KMNIST Deep Learning Classification

Yifei Wang
Department of Data Science
University of California San Diego
yiw014@ucsd.edu

Rui Zhang
Department of Data Science
University of California San Diego
r2zhang@ucsd.edu

Abstract

In this assignment, we will be classifying handwritten Japanese characters using neural networks(Or can be known as multilayer perceptrons). Not only will we build a neural network from scratch, but we will also implement a variety of functionalities which helps improve the accuracy of our model. Functionalities such as regularization, normalization, and momentum are implemented in our network. In addition to that, our network is supported with different kinds of activation functions, which allows the model to learn any pattern that is non-linear. The readers will be able to see that the multi-layer perceptron algorithm we implemented can be really good at classification tasks on high dimensional non-linearly separable data compared with its linear counterpart we saw in assignment 1. Finally, various experiments regarding this neural network will be presented and analyzed, providing some intuition to the readers on how to best finetune the model's parameters.

1 Backpropagation Gradient Check

Before we go into the actual training processes of a complete neural network, we first test the back propagation step in our model by checking the gradients of weights. We compute the slope with respect to one weight using the numerical approximation:

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

where ε is a small constant, for our testing purpose, we specifically choose10^{-2}as our ε; and$E^n$is the cross-entropy error for one pattern.

By comparing the gradient computed using numerical approximation with the one computed as in backpropagation, we should get a difference within 10^{-4}. We use tanh as our activation function and cross entropy as the loss function. For each pattern tested, we would choose one output bias weight, one hidden bias weight, and two hidden to output weights and two input to hidden weights. We choose the first pattern, the sixth pattern and the last pattern in our training set for testing purposes. And we show the difference between computed weight gradients and actual weight gradients in the below tables:

Table 1: Gradient for the 1st pattern

Weight	Approximated gradient	Actual gradient	Difference
output bias	0.000353	0.000357	4.041e-06
hidden bias	0.000135	0.000144	9.027e-06
input to hidden	3.165e-05	3.176e-05	1.077e-07
input to hidden	0.0001069	0.0001064	4.792e-07
hidden to output	0.000263	0.000255	7.571e-05
hidden to output	0.000538	0.000533	5.233e-05

Table 2: Gradient for the 6th pattern

Weight	Approximated gradient	Actual gradient	Difference
output bias	0.00199	0.00208	9.404e-05
hidden bias	0.000543	0.000584	4.129e-05
input to hidden	3.323e-05	3.326e-05	2.592e-08
input to hidden	0.000796	0.000792	3.454e-06
hidden to output	0.00440	0.00434	5.235e-07
hidden to output	0.000329	0.000325	4.173e-05

Table 3: Gradient for the last pattern

Weight	Approximated gradient	Actual gradient	Difference
output bias	0.00110	0.00113	9.404e-05
hidden bias	0.000734	0.000715	1.902e-05
input to hidden	8.364e-06	8.365e-06	7.025e-10
input to hidden	2.575e-05	2.5829e-05	7.4688e-08
hidden to output	0.000201	0.000202	1.189e-06
hidden to output	0.000509	0.000505	4.233e-06

For the output bias weight gradient, we calculate the first unit in the output layer. For the hidden bias weight gradient, we choose the first unit in the hidden layer. For the input to hidden weight gradient, we choose the [0,2], [4,5] units in the input to hidden layer. For the hidden to output weight gradient, we choose the [0,2], [4,5] units in the hidden to output layer.

As can be seen from the above three tables, all the differences between the approximated weight gradients and the actual weight gradients computed from back propagation are within acceptable error range, and therefore our implementation for the back propagation process is correct.

2 Mini-Batch Stochastic Gradient Descent with Momentum

2.1 Description of Training Process

We use a mini-batch stochastic gradient descent with momentum approach to this deep learning neural network to learn a classifier that maps each input data to one of ten label classes. We set the mini-batch size to be 129, and we use a 1-fold cross validation method for convenience when implementing. The momentum term in the update step is set to 0.9. We also use cross-validation

for early stopping of the training: the model would interrupt the training process when the error on the validation set goes up for some "patience" number of epochs. We set the patience number to be 5. And when the process is interrupted, we stop training and save the weights for the minimum validation loss.

We have one input layer, one hidden layer which has 128 units, and one output layer. We use tanh as the activation function, softmax as the output function and cross entropy as the loss function. We first shuffle the training dataset and then use a 80-20 train-validation split of data into the training set and validation set. And we z-score normalize our examples. We apply one-hot-encoding on the labels, so each label has been transformed to a 10-dimensional vector. This neural network would first do a forward pass on the data, and then it would back propagate all the deltas and update all the gradient terms. We iterate through the dataset for many epochs until the validation loss goes up for 5 epochs.

2.2 Discussion of Result

(a) Without Early Stopping

To accomplish the classification task of differentiating between 10 different classes, we build a model by using 0.0005 as our learning rate, setting the size of each minibatch to be 128, and momentum term 0.9 and we use the tanh activation function. We use 100 epochs. We then plot a training/validation loss graph in Figure 1 and a training/validation accuracy graph in Figure 2.

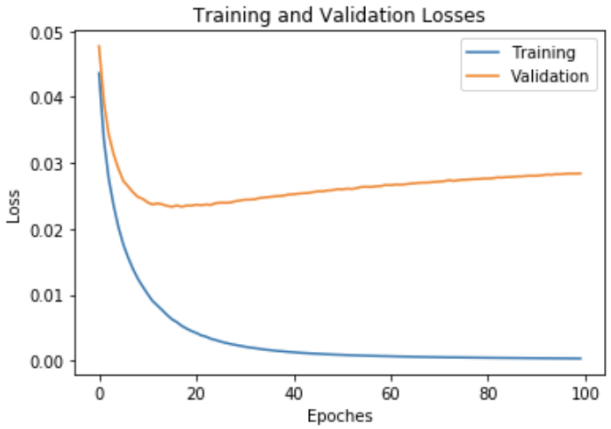

Figure 1: Training/Validation Loss for a single fold

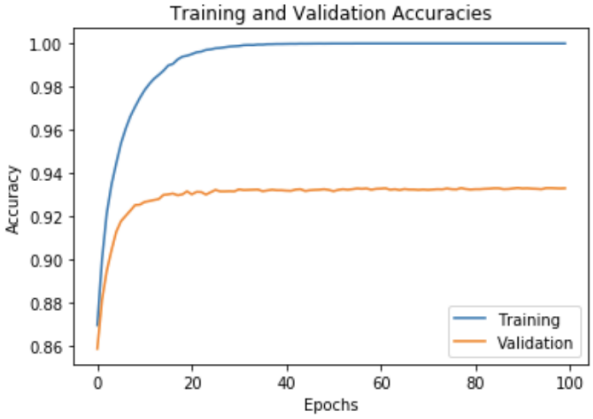

Figure 2: Training/Validation Accuracy for a single fold

We find that learning rate choice is critical, by changing the learning rate from 0.005 to 0.0005, the training accuracy and the validation accuracy improved. When the learning rate is 0.5, it is too

large, and it is hard to converge and achieve satisfying accuracy.

(b) With Early Stopping

To avoid overfitting, we also experiment with early stopping. And the early stopping epoch for our chosen hyperparameter set is about 25 epochs.

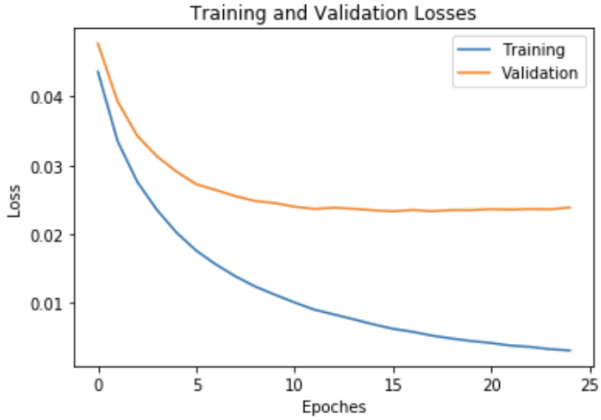

Figure 3: Training/Validation Loss for a single fold with Early Stopping

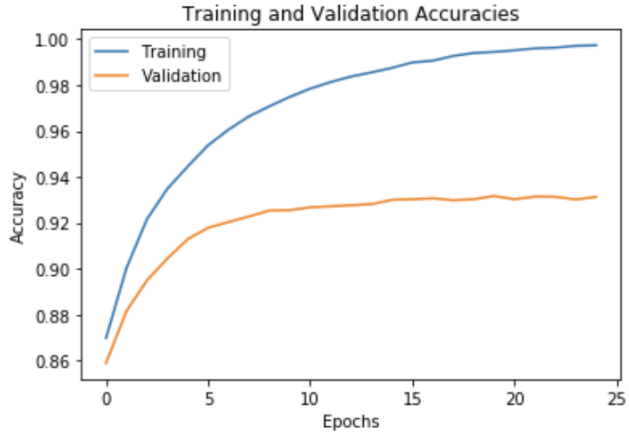

Figure 4: Training/Validation Accuracy for a single fold

By using 0.0005 as our learning rate, setting the size of each minibatch to be 128, momentum term 0.9, and tanh to be the activation function, through the early stopping method, we achieve a best test loss of 0.064 and test accuracy of 84.47%.

3 Regularization Experimentation

3.1 L2 Regularization

(a) $\lambda = 0.2$

We then consider how the regularization of the neural network changes the performance by adding L2 regularization with $\lambda = 0.2$and keeping other hyperparameters the same as in the best tanh

model. We also provide the plot of training/validation losses and accuracy curves.

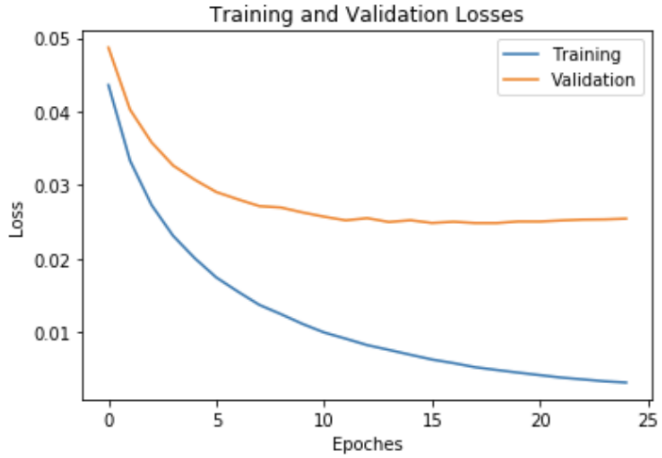

Figure 5: Training/Validation Loss for a single fold

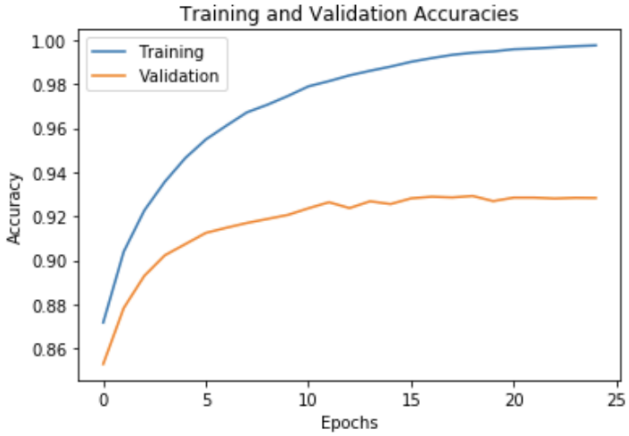

Figure 6: Training/Validation Accuracy for a single fold

We achieved a best test loss of 0.0466 and test accuracy of 86.26%.

If we just use the default setting that the starter code provided to us, the test accuracy will just be 84.01% . The reason for this is that any kind of regularization penalizes our model for overfitting. That is, instead of just letting the model learn noise that exists in the data input, it will make our model generalize better by adding some extra cost to its objective function.

(b) $\lambda = 10^{-3}$

To further explore how the strength of L2 regularization could influence the neural network changes the performance, we add L2 regularization with $\lambda = 10^{-3}$ and keeping other hyperparameters the same as in the best tanh model. We also provide the plot of training/validation losses and accuracy curves.

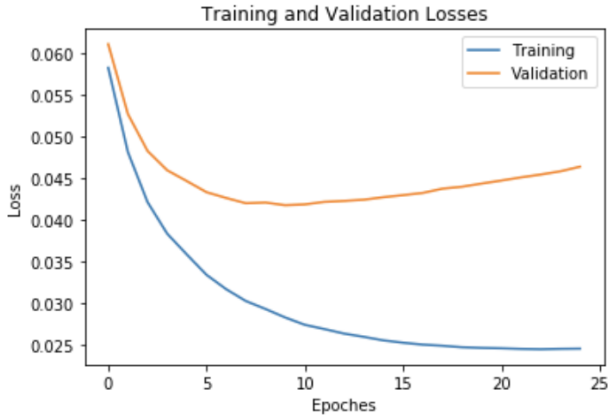

Figure 7: Training/Validation Loss for a single fold with Early Stopping

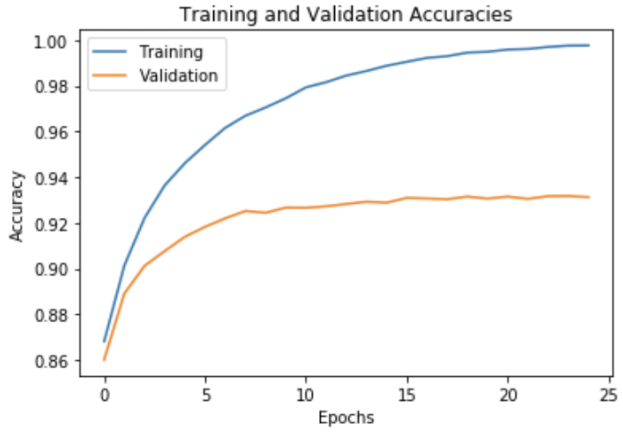

Figure 8: Training/Validation Accuracy for a single fold

We achieved a best test loss of 0.0808 and test accuracy of 84.91%.

As one can clearly see, the test accuracy of this model is less compared to the model that is with optimal L2 regularization. To account for this, we believe the network is under-regularized. In other words, the gamma is too small to make a difference. We also experiment with some large gamma(like setting gamma to be 0.5 or so). It is not surprising that the test accuracy is even worse. The reason could be quite intuitive; high gamma may disrupt the learning process, making the optimizer not be able to reach the local minimum.

3.2 L1 Regularization

(a) $\lambda = 20^{-3}$

We then consider how the regularization of the neural network changes the performance by adding L2 regularization with $\lambda = 10^{-6}$and keeping other hyperparameters the same as in the best tanh model. We also provide the plot of training/validation losses and accuracy curves.

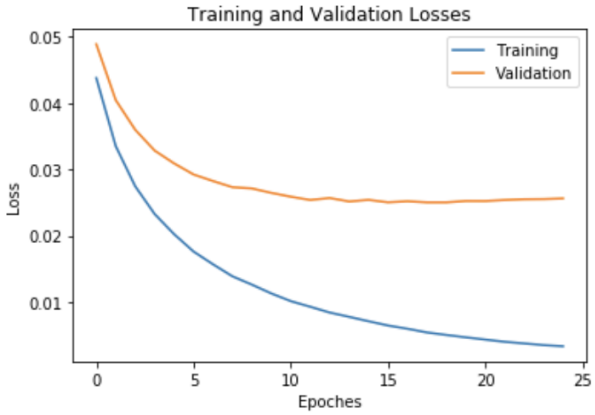

Figure 9: Training/Validation Loss for a single fold

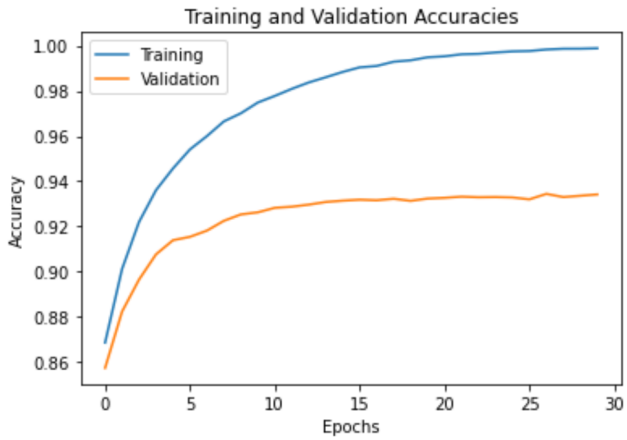

Figure 10: Training/Validation Accuracy for a single fold

We achieved a best test loss of 0.0586 and test accuracy of 84.24%.

You might notice that L1 regularization is not as good as L2 regularization in this specific task. The reason might be that L1 takes a more radical approach driving our weights to zero whereas L2 only penalizes larger weights. So, in practice, we might just use L2 regularization instead since it is recommended in class. .

(b) $\lambda = 10^{-3}$

To further explore how the strength of L1 regularization could influence the neural network changes the performance by adding L1 regularization with $\lambda = 10^{-3}$ and keeping other hyperparameters the same as in the best tanh model. We also provide the plot of training/validation losses and accuracy curves.

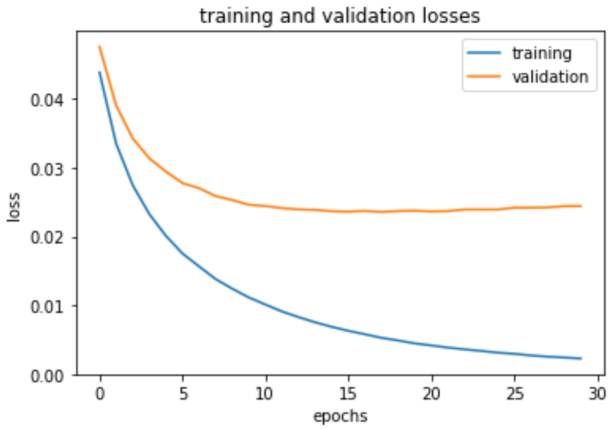

Figure 11: Training/Validation Loss for a single fold with Early Stopping

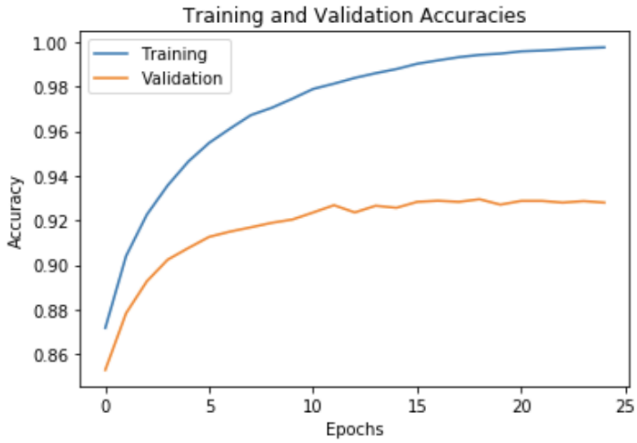

Figure 12: Training/Validation Accuracy for a single fold

We achieved a best test loss of 0.0651 and test accuracy of 84.21%.

The test result does not vary significantly as we change gamma a little bit. Notice that in this experiment with L1 we can not set the gamma to the level as it used to be when we experiment with L2 normalization. For example, In L2 experiment we conclude that setting gamma to be 0.2 yields the best test accuracy. But, if one tries to set the same value here, the weight will not be converged to a minimum.

Due to the overall poor performance of L1 experiments, we set L2 regularization as our model's hyperparameter instead.

4 Activation Experimentation

4.1 Sigmoid Activation

We then experiment with different activation functions to explore whether the model performance could be improved and compare the change in model performance. We first implemented the Sigmoid activation by only changing the activation function and keeping other hyperparameters the same as in the best tanh model. We also plot a training/validation loss graph and a training/validation accuracy graph in the figures below.

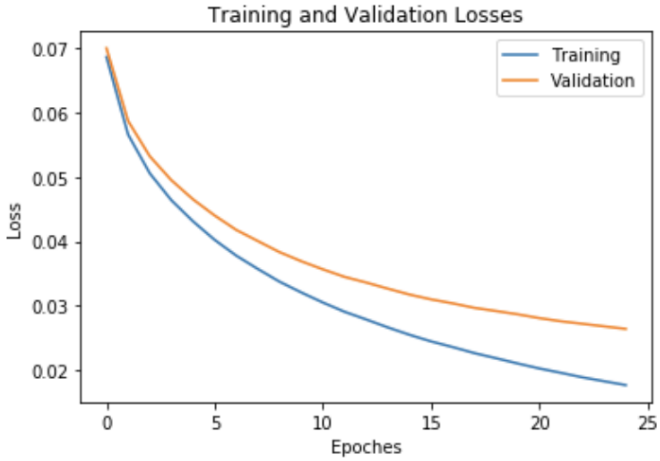

Figure 13: Training/Validation Loss for a single fold with Early Stopping

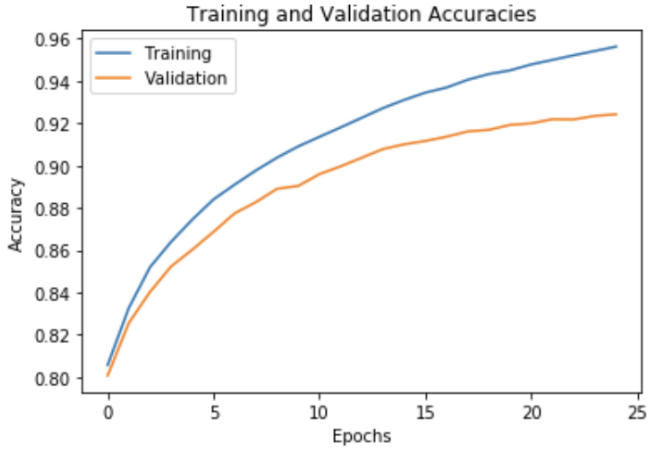

Figure 14: Training/Validation Accuracy for a single fold

By using 0.0005 as our learning rate, setting the size of each minibatch to be 128, momentum term 0.9, and Sigmoid to be the activation function, through the early stopping method, we achieve a best test loss of 0.0573 and test accuracy of 83.05%.

Sigmoid is not a recommended activation method for hidden layers. What sigmoid does is that it maps an input to a probability. If the input is too big and too small, the mapping would not be as effective as the case where the input is within the "linear range".

4.2 ReLU Activation

Apart from Sigmoid and Tanh activations, we also include discussion of the ReLU activation neural network model performance by only changing the activation function and keeping other hyperparameters the same as in the best tanh model. We plot a training/validation loss graph and a training/validation accuracy graph in the figures below.

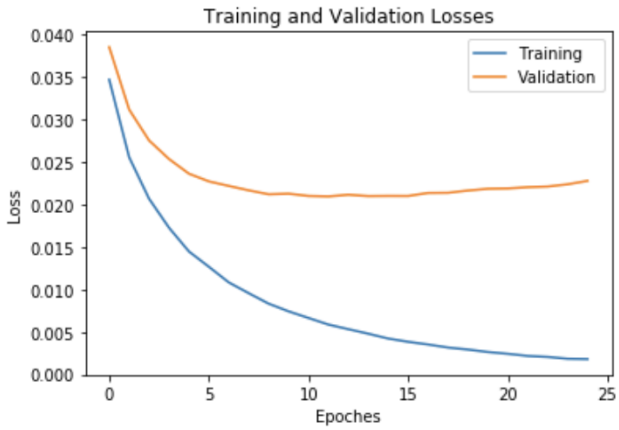

Figure 15: Training/Validation Loss for a single fold with Early Stopping

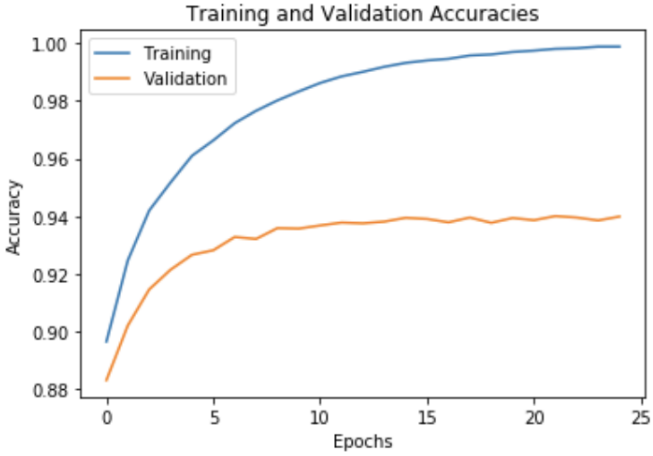

Figure 16: Training/Validation Accuracy for a single fold

By using 0.0005 as our learning rate, setting the size of each minibatch to be 128, momentum term 0.9, and ReLu to be the activation function, through the early stopping method, we achieve a best test loss of 0.0618 and test accuracy of 86.42%.

4.3 Our final choice

Since Relu is performing much better than sigmoid and tanh did (remember in the previous section the activation was set to be tannh, and the average performance is about 84% when we did not fine tune the learning rate) , we will use Relu as one of our model's hyperparameter.

5 Network Topology Experimentation

5.1 Hidden units

(a) Double the number of Hidden units to 256

We then consider how the topology of the neural network changes the performance by only doubling the number of hidden units to be 256 and keeping other hyperparameters the same as in the best tanh model. We also provide the plot of training/validation losses and accuracy curves.

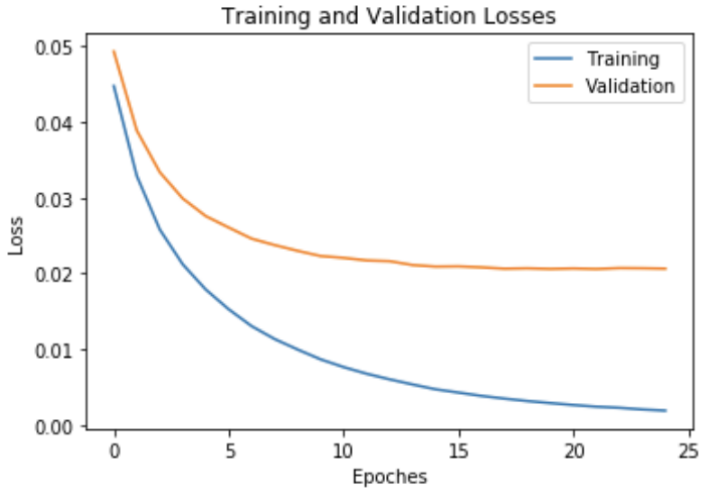

Figure 17: Training/Validation Loss for a single fold

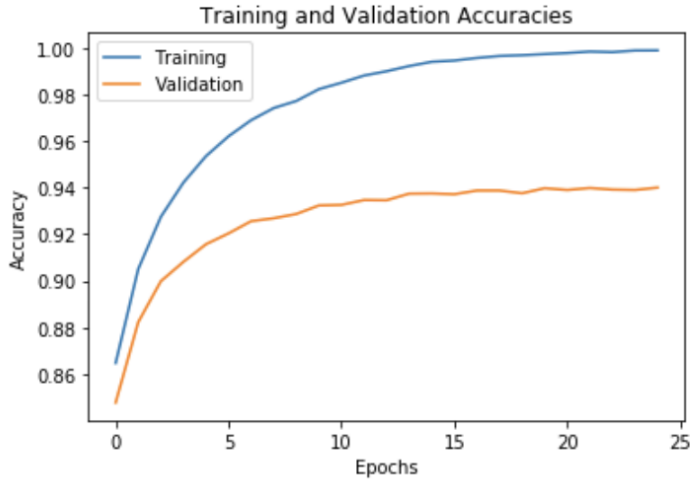

Figure 18: Training/Validation Accuracy for a single fold

We find that by doubling the number of units in the hidden layer, we achieve a test loss of 0.0557 and test accuracy of 85.89%.

Increasing the number of hidden units did increase the test accuracy. The reason might be that our previous setting of 128 hidden units made our network come with higher bias. Indeed, the fewer hidden units our model has , the fewer features our model can learn. Increasing the number of hidden units causes our model to learn more useful features of our input pattern, and due to better feature space, the model performance naturally increases.

(b) Halve the number of Hidden units to 64

To explore the effect of the number of hidden units on model performance, we halve the hidden

units in the best tanh model to be only 64 and keep other hyperparameters the same. We provide the plot of training/validation losses and accuracy curves below.

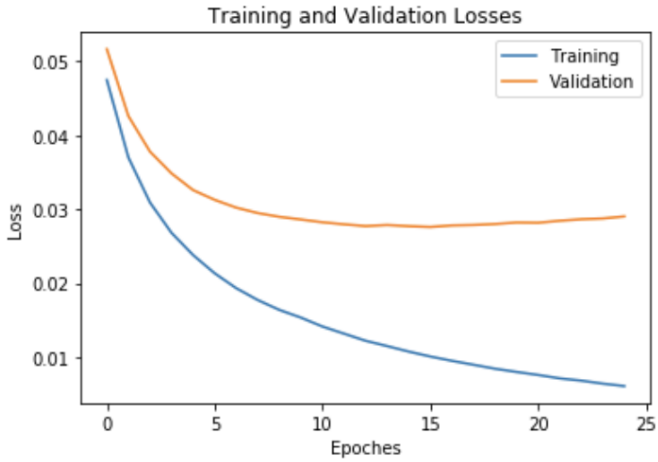

Figure 19: Training/Validation Loss for a single fold with Early Stopping

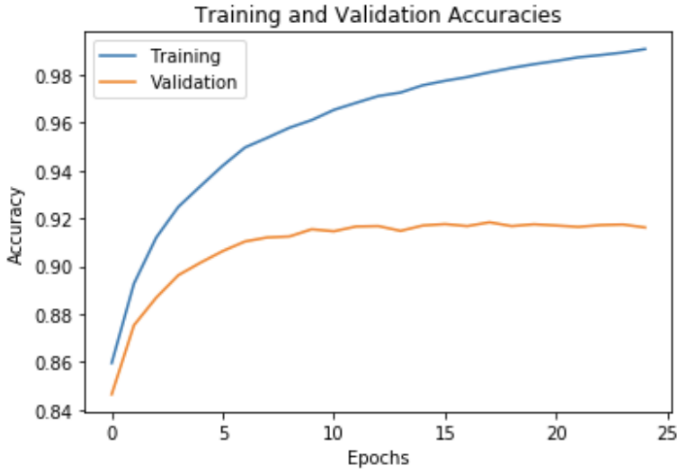

Figure 20: Training/Validation Accuracy for a single fold

We find that by halving the number of units in the hidden layer, we achieve a test loss of 0.0693 and test accuracy of 82.74%.

Hidden layer in this case may not even capture some essential information sent from the input layer. The internal representation learned may not capture all necessary detail for a good classification task. Hence we will use 256 hidden units as our model's final hyper parameter.

5.2 Double the Hidden Layers

To explore the effect of the number of hidden layers on model performance, we double the hidden layers to be 2 and have 64 hidden units on each hidden layer to have the same number of hidden units as in the best tanh model, and keep other hyperparameters the same. We provide the plot of training/validation losses and accuracy curves below.

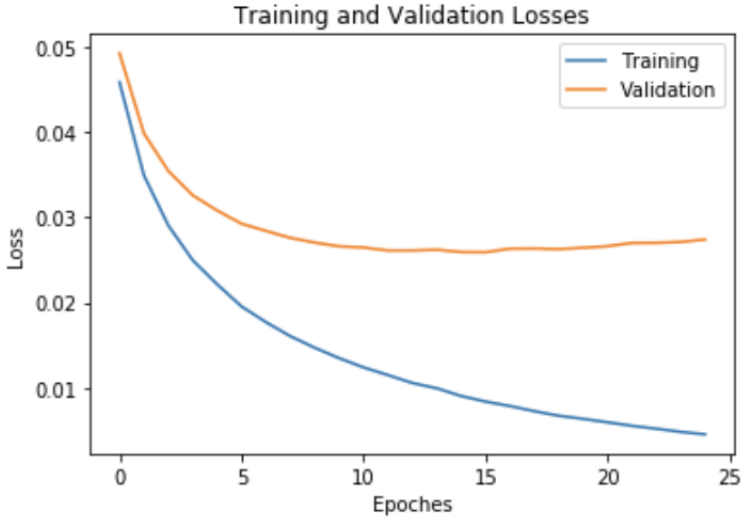

Figure 21: Training/Validation Loss for a single fold with Early Stopping

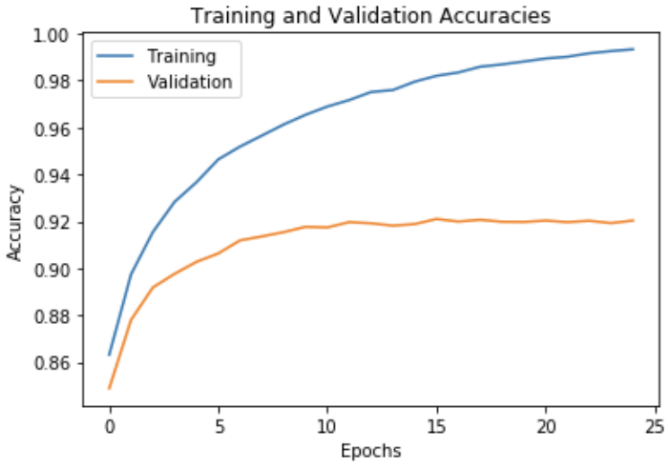

Figure 22: Training/Validation Accuracy for a single fold

We find that by doubling the number of layers in the hidden layer, we achieve a test loss of 0.0698 and test accuracy of 83.07%.

Again it shows that 64 hiddens units is not a good choice even when we double the numbers of hidden layers. If time and resources permit, we can increase the number of hidden units and hidden layers to see if there is any performance boost.

6 Final model's performance

For our final model, below are our final hyperparameters:

```
config["layer_specs"]=[784, 256,256,10]
config["batch_size"]=129
config["early_stop"]=True
config["early_stop_epoch"]=30
config["activation"]="ReLU"
config["learning_rate"]=0.0005
config["epochs"]=100
config["L2_penalty"]=0.2
config["momentum"]=True
config["momentum_gamma"]=0.9
```

With these parameters, the model reached an accuracy of 89%, which is much higher than it was using the default model hyperparameters. We believe that had more time, resources and parameter tuning tricks been invested, the final accuracy would have been even better, surpassing 90% of test accuracy.

7 Team contributions

Coding part: We wrote and tested the neural network together. Yifei Wang took care of experiments on the gradient checking and regularization part, and Rui Zhang focused on topology/activation experiments. The report is split to 50:50 where each of us wrote half of the report.