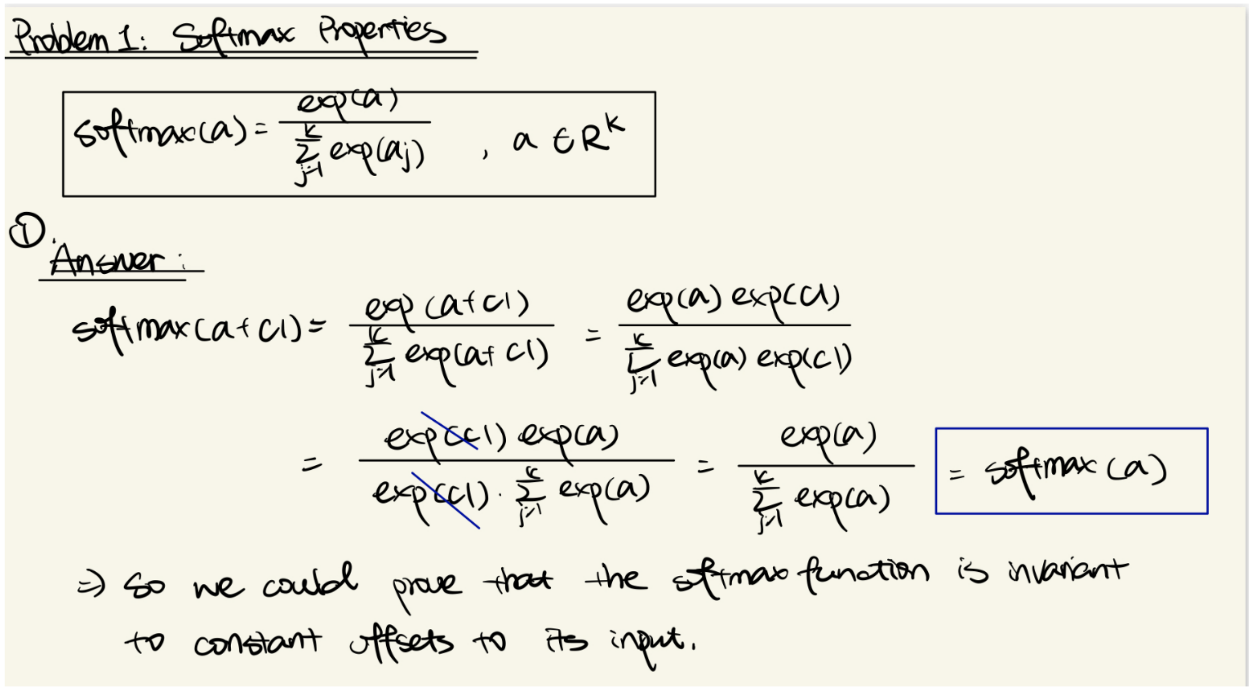
BAX 423 HW5 Deep Learning

Submitted by The Analyst Amigos

Problem 1: Softmax Properties

1. Show that the softmax function is invariant to constant offsets to its input, i.e., softmax(a + c1) = softmax(a), where c ∈ R is some constant and 1 denotes a column vector of 1s.



Problem 1: Softmax Properties

$$\boxed{\text{softmax}(a) = \frac{\exp(a)}{\sum_{j=1}^{k}\exp(a_j)} \quad , \; a \in R^k}$$

①. Answer:

$$\text{softmax}(a+c1) = \frac{\exp(a+c1)}{\sum_{j=1}^{k}\exp(a+c1)} = \frac{\exp(a)\exp(c1)}{\sum_{j=1}^{k}\exp(a)\exp(c1)}$$

$$= \frac{\cancel{\exp(c1)}\exp(a)}{\cancel{\exp(c1)}\cdot\sum_{j=1}^{k}\exp(a)} = \frac{\exp(a)}{\sum_{j=1}^{k}\exp(a)} \boxed{= \text{softmax}(a)}$$

⇒ So we could prove that the softmax function is invariant to constant offsets to its input.

2. In practice, why is the observation that the softmax function is invariant to constant offsets to its input important when implementing it in a neural network?

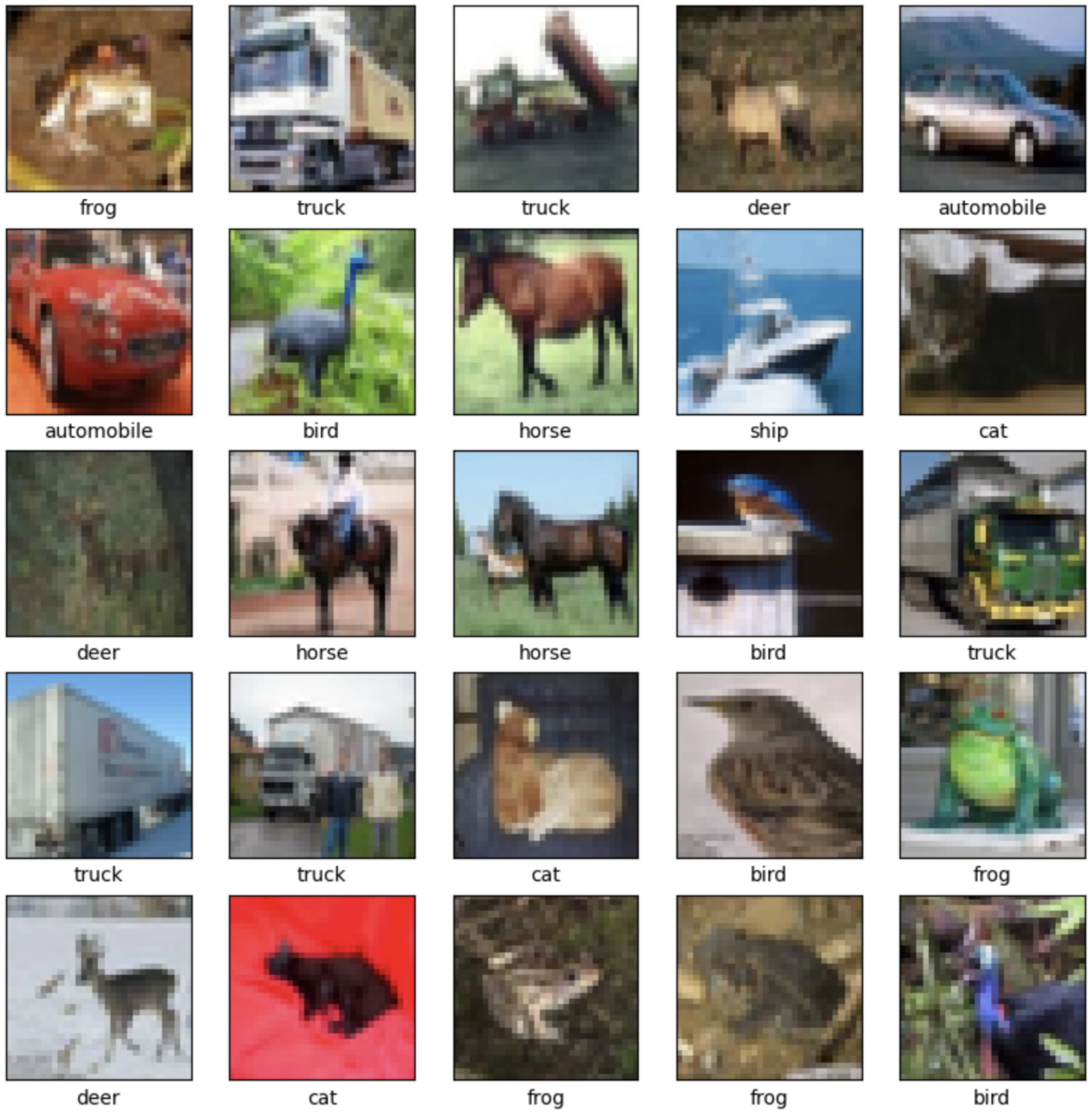
When implementing softmax function in a neural network, we might run into numerical instability problems. Specifically, the exponential function used in softmax can blow up for large input values, which could lead to NaN (Not a Number) values. This is because computers have a finite precision, and when numbers get too large or too small, they can't be represented accurately.

To prevent this from happening, a common practice is to subtract the maximum element in the input vector from all elements in the vector before feeding it into the softmax function. This technique ensures that the largest value fed into the exponential function is 0, thereby preventing large outputs. This doesn't change the results due to the invariant property, but it does improve numerical stability.

This is crucial in practical applications of neural networks, where numerical instability could cause the training process to fail.

Problem 2: Training a CNN using CIFAR-10 Data

1. Load the dataset and check how the images look like

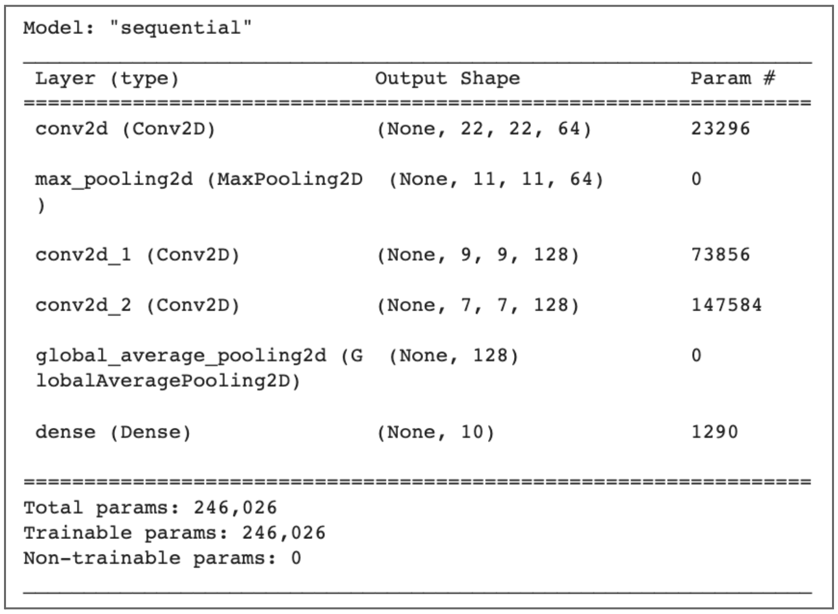


2. Train a CNN with three hidden convolutional layers that use the ReLU activation function. Use 64 11×11 filters for the first layer, followed by 2×2 max pooling (stride of 2). The next two convolutional layers will use 128 3×3 filters followed by the ReLU activation function. Prior to the

softmax layer, you should have an average pooling layer that pools across the preceding feature map. Do not use a pre-trained CNN. Train your model using all of the CIFAR-10 training data, and evaluate your trained system on the CIFAR-10 test data. Display the training loss as a function of epochs. What is the accuracy of the test data? How did you initialize the weights?

In this question, we choose **the standard Keras initializers** for the weights, which is 'Glorot uniform' or 'Xavier uniform' initializer. It aims to maintain the variance of the activations and back propagated gradients roughly constant across layers. This initialization helps to ensure that the signal from the input data is neither too small nor too large, which can cause the weights and the network to fail to learn from the data effectively.

We use the **Adam** optimizer for training, and initialize it with a learning rate of 0.001. The model will be trained for 20 epochs (takes about 1 hour). The loss function is sparse categorical cross entropy, which is suitable for multi-class classification problems where the labels are integers.

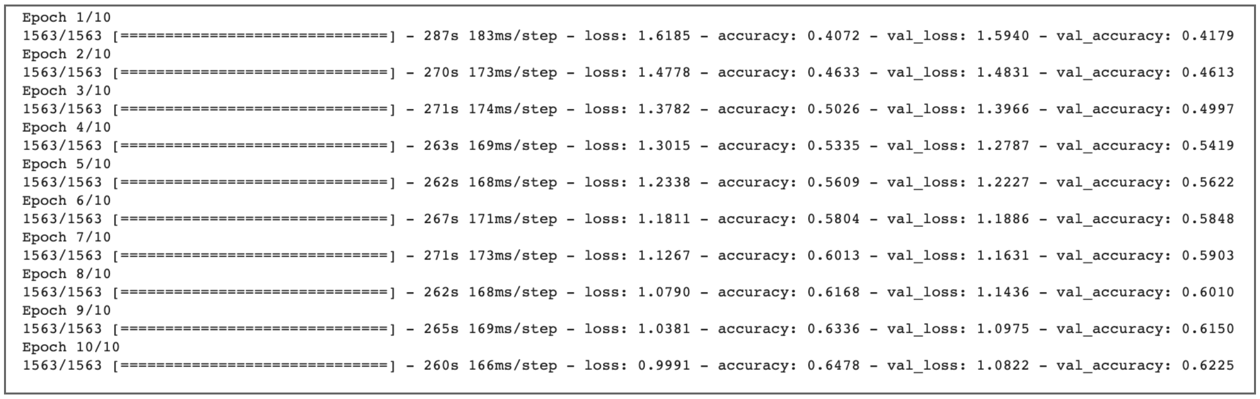


```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 22, 22, 64)        23296

 max_pooling2d (MaxPooling2D  (None, 11, 11, 64)       0
 )

 conv2d_1 (Conv2D)           (None, 9, 9, 128)         73856

 conv2d_2 (Conv2D)           (None, 7, 7, 128)         147584

 global_average_pooling2d (G  (None, 128)              0
 lobalAveragePooling2D)

 dense (Dense)               (None, 10)                1290

=================================================================
Total params: 246,026
Trainable params: 246,026
Non-trainable params: 0
_________________________________________________________________
```

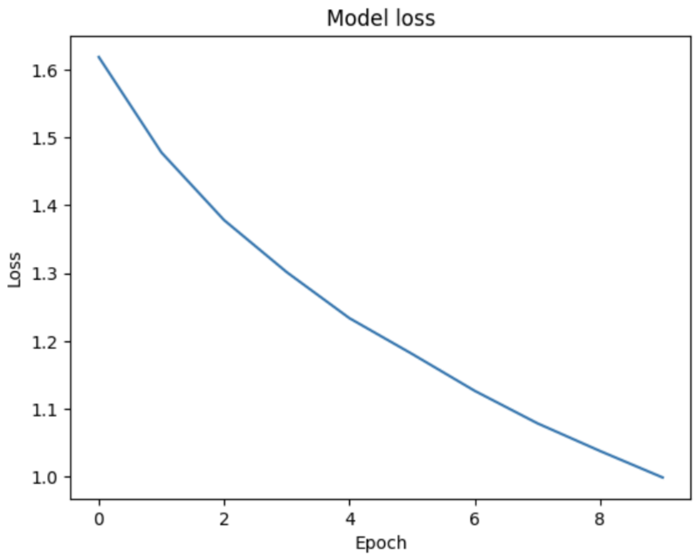
The hyperparameters are:

1) Learning rate: 0.001 (commonly used value for Adam optimizer)
2) Number of epochs: 10 (this might need to be increased if the model has not fully converged)

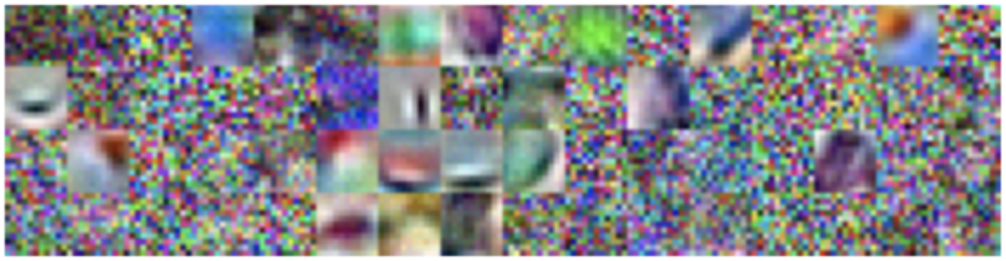
The final test accuracy is 0.6225 with around 0.3775 test error, with an increase in accuracy over ten epochs, while the loss decreased with each epoch. The loss decreased as the number of epochs increased, going from above 1.6 to below 1.0.



```
Epoch 1/10
1563/1563 [==============================] - 287s 183ms/step - loss: 1.6185 - accuracy: 0.4072 - val_loss: 1.5940 - val_accuracy: 0.4179
Epoch 2/10
1563/1563 [==============================] - 270s 173ms/step - loss: 1.4778 - accuracy: 0.4633 - val_loss: 1.4831 - val_accuracy: 0.4613
Epoch 3/10
1563/1563 [==============================] - 271s 174ms/step - loss: 1.3782 - accuracy: 0.5026 - val_loss: 1.3966 - val_accuracy: 0.4997
Epoch 4/10
1563/1563 [==============================] - 263s 169ms/step - loss: 1.3015 - accuracy: 0.5335 - val_loss: 1.2787 - val_accuracy: 0.5419
Epoch 5/10
1563/1563 [==============================] - 262s 168ms/step - loss: 1.2338 - accuracy: 0.5609 - val_loss: 1.2227 - val_accuracy: 0.5622
Epoch 6/10
1563/1563 [==============================] - 267s 171ms/step - loss: 1.1811 - accuracy: 0.5804 - val_loss: 1.1886 - val_accuracy: 0.5848
Epoch 7/10
1563/1563 [==============================] - 271s 173ms/step - loss: 1.1267 - accuracy: 0.6013 - val_loss: 1.1631 - val_accuracy: 0.5903
Epoch 8/10
1563/1563 [==============================] - 262s 168ms/step - loss: 1.0790 - accuracy: 0.6168 - val_loss: 1.1436 - val_accuracy: 0.6010
Epoch 9/10
1563/1563 [==============================] - 265s 169ms/step - loss: 1.0381 - accuracy: 0.6336 - val_loss: 1.0975 - val_accuracy: 0.6150
Epoch 10/10
1563/1563 [==============================] - 260s 166ms/step - loss: 0.9991 - accuracy: 0.6478 - val_loss: 1.0822 - val_accuracy: 0.6225
```



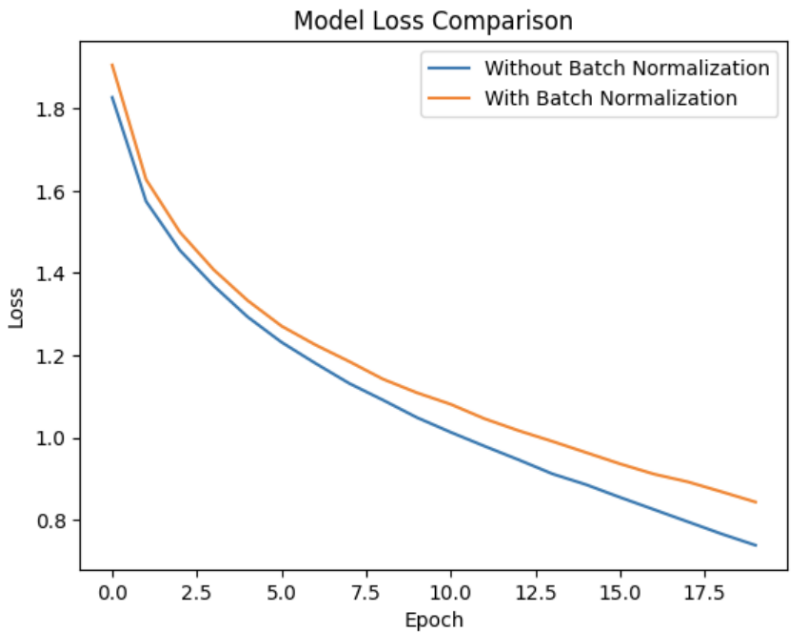
3. Visualize all of the 11×11×3 filters learned by the first convolutional layer as an RGB image array (I suggest making a large RGB image that is made up of each of the smaller images, so it will have 4 rows and 16 columns). Note that you will need to normalize each filter by contrast stretching to do this visualization, i.e., for each filter subtract the smallest value and then divide by the new largest value.



4. Using the same architecture, add in batch normalization between each of the hidden layers. Compare the training loss with and without batch normalization as a function of epochs. What is the final test error? Visualize the filters.

Based on the calculated loss with and without batch normalization and the line graphs below, we observed that the loss is larger when using batch normalization compared to when it is not used. Additionally, the gap between the two lines appears to increase as the number of epochs progresses, although both lines show a decreasing trend over the epochs. The final test error is around 0.3644, loss is 1.0788, and accuracy is 0.6356

Loss with vs. without batch normalization:



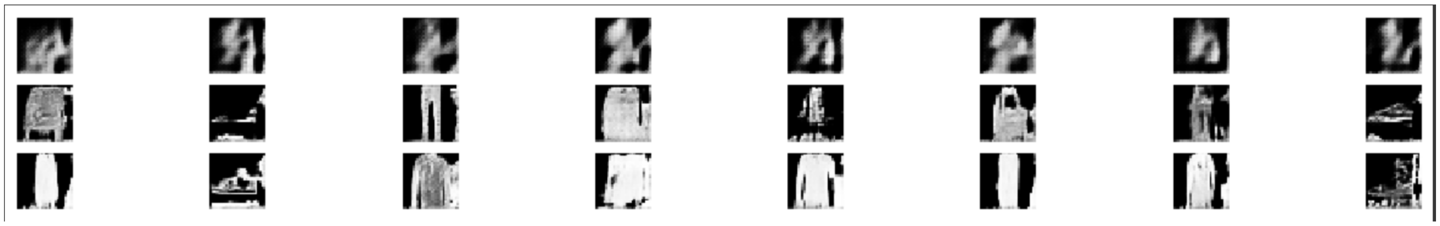
Visualized filter:

Problem 3: GAN using Fasion-MNIST Data

Part 1 - Vanilla GAN

We trained a GAN with a Discriminator and Generator. As specified in the question, both models have 3 hidden layers with ReLU as the activation function. The discriminator’s outer layer uses tanh (hyperbolic tan) as the activation function in the output layer. Since tanh ranges from -1 to 1, we normalize the images accordingly. The Generator uses signmoid as the activation function. Additionally, We chose the Adam optimizer for optimization. The parameter selection was based on suitable values provided online and several iterations until an acceptable model was created. The selected number of epochs is 50, although we may experience a better convergence if the training is increased. Below are the images created in the first, intermediate and final training stages.

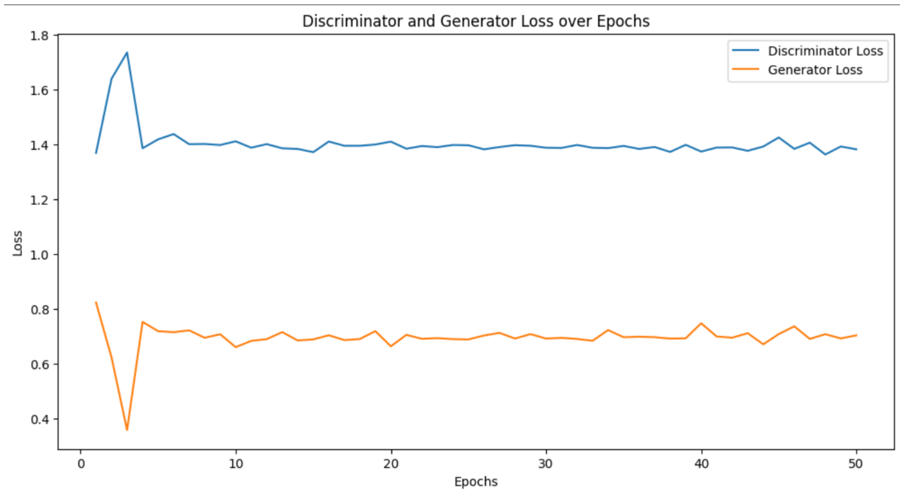
Show the generated samples from G in 1) the beginning of the training; 2) intermediate stage of the training, and 3) after convergence.



Training stages - epoch = 0, 25, 49 respectively

Based on the Discriminator and Generator Loss captured in every epoch, we have created the below plot of Loss vs epoch number.

Plot your training loss curves for your G and D.



We can see that while initially the Losses for D & G were converging, they grew further apart and stabilized around 1.4 and 0.75 respectively, not converging any further in 50 epochs

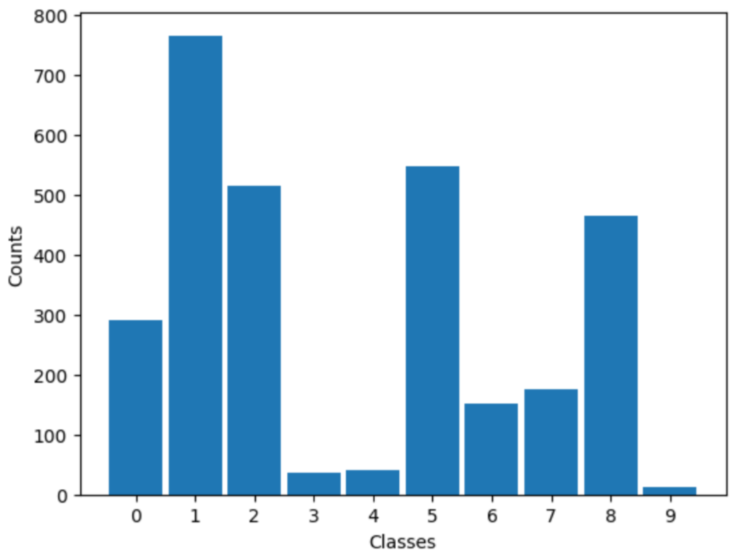
Part 2 - Mode Collapse in GANs

<u>Classifier</u>

We created the classifier by copying the discriminator but changing output layer unit count to 10. After created 3000 new samples, we see that due to mode collapse there are more samples belonging to class 2

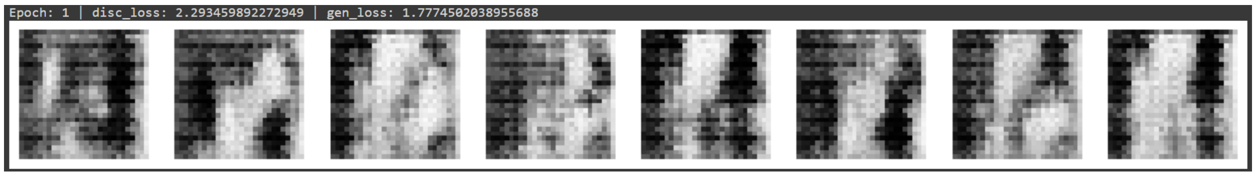
Generate 3000 samples using the generator you trained in part 1. Use the classifier you just trained to predict the class labels of those samples. Plot the histogram of predicted labels.

Final Output

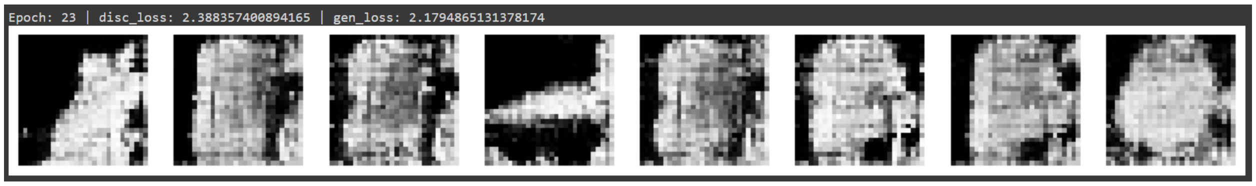


There is evidence of mode collapse as the counts for class = 1 is much higher than the rest.

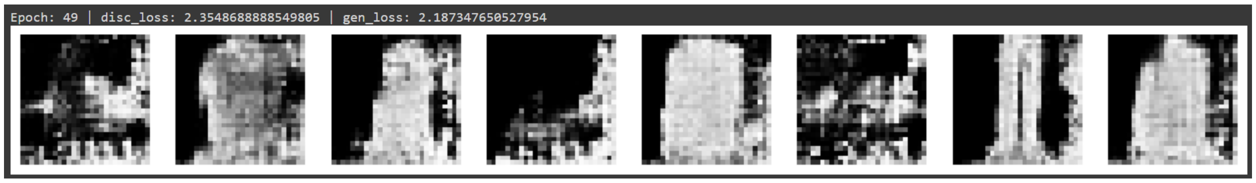
<u>Unrolled GAN-</u>



```
Epoch: 1 | disc_loss: 2.293459892272949 | gen_loss: 1.7774502038955688
```

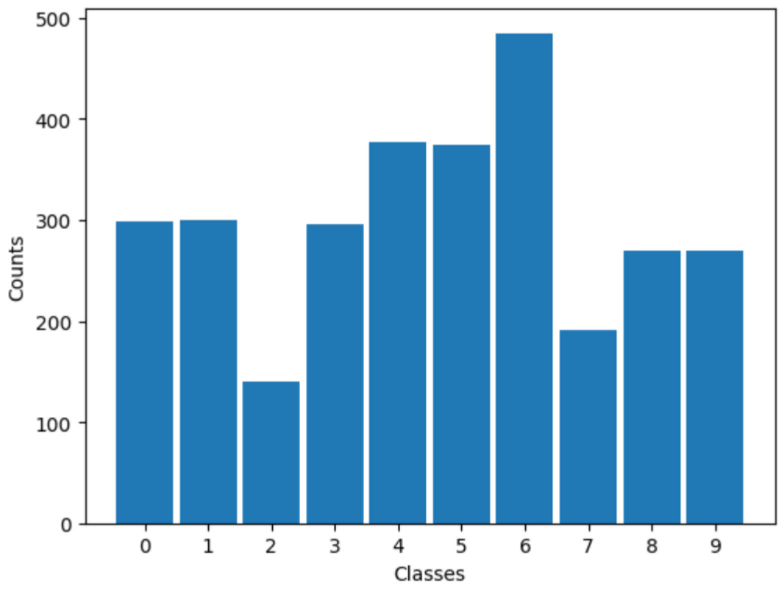


```
Epoch: 23 | disc_loss: 2.388357400894165 | gen_loss: 2.1794865131378174
```



```
Epoch: 49 | disc_loss: 2.3548688888549805 | gen_loss: 2.187347650527954
```

Train an unrolled GAN and plot the histogram from 3000 generated samples. Discuss whether unrolled GAN seems to help reduce the mode collapse problem.



While the class = 6 is higher in number we observe a near uniform distribution because the counts for classes = 0, 1,3, 8, and 9 are nearing 300. The unroll GAN did reduce the mode collapse problem.

This helps because the generator's objective is to minimize the discriminator's output on the generator's data. However, the discriminator is changing during training. If the generator could anticipate how the discriminator will change, it could make a better update. This is where unrolled GANs come in.

In an unrolled GAN, before the generator's update, we perform several steps of the discriminator's optimization (this is the "unrolling"). Then, the generator is updated based on this "lookahead" version of the discriminator. Essentially, the generator can see where the discriminator is heading, not just where it currently is. This helps prevent the generator from being "fooled" by transient weaknesses in the discriminator, thereby reducing mode collapse.