An Exploration of Basic Methods for Handwritten Digit Classification

Brandon Szeto *
Jacobs School of Engineering
University of California, San Diego
San Diego, CA 92122
bszeto@ucsd.edu

Nathaniel Thomas †
Jacobs School of Engineering
University of California, San Diego
San Diego, CA 92122
nathomas@ucsd.edu

Abstract

The paper explores backpropagation, employing numerical approximation to verify accuracy. It introduces momentum in stochastic gradient descent, demonstrating its impact on model convergence and the trade-off with slight performance penalty. The study delves into regularization experiments (L1, L2) and observes the trade-off between convergence speed and accuracy. Activation function experiments (tanh, sigmoid, ReLU) highlight their impact on test accuracy and convergence speed, with tanh outperforming others in terms of both accuracy and speed. Adjusting learning rates addresses issues with ReLU's initial high rate. We also display the most difficult to classify digits. Overall, the findings contribute insights into optimizing neural networks.

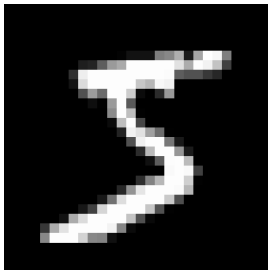


Figure 1: Sample MNIST Digit (5)

1 Data Loading

The dataset we will be using to train a multi-layer neural network is the MNIST dataset. The MNIST[3] dataset is a collection of grayscale handwritten digits commonly used for training various image

*160 lbs

†200 lbs

[3]*Modified* National Institute of Standards and Technology

Preprint. Under review.

processing systems. The dataset contains 60,000 training images and 10,000 testing images each normalized to 28×28 pixels.

1.1 Data Splitting

The MNIST dataset is split using an 80-20 ratio. Since the MNIST dataset contains 60,000 training images and 10,000 testing images, we expect to have 48,000 images and target labels in the training set and 12,000 images and target labels in the validation set. Optionally, the dataset is randomly shuffled and seeded for reproducability.

1.2 Normalization

The training and validation datasets are normalized by z-score.

$$\vec{z} = \frac{\vec{x} - \mu}{\sigma}$$

Here, $\vec{x}$ is the flattened ($28 \times 28 = 784$) vector, μ is the mean pixel value, and σ is the standard deviation over the single image. The resulting vector $\vec{z}$ (also size 784) contains the z-score normalization of the image where each value will range between -1 and 1 (unit variance) and have mean 0.

1.3 Mean and Standard Deviation

Consider an image of the digit 5[1]. Its vector representation is

$$\vec{x}^{(n)} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

where each of the values in $\vec{x}^{(784)}$ is some value between 0 for black and 255 for white. Its mean (μ) and standard deviation (σ) can be obtained by

$$\begin{aligned} \mu &= \frac{x_1 + x_2 + \ldots + x_n}{n} &&= 0.13714225924744897 \\ \sigma &= \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}} &&= 0.31112823799846606 \end{aligned}$$

These values can then be used to obtain $\vec{z}^{(n)}$, the z-score normalization of $\vec{x}^{(n)}$.

2 Softmax Regression

2.1 Background

Firstly, we perform softmax regression on a single-layer neural network (i.e. input layer directly connects to output layer). Given an input $x^{(n)}$ and c possible classes, softmax regression will output a vector $y^{(n)}$, where each element, $y_k^{(n)}$ represents the probability that $x^{(n)}$ is in class k.

$$\begin{aligned} y_k^{(n)} &= \text{softmax}(a_k^{(n)}) = \frac{e^{a_k^{(n)}}}{\sum_{j=1}^{N} e^{a_j^{(n)}}} \\ a_k^{(n)} &= w_k^\top x^{(n)} \end{aligned}$$

Here, the softmax activation function generalizes the logistic activation function for multiple classes. Now for *softmax regression*, we use a one-hot encoding for our targets. Together, we can calculate loss via the cross-entropy cost function defined as

$$E = -\sum_{k}\sum_{k=1}^{c} t_k^{(n)} \ln\left(y_k^{(n)}\right)$$

and its corresponding gradient

$$-\frac{\partial E^n(w)}{\partial w_{jk}} = \left(t_k^{(n)} - y_k^{(n)}\right) x_j^n$$

The above equations can be combined to implement the following delta rule to update the weights over each epoch.

$$\delta_j = (t_j^{(n)} - y_j^{(n)}) \qquad \text{for output unit } j$$
$$\delta_j = g'(a_j)\sum_k \delta_k w_{jk} \qquad \text{for hidden unit } j$$

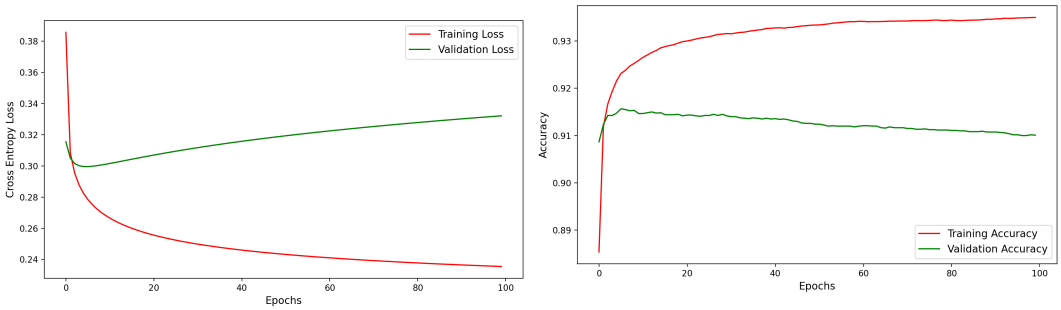
In pseudocode, we have

Algorithm 1 Stochastic Gradient Descent

$w \leftarrow 0$
for $t = 1$ to M **do**
  Randomize the order of the indices into the training set
  for $j = 1$ to N in steps of B **do**
    start = j
    end = j + B
    $w_{t+1} = w_t - \alpha \sum_{n=\text{start}}^{end} \nabla E^{(n)}(w)$
  end for
end for

2.2 Performance

In a single layer neural network as defined in `./config_4.yaml`, we obtain the following results using softmax regression over 100 epochs. We plot loss versus epochs [2a] and accuracy versus epochs [2b].



(a) Training/Validation Loss/Accuracy over 100 Epochs (b) Training/Validation Accuracy over 100 Epochs

3 Backpropagation

3.1 Numerical Approximation of Gradients

To verify the accuracy of our backpropagation implementation, we can compute the slope with respect to one weight using numerical approximation

$$\frac{d}{dw}E^n(w) \approx \frac{E^n(w+\epsilon) - E^n(w-\epsilon)}{2\epsilon}$$

We can perform this approximation for any given input to hidden weight w_{ij} and hidden to output weights w_{jk} for any type of perceptron including bias. In our experiments, we let $\epsilon = 10^{-2}$.

In the computation of numerical approximation, we compute the loss of $E^n(w_{ij} + \epsilon)$ and $E^n(w_{ij} - \epsilon)$ by performing a forward pass with the weight $w_{ij} = w_{ij} \pm \epsilon$ and reporting the loss. Here, we must be careful to only individually evaluate loss for a given weight with respect to the rest of the network. Otherwise, the reported loss is not necessarily loss with respect to the single weight w_{ij} or w_{jk} we want to observe.

3.2 Backpropagation Evaluation

In the table below [1] , we measure the approximate gradient (column 3) and the gradient computed via our implementation of backpropagation (column 2), and their absolute difference (column 4) for a given type of weight. The observed absolute difference is within $O(\epsilon^2)$ for all weight types.

Type of Weight	Actual Gradient	Approximation Gradient	Absolute Difference
w_{ij} (input to hidden)	-1.2282×10^{-3}	-1.2282×10^{-3}	6.6801×10^{-9}
w_{ij} (input to hidden)	-1.5063×10^{-3}	-1.5062×10^{-3}	8.3850×10^{-9}
w_{bj} (bias to hidden)	3.0450×10^{-3}	3.0449×10^{-3}	9.6145×10^{-8}
w_{jk} (hidden to output)	1.7730×10^{-2}	1.7730×10^{-2}	6.8067×10^{-9}
w_{jk} (hidden to output)	1.5636×10^{-2}	1.5636×10^{-2}	4.6279×10^{-9}
w_{bk} (bias to output)	-8.9961×10^{-1}	-8.9961×10^{-1}	1.2030×10^{-6}

Table 1: Gradient Values with Absolute Differences

4 Momentum Experiments

4.1 Background

We now consider a new network with a 785 neuron input layer, a 128 neuron hidden layer, and a 10 neuron output layer. The hidden layer is activated by $\tanh(\mathbf{x})$ and the output with softmax. We are training for 100 epochs.

We introduce the concept of *momentum*, which factors in the value of the previous gradient into the current Δ, which is the amount we adjust the weights by. The modified SGD algorithm includes a hyperparameter γ, which determines the strength of the momentum.

Algorithm 2 Stochastic Gradient Descent with Momentum

$w \leftarrow 0$
for $t = 1$ to M **do**
  Randomize the order of the indices into the training set
  $\Delta_0 = \mathbf{0}$
  for $j = 1$ to N in steps of B **do**
    start = j
    end = j + B
    $\Delta_{end} = \sum_{n=start}^{end} \nabla E^{(n)}(w)$
    $w_{t+1} = w_t - (\alpha\Delta_{end} + \gamma\Delta_{start})$
  end for
end for

Along with momentum, we implement *early stopping*. This technique allows us to stop training once performance stagnates on the validation set. Once we reach E consecutive epochs where the validation loss is lower than its highest value, we abort training and keep the weights with the lowest validation loss.

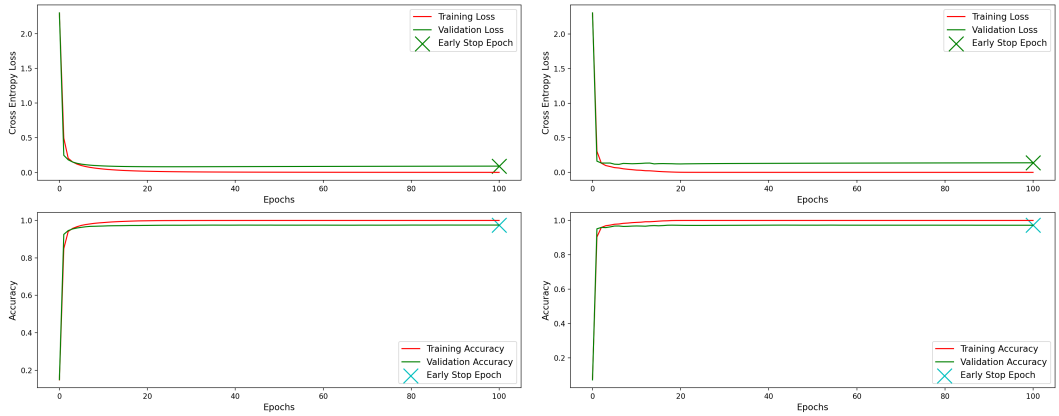
4.2 Performance

4.2.1 Initial

Our initial performance, with $\alpha = 0.001$, resulted in test accuracy of 97.68%[3a].

4.2.2 With Momentum

Adding momentum, with $\gamma = 0.9$, resulted in a slightly worse test accuracy of 96.88%[3b].



(a) Accuracy and Loss without momentum or early stopping

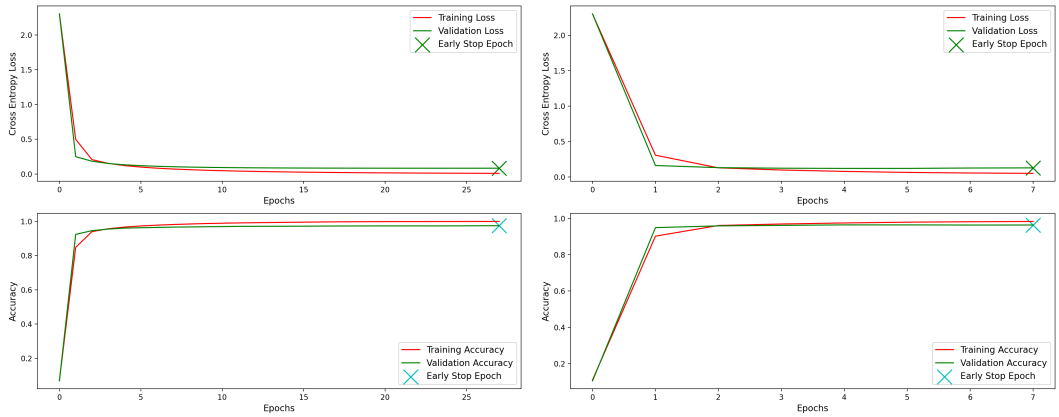
(b) Accuracy and Loss with momentum and without early stopping

4.2.3 With Early Stopping

If we have early stopping, with $E = 3$, but no momentum we get a test accuracy of 97.77% [4a].

4.2.4 With Momentum and Early Stopping

If we have both momentum and early stopping, with the same hyperparameters specified above, we get a test accuracy of 96.62%[4b].



(a) Accuracy and Loss with early stopping and without momentum

(b) Accuracy and Loss with early stopping and momentum

4.3 Observations

The test accuracy and the plots suggest that momentum accelerates model convergence. Comparing the two plots that used early stopping, we see that only 7 epochs were required with momentum

but 26 were required without. It does come with a slight penalty in model performance, however. We believe that this penalty is worth it, when accounting for the improved time and power usage of training.

Early stopping is an excellent technique to improve training times with the minor drawback of increased memory usage since an extra cache of the parameters must be stored. For small models, this is indispensible.

5 Regularization Experiments

Another hyperparameter that may help improve generalization is regularization. We introduce a new term in the computation of error, λC that measures the complexity of the model. Here, λ is the hyperparameter we tune (typically a small value) to adjust the strength of the regularization. Altogether, we have a new measure of error

$$J = E + \lambda C$$

Here, C can take various forms. Two well-known forms include L_1 and L_2 regularization.

L_1 regularization seeks to minimize $C = |W|$. Therefore,

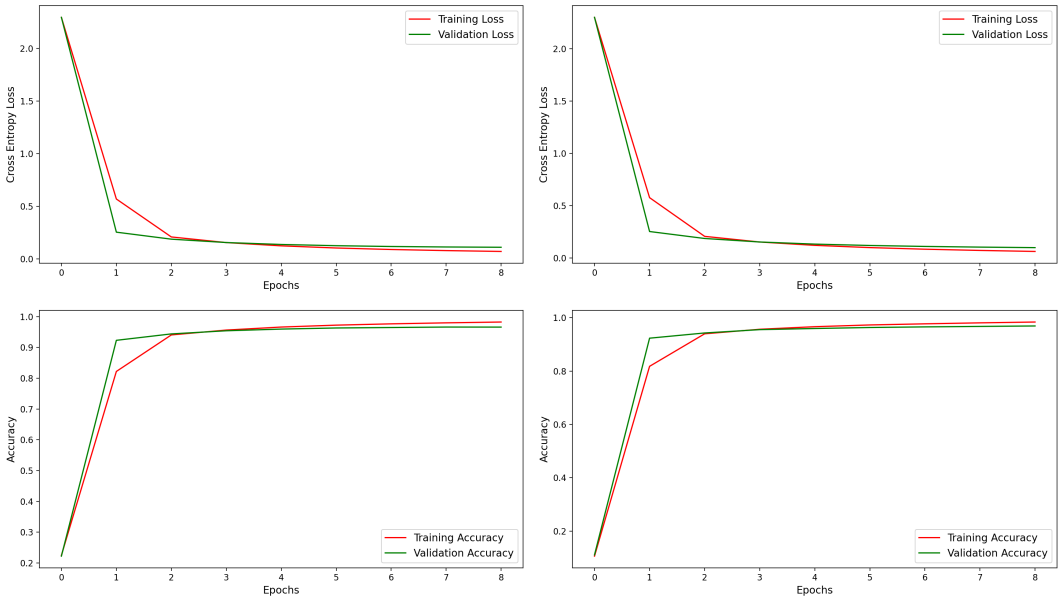
$$\begin{aligned}\frac{\partial J}{\partial w_{ij}} &= \nabla E^{(n)}(w_{ij}) + \lambda \frac{\partial C}{\partial w_{ij}} \\ &= \nabla E^{(n)}(w_{ij}) + \lambda \text{sign}(w_{ij})\end{aligned}$$

where the matrix of ones $\mathbf{1} \in \mathbb{R}^{i \times j}$. On the other hand, L_2 regularization seeks to minimize $C = \|W\|_2^2$.

$$\begin{aligned}\frac{\partial J}{\partial w_{ij}} &= \nabla E^{(n)}(w_{ij}) + \lambda \frac{\partial C}{\partial w_{ij}} \\ &= \nabla E^{(n)}(w_{ij}) + 2\lambda w_{ij}\end{aligned}$$

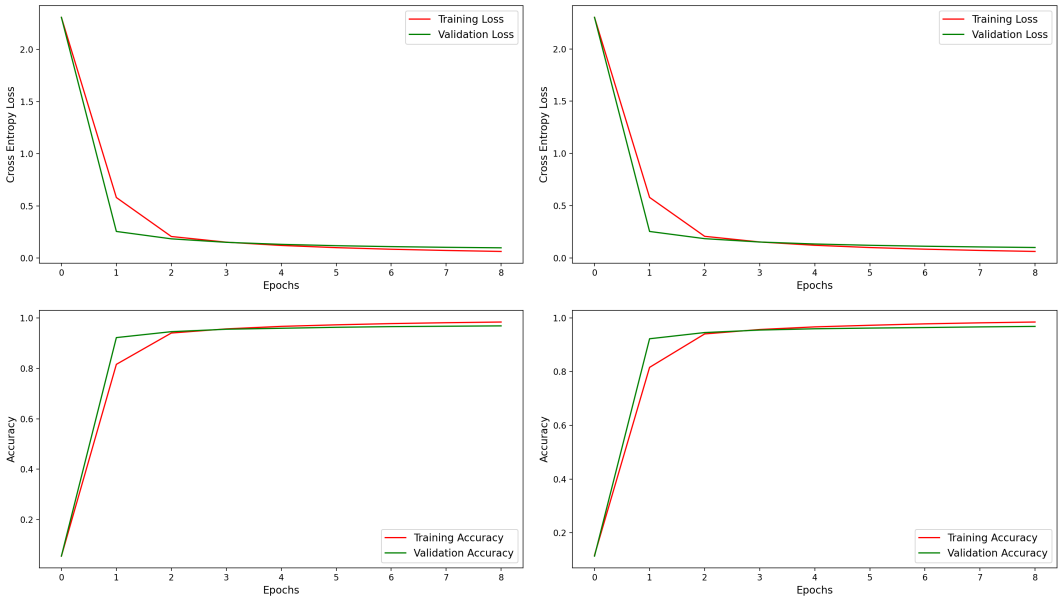
5.1 Experiments

In our previous experiments with momentum and early stopping, we achieved a test accuracy of 96.62% within 7 epochs. We will run our regularization experiments with approximately 10% more epochs (8 total epochs). We achieve the following results



(a) L_1 Regularization with $\lambda = 1e^{-2}$

(b) L_1 Regularization with $\lambda = 1e^{-4}$



(c) L_2 Regularization with $\lambda = 1e^{-2}$

(d) L_2 Regularization with $\lambda = 1e^{-4}$

5.2 Observations

Overall, we see a trade-off between the speed of convergence, and the accuracy of the model. Here, speed of convergence refers to number of epochs it takes to reach a patience limit (number of epochs where the validation error increases). In general, as the λ penalization term increases in both L_1 and L_2 regularization, the model takes longer to converge but reaches greater accuracy. As λ decreases, the model converges more quickly but at the expense of some accuracy. In our experiments, a value of $\lambda = 0.0001$ with L_2 regularization achieves the greatest accuracy.

This trade-off is related to how important minimizing the size of the weights is. In the case where λ is large, we spend more time searching for a minima given a set of small weights. In the case where

λ is small, we spend less time looking for a minima with small weights and pay more attention the gradient instead.

L_1 and L_2 5 differ in that L_1 changes by some constant regardless of how large or small the weights are, while L_2 changes by a factor of the magnitude of the weights. As a result, it is harder for L_1 regularization to converge, because there is no difference in the L_1 penalization regardless of how large the weights are. However, L_2 regularization changes proportionally to the weights, resulting in the heavy penalization of very large weights while less penalization of smaller weights.

6 Activation Experiments

6.1 Background

Neural networks utilize activation functions to introduce non-linearity into the model, enabling it to learn complex patterns and relationships in data. Three commonly used activation functions are `tanh`, `ReLU` (Rectified Linear Unit), and `sigmoid`.

6.1.1 Tanh Activation Function

The hyperbolic tangent function, $\texttt{tanh}(x)$, is defined as:

$$\texttt{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The derivative of $\texttt{tanh}(x)$ with respect to x is given by:

$$\frac{d}{dx}\texttt{tanh}(x) = 1 - \texttt{tanh}^2(x)$$

Properties of `tanh`:

- Range: $[-1, 1]$
- Zero-centered, which can aid in faster convergence during training.

6.1.2 ReLU Activation Function

The Rectified Linear Unit, `ReLU`(x), is defined as:

$$\texttt{ReLU}(x) = \max(0, x)$$

The derivative of `ReLU`(x) is:

$$\frac{d}{dx}\texttt{ReLU}(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases}$$

Properties of `ReLU`:

- Simplicity and computationally efficient.
- Prone to the "dying ReLU" problem, where neurons can become inactive during training.

6.1.3 Sigmoid Activation Function

The sigmoid function, $\texttt{sigmoid}(x)$, is defined as:

$$\texttt{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Its derivative is given by:

$$\frac{d}{dx}\texttt{sigmoid}(x) = \texttt{sigmoid}(x) \cdot (1 - \texttt{sigmoid}(x))$$

Properties of `sigmoid`:

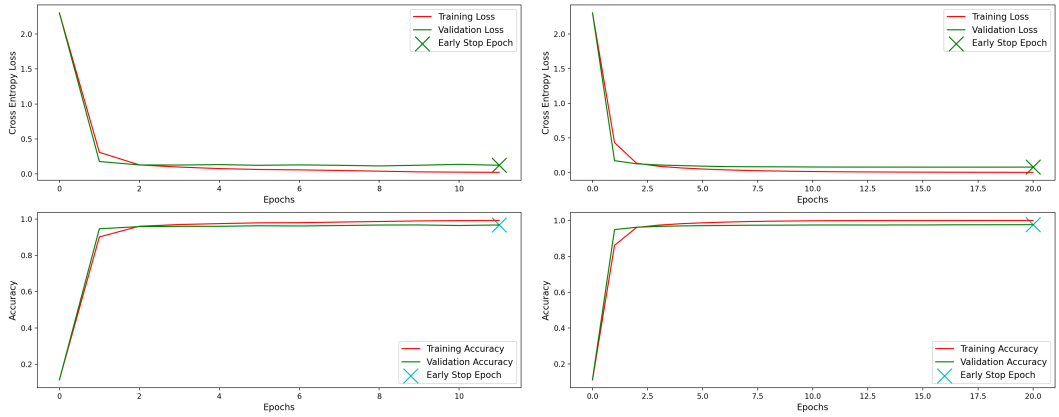
- Output range: $(0, 1)$
- Smooth gradient, facilitating gradient-based optimization.
- Susceptible to vanishing gradient problem.

6.2 Performance

We now consider the performance of our network with each activation function. We will continue to use early stopping with $E = 3$, momentum with $\gamma = 0.9$, and $\alpha = 0.001$. L1 and L2 normalization are not used.

6.2.1 Tanh

We get test accuracy of 97.15% and a stopping epoch of 10[6a].



(a) Accuracy and Loss using $\tanh(x)$ activation

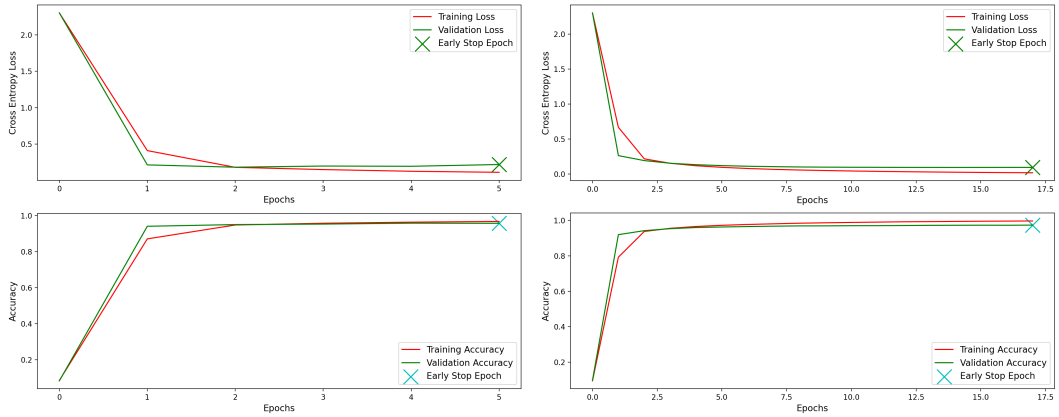
(b) Accuracy and Loss using $\texttt{sigmoid}(x)$ activation

6.2.2 Sigmoid

We get test accuracy of 97.66% and a stopping epoch of 19[6b].

6.2.3 ReLU

We get test accuracy of 95.61% and a stopping epoch of 5[7a]. The relatively low performance and early stopping epoch suggests the learning rate is too high. If we set $\alpha = 0.0001$, we get an accuracy of 97.56% on the test data and stopping epoch of 16[7b].



(a) Accuracy and Loss using $\texttt{ReLU}(x)$ activation, $\alpha = 0.001$

(b) Accuracy and Loss using $\texttt{ReLU}(x)$ activation, $\alpha = 0.0001$

6.3 Observations

With $\alpha = 0.001$ both sigmoid and tanh perform similarly on the test data, but tanh converges much faster. ReLU performs fine, but poorly compared to the other two. After seeing that ReLU was converging very fast, we tried reducing the learning rate by a factor of 10. This brought ReLU's performance up to speed, but reduced its covergence time.

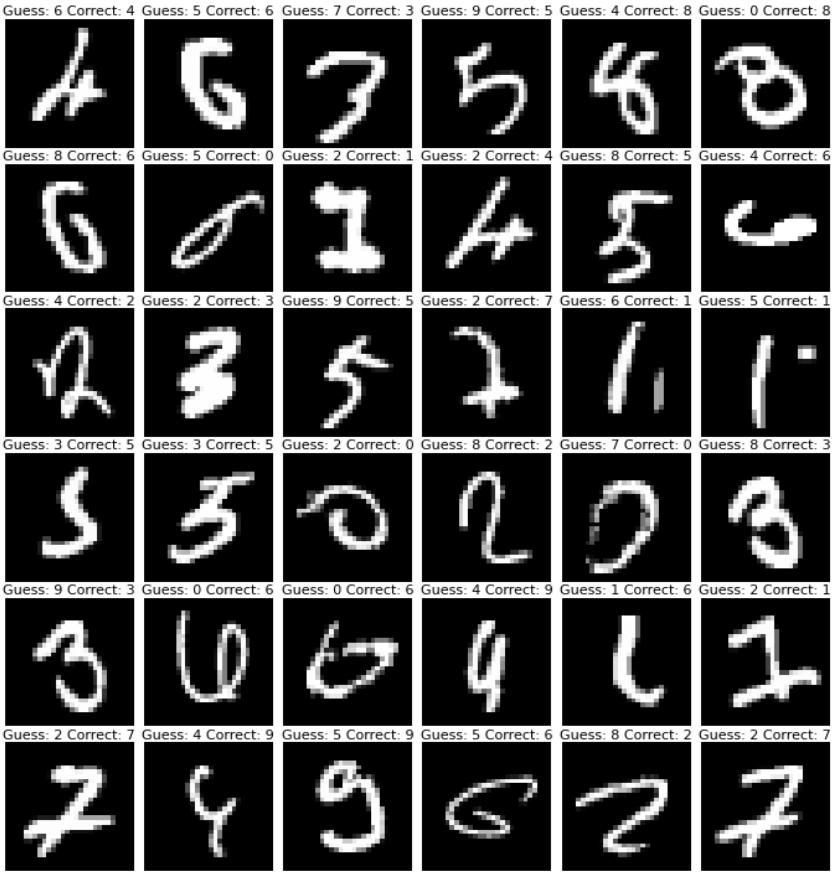


Figure 8: Digits in the test set with the highest loss

References

[1] *Lecture 3 backprop*. (2022) University of California, San Diego, pp. 23-28.

[2] *Lecture 4B tricks of the trade*. (2022) University of California, San Diego, pp. 23-28.

[3] *Lecture 4A improving generalization*. (2022) University of California, San Diego, pp. 3.