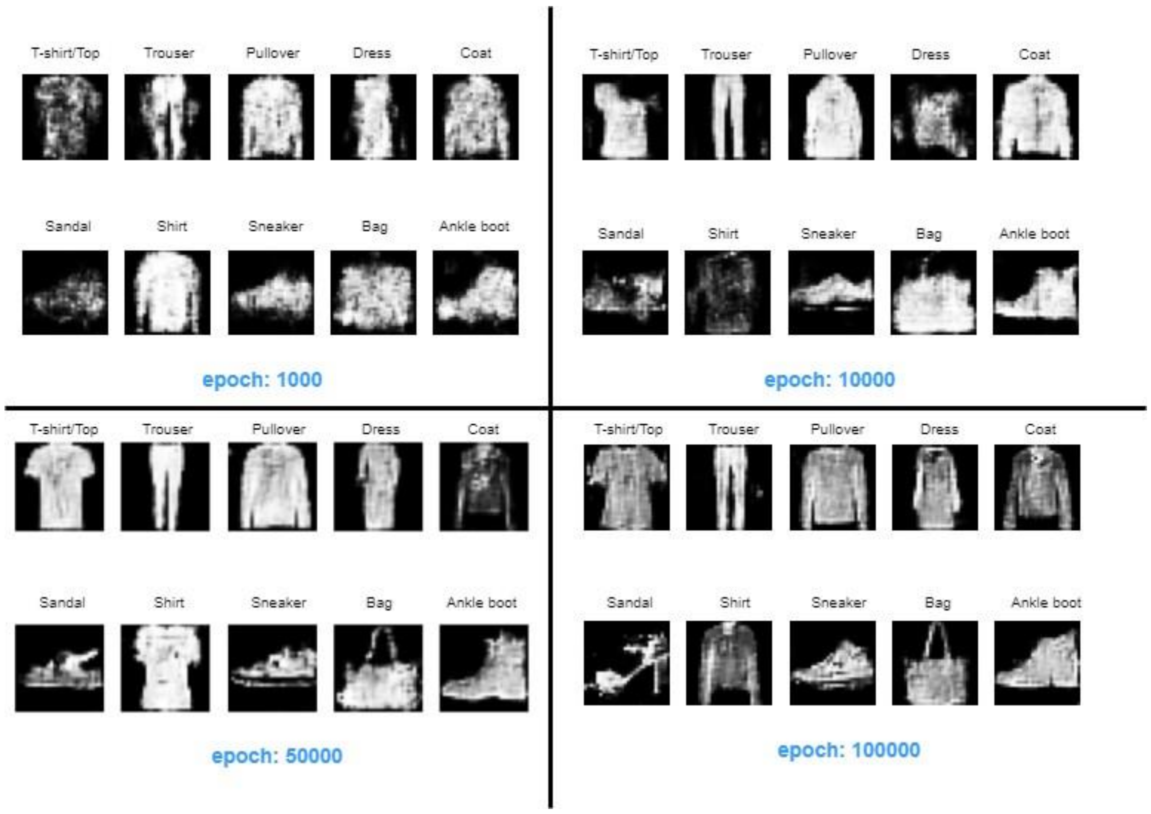


Fig.9. generated images by "Generator Y" while during the training of "Training Y"

As we observed in the plot of Training X, Generator Y also has a spike in its initial epochs (Fig 10). As we can see in Fig 10, generator loss and discriminator real loss stabilizes around the 400th epoch. Discriminator fake loss never stabilizes like in Fig 8. Whenever there is a new update to the discriminator or generator, it changes depending on these changes of the models. This is because of the nature of the GAN. Both Training X and Training Y almost have the same plot at the end except for one small difference. In Fig 8, discriminator fake loss passes discriminator real loss around 60000th epoch while in Fig 10, discriminator fake loss could not catch discriminator real loss yet. We observe that the average value of discriminator fake loss increases when time passes.

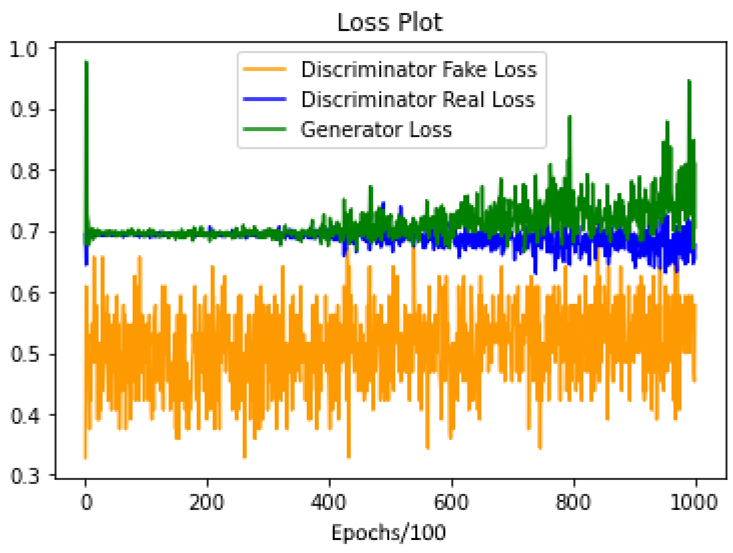


Fig.10. loss plot that is created by using the real and fake losses of Discriminator Y and generator loss of Generator Y. Every value in the x coordinate corresponds to 100. That means the total of the epoch is 100000

IV. Conclusion

We would say that both of the outputs of Generator X and Generator Y are satisfying. We observed that many different models and parameters can be implemented to train a CGAN to obtain generated images and use discriminators in certain functions. In this project, we implemented 2 generators and 2 discriminators that are matched with each other. A bigger model can give better results while taking more time to train. We created many bigger models to train with this dataset but we observed that the dataset is not complicated that much and huge models do not give better results always. The best thing to do is to find the most suited models and parameters to meet the requirements of the project. Both CGAN models we presented can be used for this project depending on the cost-quality requirements.

V. Generated Images

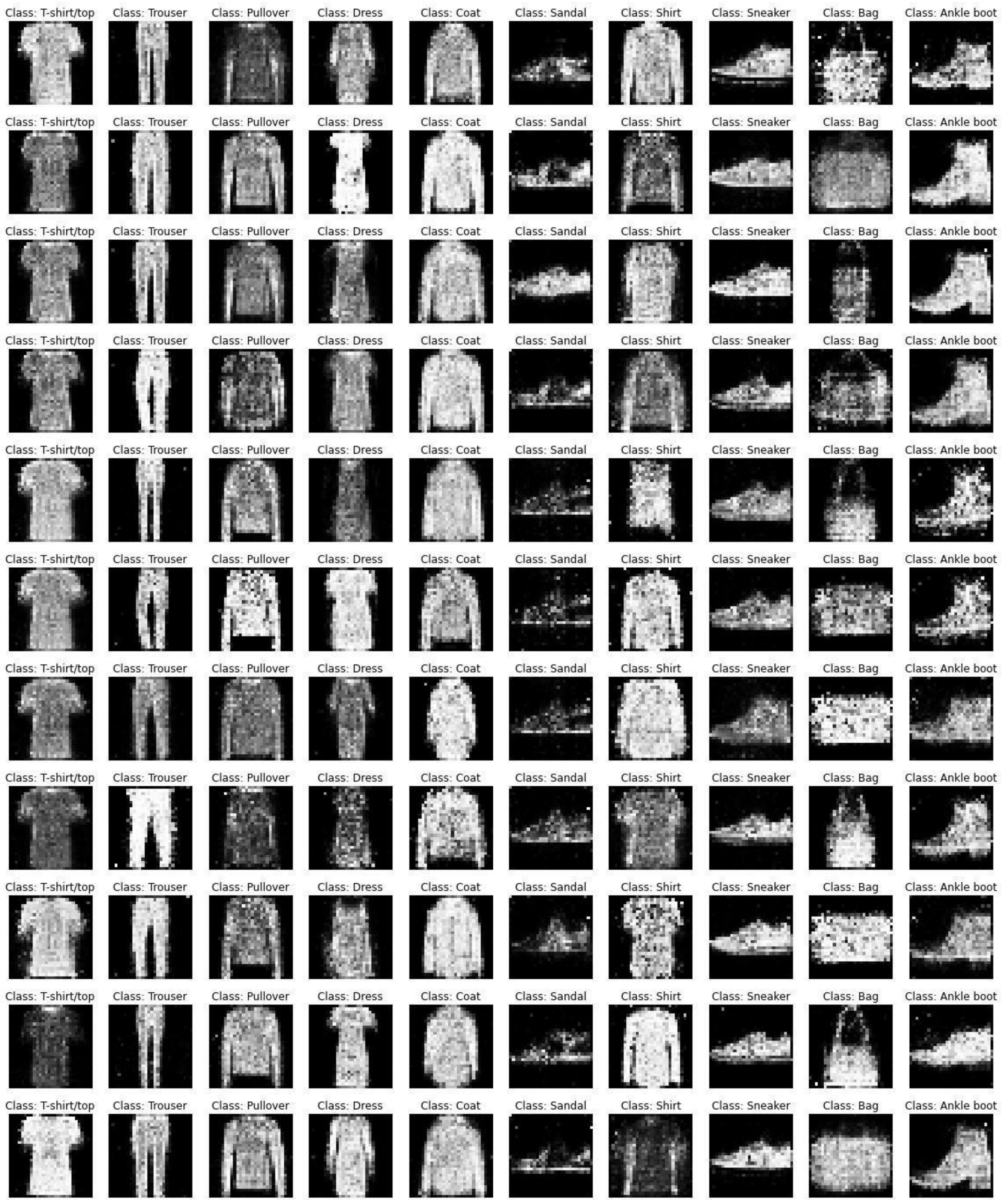


Fig.11. images generated by Generator X after the 28.3 minutes of training X models(100k epochs).

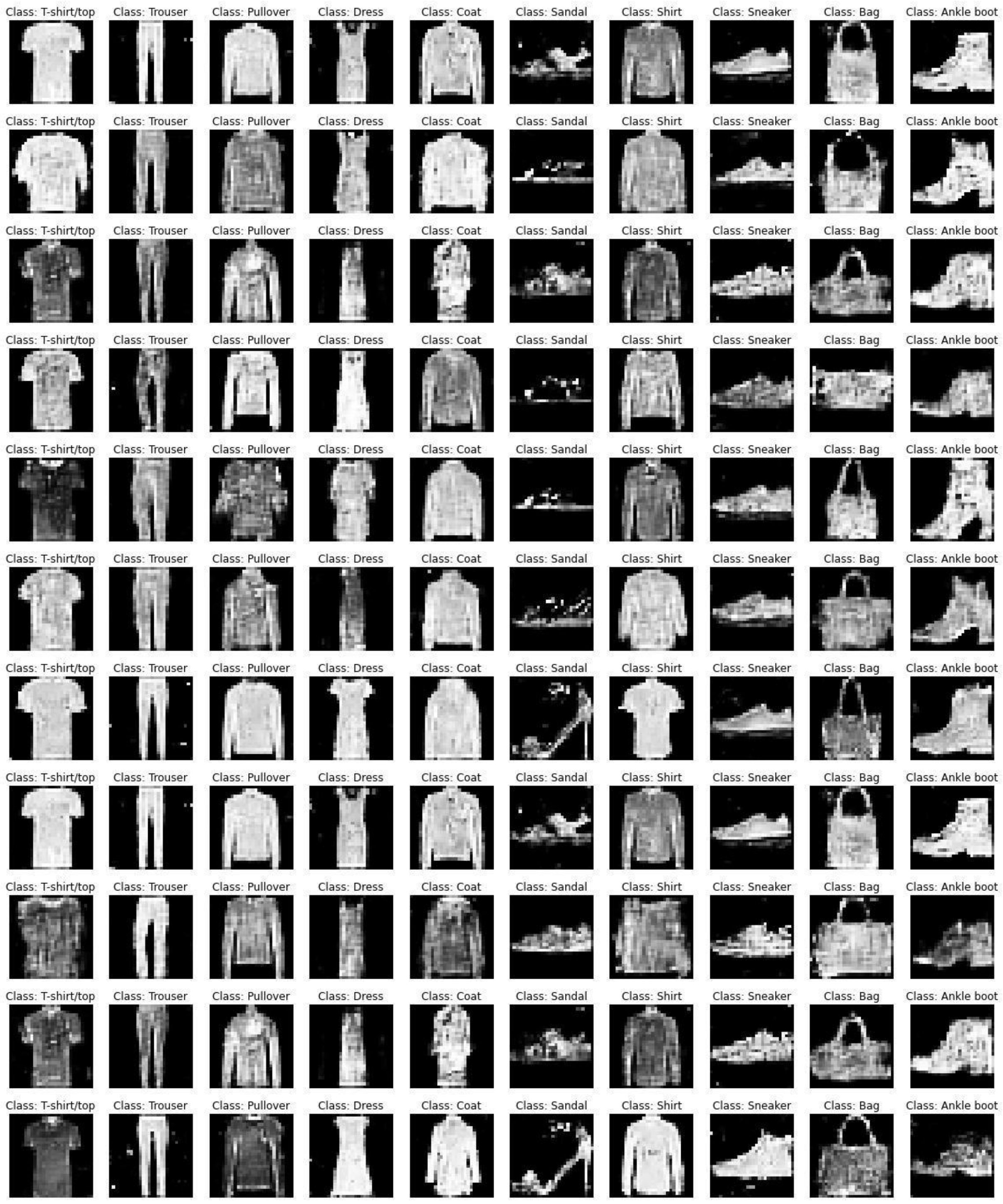


Fig.12. images generated by Generator Y after the 55 minutes of training Y models(100k epochs).