AMATH 582: HOMEWORK 4

KATIE WOOD

Applied Math Department, University of Washington, Seattle, WA
kvo24@uw.edu

Abstract. We train Fully Connected Deep Neural Networks to classify images in the Fashion-MNIST data set. This data set contains thousands of images of different types of clothing items, which due to their greater visual complexity are more challenging to classify than the images of digits in the MNIST data set. We utilize the PyTorch deep learning platform to build our neural network. We evaluate the performance of different optimizers including RMSProp, Adam, and SGD, and perform hyper-parameter tuning to maximize our classification accuracy.

1. Introduction and Overview

Correctly classifying images of clothing items presents a greater challenge than classifying images of handwritten digits. The increased detail and subtlety of each image requires more powerful classification techniques. Recently, a computational data analysis class at the University of Washington was tasked with building a deep neural network to correctly classify as many of the fashion images as possible. The scientists on the project received tens of thousands of gray-scale images of articles of clothing, which they divided into training, validation, and testing sets. They then created a neural network with an adjustable number of hidden layers, neurons in each layer, learning rate, number of training batches, and number of training epochs. By adjusting these parameters, the scientists achieved over 85% classification accuracy at baseline. Lastly, the scientists performed hyper-parameter tuning to evaluate the performance of three different optimizers and of various regularization, initialization, and normalization schemes.

2. Theoretical Background

Neural network models are fundamentally inspired by the human brain. With approximately 10^{11} neurons and 10^{14} connections, the human brain can be thought of as a powerful computational engine. Neural network models mimic what the human brain does by copying the firing behavior of individual neurons. This behavior is defined, mathematically, by one of the following functions [3]:

$$\begin{aligned}
\text{Linear:} \quad & f = x \\
\text{Step:} \quad & f = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases} \\
\text{Sigmoid:} \quad & f = \frac{1}{1+e^{-x}} \\
\text{Tanh:} \quad & f = \tanh(x) \\
\text{Rectified Linear Unit (ReLU):} \quad & f = \max(x, 0) \\
\text{Leaky (parametric) ReLU:} \quad & f = \max(x, ax) \quad \text{where} \quad a > 0
\end{aligned}$$

In this assignment, we use ReLU activation for all neurons. ReLU's wide usage in image recognition problems makes it an appropriate choice for our FashionMNIST data set [2].

A deep neural network consists of multiple layers of these neurons. Each neuron in a given layer computes a weighted sum of the inputs from the neurons in the previous layer. Mathematically, this weighted sum is the result of multiplication by a weights matrix. There is one weights matrix in between each pair of layers in the network. After the weighted sum has been computed, it passes through the neuron's activation function (ReLU, in our case) and on to the next layer. Over the course of the training process, the machine changes the entries in the weights matrices, aiming to increase the accuracy of the final output predictions [3].

Loss functions quantify the performance of neural network models. Minimizing a loss function is the goal of the model during training. In this assignment, we use Cross Entropy loss to optimize our model. This loss function derives from the Kullback-Leibler (KL) divergence metric, also known as relative entropy, which is given by

$$D_{KL}(\hat{p}_{\text{data}}||p_{\text{model}}) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}}[\log \hat{p}_{\text{data}}(x) - \log p_{\text{model}}(x)]$$

and which measures the degree of dissimilarity between the empirical distribution $\hat{p}_{\text{data}}$ defined by the training set and the model distribution p_{model}. Since only the term involving the p_{model} depends on the model, to minimize KL divergence, it suffices to minimize just that term. Hence, our model will attempt to minimize the quantity

$$-\mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}}[\log p_{\text{model}}(x)]$$

over the course of training [1].

Except for linear models, however, the global minimum of the loss function cannot in general be derived analytically. Instead, we use an iterative process called Gradient Descent to approximate the minimum. Mathematically, the gradient is a vector containing the derivatives of a function with respect to each of its variables. Since the gradient points in the direction of steepest ascent, we follow the reverse direction of the gradient in order to descend, over the course of training, to as low a point as possible on the loss function. We represent Gradient Descent by the equations

$$\vec{w}_{k+1} = \vec{w}_k - \alpha\nabla_{\vec{w}}J(\vec{w}_k; b)$$
$$b_{k+1} = b_k - \alpha\frac{\partial}{\partial b}J(\vec{w}, b_k)$$

which updates the next weights vector $\vec{w}_{k+1}$ and bias term b_{k+1}.

One caveat to this approach is that computing the full gradient becomes prohibitively expensive once the data set becomes huge. Thus, a sampling-based approach was devised, known as Stochastic Gradient Descent. This approach draws a minibatch of samples $\mathbb{B} = \{y^{(1)}, ..., y^{(m')}\}$ uniformly at random from the training set. The computational speed-up comes from holding m' fixed while the size of the training set m grows. The estimated gradient is then [1]

$$\boldsymbol{g} = \frac{1}{m'}\nabla\sum_{k=1}^{m'} L(y^{(k)}, \hat{y}^{(k)})$$

In this assignment, we utilize Stochastic Gradient Descent in our first set of attempts to configure our FCN to baseline performance ($> 85\%$ test accuracy).

3. Algorithm Implementation and Development

Our computational approach relied primarily on Python's deep learning platform PyTorch. In addition, we utilized NumPy for scientific computing, tqdm to see our training progress bar, and the train_test_split method from Sci-Kit Learn to help divide our data into training, validation, and testing sets. Visualization was provided by Matplotlib.

4. Computational Results

After initial tuning, we found the following baseline (using the SGD optimizer):

$$\text{number of hidden layers} = 2$$
$$\text{neurons per hidden layer} = 400$$
$$\text{learning rate} = 0.03$$
$$\text{number of epochs} = 40$$
$$\text{number of training batches} = 106$$

The baseline model trained in approximately 8 minutes using Google's T4 GPU, and achieved a testing accuracy of 87.2% (±2.1%). Below, we present the training loss curve and the validation accuracy curve for the baseline model:

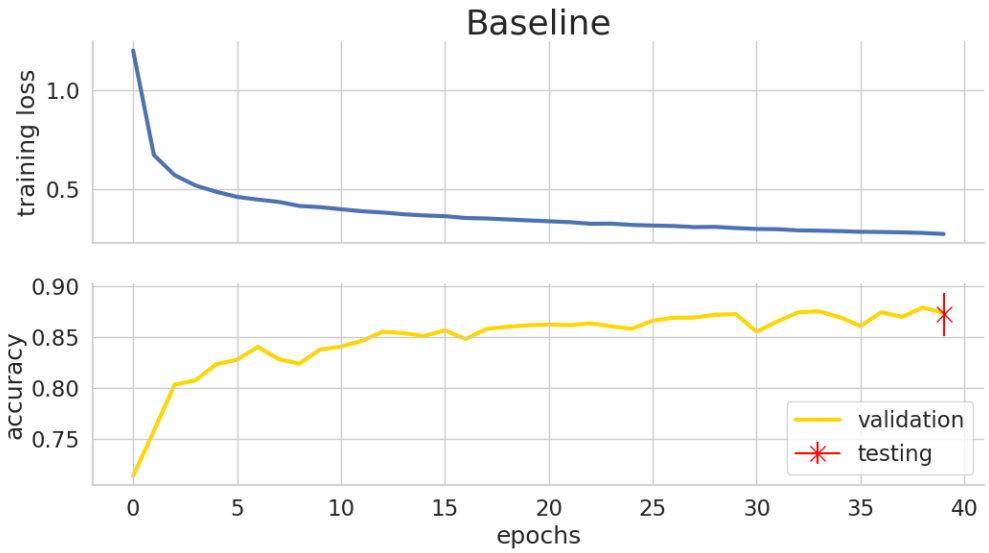

Figure 1. FCN model performance at baseline

We then varied the optimizer between RMSProp, Adam, and SGD, and varied the learning rate between 0.1, 0.01, 0.001. We found that for our data set the Adam optimizer with learning rate of 0.001 performed the best of all options considered. Interestingly, the performance of Adam and RMSProp improved as the learning rate went down, while the reverse was true for SGD. The improvement we see with Adam here, though slight, could be due to Adam's faster convergence as compared to SGD.

lr	RMSProp	Adam	SGD
0.1	21.5 ± 2.5%	9.8 ± 2.3%	88.9 ± 2.5%
0.01	81.2 ± 3.1%	87.9 ± 2.8%	85.2 ± 2.7%
0.001	88.4 ± 2.3%	89.6 ± 2.0%	76.9 ± 3.0%

Next, we performed dropout regularization, varying the dropout probability between 0, 0.2, and 0.4.

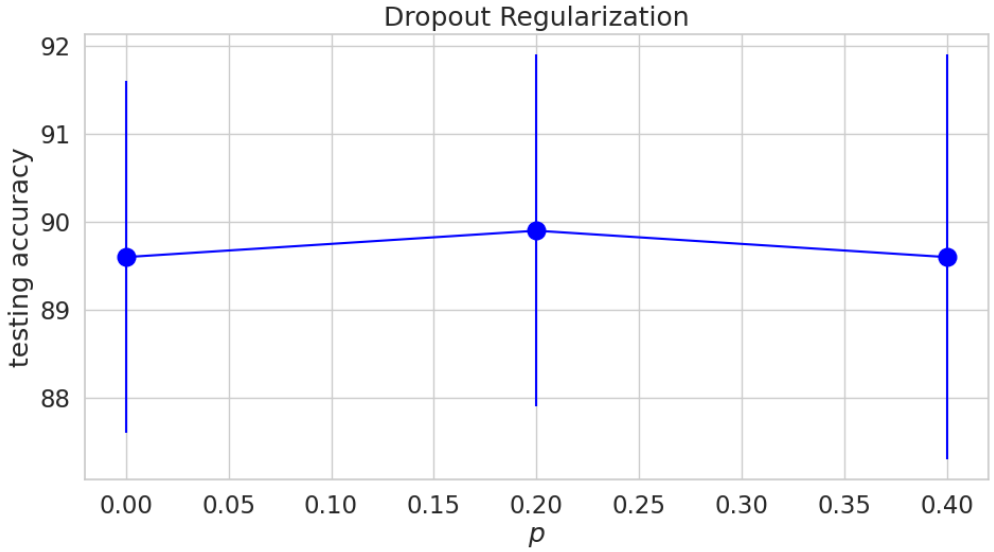

FIGURE 2. Dropout regularization with three different p values

Finally, we tried different initializations, including Random Normal and Xavier Uniform. Like Dropout regularization, Xavier Uniform showed virtually the same performance as before. Lastly, with Random Normal, we actually observed a decrease in both validation and testing accuracy. Training loss curves decreased monotonically in all trials.

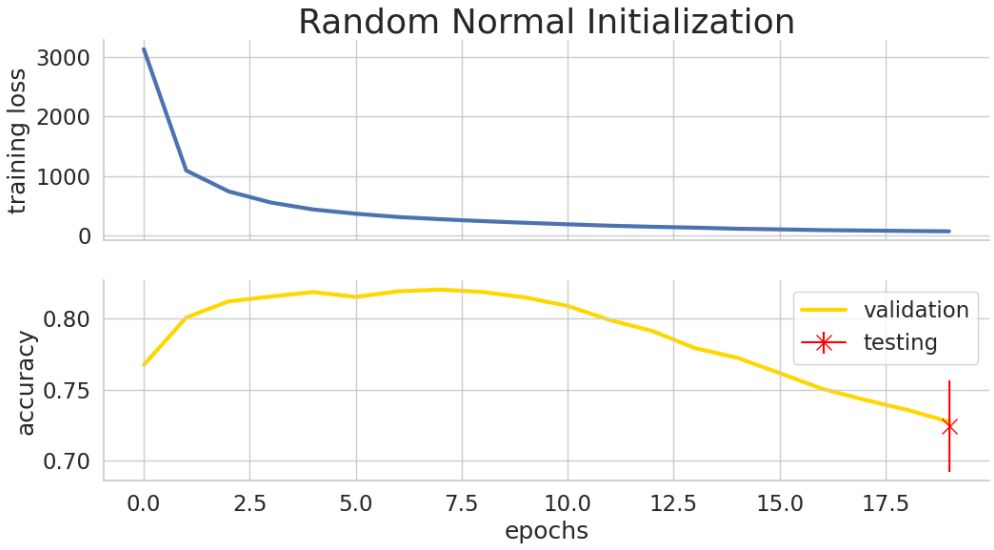

FIGURE 3. FCN model performance at baseline

5. Summary and Conclusions

In conclusion, the highest accuracy we were able to obtain was ... with ... Our inability to significantly increase baseline performance through hyper-parameter tuning could indicate a bug in the code, or that the model is already performing as well as it can.

In a future project, we could continue to tune our hyper-parameters to achieve testing accuracy of $> 90\%$. We could also apply this same neural network to MNIST, aiming for an accuracy of $> 98\%$. Lastly, we could compare the Sci-Kit Learn classifier that was most successful with MNIST to FashionMNIST, then compare its performance to our FCN model.

Acknowledgements

The author is thankful for the Homework 4 recitation and the class Discord channel.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning series. MIT Press, 2016.

[2] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[3] M. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.