Feed Forward Networks for MNIST
CSE 253: Deep Learning, Winter 2019

Osman Cihan Kilinc
Electrical and Computer Engineering
University of California, San Diego
okilinc@ucsd.edu

Gokce Sarar
Electrical and Computer Engineering
University of California, San Diego
gsarar@ucsd.edu

Abstract

In this homework, we implement a neural network in a modular fashion. We implement a sanity check method to check gradients from backpropagation. Early stopping and weight decay are used as regularization methods. Momentum is utilized to accelerate the convergence to the minimum especially with minibatches. We experiment with different hyperparameters for L2 penalty and learning rate. Sigmoid, Tanh and ReLU are implemented to experiment with different kinds of activations. Lastly, we demonstrate how the changes in the topology of the network affect the training and test accuracy.

1 Data Loader

How we read the MNIST data using the "load_data" function provided in the code can be seen in our code submission.

2 Backpropagation Sanity Check

The backpropagation algorithm or method is a mere implementation of the chain rule for the neural networks. The training of the neural networks is achieved by utilizing gradient descent to a sum-of-squares error function. Using the chain rule, we carry the gradient updates from the output layer to the input layer. The weights at every layer are adjusted according to these gradient updates. This process is repeated every step to minimize the error function.

An alternative approach can be used to evaluate the gradient updates of the backpropagation method. This mathematical equation for this alternative approach is given in the homework. For each selected weight, we perturb the weight with a small quantity ($\epsilon << 1$). The selected weight is incremented and decremented with ϵ. Then, loss is computed for both incremented and decremented values. The symmetrical central difference between the losses are used to provide a powerful check on the correctness of the backpropagation algorithm.

As asked in the homework, we randomly selected the 2 weights from input-to-hidden layer, 2 weights from hidden-to-output layer, 1 bias from input-to-hidden layer and 1 bias from hidden-to output layer. For each case, we repeat the sanity check explained above. Our results can be seen on Table 1. Our backpropagation implementation successfully passed all tests.

3 Model Training, Testing and Experiments

In this part, we use tanh activation function. The network has three layers: an input layer of 784 units, a hidden layer of 50 units and a softmax output layer of 10 units, since we have 10 digits.

Table 1: Backpropagation gradient checks

Layer Name	Backpropagation	Gradient Approximation	True/False
Input to Hidden (weight 1)	2.1217563176281814e-05	2.1218977108716786e-05	True
Input to Hidden (weight 2)	1.3735426227354431e-16	0.0	True
Input to Hidden (bias)	-1.0340434453508026e-15	0.0	True
Hidden to Output (weight 1)	2.8079369827317757e-05	2.807983742192377e-05	True
Hidden to Output (weight 2)	0.0001966908818834848	0.00019669414076384228	True
Hidden to Output (bias)	-0.0960296819566421	-0.09602962131118886	True

Using the vectorized update rule we obtained from 1(c), we performed mini-batch stochastic gradient descent to learn a classifier that maps each input data to one of the labels t 0, ..., 9, using a one-hot encoding. We used 50 hidden units. As instructed we used a momentum term weighted by 0.9. In momentum update rule, we have a velocity term as follows:

$$v(t) = \alpha v(t-1) - \eta \frac{\partial E}{\partial \mathbf{w}}$$

where the α is the momentum weighting as 0.9 and η is the learning rate. Velocity term equals the weight change: $\Delta \mathbf{w} = v(t)$. So the update rule is

$$\mathbf{w}(t) = \mathbf{w}(t-1) + \Delta \mathbf{w} = \mathbf{w}(t-1) + \alpha v(t-1) - \eta \frac{\partial E}{\partial \mathbf{w}}$$

Since the gradients as layer.dw are $-\frac{\partial E}{\partial \mathbf{w}}$, in the code we have

$$\mathbf{w}(t) = \mathbf{w}(t-1) + \Delta \mathbf{w} = \mathbf{w}(t-1) + \alpha v(t-1) + \eta \ layer.dw$$

Same is also applied for the bias.

We used early stopping with 3 consecutive epochs, meaning that if the validation error increased in 5 consecutive epochs, we stop the training (in some applications being consecutive is not necessary and earlystop counter begins counting once the validation loss gets bigger and doesn't reset to 0 when it goes down, like in Keras, but as indicated in Piazza we looked for being consecutive). We chose 3, since we are looking for consecutiveness, if we used 5 as given in the starter code, it was overfitting but due to oscillations, the algorithm was not stopping the training.

For the training procedure, we don't normalize the gradient with respect to batch size, since it was required to pass the checker.py. That's why the batch size affects our learning rate, since normalizing with the batch size can be thought as dividing by a scalar, which is equivalent to dividing the learning rate. So when we don't have this normalizations, we need to use relatively smaller lerning rate. For this part with tanh, we experimented with 10, 20, 50 and 100 and the best batch size for tanh was 20. With this batch size we used a learning rate of 0.001. Due to the early stopping, the training continued for 93 epochs. We shuffle the training data in every epoch. In Fig. 1, the accuracy for the training data and the validation data can be seen on the left, where as on the right the loss for them can be seen. The accuracy on test set using the best weights obtained through early stopping is 93.45%.

4 Experiments with Regularization

Regularization yields a better generalized model and prevents overfitting. A good analogy is trying to fit a square in to a circle. If the square is too small, than there will be a lot of area left outside the square. If the square is too big, then obviously it is impossible to fit the square into the circle. But if length of the square is equal to the circle's diameter, then we can smooth corners of the square to completely fill the circle. Regularization provides us the tools to achieve smoothings. Weight decay is one such tool.

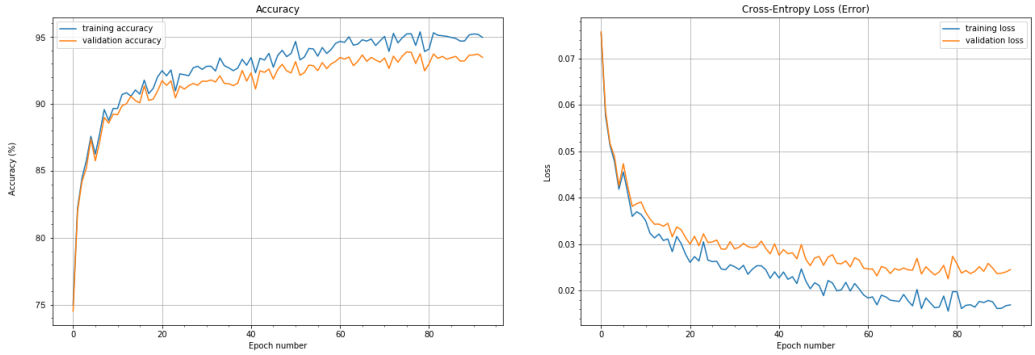

Figure 1: The results of training with tanh nonlinearity. (Left) The accuracy over 93 training epochs where training is stoppped with early stopping, along with the validation accuracy. (Right) The training and validation loss

By using L2 penalty, the weights are encourage to be small. This is achieved by simply adding a L2 penalty term to the update rule for each weight. The update rule is rewritten as:

$$w_{t+1} = w_t - \eta \frac{\partial E}{\partial w_t} - \eta \lambda w_t,$$

where λ is the L2 penalty and η is the learning rate.

Table 2 demonstrates the test accuracies on our experiments. Both Tanh and ReLU activation functions were used to observe the effects of regularization. Batch size was 100 for all experiments conducted. As required in the homework, 10% more epochs were used during training. Compared to no regularization, there is a considerable increase in accuracy when regularization is used. It can also be observed that different activation functions require different hyperparameters to achieve the best test accuracy. The graphs obtained for the setting with the highest test accuracy are shown in figure 2. For this particular setting, early stopping was triggered after 5 consecutive decreases in validation accuracy. However, we still reached the highest accuracy with this default architecture.

Table 2: Experiments with regularization

Activation	**Learning Rate**	**L2 Penalty**	**Accuracy**(%)
ReLU	0.1	0.00001	96.75
ReLU	1	0.00001	95.41
ReLU	0.01	0.0001	95.55
Tanh	0.01	0.0001	96.42
Tanh	0.1	0.0001	97.12
Tanh	1	0.0001	94.57

5 Experiments with Activations

Since we are asked to start with the network of part c, again in this part we didn't use regularization, but only momentum with gamma of 0.9 and the early stopping with 3 consecutive epochs to be consistent with part c. First, we will report the training results with sigmoid and ReLU and then compare the nonlinearities. In our code, for not using regularization, config["L2_penalty"] should be set to 0.

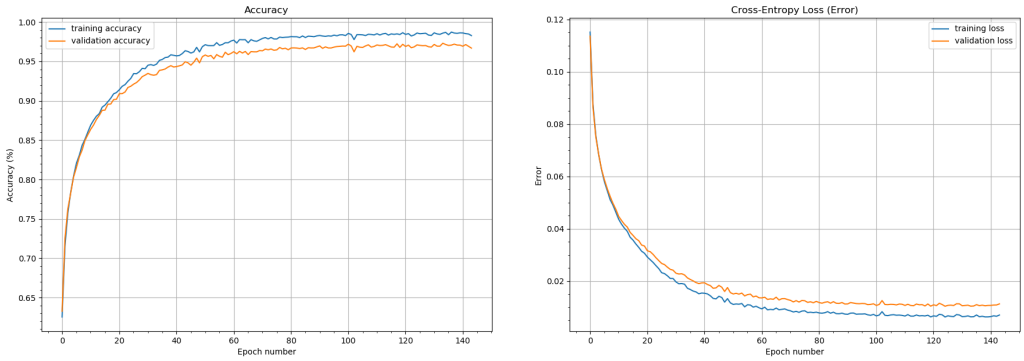

Figure 2: Results using the best settings for Tanh. (Left) The accuracy over 144 training epochs where training is stoppped with early stopping, along with the validation accuracy. (Right) The training and validation loss

5.1 Sigmoid

For sigmoid, the best combination of batch size and the learning rate was with a learning rate of 0.001 and a batch size of 10. Early stopping (consecutive 3 epochs) stopped the training at 127th spoch. We shuffled at each epoch. In Fig. 3, the accuracy and loss for the training and the validation set can be seen. The test accuracy is evaluated as 94.62%

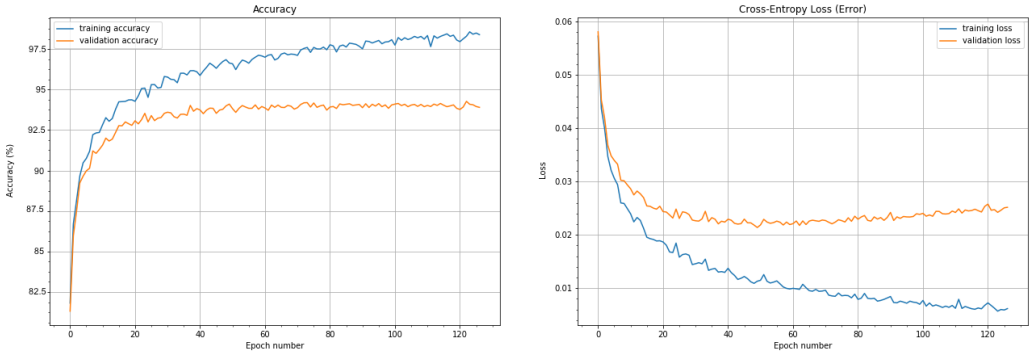

Figure 3: The results of training with sigmoid nonlinearity. (Left) The accuracy over 127 training epochs where training is stoppped with early stopping, along with the validation accuracy. (Right) The training and validation loss

The first thing to be noticed in this figure is the severe overfitting, which was less noticable with tanh nonlinearity. Although we got a better result (around 1% better) with sigmoid, we can't conclude that it is necessarily better, because of the noticeable overfitting, which was not that severe with tanh. By changing the early epoch number, tanh could get better results (we wanted to use the same early stopping for all of them and increasing it would end up having a more severe overfitting with sigmoid. So setting this threshold, we experimented with different batch sizes and learning rates and report here the best result and among those sigmoid had a slightly better accuracy).

In the literature it has been shown that tanh is a better non-linearity, since not all of its outputs are nonnegative. With sigmoid, all of the outputs are nonnegative and it leads the next layer to have all non-negative inputs. As we saw in the lecture from one output unit of a layer all the δs would have the same sign for all of the weights and if all the inputs are positive, then weight changes can be

either all positive and negative, which leads the optimization to fail. Nevertheless, for this specific problem, we couldn't see the superior effect of tanh. Nonetheless, since we have only one hidden layer and since MNIST dataset is fairly easy to classify, the fact that we didn't see superiority of tanh can be understandable. The deeper the network gets, the more important the characteristics of the activation we use.

5.2 ReLU

For ReLU the most obvious thing was the weight explosion with large learning rate, where in our case we couldn't use 0.001. That's why we needed to decrease the learning rate to 0.0001 and play with the batch size to get the optimum result. Regularization would have been another solution to this problem, but since with arranging batch size and learning rate we got a decent result and it is not required, we didn't experiment with regularization. For ReLU we used a batch size of 50. The training ended at 124 epochs with an early stopping looking for validation loss increase in 3 consecutive epochs. Again we shuffled in every epoch. In Fig. 4, the accuracy and loss for the training and the validation set can be seen. The test accuracy is evaluated as 93.3%

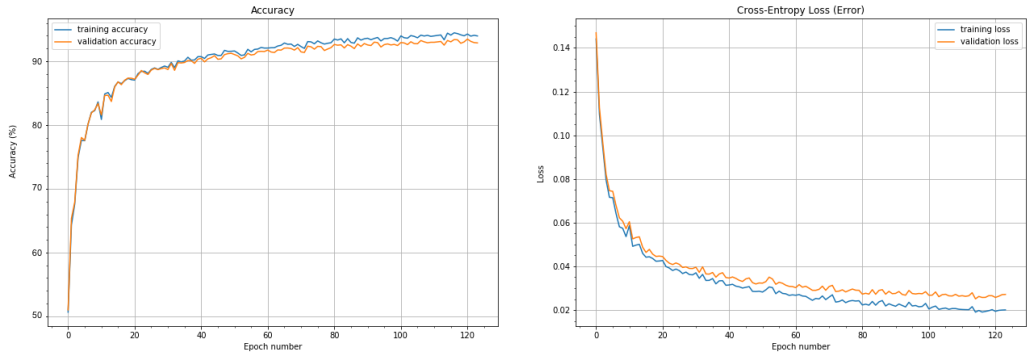

Figure 4: The results of training with ReLU nonlinearity. (Left) The accuracy over 124 training epochs where training is stoppped with early stopping, along with the validation accuracy. (Right) The training and validation loss

Again, in the literature, ReLU is known to be better than both tanh and sigmoid since it prevents vanishing gradient. Both sigmoid and tanh has a bell-shape derivative having maximum at $x = 0$ (x is the input of the nonlinearity), where the maximum of sigmoid derivative is 0.25 and maximum of tanh derivative is 1. So the derivative of sigmoid has a very small derivative for all x values whereas the derivative of tanh has a smaller than 1 vallue for almost all of the x values. In a deep network these derivatives are getting multiplied multiple times for each layer in backpropagation and bby the time backpropagation reaches the input, the gradient becomes so small that it doesn't affect the weights for update anymore. With ReLU this can be solved since it has a 0 derivative for negative x values and a derivative of 1 for the non-negative c values. So it is obvious that having a constant derivative value which is equal to 1 for all positive numbers would solve the vanishing gradient problem in deeper networks (moreover leaky ReLU is preferred over ReLU, since it also have a nonzero constant gradient for negative x values). So till here it is clear that ReLU would be very beneficial in deep networks.

Nevertheless in a shallow network as we have here, vanishing gradient problem is not that simple, so that ReLU doesn't improve the results significantly, but moreover it causes weight explosion. ReLU is not bounded as sigmoid or tanh, so that if its input is positive and very large, the output will be also very large, which will affect the backpropagation since this large output will be multiplied with the upcoming delta to propagate the error to the input layer. So since ReLU is not bounded, we should either use smaller learning rate or regularization. In this problem we solved this issue with small learning rate and got a comparable result to tanh and sigmoid. In Fig. 4, it seems to be a better

characteristic against overfittinh, but it is misleading since its learning rate was smaller. Overall, this assignment was not appropriate for us to see the superiority of ReLU.

5.3 Comparison

In this part, we just aimed to do a fair comparison among all activations, where we hold everything equal, but just change the activations. We trained for 500 epochs without early stopping, hold the best weight according to the lowest validation loss. We used a batch size of 50, a learning rate of 0.0001 and momentum update rule with the momentum weight as 0.9. The accuracy and loss plots can be seen in Fig. 5, 6 and 7. The test accuracy with the best weights can be seen below in the Table 3.

Table 3: Backpropagation gradient checks

Activation	**Accuracy**(%)
ReLU	94.35
Tanh	93.47
Sigmoid	91.65

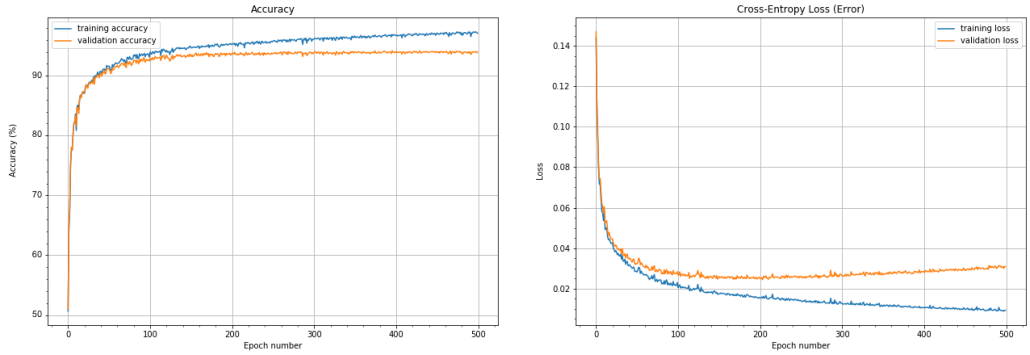

Figure 5: The results of training with ReLU nonlinearity. (Left) The accuracy over 500 training epochs where training was not stoppped with early stopping since it is set to 5 consecutive epochs, along with the validation accuracy. (Right) The training and validation loss

As can be seen ReLU gave the best results and tanh gave the worst results with this training procedure, whereas out expectation for tanh to be higher than sigmoid. Since we pass all checker code and also get a very high accuracy, we don't think that tanh is giving worse result due to implementation, but just this network is shallow that we can't see it's superiority over sigmoid. A second thing we observed is the fact that the training loss with ReLU oscillated more than sigmoid and tanh, although these oscillations are not severe and small in magnitude. It might be due to the fact that ReLU is not bounded like tanh and sigmoid so that even with a small learning rate the weight change with mini-batches is bigger than sigmoid and tanh and is enough to result in those oscillations. Lastly, ReLU showed a larger overfitting, where we can see in the plot that validation loss gets larger over epochs.

6 Experiments with Network Topology

For every task, the network topology is one of the most important considerations. Network topology should be according to the complexity of the task. Bigger graphs have more coefficients, which might engender overfitting. At first, all experiments were done with the default network that consisted of only one hidden layer with 50 neurons. In this part, we try various topologies. Batch size

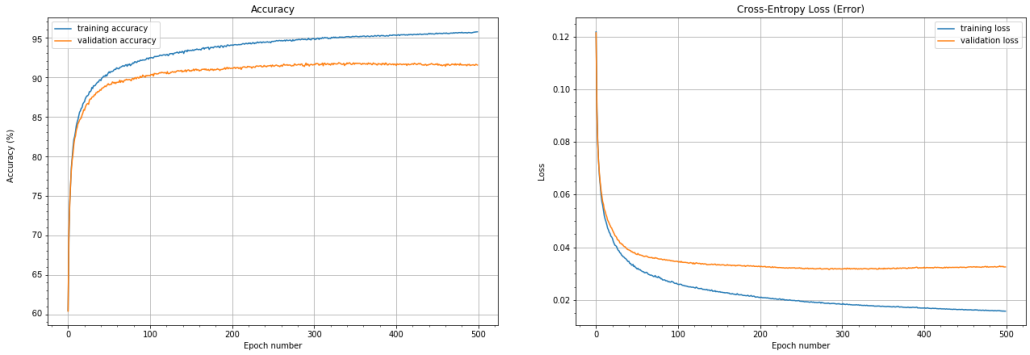

Figure 6: The results of training with tanh nonlinearity. (Left) The accuracy over 500 training epochs where training was not stoppped with early stopping since it is set to 5 consecutive epochs, along with the validation accuracy. (Right) The training and validation loss

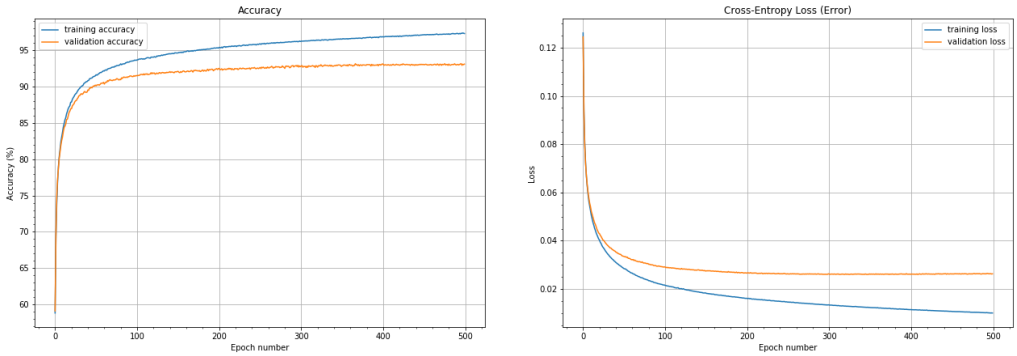

Figure 7: The results of training with sigmoid nonlinearity. (Left) The accuracy over 500 training epochs where training was not stoppped with early stopping since it is set to 5 consecutive epochs, along with the validation accuracy. (Right) The training and validation loss

was 100 for all experiments conducted. When we decrease the number of neurons by half in the hidden layer, the test accuracy decreases about 1% to 96.2%. When we double the number of neurons in the hidden layer, the test accuracy increases about 1% to 98.09%. The test results can be seen on table 4. Figure 8 and figure 9 demonstrates the graphs obtained for the cases on the table.

Table 4: Experiments with topology

Topology	**Activation**	**Learning Rate**	**L2 Penalty**	**Accuracy**(%)
784-25-10	Tanh	0.1	0.0001	96.2
784-100-10	Tanh	0.1	0.0001	98.09

The number of hidden layers is also considered. In the second part of our experiments, we add one hidden layer to the network. Although the total number of neurons in the hidden layers are equal to the total number of neurons in the default network, we see a decrease of about 1% in test accuracy to 96.14%. Figure 10 demonstrates the results for two hidden layer architecture. As demonstrated on figure 8, figure 9 and figure 10, the convergence rate was not affected with respect to number

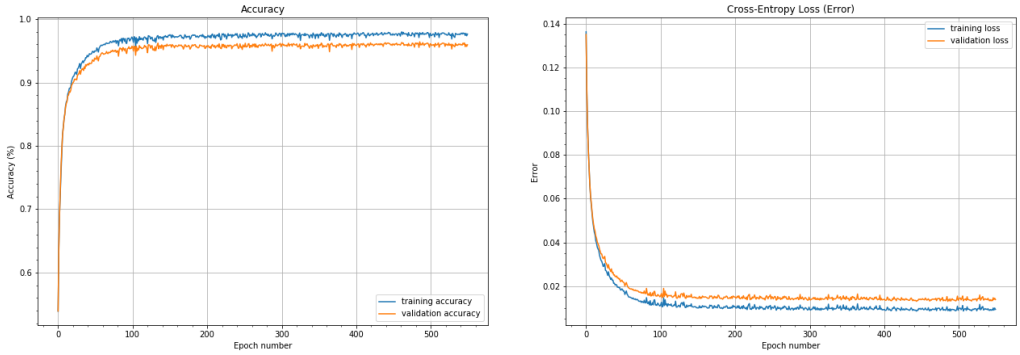

Figure 8: The number of neurons in the hidden layer halved. (Left) The training and validation accuracy (Right) The training and validation loss

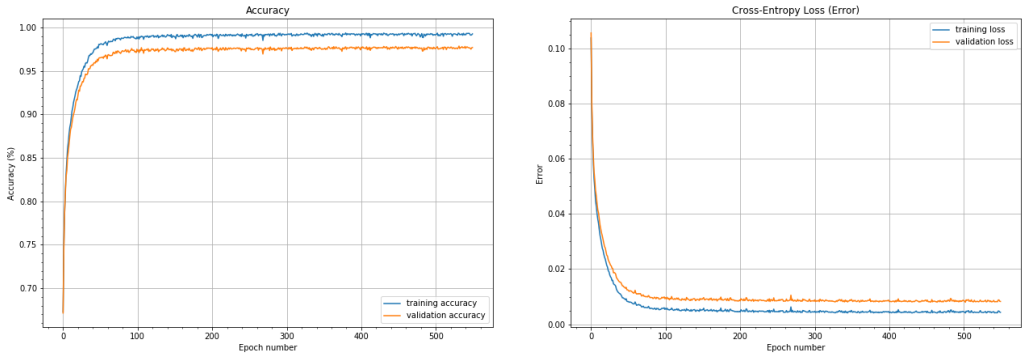

Figure 9: The number of neurons in the hidden layer doubled. (Left) The training and validation accuracy (Right) The training and validation loss

of epochs. However, it takes longer for an epoch to be completed as the number of neurons in the hidden layers increase and as new hidden layers are added to the architecture.

Table 5: Experiments with topology

Topology	**Activation**	**Learning Rate**	**L2 Penalty**	**Accuracy**(%)
784-10-10-10	Tanh	0.1	0.0001	92.76
784-25-25-10	Tanh	0.1	0.0001	96.14
784-50-50-10	Tanh	0.1	0.0001	97.67

7 Individual Contributions to the Project

For this assignment, Gokce Sarar worked on Activations Class, Layers Class, backward pass and loss function of the Neuralnetwork class, the test function and the use of momentum, while Osman Cihan Kilinc focused on the forward pass of the Neuralnetwork class, the checker methods(b), the trainer function, early stopping and the weight regularization. Both of them helped each other with coding and debugging. The workload for the report was distributed evenly. Osman Cihan Kilinc

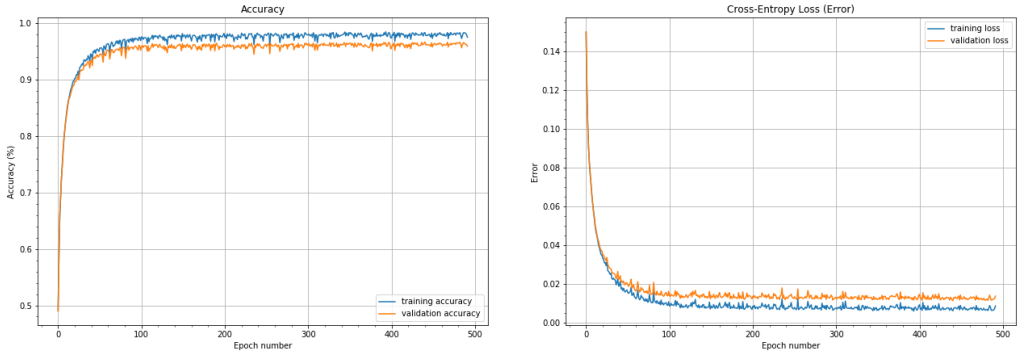

Figure 10: Training results of two hidden layer architecture. (Left) The training and validation accuracy (Right) The training and validation loss

reported questions **b**, **d** and **f**, while Gokce Sarar reported **a** and **c**. Lastly, both of them proofread the report.