Simple Neural Networks for Fashion MNIST classification from scratch

Aditya Kulkarni
A53322526
CSE 253
UCSD
adkulkar@eng.ucsd.edu

Shreyas Rajesh
A53324553
CSE 253
UCSD
s1rajesh@eng.ucsd.edu

Abstract

This report has been completed towards partial fulfillment of the course CSE 253 for Programming Assignment 2 part II. We have detailed in this report the implementation of a simple multi-layer perceptron neural network from scratch and provided the results of training this network on the fashion MNIST dataset for a classification task. We have further compared and analysed various activation functions, regularisation techniques and network architecture to fully understand the effect of each of these model design parameters. The best performance we obtained for a single(hidden) layer neural network was 84.51% and was 82.91% on a two layer neural network. Each experiment has been described in detail through the report.

1 Introduction

In this assignment we attempt to understand how to implement and train a simple neural network from scratch. We choose a single/two layer neural network and implement each of the layers along with their corresponding gradients for backpropagation by hand. We further analyse the effect of various network design and hyperparameter choices on the results and detail why we expect the behaviours observed. The report is structured to explain the implementation of the network first for a base case with flexibility to changes of network and hyperparameters and then describe our results with each of the various hyperparameters and network choices.

2 Backprop implementation

We start the implementation of the backprop for this network from the provided starter code and implement each layer in independent classes each with forward and backward methods for the forward and backward pass through the respective layer. We calculate the derivative for each layer by hand and implement the vectorized version of these gradients in the algorithm. We further tested our implementation with the checker file provided for convenience and passed all checks. We also check the implementation of backprop using the technique mentioned in *Question 3 part b* in the assignment. We computed the loss function taking a small value of $\epsilon = 0.01$ and adding and subtracting it from a single weight/bias in the network keeping all other weights and biases the same.

We use these loss values and obtain the gradient using the numerical approximation formula

$$\frac{d}{dw}E(w) \approx \frac{E(w+\epsilon) - E(w-\epsilon)}{2\epsilon}$$

We then compare these values with gradients computed in the backward pass through the network. Below is a table detailing these results. These results are an average over 10 examples chosen one from each class.

Paramter considered	Numerical Approx.	Gradient	Difference
Output bias weight	0.00005661267	0.00005661173	0.0000000094
Hidden bias weight	0.00627625973	0.00627583427	0.0000042546
Hidden to output weight 1	0.0020275710	0.0020275374	0.00000003360
Input to hidden weight 1	0.0195987637	0.0195976769	0.00000108684
Hidden to output weight 2	0.0004186663	0.0004186594	0.00000000691
Input to hidden weight 2	0.11597888878	0.1159697500	0.00000913878

We notice that the numerical approximation and the gradient value agree to the order of $O(10^{-4})$ and infact better in most cases which is within $O(\epsilon^2)$ as expected. The values reported are computed for and averaged over 10 examples, one from each class.

3 Momentum update

In this section we use the vectorized update rules for the input to the hidden layer and the hidden to output layer respectively which are given as follows,

$$w = w + \alpha(x^T\delta)$$

where x are the inputs from the flattened image and δ is as described in the written part of the programming assignment a cumulative of all previous gradients upto that point. w represents the weights from the input to the hidden layer and α is the learning rate.

$$w = w + \alpha(h^T\delta)$$

where h is the outputs of the hidden layer and δ is as described in the written part of the programming assignment a cumulative of all previous gradients upto that point. w represents the weights from the hidden to the output layer and α is the learning rate.

Further, we implement mini-batch stochastic gradient descent, by shuffling the dataset every epoch and splitting the dataset into mini-batches of 128 elements each so as to obtain a middle ground between batch gradient descent and stochastic gradient descent considering each image independently, so as to include the advantages of both techniques of quick updates but better updates. We also implement momentum based SGD, which uses a momentum term in the update rule that helps us push the training the right direction faster. The formula for the update with momentum we have used is,

$$v = \gamma v + (1-\gamma)\alpha\frac{\partial E}{\partial w}$$

$$w = w + v$$

where v is the momentum computed based on the previous step, γ represents the weight we give to the previous momentum, and α is the learning rate.

We have also implemented early stopping to save the best weights as the ones where the validation accuracy is lower and the model hasn't overfit. After, the validation accuracy increases for 5 epochs continuously we stop considering the weights. These best weights are used at test time. We observed the best results with a learning rate of 0.02. We used a single layer network with dimensions [784,50,10], tanh activation and ran it for 100 epochs. As mentioned earlier all results are obtained with a batch size of 1024. Below we have reported our validation and train loss and accuracy curves for the above implementation. The final test accuracy we observed on the early stopping saved weights for best choice of learning rate was, 81.52%.

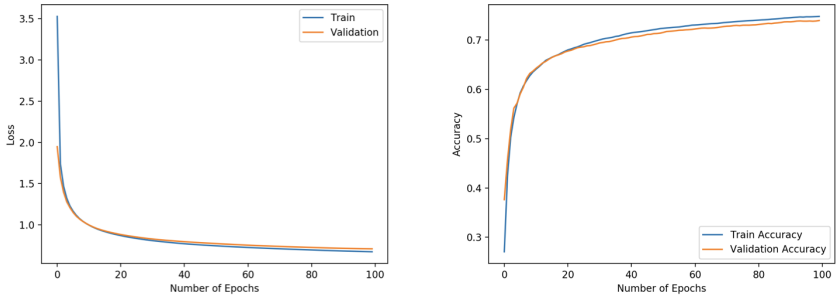

Figure 1: (Left)Plot of loss vs number of epochs for train and validation sets
(Right)Plot of accuracy vs number of epochs for train and validation sets(lr=0.01)

4 Regularization

In this section we implement regularization into the earlier network. The aim of the regularization is to reduce the extent of overfitting of the network that is to ensure that the validation loss doesn't increase after the model is trained. We have implemented regularisation by adding the square of the 2-norm of the weights weighted by the decay value(i.e. L2 penalty) to the loss function and considered its corresponding derivative in the update rule which changes the update rule to include an additional term of the L2 penalty multiplied by the weight matrix. On implementing this and testing with various configurations of L2 penalties, we observed that using a regularizing factor(λ) of 0.02 works best for our network, although the differences are extremely minor as expressed in the images below. Below included are the loss and accuracy plots with the regularization terms in the loss and weight update for regularizations of 0.01 and 0.001 respectively. A total of 550 epochs were used with regularization. We notice that after reaching its lowest value the regularization term

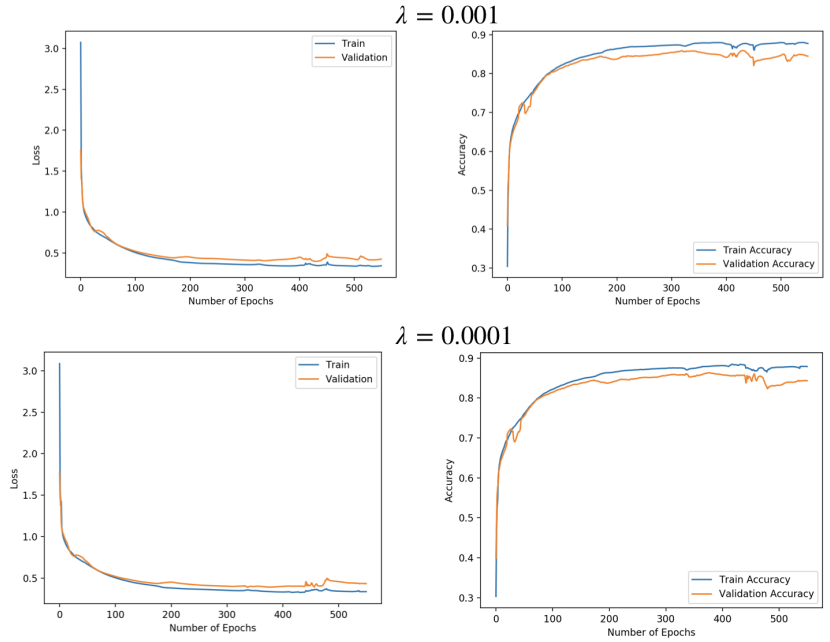

Figure 2: (Left)Plot of loss vs number of epochs for train and validation sets
(Right)Plot of accuracy vs number of epochs for train and validation sets(lr=0.01)

helps to a very small extent the validation loss from increasing too much, meaning that the model isn't overfitting as much as the previous case and the validation loss/accuracy track the training

loss/accuracy better than earlier. We observe that the test accuracy increases to 84.31% since we have avoided overfitting.

5 Activation

We now consider the original network and experiment with various activations for the hidden layer of the network. We consider the momentum SGD however don't consider regularization as expected from the question. We use the same network with layers of size [784, 50, 10] for all the activations. Below are the plots for the various activation functions. These activations do not have a very major

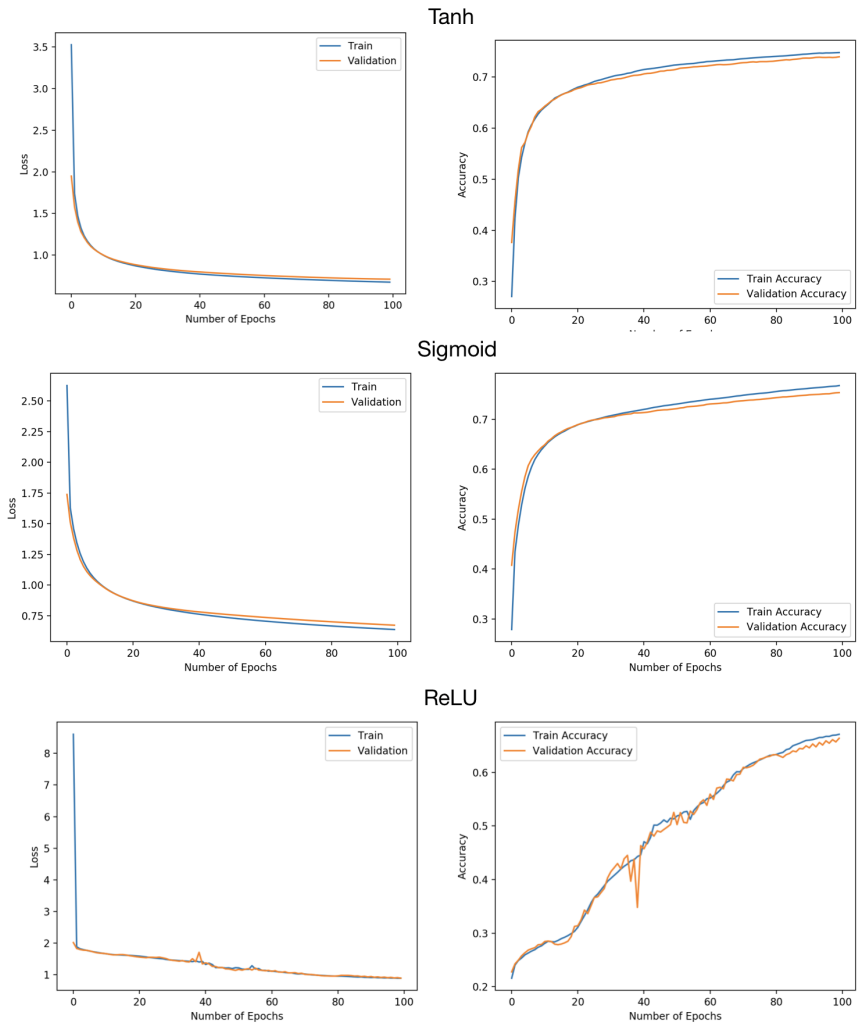

Figure 3: (Left)Plot of loss vs number of epochs for train and validation sets
(Right)Plot of accuracy vs number of epochs for train and validation sets(lr=0.01)

effect on the training. The models still continue to learn as expected. However, the relu activation behaves a little differently from the others. From the loss curves it is quite clear that tanh or sigmoid is the best choice of activation function for this problem as the loss reduces smoothly as well as doesn't overfit to the data within these 100 epochs. We notice that the results on the test data are fairly similar with the various activations.

We notice best results with **sigmoid** activation by a small margin. The test accuracies we obtained were as follows.

Activation type	Test Accuracy
Tanh	84.31%
Sigmoid	84.51%
ReLU	74.12%

6 Network Architecture

In this section we experiment with various network architectures and report our findings. We initially doubling and halving the number of hidden nodes in the single layer network. We observe improvement in one of the cases. As expected, we observe a decrease when we half the number of nodes indicating the model isn't large enough to learn the features well as compared to a 50 unit hidden layer and also observe a dropoff in accuracy when we double the number of nodes which is a clear indication of overfitting of the model. We suspect this is mainly because the dataset is a fairly easy one. However, the loss decreases similarly in both cases. We also consider the case of two layers by finding the number of nodes in the hidden layer that ensures similar number of total parameters. We compute this value to be around 47.15 and hence use two hidden layers with 47 nodes in each one. We notice that there too the accuracy is slightly lower since the model overfits to the data due to the large number of parameters and hence for a simple problem of this nature, it is better to use a single hidden layer and just a single non-linearity rather than two. We have plotted the loss and accuracy curves for these cases below.

The test accuracies are tabulated as follows

Network Architecture	Test Accuracy
784,50,10	84.31%
784,25,10	78.40%
784,100,10	82.83%
784,47,47,10	80.08%

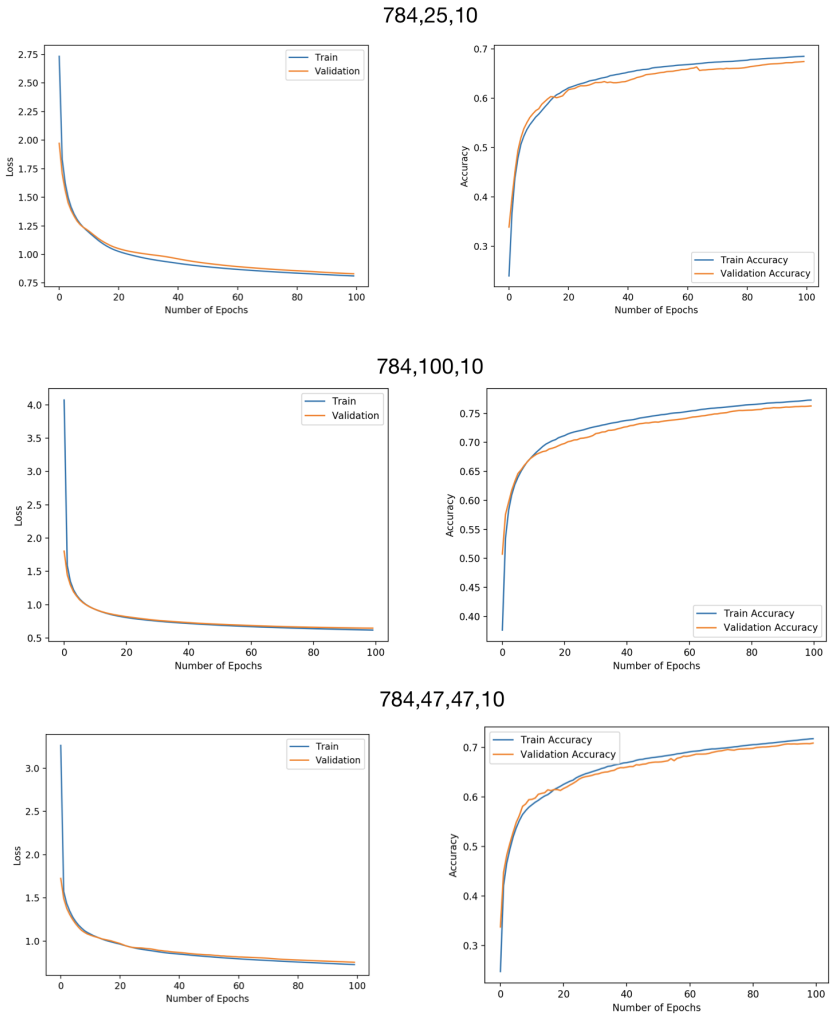

Figure 4: (Left)Plot of loss vs number of epochs for train and validation sets
(Right)Plot of accuracy vs number of epochs for train and validation sets with lr=0.01

7 Conclusion

We studied and understood the behaviour of the network for various changes in hyperparameters of the network and their effect on the performance of the model. As we expected backprop matched the numerical approximation, momentum gave us better convergence than just stochastic gradient descent, regularization helped avoid overfitting, sigmoid activation outperforms other activations for this problem and using a single layer neural network is sufficient for this problem. We observed the best accuracy of 84.51% on the test dataset.

8 Contributions

Shreyas worked on implementation of the backpropagation for the assignment, Aditya worked on the momentum update for the assignment. We then pair programmed to debug and generate the specific cases for the report. Both students contributed equally.