Experimentation with Regularization, Activation, and Network architecture in Multi-layer Neural Networks to Improve Performance on Classification of SVHN Dataset

Lucas Nguyen, Paul Pan
Department of Computer Science and Engineering
University of California - San Diego
La Jolla, CA 92093
lnguyen.professional@gmail.com, jpan@ucsd.edu

Abstract

There are many methods to optimize Neural Network performance including standardizing data, tuning hyperparameters, using momentum, early stopping, regularization, different activation functions, and changing both dimensions of network architecture. In this study we implement a Neural Network classifier from scratch to classify the SVHN dataset with all of the mentioned methods, and emphasis on experimental results of the last 3. We plot training and validation losses and accuracies, then evaluate test accuracy, improving from 75% to upwards of 84%.

1 Backpropagation

We implement a Neural Network from scratch using Python and NumPy, however since backpropagation is often tricky to implement, we verify our version by calculating the gradient by hand then comparing that to the gradient returned by backpropagation. Specifically, we increase and decrease a single weight by a small quantity (0.01 here), calculate the difference between resulting losses, use the equation below to calculate gradient, then compare that to the gradient propagated backward to that weight. If our implementation is correct, then the difference between calculated and propagated gradients should be in the range of or better than the quantity squared, O(0.0001). We perform this test on a batch containing the first 5 training examples for several different weights to make sure all weights are learning correctly.

Weight	Calculated Gradient	Backpropagated Gradient	Difference
Output Bias	0.059802	0.05924	0.00055
Hidden Bias	-0.00394	-0.00398	0.00004
Hidden to Output 1	-0.00561	-0.00561	9.21964e-09
Hidden to Output 2	0.00327	0.00327	-4.33565e-08
Input to Hidden 1	-0.00158	-0.00158	1.19733e-08
Input to Hidden 2	-0.00049	-0.00049	-6.87756e-08

Table 1: Calculated versus Backpropagated Gradients

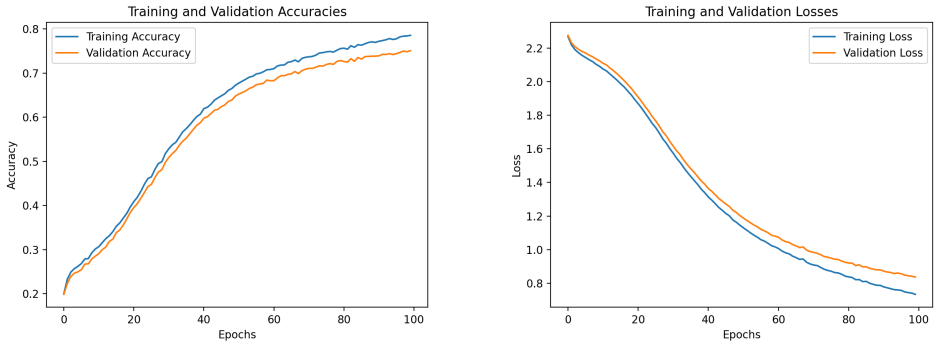
All of our gradient differences remained below the defined limit of the quantity squared, O(0.0001). It is worth noting that the difference remained in the same magnitude for the biases but for hidden units the difference was much smaller. This might have to do with how many bias units there are compared to weights per layer.

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

2 Training

We first standardized the training data using the equation below, then standardized the test data with the mean and standard deviation of the testing data. Next we split the training data into training and validation by 80:20 split. We train the model for 100 epochs, performing mini-batch learning with batches of size 128. The model is trained with tanh activation, momentum of 0.9 [see equation below], softmax output and early stopping, where if validation loss increases for 5 consecutive epochs we stop training. Loss is evaluated with cross-entropy. After each epoch we record training and validation losses and accuracies, and plot both below. We achieve 72% accuracy on the test set. However, we noticed the model was training fairly slow and not using all of its epochs, so we increased learning rate to 0.01. Training with this learning rate, we got an accuracy of 75%.

$$tanh(x) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{1}$$



3 Experimentation with Regularization

3.1 L2

Regularization adds a term to the cost function based on the weights to limit how big weights can get when training. We tried training with L2 regularization, which adds the squared magnitude of the weights to the cost function. In our implementation we used the derivative of the term to add regularization to the update rule. We trained regularization for 110 epochs and with learning rate of 0.01 and regularization penalty of 0.0001. L2 regularization performed 77% which is 2% better than the default configuration, probably because regularization prevents overfitting during training so the model generalizes better.

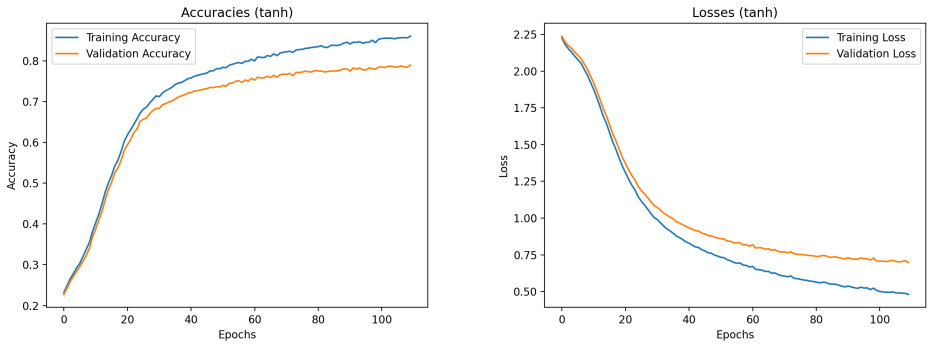


Figure: L2 Regularized Accuracies and Loss

3.2 L1

We also tried L1 instead of L2 regularization, which affects learning in the same way but uses the weights rather than their magnitude to penalize loss. In the update rule we add the derivative of the

term, which is just the sign of the weights. We actually saw about the same performance, as test accuracy 77% (76.756% to be exact) was the same as L2 regularization. Although it is possible this is an issue in implementation, one explanation for this result is that our weights are already fairly small so size of weights doesn't have much impact on the regularization term.

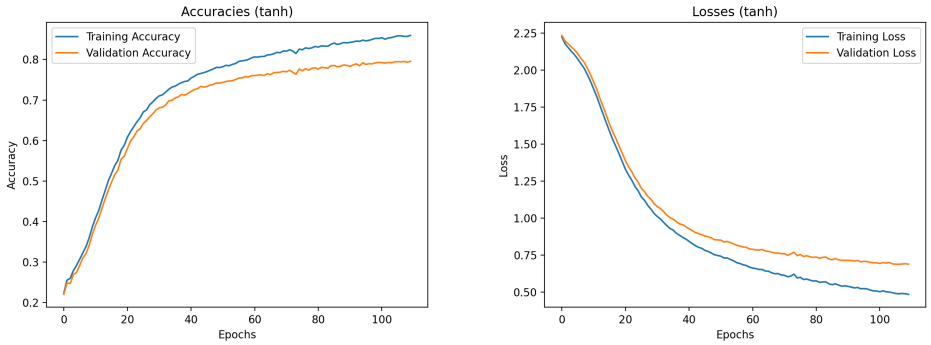


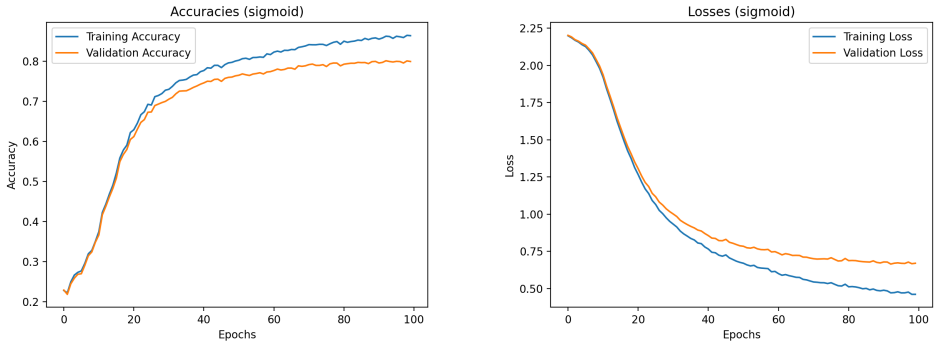
Figure: L1 Regularized Accuracies and Loss

4 Experimentation with Activations

4.1 Sigmoid

We trained the network using the default configuration, but using sigmoid instead of tanh activations. With the default configuration the loss graphs suggested the model was training too slow, as the loss was still decreasing at the same rate as when it started after 100 epochs. The final test accuracy was 33%. We tweaked the model by increasing the learning rate to 0.05. With this setup the Sigmoid performed better than before and almost the same compared to tanh, with test accuracy of 78%.

$$sigmoid(z) = \frac{1}{1 + e^{-z}} \tag{2}$$



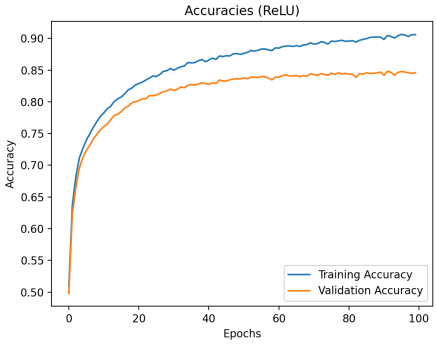
4.2 ReLU

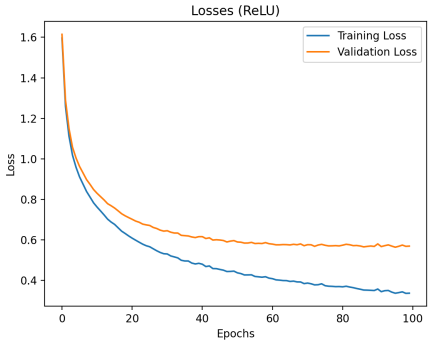
We also trained the network with ReLU activations. We started with a learning rate of 0.05 as we did for Sigmoid activation, however this appeared too high for ReLU as training and validation losses split towards later epochs and validation loss actually increased. Test accuracy evaluated to 82% which was better than the previous activation functions but we thought we could do better. We decreased learning rate to 0.01 and got a test accuracy of 84%.

It is worth nothing that ReLU losses dropped faster than tanh, and performed best out of all of the activations. One potential explanation for this is that Sigmoid and tanh suffer from vanishing gradients; as the prediction gets more accurate i.e. approaches 1 or 0, the derivative of sigmoid approaches 0. Since the update rule uses the slope of Sigmoid to change weights, as predictions get

more accurate, the gradients with respect to the weights would get smaller and predictions would approach targets slower. The derivative for ReLU on the other hand is 1 no matter how big inputs get and thus ReLU doesn't experience this problem. This is why ReLU is used more often in modern neural networks.

$$ReLU(z) = max(0, z) \tag{3}$$

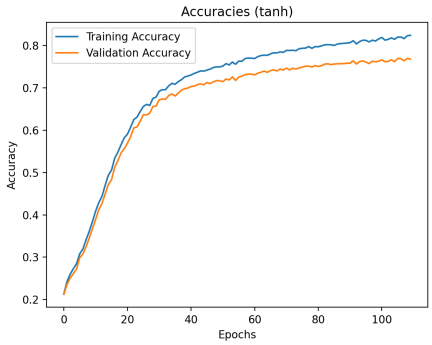


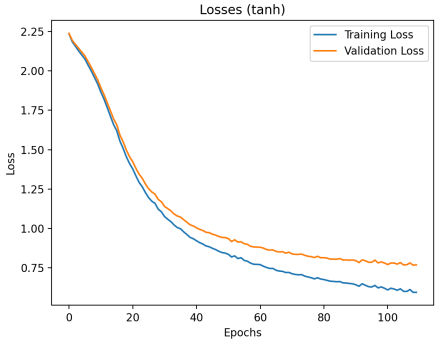


5 Experimentation with Network Topology

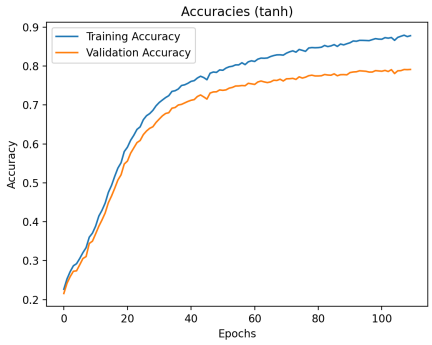
5.1 Changing the Hidden Units

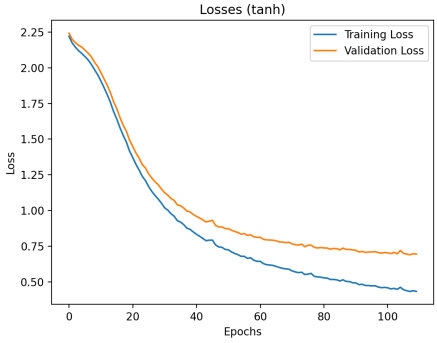
We halved the number of hidden units from 128 to 64 with a learning rate of 0.01 and got test accuracy of 74%, which is less than 1% less than the default. As nodes decrease, accuracy should go down as there are less nodes to model patterns in the image data, however the decrease is so low it suggests not a lot of information is needed to model features in SVHN.





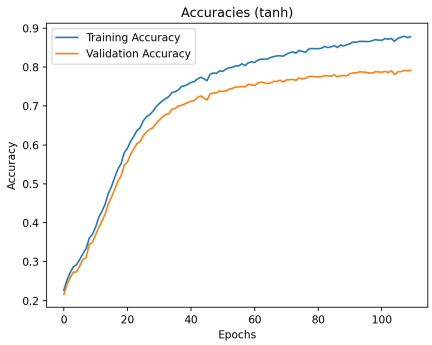
We doubled the number of hidden units from 128 to 256 with a learning rate of 0.01 and got test accuracy of 77%, which is 4% higher than normal. Intuitively this makes sense as more hidden units means more nodes to model patterns passed in.

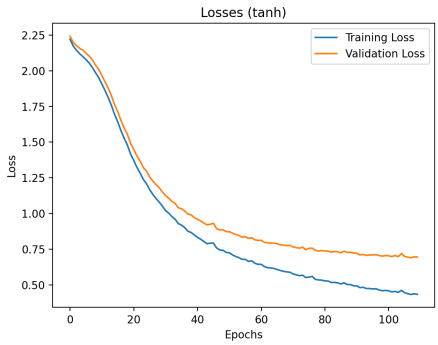




5.2 Changing the Hidden Layers

Finally we doubled the number of layers from the standard configuration, to two 128 unit layers using learning rate of 0.01, achieving test set accuracy of 80% which is 5% higher than baseline. Accuracy is higher compared to baseline and compared to changing number of units because more layers means more complex features can be learned instead of just more features.





6 Contributions

Lucas: I wrote the report, started forward and backpropagation, wrote some of batch learning, plotting, and preprocessed the data.

Paul: I implemented most of the logic for the network layers, and I also did the majority of the backpropagation and forward propagation. Also, I tested the hyperparameters and helped debug any shape errors in NumPy.

We both worked on getting backpropagation to work, implementing forward and backward passes in the network, writing activations and their derivatives, loss and softmax and helped each other debug.