Neural Network for Fashion MNIST Dataset Using Low-Level Functions

Connor Gag
cgag@ucsd.edu

Shivangi Karwa
skarwa@ucsd.edu

Abstract

This project focuses on developing a neural network to predict types of clothing in the Fashion MNIST dataset. We did this using various architectures and hyperparameters, without the use of popular machine learning libraries such as TensorFlow and PyTorch. The main package that we did use is NumPy, which is helpful for tasks such as calculations during backpropagation. Additionally, we added momentum and regularization and tested different activation functions. We experimented with hyperparameter tuning for regularization and found that L2 regularization converged in smoother way than L1 regularization, but their accuracies were about the same. In our experiments with activation functions, we found that ReLU performed better than Sigmoid. Overall, our best accuracy was about 88% on the test data, which we achieved using momentum and regularization.

1 Data visualization

We used the FashionMNIST Dataset, which has different types of clothing.

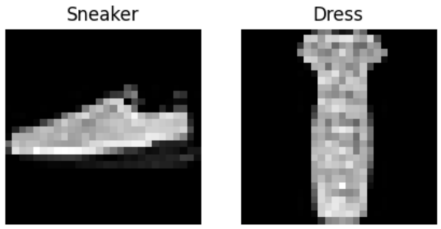


Figure 1: Data Visualization

Data Label	Article of Clothing
0	T-Shirt
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle Boot

Figure 2: Data Labels and Their Meanings

2 Data preprocessing

The FashionMNIST Dataset, consists of 70,000 28x28 images of different types of clothing. One training example consists of the 784 pixel values of an image.

We split the training data into a 80%/20% train/validation split.

We normalized the data by subtracting the mean of the whole image from each pixel, then dividing each pixel by the standard deviation.

Data Set	Number of Examples
Training Set	48,000
Validation Set	12,000
Test Set	10,000

Table 1: Data Set Splits

Statistic	Before Normalization	After Normalization
Mean	0.26914766	9.73137e-09
Standard Deviation (Std)	0.29125977	1.0

Table 2: Comparison of Mean and Standard Deviation Before and After Normalization for One Image

3 Softmax regression

We trained the model using stochastic gradient descent. We added a softmax function as the output activation function in our last layer. This gave us the probabilities of our model's predictions:

$\sigma(a_j) = \frac{e^{a_j}}{\sum_{i=1}^{K} e^{a_i}}$

As for our architecture, we used two layers: an input layer (784 neurons) and an output layer (10 neurons).

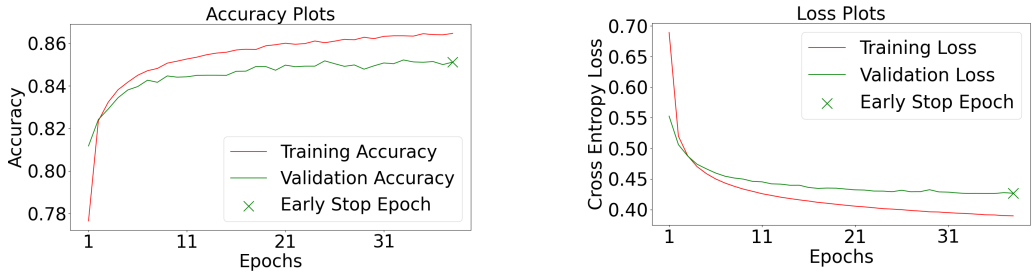
For our hyperparameters, our learning rate was .01 and changing the learning rate to .001 resulted in a slightly worse accuracy in the 100 epochs. Increasing the epochs would allow the model with this learning rate the time it needs to converge.

We set training to run for 100 epochs, but it stopped around 34 epochs because of our early stopping policy (3 epochs). Changing our early stopping policy to 5 epochs resulted in the model training for longer with no increase in accuracy.

We did not yet use regularization or momentum at this point. We used the config_4.yaml.

- The test accuracy is **84.15%**.

- The test loss is 0.45745.



(a) Accuracy

(b) Loss

Figure 3: Loss and Accuracy with Softmax

4 Backpropagation

To test our backpropagation, we implemented a gradient check. This manually computes the gradients by finding the difference in loss between small changes in a single random weight. This loss is then used to compute the gradient. This is the equation used:

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

Epsilon = 10^{-2}

We then compared this with the gradient computed with our backwards pass.

Architecture: 785 (+1 bias) $\rightarrow$ 128 (+1 bias) $\rightarrow$ 10

These gradient tests were done with a trained model.

Weight Type	**Weight Index**	**Gradient (Backprop)**	**Gradient (Numerical)**	**Difference (Abs)**
Output Bias Weight	784, 79	1.831570e-13	-2.775558e-13	4.607127e-13
Hidden Bias Weight	128, 1	0.001381	0.001381	2.291352e-08
Hidden $\rightarrow$ Output Weight 1	304, 81	7.494840e-06	3.170658e-06	4.324181e-06
Hidden $\rightarrow$ Output Weight 2	181, 40	-4.270080e-14	-3.365363e-14	9.047170e-15
Input $\rightarrow$ Hidden Weight 1	44, 7	-5.081859e-07	-5.081944e-07	8.461761e-12
Input $\rightarrow$ Hidden Weight 2	47, 2	-7.732324e-09	-7.732463e-09	1.382703e-13

Table 3: Gradient verification for random weights.

5 Momentum experiments

We used the vectorized update rule and performed mini-batch gradient descent to train our classifier. The classifier was implemented with a neural network consisting of 128 hidden units and momentum integrated into the gradient descent process. The hyperparameters were sourced from *config_6.yaml* file and detailed are as follows -

Learning rate - 0.001
Early stop epoch: 5
Momentum gamma - 0.9

The momentum was incorporated to the backward pass in our neural network model in the Layer class. The update rule for weights W with momentum is typically:

$$v = \gamma v - \eta \nabla W$$

$$W = W + v$$

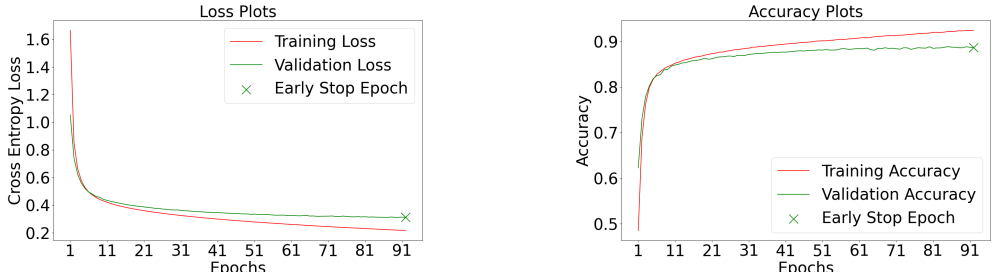
where, v is the velocity vector γ is the momentum η is the learning rate

To analyze the effectiveness of momentum, we conducted experiments using various learning rates and early stopping parameters. The best performance was achieved using the hyperparameters specified above. The final test accuracy of the model was:

Test Accuracy: **88.18%**

Momentum helped accelerate convergence and provided more stable training compared to gradient descent without momentum. The early stopping mechanism helped prevent overfitting.

The following plots illustrate the model's performance through the epochs -



(a) Loss (b) Accuracy

Figure 4: Loss and Accuracy with Momentum

6 Regularization Experiments

We added regularization to the backward pass in our neural network model which involved modifying the gradients of the weights during backpropagation. Regularization is necessary to prevent overfitting. The mathematical formulations for adding regularization are as follows -

L2 Regularization -

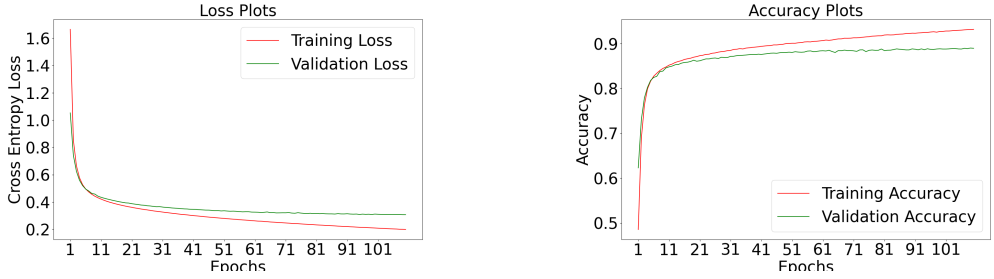
$$\frac{\partial L}{\partial W} = \frac{\partial \text{Loss}}{\partial W} + \lambda W$$

L1 Regularization -

$$\frac{\partial L}{\partial W} = \frac{\partial \text{Loss}}{\partial W} + \lambda \cdot \text{sign}(W)$$

where,
W, represents the weight
λ, represents the regularization strength

A new config file, *config_7.yaml* was created. We also increased the number of epochs by **10%** and we tested on a range of regularization strengths ranging from 1e-2 to 1e-4 for L2 Regularization, the results were as follows:



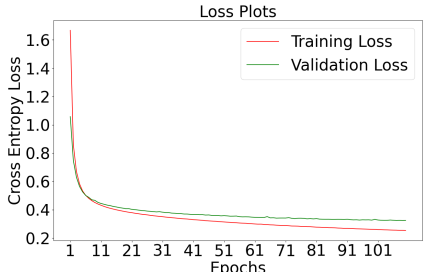
(a) Loss (b) Accuracy

Figure 5: Loss and Accuracy with L2 regularization (1e-4)

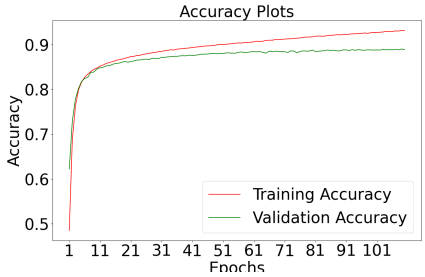
The best L2 performance on test set (**88.52%**) was achieved with λ = **1e-4**. There was an optimal increase in performance as λ was decreased.

Similarly, we also tested the network with L1 regularization with values ranging from 1e-3 to 1e-6 and found that the best test accuracy of **88.53%** with regularization strength of **1e-5**.

Overall, L1 regularization showed a higher sensitivity to λ values, with larger performance variations. But, similar final test results suggests that both methods were effective at preventing overfitting.



(a) Loss



(b) Accuracy

Figure 6: Loss and Accuracy with L1 regularization (1e-5)

7 Activation Experiments

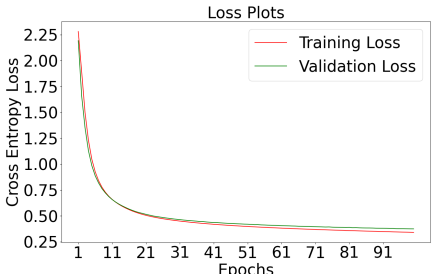
All previous experiments utilized **tanh** as the activation function for the hidden layer. So, we tried alternative activation functions such as **ReLU** and **sigmoid** -

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
ReLU: $f(x) = \max(0, x)$

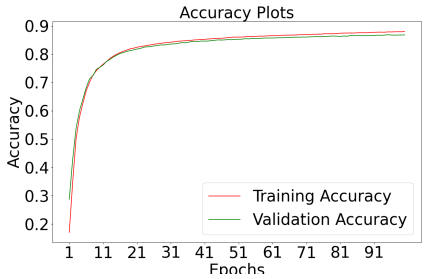
A new config file, *config_8.yaml* was created which included all the hyperperparameters. All the experiments related to the activation functions were performed using that.

We achieved the following results:

7.1 Sigmoid



(a) Loss



(b) Accuracy

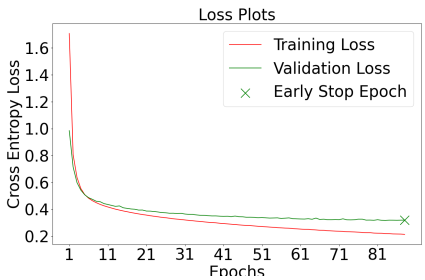
Figure 7: Loss and Accuracy with Sigmoid function

Final test accuracy: **85.83%**

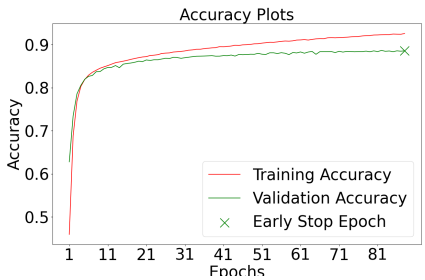
7.2 ReLU

Final test accuracy: **87.91%**

According to the results below, Neural Network with ReLU activation function was able to achieve a higher test accuracy than sigmoid. From the graphs in 8 and 7, it can be observed that ReLU was able to converge faster, as evidenced by the steeper initial loss descent. In addition to this, early stopping was triggered for ReLU which suggests better optimization.



(a) Loss



(b) Accuracy

Figure 8: Loss and Accuracy with ReLU function