Udaikaran Singh and Wesley Kwan
Gary Cottrell
CSE 190
10/23/2018

PA2: Mutli-Layer Neural Network on MNIST dataset

Udaikaran Singh
Department of Computer Science
University of California, San Diego.
La Jolla, CA 92093.

Wesley Kwan
Department of Computer Science
University of California, San Diego
La Jolla, CA 92093

Abstract

In this assignment, we created a multi-layer neural network to classify digits from the MNIST dataset. After implementing a basic network with forward propagation and backward propagation and trained it with stochastic gradient descent, we made implemented modifications, such as momentum, regularization, different activation functions, and different network topology to see how it affected our network.

In our results, we found that the regularization penalty very harshly penalizes how well the network fits to the training data. For example, when the regularization penalty was just 0.001, the accuracy on the training set could baselined around 82%, while without regularization, it could get 90% accuracy. On the other side, we found that regularization keeps the model from overfitting the training data pretty well.

When testing the activation functions, we found that the relU works marginally better than other activation functions, and it trains on the network faster.

Lastly, when testing network topology, we found that a network with more hidden layers, or a network with multiple hidden layers, tend to quickly overfit the training data, and does not generalize as well as a network with less hidden units.

(a) Reading in MNIST data

Since reading in the data does not require further explanation, this part will be dedicated on explaining the hyperparameters used in our network.

The hyperparameters we used for the neural network was training on all 50,000 training samples. The training type is stochastic gradient descent. The number of epochs for Part C is 100, but the point of overfitting was found to be 26 epochs. The learning rate used was 0.001

Momentum was implemented based on method defined by Geoffrey Hinton, and described on the lecture slides. The main idea of it to set the multiplier on each weight at 1. When calculating the gradient, the gradient for each weight is compared to the previous gradient. If the sign of the gradient is the same, we add 0.05 to the multiplier. If the gradient is opposing the previous gradient, then we multiply the multiplier by 0.95. Also, the gamma value we used for momentum was 0.9.

Regularization was implemented by the function L(w) = $\frac{\lambda}{2} * \sum_{L} \sum_{i} \sum_{j} (w_{ij}^{L})^2$.

(b) Gradient Checker

The gradient was checked by comparing approximations of the gradient and the actual gradient our network computed to ensure the gradient is correct. It was implemented by taking the weights or bias and changing them by some epsilon, finding the loss going through forward propagation, and using the loss for approximating the gradient. That gradient would be compared to the gradient that the network would have computed in back propagation. Some of the approximations we found we 0, which makes sense since a small change in one weight out of the entire network would've changed the loss by a miniscule amount, leading E(w+e) – E(w-e) to be 0.

We used epsilon = 0.1.

	Gradient Approximation E(w+e) – E(w-e) / 2e	Actual Gradient
Input to hidden weight 1	0.0	3.836095938522184e-17
Input to hidden weight 2	7.227379809904289e-07	-1.5189384418679225e-06
Hidden bias weight	0.0	-3.836095938522184e-17
Hidden to output weight 1	6.245004513516506e-16	-1.6111442405766055e-16
Hidden to output weight 2	-5.551115123125783e-16	5.46151463875278e-16
Output bias weight	3.366672782868785e-05	-6.40231002820909e-05

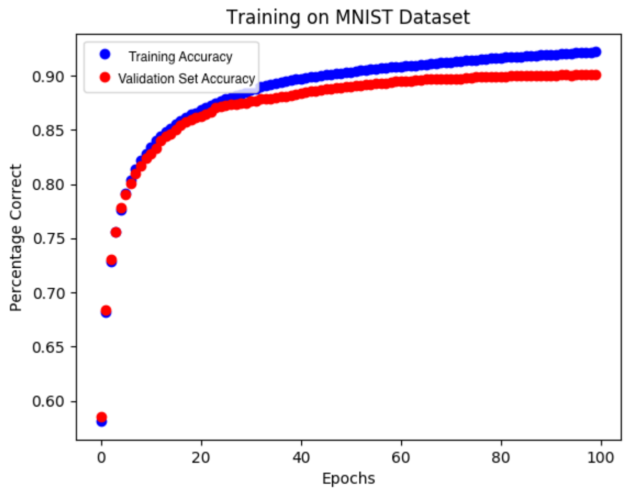
All the gradients are approximately equal (within epsilon squared).

(c) Finding Optimal Number of Epochs

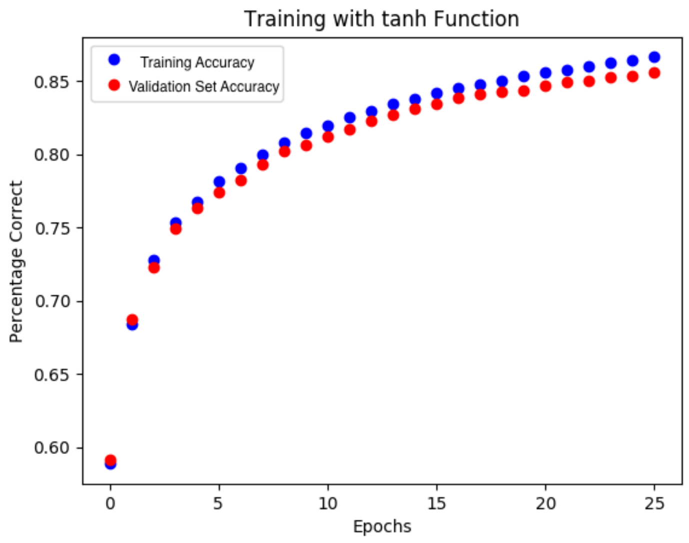
The training procedure we used to determine the necessary amount of epochs to use the validation set as the holdout set and claim that when the error on the validation set begins to rise for multiple epochs (for use, this was 5), then we can conclude that the network is beginning to overfit on the training data.

We tested over 100 epochs. When the cross-entropy error was larger than the previous cross entropy error, we increment a counter. The point at which we have 5 epochs of which the error on the validation set is going up, we conclude that the model is beginning to overfit.

We found that the optimal amount of epochs before the model begins to overfit is 26 epochs. For the rest of the experiments, we used 26 as the parameter for number of epochs, except for testing regularization, in which we 10% more epochs.



Accuracy: 87.41% correct on the test set (after 100 epochs)

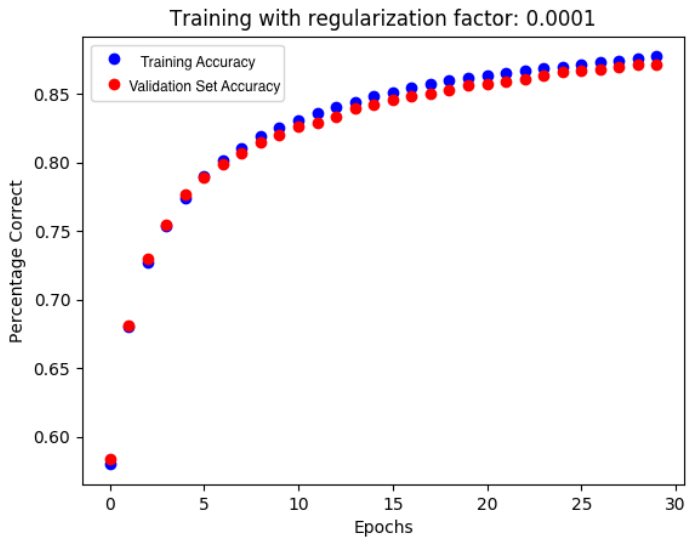


Accuracy: 86.05% correct on test set (after 26 epochs)

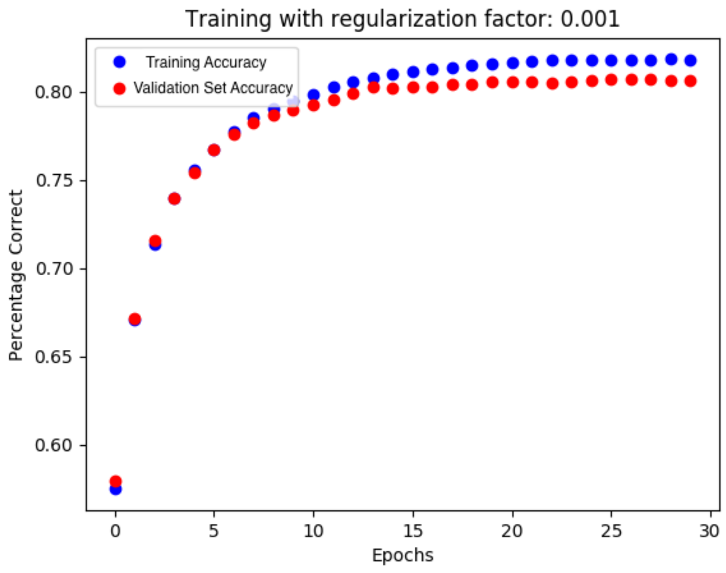
(d) Experimenting with Regularization

We found that with an increased regularization penalty, there is lowered accuracy on all sets. Also, we saw that with a higher regularization penalty, the accuracy on the training set and validation set began to level out quicker.

It seems that when there is no regularization penalty, the model will continue continuously learn on training set. However, with the regularization penalty, the accuracy on the training set begins to level out rather than continuously growing, because the regularization term in the objective function penalizes model complexity.



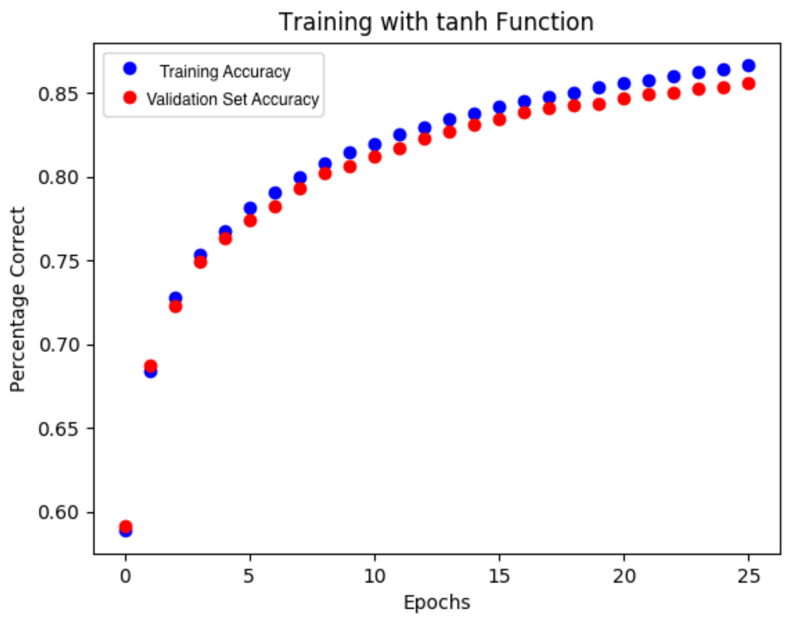
Accuracy on test set with 0.0001 regularization: 81.32%



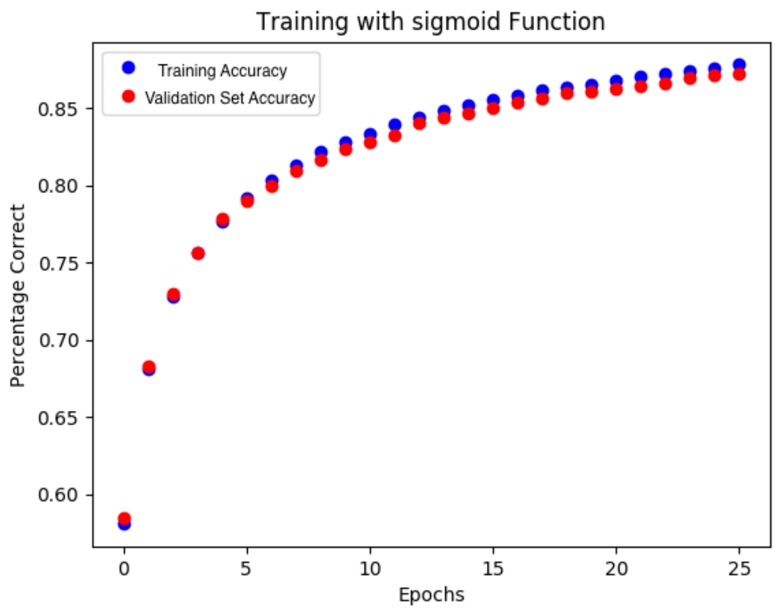
Accuracy on test set with 0.001 regularization: 77%

(e) Experiment with Activations

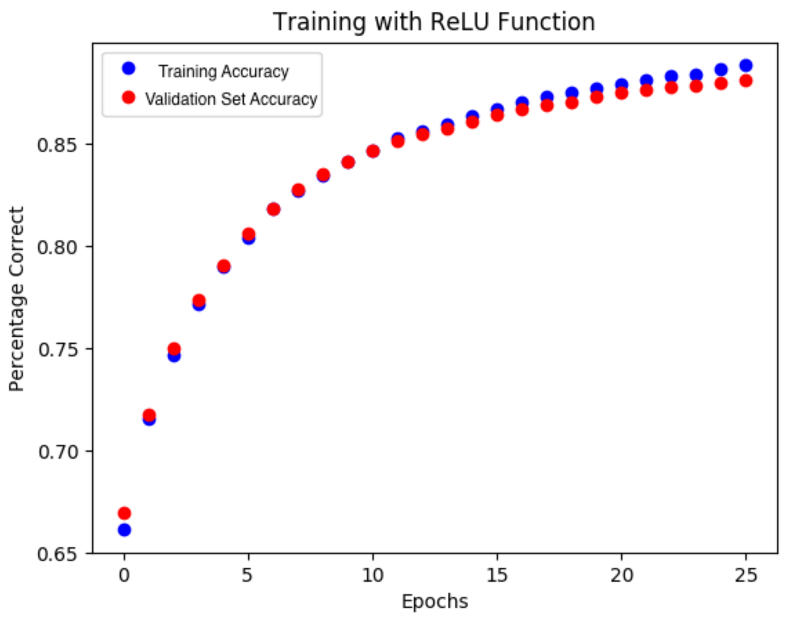
We found that each of the activation functions produces similar results in terms of training and their accuracy on the training set. Generally, the sigmoid and ReLU performed better than tanh in training. The ReLU activation function trained faster than the activation functions. This likely happened because the gradient is faster for the computer to compute.



Accuracy with tanh: 86.05%



Accuracy with sigmoid: 87.41%



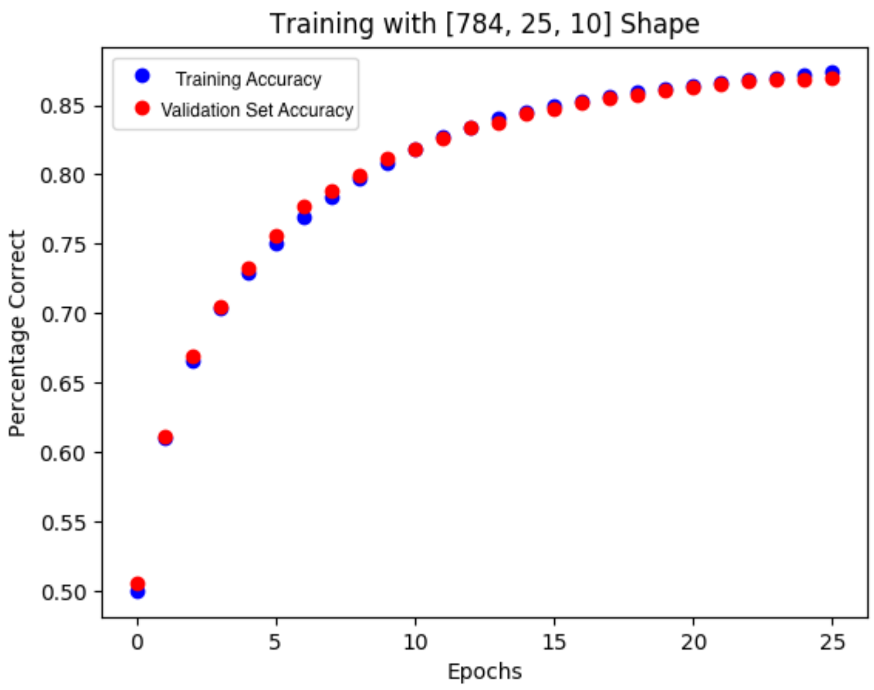
Accuracy with ReLU: 87.62%

(f) Experiment with Network Topology

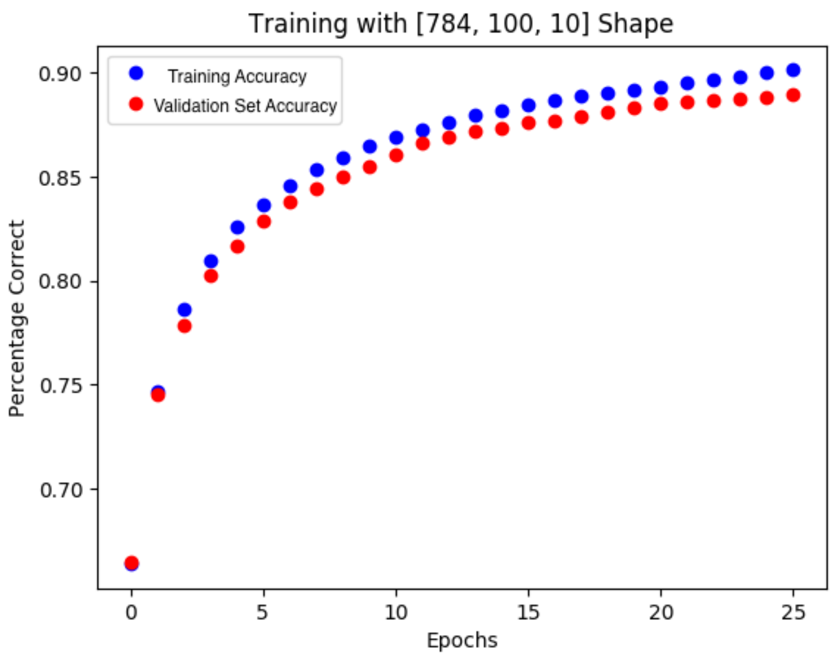
Using our model with 50 hidden units and tanh as the activation function, we got an accuracy of 86.05% on the test set. It seems that the model with 25 hidden units performs just as well 50 hidden units. However, when using 100 hidden units, or 2 layers of 47 hidden units, the model performed worse on the test set.

Based on the charts below, the more complicated models (one with 100 hidden units and 2 hidden layers) trained better than the 25 hidden unit model. However, the more complicated models also have the accuracy on the training set keep growing, while the accuracy on the validation set levels out. This indicates that the more complicated models are beginning to overfit to the training set.

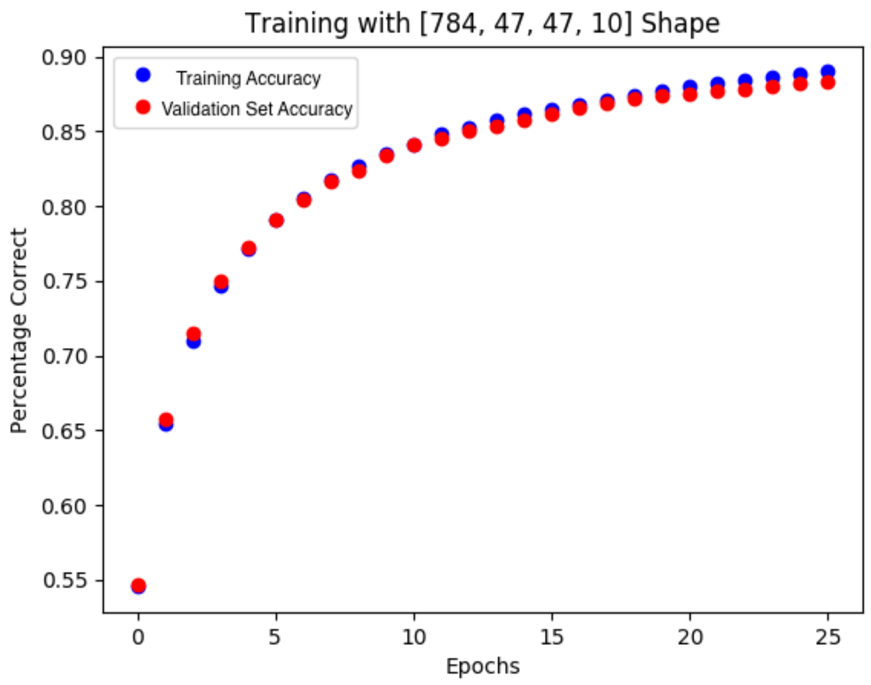
Likely, the more complicated models are identifying internal representations within the characters that are beneficial for improving accuracy on the training set, but do not generalize well. This would also explain why the neural network with 25 and 50 hidden units performs better, because it needs to learn fewer internal representations of the digits with the hidden unit to effectively classify them.



Accuracy on 25 hidden units: 86.92%



Accuracy on 100 hidden units: 83.12%



Accuracy on 2 hidden layers of 47 units: 82.4%

Individual Contributions

Both of us worked together the Activation, Layer, and Neuralnetwork class in neuralnet.py. We also both together implementing and testing forward and back propagation. We both wrote the report together.

Wesley:
I mostly worked on loading in the data, testing back propagation, and checking the gradient. I worked mainly on the load_data, back_pass, loss_func, and test functions in neuralnet.py and the gradientchecker.py file.

Udai:
I worked on implementing the trainer method in neuralnet class, implementing regularization to the objective function and back propagation, and implementing the momentum mechanism in the weight and bias update. Also, I set up the experiments and plots derived from those experiments.