DEEP NEURAL NETWORKS AND FASHION-MNIST

SAMIP KARKI

Applied Mathematics, University of Washington, Seattle, WA
karki1@uw.edu

Abstract. This project introduces deep neural networks and the various ways to tune the hyper-parameters of a deep neural network. I use the Fashion-MNIST classification task as a toy problem to investigate the effect of various network shapes, different optimizers, the effect of regularization, different initialization conditions, and batch normalization

1. Introduction and Overview

Deep neural networks (DNNs) are one of the most famous algorithms to come out of the field of machine learning. Although relatively simple in concept compared to other neural network archetypes, there many techniques in which the scientist using DNNs can tune their network for greater performance. This report will survey the different hyper-parameters one can tune in a DNN in the context of the Fashion-MNIST classification task.

Date: August 30, 2024.

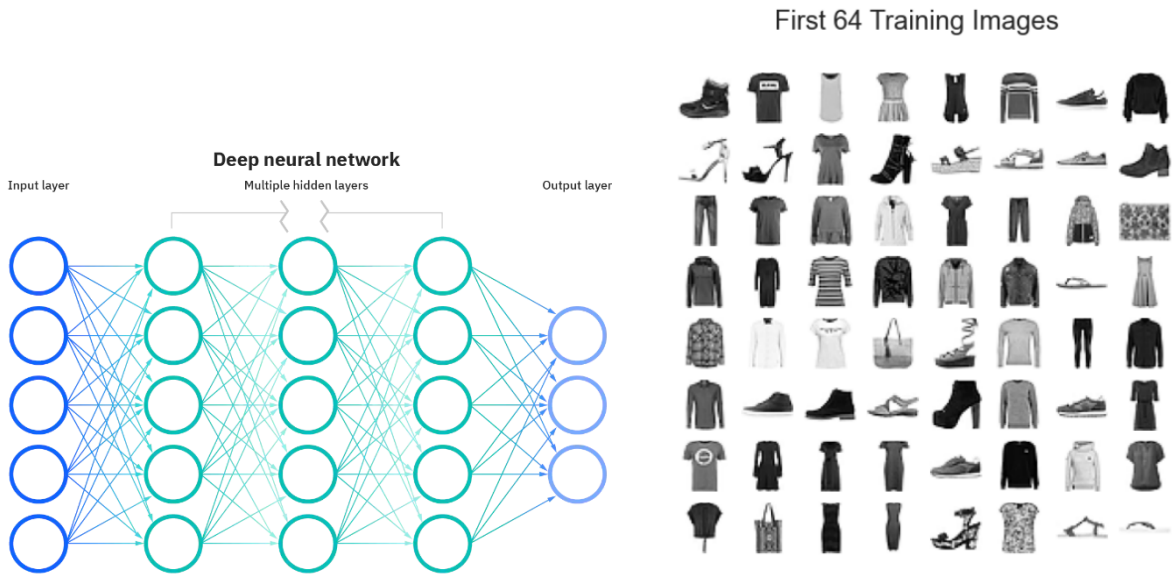


Figure 1. (Left) Illustrations of deep neural network [1]. (Right) Fashion-MNIST dataset [2].

2. Theoretical Background

A illustration of DNNs are given in figure(1). Inputs propagate through the network in the following way. If $\vec{x}^i$ is the output of layer i, and $\vec{w}^i$ are their weighted connections between the layer i and layer $i+1$, then $\vec{x}^{i+1} = f(\vec{w}^i \cdot \vec{x}^i)$ where f is a activation function.

The network is trained such that the quantity produced at the output layer is similar to a desired label. A loss function is used to measure the degree of "rightness" of the network's output. In this work, I use the cross entropy loss:

$$L(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y})) \tag{1}$$

where $\hat{y}$ is the output of the network and y is the desired label. The weights in the network are iteratively updated during the training process using gradient descent.

$$\vec{w}_{k+1} = \vec{w}_k - \alpha \nabla_{\vec{w}_k} L(\hat{y}, y) \tag{2}$$

where α is the learning rate. If a network only had the input and output layer, then

$$\nabla_{\vec{w}_k} L(\hat{y}, y) = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \nabla_{\vec{w}} z \tag{3}$$

with $z = \vec{w} \cdot \vec{x}$. In a network with more layers, partial derivatives of weights occurring in the first layers will be nested in the weights of layers toward the end of the network. By the chain-rule, partial derivatives are computed backwards across the network, layer by layer, to compute the total gradient. This process is called backward propagation.

Stochastic gradient descent computes the gradient and updates the weights after each input is passed through the network. Often, the training is done in batches, where the gradient of the loss is accumulated as a handful inputs are passed through. Some more advanced gradient descent techniques include AdaDelta, which uses an adaptive learning rate, and Adam, which uses an accelerated gradient vector and adaptive learning rate.

As mentioned before, the training dataset can be divided into batches. An entire pass through of the training dataset is called an epoch. After each epoch, a validation is done by 'quizzing' the network on a subset of the training data.

There are additional ways to tune a network. One can consider regularization, which changes the loss function to avoid overfitting in the training dataset, batch normalization, which may help with the issue of exploding or vanishing gradients, or different initialization conditions of the network. All that has been described above about the larger rules of the network constitute what is called hyperparameter tuning.

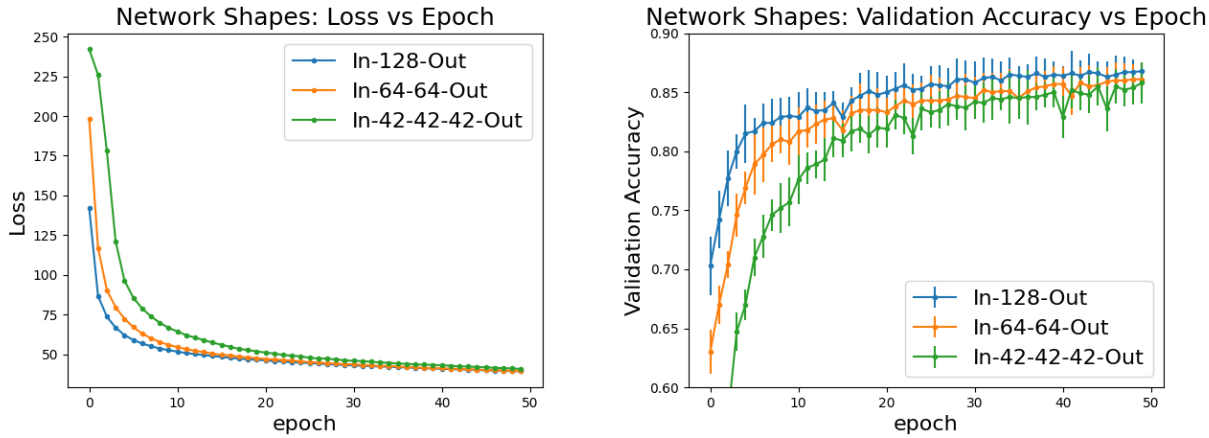
3. Algorithm Implementation and Development

This project makes extensive use of `torch` and related packages. The Fashion MNIST dataset was taken from the torchvision package. Neural network model layers, relu activation functions, cross entropy loss, optimizers, was done with `torch` commands. The splitting of the training and testing data was done with `sklearn`.

4. Computational Results

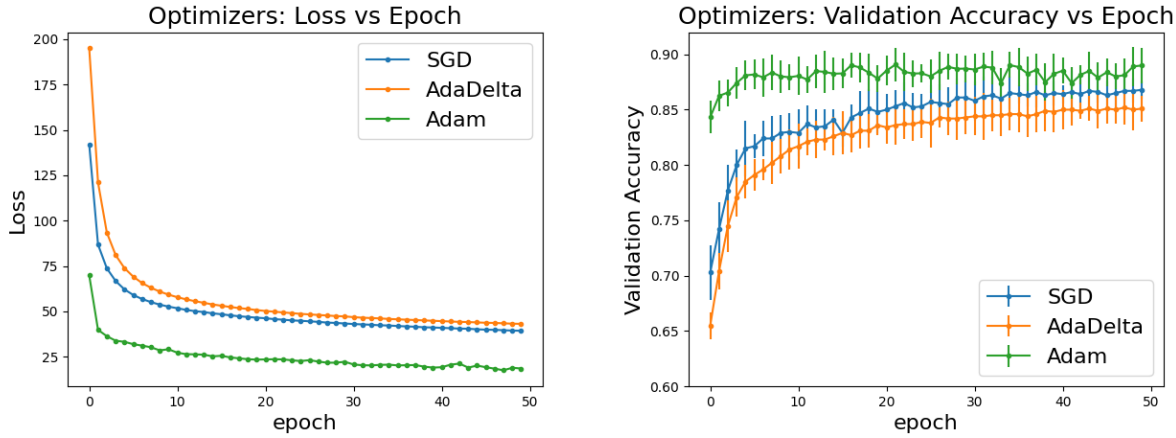
The Fashion MNIST dataset of 28×28 pixel images has $60,000$ images in the training set and $10,000$ images. Furthermore, the images in the training dataset are split into batches of size 512, where 51 of these were used for validation. The test batches are size 256.

The first thing I explored was the effect of different network shapes. I considered 3 different networks with similar number of neurons: one with 1 hidden layer with 128 neurons, one with 2 hidden layers with 64 neurons, and one with 3 hidden layers with 42 neurons. Each of these networks used cross entropy loss and stocastic gradient descent with learning rate of 0.01. Relu activation was used in all the layers except the output layers. In figure(2), you can see the results of



Network	Testing Accuracy (%)
In-128-Out	85.20 ± 2.21
In-64-64-Out	85.23 ± 1.94
In-42-42-42-Out	84.80 ± 1.88
In-128-Out (Dropout)	85.11 ± 2.57

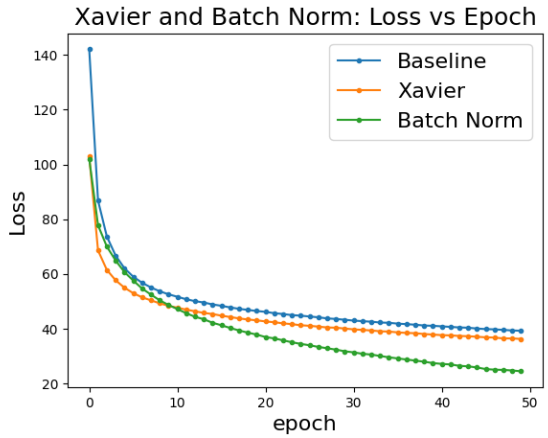
FIGURE 2. Comparison between different network shapes.

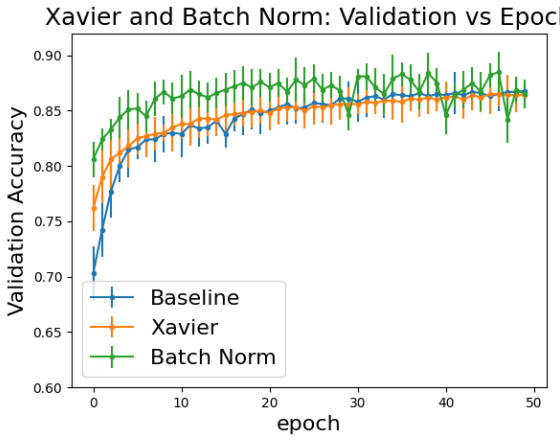


Optimizer	Testing Accuracy (%)
Stochastic Gradient Descent	85.20 ± 2.21
AdaDelta	84.28 ± 2.31
Adam	87.55 ± 2.12

FIGURE 3. Comparison between different optimizers

this. From the loss and the validation accuracy curves, we can see the networks with denser layers, like the single 128 hidden layer network, seem to perform better, especially at less than 10 epochs. As the number of epochs increases near 50, these different networks start to converge, and the error in their validation curves begins to overlap. Likewise, after 50 epochs of training these networks performed very similarly on the testing data, each falling within the error bounds of each other's testing accuracy. This test suggests that for Fashion MNIST classification, if neuron number is constant, denser layers tend to perform and train better with given a small number of epochs.





Network	Testing Accuracy (%)
Baseline	85.20 ± 2.21
Xavier	85.85 ± 2.07
Batch Normalization	86.61 ± 2.13

FIGURE 4. Evaluation of Xavier initialization and batch normalization.

Comparing the validation curves to the testing accuracy, it seems like none of the three networks shapes had any issues with overfitting, as the testing accuracy fell within the validation accuracy at the 50^{th} epoch. To make sure, I trained an additional network with the single 128 hidden layer network shape with added dropout regularization, which should aid in negating the overfitting effect. The testing accuracy of this dropout-network is given in the table in figure(2). It can be seen that there was no real difference with the added regularization, which was to be expected because the networks were able to perform similarly in the validation and testing steps.

Keeping the single 128 hidden layer network as a baseline, I explore the effect of different optimizers: Stochastic Gradient Descent (SGD), AdaDelta, and Adam. For each of these optimizers, I keep the learning rate constant at 0.01 (although this may not be appropriate because this parameter can mean something different for each optimizer, I did not want to change too many things at once). The results are summarized in figure(3). Out of the three optimizers, there was a clear winner. While the SGD and AdaDelta networks performed similarly, falling within each other's error bounds for validation and testing accuracy, the Adam network performed $2-3\%$ better than the other two in validation and testing. In fact, looking at the validation graph, the Adam network with a single epoch of training competitive with the other two networks with 50 epochs of training. It was expected that Adam would converge the fastest, as the Adam optimizer uses 'accelerated' gradient vectors to converge faster.

Finally, I considered the effect of batch normalization and Xavier initialization. These can be useful because they help to solve the issue of vanishing and exploding gradients during training. Batch normalization refers to normalizing the outputs at every layer. Xavier initialization refers to initializing the weights from a normal distribution dependent on the number of input and output neurons. In figure(4), using the single 128 hidden layer with SGD as a baseline, I see how adding Xavier initialization or batch normalization effect the training and testing of the network. As can be seen, Xavier normalization makes the network train more quickly at the start, but then has little effect later on. Batch normalization makes the network train faster and reach an overall higher accuracy in during testing.

5. Summary and Conclusions

This project was an introduction to DNN and the various ways one can tune hyperparameters in a DNN. I found this to be very useful and practical considering how ubiquitous DNN are in the world of machine learning. I am sure I will refer back to this project if I ever need a refresher on how DNNs work.

References

[1] IBM Cloud. Neural networks. `https://www.ibm.com/cloud/learn/neural-networks`. Accessed on March 8, 2024.
[2] E. Slizerman. Amath582 homework4.