CMP SC 8770 & ECE 8890
Spring 2019
Neural Networks

Project 1 Convolutional Neural Network (CNN)

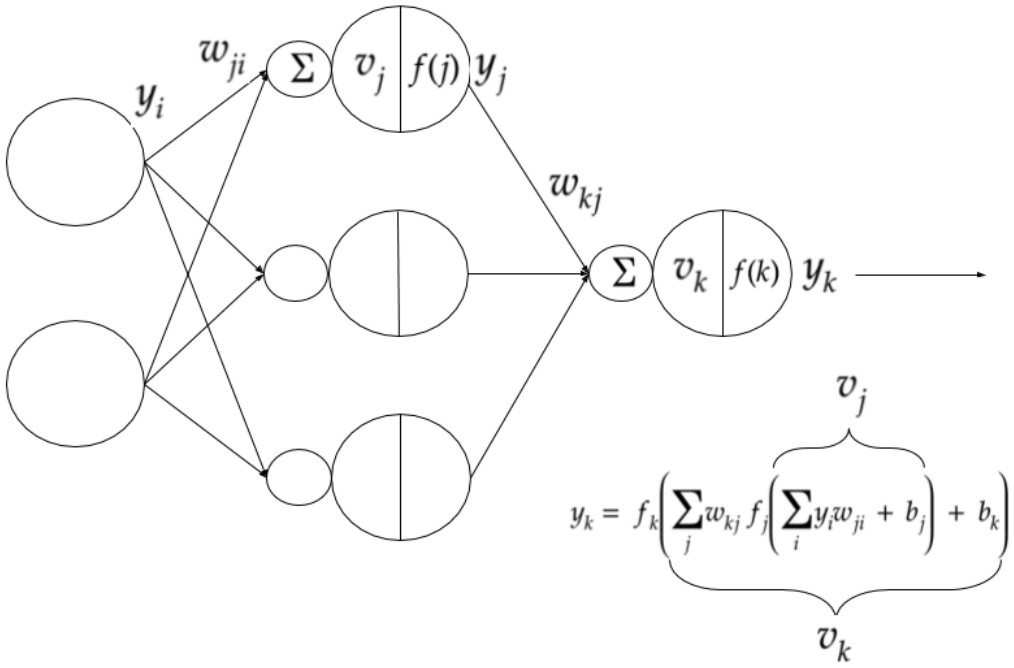
Luke Guerdan

March 19, 2019

I checked with Dr. Anderson, who said that 32 pagis is OK

Technical Description

Neural Networks: Forward and Backward Propagation



While explaining the forward and backward pass of a network, it is helpful to have a visual guide. The above figure shows a simple multi-layer perceptron with intermediary variables used during the forward and backward pass. The equation included in the diagram shows that the output for a single node is:

$$y_k = f_k\left(\sum_j w_{kj} y_j + b_k\right)$$

Where w_{kj} is the weight connecting neuron j to neuron k, $f_k()$ is the nonlinear activation function, and b_k is a bias term. The overall output of the network can be computed by sequentially calculating the output of each node for all neurons in layer i, followed by layer j, then finally the output layer k. Here, y_k is the overall output of the network.

The purpose of training a neural network is finding a set of parameters $\Theta = \{W, b\}$ which minimize the error between the predicted and ground truth labels of a dataset. For training on the MNIST dataset, this could be the sum of squared difference between the target d_k and the network output y_k, where labels are encoded as a one-hot vector. This error function can be defined at epoch t as:

$$E(t) = \frac{1}{2}\sum_{k}(d_k(t) - y_k(t))^2$$

where the error for a single output node is expressed as:

$$e_k(t) = d_k(t) - y_k(t)$$

In order to reduce the error of the network, each weight w needs to be updated in the opposite direction for the error gradient. Without loss of generality, this is shown for the specific weights shown in the above diagram. The chain rule can be applied to find the change in error with respect to a specific weight leading to the output layer such that:

$$\frac{\partial E(t)}{\partial w_{kj}(t)} = \frac{\partial E(t)}{\partial e_k(t)} \times \frac{\partial e_k(t)}{\partial y_k(t)} \times \frac{\partial y_k(t)}{\partial v_k(t)} \times \frac{\partial v_k(t)}{\partial w_{kj}(t)}$$

The error delta can then be defined for an output node as:

$$\delta_k(t) = -\frac{\partial E(t)}{\partial v_k(t)} = e_k(t) \times f'(v_k(t))$$

Since there is no explicit error value defined for internal nodes in the network, an estimated delta can be computed by multiplying the deltas at the next layer by their connecting weights and summing:

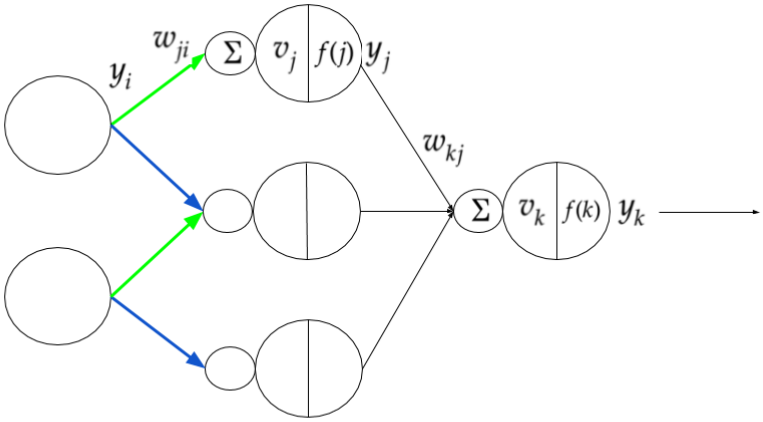
$$\delta_j(t) = f'(v_j(t))\sum_{1}^{k_n}\delta_k(t) \times w_{kj}(t)$$

Once these deltas are calculated, the weights for the network at time step $t+1$ can be calculated as:

$$w_{kj}(t+1) = w_{kj}(t) - \alpha\delta_k(t)y_j(t)$$

$$w_{ji}(t+1) = w_{ji}(t) - \alpha\delta_j y_i(t)$$

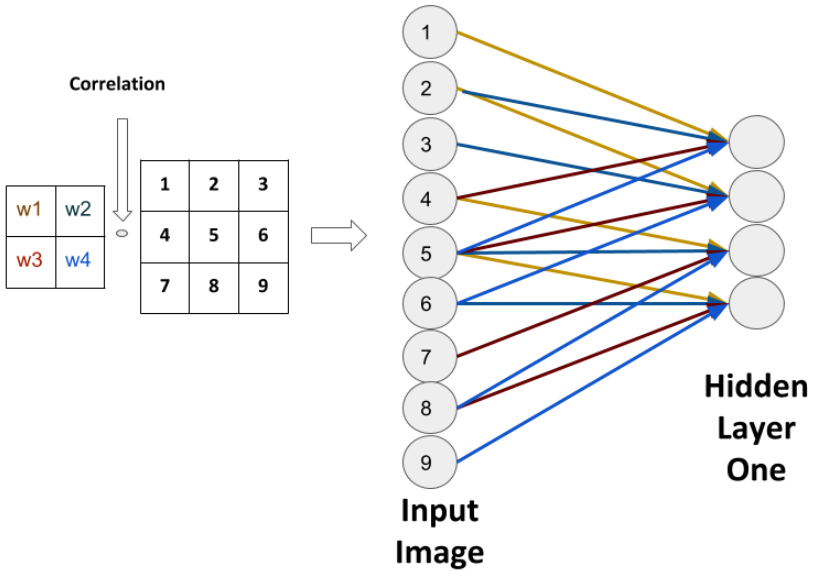
Shared Weight & Convolutional Networks



In the above modified diagram, the number of weights connecting the first and second layers are reduced from six to two, and these weights are shared between the neurons in the first and second layer. While before each weight was only used a single time in a forward pass through the network, each weight at the first layer is now used twice. This means that the weight update needs to consider all nodes the weight connected to.

$$w_{ji}(t+1) = w_{ji}(t) - \alpha\left(y_i^{(1)}(t)\,\delta_j^{(1)}(t) + y_i^{(2)}(t)\,\delta_j^{(2)}(t)\right)$$

Where the superscripts refer to the top two nodes in the first and second layer. These shared weights can also be designed to correspond with a filter sliding over an image. When this architecture is used, this corresponds to learning an optimal filter to extract a particular pattern. The filter then becomes most active when the filter finds a match in the input image.



Activation Functions

Nonlinear activation functions are important since they allow the network to learn nonlinear features from the data. When a nonlinear activation function is used, it is possible to find a neural network whose output $g(x)$ satisfies $|g(x) - f(x)| < \varepsilon$ for all inputs x, provided enough hidden neurons [Universal Approximation Theorem; 2]. One popular activation function is the logistic function:

$$\sigma(x) = \frac{1}{1 + e^{-\lambda x}}$$

$$\sigma'(x) = \lambda * \sigma(x) * (1 - \sigma(x))$$

Where λ is a slope parameter controlling how gradual the slope is. One drawback to the logistic function is that its outputs are always positive and therefore must have a mean that is positive–this can cause problems in the following layer, because if another logistic function is used, only half of its input domain would be used unless most weights are negative [1]. The $tanh$ function centers the output to 0, which can aid learning in subsequent layers.

$$tanh(x) = \frac{e^{\lambda x} - e^{-\lambda x}}{e^{\lambda x} + e^{-\lambda x}}$$

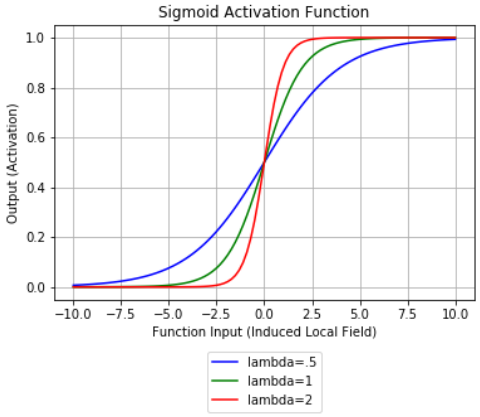
$$tanh'(x) = \lambda * \left(1 - tanh(x)^2\right)$$

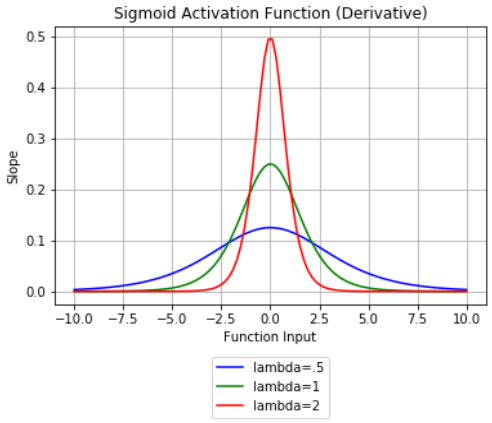
Both the $tanh(x)$ and $\sigma(x)$ nonlinearities have the disadvantage of saturating as the input approaches $-\infty$ or ∞. The ReLU and leaky ReLU are simple nonlinearities which are small (or zero) if the input is less than zero and the identity if the input is greater than zero. Here, small is defined by a 'leaky' parameter (here γ) which allows some gradient to flow back through the network to prevent the dying neuron problem. When $\gamma = 0$, the activation is a standard ReLU.

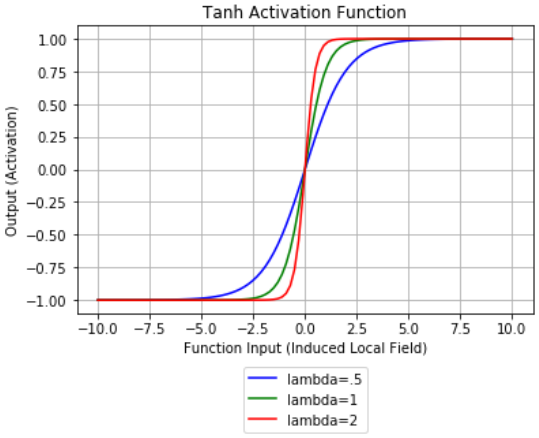
$$ReLU(x) = max(\gamma * x, x)$$

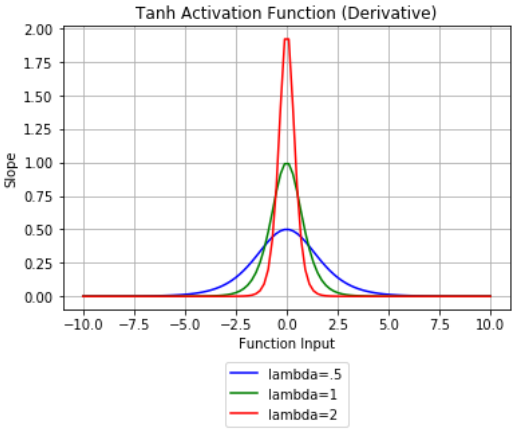
$$ReLU'(x) = \begin{cases} \gamma & if\ x < 0 \\ 1 & if\ x \geqslant 0 \end{cases}$$

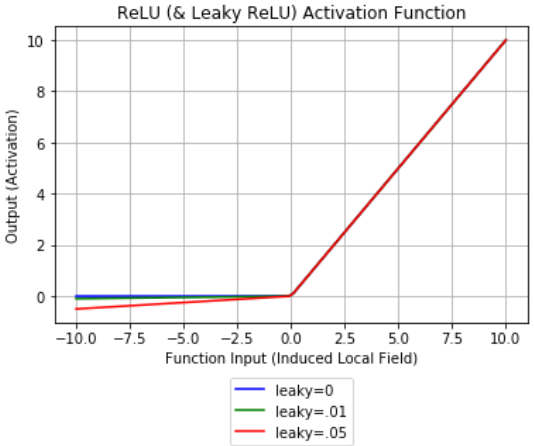
The following page includes plots for these nonlinearity functions and their derivatives across and hyper parameters.

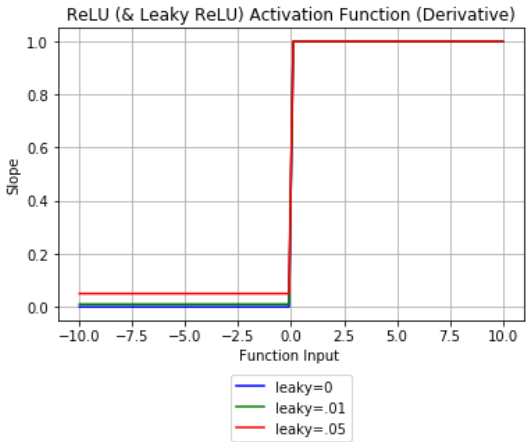

Sigmoid Activation Function
Output (Activation)
Function Input (Induced Local Field)
lambda=.5
lambda=1
lambda=2


Sigmoid Activation Function (Derivative)
Slope
Function Input
lambda=.5
lambda=1
lambda=2


Tanh Activation Function
Output (Activation)
Function Input (Induced Local Field)
lambda=.5
lambda=1
lambda=2


Tanh Activation Function (Derivative)
Slope
Function Input
lambda=.5
lambda=1
lambda=2


ReLU (& Leaky ReLU) Activation Function
Output (Activation)
Function Input (Induced Local Field)
leaky=0
leaky=.01
leaky=.05


ReLU (& Leaky ReLU) Activation Function (Derivative)
Slope
Function Input
leaky=0
leaky=.01
leaky=.05

Code Description

The CNN for parts two and three was implemented as a Python class (part two CNN_P2F.py; part three CNN_P3.py). The differences between these classes are very small and mostly involve a different weight initialization scheme. Forward and backward propagation were both implemented in a generic vectorized fashion so that the image size, filter size, and number of filters can be dynamically changed during configuration. The convolution operation was also implemented as a vectorized toeplitz matrix. In practice, however, this matrix took too long to build for each sample in part three (~ 5 seconds), so a faster scipy 2D correlation implementation was used for the forwards pass instead. The toeplitz is still useful for backprop since it's nonzero entries can be used to Boolean index the duplicate input array during convolution weight updates, and the toeplitz transpose is used for deconvolution.

To import the class (or data wrapper for interfacing with the dataset), import the class for the respective part:

```
from data_provider import DataProvider
from CNN_PT2F import ConvNeuralNetwork
from CNN_PT3 import ConvNeuralNetwork
```

Note above that the class imports above need to have different aliases if they are both going to be used in the same file. The Python class provides a simple abstraction for model experiments. For example, to train a model using ReLU on the smaller part 3 architecture for 100 epochs:

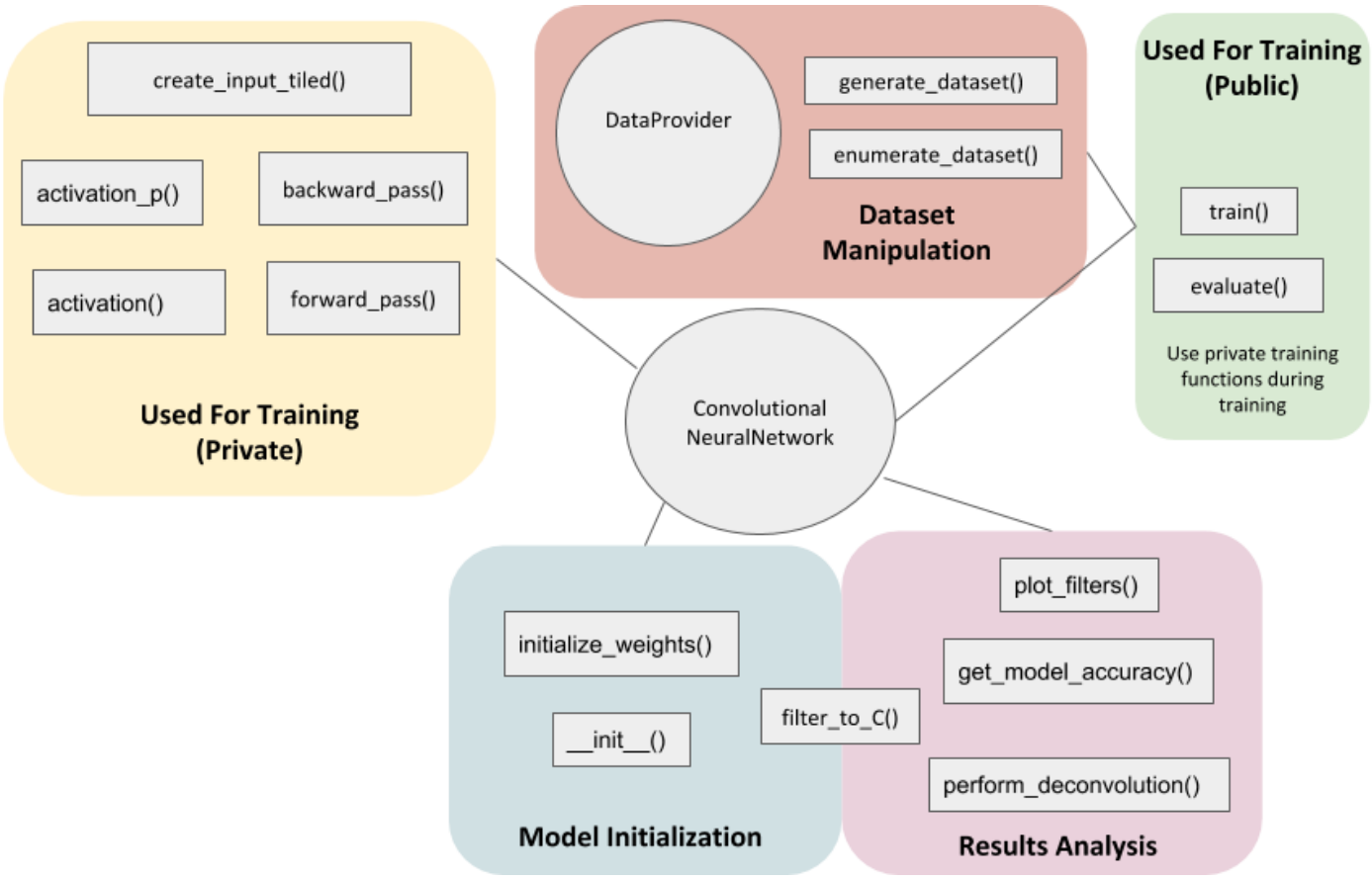
```
parts = ['3a', '3b']
cnn =  ConvNeuralNetwork(part=parts[0],
                         alpha=.001,
                         subset_size=subset_size,
                         activation_function='relu',
                         relu_alpha=.01)

train_mses, test_mses, grads = cnn.train(epochs=100, shuffle=True)
```

Though convolutional parameters can be provided directly (i.e filter_width, img_width, num_filters), specifying the part will do so automatically. Once a model is trained, functions are available to get information about the trained model:

```
cnn.get_model_accuracy()
cnn.perform_deconvolution()
cnn.plot_filters()
```

Examples of using the code can be seen in P2_experiments.ipynb, P3_experiments.ipynb, and P4_experiments.ipynb, which all include the code used to run the reported results.



The above diagram shows the interactions of various components of code, grouped by the functionality. Functions within each region rely on one another, and there are some inter group interactions which are not included. Specifically:

- Private training functions are called by public training functions.
- Filter_to_C (toeplitz creation) is used during model initialization, back propagation, and results analysis.
- A data provider instance can be provided directly to a training function to override the instance method. This for running multiple experiments because loading the full dataset can be time consuming.

The DataProvider also enables dynamically changing the sample proportion, which was used to train on a subset of data. This was used for selecting the first 100 samples in part 3. Additionally, the DataProvider performs the input normalization described below.

Experiments and Results

Experiments

In Part 2 and Part 3, several parameters were manipulated in order to understand the influence of various factors on the training process. These included: (a) the activation function, (b) the learning rate and (c) data presentation (randomized vs. serial). Specifically, the tanh, sigmoid, and ReLU nonlinearities were evaluated–non-linearity parameters such as the ReLU leak constant were also assessed. The specifics of the non-linearity and learning rate will be provided in figure captions corresponding with each experiment. Unless otherwise stated, data was presented in a randomized order.

The Part 2 dataset presented no computational challenge, and the full dataset was used for training and testing in the included experiments. The Part 3 dataset, however, did pose a larger barrier to the CPU-based numpy implementation of the network. Each epoch of training and testing on the full dataset required 72 seconds of CPU time (Red Hat v7.6 with Intel Xeon Gold 6140 CPU @ 2.30GHz). Therefore, initial experiments comparing the influence of various parameters were ran on a subset of the data (first 100 training and testing examples for each of the 10 classes). Then, a final experiment was conducted on the full MINST subset included with the project.

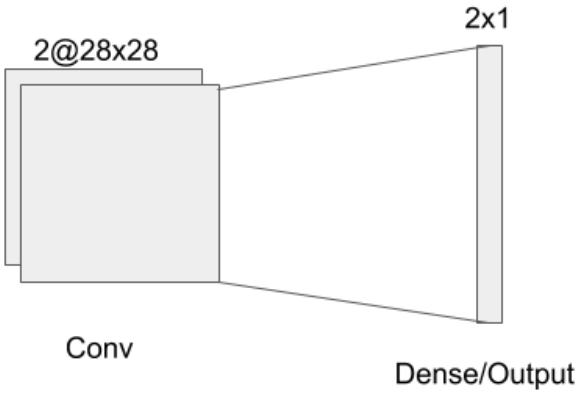
Experiments report a convergence plot reflecting the *Mean Squared Error* for the training and test data as a function of epochs. The final training and testing loss (MSE), as well as the final train and test accuracy after the last epoch will also be reported for each experiment as a reflection of the model's performance. All experiments (except for the final part 3 one on the full dataset) ran for a fixed 200 epochs–this was to maintain consistency across Part 2 and Part 3; since Part 3 was too computationally intensive to train to below a predefined epsilon, a fixed stopping point was set and maintained. Some plots also report the sum of squared deltas at the convolutional layer. This metric is included because it can help elucidate how the network is functioning internally, and is useful for understanding how hyperparameters affect this process. This metric is calculated as:

$$Mag\ Conv\ Deltas\ = \frac{1}{N}\sum_{M} \| \delta_{Conv} \|$$

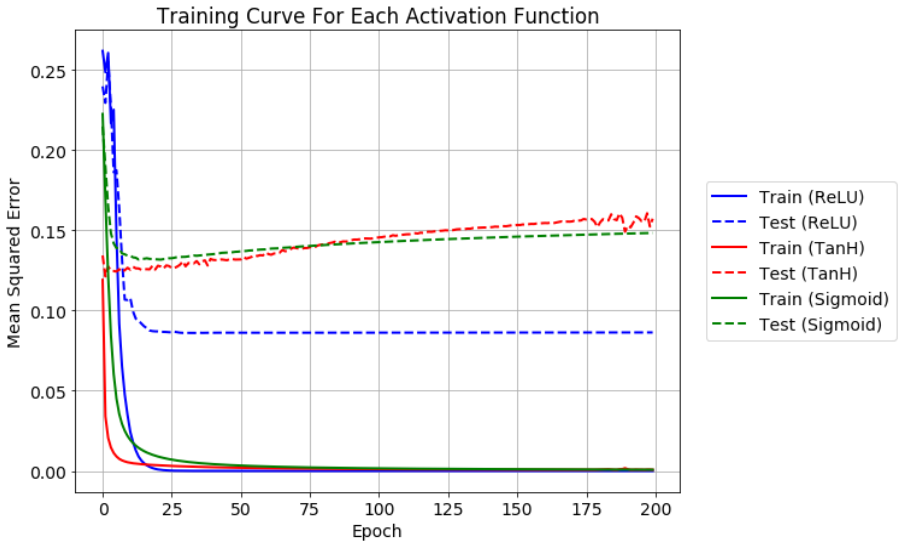
Where M is the number of nodes in the flattened convolutional layer, and N is the number of training samples. This metric is only reported on the training data since the deltas are only involved in backpropagation. The filters learned through the network are also visualized for some experiments, and deconvolution is performed and included in the final results, giving some intuition as to the features the network is learning.

Part 2 Results

Part two involved training a simple convolutional neural network with two 28x28 filters matching the size of the input image. These two filters were then used to compute a 2x1 response field mapping to the network output.



The first experiment examined differences in training between the ReLU, TanH, and Sigmoid activation functions. All three functions quickly dropped in MSE then leveled off, with the training error converging much lower than the test error. ReLU performed best with a test accuracy of 89 %. All three activation functions converged on the training data, producing an accuracy of 100 %. Interestingly, the Sigmoid and TanH loss both begin to gradually increase as training continued, indicating the model may be overfitting. ReLU, however, remained at a steady test loss after it initially converged.

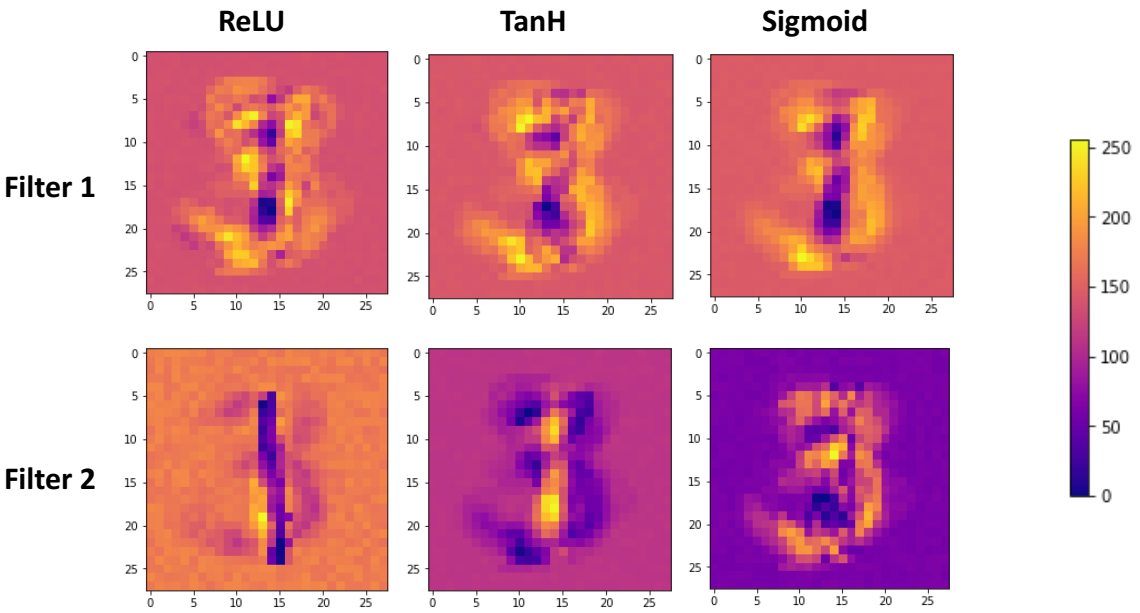


Convergence plot for the ReLU, TanH, and Sigmoid activation functions training on the Part 2 dataset. All $\alpha = .01$ *with shuffling enabled, ReLU* $\gamma = .01$ *, tanh slope = 1, sigmoid slope = 1.*

	Train MSE	Test MSE	Train Accuracy	Test Accuracy
ReLU	*1.13 e-06*	*0.086*	*100 %*	*89%*
TanH	*6.98 e-03*	*0.157*	*100 %*	*79%*
Sigmoid	*7.68 e-03*	*0.148*	*100 %*	*78%*

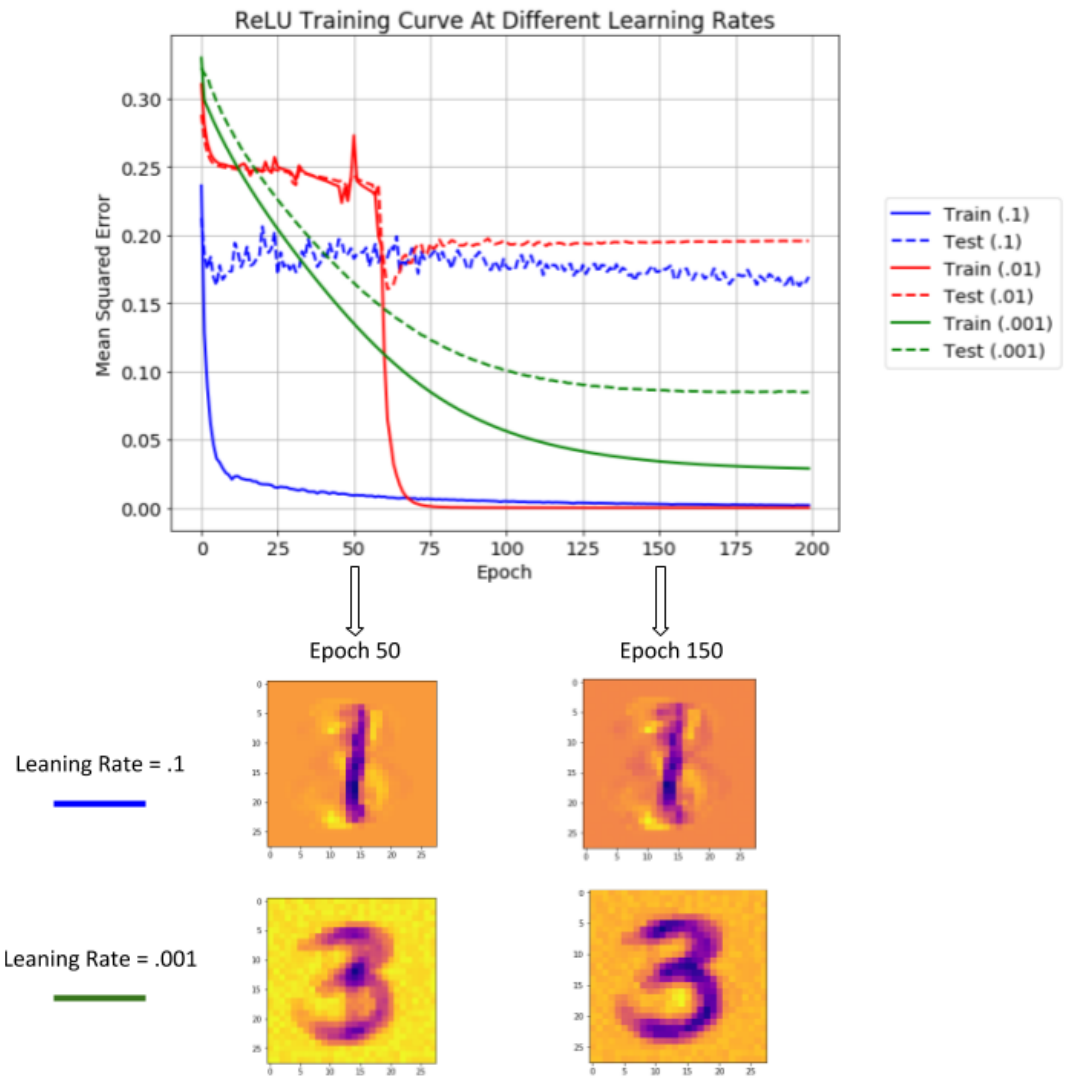
Accuracy and final MSE for the above training curves.

The filters learned by the network using each of the activation functions resembles a combination of the classes included in the dataset. Though I anticipated that one filter would be a three, and the other, a one, the resulting filters were a combination of the two. The empty areas of the three appear to be particularly discriminative of a one, while the surrounding space is more indicative of a three. Interestingly, these filters are also opposite signs. One becomes positive in response to a three and negative in response to a one, while the other becomes negative in response to a three and positive in response to a one. It would be interesting to seed the filters with an image of a three and a one and measure whether this filter is maintained, or if the below patterns re-appear.



Filters learned by each of the three activation functions. Images were produced by mapping numpy float64 range to uint8 [0, 255] before plotting.

It is possible that the resulting filter is 'seeded' by a factor of chance–a combination of the initialized weights and the first several training examples. If this is the case, this pattern may be more prevalent for higher learning rates when the influence of each training example is larger. Therefore, the next experiment examined effect of different learning rates on convergence of the network (using ReLU as an activation), to see if this would affect the learning rate and resulting filters.



The effect of different learning rates on the convergence of the network when the ReLU function ($\gamma = .01$ for all experiments; shuffling enabled; $\alpha = .1,\ .01,\ .001$). The top of the figure shows convergence plots, while the bottom shows filters learned at a high and low learning rate at different stages of the training process. The filter plotted at Epoch 50 & 150 is the same.

Shuffling Enabled

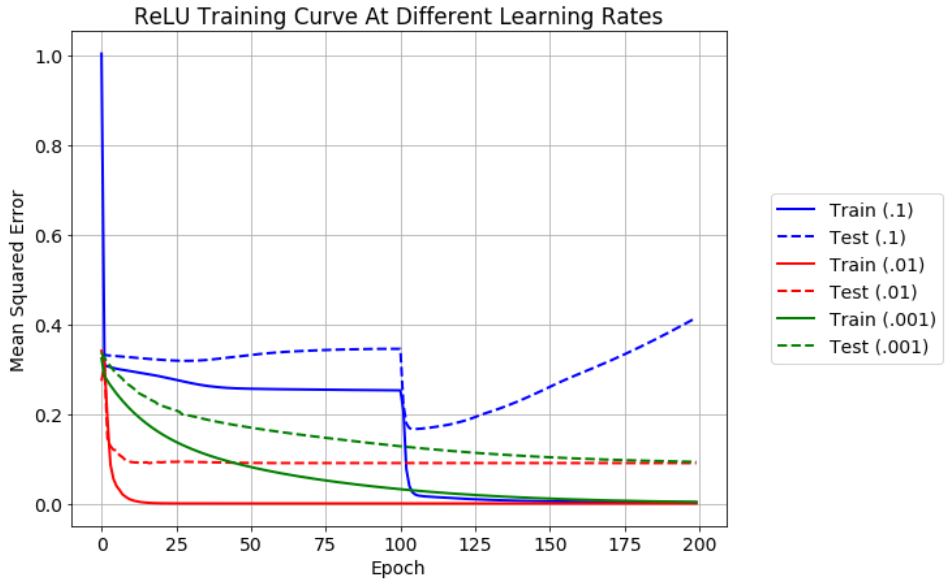
	Train MSE	Test MSE	Train Accuracy	Test Accuracy
ReLU ($\alpha = .1$)	*1.84 e-03*	*0.169*	*100 %*	*76%*
ReLU ($\alpha = .01$)	*3.46 e-06*	*0.195*	*100 %*	*74%*
ReLU ($\alpha = .001$)	*2.86 e-02*	*0.084*	*97 %*	*92%*

The network converged more stably (albeit slowly) when trained with a lower learning rate. Though this ultimately led to a lower training accuracy, the resulting test accuracy was higher. Higher learning rates showed more sporadic MSE, with the models eventually becoming stuck

at a high test loss. This seems to indicate that the network (under higher values of α) may have been trapped vacillating between variations in each sample rather than making steady progress towards a lower overall error–periodically (i.e. epoch 60 with $\alpha = .01$), the model found an area of the error surface where it could converge more fully instead of continuously vacillating.

Interestingly, the filters learned with a lower learning rate appear to be crisper and of higher quality. One hypothesis is that this is due to a more steadily guided weight update in the desired direction, as opposed to larger updates aligned with one sample at a time. The progression of the filter across epochs is also insightful–whereas the filter from $\alpha = .1$ appears to get less crisp over time, the $\alpha = .001$ filter appears to increase in quality (as seen by brightening in the area a 1 would usually be).

Since the relative contribution to the weight update (learning rate) had an effect on the smoothness of the training curve, it is also interesting to consider whether the order (fixed or random) of each update has an effect. Perhaps a non-random data presentation could attenuate the more sporadic training curve observed for higher learning rates. Therefore, the same experiment was conducted with fixed data presentation and included below.



The same configuration as the plot above, except with shuffling disabled. This emphasizes the smoother training curve (especially of test error) when the data is not shuffled.

Shuffling Disabled

	Train MSE	Test MSE	Train Accuracy	Test Accuracy
ReLU ($\alpha = .1$)	*2.22 e-03*	*0.414*	*100 %*	*75 %*
ReLU ($\alpha = .01$)	*4.96 e-07*	*0.090*	*100 %*	*89%*
ReLU ($\alpha = .001$)	*3.30 e-03*	*0.0928*	*100%*	*86%*

Accuracy and final MSE for the above training curves with shuffling disabled.

Though the convergence plots for non-random presented data appear to be more stable, the overall result isn't necessarily better–the random data presentation with $\alpha = .001$ still out performed all models trained on a standard with fixed data presentation. For each of the learning rates, the convergence plot shows different behaviors. The high learning rate condition ($\alpha = .1$), demonstrated a strong overfitting, with diverging train and test error curves. The middle level learning rate ($\alpha = .01$) ended up performing the best during the experiment, but showed a curve similar to the random presentation experiment in which error rapidly dropped then remained at the same value the remainder of the experiment. The lowest learning rate ($\alpha = .001$), displayed similar behavior of a smooth, continuous decrease in test error. This condition may have ultimately converged to a lower value if training had continued.

Whereas the learning rate will affect how much of the gradient from a training example is applied to the weight update, nonlinearity-specific parameters control how much gradient flows backwards through the network in a given sample. I was curious to see how changing the ReLU leak constant would affect training, and if increasing it would result in more overall gradient reaching the convolutional output layer. The *Mag Conv Deltas* measurement can help to understand this process. However, since the convolutional delta term is a product of the backwards-flowing gradient *and* the nonlinearity evaluated at the induced local field, it is difficult to tie this only to backwards flowing gradient. However, one clear interpretation of the *Mag Conv Deltas* term is that it is tied with how much the convolutional weights are being updated that particular sample.

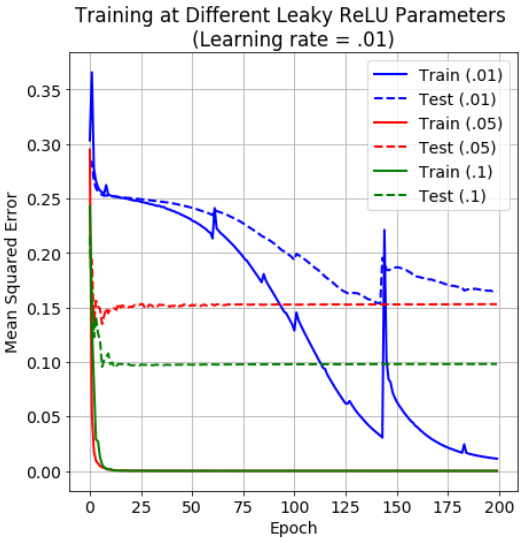
To evaluate the effect of non-linearity specific parameters, the next experiment includes a comparison of different ReLU leak constants ($\gamma = .01$, $\gamma = .05$, $\gamma = .1$) across different learning rates ($\alpha = .01$, $\alpha = .001$). I hypothesized that a larger γ would lead to less stability in training, but a γ that is too small would lead to slow convergence. I also predicted that a higher γ would result in larger deltas at the convolutional layer.

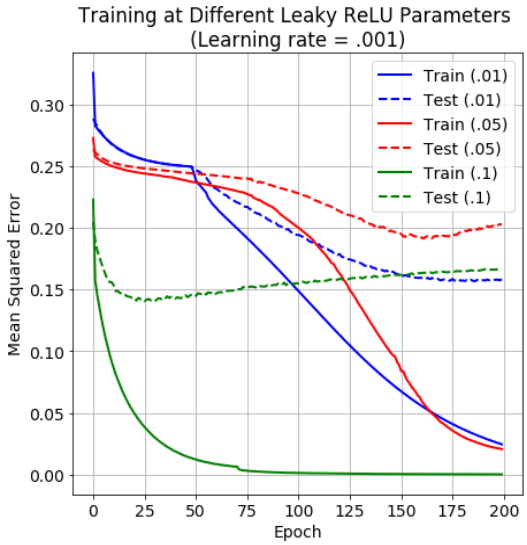
Learning Rate ($\alpha = .01$)

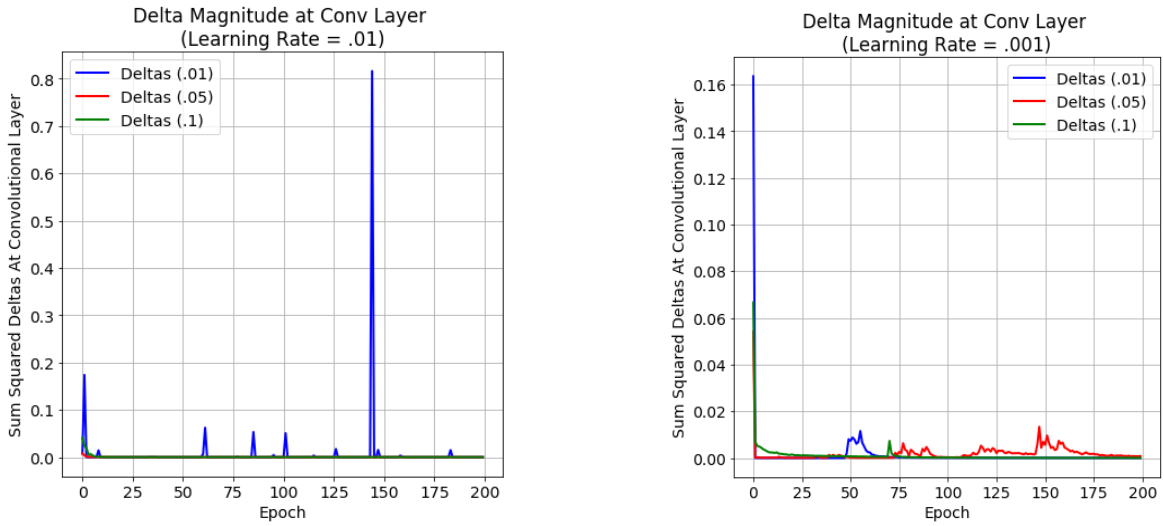
	Train MSE	Test MSE	Train Accuracy	Test Accuracy
ReLU ($\gamma = .01$)	*1.11 e-02*	*0.164*	*100 %*	*74 %*
ReLU ($\gamma = .05$)	*9.45 e-06*	*0.152*	*100 %*	*77 %*
ReLU ($\gamma = .1$)	*1.43 e-04*	*0.098*	*100 %*	*89 %*

Learning Rate ($\alpha = .001$)

	Train MSE	Test MSE	Train Accuracy	Test Accuracy
ReLU ($\gamma = .01$)	*2.46 e-02*	*0.158*	*100 %*	*79%*
ReLU ($\gamma = .05$)	*2.08 e-02*	*0.2029*	*97 %*	*74 %*
ReLU ($\gamma = .1$)	*3.59 e-04*	*0.166*	*100 %*	*74 %*







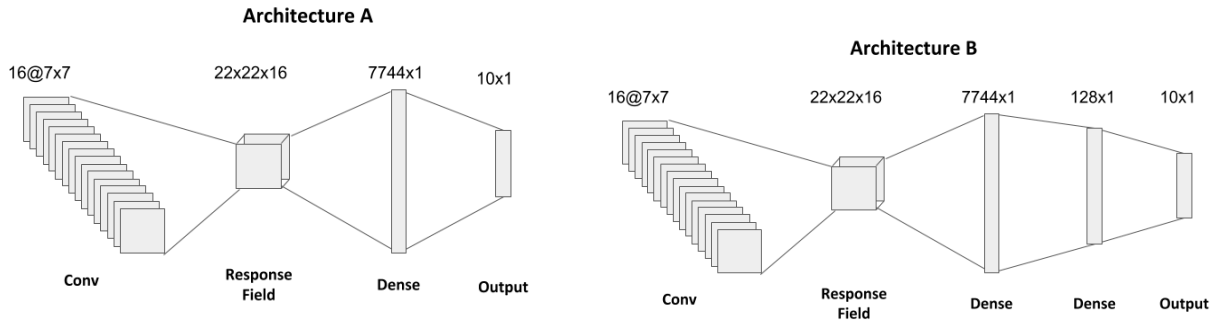
The effect of the ReLU leak constant and learning rate on convergence of the network.

Results showed that these predictions were not correct. For a higher learning rate, a larger γ performed better, while a smaller γ performed better for a lower learning rate. The convergence of the lower γ was also not the smoothest–in fact, it was the most varied. The smooth convergence of training at lower learning rates across γ does replicate the findings of the previous experiment, however. The sum of squared deltas reveals more gradient was flowing through the network during periods of faster convergence, although there does not appear to be a large baseline difference due to the γ value. Interestingly, there was a relationship between γ and α, with the optimal γ being a factor of 10 larger for this experiment. Overall, this suggests that there may be a relationship between γ and α, but this relationship can't be established conclusively from one experiment alone.

Part 2 offered some insights as to how various network and training parameters affect training, which then guided experiments for Part 3.

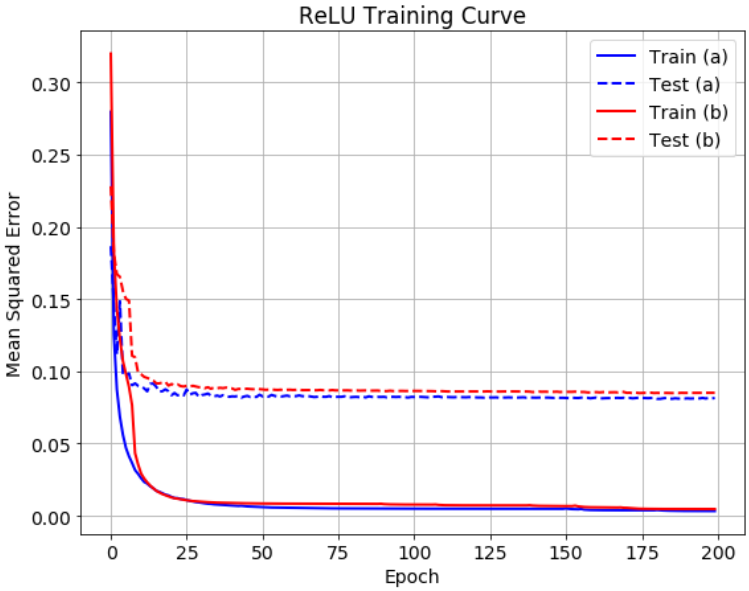
Part 3 Results

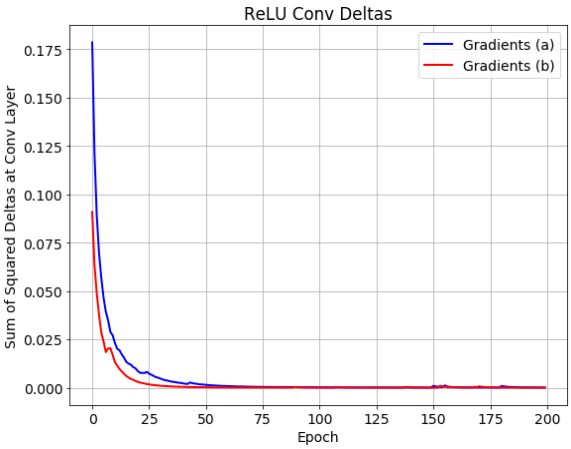
Part three included the adoption of the original model to include a total of 16 filters mapping to 10 output nodes (matching one-hot encoded labels). Part three also included two variations of the model: one with a dense layer mapping the convolutional response field directly to the output (Architecture A), and another including an extra dense layer of 128 neurons (Architecture B).

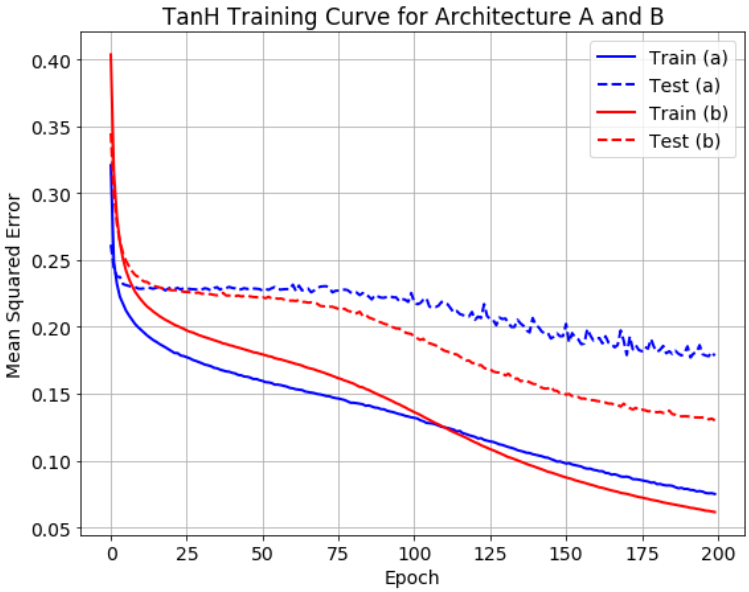


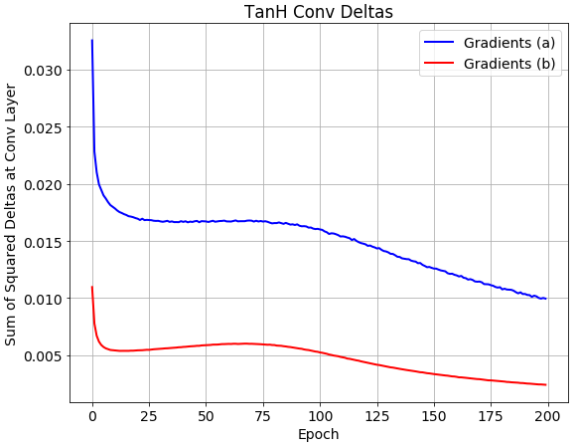
In practice, these models were more difficult to train and required careful weight initialization and nonlinearity tuning to avoid vanishing gradients. Therefore, I adopted a better weight initialization strategy suggested in [1]. Specifically, this included (a) input normalization with subtracting the mean pixel intensity and dividing by the standard deviation, and (b) weight initialization involving a Gaussian with the mean at zero and standard deviation at an appropriate value. Though I initially set the standard deviation to one, this proved to be too large, so I adopted the suggested $\sigma = m^{-1/2}$, where m is the number of inputs to the unit [1]. Part 3 also required making the slope of the TanH and Logistic function more gradual.

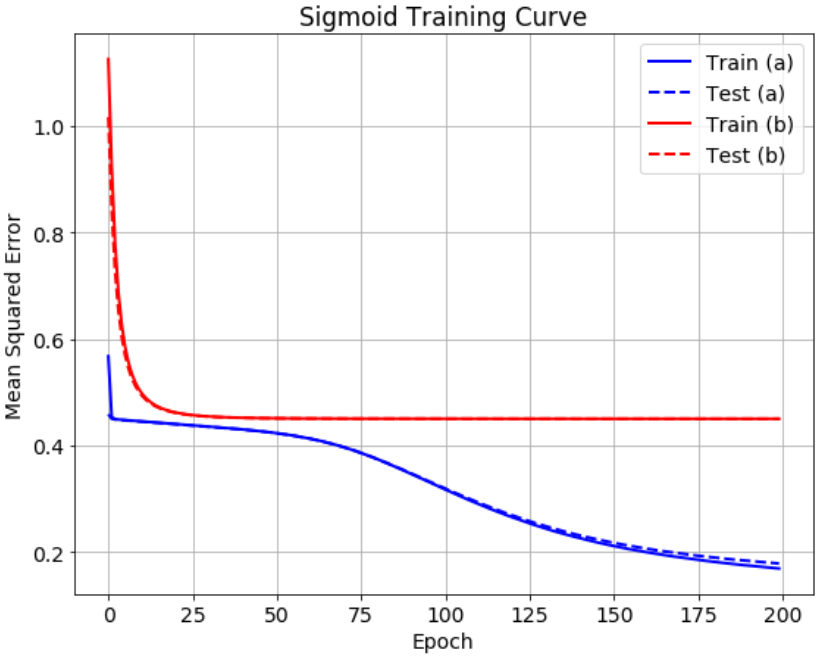
The first experiment in part 3 examined differences in training with the ReLU, TanH, and Sigmoid activation functions on the MNIST dataset with all digits (first 100 samples of each class). Experiments report error and accuracy for both architectures, as well as the training curve and magnitude of convolutional deltas.

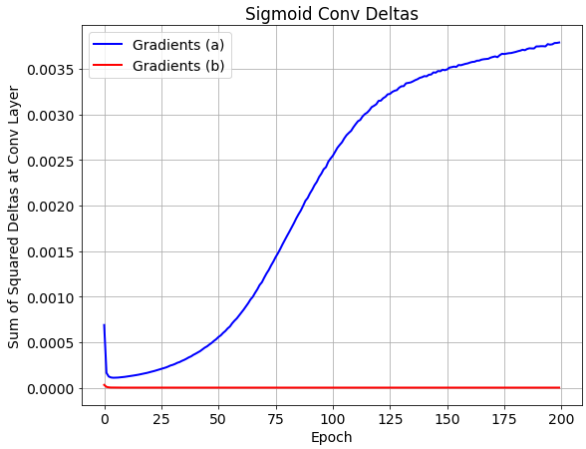

ReLU Training Curve
Train (a)
Test (a)
Train (b)
Test (b)
Mean Squared Error
Epoch


ReLU Conv Deltas
Gradients (a)
Gradients (b)
Sum of Squared Deltas at Conv Layer
Epoch


TanH Training Curve for Architecture A and B
Train (a)
Test (a)
Train (b)
Test (b)
Mean Squared Error
Epoch


TanH Conv Deltas
Gradients (a)
Gradients (b)
Sum of Squared Deltas at Conv Layer
Epoch





Left shows the error convergence plot for each activation function for both architecture A (shown in blue) and architecture B (shown in red). The smaller graph to the right shows sum of squared deltas at the convolutional layer during training. All training was done with $\alpha = .001$ *Activation function parameters: TanH (slope=.5), ReLU (leaky = .01), Sigmoid (slope = (.7, .7, .05) architecture B, slope = (.7, .05) architecture A).*

TanH

	Train MSE	Test MSE	Train Accuracy	Test Accuracy
Architecture A	*0.075*	*0.178*	*99.5 %*	*91%*
Architecture B	*0.0615*	*0.129*	*98.5 %*	*91.4%*

ReLU

	Train MSE	Test MSE	Train Accuracy	Test Accuracy
Architecture A	*3.29 e-03*	*8.15 e-02*	*99.5 %*	*92.9%*
Architecture B	*4.67 e-03*	*8.51 e-03*	*99.1 %*	*91.5 %*

Sigmoid

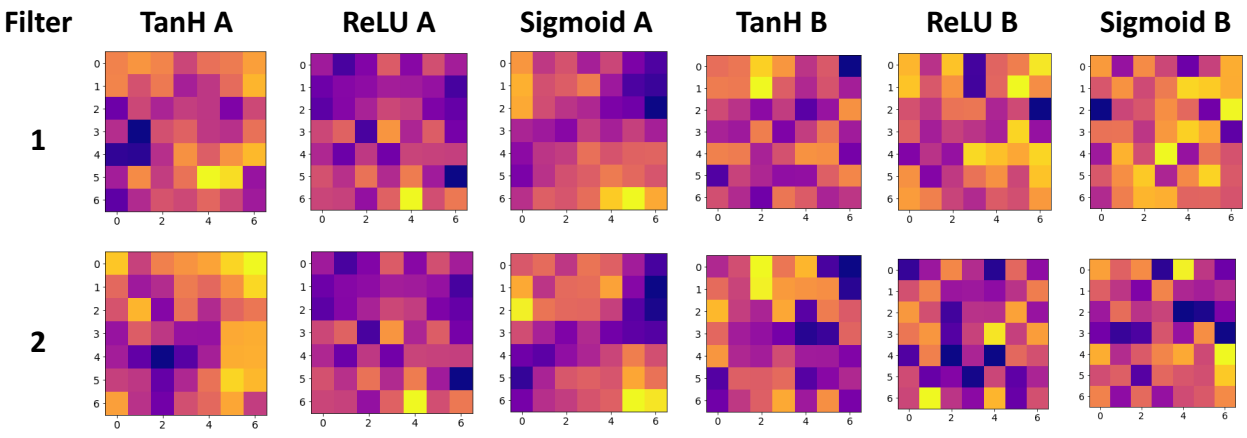
	Train MSE	Test MSE	Train Accuracy	Test Accuracy
Architecture A	*0.169*	*0.178*	*86.3 %*	*85 %*
Architecture B	*0.449*	*0.449*	*24.6 %*	*25.1 %*

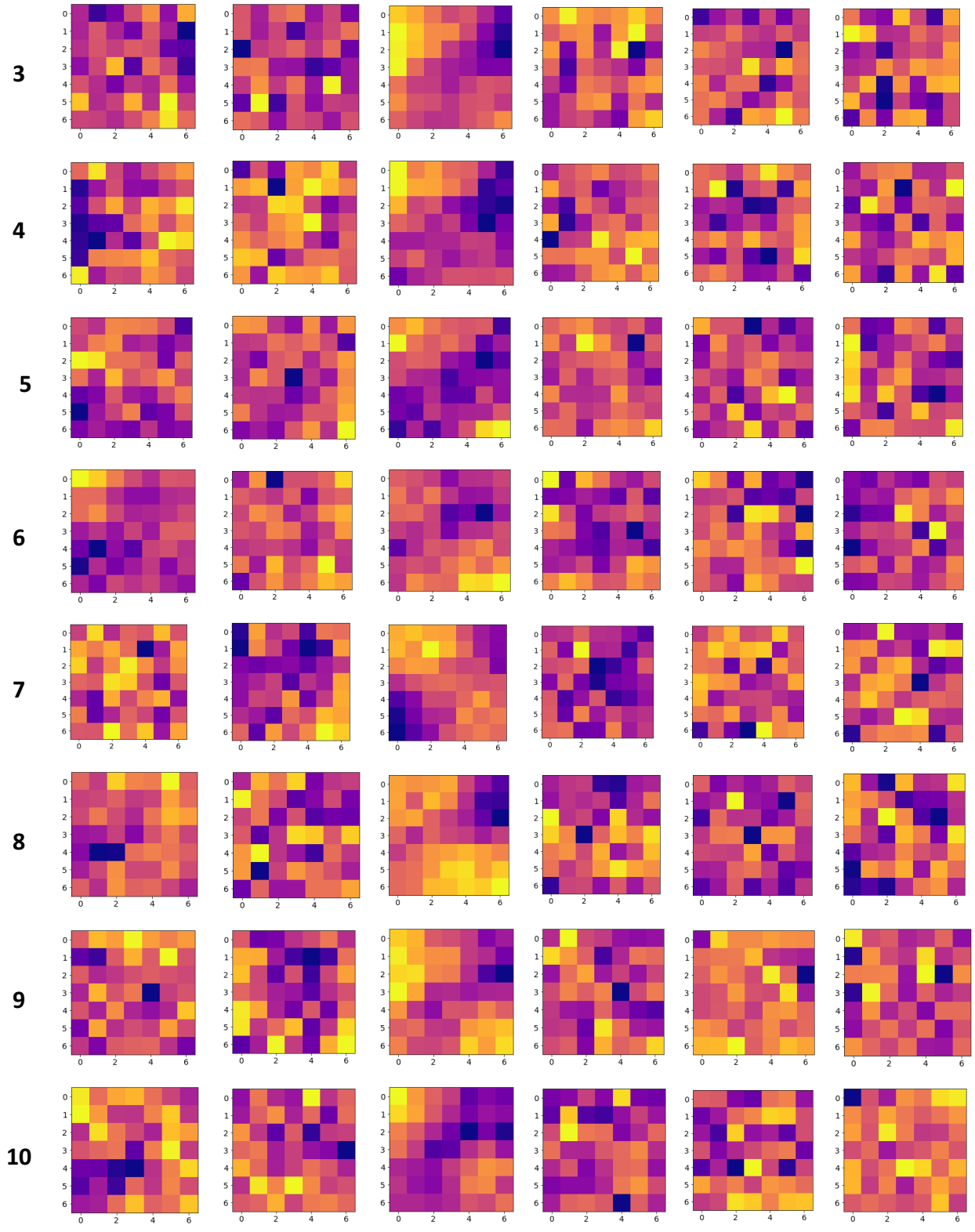
Accuracy and loss metrics for the above plots, showing performance across activations and architectures.

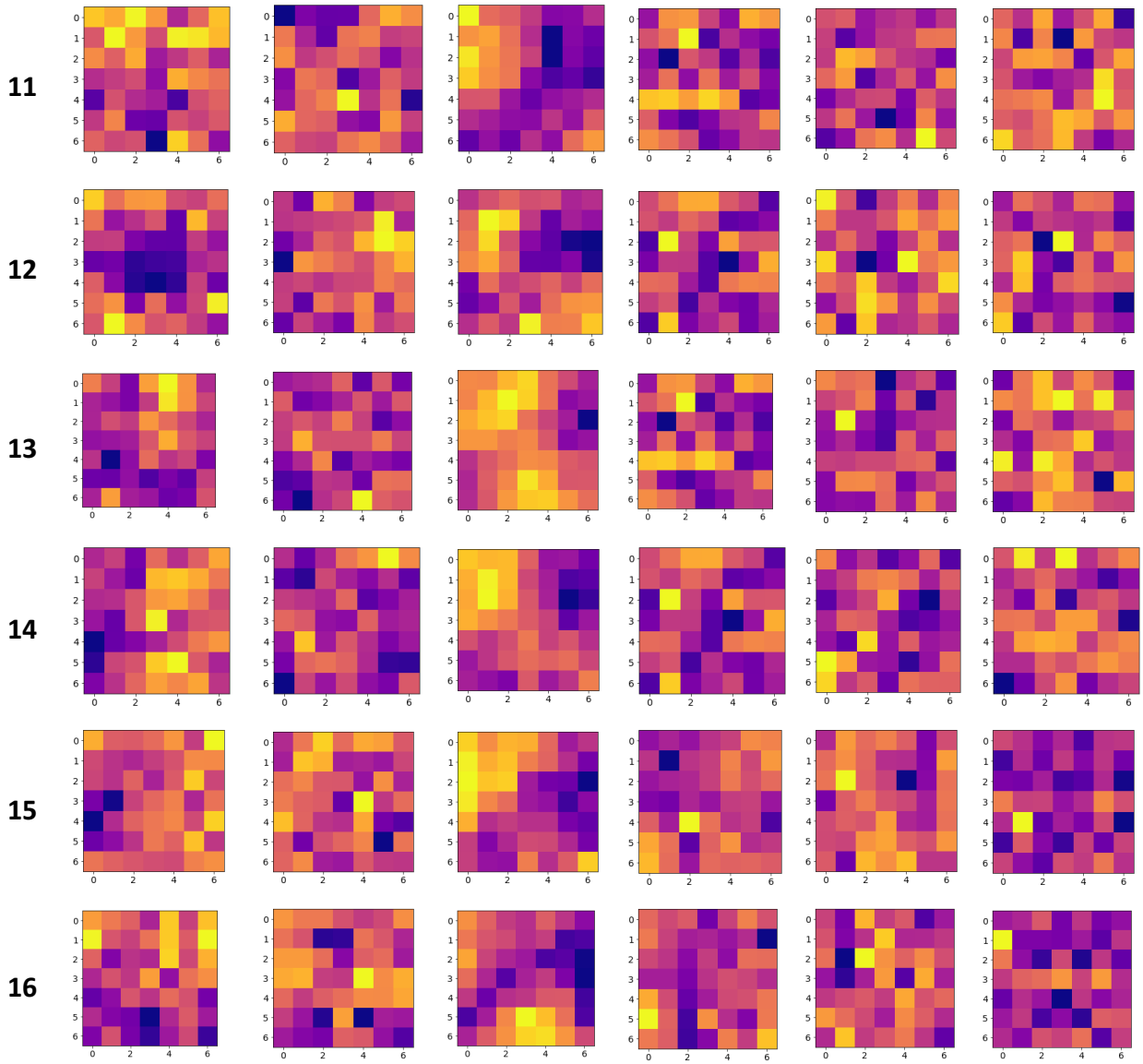
I hypothesized that the total gradient reaching the convolutional layer would be smaller with architecture (b) due to the extra layer to propagate deltas through. This was correct, but did not negatively impact the final results. Overall, ReLU performed the best, followed by TanH, and finally the Sigmoid. It was not conclusive as to which architecture performs better. The deltas curve complements the information in the training curve. Whereas the ReLU curve delta magnitude is initially very high then quickly levels off to a comparatively low value, the TanH curve gradually decreases as the network converges. As the sigmoid architecture A finally begins to converge, the deltas also increase. One interesting observation is the scale difference in the deltas. ReLU is an order of magnitude higher than TanH, which is an order of magnitude higher than Sigmoid. There are several factors which may have contributed to this, but one explanation is the different response characteristics of the three functions' derivatives (see page 6). Whereas ReLU does not constrain positive induced local fields to a smaller interval, Sigmoid and TanH does. This could be understood better by plotting both terms which are multiplied to calculate the deltas at the convolutional layer.

It is also possible to plot the 16 7x7 filters corresponding with the model to see which patterns they learned. The plot below shows the result of this. Interestingly, the filters for the best performing models appear noisy and do not reflect intuitive filters which would be useful for feature extraction. One model - the Sigmoid model with Architecture A - did learn smooth filters which convey colour gradients and smooth contours. However, this model did not demonstrate a high overall accuracy comparatively.

This suggests that a gradual, steady convergence may be helpful for extracting meaningful edge filters, supporting the results from Part 2 demonstrating the effect of learning rate on the learned filters.




3
4
5
6
7
8
9
10

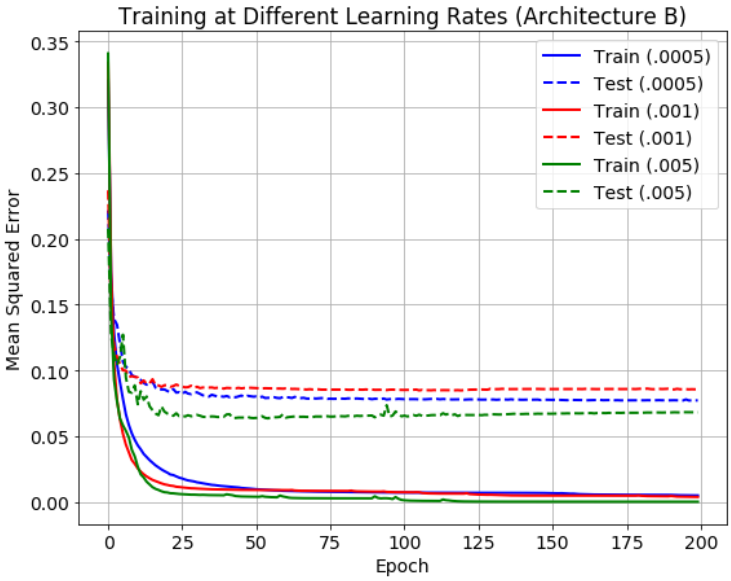


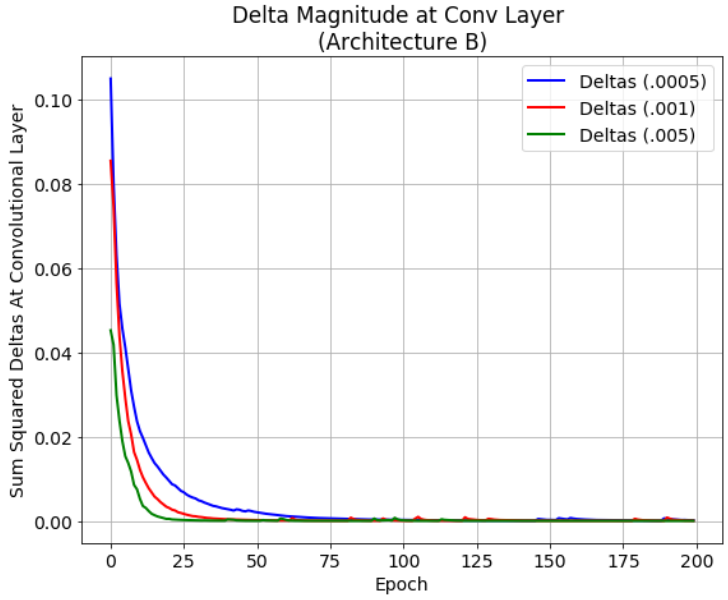
Filters extracted from the above experiment for each of the three activation functions across architecture A and B. Filters are plotted using the same method described for Part 2.

I was interested to learn whether a lower learning rate would produce smoother looking filters in part three, as it did in part two, so also decided to run another similar experiment involving different values of α with the ReLU nonlinearity. I intended to run this experiment on both architectures, however, the model produced an overflow error for architecture A with some values of α. I received these errors while first implementing part 3–they occurred especially for higher learning rates, which also occured here (for α=.005), the highest value tested. It is interesting that this only occurs with architecture A. The extra dense layer in B may have provided protection from large variations in model updates at a high learning rate.

Results reveal that there is not a large difference in terms of convergence or the extracted filters (most appear as nearly random noise) due to learning rate for the ReLU function. In fact, higher and lower learning rates perform better than the middle choice, but validation accuracy is close enough that this may be due to chance.

The delta magnitude plot suggests that (a) more gradient is flowing back through the network when the learning rate is lower, (b) activations from the convolutional layer fall closer to zero, or a combination of both. If (a), this may suggest that the lower learning rate is causing less nodes deeper in the network to become 'dead', and therefore they continue to propagate gradient backwards.



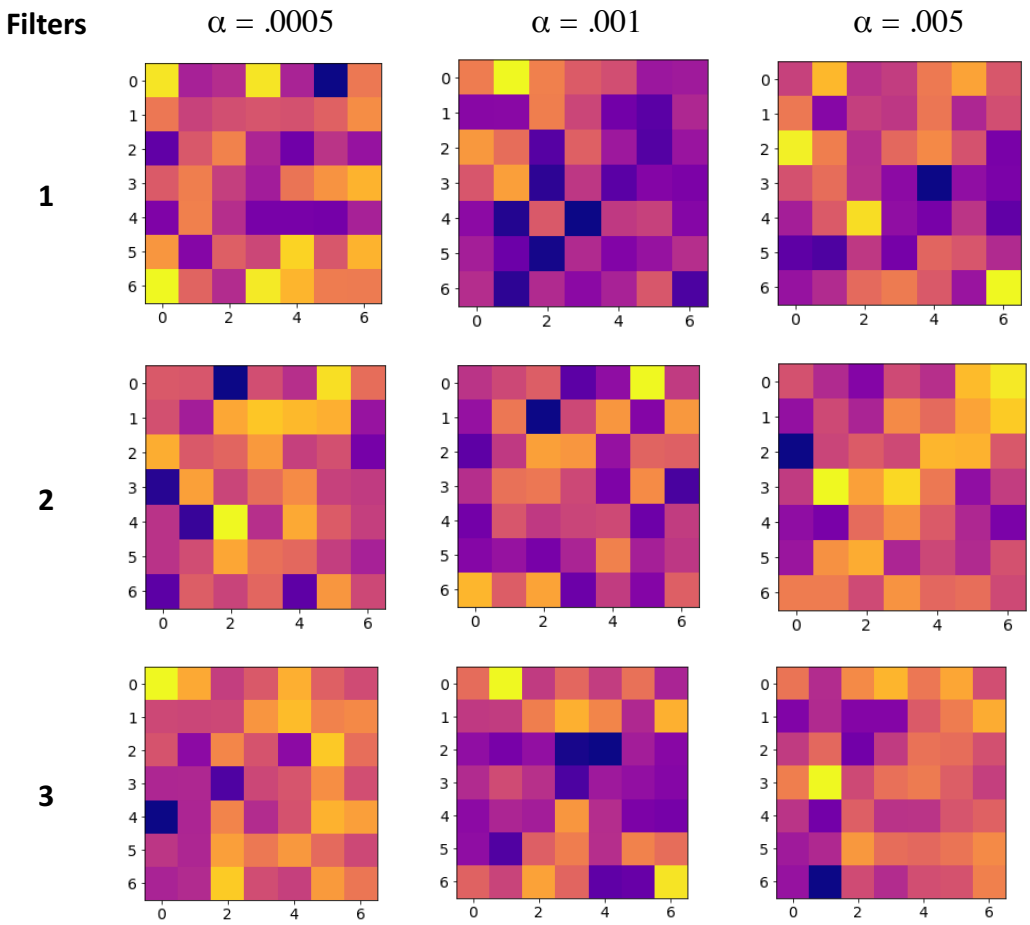


Error convergence plots for network trained with ReLU non-linearity and different learning rates ($\alpha = .0005,\ .001,\ .005$; leaky parameter set at .01 for all experiments; architecture B). The top plot shows a convergence plot, while the bottom shows the corresponding sum of squared deltas. This experiment was also ran on architecture A, but an overflow error occurred for $\alpha = .005$ so results were not plotted.

Architecture B

	Train MSE	Test MSE	Train Accuracy	Test Accuracy
$\alpha = .0005$	*4.87 e-03*	*7.71 e-02*	*99.1 %*	*93.7 %*
$\alpha = .001$	*3.73 e-03*	*8.54 e-02*	*99.3 %*	*92.2 %*
$\alpha = .005$	*1.16 e-04*	*6.80 e-02*	*100 %*	*93.9 %*

Performance metrics corresponding with the above experiment on learning rate.



First three filters generated by each experiment on learning rate. Filters are representative of the sixteen total filters learned.

In addition to the experiment on activation function and learning rate, another was conducted on data presentation (random vs. fixed) for both architectures. The convergence plots for the two settings appear nearly identical for both architectures, however shuffling data between epochs does appear to give a modest performance boost. I predicted that the filters learned through fixed data presentation may be more clear due to less randomness in the presentation of each sample, but there did not appear to be a major effect. Performance was roughly the same between the architectures.

The information provided regarding data presentation in [1] suggests that there is a large difference when many samples of the same class are presented sequentially. Therefore, it would also be interesting to compare a model trained on the non-shuffled dataset (all images of each class presented sequentially), to see how this performs.

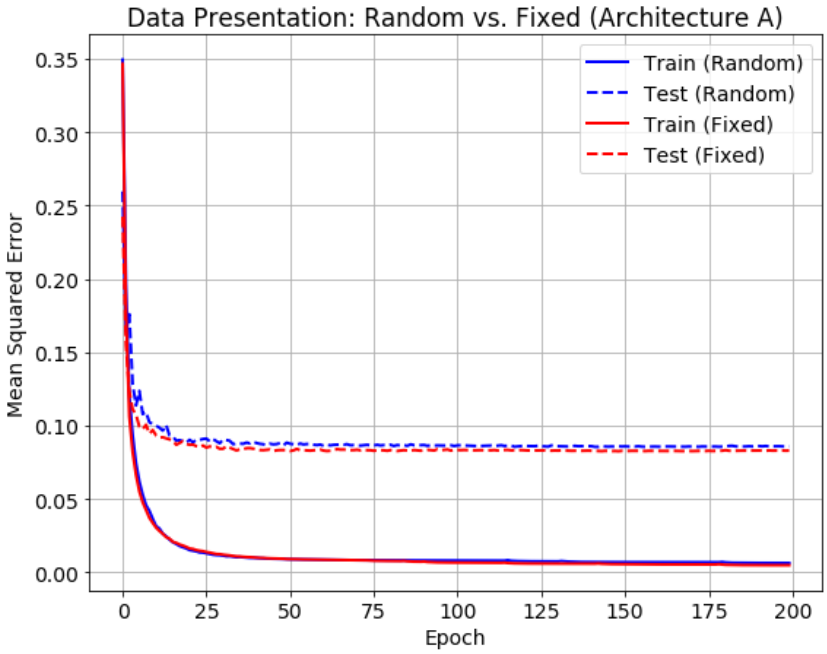
Architecture A

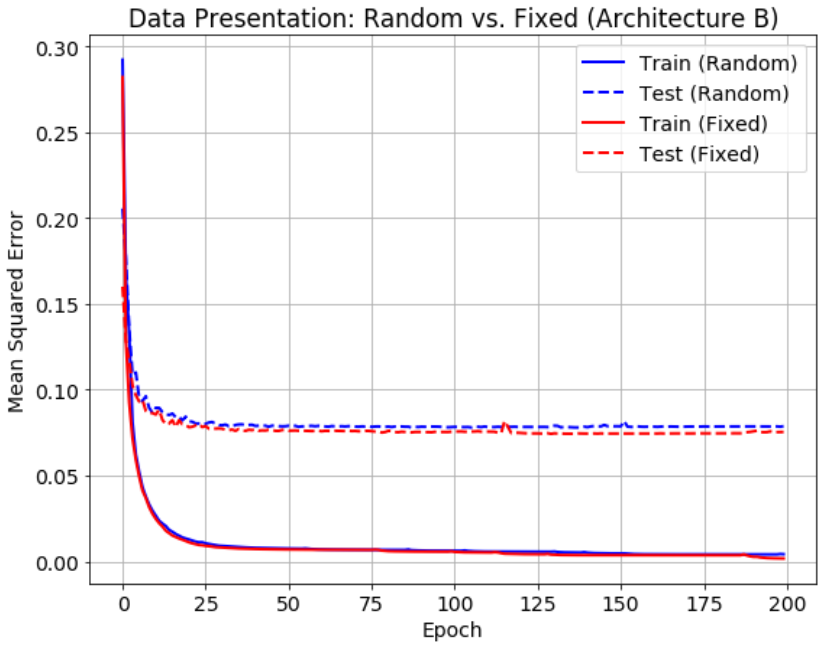
Data Presentation	**Train MSE**	**Test MSE**	**Train Accuracy**	**Test Accuracy**
Random	*6.28 e-03*	*8.57 e-02*	*98.8 %*	*93.2 %*
Fixed	*4.76 e-03*	*8.29 e-02*	*99.2 %*	*92.7 %*

Architecture B

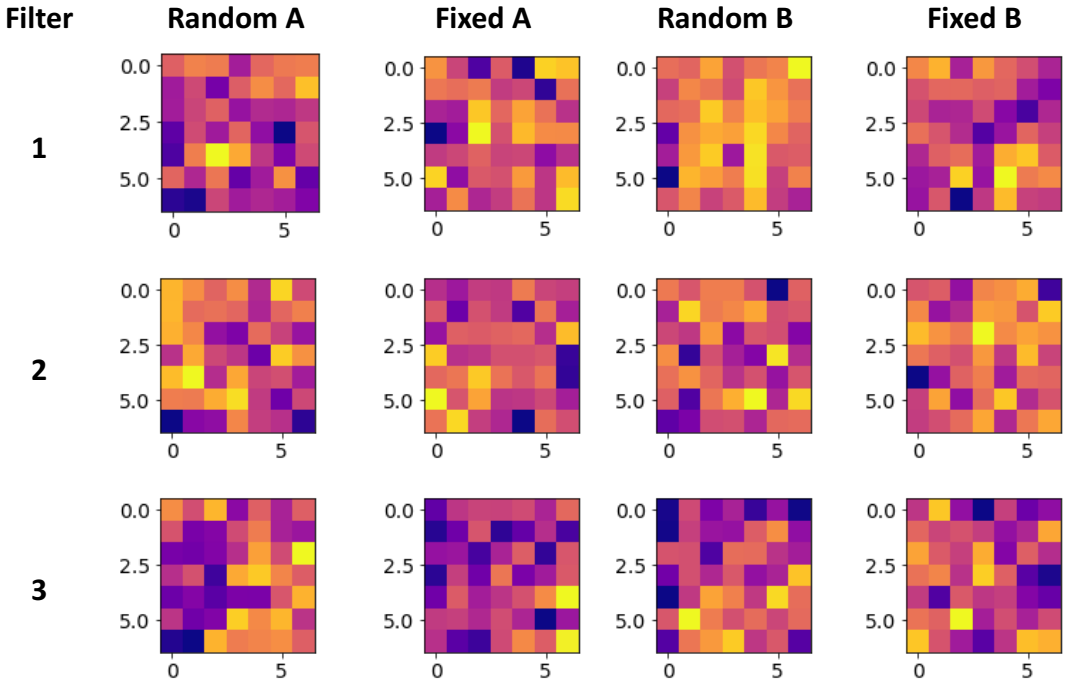
Data Presentation	**Train MSE**	**Test MSE**	**Train Accuracy**	**Test Accuracy**
Random	*4.26 e-03*	*7.87 e-02*	*99.4 %*	*93.5 %*
Fixed	*1.78 e-03*	*7.54 e-02*	*99.7 %*	*92.7 %*

Performance metrics corresponding with the above experiment on data presentation.



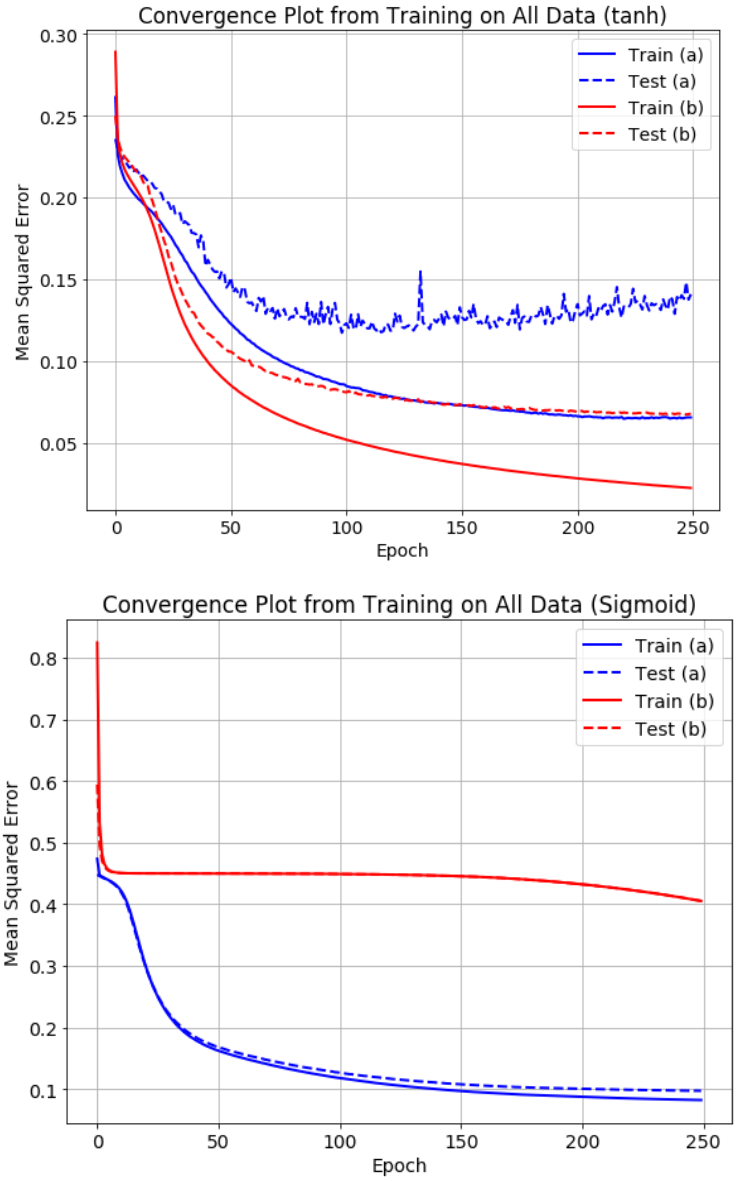


Error convergence plots for network trained with random and fixed data presentation ($ReLU with\ \gamma = .01,\ \alpha = .001$). The top shows training for architecture A, while the bottom shows training for Architecture B.



First three filters learned in the experiment on fixed vs.random data presentation.

After the above experiments were conducted on a subset of the project data, an additional experiment was conducted on the full dataset for the TanH and Sigmoid nonlinearities; ReLU was omitted in case another overflow error occurred in the limited training period. A slightly longer training period of 250 epochs was used for this experiment.



Convergence plots for network architecture A and B trained on the full dataset for 250 epochs. Both experiments used $\alpha = .001$. *TanH used slope parameter of .5, while Sigmoid used slope parameters of sig_lambdas=(.7,.7) for architecture A and sig_lambdas=(.7,.7,.05) for architecture B.*

Both TanH architectures performed better in than sigmoid in this experiment, and this TanH experiment also out-performed the TanH models trained on the subset of the project data. The only variables changed different for this TanH experiment other than dataset size was number of epochs to train, which may have also contributed to the higher accuracy when training on the full data. Interestingly, the TanH experiment training curve did demonstrate a difference between the architectures–test loss was much more sporadic with architecture A.

The sigmoid architecture B experiment mistakenly used slope parameters of (.7, .7), as opposed to (.7, .05), which may explain the slow convergence of this experiment. This is illustrative of the effect of slope parameters on training, but can not be used to make conclusive claims since it wasn't tested in an isolated context. With more time, both sigmoid architectures likely would have converged more. The confusion matrix is also illustrative of the types of misclassified examples–for example the model often predicted a 9 when the real label was a 4 or a 7.

Architecture A

Activation	**Train MSE**	**Test MSE**	**Train Accuracy**	**Test Accuracy**
Sigmoid	*8.24 e-02*	*9.73 e-02*	*92.4 %*	*90.3 %*
TanH	*6.57 e-02*	*.141*	*99.6 %*	*94.7 %*

Architecture B

Activation	**Train MSE**	**Test MSE**	**Train Accuracy**	**Test Accuracy**
Sigmoid	*.495*	*.499*	*40.5 %*	*40.5 %*
TanH	*2.26 e-02*	*6.80 e-02*	*99.6 %*	*94.8 %*

Performance metrics for experiment training on entire dataset.

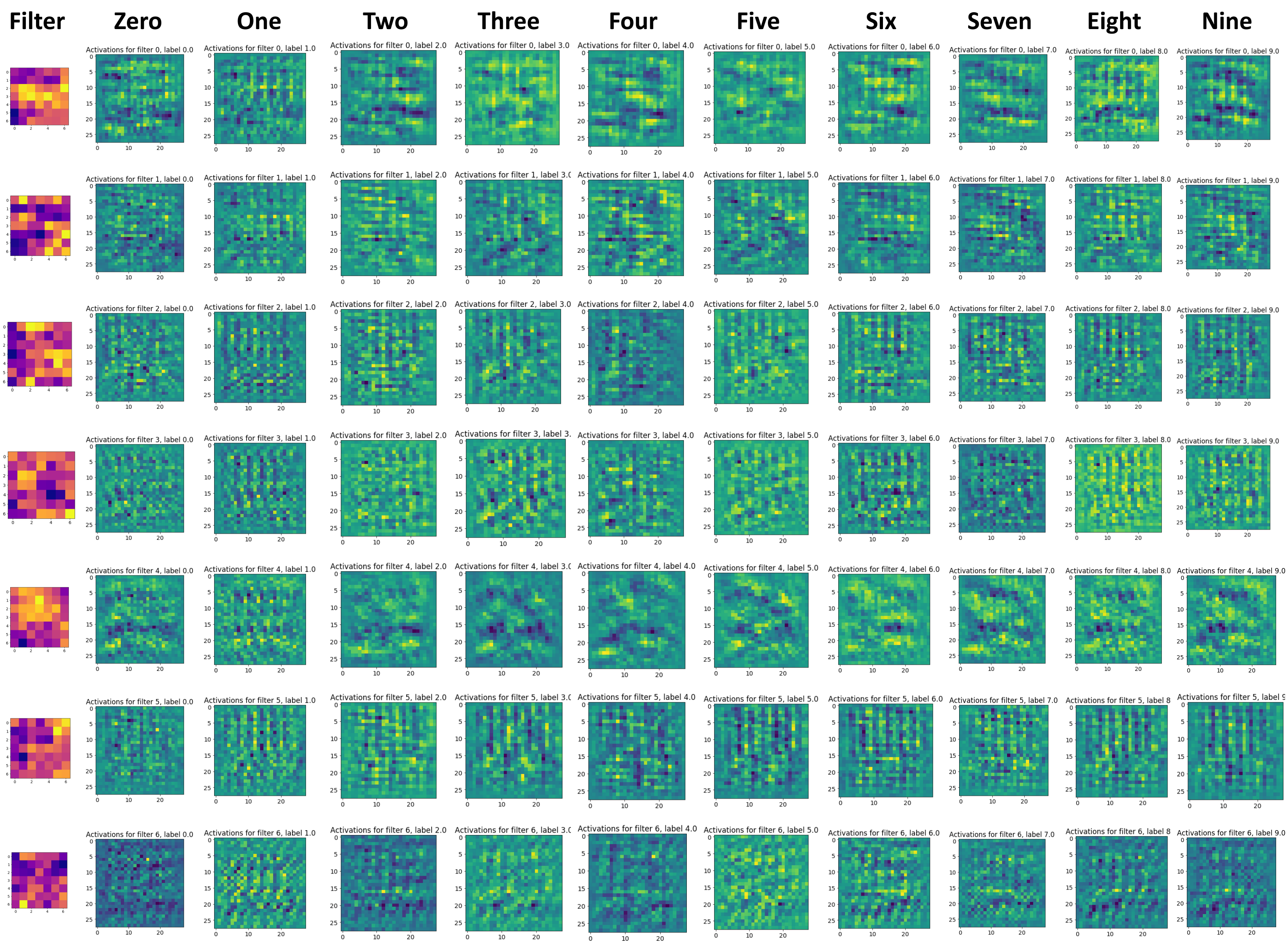
		Actual									
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Predicted	**0**	*489*	*0*	*0*	*4*	*0*	*6*	*4*	*2*	*8*	*1*
	1	*0*	*484*	*3*	*0*	*2*	*0*	*0*	*0*	*1*	*0*
	2	*1*	*8*	*473*	*4*	*1*	*0*	*2*	*4*	*5*	*1*
	3	*1*	*1*	*5*	*463*	*0*	*4*	*0*	*2*	*4*	*9*
	4	*2*	*2*	*1*	*1*	*475*	*3*	*1*	*4*	*5*	*8*
	5	*0*	*0*	*0*	*10*	*0*	*474*	*8*	*0*	*0*	*0*
	6	*0*	*0*	*0*	*0*	*2*	*2*	*483*	*0*	*8*	*0*
	7	*0*	*0*	*5*	*7*	*0*	*1*	*0*	*455*	*1*	*1*
	8	*5*	*3*	*8*	*4*	*0*	*9*	*1*	*2*	*468*	*1*
	9	*2*	*2*	*5*	*7*	*20*	*1*	*1*	*31*	*0*	*479*

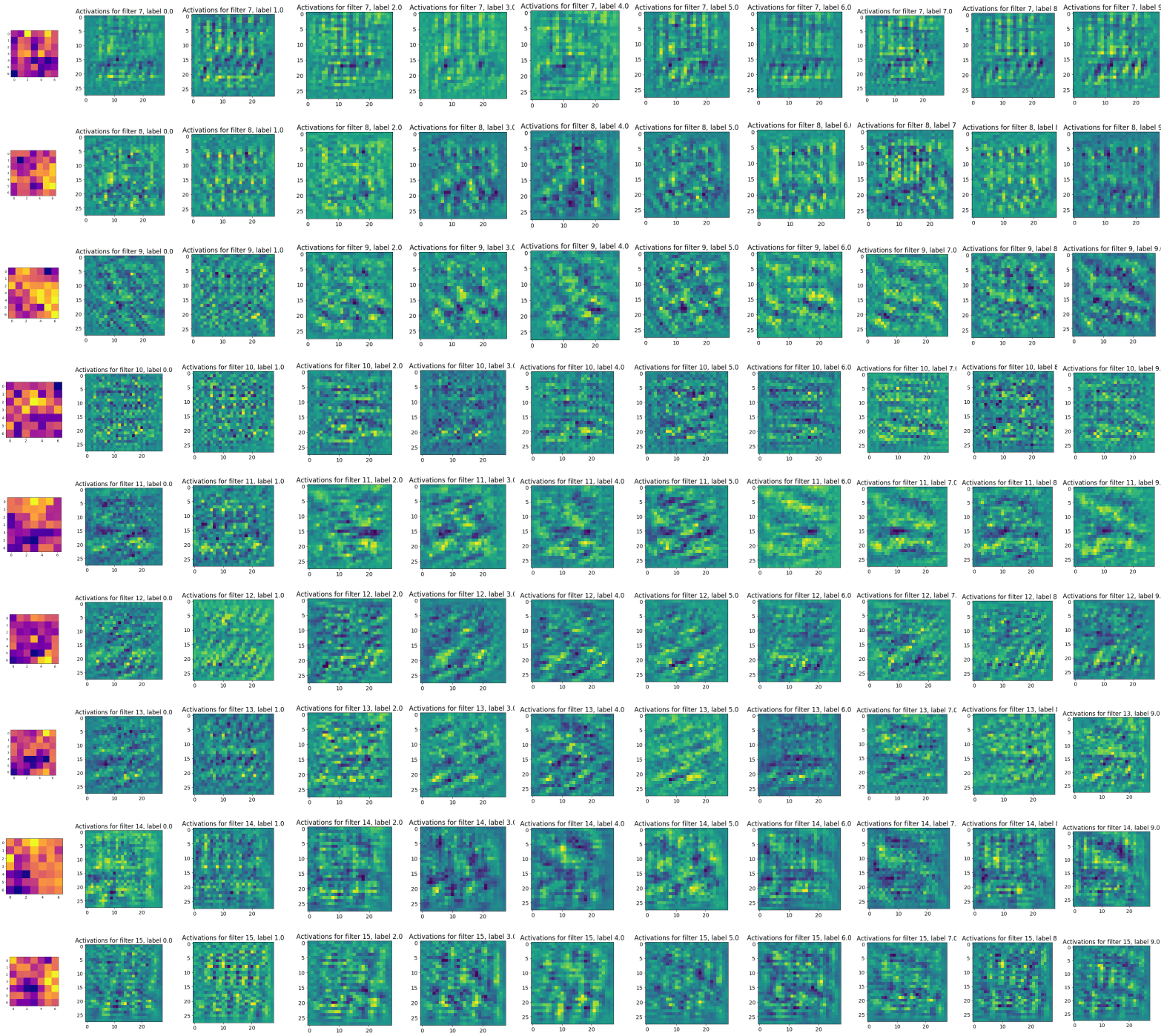
Confusion matrix for best performing model, TanH architecture B (test data).

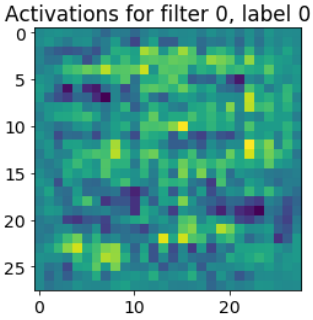
Part 4 Results

Deconvolution was performed on the best performing model from the last experiment in Part 3 by using the inverse toeplitz multiplied by the convolutional response field after feeding the first image of each class through the network. The main table below shows the filters and corresponding activation maps for the filters deconvolved with the input. Below, a Filter 1 for input zero is enlarged for a better view.

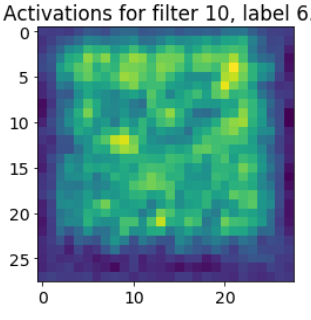
Input Activation Maps (Deconvolution)







Left shows enlarged filter from TanH, while left shows filter from poor-performing sigmoid.



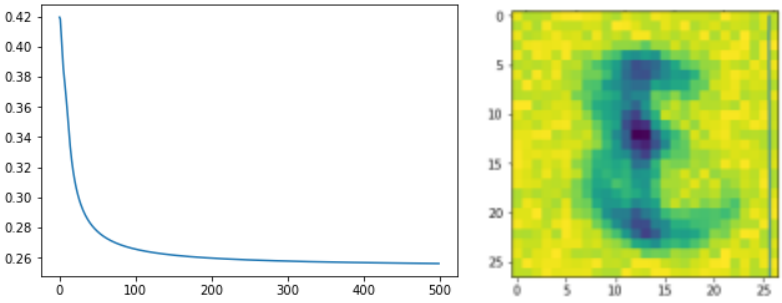
I did not anticipate these results, instead expecting a clear image of different aspects of each digit. However, this result does make sense, because the much smaller filter size, coupled with the larger number of filters, means that information may have been represented differently than with Part 2.

I did notice some interesting patterns in the deconvolved filters. The outer three to five pixels bordering the image tend to be much less sensitive than pixels towards the center of the image. This makes sense since these pixel areas don't contain discriminative information useful for classification. The included sigmoid filter used in the poorly-performing experiment parallels this result, with the bordering region being weighed negatively. This sigmoid network seems to have only learned which regions to pay attention to, not discriminative features within those regions.

There is also a relationship between the filter and resulting deconvolution operations. Filters that display a more focal characteristic (one or two isolated strong positive pixels), result in filters with strided isolated areas of high activation. Those filters with more of a gradient, however, appear to have regions of gradual intensity gradients which are less abrupt. I am not entirely sure why these filters didn't turn out as the canonical ones given in [2], it may simply be that more convolutional layers, data, and training time are necessary.

Lessons Learned

Some of the most valuable lessons learned during the project were learned the hard way–debugging code errors or unexpected training behavior (i.e. vanishing gradients). While solving these issues, I was forced to step through each computation and understand how various components were interacting. There were two issues which most clearly demonstrate this. After implementing Part 2, the network converged over the first several epochs, but then quickly leveled at a high training loss. Even more interestingly, the network appeared to be learning filters that made sense for the data:



Erroneous convergence plot for part 2 training data, and learned filter.

After debugging for several hours, I discovered that the output of the convolutional layer was quickly saturating at 1 (was using logistic activation function). This caused me to examine the induced local field to discover the bias update was being applied twice.

The second, more interesting problem arrived after implementing Part 3, when I discovered that the gradient being propagated backwards through the network was very weak (the vanishing gradient problem). Through solving this issue, I learned a hard lesson regarding the interplay between activation functions, initialization, and input normalization. I learned that in order for outputs to not saturate, the distribution of the induced local fields needs to fall within the sweet spot activation function. However, activations ended up being very large, causing sigmoid and tanh to saturate. I tried to solve this problem with several strategies: (a) alter the sigmoid and tanh slope to allow for a larger non-saturating range, (b) initialize biases according to a heuristic such that they would re-center activations around the mean, and (c) alter weight initialization to reduce saturation. None of these methods functioned very well, and while reading [1], I discovered that applying input normalization, initializing weights with zero mean and a variance proportional to the number of input nodes would solve the problem. This gave me a natural intuition regarding why ReLU functions well in many situations, and why batch normalization allows for quicker and more robust convergence.

I also learned about the process of translating equations to well-functioning code, and that writing efficient implementations becomes especially important. I challenged myself to come up with a method of using a direct matrix multiplication to apply correlation, and other matrix multiplication strategies for all aspects of forward and backward propagation. Though this likely took longer to implement, I am proud of the resulting code and learned more about matrix operations in numpy along the way. However, this did give me a strong sense of how easily things can go wrong, especially when multiplying 3D tensors, as something as subtle as an incorrect transpose axis can throw off the entire computation.

More importantly, the coding process taught me that non-erroring code isn't necessarily correct code; implementing forward and backpropagation require careful attention to detail in order to produce correct results. What is deceiving about this is that several times, partially-correct solutions still converged, albeit less effectively. This underscores the importance of gradient checking and other testing during the implementations. Finally, the coding process gave me an acute appreciation of the work that is required to provide a fully-featured functioning implementation of deep learning library such as TensorFlow or PyTorch.

I most enjoyed coming up with different ways of visualizing what is occurring in the network during training (the deconvolution and delta magnitudes, for example). I would like to pursue these areas further in a research context, as there are likely better ways of understanding the

information representation learned by the network during training. I would like to also explore this in the context of a larger network with more layers, pooling, batch norm, and other modern advancements that speed up learning.

References:

[1] LeCun, Yann A., et al. "Efficient backprop." Neural networks: Tricks of the trade. Springer, Berlin, Heidelberg, 2012. 9-48.

[2]Zeiler, Matthew D., and Rob Fergus. "Visualizing and understanding convolutional networks." European conference on computer vision. Springer, Cham, 2014.