EECS 8770

Spring 2019 - Project 1

Max Highsmith
03/12/2019

Introduction 1

Technical Description 2

Code Description 6

Experiments 8

Lessons Learned 28

Technical Description

Mathematical derivations.

Notational Note

When denoting vector or matrix variables we use a capital letter to indicate the entire structure and an indexed lowercase letter to indicate a specific component value.
As illustration

$$X^{(0)} = \begin{bmatrix} x_{11}^{(0)} & \dots & x_{1n}^{(0)} & \dots & x_{1N}^{(0)} \\ \dots & \dots & \dots & \dots & \dots \\ x_{m1}^{(0)} & \dots & x_{mn}^{(0)} & \dots & x_{mN}^{(0)} \\ \dots & \dots & \dots & \dots & \dots \\ x_{M1}^{(0)} & \dots & x_{Mn}^{(0)} & \dots & x_{MN}^{(0)} \end{bmatrix}$$

When denoting indices we use a Mathcal formatted letter to represent the set of indices, the lowercase letter to indicate an element from the set of indices and the capital letter to indicate the largest value in the set of indices. As illustration $\mathcal{J} = \{1, ..., j, ..., J\}$.

DataSet:
This Paper uses deep learning to train a Neural Network in Image classification with a subset of the mnist dataset. We treat each image entering the network as a (C*D) array denoting the height and width of the image and label the tth sample as $X^0(t).$ For the MNIST dataset C=D=28.

We then pass the image through a convolutional layer with L filters, each of size $M * N$.
For experiment 2 $L = 2$ and in experiment 3 $L = 16$. The sample is convolved with each filter, producing a receptive field $Z(t)$ via the convolution equation

$$Z_{l,m,n}(t) = \sum_{\mathbb{J}} \sum_{\mathbb{I}} a_{i,j} * x_{m+i,n+j}(t)$$

We know via convolutional arithmetic that the dimension of this new layers output is determined by the equation

$$M = \frac{C-I+1+(2*padding)}{stride}$$

$$N = \frac{D-J+1+(2*padding)}{stride}$$

For this project we only consider networks of stride=1 and padding=0
After the sample has passed through the first convolutional layer we apply a nonlinearity, also known as an activation function to all components of the output M*N*L Matrix. During this project experiments are run on a variety of different non-linearities which will be described after the network overview, so for now we denote a the first non-linearity as
ϕ^0 and label the output as

$$X^1(t) = \phi^0(Z^0(t))$$

Perceptron

To pass the sample onto the perceptron we collapse the indices $\mathbb{L}, \mathbb{I}, \mathbb{J}$ into a single input dimension $\mathbb{K}$ where $K = L * M * N$ and
$k = l * MN + m * N + n$. A forward pass through a perceptron with \$R\$ neurons is evaluated by the following
$Z^1(t) = W^1 * X^1(t)$ or componentwise $Z^1_r(t) = \sum_{\mathbb{K}} w_{r,k} * x^1_k(t) + b^1$

We then apply another nonlinear activation function ϕ^2 to yield the networks label-encoded prediction
$x^2_r(t) = \phi^2(x^1)$

Loss and Accuracy
The next step is to compare the Network prediction with label assigned to our sample. We denote the generic loss function E and consider the standard squared error function

$$E(X^{(2)}(t), P(t)) = \frac{1}{2}(X^{(2)}(t) - P(t))^2$$

The result from this loss will be used in training the Network, however it is also necessary for the task of classification to actually make a decision on how the current sample is being classified. To this the network uses a simple Decision function by predicting the label corresponding to the component of $X^{(2)}_r(t)$ with the highest value.

BackPropagation

Now to actually go about training the network a process called back propagation is used. The basic idea behind backpropagation being to evaluate the partial of our loss function with respect to each tunable parameter in the network and nudge the weights in the direction of the lowest gradient. In this case there are 4 categories of tunable parameters namely the perceptron weights and bias and the Convolutional Filter weights and bias

$\frac{\partial E}{\partial w_k^{(1)}}$, $\frac{\partial E}{\partial b^{(1)}}$ $\frac{\partial E}{\partial a_{l,i,j}}$, $\frac{\partial E}{\partial b_l^0}$

In order to determine these partials we will use the chain rule starting from the output layer. Taking the partial of the Squared error function we get

$$\frac{\partial E}{partial X^{(2)}(t)} = X^{(2)}(t) - P(t)$$

.

We also evaluate the partial of our activation function to get

$$\frac{X^{(2)}(t)}{Z^{(2)}} = \phi(Z^{(2)})'$$

The gradients of ϕ are dependent on the selection of our activation function and are described in more detail in the section on Activation function.

The partial for the feedforward perceptron output with respect to the weight is

$$\frac{\partial Z_r^{(1)}(t)}{\partial w_{rk}^{(1)}} = x_k(t)$$

.

And the partial for the feedforward perceptron with respect to the bias is unaffected by input so

$$\frac{\partial Z^1(1)_r(t)}{\partial b^{(1)}} = 1$$

.

Applying the chain rule we have our desired deltas which will be applied to the weights in the learning step.

$$\delta w_{r,k}(t) = \frac{\partial E}{\partial w_{r,k}^{(1)}(t)} = \frac{\partial E}{\partial z^{(2)}(t)} \frac{\partial z_k^{(2)}(t)}{\partial z^{(2)}(t)_k} \frac{\partial z^{(2)}}{\partial w_{r,k}^{(1)}}$$

$$\delta b^{(1)} = \frac{\partial E}{\partial b^{(1)}(t)} = \frac{\partial E}{\partial z^{(2)}(t)} \frac{\partial z_k^{(2)}(t)}{\partial z^{(2)}(t)_k} \frac{\partial z^{(2)}}{\partial b^{(1)}}$$

Convolutional Layer

We also need to evaluate the partials for the convolution parameter

$\frac{\partial E}{\partial a_{l,i,j}}$ and $\frac{\partial E}{\partial b_l^{(0)}}$

Evaluating the partials of the convolutional layer requires the error for each $x_r^{(1)}$ to be passed backwards. Through the chain rule this error is

$$\frac{\partial E}{\partial X^{(1)}(t)} = \frac{\partial E}{\partial X^{(2)}(t)} \frac{\partial x_k^{(2)}(t)}{\partial z^{(2)}(t)_k} \frac{\partial z^{(2)}(t)}{\partial X^{(1)}(t)}.$$

Now for each $a_{l,i,j}$ in the convolutional filters we back propagate the error $\frac{\partial E}{\partial x_k^{(1)}(t)}$ for every $x_k^{(1)}(t)$ which is reached by $a_{l,i,j}$ during the convolution operation. Because every x_k is generated by a pass through a shared weight filter this means we must back propagate every the error from neuron x_k such that k corresponds to the same l.

So we let $\delta_x^{(1)}{}_{l,i,j} = \sum_l x_{l,i,j}^{(1)}$

And have our formula for the convolutional partials:

$$\frac{\partial E}{\partial a_{l,i,j}} = \frac{Z^{(0)}}{X^{(0)}_{i:C-(I-i)+1,j:D-(J-j)+1}} \frac{X^{(1)}_{l_{l,:,:}}}{Z^{(0)}_{l,:,:}} \delta X_l^{(1)}$$

$$\frac{\partial E}{\partial b_l} = \frac{X^{(1)}_{l,:,:}}{Z^{(0)}}{}_{l,:,:} \delta X^{(1)}_{l,:,:}$$

Finally, during the training step the network updates its trainable weights via the following equations

$$w_{r,k} := w_{r,k} - \alpha * \frac{\partial E}{\partial w_{r,k}}$$

$$b^{(1)} := b^{(1)} - \alpha * \frac{\partial E}{\partial b^{(1)}}$$

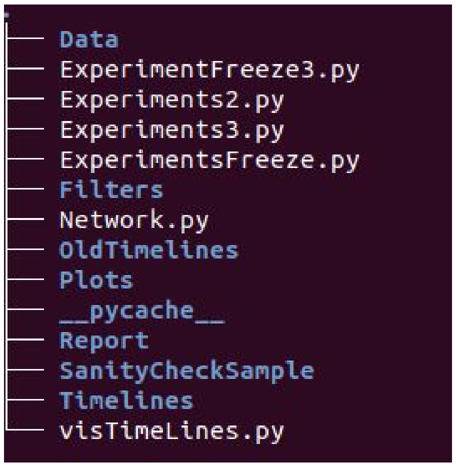
$$a_{l,i,j} := a_{l,i,j} - \alpha * \frac{\partial E}{\partial a_{l,i,j}}$$

$$b_l^{(0)} := b_l^{(0)} - \alpha * \frac{\partial E}{\partial b_l^{(0)}}$$

Where α here refers to the learning rate. This nudges each parameter in the direction where the gradient of loss is smallest.

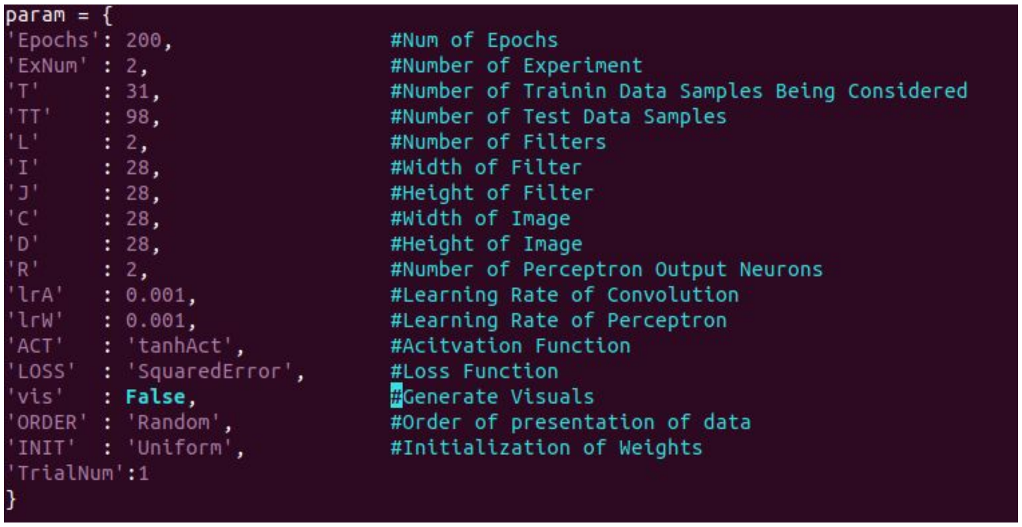
Code Description

The directory for my submission contains the following:



```
.
├── Data
├── ExperimentFreeze3.py
├── Experiments2.py
├── Experiments3.py
├── ExperimentsFreeze.py
├── Filters
├── Network.py
├── OldTimelines
├── Plots
├── __pycache__
├── Report
├── SanityCheckSample
├── Timelines
└── visTimeLines.py
```

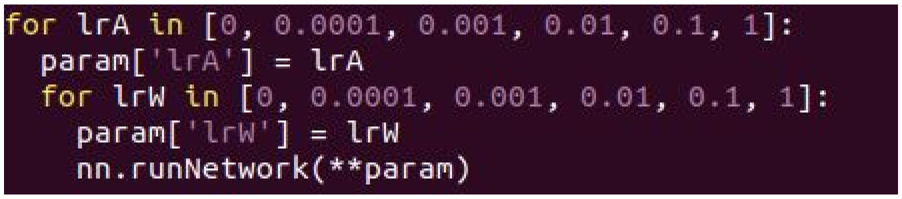
Data: holds the Mnist Data Provided
Filters: Images of the filter weights
Timelines: holds numpy arrays tracking the loss and accuracy of test and training evaluation
Plots: Holds all the graph visuals show in this paper
Network.py:
Experiments 2 and 3 are coded up using the same base network function which accepts a dictionary of arguments as a parameter. These arguments make up the Hyper Parameters which distinguish the network as well as a few auxiliary commands.
Here is a list of all the available parameters for the network. For the most part I tried to make the variable names match the letters used in the mathematical description.



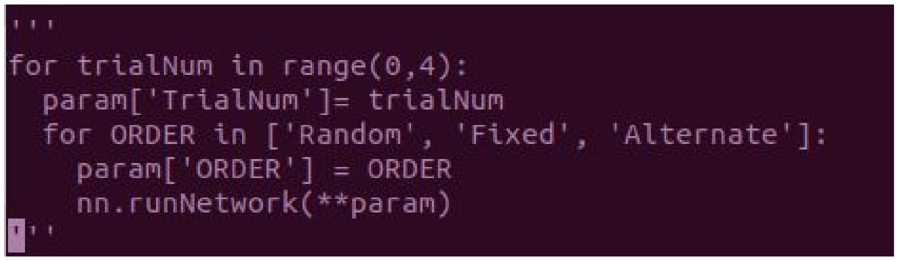
```
param = {
'Epochs': 200,                #Num of Epochs
'ExNum' : 2,                  #Number of Experiment
'T'     : 31,                 #Number of Trainin Data Samples Being Considered
'TT'    : 98,                 #Number of Test Data Samples
'L'     : 2,                  #Number of Filters
'I'     : 28,                 #Width of Filter
'J'     : 28,                 #Height of Filter
'C'     : 28,                 #Width of Image
'D'     : 28,                 #Height of Image
'R'     : 2,                  #Number of Perceptron Output Neurons
'lrA'   : 0.001,              #Learning Rate of Convolution
'lrW'   : 0.001,              #Learning Rate of Perceptron
'ACT'   : 'tanhAct',          #Acitvation Function
'LOSS'  : 'SquaredError',     #Loss Function
'vis'   : False,              #Generate Visuals
'ORDER' : 'Random',           #Order of presentation of data
'INIT'  : 'Uniform',          #Initialization of Weights
'TrialNum':1
}
```

Within Network.py I define the module activation functions, gather the dataset then iterate through training data forward pass, training data backward pass and test data forward pass.

Experiments.py:
For example in Experiments.py This line of code iterates through the different learning rate parameters considered and then runs the network for each



```
for lrA in [0, 0.0001, 0.001, 0.01, 0.1, 1]:
  param['lrA'] = lrA
  for lrW in [0, 0.0001, 0.001, 0.01, 0.1, 1]:
    param['lrW'] = lrW
    nn.runNetwork(**param)
```



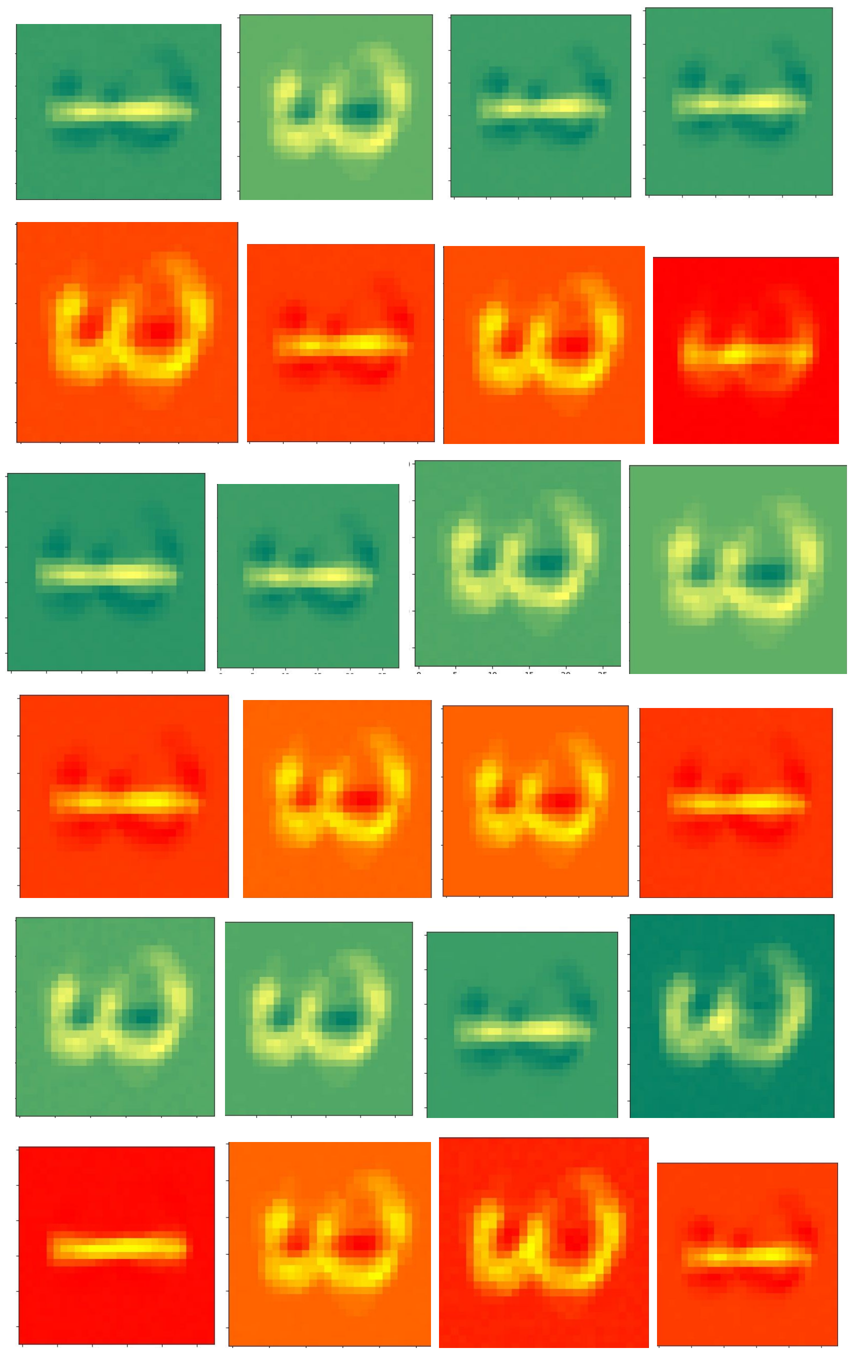
```
'''
for trialNum in range(0,4):
  param['TrialNum']= trialNum
  for ORDER in ['Random', 'Fixed', 'Alternate']:
    param['ORDER'] = ORDER
    nn.runNetwork(**param)
'''
```

visTimeLines.py: generates the visuals with a GenerateGraph() function which gathers the Timeline Data and graphs according to the variable specified with the following inputs

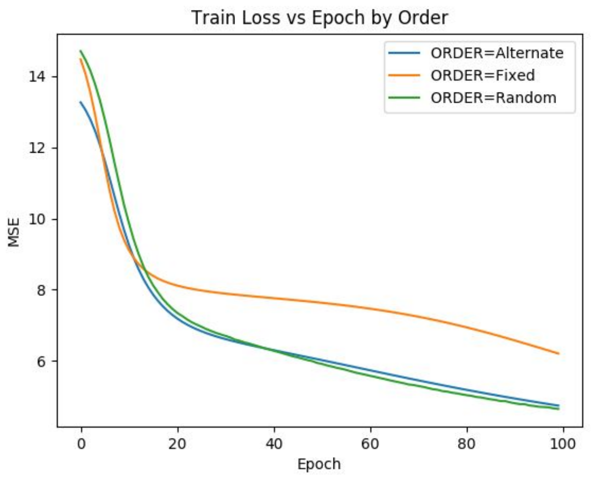
GraphName	= "title of Graph"
Param_Variation_Keys	= "Array of Hyperparameters to Compare"
Param_Variaion_Values	= "Nested Array of Hyperparameter values
**baseParam	= the dictionary defining all Hyperparameters not being varies in graph

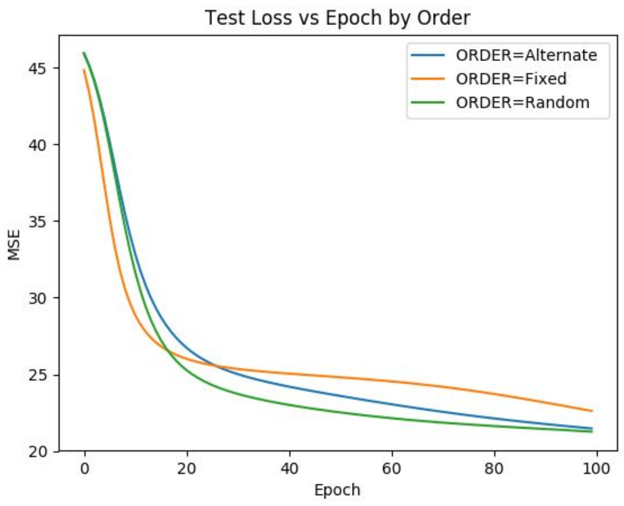
Experiments:

Ordering Experiments:

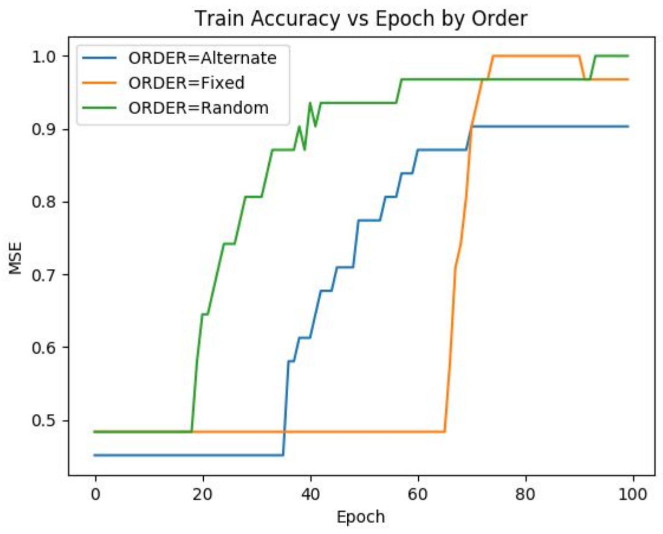


The Filters for the Ordering Experiments:Fixed, Alternate, Random





In the ordering experiments the fixed ordering appears to train the slowest and performs the worst on the training loss, but ultimately obtains the best test accuracy and comparable test loss.



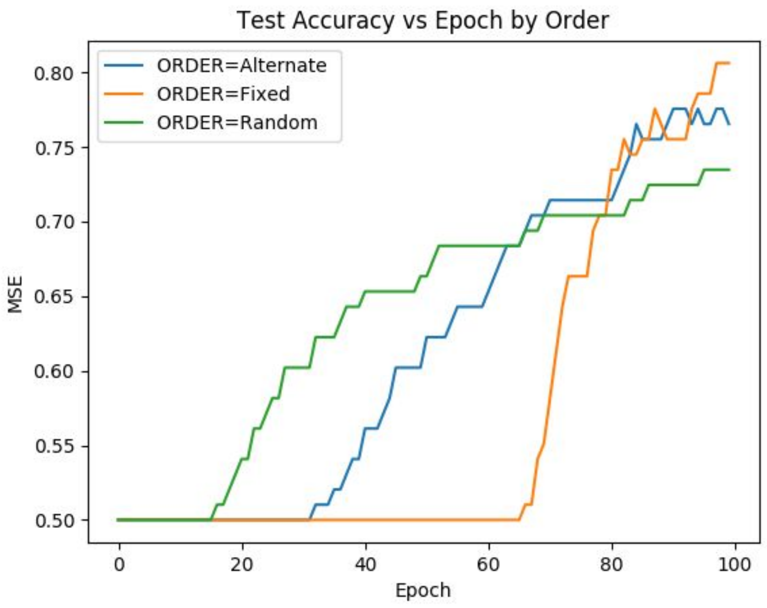


Figure:2 100 epochs at learning rate 0.001 with hyperbolic tangent activation function.

Activation Function Experiments:

In this project I consider three different activation functions, The sigmoid, the hyperbolic tangent and the rectified linear unit. I also run experiments using no activation function as a benchmark The definitions for each activation are as follows:

Hyperbolic Tangent:

$$x = \phi_{th}(z) = \frac{e^{(2z)} - 1}{e^{(2z)} + 1} \qquad \frac{\partial \phi_{th}}{\partial z} = 1 - \phi_t(z)$$

Sigmoid:

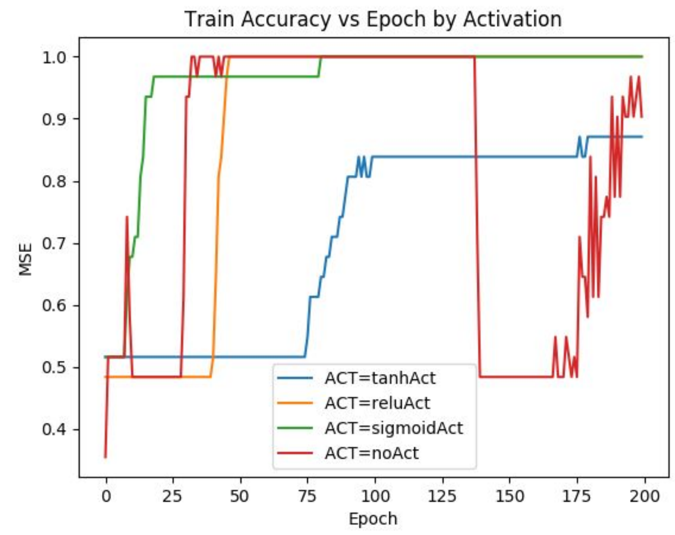
$$x = \phi s(z) = \frac{1}{1 + e^{-z}} \qquad \frac{\partial \phi_s}{\partial z} = \phi_s(z)(1 - \phi_s(z))$$

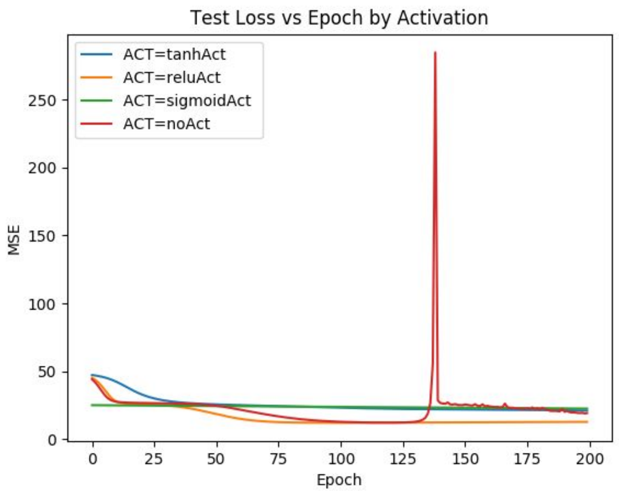
Relu:

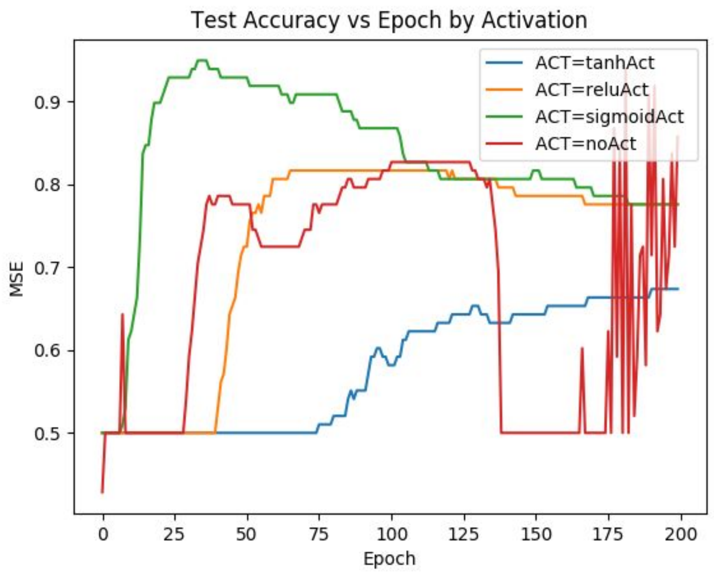
$$x = \phi_r(z) = max(0, x) \quad \frac{\partial \phi_r}{\partial z} = \begin{cases} 0 & x \leq 0 \\ 1 & 0 \leq x \end{cases}$$

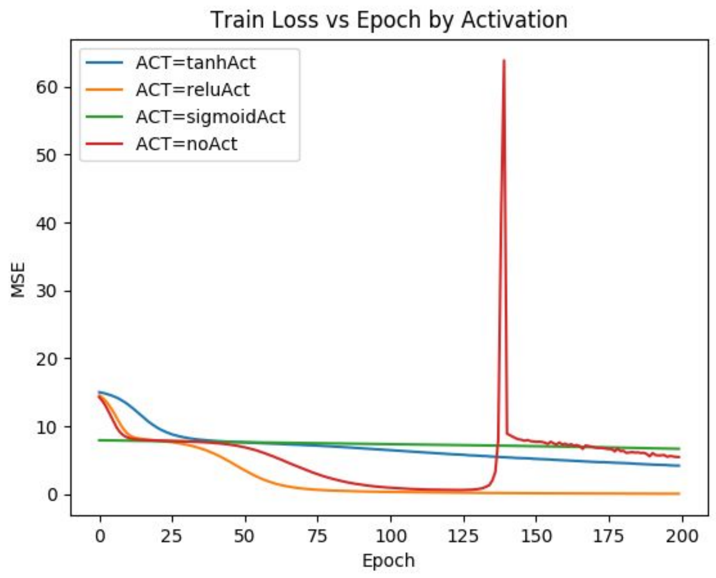
None

$$x = z \ \frac{\partial \phi_n}{\partial z} = 0$$

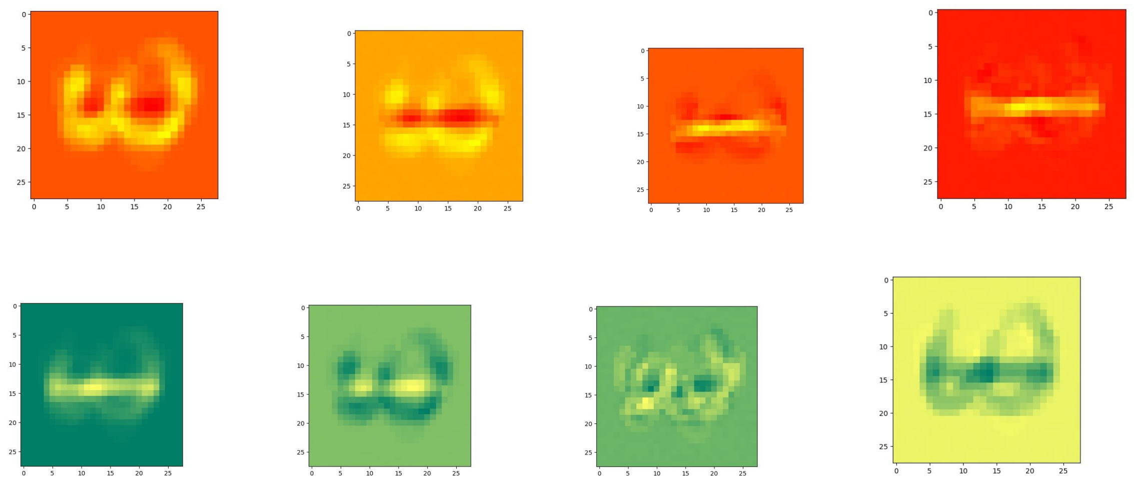

Train Accuracy vs Epoch by Activation
MSE
Epoch
ACT=tanhAct
ACT=reluAct
ACT=sigmoidAct
ACT=noAct


Test Loss vs Epoch by Activation
MSE
Epoch
ACT=tanhAct
ACT=reluAct
ACT=sigmoidAct
ACT=noAct

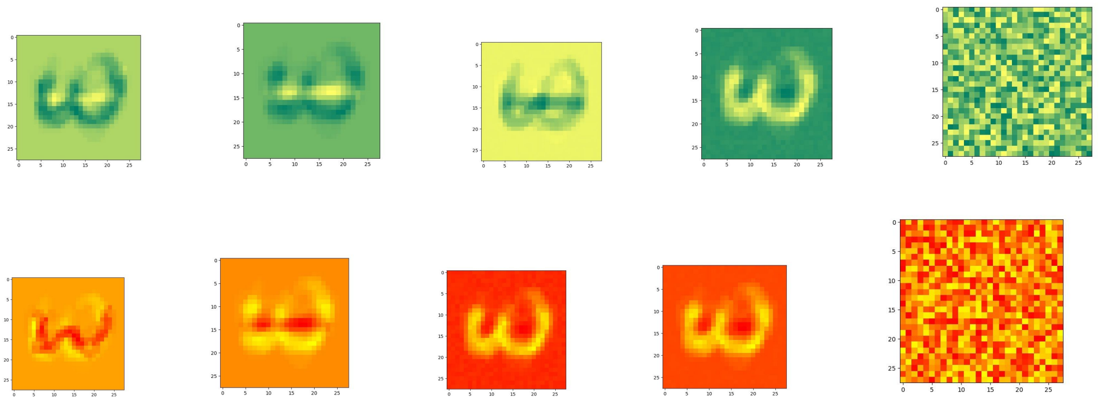




If no activation function is applied then the Network devolves to a sequence of pure matrix multiplications. This is particularly interesting because matrix multiplication is closed operation so without the activation function imposing nonlinearity, the entire network could technically be represented as a single matrix. This is consistent with the lower representation capacity observed by the relu activation in comparison to the other activation functions. In these experiments the sigmoid activation seemed to converge the quickest, though relu ultimately reached comparable accuracy in the test dataset.



Filters: Left to right Tanh, Sigmoid, Relu, None

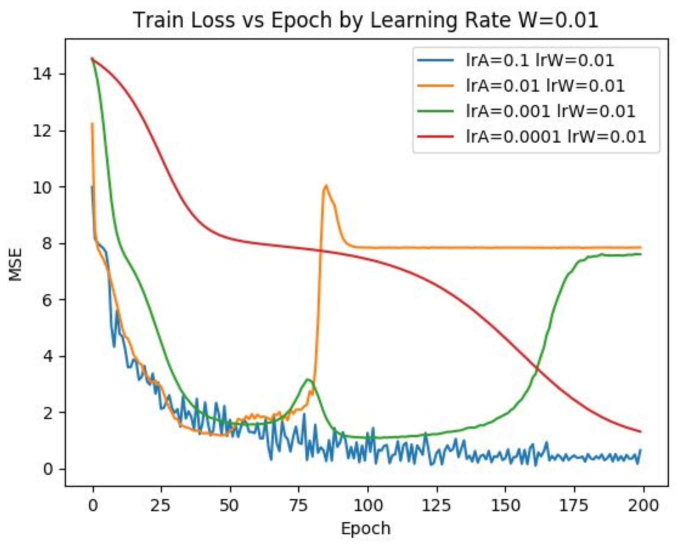


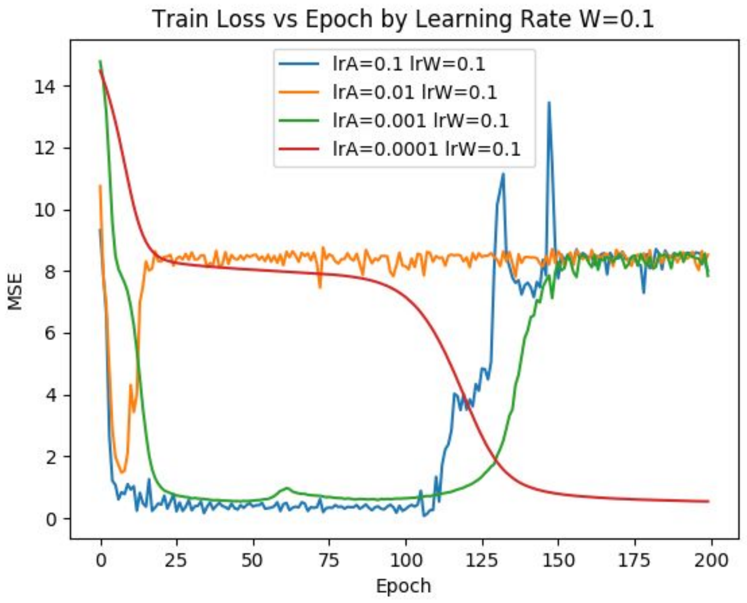
Filters Left to right lr 0.1, 0.01, 0.001, 0.0001

Learning Rate Experiments:

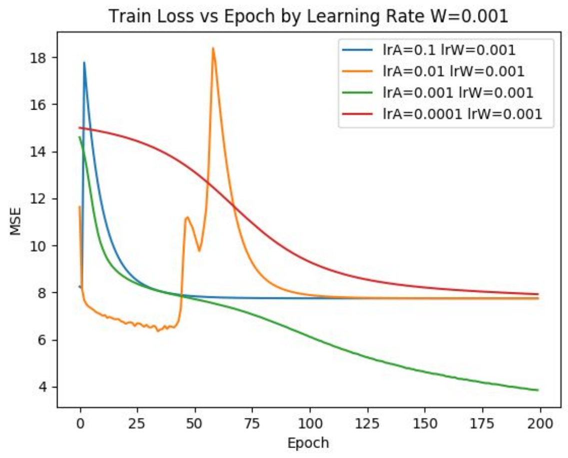
Different Learning Rates for Perceptron and Convolutional Layer:

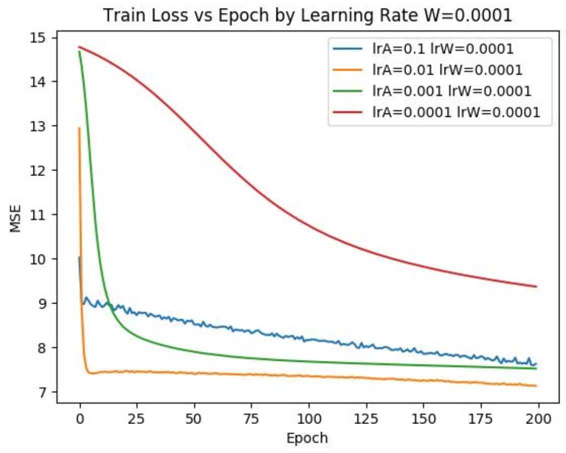
I observed the effects of different learning rates on the network training, varying the learning rate for the perceptron and Convolutional filters. The graphs indicate that the most effective learning rates seem to occur when the perceptron has a fairly quick learning rate and the convolutional layers learning rate is slower.

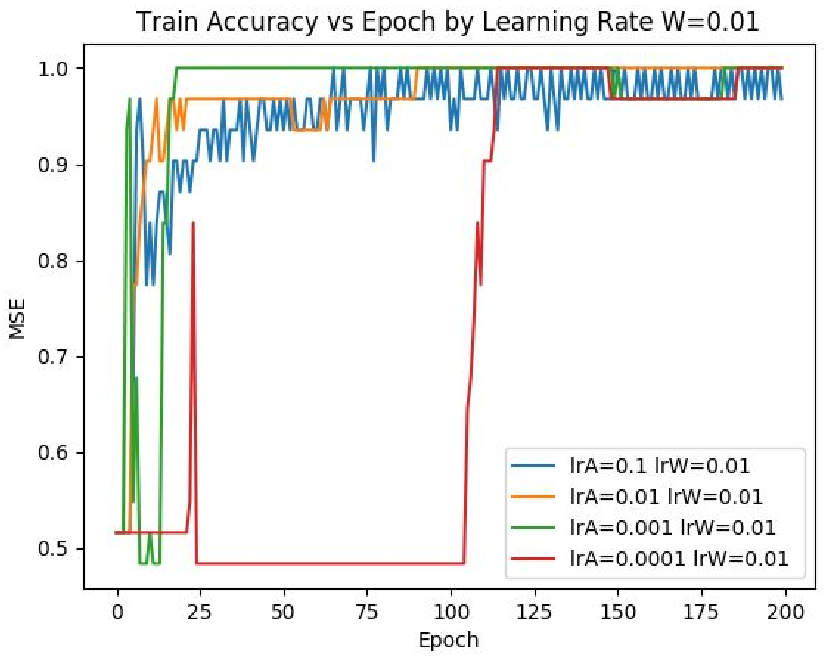


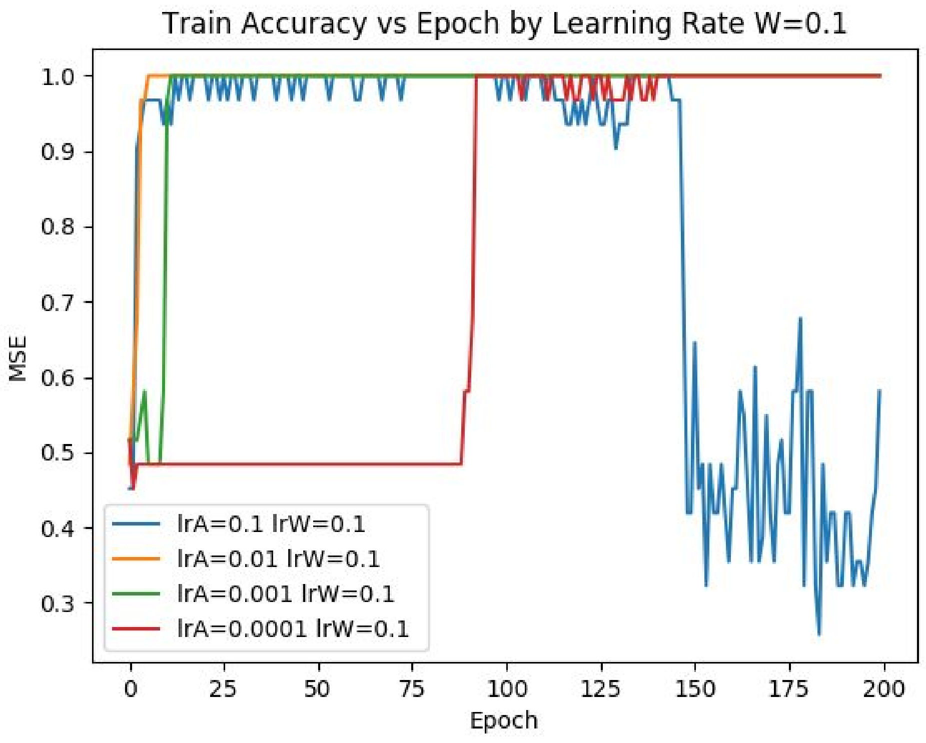


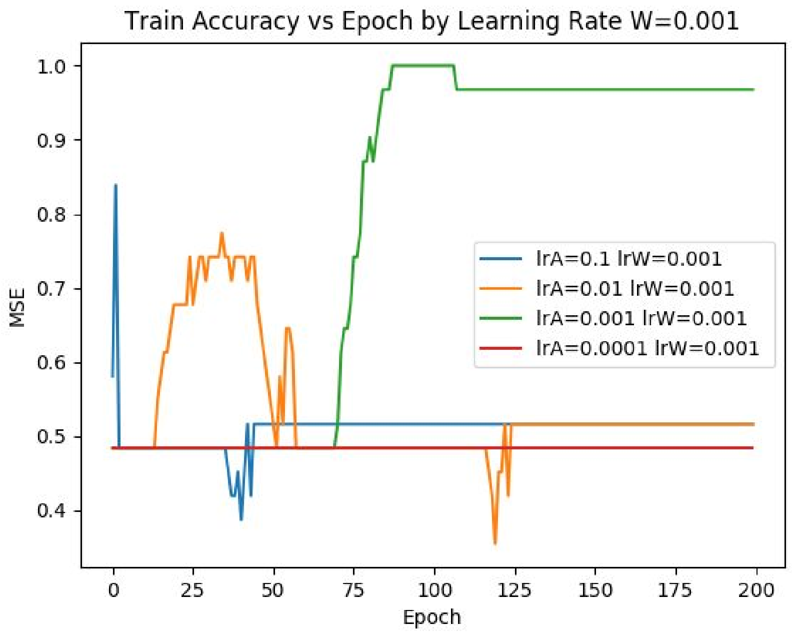
Large learning rates consistently converged faster on the training data, but often devolved to a sporadic behavior after sufficient epochs had passed. The slower learning rates generally converged to the lowers training and test loss.

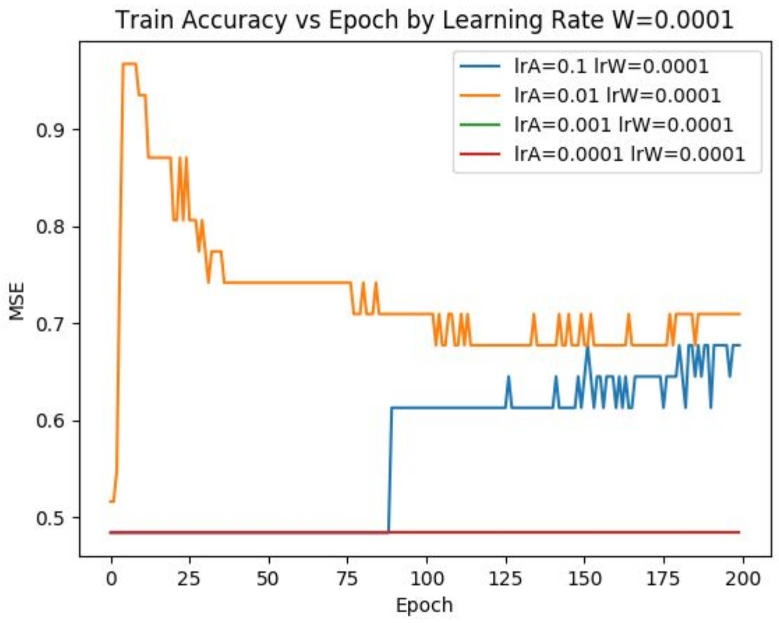

Train Loss vs Epoch by Learning Rate W=0.001
lrA=0.1 lrW=0.001
lrA=0.01 lrW=0.001
lrA=0.001 lrW=0.001
lrA=0.0001 lrW=0.001
MSE
Epoch

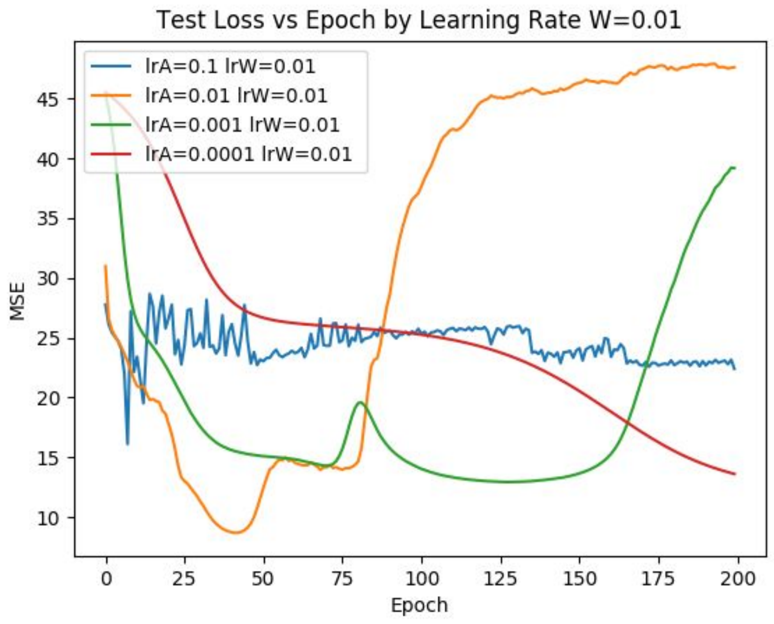

Train Loss vs Epoch by Learning Rate W=0.0001
lrA=0.1 lrW=0.0001
lrA=0.01 lrW=0.0001
lrA=0.001 lrW=0.0001
lrA=0.0001 lrW=0.0001
MSE
Epoch

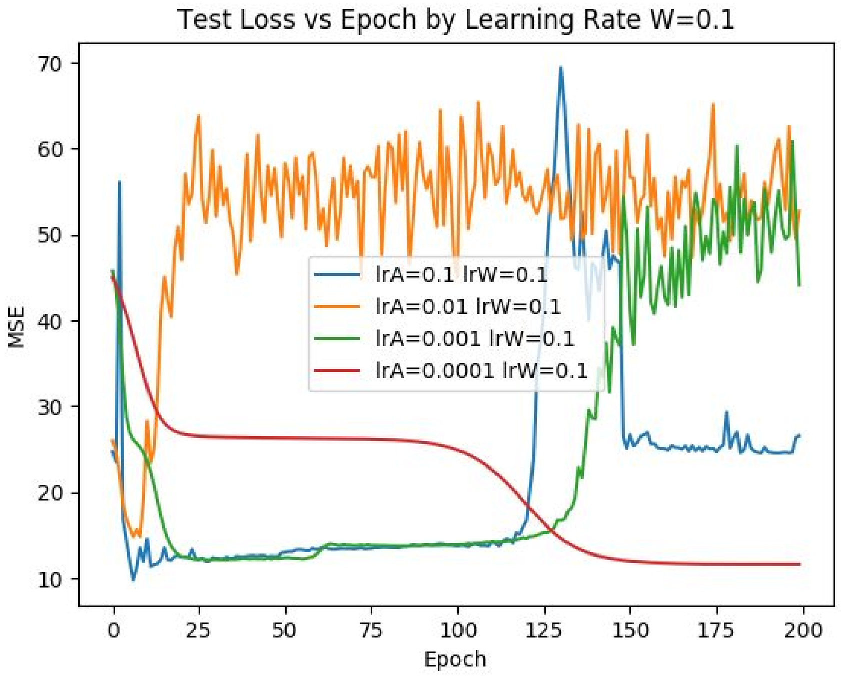

Train Accuracy vs Epoch by Learning Rate W=0.01
MSE
Epoch
lrA=0.1 lrW=0.01
lrA=0.01 lrW=0.01
lrA=0.001 lrW=0.01
lrA=0.0001 lrW=0.01

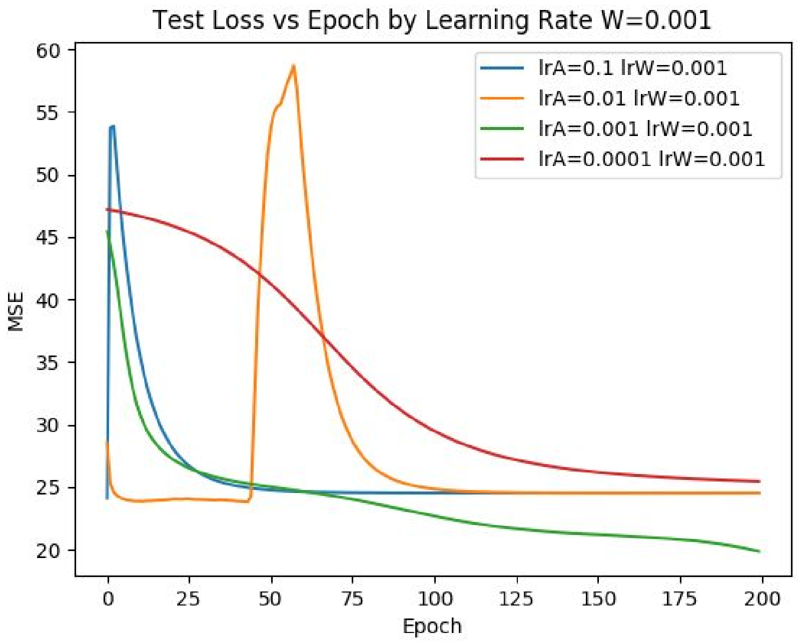

Train Accuracy vs Epoch by Learning Rate W=0.1
MSE
Epoch
lrA=0.1 lrW=0.1
lrA=0.01 lrW=0.1
lrA=0.001 lrW=0.1
lrA=0.0001 lrW=0.1

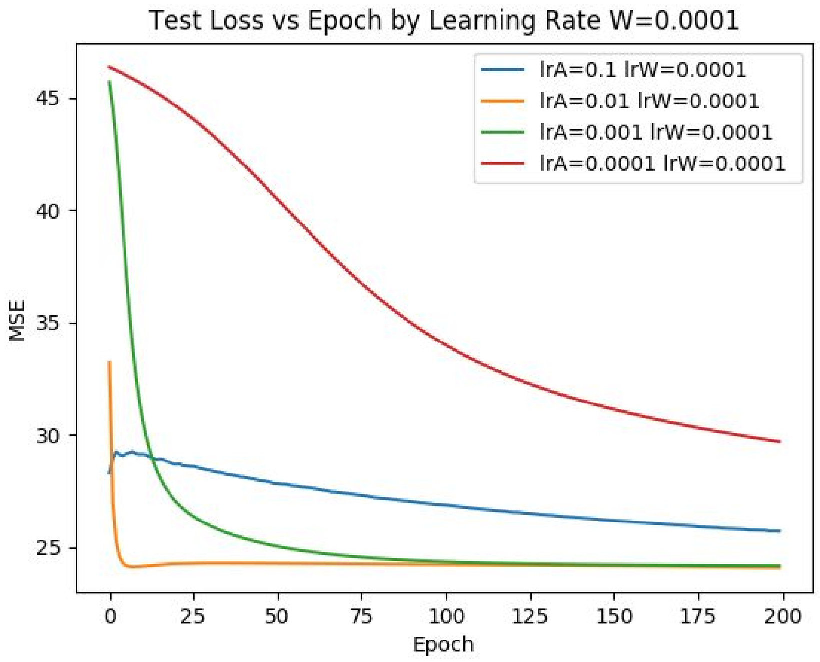

Train Accuracy vs Epoch by Learning Rate W=0.001
MSE
Epoch
lrA=0.1 lrW=0.001
lrA=0.01 lrW=0.001
lrA=0.001 lrW=0.001
lrA=0.0001 lrW=0.001

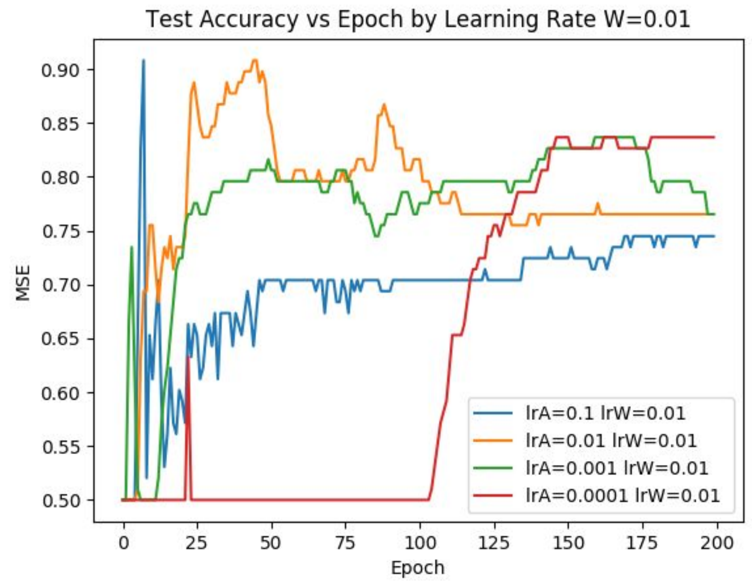

Train Accuracy vs Epoch by Learning Rate W=0.0001
MSE
Epoch
lrA=0.1 lrW=0.0001
lrA=0.01 lrW=0.0001
lrA=0.001 lrW=0.0001
lrA=0.0001 lrW=0.0001

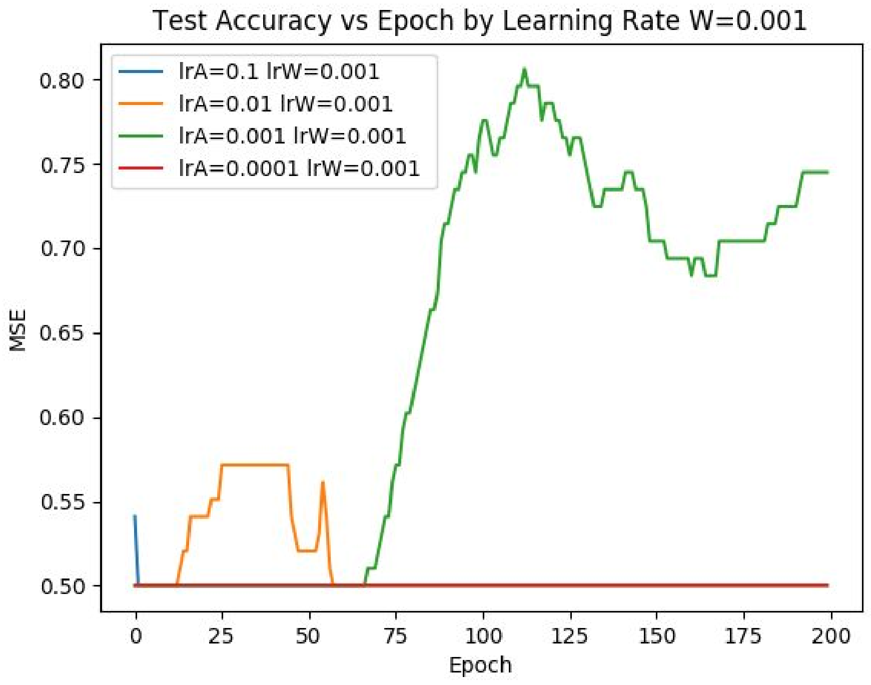

Test Loss vs Epoch by Learning Rate W=0.01
lrA=0.1 lrW=0.01
lrA=0.01 lrW=0.01
lrA=0.001 lrW=0.01
lrA=0.0001 lrW=0.01
MSE
Epoch
45
40
35
30
25
20
15
10
0
25
50
75
100
125
150
175
200

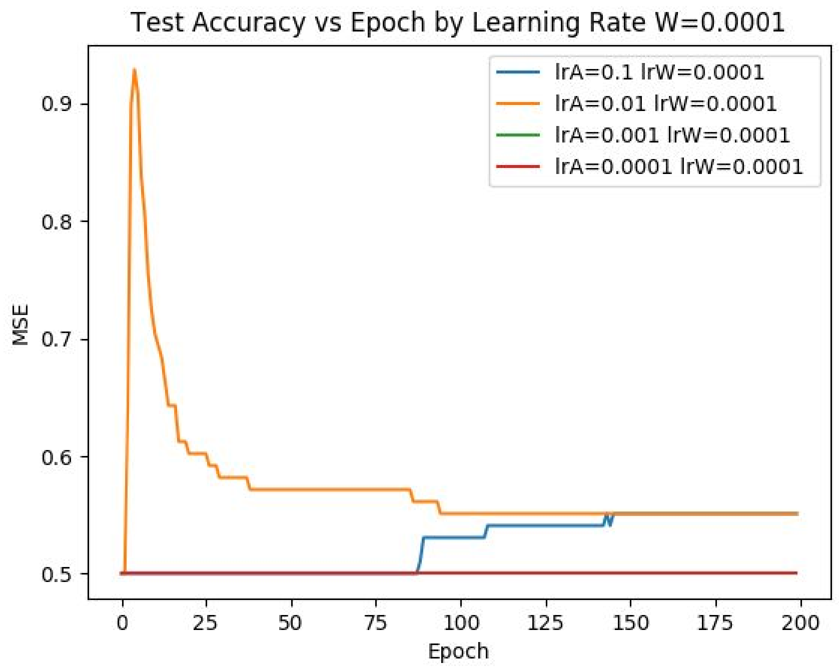

Test Loss vs Epoch by Learning Rate W=0.1
lrA=0.1 lrW=0.1
lrA=0.01 lrW=0.1
lrA=0.001 lrW=0.1
lrA=0.0001 lrW=0.1
MSE
Epoch
70
60
50
40
30
20
10
0
25
50
75
100
125
150
175
200

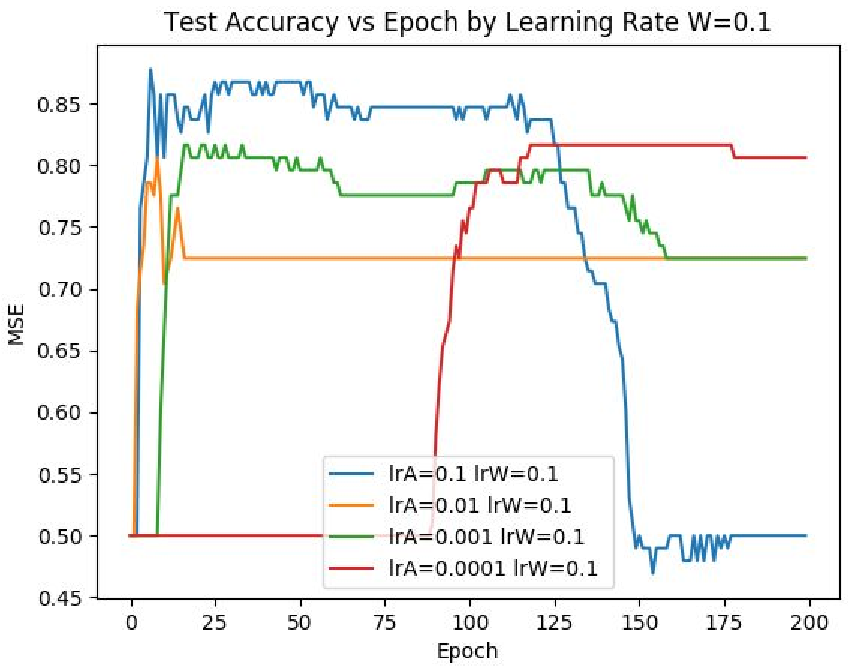

Test Loss vs Epoch by Learning Rate W=0.001
MSE
Epoch
lrA=0.1 lrW=0.001
lrA=0.01 lrW=0.001
lrA=0.001 lrW=0.001
lrA=0.0001 lrW=0.001
60
55
50
45
40
35
30
25
20
0
25
50
75
100
125
150
175
200


Test Loss vs Epoch by Learning Rate W=0.0001
MSE
Epoch
lrA=0.1 lrW=0.0001
lrA=0.01 lrW=0.0001
lrA=0.001 lrW=0.0001
lrA=0.0001 lrW=0.0001
45
40
35
30
25
0
25
50
75
100
125
150
175
200


Test Accuracy vs Epoch by Learning Rate W=0.01
MSE
Epoch
lrA=0.1 lrW=0.01
lrA=0.01 lrW=0.01
lrA=0.001 lrW=0.01
lrA=0.0001 lrW=0.01

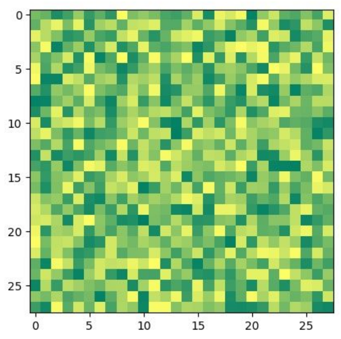

Test Accuracy vs Epoch by Learning Rate W=0.001
MSE
Epoch
lrA=0.1 lrW=0.001
lrA=0.01 lrW=0.001
lrA=0.001 lrW=0.001
lrA=0.0001 lrW=0.001

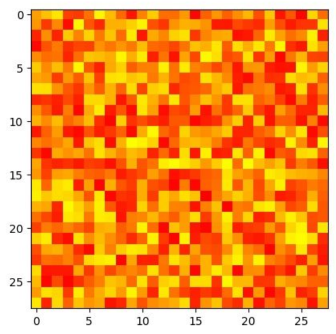

Test Accuracy vs Epoch by Learning Rate W=0.0001
lrA=0.1 lrW=0.0001
lrA=0.01 lrW=0.0001
lrA=0.001 lrW=0.0001
lrA=0.0001 lrW=0.0001
MSE
0.5
0.6
0.7
0.8
0.9
0
25
50
75
100
125
150
175
200
Epoch

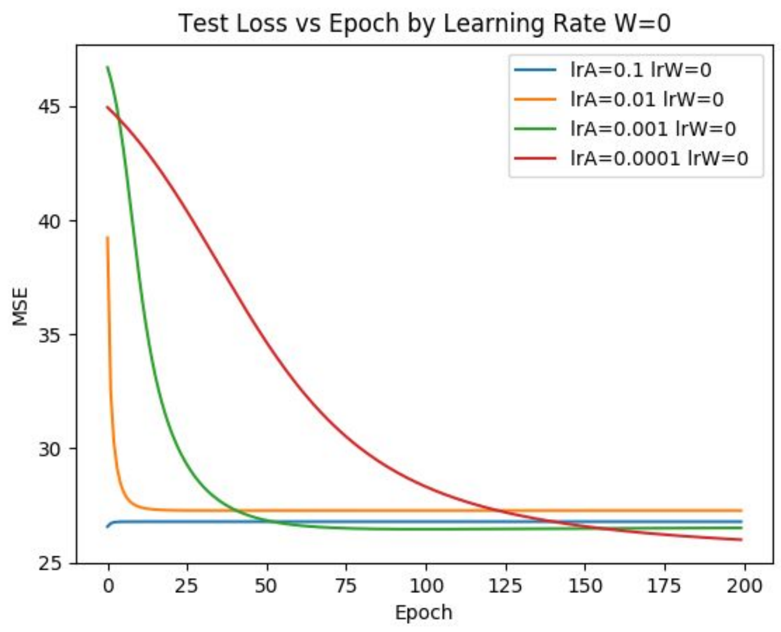

Test Accuracy vs Epoch by Learning Rate W=0.1
lrA=0.1 lrW=0.1
lrA=0.01 lrW=0.1
lrA=0.001 lrW=0.1
lrA=0.0001 lrW=0.1
MSE
0.45
0.50
0.55
0.60
0.65
0.70
0.75
0.80
0.85
0
25
50
75
100
125
150
175
200
Epoch

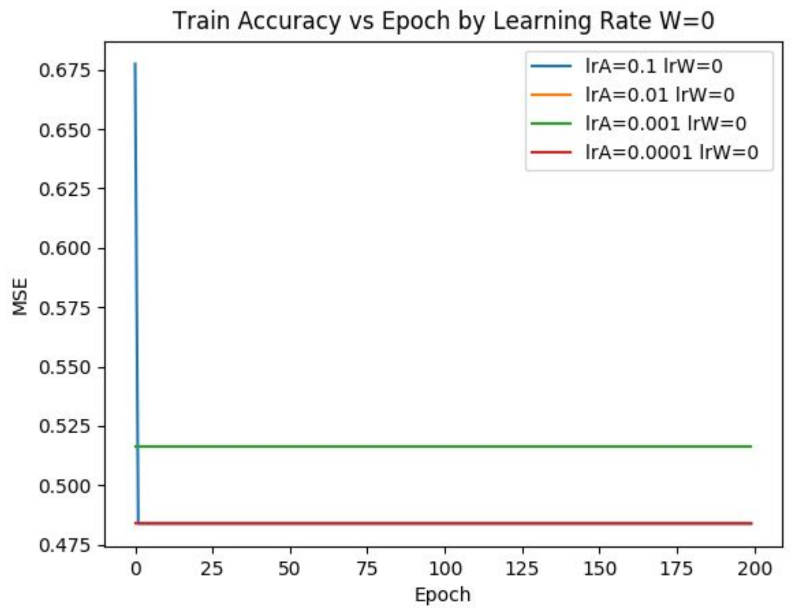
Frozen Learning Rates:
For the majority of the experiments considered, the network utilizes the power of both the convolutional and standard feed-forward perceptron for classifying the data set. I was curious what the effects of the overall network training would be if either the Convolutional Layer or Perceptron were given learning rates of zero, effectively fixing the effects of the Network at those particular values.

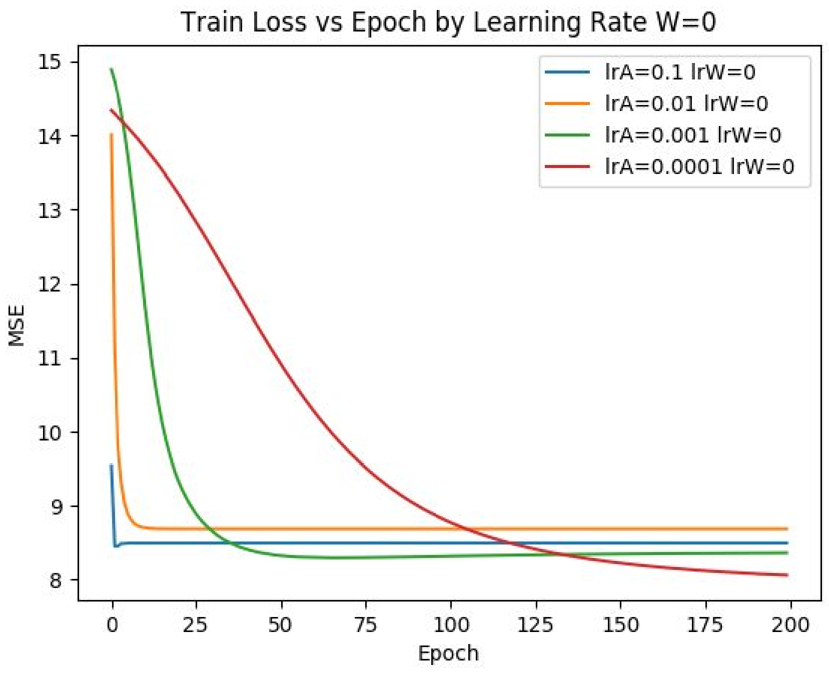
Freezing the Convolutional Layer:
As would be expected, in the experiments where the learning the weights of the convolutional layer are not updated so the kernel remains a block of random values indefinitely.

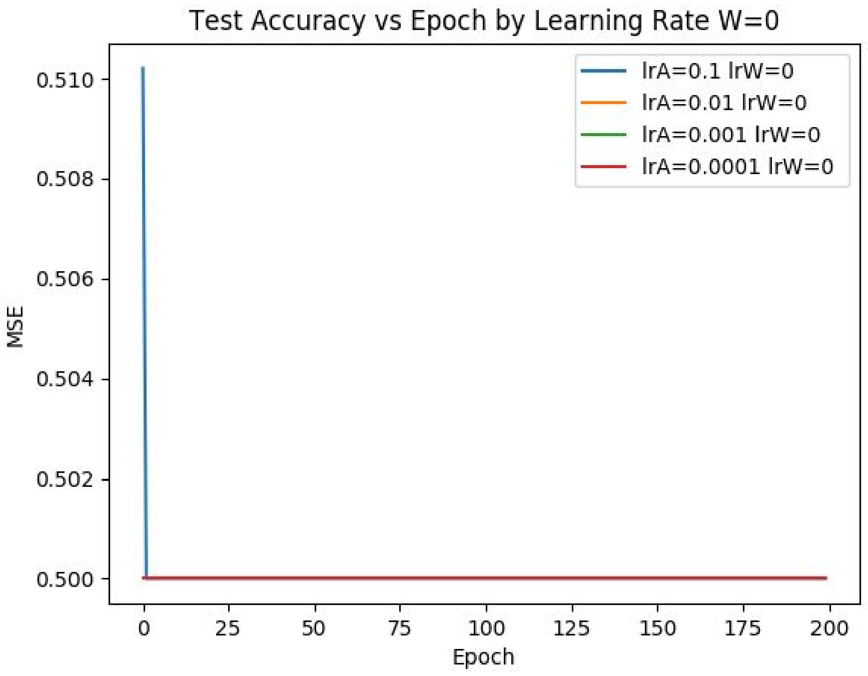





Test Loss vs Epoch by Learning Rate W=0
lrA=0.1 lrW=0
lrA=0.01 lrW=0
lrA=0.001 lrW=0
lrA=0.0001 lrW=0
MSE
Epoch
45
40
35
30
25
0
25
50
75
100
125
150
175
200


Train Accuracy vs Epoch by Learning Rate W=0
lrA=0.1 lrW=0
lrA=0.01 lrW=0
lrA=0.001 lrW=0
lrA=0.0001 lrW=0
MSE
Epoch
0.675
0.650
0.625
0.600
0.575
0.550
0.525
0.500
0.475
0
25
50
75
100
125
150
175
200





In the convolutional layer is now just a function adding a type of noise into the data samples. The graph shows that while the trainin loss still trains over time the general accuracy does not seem to improve or generalize to the test data very effectively.

Freezing the Perceptron with initial weights set as an Identity Matrix:

I noticed that in many previous experiments where the Convolutional Layer was being trained, the 2 convolutional kernel weights often looked very similar to one another and would not always take on features of both images of ‘3’ and ‘1’. I tried setting the perceptron weights to be an identity matrix mapping the outputs of the convolutional layers directly to the errors being considered in the Loss metric. This necessitated that the error back propagated to each convolutional filter only encapsulated the disparity along the corresponding component of the label encoding. My Hope was that this would cause the filters to learn the features corresponding to their respective components

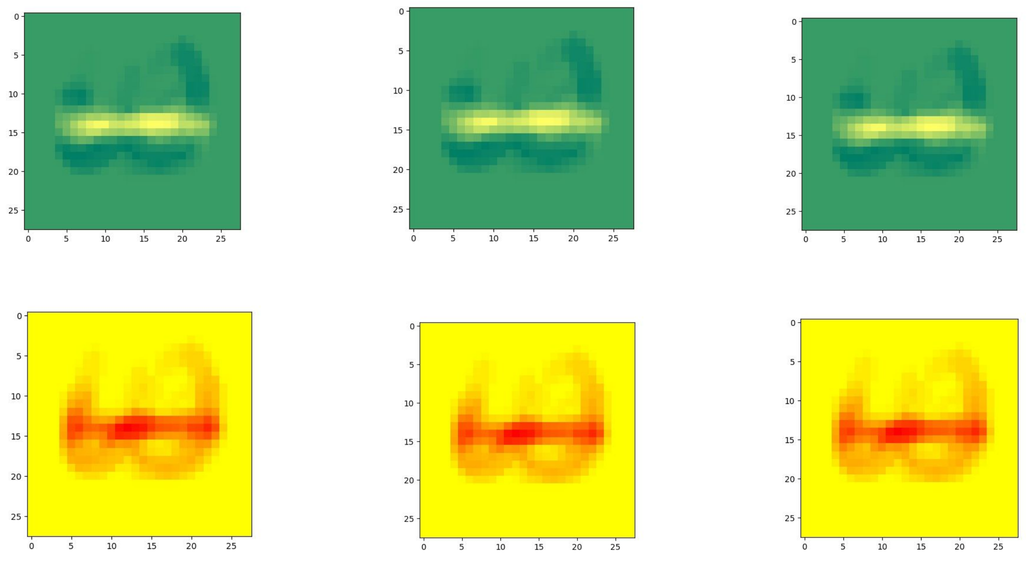


Figure: Trials of Frozen Perceptron weights

It wasn’t effective at its goal but I realized that because part 2 is a binary classification this is essentially just training two identical network with a 28*28 layer. I tried the same strategy using the part 3 data set but keeping 28*28 filters and using 10 output layers so that the perceptron was an identity matrix in $\mathcal{Z}_{10}$

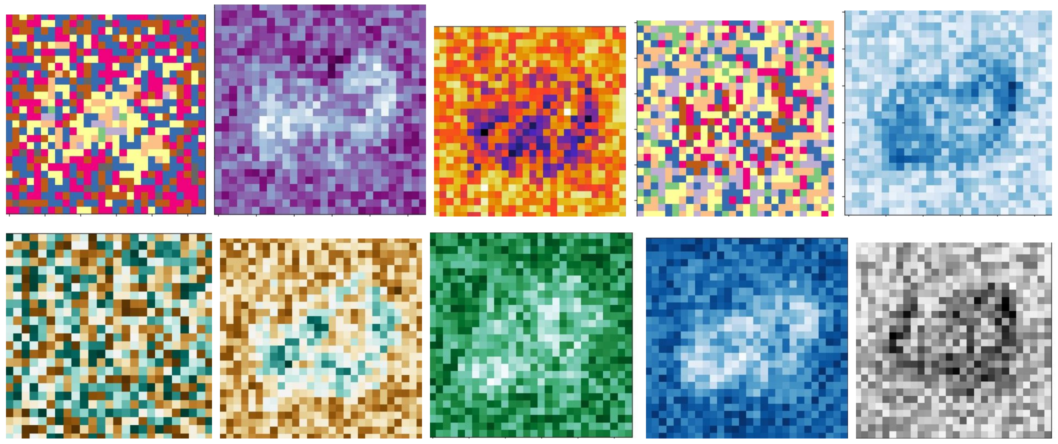


Figure:Frozen Perceptrons of Mnist Data

While this was not purely effective as many of the filters look basically just like noise, there are a few numbers which seem to start to take the shapes of their corresponding labels, namely 2, 3, 8 and 9.

What I Learned

I learned a variety of concepts in the course of this project, the most conspicuous of which being a renewed appreciation for the value of preexisting neural network libraries like tensorflow and pytorch, which are capable of handling this complex algorithmic work under the hood. What I find particularly impressive is how optimized those packages are in comparison the the hard coded 2 layer networks built in this experiment. With my current hardcoded convolution backprop algorithm the network takes forever to train when using convolution filters of size 7, but I have used considerably more complex networks in the past that converge quickly.

Work with this project also enhanced my appreciation of the importance of hyperparameter selection in training a network effectively. There were many times when debugging the code where the network would behave sporadically or freeze and I would trace through the progression of variables to find that the delta values had vanished to a matrix of zeros or that they were so large that they would overwhelm the information already contained within the weights that were being updated. I had heard about the vanish gradient problem in the past but did not realize that it could be an issue for a network this small. This gives appreciation for why more sophisticated gradient descent variants have utility such as having different learning rates for each parameter.

I realized that seemingly minor details such as the order in which data is presented to a network can make a consistent difference in the rate at which a Network converges or the ultimate accuracy that is ever obtained. Prior to this project the notion that a different activation function or learning rate would impact the network training seemed somewhat intuitive even if the actual logistics of how these variants impacted training were still cloudy. But it seems that every hyper parameters is worth at least considering.

I gained a more rigorous understanding for why a nonlinear activation function is necessary in the network architecture, as it prevents the problem form completely devolving to a linear regression problem for an individual matrix. I understand that regression is a big part of what is taking place in the neural networks but there is a bit more going on.

I learned that filters often behave in a manner very different from what I initially expected, even when they are working. Because experiment 2 used 2 convolutional filters and was classifying data into 2 categories I assumed that one filter would learn to find ‘3’s and the other would look for ‘1’s. However, many of the times both filters would search for the same value. Also the filters rarely resembled ‘3’s or ‘1’s explicitly, but instead resembled ‘3’s overlayed with ‘1’s or vice versa, where the pixels corresponding to where a ‘3’ might exist being very high valued and the pixels where a ‘1’ might exist being very low valued.

I have now been exposed to a real world application of the Calculus 3 chain rule combined with linear algebra which is in a way very satisfying as it helps solidify that there was in fact utility in learning this higher level mathematics.

Most importantly I have learned to come to office hours. I spent an enormous amount of time debugging this code up until the due date trying to find the errors in my code. When I finally came for help I realized that my backpropagation algorithm was wrong and the network was failing to converge because of a logical error in the algorithm being designed, not because of a coding issue. After that writing up the code was fairly quick, though the experiment 3 networks are not training in time for the friday due date. I apologize for the late work and will have a more rigorous presentation on the next project.