CS462: HW1 - Architecture and Hyperparameter Search

Yash Chennawar | yc1376@scarletmail.rutgers.edu

November 18, 2025

1 The Basic Model

The linear softmax classification model $F(\vec{x}) = A\vec{x} + \vec{b}$ has $28 \cdot 28 = 784$ input features and 10 output features. The dimensions of matrix A are 10×784 and the dimensions of the $\vec{b}$ vector are 10×1. So, the total number of trainable parameters is $10 \cdot 784 + 10 = 7850$.

To minimize loss without overfitting, I tried various numbers of epochs and learning rates.

- $\alpha = 1.0$: noisy training loss and worst loss and accuracy. This learning rate is too high for SGD.
- $\alpha = 0.7$: provided a good baseline as it converged well with a test accuracy of 0.9035. The training plateaued at around 120 epochs.
- $\alpha = 0.5$: similar to 0.7 but had slightly better accuracy and loss.
- $\alpha = 0.1$: had very low loss and high accuracy, over 200 epochs. This suggests that a low learning rate and a high number of epochs may be the best strategy for this problem.
- $\alpha = 0.05$: showed the best loss and accuracy for the test and train datasets. I will go more in depth about this configuration.

Dataset	**Loss**	**Accuracy**
Train	0.2396	93.28%
Test	0.2869	92.21%

Table 1: Learning Rate = 0.05, Epochs = 400

This choice of learning rate and number of epochs has produced the best results. I kept decreasing the learning rate as I saw continuous improvements for all the metrics being tracked as learning rate decreased. I tried choosing a large number of epochs to train for to see when training is no longer worthwhile. I noticed that from epochs 350 to 400, the train loss decreased by 0.003, this is a 0.00006 change in loss per epoch. This change is incredibly small and suggests that training has plateaued. The loss would still likely decrease with more

epochs but it would be by such a small amount that there is no real benefit to it: additional epochs have diminishing returns. I also ensured that the model has not overfit as the gap between train and test loss and accuracy is very small: the model generalizes well.

I tried training the model with 3 different random seeds and all 4 metrics being tracked were very similar, given a constant learning rate of 0.05 and 350 epochs. I chose to train until 350 instead of 400 since loss plateaued around at around 350 epochs. This means that the performance of the model does not significantly change with a different initialization of parameters.

2 A Fixed-Size Layer Model

Parameters(k, m) in terms of input size n_{in}, output size n_{out}, number of hidden layers k, and number of nodes per hidden layer m.

The connection from the Input Layer to Hidden Layer 1 has the following number of trainable parameters. There are m nodes in the first (and every) hidden layer multiplied by n_{in} input nodes. Then, there are m bias terms from the hidden layer.

$$(n_{in} \cdot m) + m$$

The connection from each hidden layer to the next hidden layer follow a similar pattern. Each layer has m nodes connected to m nodes, and each next layer has m bias terms. So, the following expression represents the number of parameters in the hidden layers.

$$(k-1) \cdot (m \cdot m + m)$$

Finally, the connection from the last hidden layer to the output layer also follows a similar pattern. m nodes are connected to n_{out} nodes, with n_{out} bias terms. So, the following expression represents the parameters in this last part.

$$(m \cdot n_{out}) + n_{out}$$

All together, the following formula is the number of trainable parameters in the network.

$$\text{Parameters}(k, m) = P = [(n_{in} \cdot m) + m] + [(k-1) \cdot (m \cdot m + m)] + [(m \cdot n_{out}) + n_{out}]$$

For a fixed number of parameters P, we need to find the smallest and largest values of k that satisfy Parameters$(k, m) = P$. The formula can just be rewritten with some simple algebra, after substituting $n_{in} = 784$ and $n_{out} = 10$. This results in the following quadratic.

$$P = (k-1)m^2 + (k+794)m + 10$$

First, we find the smallest and largest values that k can take for a given P.

- Since $k \geq 1$, the smallest k is always 1.

- The largest k, which is k_P, maximizes the number of layers. This means that the number of nodes in each layer must be minimized, therefore let $m = 1$ (as $m \geq 1$). Substituting this m into the quadratic from above gives the following expression.

$$\begin{aligned} P &= k_P - 1 + k_P + 794 + 10 \\ P &= 2k_P + 803 \\ k_P &= \frac{P - 803}{2} \end{aligned}$$

  For a given P, the valid range of values k can take is in the interval $[1, \lfloor \frac{P-803}{2} \rfloor]$. k_P should always be rounded down if it is not an integer since a fractional number of layers is not meaningful, and rounding down would keep it within the P budget as opposed to rounding up.

Now based on the determined k for a chosen P, we must find valid values of m. To do this, solve the quadratic formula from before for m with the quadratic formula. Only the positive root is relevant here since there is no such thing as a negative number of nodes.

$$\begin{aligned} 0 &= (k-1)m^2 + (k+794)m + 10 - P \\ m &= \frac{-(k+794) + \sqrt{(k+794)^2 - 4(k-1)(10-P)}}{2(k-1)} \end{aligned}$$

Now for a chosen P, $m_P(k)$ is the number of nodes per layer that moves Parameters(k, m) close to P. To find this, I first roughly found a learning rate that works for both deep and shallow networks. Through experimentation, I saw that $\alpha = 0.1$ performed the best. With this constant learning rate, I tested 3 different large P values (100000, 250000, and 500000), with a large assortment of various k values, ranging from 1 to 50. For each P and for each k, I found the corresponding m such that it satisfies the quadratic formula from before. With this configuration, I trained and tested the full neural network for 25 epochs. The following graph shows the results on the train and test data.

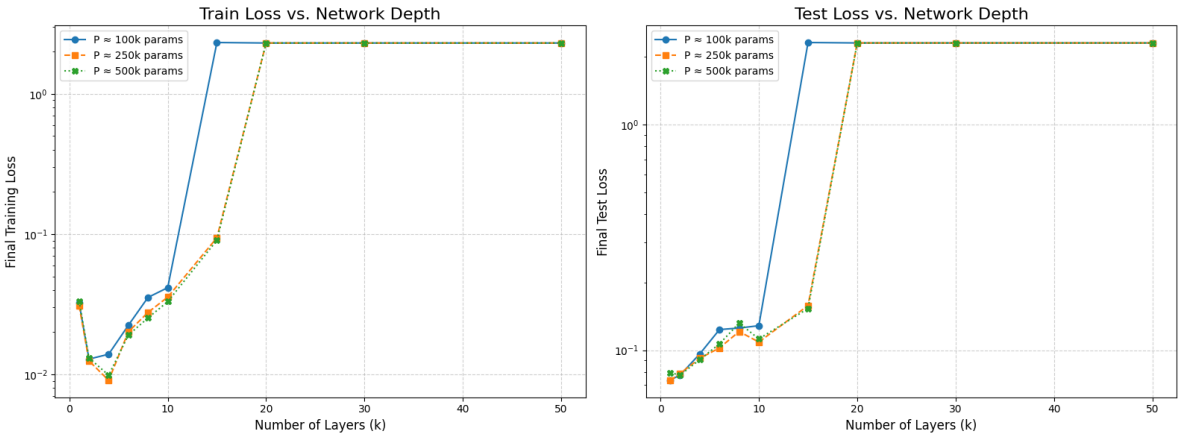

Figure 1: Train and Test Loss over P and k values

Based on these graphs, there is a clear correlation between number of layers k and the final loss for both train and test data. As the number of layers increases, the loss increases as well. Up until 10 layers, the loss slowly increases but after that, it starts to jump higher. The best configuration seems to be between 1 to 10 layers in the network, for all the given P values. This shows that shallow but wide networks learn better than deep but narrow networks, for this MNIST problem. The best performance was generally with just 1 or 2 hidden layers, and this shows that the features needed to classify digits are not complex enough to require many layers.

As k becomes too large, at around $k = 15$, there is a sharp spike in loss. This is because the number of nodes in each layer must dramatically decrease to remain within the parameter budget P. These narrow layers cause an information bottleneck, where the important gradient information was not able to flow to later layers. The model failed to learn and had very high loss; the model did not perform better than random guessing.

Additionally, $P = 100k$ networks had slightly higher loss than the $P = 250k$ and $P = 500k$ networks, for most of the k values. The 250k and 500k parameter models both performed very similar to each other. This shows that increasing the parameters from 100k to 250k had some improvements but increasing from 250k to 500k did not yield much of an improvement. This reveals the diminishing returns of having too many parameters when a model can sufficiently learn the problem with 250k parameters.

The best model minimizes the test loss as this shows the model's ability to generalize on new data. Based on the data, the best configuration was with $P = 250k$ parameters, when $k = 1$ and $m = 314$. A summary of the results is in the table below.

Dataset	**Loss**	**Accuracy**
Train	0.0318	99.11%
Test	0.0682	97.88%

Table 2: $k = 1$ & $m = 314$ for $P = 250k$; $\alpha = 0.1$, Epochs = 25

3 Improving Architecture

I tried a few different network configurations with a low number of layers, as summarized in the table below. All of these networks were trained with a learning rate of 0.1 for 25 epochs, and each has roughly the same total parameters

Architecture Shape	Layers	Test Accuracy
Bottleneck/Diamond	[250, 300, 200]	0.9782
Deep Funnel	[280, 220, 150]	0.9777
Uniform	[300, 300]	0.9774
Shallow Funnel	[390, 260]	0.9768
Pyramid/Increasing	[290, 310]	0.9762

Table 3: Comparison of Network Architectures by Test Accuracy

I consistently used few layers due to the results from the last section, where we showed that too many layers could result in poor performance. Through this experiment, I saw that the network can perform slightly better with non-uniform architecture. The best performance was with a bottleneck shape with hidden layer sizes of 250, 300, and 200. The increasing pyramid shape performed the worst while the funnel performed relatively well. But, a change in architecture is not actually providing much of an improvement, suggesting that other hyperparameters should be tuned as opposed to architecture.

Regularization, specifically with weight decay, did result in some small improvements. Weight decay prevents overfitting by adding a penalty to the loss function proportional to the weights of the model. With weight decay of 1e-5, the final test accuracy of the model, for the [250, 300, 200] architecture, was 97.9%. Without decay, it was 97.82%. It was already a strong model and decay only made it marginally better. Decay is a beneficial technique for other datasets but for this dataset and architecture, it is unnecessary.

4 Training Effects

For this section, we need to try batch sizes equal to every factor of 60000 (since there are that many data points), so there are 60 batch sizes to test. All 60 batch size options must then be tested for 3 different learning rates, namely 0.1, 0.05, and 0.01. The results are shown below.

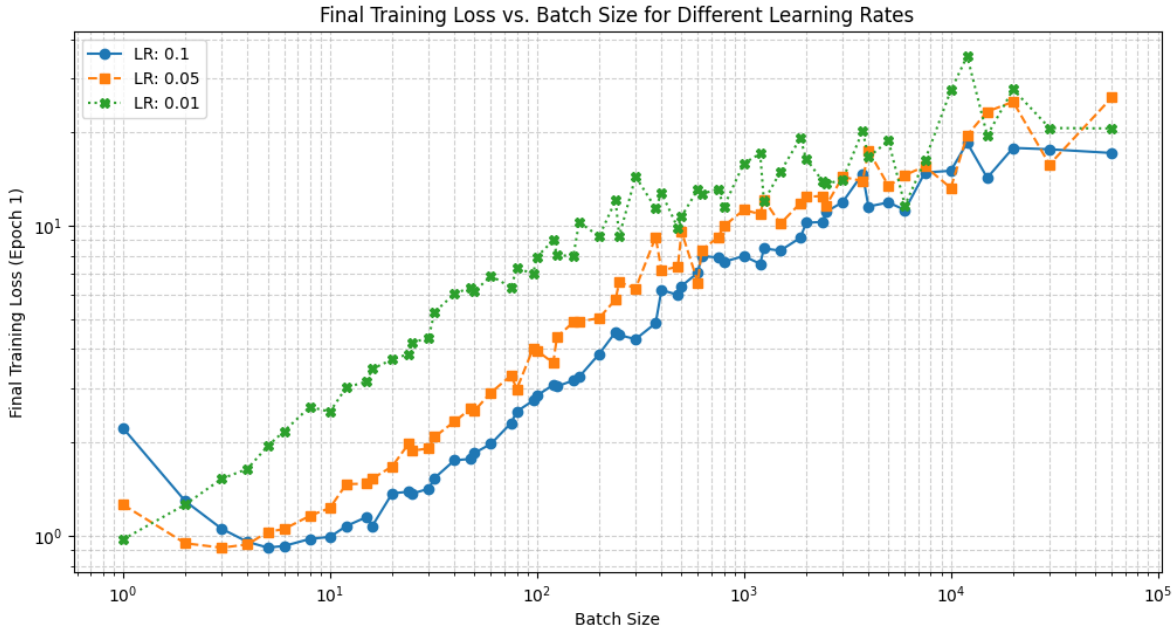

Figure 2: Train Loss over Batch Sizes

For the 0.1 and 0.05 learning rates, there is a 'U' shape curve. In the beginning for very few batches, the loss is high due to noisy updates. Then the loss reaches a minimum at around a batch size of 3 to 10. It starts to increase again at larger batch sizes due too few updates.

Interestingly, for the learning rate of 0.01, the loss increases as batch size increases. It performs best at the smallest possible batch size of 1. This is because at such a small learning rate, there will not be many noisy updates. For a batch size of 1, the model is making 60,000 tiny steps in one epoch, which allows the model to slowly train to the optimal loss value. On the other hand, for a batch size of 60, we only get 1000 tiny steps, which means there is less progress. Even though the gradient is more accurate with a larger batch size, it is not enough since the number of updates is too low.

In general for all three learning rates, a large batch size has a high loss. Even though the gradients would be more accurate for larger batches due to it being a better estimate, it will make good updates per epoch but in just 1 epoch, a few good updates is not enough. For large batch sizes, there is slow progress early on.

Overall, the best learning rate was 0.05 with a batch size of 3, for one epoch. This gave a loss of 0.915.

Now using these optimized hyperparameters, I tried different optimizers and activation functions. Below is a table comparing train and test loss for all these configurations

Optimizer	Activation	Train Loss	Test Loss
sgd	sigmoid	2.211411	1.757226
sgd	tanh	0.374073	0.242148
sgd	relu	0.366630	0.283933
sgd	elu	0.393658	0.245569
adam	sigmoid	4.219674	3.157607
adam	tanh	6.718481	4.698820
adam	relu	2.364481	2.304824
adam	elu	6.622661	5.445289

Figure 3: Train and Test Loss for Activations and Optimizers

The best test loss was 0.242, with SGD optimizer and Tanh activation, but the ELU activation was a close contender. SGD surprisingly performed better than Adam for this specific network configuration and dataset. Adam's best loss was 2.304, which is almost 10 times higher than SGD's. ReLU also performed well for SGD but did not beat Tanh and ELU. Sigmoid performed poorly with both SGD and Adam, and this is expected due to its vanishing gradient problem.

5 CNNs vs Dense Networks

The goal for this section was to develop a convolutional neural network (CNN) with the least amount of total parameters and one convolutional layer that matches or beats the optimized dense network created earlier. The final CNN architecture that was chosen had one convolutional layer of 16 output channels with a kernel size of 5, and then a fully connected dense layer of 64 nodes. The final dense architecture chosen was the bottleneck shape of 250, 300, and 200 neurons by layer. The CNN architecture was finalized with trial and error at first. I saw that with fewer parameters, the model did not train as well, meaning that it could not capture all the intricacies of the problem. I increased both the output channels and kernel size, leading to an improvement in loss. I also increased the number of neurons in the fully connected layer to account for the significant increase in output channels. Both models were trained with a learning rate of 0.1 for 20 epochs, and their loss functions are in Figures 4 and 5.

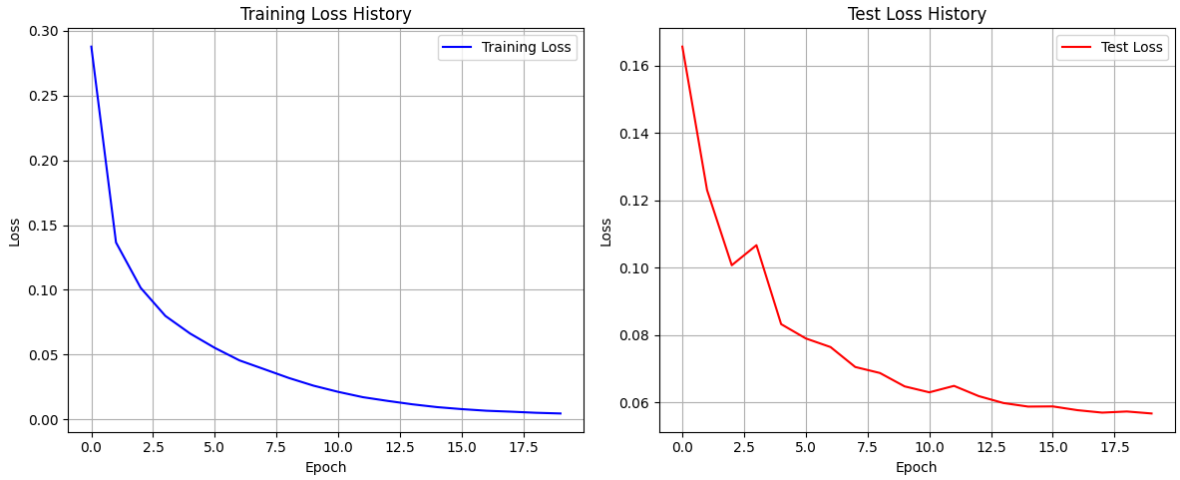

Figure 4: Train and Test Loss for CNN

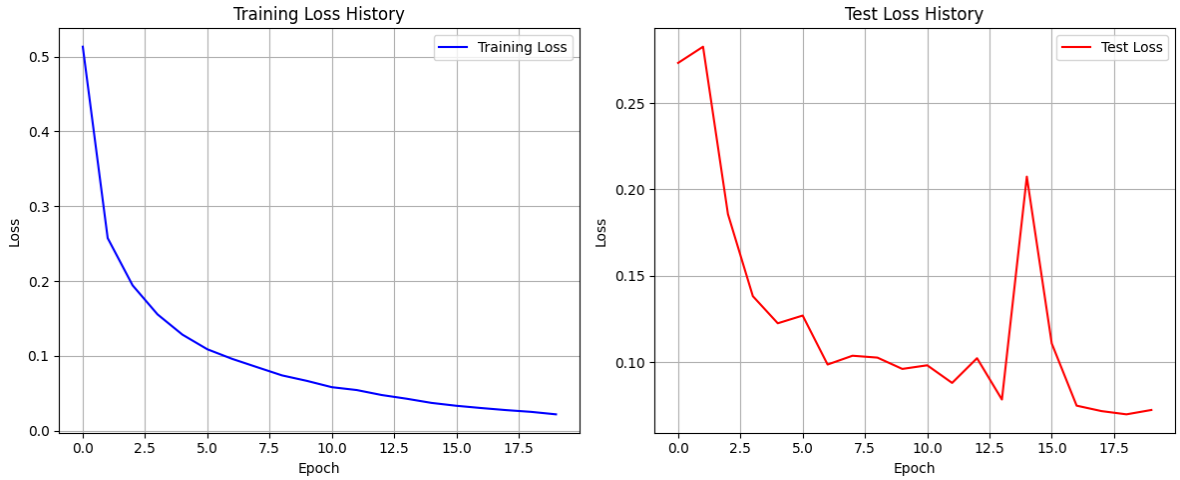

Figure 5: Train and Test loss Loss for Dense Networks

With the fully trained networks, I got the following results. Clearly, the CNN model performed better than the Dense model. But, both performed exceptionally so the marginal increase in test accuracy is not very important. An important note is that the CNN had significantly more parameters than the dense network. But, I experimented with CNNs with fewer parameters and was unable to beat the accuracy of the dense network, so I had to resort to this configuration.

Metric	CNN Model	Dense Model
Train Loss	0.0047	0.0219
Train Accuracy	0.9997	0.9934
Test Loss	0.0567	0.0722
Test Accuracy	0.9825	0.9787
Parameters	590954	333760

Table 4: Comparison of CNN and Dense Network Performance

Using two stacked convolutional layers can improve network performance while having fewer parameters. As a reminder, the original CNN architecture has one convolutional layer with 16 output channels and a 5×5 kernel. I designed a new CNN with 16 output channels linked to 8 output channels, both with 5×5 kernels. Both had a dense layer of 64 neurons. Both were trained with a learning rate of 0.1 for 20 epochs.

Metric	Value
Train Loss	0.0019
Train Accuracy	0.9999
Test Loss	0.0295
Test Accuracy	0.9873
Parameters	209138

Table 5: CNN with Two Convolutional Layers

The second CNN had marginally better test accuracy compared to the first CNN. But more importantly, the 2 layer CNN has almost three times fewer parameters than the 1 layer CNN. This is because the second, smaller convolutional layer processes and scales down the output of the first layer, resulting in a smaller layer to be flattened for the dense network. This reduces the parameters needed by the dense layer, creating a more powerful and efficient model.