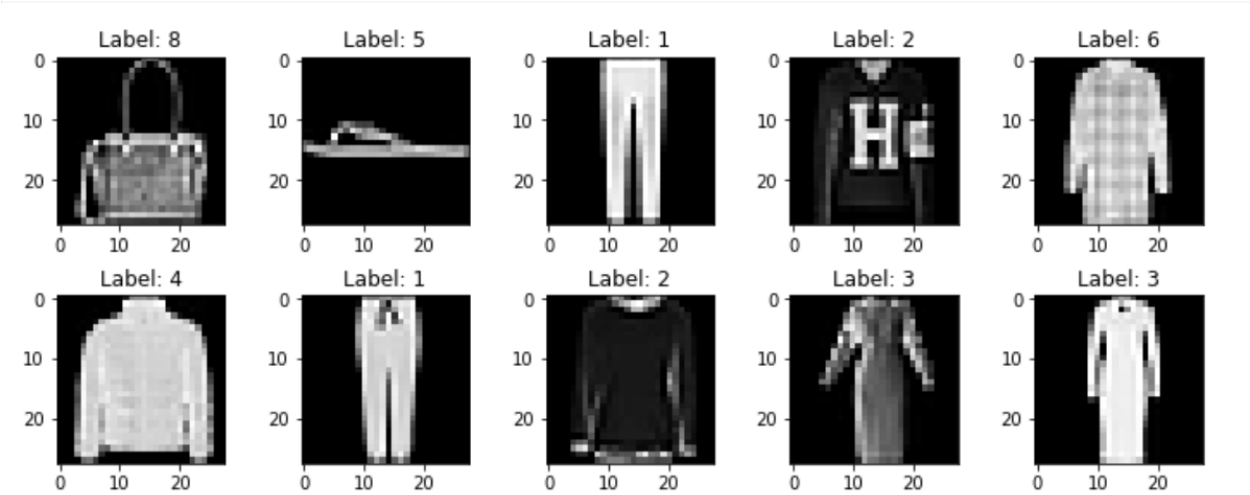
ESE 545 Data Mining - Project
Efficient Classification Techniques and Analysis on Fashion MNIST Dataset

Tejas Srivastava
tjss@seas.upenn.edu

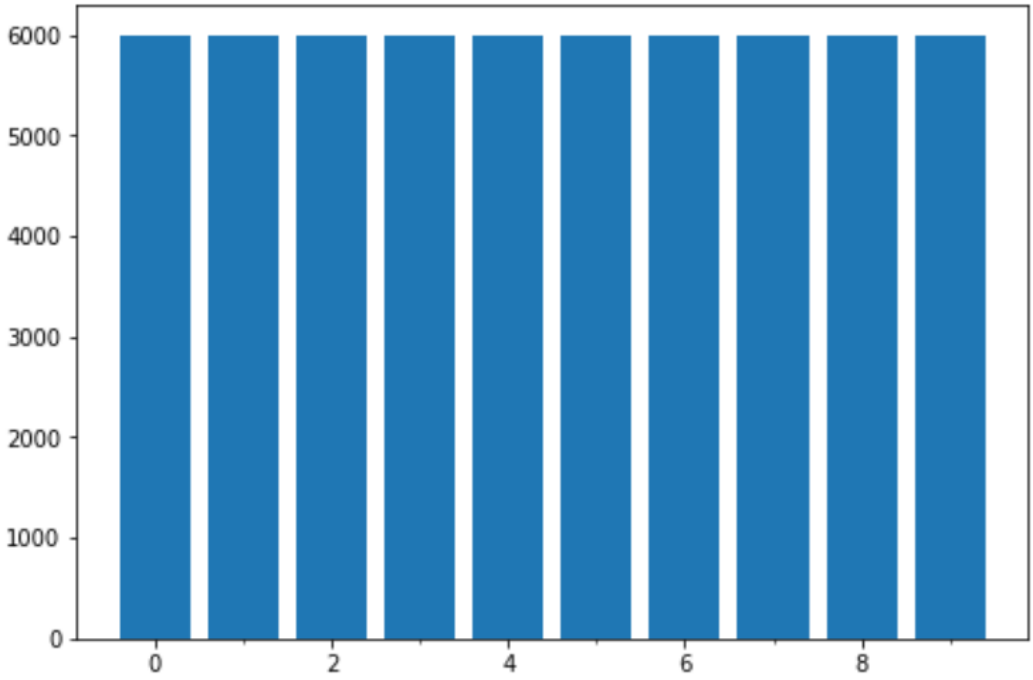
March 29, 2020

Download and preprocess the data

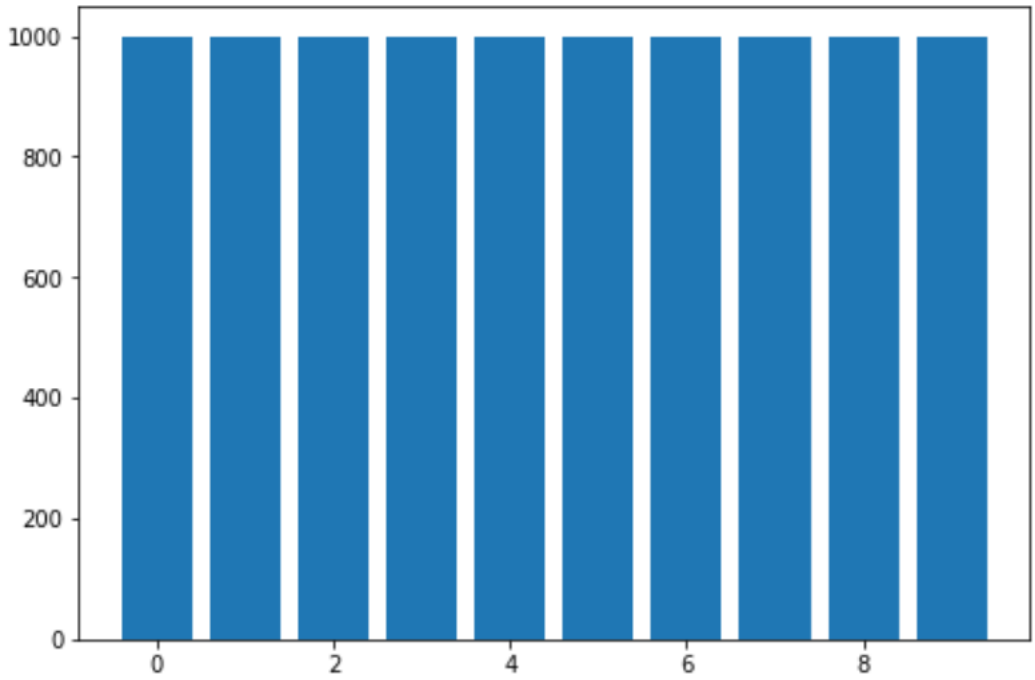
1. Loaded the given numpy data as numpy arrays.
2. There were 60,000 train images and labels, and 10,000 test images and labels. Each image was a single channel(BW) image of the dimension (28x28).
3. Below are some images from the dataset.



4. There was no class imbalance, infact the data was well distributed and each class had an exactly equal number of instances in both the training and testing data.
5. Train dataset Class frequency Distribution



6. Test dataset Class frequency Distribution



7. Further, discarded all the images of classes other than the classes 2, 5, 7. For SVM inputs, flattened all the images, appended a column of ones to the dataset take in case

the bias in w (image), and reshaped the labels from (n,) to (n,1) to adjust their shape for matrix multiplication.

8. Also, processed the data for the implementation of one vs one multiclass classifier, in which I made separate datasets for classes 2 and 5, classes 5 and 7 , and classes 2 and 7.

9. Also, changed the labels to the following values for each of the classifier.

   - 2 vs 5 classifier: 2: 1, 5: -1
   - 7 vs 2 classifier: 7: 1, 2: -1
   - 5 vs 7 classifier: 5: 1, 7: -1

Using Pegasos to train a classifier on the training images.

1. Ran Pegasos for dataset of the classes 2 and 5. Since the data was already reshaped (flattened, column for bias added, and labels reshaped) no processing was required on the dataset.

2. Further, made a function initPegasos() which is used to initialise the various hyperparameters such as regularization parameter(λ), batch size, and the number of iterations. The function also calculates the limit of norm of 'w' i.e. $\frac{1}{\sqrt{\lambda}}$

3. The function calc_project() is used to calculate the norm of 'w' and if the norm is greater than the limit $\frac{1}{\sqrt{\lambda}}$, then takes the projection of 'w', and returns that.

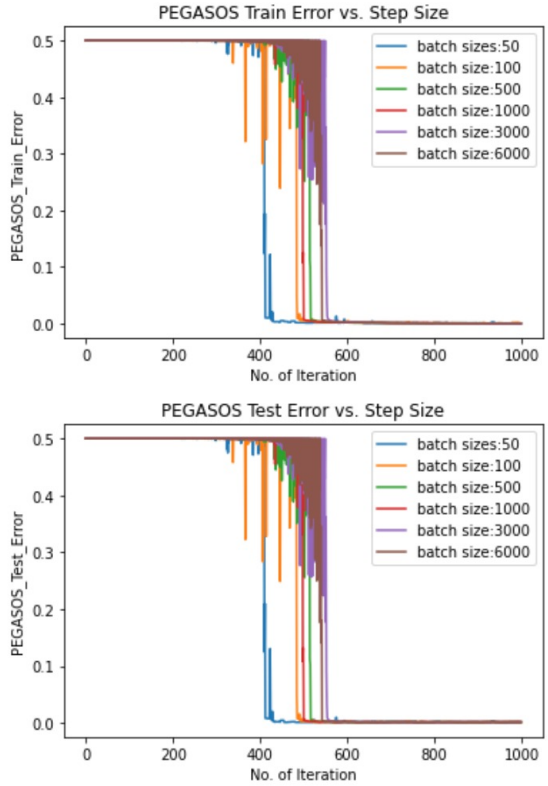
4. Then for the main pegasos function, we initialised the hyperparameters using the initPegasos() function, then randodmly initialized the weights vector, took its projection if its norm was grater than the limit and then iteratively kept updating the weights.

5. In each iteration for weight update, we calculated the learning rate from the current iteration, as $\frac{1}{\lambda * t}$. Further, we randomly selected indices for the minibatch, and for the sampples in the minibatch, calculated the productt of predicted values and real values ((X@w).y). For all the points for which this value was less than 1, i.e. they were misclassified, for these values we further summed their values of (y.X) and used it in the gradient calculation. And then finally updated the weights according to the calculated value and took its projection.

6. In the pegasos algorithm, the batch size ,the regularization parameter and the number of iterations to run were the hyperparameters to be chosen. So, we ran the algorithms for different values of these parameters and got the following results.

7. For Batch size, we ran the algorithm for batch sizes of [50,100,500, 1000, 3000, 6000], and plotted the errors. The results obtained were as folllows.

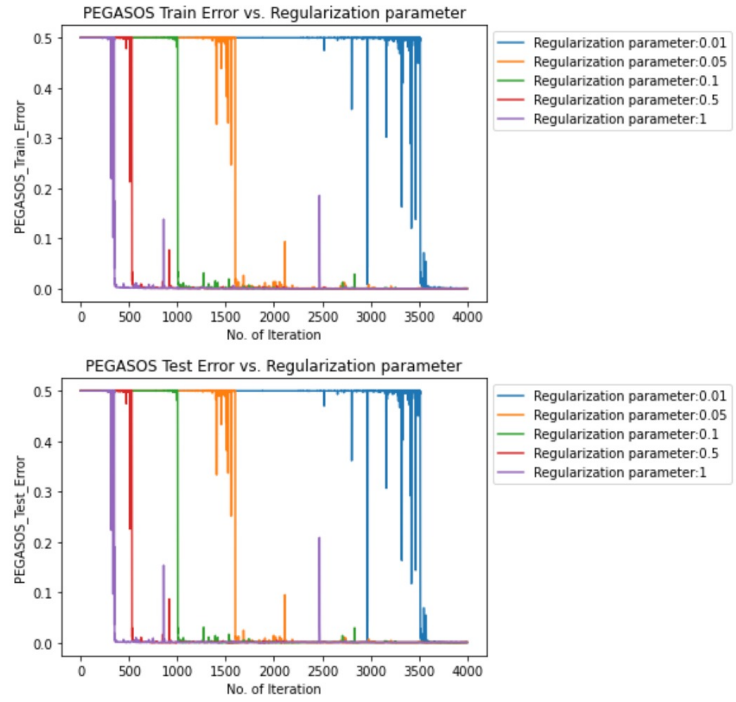
```
Batch size: 50; Running time: 5.719954490661621; Final Test Error.0.0010000000000000009
Batch size: 100; Running time: 5.978993892669678; Final Test Error.0.0010000000000000009
Batch size: 500; Running time: 7.388217926025391; Final Test Error.0.0014999999999999458
Batch size: 1000; Running time: 9.107942342758179; Final Test Error.0.0014999999999999458
Batch size: 3000; Running time: 16.171944618225098; Final Test Error.0.0014999999999999458
Batch size: 6000; Running time: 26.28724765777588; Final Test Error.0.0014999999999999458
```



8. Thus, we see that as we increased the batch sizes, the number of iterations it took for the algorithm to converge increased, and so did the run time of the algorithm. Since in each iteration, we take into consideration a larger mini batch. But the change is not considerable and the results do not vary largely for different batch sizes.

9. For the regularization parameter also, we ran the algorithm for the following values, [0.01, 0.05, 0.1, 0.5,1], and below are our findings.

```
Regularization parameter: 0.01; Running time: 26.01049780845642; Final Test Error.0.0004999999999999449
Regularization parameter: 0.05; Running time: 25.973782539367676; Final Test Error.0.0020000000000000018
Regularization parameter: 0.1; Running time: 25.317251205444336; Final Test Error.0.0010000000000000009
Regularization parameter: 0.5; Running time: 25.083964824676514; Final Test Error.0.0010000000000000009
Regularization parameter: 1; Running time: 24.349424600601196; Final Test Error.0.0014999999999999458
```

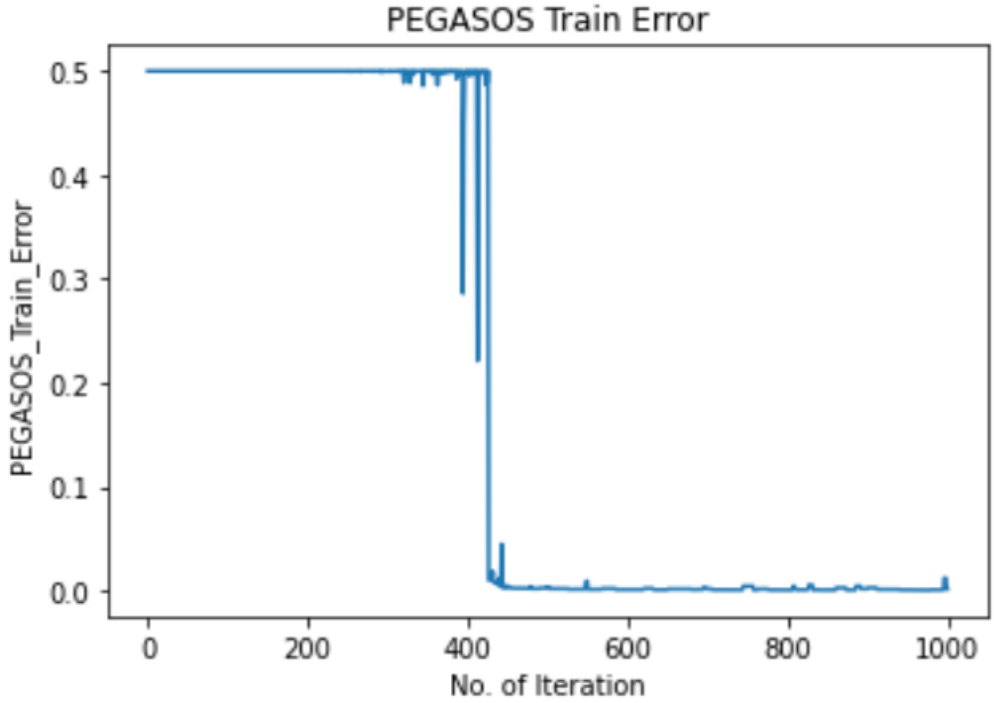


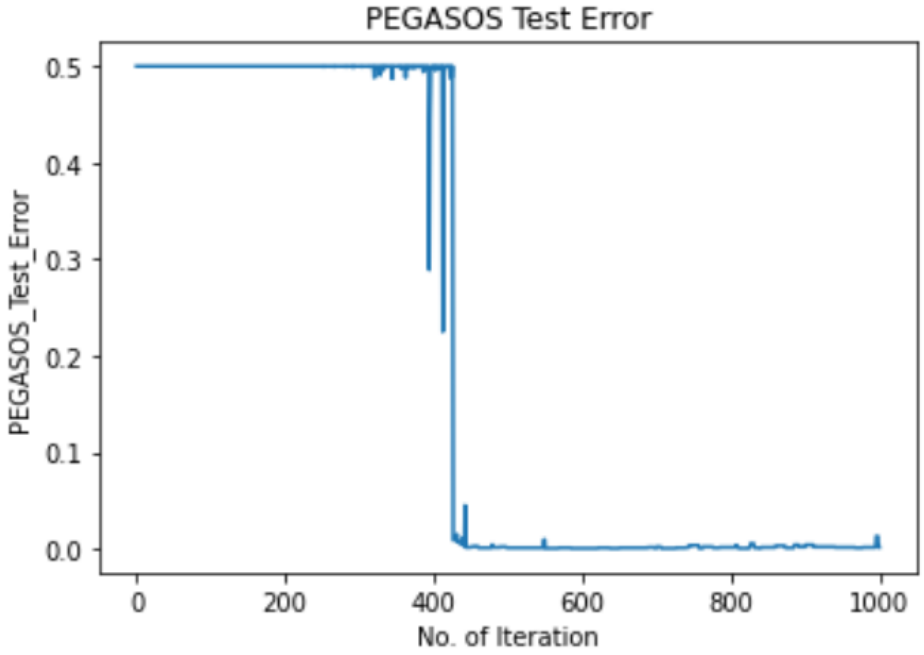
10. We see that the larger our regularization parameter, the faster it converges which is evident from the above plot. This is because very small regularization parameter values indicate high learrning rates which may cause the model to oscillate and not converge quickly .hus for higher values of regularization parameter, the model is performing well as learning rates are comparatively smaller, weights are having lower magnitudes and hence model is able to converge faster.

11. Thus, after adjusting these values we finalised our values as regularization parameter as 1 and batch size as 50, and we got the following results.

    - final train accuracy: 1.0
    - final test accuracy: 0.995

12. The final graphs obtained were as follows:





Using AdaGrad to train a classifier on the training images.

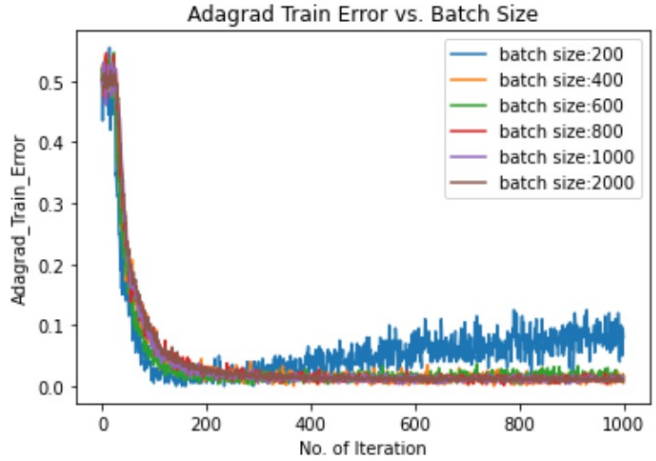
1. Some minor changes were made to the initial Pegasos implementation for Adagrad. The gradient variable G was taken as a vector of zeros of same length of the weights (785), since we will have different gradient valuees for each feature in the input. And a fudge factor of 10^{-6} was added to avoid division by 0.

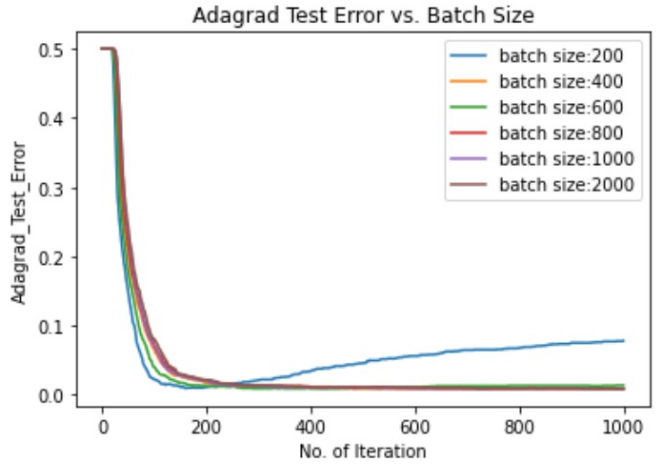
2. And within weight updation step in each iteration, the update was divided by root the G factor, and it was updated by adding the sum of delta^2.

3. Since the learning rate for each feature differs here and depends upon the frequency of the occurrence of the feature, this performs better then Pegasos.

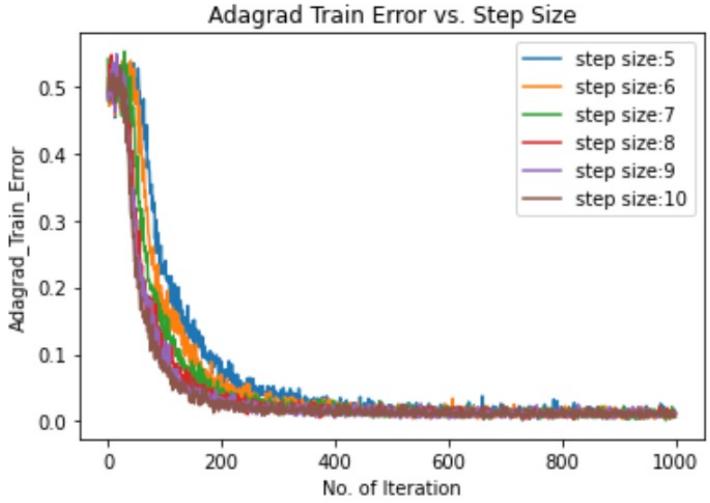
4. Again for deciding the hyperparameters, we plotted the training and testing plots for different values of hyperparameters. Below are our results and findings.

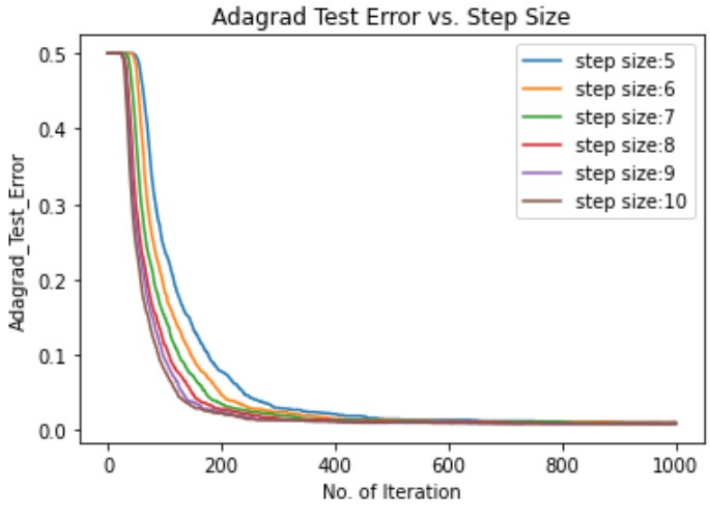
```
Batch size: 200; Running time: 7.050176382064819; Final Test Error.0.078
Batch size: 400; Running time: 7.537499904632568; Final Test Error.0.009
Batch size: 600; Running time: 8.404783248901367; Final Test Error.0.0135
Batch size: 800; Running time: 9.007445096969604; Final Test Error.0.0085
Batch size: 1000; Running time: 9.794593811035156; Final Test Error.0.009
Batch size: 2000; Running time: 14.084344148635864; Final Test Error.0.0085
```





```
Step size: 5; Running time: 9.731626987457275; Final Test Error.0.0105
Step size: 6; Running time: 9.470583438873291; Final Test Error.0.01
Step size: 7; Running time: 9.328916788101196; Final Test Error.0.009
Step size: 8; Running time: 9.207078456878662; Final Test Error.0.009
Step size: 9; Running time: 9.16426157951355; Final Test Error.0.009
Step size: 10; Running time: 9.01799988746643; Final Test Error.0.0085
```





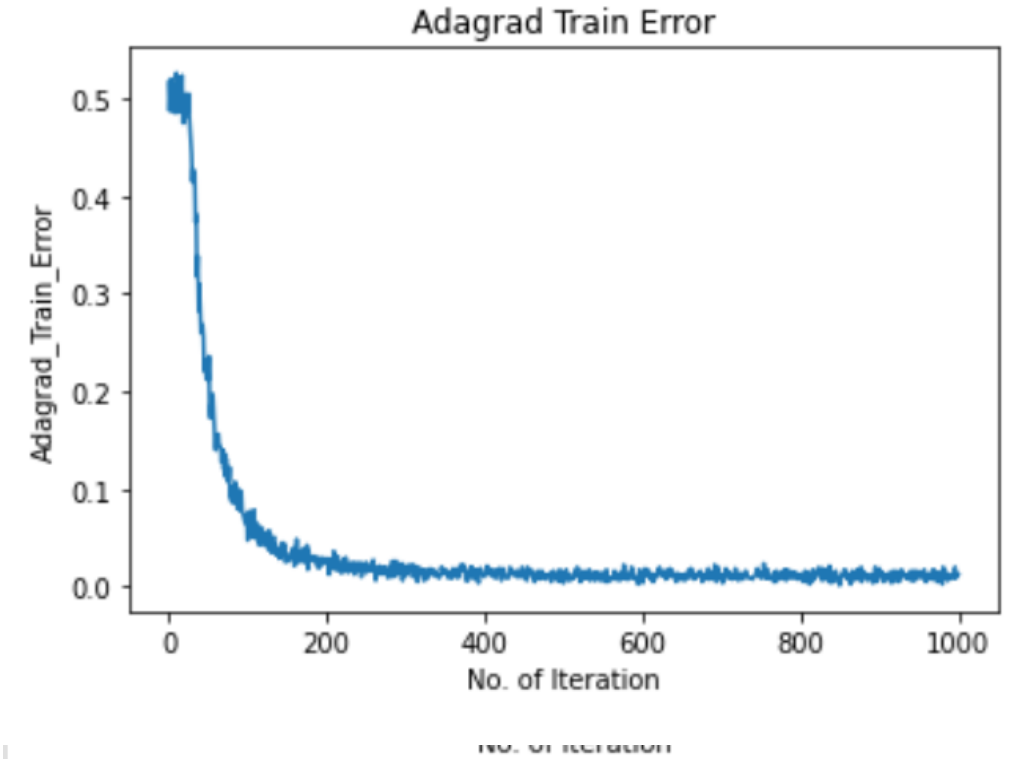
5. We see if we take a small batch size(200), it converges very fast, but as we keep running it over several further epochs it probably overfits and and thus there is an increase in the error. So we should choose a considerable batch size, for our final implementation we chose the batch size as 400.

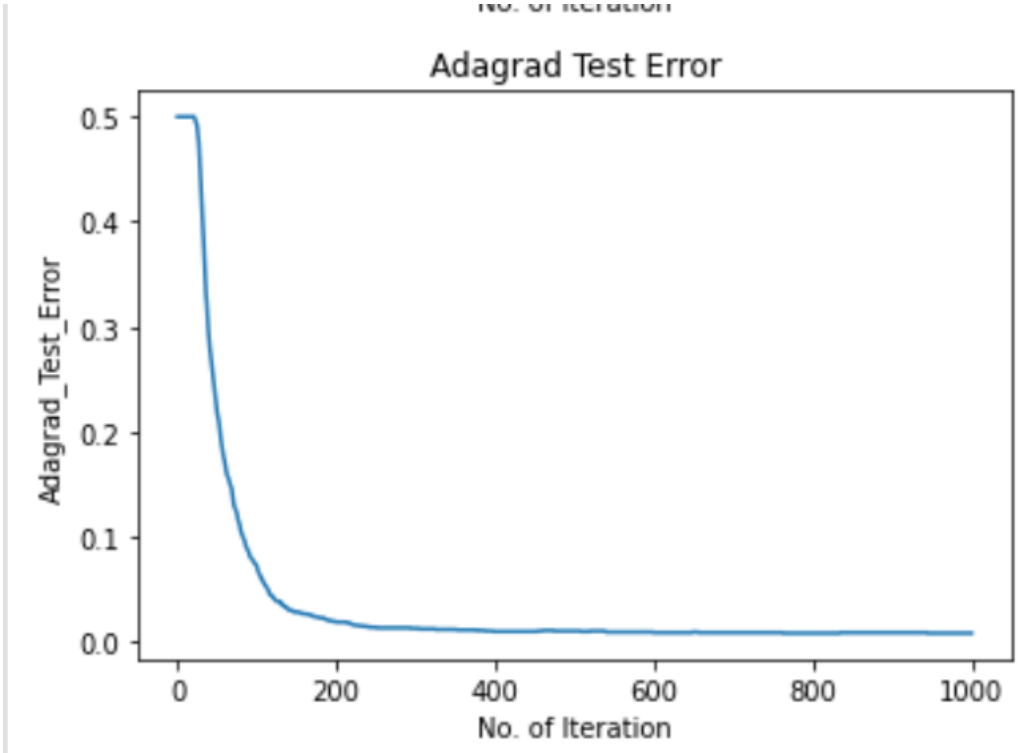
6. Also, we note that the larger the step size the faster it converges.

7. Thus the performance of Adagrad remained almost same for the various hyperparameters and there were minor differences in the number of iterations to reach convergence.

8. But the number of iterations required for Aagrad to converge were much less than that of Pegasos and this was mostly because Adagrad was able to learn rare and informative features due to the modification to the learning rate and hence converged faster.

9. The final train and test plots were as follows:





10. And we obtained a final train accuracy of 0.999 and train accuracy of 0.99775.

Comparison of Pegasos with adagrad We observed that Adagrad performs much better than Pegasos as Adagrad converged faster and with a smaller regularization parameter. This was because in Pegasos, all the features were treated equally and were learned at the same rate becuase of same eta. Also, the learning rate decreased with time and had a combined weight for all the features, thus resulting in possible loss of information as weights of some rare features which might be more informative, were not learned equally well. All the features were given equal importance which made the model bias towards frequently occurring features. Whereas in Adagrad, this drawback was improved by adapting the learning rate based on the gradient values of each features. Thus the model was able to learn equally from rare and informative features as well.

Disadvantage of Adagrad and proposed improvement

1. Adagrad works better than regular stochastic gradient descent as it changes the learning rate η according to each feature separately after each iteration along with the weight vector. Thus making smaller updates for weights of features which occur frequently in the dataset and thus resulting in high learning rates for these features, whereas for features which do not occur very frequently, large updates are done. Thus increasing their learning rates. η However, the problem may arise as the magnitudes of these gradients get larger, because in each iteration w is updated as follows.

   $$w = w - \delta_t * \eta_t / \sqrt{G}$$

   and we additively update this 'G' as

   $$G = G + \delta_t^2$$

   Thus this value of G always keeps increasing because δ_t^2 is always positive, thus resulting in the shrinkage of learning rate as G becomes sufficiently large and largely slowing down the learning.

2. One way to solve this problem is that instead of summing up all of the past gradients , we can take the sum of gradients only till a specific point in the past. This will ensure that G does not increase very much, This can be achieved by tweaking the update rule of G as follows:

   $$G = \alpha G + (1 - \alpha)\delta_t^2$$

   where α takes values between 0 and 1. And keeping the weight update rule as it is. i.e.

   $$w = w - \delta_t * \eta_t / \sqrt{G}$$

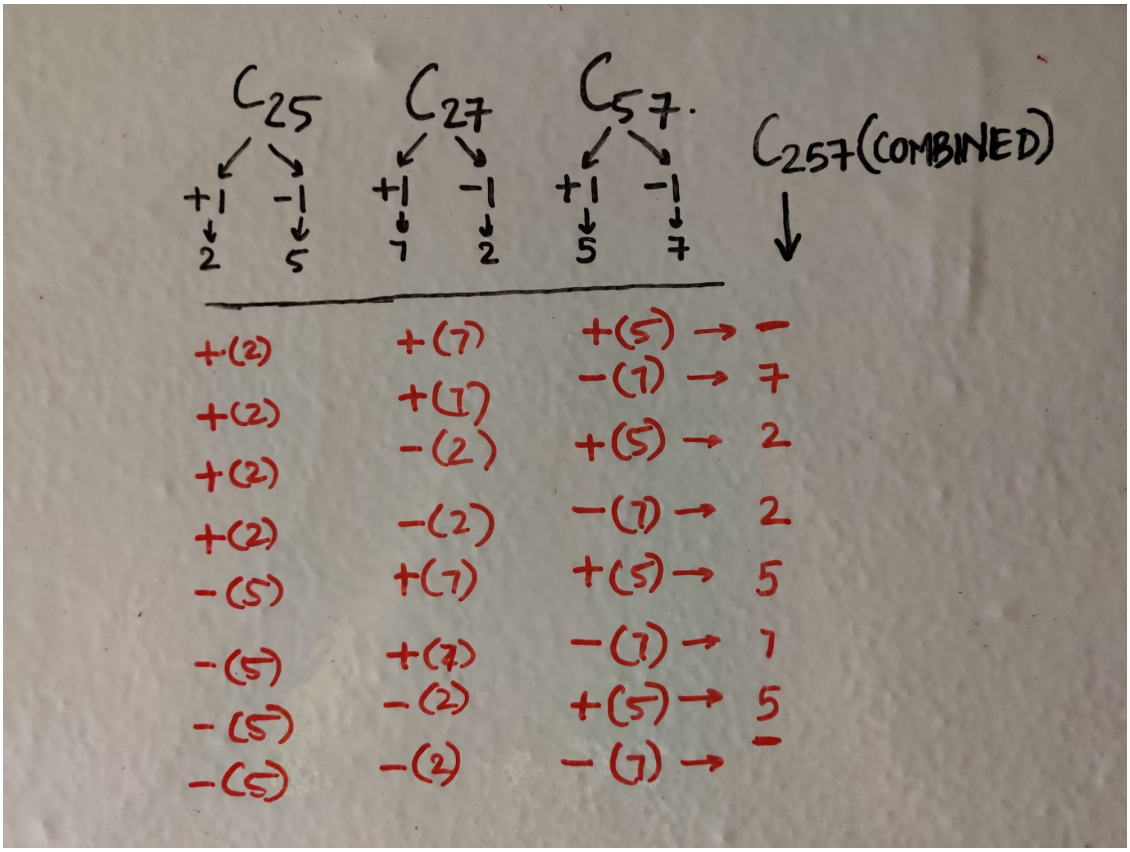
   Thus, by using this approach, we accommodate the learning rates of certain previous gradients along with the square of the new gradient values in the learning rate, which will ensure that the value of G does not shoot off to very high values, and also maintains the adaptive nature of Adagrad algorithm for rare and informative features.

Multiclass classification using binary classifiers by the one vs one approach

1. Here, we train ${}^mC_2$ binary classifiers on our training dataset, and further use majority voting of the classifiers to classify each example as belonging to one of the 'm' classes. For this case m=3, thus we had to train only three classifiers.

2. Thus as mentioned already in the first part, the while dataset of classes 2, 5 and 7 was broken into three datasets each consisting only samples from classes 2 & 5,2 & 7, and 5 & 7.

3. Adagrad with the best chosen hyperparameters was used to train all the three classifiers, C_{25}, C_{27}, C_{57} and the results of w@X was checked for each classifier. As we had already labelled them as each of the class as +1 and -1 already (see problem 1), we had three classifiers predict a certain class for each example, and the following cases of values were possible for any given data example.



Here each sign represents the sign of X@w for each classifier thus giving the corresponding prediction(written in brackets) for the mentioned classifier.

In the six cases, the final prediction was made by taking the simple majority within the predictions if the three classifiers which is evident from the above diagram. But in the first and the last case, where each of the three classifiers give predict classes, the decision of the classifier with the maximum of the modulus value of (X@w) was taken as the final decision as it was most confidently able to classify the given data point and thus had the point farthest from its hyperplane.

4. The final accuracies and the training plots for the multiclass classification were as follows:

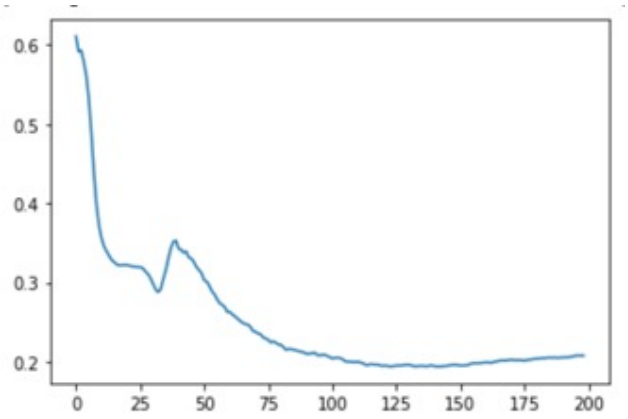


Figure 1: Training plot Multiclass classifier

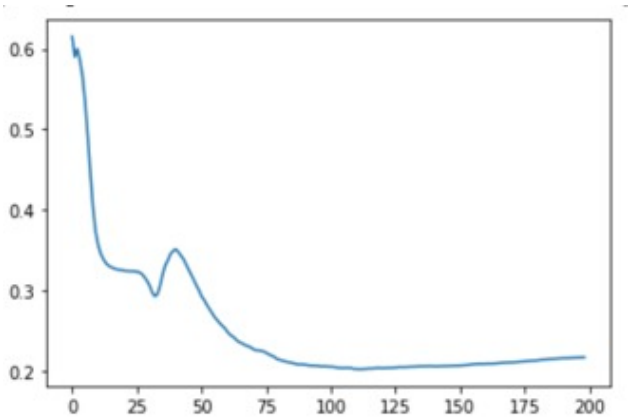


Figure 2: Testing Plot Multiclass Classifier

5. The best train accuracy for the multiclass classifier was 83.889% and the test accuracy was 82.73%. We found that these results were unsatisfactory since the instances from class 5 and class 7 were not entirely seperable. Thus it was difficult to achieve a high accuracy, in the order of accuracies obtained for binary classifications, and thus SVM might not be the best method to perform this multiclass classification.

Using a CNN for multiclass classification

1. Here, we used PyTorch to build a Convolutional Neural Network to classify the images. The model which was used was as follows:

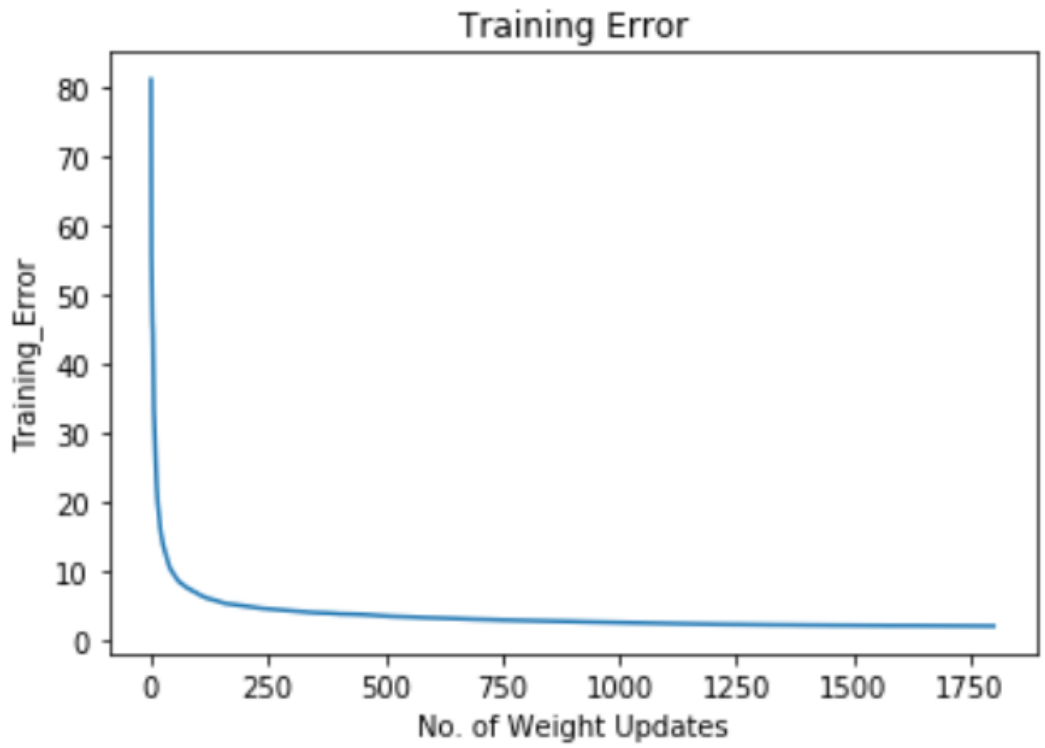
```
class My_Model(nn.Module):
    def __init__(self, num_of_class):
        super(My_Model, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=1))
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=1))
        self.fc1 = nn.Linear(10816, 64)
        self.fc2 = nn.Linear(64, num_of_class)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        return out
```

2. We used Cross Entropy loss as this was a classification problem and Adam optimizer with a learning rate of 0.005 and a batch size of 100, and ran for 20 epochs.

3. We achieved a final training accuracy as 0.99429 and a final testing accuracy of 0.98333.(98.34%)

4. The training and testing plots for losses are as shown.



Comparison of Multiclass Classifiers

1. On comparison of the CNN and the AdaGrad based one-vs-one multiclassifier, we see that the CNN performs better than the latter.

2. This was because in the one vs one classifier was not able to well distinguish between the instances of class 5 and class 7, and it was also clear why, as both the classes were visually very similar, the plot for the binary classification of class 5 vs 7 was quite noisy.

3. Also, the one vs one classifier, could have predicted the wrong value when predictions of each of the classifier was different, and took the decision of the classifier from which the point had the largest distance from the hyperplane, which could be wring in certain cases.

4. And, the CNN works well for the classification of images, because it takes into consideration and learns the spatial properties of the image, and learns the local(low level) features and the high level features present in the image to further classify them. CNN was also able to handle the non linearities present in the dataset whereas the one-vs-one multiclass classifier was a linear classifier so could not classify the data as good as the CNN did.