AMATH 582: HOMEWORK 4

HUNTER LYBBERT

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

ABSTRACT. In this report we survey fully connected linear neural networks and the most common hyperparameters to consider for tuning. Various methods of optimization are evaluated with a variety of learning rates and momentums. Dropout and batch normalization are discussed and implemented as well. Methods of hyperparameter tuning, comparing models and results are presented. The task we are applying this to is the well known 10 class classification problem with the FashionMNIST dataset.

1. INTRODUCTION AND OVERVIEW

In this report we further our understanding of machine learning concepts focusing on deep learning, the study of neural network based model architecture.

Again, our setup is a common supervised learning problem, given a collection of N data points with labels in classification or target values in a regression setting

$$\big\{(\boldsymbol{x_0}, y_0), (\boldsymbol{x_1}, y_1), ..., (\boldsymbol{x_{N-1}}, y_{N-1})\big\}.$$

The data is denoted as a matrix X and a vector of target values or class labels $\boldsymbol{y}$. We then are looking for a function f which takes in the training data and most accurately predicts the target values or class labels, written in optimization form we are looking for the following

$$\text{(1)} \qquad f_{MLE} = \underset{f}{\text{argmin}} \frac{1}{2\sigma^2} ||f(X) - \boldsymbol{y}||_2^2$$

where σ^2 is the variance of the normally distributed error terms $\epsilon \sim \mathcal{N}(0, \sigma^2)$ defined by $\epsilon_i = y_i - f(x_i)$ So said another way we are trying to minimize our errors in the classification task. The specific class of functions f to be considered to solve the problem is a Fully Connected Neural Network (FCN). We will treat the theoretical background of these methods in the next section 2.

Before proceeding, we would like to acknowledge the critical use of the following packages in our analysis. Namely, Matplotlib was used to create all plots and animations [3]. Addiitonally, PyTorch [2] was crucial for easily implementing the desired architecture while Ray Tune [4] was used to help with hyper parameter tuning.

2. THEORETICAL BACKGROUND

For the purpose of this report we will be concise and straight to the point about the methods, architectures, and hyperparameters used in our study. The motivation for constructing a neural network is quite literally the network structure of the neurons in our very own brains (see Figure 1).

Key components of a biological neuron which are reflected in the artificial neuron in our models are the Dendrites which receive the signals into the neuron, this is like the weights receiving input from the previous layer. The Cell body combines these signals like our neuron which calculates
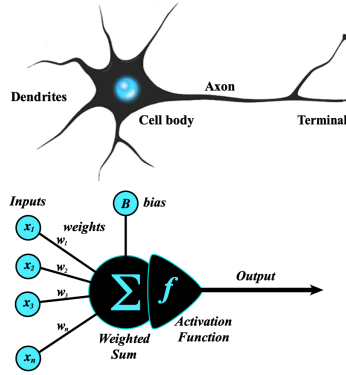

Date: March 14, 2025.

FIGURE 1. Neural networks are truly motivated by the biological behavior of neurons in our brains. (Image from [1])

a weighted sum of the inputs plus a bias (like in regression). Depending on the signal a neuron received it may fire and pass along the signal through the axon to the neurons down stream from it, artificially our nonlinear activation functions act as this thresholding decision maker determining when to pass on the signal and at what strength to the next layer. The power of a neural network is creating a large network that is comprised of very wide layers (many neurons) and or is many layers deep. However, the basic building block is this artificial neuron we have just described.

Another fundamental piece of information to describe, before we get into the architecture and hyperparameters, is what a forward and backward pass through the network are. Firstly, a forward pass is simply taking the data we are using to try and make predictions of some sort and passing it into the network. More mathematically we have the following expression for each neuron

$$(2) \qquad y = f\left(\sum_{i=1}^{n} x_i w_i + b\right)$$

where x_i are the elements of our input vector $\boldsymbol{x}$ or the results of a previous layers output (past y's). The w_i's are the weights of the connections between these previous layers outputs or original inputs. We also add a bias term b. Next, the weighted sum is passed through the nonlinear activation function f to produce either a final prediction if this is the final layer in the network or creating the input for the next hidden (not first or last) layer in the network. Secondly, a backward pass is when we grade how well the network predicted the target output and we propagate the information about how wrong we were back through the network to improve our prediction a little for the next forward pass. Once again, we have a more concise mathematical expression for this, which we will get to momentarily. Recall the expression (1) which we are trying to argmin, this metric or measure of error we are trying to minimize is called a loss function. We can choose different loss functions, let's denote our arbitrary choice of loss function as $L(\hat{y}, y)$ where $\hat{y}$ is the predicted label or value from the network and y is the target value or label. Now we can precisely define backward pass

$$(3) \qquad \nabla_{\boldsymbol{w}} L(\hat{y}, y) = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \nabla_{\boldsymbol{w}} z.$$

We are referring to variables as represented here in this more graphical representation of forward and backward passes

$$\boxed{\boldsymbol{w}^T \boldsymbol{x} + b = z} \rightarrow \boxed{\sigma(z) = \hat{y}} \rightarrow \boxed{L(\hat{y}, y)}$$

$$\boxed{\nabla_{\boldsymbol{w}} z} \leftarrow \boxed{\frac{\partial \hat{y}}{\partial z}(z)} \leftarrow \boxed{\frac{\partial L}{\partial \hat{y}}(\hat{y}, y)}.$$

For completion and yet in favor of being concise, we include the following expression for gradient descent (GD) which is how we update the weights is

(4)
$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w}_k; b)$$

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

where α is known as the learning rate controlling the size of the step in the direction of the gradient. There is, of course, more to say about exactly how back propogation works or how autograd works in pytorch but for the sake of this report we will leave it here and begin our discussion of the optimizers and hyperparameters.

2.1. **Architecture.** Deciding the architecture of the model can have large influence on the computational performance and accuracy of your model. The only considerations after one has decided to use a fully connected network is the number of hidden layers and the size or number of neurons in each hidden layer. A deeper network can be necessary for some problems but should not be considered the best practice. One reason to hesitate to use a very deep network is the potential to have vanishing or exploding gradients. The back propagation across a deep network can begin to get very small causing the optimization to stall and no longer progress. A wide network can't learn as complex a structure in the data, however, it will train faster and can still do well on small tasks. A developer needs to make the best decision given their priorities.

2.2. **Optimizer.** There are several variations of gradient descent (4) which we are going to describe here. They each attempt to overcome a few of the challenges with basic gradient descent.
  (1) Stochastic Gradient Descent (SGD):
      Unlike basic GD which makes updates based on every datapoint in the training set, SGD makes updates more frequently after a batch of data, mini-batch, or even every single datapoint. The expression for it does not vary enough from the basic one to warrant adding again here.
  (2) Adam:
      This adds the ability to vary the learning rate throughout the training process as well as adds a *momentum* term which conveys a sense of certainty that the direction we're moving in is the right one based on where we moved in the previous iteration. Mathematically this alters equation (4) only slightly (for just the weights) we have

(5)
$$\boldsymbol{v}_k = \eta \boldsymbol{v}_{k-1} + \alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w}_k; b)$$

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \boldsymbol{v}_k$$

      where η is our momentum parameter which takes on values between 0 and 1 but is commonly close to 0.9. Furthermore, our physical intuition for momentum does not lead us astray here. Imagine, a ball descending down hill, if it has a lot of momentum when it meats a small basin it will likely roll into it and continue out of it if the other side of the basin is not too steep. If the ball escapes the small basin it will likely continue to descent to an even lower basin and so on until it reaches perhaps the lowest point in the greater region. Similarly, momentum helps the optimization algorithm avoid local minimums that can trap our methods from arriving at a more globally optimal solution. We only claim "more" because it is extremely

hard to locate the true global minimum in such a high dimensional nonlinear optimization problem.
(3) RMSProp: We only mention that this optimizer method exists and was used in our hyperparameter tuning to be discussed later. As far as the finer details of the method, for our purposes here it suffices to say it is an extended version of AdaGrad which itself introduces an adaptive learning rate. For further details, **TODO: Add reference for further reading.**

2.3. **Dropout. TODO: Update** Describe briefly why we do this to avoid overfitting

2.4. **Batch Normalization. TODO: Update** Describe how this helps things be a little more stable

2.5. **Weight Initialization. TODO: Update** Describe how the starting point matters a lot on such a complicated loss surface

3. Algorithm Implementation and Development

TODO: Update

(1) discuss how you implemented the model architecture with variable layers and sizes
(2) discuss how you tried to implement Ray Tune
(3) Describe your ModelTrainingArtifact Class
(4) Describe how you chose to hyperparameter tune across all hps instead of doing it incrementally (for the sake of thoroughness).
(5) Describe how you logged the experiment results and the short comings of that.

4. Computational Results

TODO: Update

| Classifier	Digits Used	Train Accuracy	Test Accuracy	CV Score
Ridge Classifier	1 and 8	0.965	0.980	0.963 ± 0.00281
Ridge Classifier	3 and 8	0.961	0.964	0.959 ± 0.00590
Ridge Classifier	2 and 7	0.981	0.973	0.981 ± 0.00226

TABLE 1. **TODO: update me**

5. Summary and Conclusions

TODO: Update

(1) further work is going to include finding a better way to share and convey the configuration in the visuals
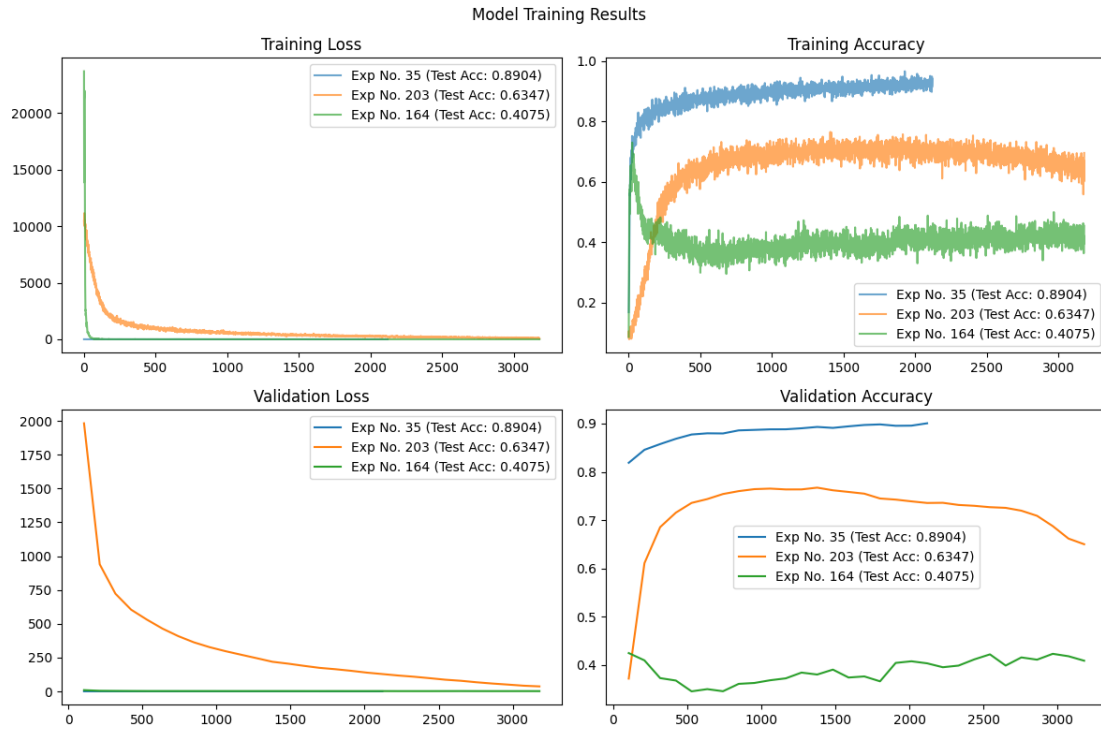(2) screw ray tune

Acknowledgements
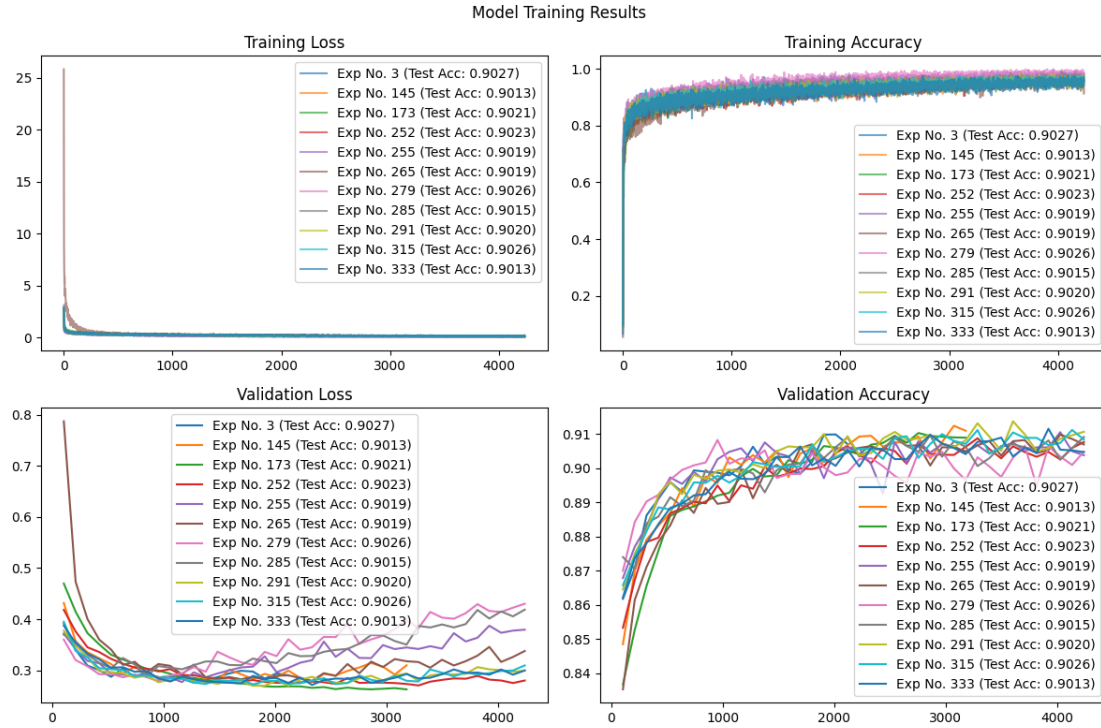
FIGURE 2. **TODO: Update caption**

REFERENCES

[1] Deep Learning (DL) — mriquestions.com. `https://mriquestions.com/what-is-a-neural-network.html`. [Accessed 13-03-2025].

[2] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Apr. 2024.

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

[4] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.
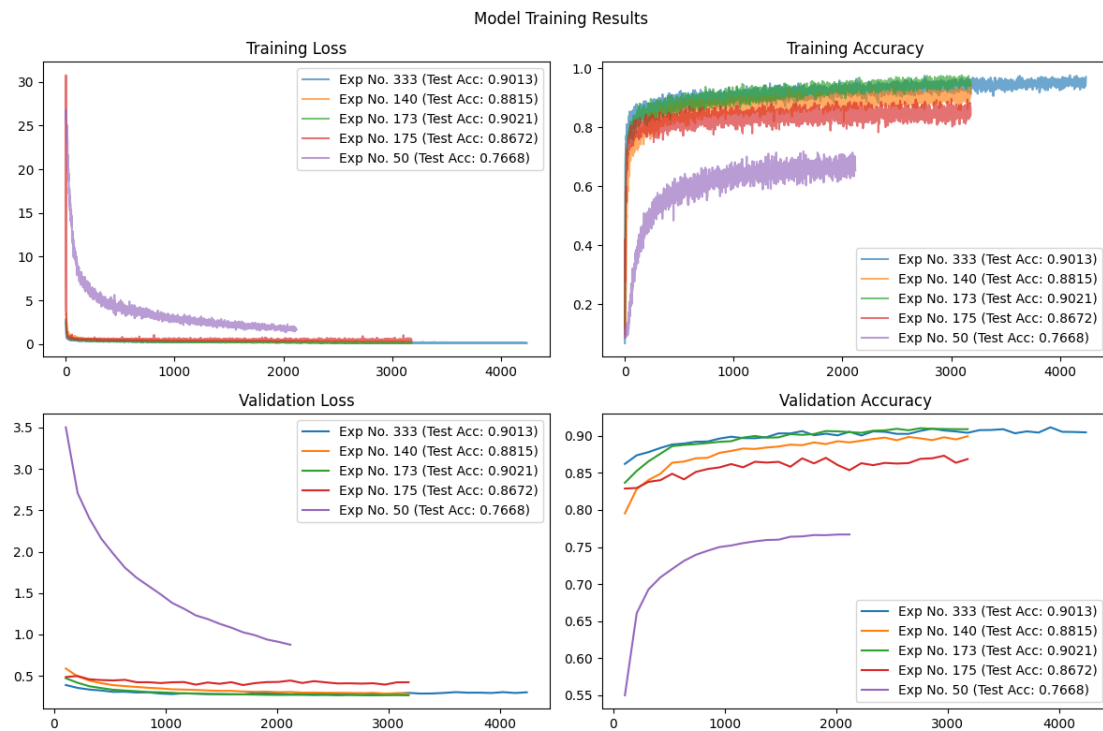
FIGURE 3. **TODO: Update caption**
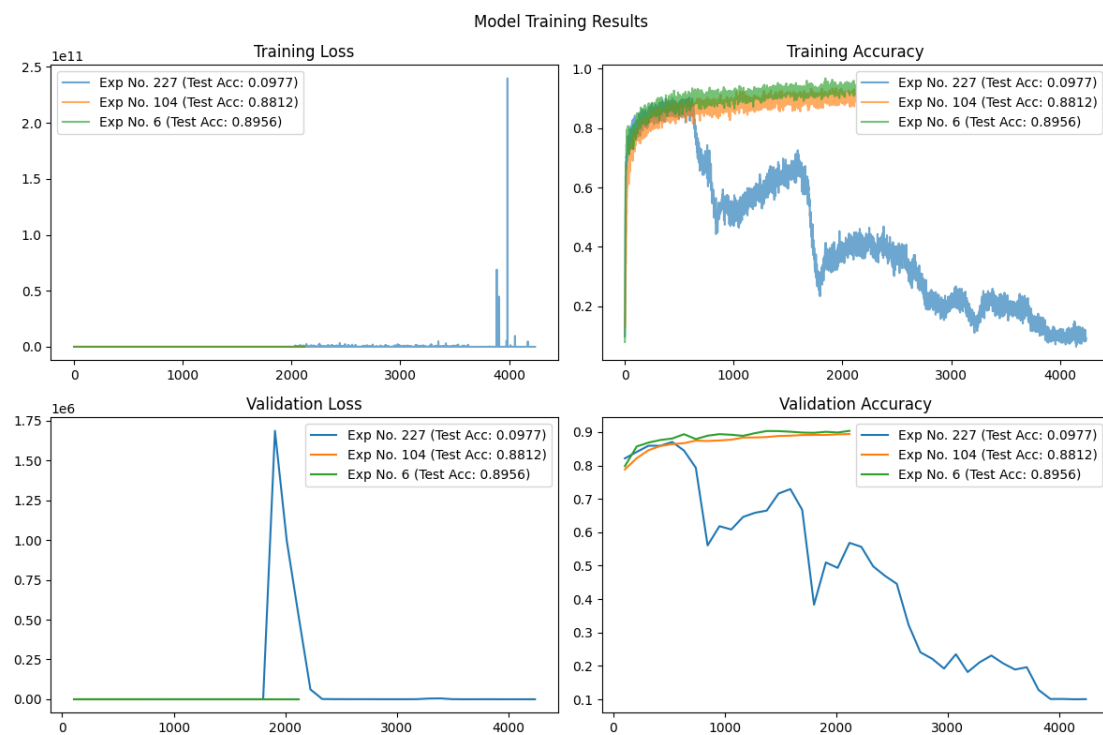
FIGURE 4. **TODO: Update caption**

FIGURE 5. **TODO: Update caption**