Deep Learning Project

Marwan MASHRA - Felix HERRON - Abdurrahman Shahid

April 7, 2022

1 Introduction

In this project, we aim to classify 28x28 grayscale handwritten digits from 0-9, using the benchmark MNIST dataset. We use a pre-determined train-test-validation split of 50,000 training images, and 10,000 test and validation images each. Thus, formally, on our dataset D:

$$D \subset \frac{\mathbb{Z}_{[0-255]}}{255}^{28*28\times 70{,}000} \tag{1}$$

and there are 14,049,280,000 possible images in our dataset. Here, $[0-255]$ means all integers between 0 and 255 inclusive.

We aim to learn a function to differentiate between digits using neural networks. Each neural network will take as input each image in vectorized form, or formally, $x \in \frac{\mathbb{Z}_{[0-255]}}{255}^{784}$, and produce ten logits, one for each possible digit. In other words, we are looking for a function $f : \frac{\mathbb{Z}_{[0-255]}}{255}^{784} \to \mathbb{R}^{10}$. We will readout the index of the logit with the highest value, and that will be our prediction.

To build our model, we must pick a loss function to minimize. We have chosen the cross-entropy loss for the following logic: we wish to pick the logit with the highest value compared to all of the other logits. Thus, it would make sense to maximize the difference between the gold logit and all the other logits. In order to standardize, we first employ the softmax function to the logits, so each has an activation between 0 and 1:

$$softmax(\vec{v})_i = \frac{e^{v_i}}{\sum\limits_j e^{v_j}} \in [0,1] \tag{2}$$

Hence, maximizing $softmax(v)_{gold}$ maximizes the difference between the gold logit and each other logit. The log function is strictly increasing; hence, maximizing the log function will be equivalent to maximizing the function without the log. Likewise, maximizing a function is equivalent to minimizing its opposite. Therefore, for input x and gold logit i, we choose to minimize the cross-entropy loss function:

$$L_{c-e}(x,i) = -\log \frac{e^{f(x_i)}}{\sum\limits_j e^{f(x_j)}} = -\log softmax(f(x))[i] \tag{3}$$

Thus, given our dataset, D we are looking for f^*:

$$f^* = argmin_{f=(W,b)} \sum_{x,i\in D} L_{c-e}(x,i) \tag{4}$$

2 Simple Linear Model

We start by implementing a simple linear model, which is a single layer perceptron.

2.1 Architecture

The architecture of a single layer perceptron is quite simple. Let $x \in \mathbb{R}^m$ be a given data point of size m (with m feature), $o \in \mathbb{R}^n$ be the probability distribution outputted by the network of size n (with n different possible class), $y \in \mathbb{R}$ be the final output of the network (the single digit), $z \in \mathbb{R}^n$ be an affine transformation parametrized by the weights $W \in \mathbb{R}^{n \times m}$ and the bias $b \in \mathbb{R}^n$. Then to obtain y, we start by applying the affine transformation:

$$z = Wx + b \tag{5}$$

Then we apply the softmax to z to obtain o

$$o = softmax(z) \tag{6}$$

Finally, the output y will be

$$y = argmax(o) \tag{7}$$

In our case, m is the size of the flattened images ($784 = 28 \times 28$) and n is the number of possible digits (10).

2.2 Training

Before the training, we initialize the parameters of the model W and b following a distribution law (such as in Xavier Glorot or Kaiming He Initialization). Then we start the training. The training of the network consists of a loop in which we preform a certain number of iterations (epochs) to optimize the parameters. In each iteration, we visit all data points once, updating the weights of the model to minimize the loss function using a gradient-based optimization algorithm. Here, we'll be updating the weights of the model once for each single data point. This process can be divided in two steps, the **forward pass** and the **backward pass**.

2.2.1 Forward Pass

During the forward pass, we take the data point x and pass it through the network, until obtaining the logits vector z.

$$z = Wx + b \tag{8}$$

2.2.2 Backward Pass

During the backward pass, we compute the gradient of the Loss function with respect to the parameters W and b, and apply gradient descent to update those parameters. In our case, we're using the negative log likelihood as a loss function:

$$L(z, gold) = -\log \frac{e^{z_{gold}}}{\sum\limits_j e^{z_j}} = -z_{gold} + \log \sum_j e^{z_j} \tag{9}$$

We know that the gradient of L with respect to W is equal to

$$\nabla_W L = \begin{bmatrix} \frac{\partial L}{\partial W_{1,1}} & \frac{\partial L}{\partial W_{1,2}} & \cdots & \frac{\partial L}{\partial W_{1,m}} \\ \frac{\partial L}{\partial W_{2,1}} & \ddots & & \\ \vdots & & \ddots & \\ \frac{\partial L}{\partial W_{n,1}} & \cdots & & \frac{\partial L}{\partial W_{n,m}} \end{bmatrix} \tag{10}$$

However, L is not directly written as a function of W. Therefore, to compute $\frac{\partial L}{\partial W_{i,j}}$ but as a function of z, we use the chain rule as following:

$$\frac{\partial L}{\partial W_{i,l}} = \sum_j \frac{\partial L}{\partial z_j} \cdot \frac{\partial z_j}{\partial W_{i,l}} \tag{11}$$

First, we need to compute $\frac{\partial z_j}{\partial W_{i,l}}$. Looking at 8, we see that z_j can be expressed as the following

$$z_j = \sum_k W_{j,k} x_k + b_j \tag{12}$$

We know that $\frac{\partial}{\partial W_{i,l}}(W_{j,k}x_k + b_j) = x_k$ only if $i = j$ and $l = k$, and is equal to 0 otherwise. As a consequence

$$\begin{aligned}
\frac{\partial z_j}{\partial W_{i,l}} &= \frac{\partial}{\partial W_{i,l}}(\sum_k W_{j,k} x_k + b_j) && (13)\\
&= \frac{\partial}{\partial W_{i,l}}(W_{j,l} x_l + b_j) && (14)\\
&= \begin{cases} x_l \,, & if\ i = j \\ 0 \,, & otherwise \end{cases} && (15)
\end{aligned}$$

From 15 and 11, we obtain

$$\frac{\partial L}{\partial W_{i,l}} = \frac{\partial L}{\partial z_i} \cdot x_l \tag{16}$$

And from 16 and 18, we have

$$\nabla_W L = \begin{bmatrix} \frac{\partial L}{\partial z_1} \cdot x_1 & \frac{\partial L}{\partial z_1} \cdot x_2 & \dots & \frac{\partial L}{\partial z_1} \cdot x_m \\ \frac{\partial L}{\partial z_2} \cdot x_1 & \ddots & & \\ \vdots & & \ddots & \\ \frac{\partial L}{\partial z_n} \cdot x_1 & \dots & & \frac{\partial L}{\partial z_n} \cdot x_m \end{bmatrix} \tag{17}$$

$$\nabla_W L = \begin{bmatrix} \frac{\partial L}{\partial z_1} \\ \vdots \\ \frac{\partial L}{\partial z_n} \end{bmatrix} \cdot \begin{bmatrix} x_1 & \dots & x_m \end{bmatrix} \tag{18}$$

$$\nabla_W L = \nabla_z L \cdot x^T \tag{19}$$

Now we just need to compute $\nabla_z L$. Looking at 9, we see that

$$\nabla_z L = softmax(z) - 1[i = gold] \tag{20}$$

Finally, we just have to update W using gradient descent

$$W^{t+1} = W^t - \gamma \cdot \nabla_z L \cdot x^T \tag{21}$$

With γ being the step size, and W^t the weights at the iteration t

Now, let's compute $\nabla_b L$ in the same way

$$\nabla_b L = \begin{bmatrix} \frac{\partial L}{\partial b_1} \\ \vdots \\ \frac{\partial L}{\partial b_n} \end{bmatrix} \tag{22}$$

$$\frac{\partial L}{\partial b_i} = \sum_j \frac{\partial L}{\partial z_j} \cdot \frac{\partial z_j}{\partial b_i} \tag{23}$$

Applying the same method from 13 to 15, we obtain that

$$\frac{\partial z_j}{\partial b_i} = \begin{cases} 1\ , & if\ i\ =\ j \\ 0\ , & otherwise \end{cases} \tag{24}$$

By substituting that in 23, we obtain

$$\frac{\partial L}{\partial b_i} = \sum_j \frac{\partial L}{\partial z_j} \cdot 1[i = j] \tag{25}$$

$$= \frac{\partial L}{\partial z_i} \tag{26}$$

And we conclude that

$$\nabla_b L = \nabla_z L \tag{27}$$

Then we update b as we did with W in 21

$$b^{t+1} = b^t - \gamma \cdot \nabla_z L \tag{28}$$

2.3 Momentum SGD

In addition to traditional gradient descent, we also implemented momentum SGD. The main point of this tweak is to speed up model convergence by choosing a descent direction that is more directly towards the target local minimum. It frequently comes to pass that the gradient descent direction is not the most direct way to the target - see figure 1. The basic idea is that gradients from the past several training iterations are taken into account, with decreasing weight the further back in time they are. This serves to potentially average out the loss in non-optimal directions and allow the use of a greater learning rate to more quickly converge.

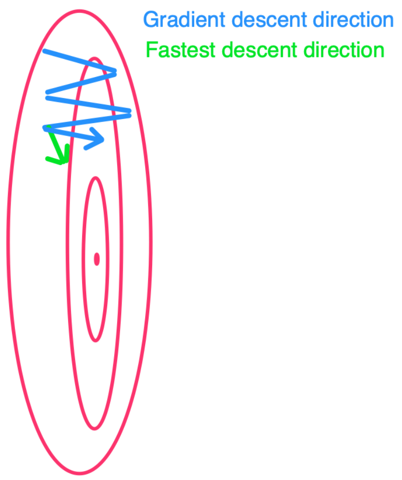
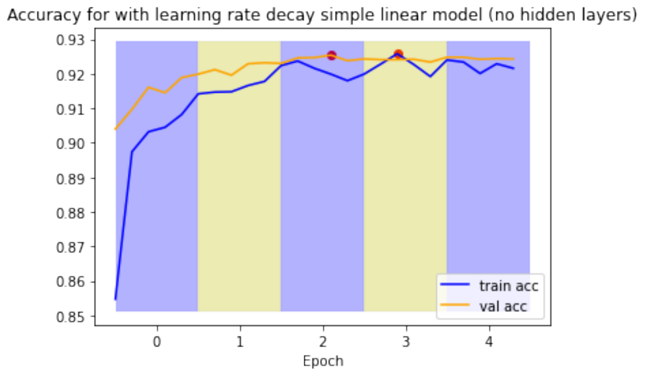


Figure 1: By choosing a weighted average of the past directions, the extraneous movement horizontally might be cancelled out

2.4 Training and Results

There were three principal hyperparameters which we sought to cross-validate: initial learning rate, whether to use a decreasing learning rate, and whether to use momentum SGD. We used the following elementary decreasing learning rate algorithm: if training loss decreased from the previous round, then halve the learning rate.

For each hyperparameter combination (training was sufficiently quick and there were sufficiently few combinations, 32 to be exact), we trained the model on the training data for five epochs. Five times per epoch, we evaluated the model on all three datasets - train,



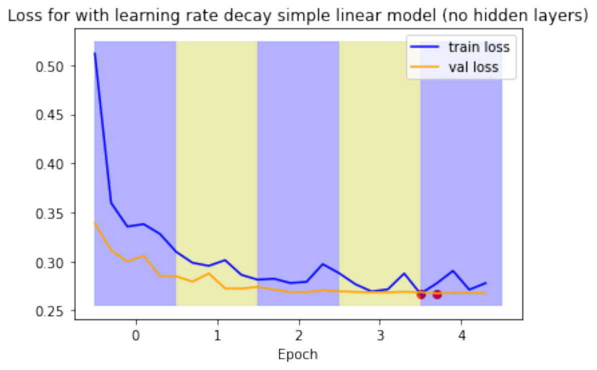


Figure 2: Loss decreased and accuracy increased as the simple linear model continued training. At some point, a threshold was reached, and no more improvement was made.

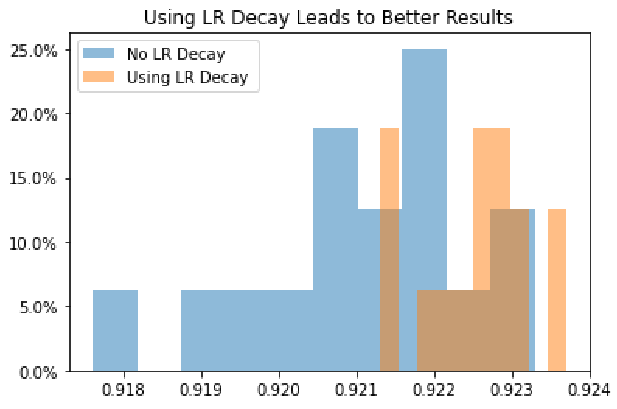


Figure 4: Using the momentum SGD optimizer caused a decrease in convergence time

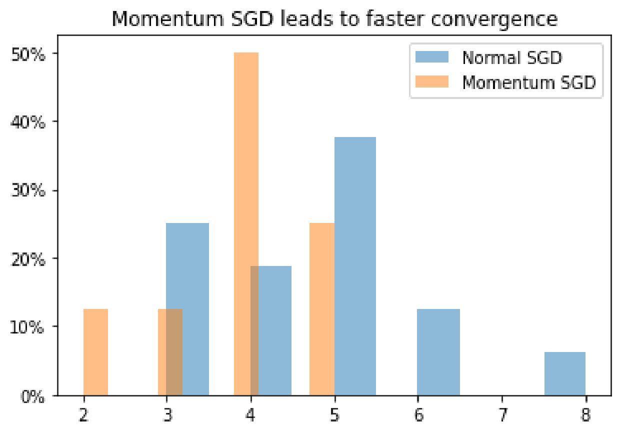


Figure 5: Implementing LR Decay led to better accuracy

validation, and test. After running, we determined which configuration of the hyperparameters produced the best validation results, and took the test result for those configurations as the model's score. This allowed us to avoid overfitting (by not necessarily taking the model with the highest training score), but also not taking an artificially high validation score (by re-evaluating on the test-set, we avoid cherry-picking).

As can be observed in figure 3, all of our models reached at least 91.8% testing accuracy, with a mean of 92.2%.

We also tried multiple learning rates, although there was no correlation between learning rate and testing accuracy.

Furthermore, we were able to see evidence of the momentum SGD causing an increase in converging time. Figure 5 shows the number of epochs which were required to converge to within 1% of the best accuracy for models which both used momentum SGD and those which did not. In the figure, one can also observe that it did not take all five epochs to reach the threshold accuracy for this model. No model needed more than two updates[1] to come within 1% of its maximal accuracy.

Lastly, we observed that using LR decay lead to better results, as the model was thus able to descend closer to its local maximum. This is illustrated in figure 4.

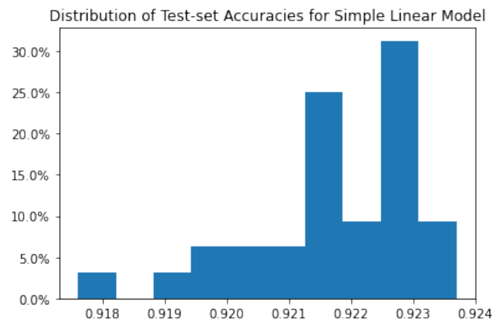


Figure 3: The classification accuracy for the simple linear model was very consistent across hyperparameter configurations

[1]We are measuring accuracy 5 times per epoch, so a value of 8 is within epoch 2

One remark on these results: we didn't have the time to replicate our trials several times and perform multiple iterations for each hyperparameter combination. That would have given us more data and allowed us to make these claims more strongly.

3 Deep Multi-layer Model

A deep neural network is a generalization of the simple linear network, which we discussed in the previous section. In deep neural networks, hidden layers of neurons are interpolated between the input and output layers. Our models are fully connected - that means for each hidden layer l^i and the layer before it l^{i-1} (sometimes l^{i-1} is the input layer), for each neuron l^i_n in l^i and each neuron l^{i-1}_m, there is a connection going from l^{i-1}_m to l^i_n. Like for the single-layer models, this model is parameterized by weights W, biases b, and (non-linear) activation functions for every neuron. What makes deep learning more powerful than the earlier linear models are the non-linear[2] activation functions. They allow deep models to learn more complex patterns, which linear models cannot. The more neurons in each layer, and the more layers, the more expressive the model can become. However, the increased complexity also leads to slower training time, reduced interpretability of the model's inner workings, and more increased susceptibility to overfitting.

In a deep classification model, the hidden layers are responsible for learning a transformation of the data points and transforming them into this space where they're linearly separable. Then, the last layer (aka the classification layer) is a linear model that will take the extracted features and learn to classify the data. The structure of the hidden layers varies based on the model we're building (Transformer, CNN, Multi-layer Perceptron). In our case, we are using a Multiplayer Perceptron.

3.1 Architecture

A Multi-layer Perceptron consists of one or several consecutively connected hidden layers which are affine transformations just like the single-layer perceptron. To compute the activation a^l_n of neuron n in layer l, we need as input the output of the previous layer, which transform using the appropriate weights and biases, as well as the non-linear function σ:

$$a^l_n = \sigma\Bigg(b^l_n + \sum_{m \in \text{previous layer}} (a^l_m * w^l_{n,m})\Bigg) \tag{29}$$

They all then forward their output to the next hidden layer, except for the last hidden layer, which forward its output to a single-layer linear model.

More formally, let $x \in \mathbb{R}^m$ be a given data point of size m (with m feature), $o \in \mathbb{R}^n$ be the probability distribution outputted by the network of size n (with n different possible class), $y \in \mathbb{R}$ be the final output of the network (the single digit), $z^{(i)} \in \mathbb{R}^n$ be the output of the layer i, which is a linear projection parametrized by the weights $W^{(i)} \in \mathbb{R}^{n \times m}$ and the bias $b^{(i)} \in \mathbb{R}^n$ to which we apply the activation function σ. $z^{(t)}$ the last output (linear) layer. Then to obtain y, we apply the affine transformation and the activation functions as following:

$$\text{Hidden Layers} \begin{cases} z^{(1)} = \sigma(W^{(1)}x + b^{(1)}) \\ z^{(2)} = \sigma(W^{(2)}z^{(1)} + b^{(2)}) \\ \quad\vdots \\ z^{(t-1)} = \sigma(W^{(t-1)}z^{(t-2)} + b^{(t-1)}) \end{cases} \tag{30}$$

[2]the function must not necessarily be linear, but if it is not linear, the model can be algebraically reduced to a simple linear model and hence is no more powerful than one

$$\text{Similar to a single-layer model} \begin{cases} z^{(t)} = & W^{(t)} z^{(t-1)} + b^{(t)} \\ o = & softmax(z^{(t)}) \\ y = & argmax(o) \end{cases} \tag{31}$$

3.2 Implementation

In practice, the deep network was implemented as several connected components (Tensors). Each tensor receives an input from the previous tensor, and sends its output to the next tensor (see Figure 6). Those tensors can be computational components (Linear projection, Relu, Hyperbolic tangent, Negative Log Likelihood...etc), or parameters of the model (Weights, bias).

Furthermore, each tensor (that is not a parameter) has a backward method that varies according to the type of tensor (Relu, Loss, Tanh..Etc). This method computes the gradient of the Loss function w.r.t to the tensor, and then passes this gradient as an incoming gradient to the tensor behind. This process is called back propagation and will be discussed later in the **backward pass**.

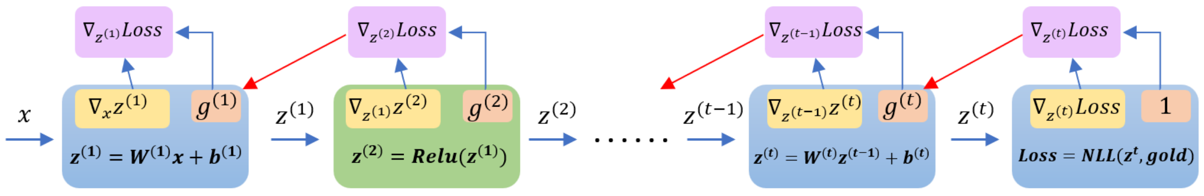


Figure 6: Implementation of the Deep Network

3.3 Training

The training of a multi-layer perceptron is similar to the training of a single-layer perceptron. We do a certain number of iterations (epochs) over all data points using (stochastic) gradient descent to update the parameters of the model in the opposite direction of the steepest increase in model's loss. However, since we have several layers, we'll be updating the parameters of each layer recursively using the back propagation algorithm.

3.3.1 Forward Pass

During the forward pass, we take the data point x and pass it through each layer (or tensor in our case) of the network, until obtaining the logits vector (z^t in Figure 6 and in Equation 31).

3.3.2 Backward Pass

During the backward pass, the gradient of the Loss function is computed w.r.t each parameter W^i, b^i. This is done through back propagation and was implemented as illustrated in the figure 6. The gradient of the Loss function w.r.t the tensor is computed using the incoming gradient of the front tensor. Then, the tensor passes the gradient to the tensor behind as an incoming gradient.

More formally, the gradient of each tensor is computed as following:

If the tensor is a linear projection

$$\nabla_{z^{(i)}} Loss = < W^{(i+1)}, g^{(i+1)} > \tag{32}$$

If the tensor is the negative log likelihood, then the incoming gradient will be 1

$$\nabla_{z^{(i)}} Loss = softmax(z^{(i)}) + \tag{33}$$

If the tensor is a Relu

$$\nabla_{z^{(i)}} Loss = g^{(i)} \cdot v \tag{34}$$
$$\tag{35}$$

(with v being a vector of size $\dim(z^{(i-1)})$, with 1 if $z^{(i-1)_j} > 0$ and 0 otherwise)

If the tensor is a hyperbolic tangent

$$\nabla_{z^{(i)}} Loss = g^{(i)} \cdot (1 - tanh(z^{(i-1)})^2) \tag{36}$$

Then to update the parameters using (stochastic) gradient descent, we compute their gradient as following:

$$\nabla_{W^i} Loss = g^{(i)} \cdot z^{(i-1)} \tag{37}$$
$$\nabla_{b^i} Loss = g^{(i)} \tag{38}$$

3.4 Results

In addition to the hyperparameters which we cross-validated for our simple linear model, here we add three more: number of hidden layers, hidden layer size, and activation function. However, for the deep model, training time took much longer and the number of possible hyperparameters grew large. Hence, we did a randomized cross validation rather than a grid search, testing random permutations of hyperparameters. We also did trials by varying only one variable and keeping everything else fixed, to determine some correlations, as elaborated upon further in this section. We had the idea too late, but next time it would be time-saving to do this cross-validation on smaller training sets to speed up training; this would still give a reasonable impression about where hyperparameter local maxima are.

In general, and not surprisingly, we got significantly improved results for our deep learning model -for example, consider figures 8. Our best result was 98% accuracy.

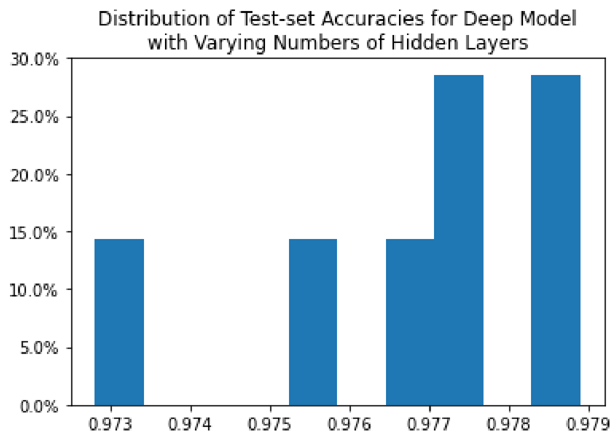
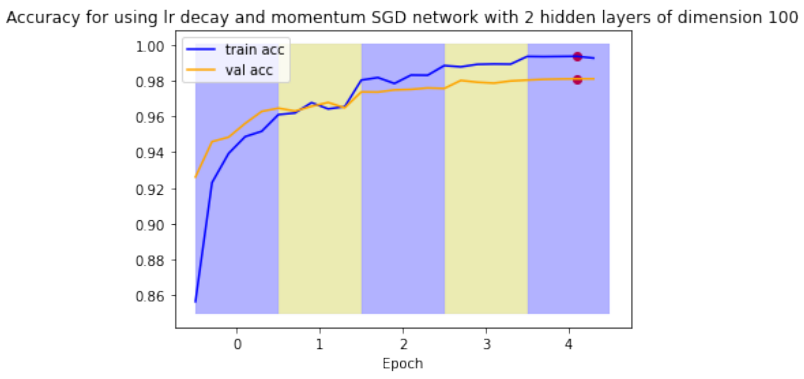


Figure 7: Our results improved with the more complex model.



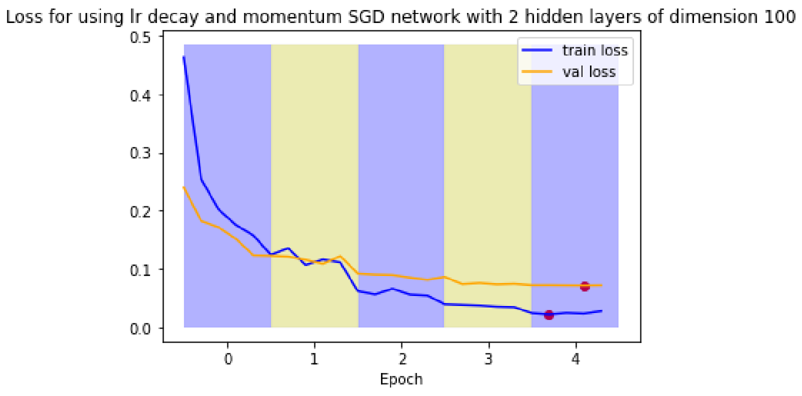


Figure 8: Loss decreased and accuracy increased as the deep model continued training. At some point, a threshold was reached and no more improvement was made.

Because of lack of time and resources to cross-validate everything, based on observations from the simple linear model, we fixed the optimizer to momentum SGD and learning rate decrease to true; we took learning rate of 0.1 as it seemed not to make a big difference, particularly given decreasing LR.

We had several hypotheses about other hyperparameters which we wanted to test. For example, whether the number of neurons per hidden layer had an impact on the test accuracy. Indeed, as the number of neurons per hidden layer increased, so did test accuracy - see figure 9. The increase continued through 100 neurons per layer - unfortunately, we didn't have time to test whether this trend continues. Increasing the size of layers also increases the chances of overfitting, so probably this trend wouldn't continue forever.

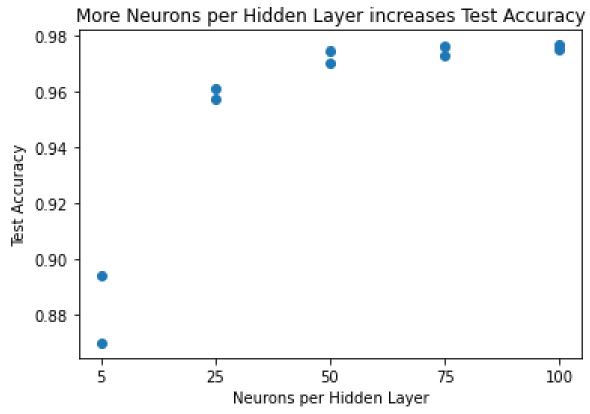


Figure 9: For a test of two hidden layers, using other boilerplate parameters, increasing the number of hidden layers increased test accuracy. Furthermore, the sighly higher scores observed are from ReLU, while the lower scores are from tanh.

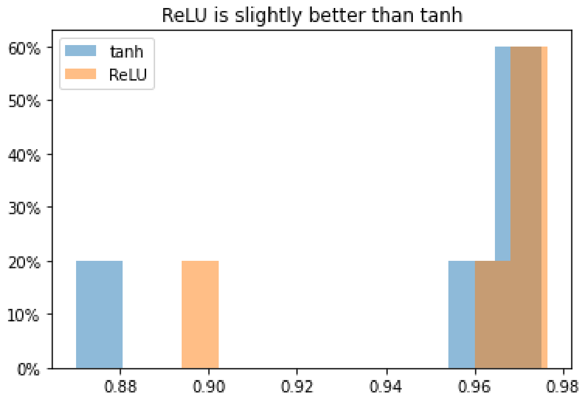


Figure 10: Loss decreased and accuracy increased as the deep model continued training. At some point, a threshold was reached and no more improvement was made.

We also were curious about the merits of using tanh vs relu as activation functions. Figures 9 and 10 display how ReLU performed slightly better in our trials than tanh.

As for the previous model, we did not have the time and computing resources to do as thorough a cross-validation as we would have liked. It is possible that a denser network with more hidden layers and more neurons per layer, trained for more epochs, could have gotten a test score higher than 98%.