Image Classification with Fully Connected Neural Networks

Robin Sarah Chartrand
rchartra@uw.edu

March 3, 2024

Abstract

Neural networks are a powerful tool for many classically difficult computational tasks such as image classification. They are made up of layers of neurons connected by weights which are adjusted in successive cycles to improve the model's performance over time. This performance optimization is done by minimizing the model's loss function by various optimization methods. In this paper a fully connected neural network was built and trained to classify images from the Fashion MNIST dataset. Hyper parameter tuning was performed by adjusting parameters, testing various optimization functions, and using additional techniques to ultimately obtain an optimal testing accuracy of 89.736%.

1 Introduction

Neural networks are a machine learning technique that is powering the forefront of today's technological advances. Modeled loosely on the human brain, these models train themselves on a set of data and over time teach themselves how to perform the task at hand. Once properly trained, a neural network can be used to classify new samples with remarkable accuracy. The Fashion MNIST dataset is a classic dataset used as a benchmark classification test for machine learning algorithms. It is composed of 60,000 images of clothing items all belonging to one of ten classes, as well as 10,000 images to be used for testing of the model. In this study a fully connected multilayer neural network was trained to classify each image and multiple performance boosting methods were analyzed to see their effect on the model's performance.

2 Theoretical Background

Neural networks are interconnected layers of neurons which perform classification on data and then optimize their own performance to increase their accuracy over many successive cycles known as epochs. Before training begins, typically the dataset's features and labels are divided into three subsets: a training set, a validation set, and a testing set. These subsets can also be further divided into batches which will each be iterated through during an epoch and then their results are averaged to produce that epoch's result. The training set is used by the model to train itself to be able to accurately label the samples using the training features. The validation set is used at the end of each epoch to test the model's current ability to classify new data not used in the training process. The testing set is used at the very end of the training process to test the accuracy of the final model. This methodology allows us to monitor the training process as well as counteract overfitting by ensuring the model is still applicable to new data.

Neural networks themselves are structures made up of layers of individual computational units known as neurons. Each neuron first takes in input from the neurons in the previous layer and then multiplies that input by a set of weights and adds a bias term. It then applies an activation function to the input data. There are many options for this function, but they all have a similar step like shape beginning at one value and transitioning to another at a certain point with varying degrees of smoothness. In this study the ReLU activation function was chosen for all neurons (Figure 1). If the combined input is above a certain threshold, the neuron activates and passes the data on to the next layer. If not, then no data is passed

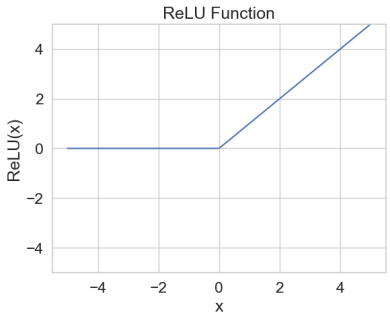

Figure 1: The ReLU activation function.

forward. Neurons are grouped together in layers, which are combined to form a stack of an input layer, some number of hidden layers, and an output layer.

To train the model, the training features are propagated through the neural network until they reach the output layer. Loss, or the model's current accuracy, is calculated via a loss function throughout the training process. There are many ways to calculate loss, though they all work by comparing the classification labels produced by the model to the known labels of the training set in some way. In this study the cross entropy loss was used. It is geared for working with probabilities as it compares the distributions of the results rather than the values themselves and calculates the loss logarithmically:

$$L(y_{pred}, y_{true}) = -(y_{true}\log(y_{pred}) + (1 - y_{true})\log(1 - y_{pred}))$$

The loss function is optimized by an optimization function to find the weights and biases that will reduce the loss function in the next epoch. The pace at which optimization occurs can be controlled via a learning rate parameter α that should be tuned according to the specific dataset being used. The network is then updated with the new weights and biases through a process known as back propagation. There are many optimization functions available and multiple were tested in this study. The ones used were stochastic gradient descent (SGD), Adam, and root mean square propagation (RMSProp). SGD uses the gradient descent optimization method (shown below) and updates the weights and biases of the model each time the loss is calculated.

$$J = \frac{1}{n}\sum_{i=1}^{n} L(y_{pred,i}, y_{true,i})$$

$$\vec{\omega}_{k+1} = \alpha \nabla_{\vec{\omega}} J(\vec{\omega}_k; b)$$

$$b_{k+1} = b_k - \alpha \frac{\partial}{\partial b} J(\vec{\omega}; b_k)$$

Adam and RMSProp are more advanced versions of SGD that adjust the learning rate during the training process according to the calculated gradients to better minimize the loss [1]. There are many parameters to determine in the aforementioned structure and training process. The adjustment of these parameters as well as the implementation of additional performance improving methods is known as hyper parameter tuning. Some common methods to employ, and the ones tested in this study, are dropout regularization, initialization, and batch normalization. Dropout regularization works by randomly turning off neurons in a layer according to some probability. This can help with overfitting by preventing the network from relying on just a few neurons overly specialized to the training data and forcing the other neurons to also be a part of the classification process [8]. Initialization is the strategic setting of the initial weights for the network in such a way that improves the optimizer's chance of converging to the global minimum. Different initialization methods are known to work better for different activation functions. For example the Xavier normal initialization method is known to work better for hyperbolic tangent and sigmoid activation functions, and the Kaiming He method is known to work better for the ReLU activation function [2]. Finally, batch normalization is a technique that stabilizes networks and prevents the divergence of the gradients towards 0 or infinity. This is especially important for deep neural networks with a large number of layers as each

new layer increases the odds of the gradient vanishing or exploding. It is done by normalizing the outputs of each layer to fix the means and variances of layer inputs which ultimately stabilizes the model [4].

3 Algorithm Implementation and Development

The Pytorch python library [6] was primarily used to build, run, and test the aforementioned neural network methodology. The dataset was downloaded using Torchvision [6]; split into training, validation, and testing subsets using Scikit-learn [7]; and then further split into batches using Pytorch [6] utilities. The model was defined as a class extended from the Pytorch [6] `Module` class with parameters corresponding to the input dimension, output dimension, the number of hidden layers, the number of neurons in each hidden layer, and whether to use various performance optimization methods. The images have 28 x 28 resolution so they were entered into the model as 784 length vectors, and the output was a size 10 vector corresponding to the ten classes possible for the image. The layers were defined in the class initialization process and the `ReLu()` activation function was assigned in the model's `forward()` function. The model was trained by iterating through each training batch, calculating the training loss, performing optimization, and propagating backwards. The average training loss over the batches was calculated using the `CrossEntropyLoss()` function and recorded as the training loss for the epoch. The model was then tested on each validation batch, and the average score was recorded as the validation accuracy for the epoch. This process was repeated for the specified number of epochs and then the final model was tested on each testing batch whose results were once again averaged to get a final score for the testing set. The parameters were adjusted to get a solid baseline performance of 85% accuracy on the training set. Hyper parameter tuning was then performed by further adjusting parameters, testing different optimization functions and initializations, and using methods such as dropout regularization and batch normalization all through their corresponding methods in the Pytorch package [6]. For each test the parameters, training loss plot and final loss value, validation accuracy plot and final validation score, and the test score were each recorded in a Pandas [5] `DataFrame`. The training loss and validation plots were made using the python Matplotlib package [3].

4 Computational Results

A baseline model was trained with the parameters listed in Table 1 and had a final testing score of 85.459% accuracy. The training loss and validation accuracy curves are shown in Figure 2. The curves are both smooth and they converge to pretty good values, however the loss could certainly converge to a lower value and we would like to have a higher testing accuracy if possible.

Optimizer	Learning Rate	Training Batches	Hidden Layers	Hidden Layer Dims.	Epochs
SGD	0.01	106	2	[400, 400]	40

Table 1: Parameters for the baseline model.

To improve the model three optimization functions were tested and compared: SGD, Adam, and RMSprop. Three different learning rates were compared for each optimizer as well. The results are tabulated in Table 2. The Adam optimization function with a learning rate of 0.001 had both the highest testing accuracy and the lowest final loss value. The RMSprop function was a close second however. It makes sense that these methods would perform better because they improve upon SGD by adjusting the learning rate and other parameters throughout the training process. The role of the initial learning rate should not be understated however, as no optimizer worked better than the rest for all learning rates. The Adam optimizer with a learning rate of 0.001 was selected as the best optimization function and used for the remainder of the tests.

When validation loss and training loss are compared for these parameters it is clear that something is off (Figure 3). While the training curve converges towards zero, the validation curve actually increases over the epochs. This is indicative of overfitting. To resolve this, dropout regularization was tested with three different dropout probabilities (Figure 4). Using dropout regularization greatly reduced overfitting, though

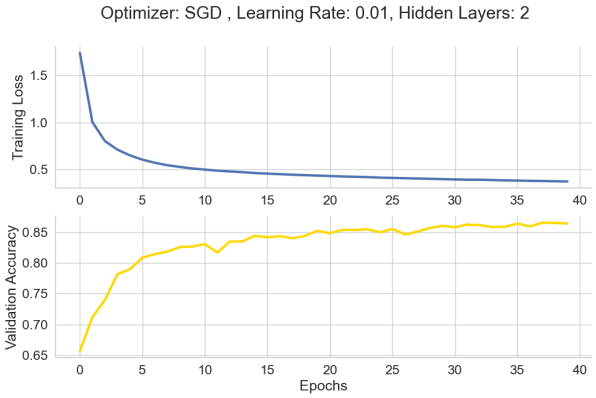

Figure 2: The training loss and validation accuracy curves for the baseline model.

Optimizer	Learning Rate	Testing Score	Final Training Loss	Final Validation Score
Adam	0.001	89.648	5.488	89.608
RMSprop	0.001	89.453	7.001	89.622
SGD	0.100	87.656	17.418	88.318
Adam	0.010	87.246	19.217	88.141
SGD	0.010	85.459	37.299	86.422
RMSprop	0.010	84.570	30.754	85.869
SGD	0.001	76.465	67.611	78.090
Adam	0.100	20.029	208.694	19.359
RMSprop	0.100	16.299	342.450	16.487

Table 2: Success measures for the various optimizers and learning rates.

there was no significant difference between the different dropout probabilities. As such the dropout method was used for the remainder of the tests with the default dropout probability of 0.5.

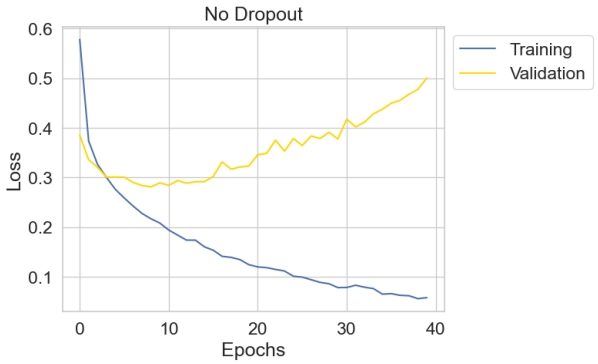

Figure 3: The training loss and validation loss curves for the new baseline model with the Adam optimizer and a learning rate of $\alpha = 0.001$.

Three different weight initialization methods with their default parameters were then tested: random normal, Xavier normal, and Kaiming normal. The results are tabulated in Table 3. The differences were all slight, however Kaiming uniform initialization did have the greatest increase in the testing and final validation score as well as resulting in a lower final loss than without initialization. This makes sense as Kaiming initialization is known to work well for neural networks employing the ReLu activation function [2]. Kaiming uniform initialization was therefore employed in the model and used for the remaining test.

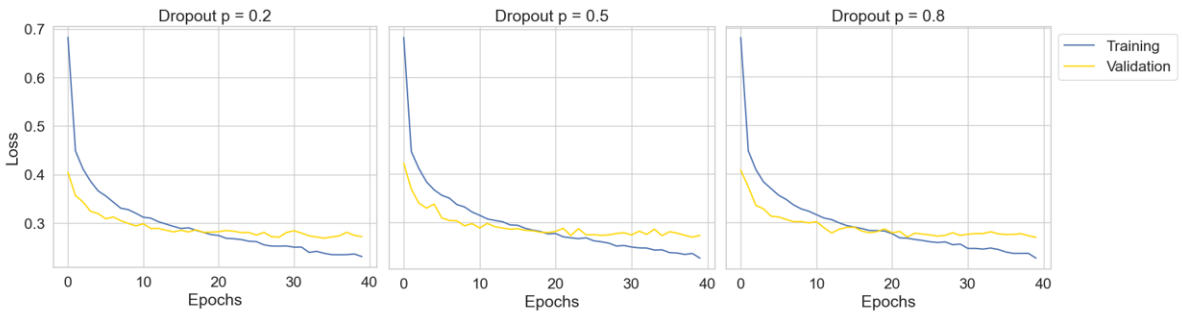

Figure 4: The training loss and validation loss curves for the model using various dropout probabilities.

Intialization	Testing Score	Final Training Loss	Final Validation Score
Kaiming Uniform	89.736	22.308	90.412
Xavier Normal	89.541	22.164	89.864
None	89.258	23.222	90.126
Random Normal	88.252	32.821	89.046

Table 3: Success measures for various initializations as well as for no initialization.

Finally, batch normalization was tested on the model by normalizing the outputs of the initial layer and each of the hidden layers. The results in comparison to the non normalized results are tabulated in Table 4. Batch normalization actually decreased the testing accuracy quite significantly. This may be due to the fact that batch normalization is a method that is greatly helpful in large "deep" neural networks where errors grow over many propagations through the model's layers. This model however only has two layers, so it is unlikely that the gradients will grow out of control in such a short period of time. As such the normalization may just be getting in the way of the model's classification abilities.

Batch Normalization	Testing Score	Final Training Loss	Final Validation Score
False	89.736	22.308	90.412
True	87.734	33.128	89.348

Table 4: Success measures for the model using batch normalization as well as for not using batch normalization.

The resulting final hyper parameterized model therefore has the parameters listed in Table 1 except for it now uses the Adam optimizer with a learning rate of $\alpha = 0.001$. It also uses dropout regularization with a dropout probability of 0.5 and Kaiming uniform initialization for the initial weights. It has the final accuracy measures in Table 5 and it has the validation accuracy and training loss curves in Figure 5.

5 Summary and Conclusions

Using hyper parameter tuning the model's accuracy on the testing set was increased by 4.277% and the validation accuracy reached above 90%. This shows that while neural networks are powerful computational tools, their success is highly dependent on the parameters used which depend greatly on the dataset as well as the architecture of the model. Hyper parameter tuning is therefore an essential part of the model training process. It should be noted however that due to the training time of the model only one trial was done for each parameter setting tested. It's therefore possible that these exact parameters are not indicative of the average performance increase caused by the parameter change. In a future study these results could be made more rigorous by conducting multiple model runs at each parameter value and comparing the means and standard deviations of the trial results.

Model Version	Testing Score	Final Training Loss	Final Validation Score
Final	89.736	22.308	90.411
Initial	85.459	37.299	86.422

Table 5: Success measures for the final hyper parameterized model as well as the original baseline model.

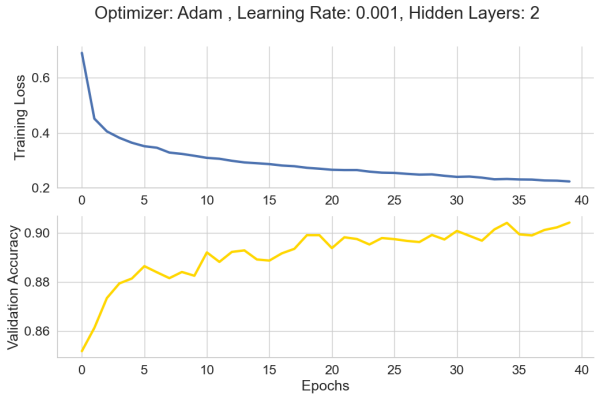

Figure 5: The training loss and validation accuracy curves for the final hyper parameterized model.

6 Acknowledgements

The author would like to thank Professor Shlizerman for his introduction to neural networks and his discussion of their implementation in Python. The author is also grateful to Grace Zhou for her discussion on validation loss curves.

References

[1] J. Brownlee. Gentle introduction to the adam optimization algorithm for deep learning, Jan 2021.

[2] J. Brownlee. Weight initialization for deep learning neural networks, Jan 2021.

[3] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[4] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[5] T. pandas development team. pandas-dev/pandas: Pandas, Feb. 2024.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.

[7] F. Pedregosa, G. Varoquaux, and et. al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] A. Tam. Using dropout regularization in pytorch models, April 2023.