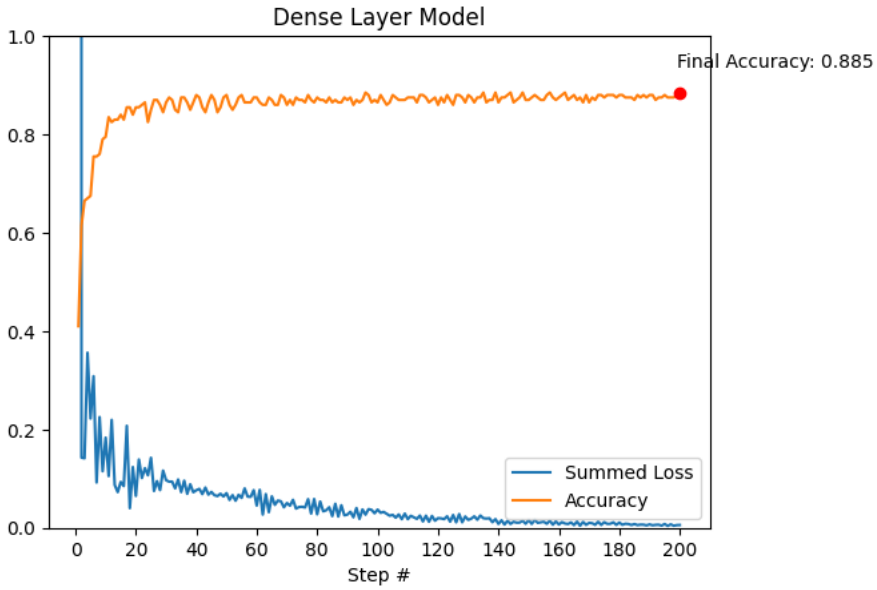
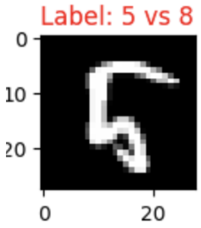
Neural Network Technical Report

My network architecture is essentially a tweaked variant of the provided architecture. Initially made small tweaks to the parameters of the network (mainly following these guidelines), but I found that those guidelines are mainly for efficiency, whereas my goal is maximizing accuracy. I initially had a network with three linear hidden layers with sizes that gradually went from the size of the input layer to the size of the output layer, and it achieved .86 ultimate accuracy with relatively fast speeds. While I learned a lot of lessons from experimenting with this earlier network, I ended up brute-forcing the architecture for the sake of maximizing accuracy. By brute-forcing, I mean I just put as many nodes as I feasibly could (while keeping training time < 30 minutes) inside of my network. So, I put three hidden layers of 2048 nodes each, alternating with three ReLU activation layers. With that architecture, I would occasionally achieve my goal of 0.9 accuracy, but I more consistently got an ultimate accuracy of ~0.89, occurring after an early plateau which I tried and failed to push past.

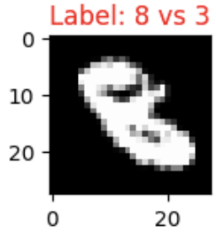
Below is my ultimate network's performance graph:



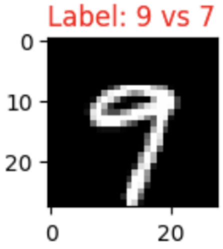
The instances of failure for the network generally came from examples which would be challenging for a human to decipher, or examples which defied the usual patterns that define numbers.



This failure is unusual, but my best guess would be that the curved tail at the bottom, nearly forming a loop. made the network guess that it was an eight.



This is an instance of the image being barely ledgible, even for a human, and my guess would be that the slanted nature of the number is one of the things that threw the network off.

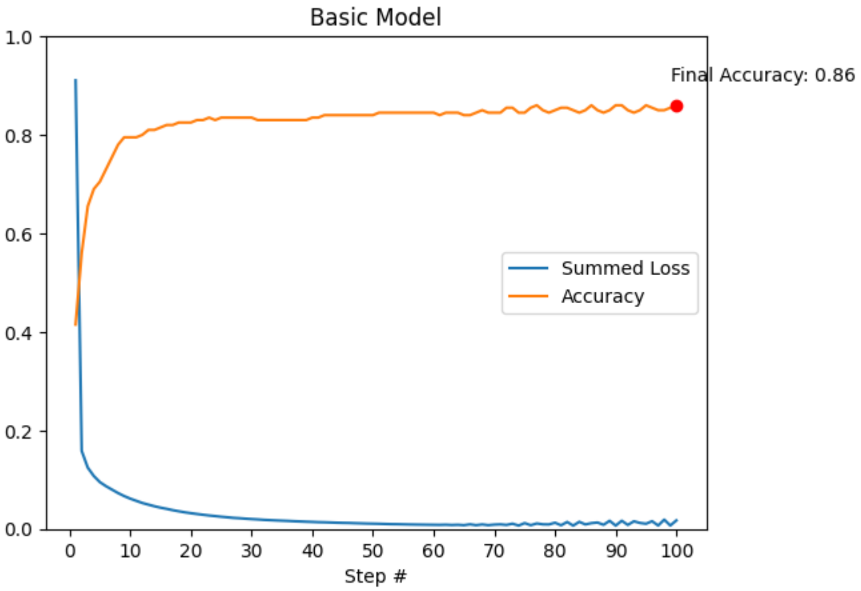


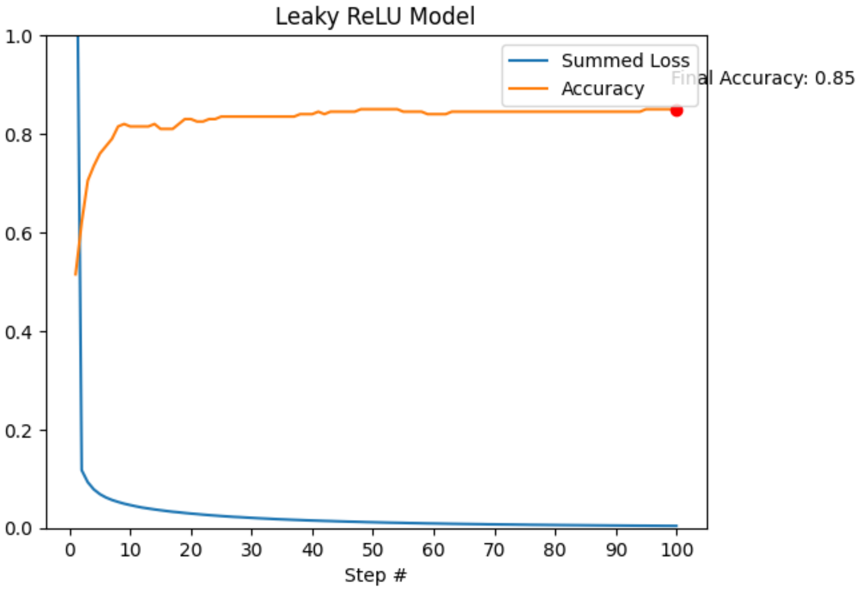
In this case, the loop at the top of the 9 was likely too small for the network to recognize. So, it interpreted the top loop as a line, leading the network to see it as a 7.

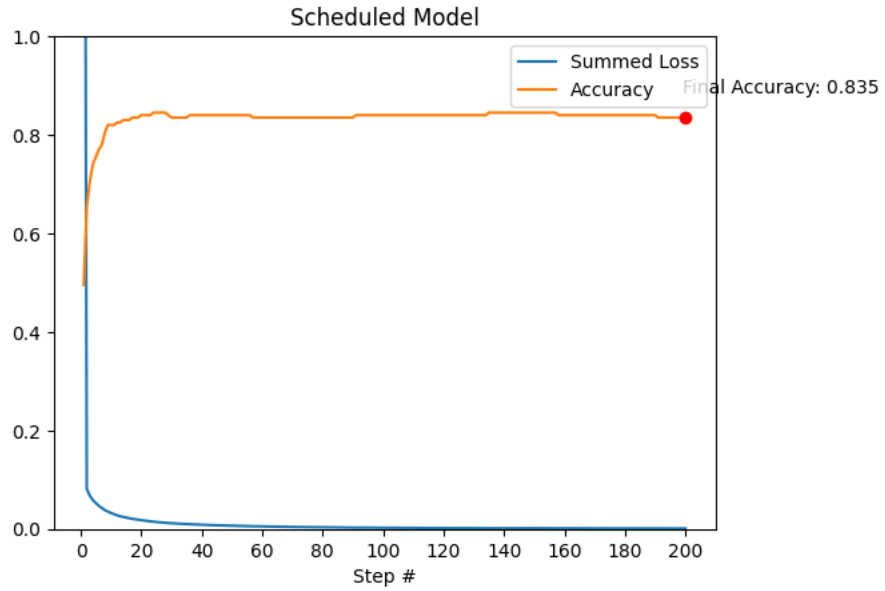
Extra Credit

I experimented with a new activation function that I found, which is called Leaky ReLU or PReLU. Instead of totally removing negative inputs like ReLU does, it significantly diminishes them by multiplying them by a small coefficient α, which, in my case, was 0.01. Replacing the ReLU layers of my original network with Leaky ReLU layers led to a small, inconsistent increase. Sometimes it would increase the ultimate accuracy by as much as 0.03, but other times it would be about the same, and sometimes less, than the basic network's performance. I guess this makes sense, because of how similar PReLU and ReLU are.

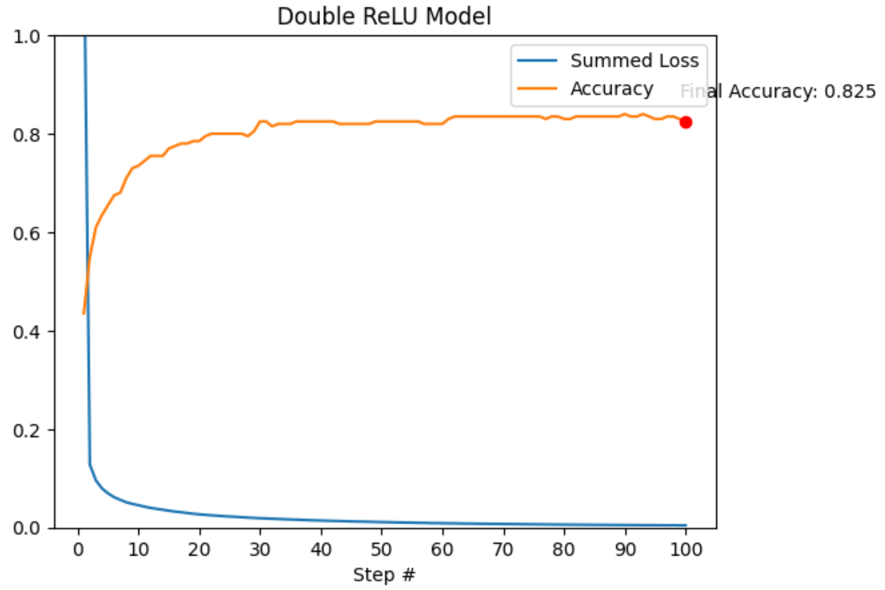
I also experimented with many new approaches and architectures, the performances of each can be found below. Each of the following networks were based on the "Basic Model," which acted as a control. The Basic Model had two hidden layers, the first with 512 nodes and the second with 360. Between each linear layer was a ReLU layer. The altered networks, while having differences in various parameters, all have those two hidden layers with the same number of nodes as the Basic Model. Additionally, the performance of the previously described network utilizing Leaky ReLU can be seen below.


Basic Model
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60
70
80
90
100
Step #
Summed Loss
Accuracy
Final Accuracy: 0.86

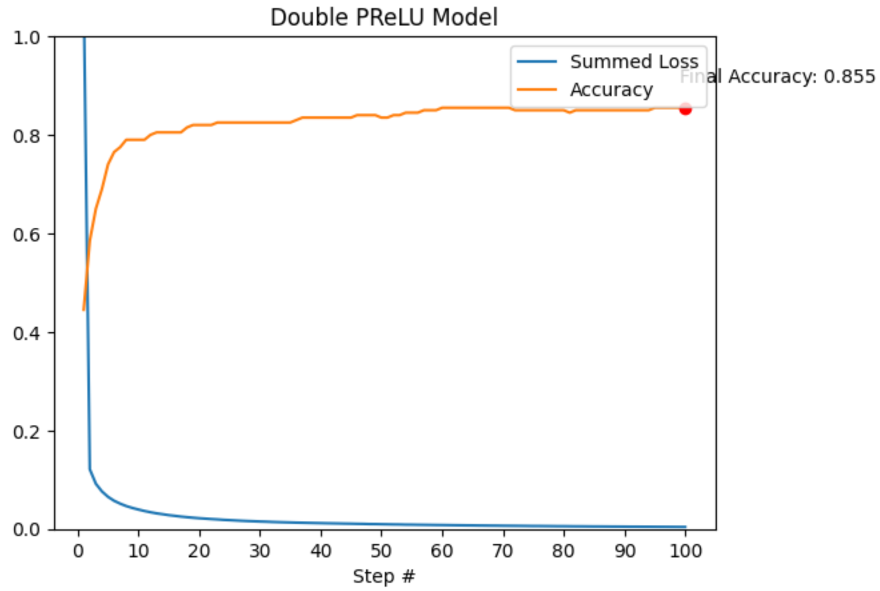

Leaky ReLU Model
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60
70
80
90
100
Step #
Summed Loss
Accuracy
Final Accuracy: 0.85

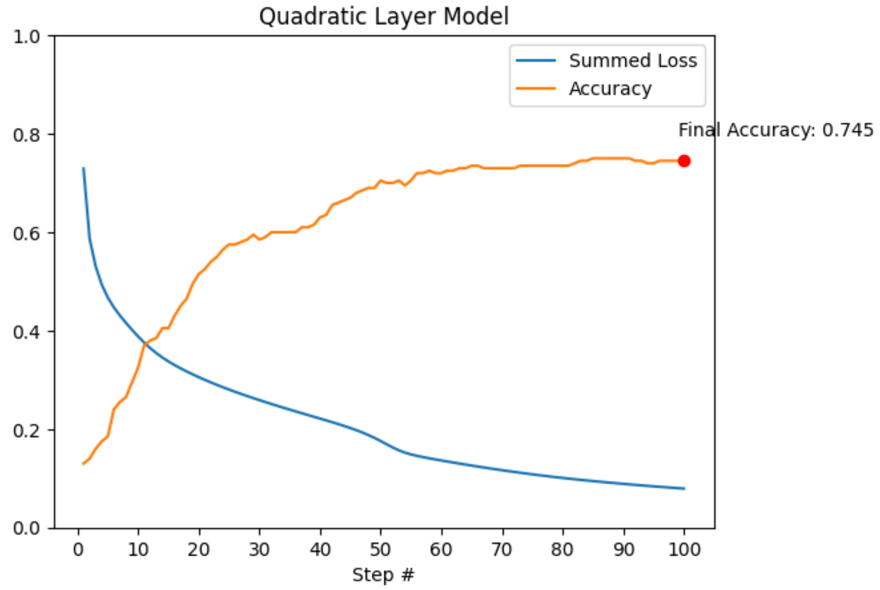


Scheduled Model



Double ReLU Model





First off, I tried adding learning rate scheduling to my basic network, which entails decreasing the learning rate after every few epochs to minimize plateauing. However, I was shocked that it seemed to *decrease* performance. This didn't make much sense to me. While I probably could have tuned it in different ways to get a more ideal result, it indicated that the learning rate wasn't to blame for the performance plateauing, which was very helpful as I debated adding the scheduling to my ultimate network.

Additionally, I experimented with different architectures, the first of which being a network with two ReLU activation layers for each nonlinear layer. Like the PReLU network, it seemed to marginally improve performance, but inconsistently so. Similarly, I tried a network with two PReLU layers for every linear layer, and while the resulting performance increase was still marginal, it was much more consistent, which is why I used a similar architecture for my final network.

Finally, I was curious to see what would happen if I replaced my network’s linear layers with layers containing weighted quadratic functions, and the result was a significantly less accurate network. Quadratic equations can produce a much wider range of outputs, meaning that a slight change to the weight on these quadratic layers changes the results significantly, and while that may be ideal if the optimal weight is large, since the quadratic function would reach it in less epochs, it isn’t ideal otherwise, since it would take more epochs to descend to the weight producing the ideal output. For example, if there was a node that consistently had 16 as an input, a change of 0.1 to the weight in its linear layer would cause the output to change by 1.6, whereas a change of 0.1 to the weight in a quadratic layer would cause the output to change by 25.6. So, if the ideal output was close to the initial output, it would take the quadratic layer more epochs to reach it, since changes to the weight cause larger, less precise changes to the output that would cause the node’s weight to vary more wildly with each epoch, which can be seen in the jagged nature of the quadratic network’s performance graph.

Overall, these experiments taught me the sheer magnitude of options available to a data scientist who wants to tweak their network, and how challenging it can be to sift through and test each option to maximize performance. If I was to improve performance even more, I would experiment with data processing (e.g. cropping images to a section containing only the non-white pixels), or I would figure out a way to test the various neural network parameters efficiently and simultaneously to more rapidly arrive at an ideal configuration. However, I found that increasing the amount of nodes and layers is the most consistent way to improve performance, and that’s exactly what I did with my final network.