Team Neural Nerds

CS51 Final Project

Neural Networks for Handwritten Digit Recognition

Team Members:
Todd Kawakita
Lisa Yao
Lucas Hure-Maclaurin
Maria Shen

Teaching Fellow:
Willie Yao

May 5, 2013

Neural Networks for Handwritten Digit Recognition

`http://www.youtu.be/KsSHHTUqezE`

Team Neural Nerds implemented a multi-layer neural network to train a computer to recognize handwritten digits from 0 to 9. First, we created a fully connected network that delivers recognition rates up to 92 percent, and then, in order to better the performance of our program, we implemented a convolutional neural network. Our convolutional neural network is connected and running, but so far operates at low accuracy despite our best efforts.

1 A Short Overview of Neural Networks

A neural network is a computational structure consisting of a collection of individual processing nodes–or neurons–that propagate a set of inputs through a network of neurons connected via weighted edges to determine an output. These weighted edges and the values for each neuron are summed, h_j, and plugged into an activation function, $g(h_j)$, through a process of **forward propagation**.

$$h_j = \sum_i x_i v_{ij}$$

$$g(h_j) = \frac{1}{1 + e^{-h_j}}$$

In the case of digit recognition, each input, a $14x14$ image, constitutes an input layer with each pixel or node i forward propagating through a hidden layer of j nodes and arriving at the output layer with k nodes, in this case 10 nodes identifying a digit from 0 to 9.

After propagating an input to the output layer, the weights are updated through a process of gradient descent to minimize the error between the expected value for a set of inputs and the value predicted by the network. The error function for our multi-layer neural network is the sum of the squared residuals:

$$E(w) = \frac{1}{2} \sum_{k=1}^{N} (t_k - y_k)^2$$

By calculating the partial derivative of the error function with respect to the weights, δ_j and δ_k, we can update the weights between the two layers, w_{jk} and v_{ij} in a direction minimizing the error function through the process of **back propagation**.

$$\begin{aligned} \delta_k &= (t_k - y_k) y_k (1 - y_k) \\ \delta_j &= a_j (1 - a_j) \sum_k w_{jk} \delta_k \\ w_{jk} &\leftarrow w_{jk} + \eta \delta_k a_j \\ v_{ij} &\leftarrow v_{ij} + \eta \delta_j x_i \end{aligned}$$

2 How to Run Our Code

2.1 Via the Command Line

Direct yourself to the folder **neural_nerds**. Then put the following in the command line if you would like to **test** the network for different parameters:

python neural_net_main.py **-e** 10 **-r** 0.3 **-t** hidden **-m** test **-i** 0.1 **-n** 5

Here, we are running our neural network by calling main() with 10 epochs **-e**, a learning rate **-r** of 0.3, a network type **-t** hidden (simple neural network), in **-m** mode test, incrementing the learning rate by **-i** 0.1 and running our function Train() **-n** 5 times for the incrementation of the learning rate.

Other options include **training** the network to take in digit inputs and **running** the network based off of a text file with saved weights. Note that **train** and **run** only have 4 command line arguments.

python neural_net_main.py **-e** 10 **-r** 0.3 **-t** hidden **-m** train
python neural_net_main.py **-e** 10 **-r** 0.3 **-t** hidden **-m** run

The train option will return output describing the performance of our neural network for each epoch or cycle of forward and back propagation for the **training** and the **validation** data, respectively.

```
* * * * * * * * *
Parameters => Epochs: 2, Learning Rate: 0.900000
Type of network used: HiddenNetwork
Running mode: test
Input Nodes: 225, Hidden Nodes: 15, Output Nodes: 10
* * * * * * * * *
1 Performance: 0.87477778 0.858
2 Performance: 0.89977778 0.873
3 Performance: 0.90800000 0.874
```

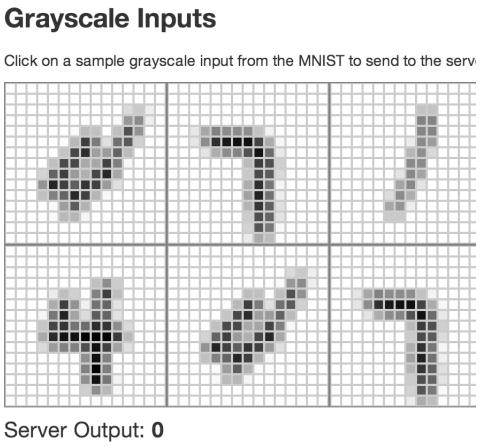
2.2 Via the User Interface

In the terminal, start the local python server by typing the following into the command line:

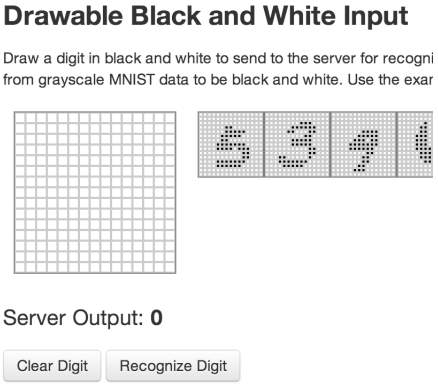
python webserver.py

This will cause **index.html** to pop up in your favorite browser. You will then treated with an interactive user interface with two components:

- a grayscale image recognition interface with sample images and



- a black and white input interface that allows the user to draw his or her own digits for the neural network to recognize.



Our grayscale image interface shows the images that are part of the MNIST dataset we used for training our neural network. These images are grayscale as a result of antialiasing, as described in MNIST's description of its work.

The black and white image interface allows for user input. To avoid having to write thousands of numbers to generate a black and white training data set, we converted the MNIST data to go from being grayscale to being just black and white. We then trained our network on this "new" training data. The results aren't always as reliable as the grayscale image recognition but we think it's pretty cool!

3 Implementation: Design, Justification, and Outcomes

3.1 The Multi-layer Neural Network with 3 Layers

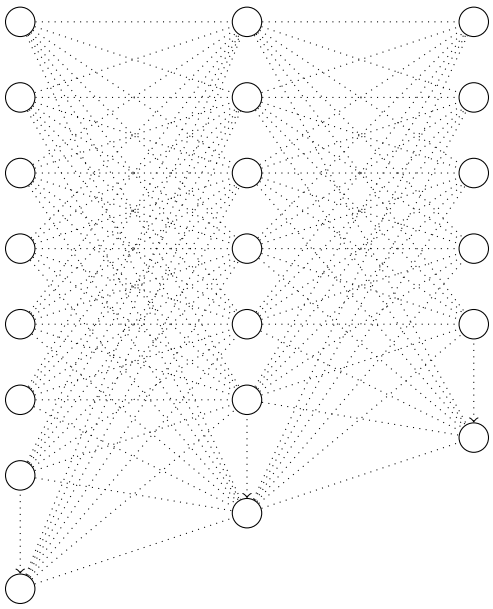
Our basic neural network took advantage of support code provided by CS181: Machine Learning. We designed methods for:

- customizing our network for digit recognition,
- creating nodes,

- connecting the 3 layers,
- forward propagating the network, and
- backpropagating the network.

After a network has been trained, we enabled it to save the trained weights to a text file that could be read in and loaded into a neural network data structure. We also modified the main() function to accept additional command line arguments and related files to implement other run modes beyond just **training** but also **running** using the saved weights, and **testing** for cycling through different parameters. We also wrote a function to try to counter the anti-aliasing effect present on the images in the original MNIST dataset to turn them grayscale values into from black and white values that could be trained to work with the user drawn images.

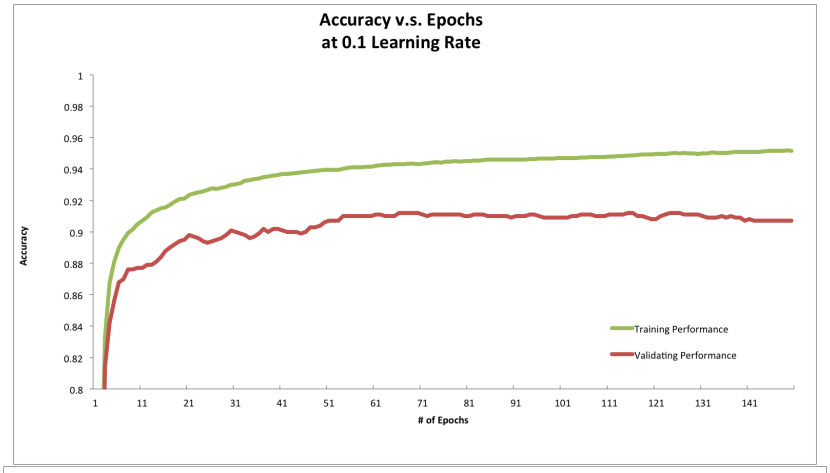
Ultimately, we decided to go with 3 layers of fully-connected input, hidden, and output layers containing **225** input nodes (more on 225 in the next section), **15** hidden nodes, and **10** output nodes. Though we could have had a multitude of hidden layers in our neural network, we chose to have just one hidden layer, which coincided with the existing literature (Stephen Marslands Machine Learning, an Algorithmic Approach, Chapter 3). Below is a simplified representation of the multilayer neural network we implemented.

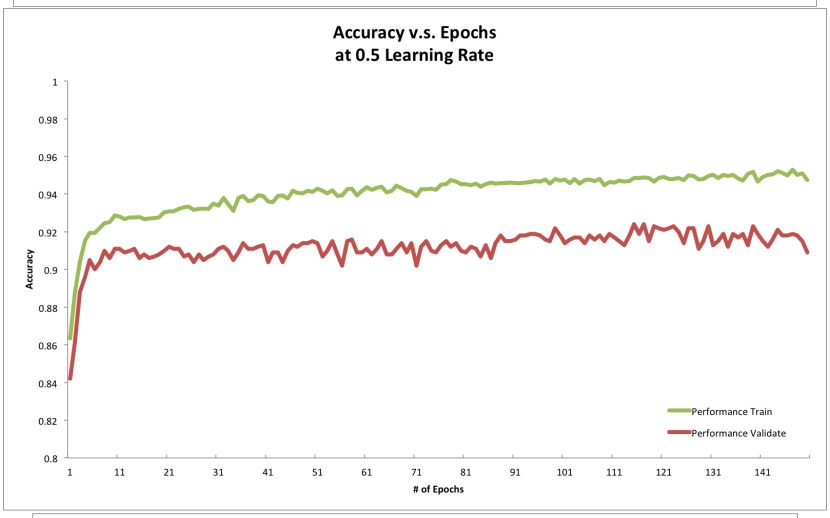


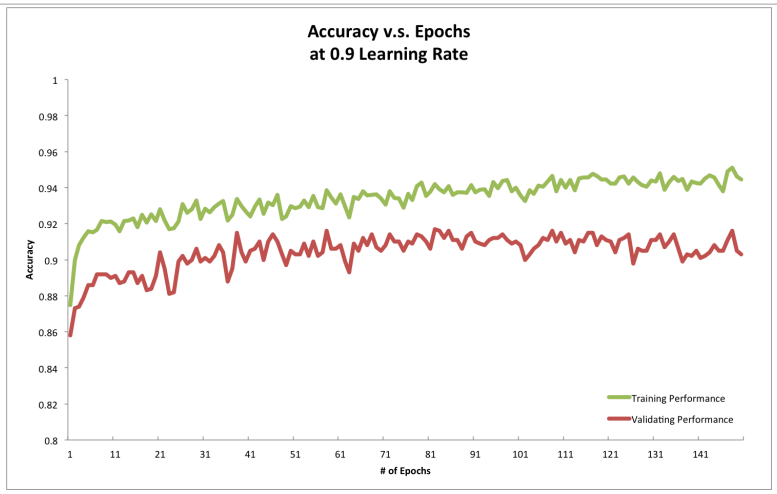
Outcomes

Our multilayer neural network was a success. We were able to test our multi-layer neural network for overfitting and observed how the learning rate affected accuracy. Our testing data is presented on the next page.

The three diagrams below display the potential for overfitting a neural network. As the number of epochs increases, the performance over the training set (which is used to train the network in the first place) diverges from the performance on the validation set (a set of new data). As a neural network is overfit, it loses the ability to generalize to different data sets.

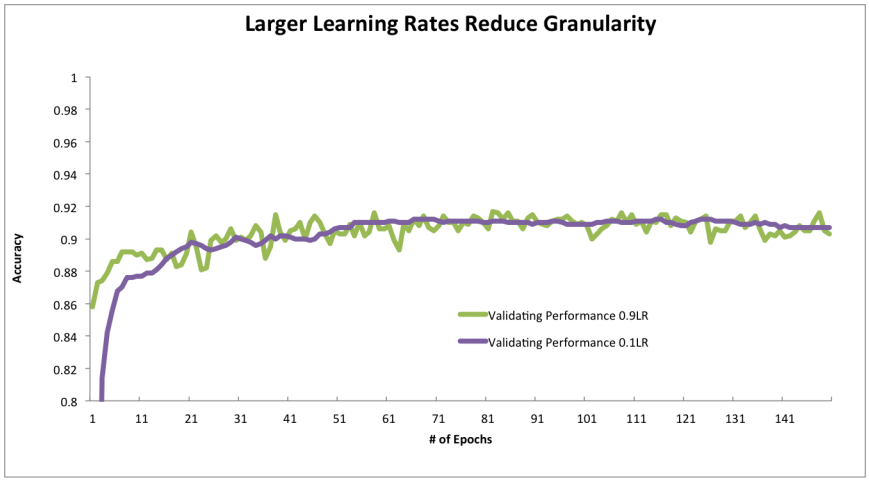




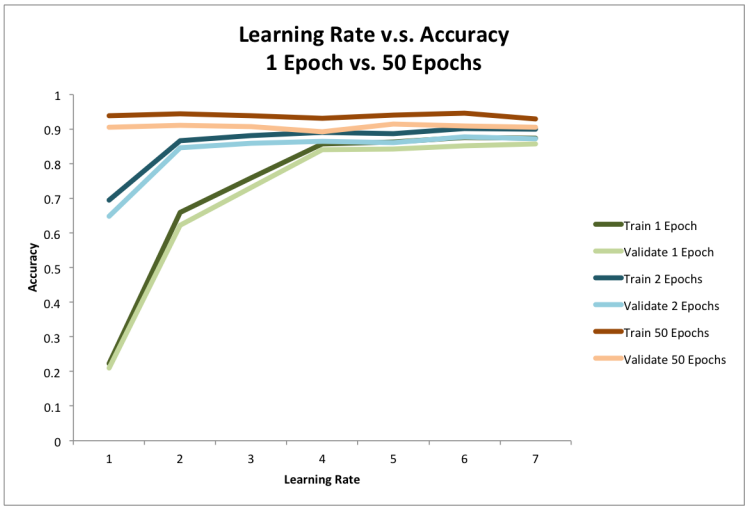


A second important key point is that as the learning rate increases, the neural network loses granularity in training by self-correcting too strongly at each step. Smaller learning rates

allow the network to maintain accuracy but take slightly longer to reach maximum accuracy. An ideal training process would use large learning rates initially to quickly reach the range of maximum performance and then reduce the learning rate thereafter to fine tune the accuracy.



At one epoch, the accuracy of our neural network increased with learning rate. At 50 epochs, the learning rate did not affect performance. The network peformance over the training and the validation sets do not differ based on the learning rate.



3.2 The Convolutional Neural Network

We took a number of steps to create our convolutional network from our more basic multilayer neural network. We:

- created two additional convolutional layers, sandwiched in between the initial input and hidden layers.
- established a new method of connectivity between layer 0 and 1 and layer 1 and 2 which did not have fully connected nodes (the exact nature of this took a long time to understand requiring an in-depth reading of a broad range of literature, diagramming, and attempting multiple designs),

- added additional functionality for the additional layer in forward propagation, and
- changed our back propagation to account for the shared weights.

Forward propagation of our convolutional network was programmatically similar to the simple network, which was feasible once our connectivity was accurate. Back propagation, on the other hand, demanded most of our time and attention. Backpropagation differed substantially from a fully connected network due to the existence of shared weights between layers 0 and 1 and layers 1 and 2. However, we ultimately determined a way to calculate errors for weights and update them based on carefully adding the partial derivatives for the shared weights.

Outcomes

In the end, our convolutional neural network did not achieve the increased accuracy that we had intended. The accuracy was about 11.4% relative to the expected high 95% we expected. This could be due to a number of factors:

- The back propagation process of updating shared weights and the associated memory management of these shared weights could have been incorrect.
- We also spent a lot of time confirming, implementing, and testing our connectivity in the convolutional neural network. We are confident in this structure, but there is still a small possibility of latent error.

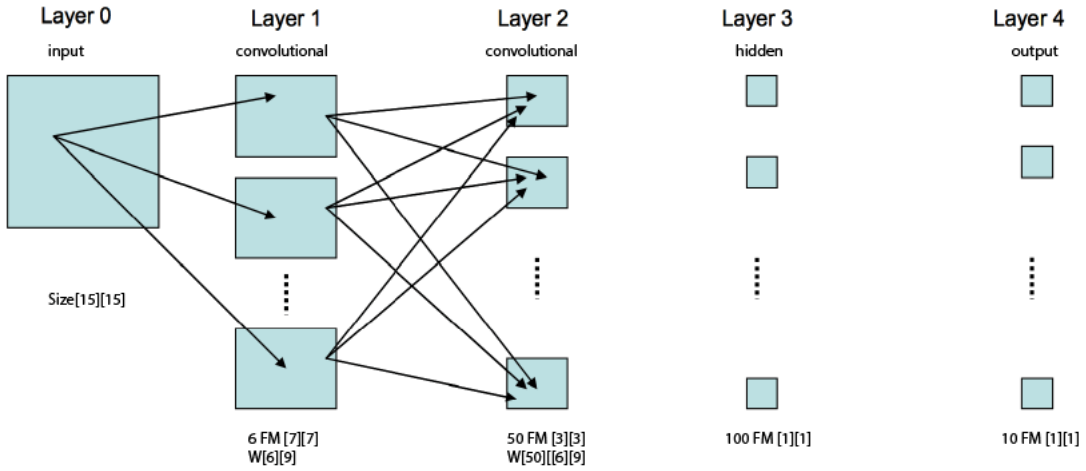
Design Decisions

In implementing the convolutional neural network, we had to make a number of design decisions to make our existing framework generalizable and to accomodate the functionality of a convolutional neural network:

- **Global shared weights array**. To implement shared weights, we created a global array of weights that could be modified and accessed universally. The benefit of a global variable was that any modifications would be represented in all instances of the variable. We also created a multidimensional array with each subarray containing weights relevant to a single feature map in a convolutional layer. Each of these subarrays was passed to the appropriate node in a feature map so that all nodes in a feature map contain the same set of weights. Changes in the global array were present in the corresponding subarray contained in nodes.
- **Kernel size and shapes**. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis by Simard, Steinkraus, and Platt indicates that the width of the kernel should be chosen to be centered on a unit, or odd-sized (page 3), since this configuration has enough overlap so that information is not lost. A balance needs to be struck so that the kernel has sufficient overlap over the unit, but not so much that there is redundant computation. Simard and co used a 5x5 kernel for their 28x28 image. Since we had a training data set with images that were 15x15 (the original images were 14x14, but we decided to give it a 1 pixel padding), we decided to use a 3x3 kernel to achieve roughly the same proportion of kernel-to-unit coverage (4%).

- **Padding size**. Though the MNIST images were $14x14$, we added 1 pixel of padding on two sides of our image to make a $15x15$ image that we also used for our more basic neural network without detriment to our initial performance. We did this for a few different reasons. Extrapolating from $28x28$ images described in the literature with a kernel size of $5x5$, we determined a kernel size of $3x3$ would be most appropriate as it would provide approximately the same surface area coverage as a $5x5$ kernel for $28x28$. Additionally, for our convolutional network to arrive at a layer of $3x3$ feature maps in layer 2, we realized that we needed to make our inputs $15x15$. By having a window offset of 2 horizontally and vertically, we would result in 7 windows or kernels across and 7 going down. This process would create a $7x7$ feature map in the first convolutional layer. In the second convolutional layer, there would be $3x3$ feature maps with the same exact size of our kernels. As Simard, Steinkraus, and Platt write, padding does not significantly affect the performance (Simard, Steinkraus, Platt, page 3). But in our case, it did allow for cleaner math in going from one layer to the next!

The following is a diagram of the convolutional neural network adapted from "A Convolutional Neural Network Approach for Handwritten Digits Recognition". We show the five layers of our convolutional neural network and the appropriate sizes of each layer. Each arrow between layer 0 and 1 and layer 2 and 3, refers to all the 9 nodes in a kernel of $3x3$ pointing to a node in the next layer. Though no arrows are shown between layers 2 and 3 and layers 3 and 4, note that these layers are fully connected.



3.3 UI for Grayscale Images and B&W

We wanted to be able to allow our users to interact with our program on a more visual level and explore the datasets we used to implement our program. To create this UI, we wrote different scripts that used Javascript (mostly jQuery and AJAX) and Python. Our UI is divided into two parts:

The first part allows the user to see pre-existing handwritten digit samples and how well our program performs against these images. In order to do this, we had to:

- convert our training dataset into images that could be displayed in HTML,

- create supporting code that allowed users to click on various images,
- convert the selected image back into the format that can be read by our program, and
- pass the output from our program over the server to dynamically display in HTML.

Because the images were in grayscale, we used our grayscale training data to train the program beforehand, saved the weights from that training session, then accordingly fed the same weights into the neural network in order to solicit the correct output.

The second part of our UI allows users to draw their own digit and see how well our program recognizes the digits. Because handwritten digits are black and white, we created a separate training dataset to be able to support this functionality. In order to do this we:

- converted our existing gray-scale training dataset into a purely black-and-white dataset,
- trained our neural network with the black-and-white dataset and saved the correct weights from the training,
- created a HTML interface that allowed users to draw their own digit,
- save created digit as an image,
- convert that image into data that our neural network could process,
- pass the data over the server, and
- return the neural networks output over the server.
- On the right side of the interface, we give small examples of the black and white training dataset for the users reference when drawing their own digit.

4 Reflections

4.1 Roadblocks and Solutions

Shared weights. We had accidentally designed our shared weights to point to the same location in memory by expanding a small array. Therefore, all shared weights were identical. We printed out the individual hex addresses of each weight to confirm this. We fixed this problem creating new weights individually appending each to our arrays of shared weights. We also realized that our weights were not being initialized in the convolutional network due to their different creation method from weights in the simple layer. We resolved this by initializing the weights contained in the arrays of shared weights directly inside our Custom Network function. A final challenge in managing the shared weights of a convolutional network was in finding a way to create an array of weights that could be mutable from any location. We created a global array of weights for each layer which contained subarrays for each feature map in the next layer. We tested using print functions to confirm that modifications in one location was represented in all other instances. We thought deeply about memory access in Python.

Indexing. We represented all the nodes in each layer as an array. We also stored errors and weights in arrays. Having faced difficulty implementing the backpropagation process for the convolutional layers, we initially attempted to hard code the indexing of the connections between nodes and shared weights from one layer to another. This led to indexing errors and extensive debugging. We then created formulas to generalize the process of iterating through and accessing nodes, weights, and errors. In retrospect, this was probably one of the instances in which planning ahead more would have saved us time and headaches.

Connectivity. The connectivity of the convolutional neural network was significantly difficult from the simple network. We spent dedicated much time to studying literature in order to understand how to connect convolutional layers. We made a key error in failing to create a new set of shared weights for each feature map in a convolutional layer.

Backpropagation. The backpropagation process in convolutional layers were also significantly different from that of a simple network. For the simple network, we benefitted from many resources detailing the error calculation process. For backpropagation, we initially attempted to adapt the formulas, but did not understand how to derive errors for the convolutional layers. Ultimately we met with a machine learning student researcher to understand the process.

4.2 Lessons

One design mistake we made was to try to use as much of our simple neural network code as possible in creating our convolutional network. This actually limited our capacity to execute our theories. We finally benefitted from rewriting many functions from the simple network for our convolutional network. Nevertheless, we learned a lot through the process of coding and debugging that we could not have by purely designing.

In general, when confronted with complex problems involving many different factors, we took the approach of trying to initially limit the number of factors to those absolutely essential to getting out program to work, after which we would add the excluded factors one by one until we got everything to work properly.

We learned how to use PDB, a Python debugger very similar to GDB for C, which allows one to navigate through code efficiently and precisely. This not only helped us debug our code, but also strengthened our understanding of how the code was running through certain complex processes, as we were able to follow them step by step. This was particularly helpful with backpropagation.

Finally, the one recurrent lesson throughout our project was that it is imperative to have a solid grasp of theory and a thorough design before programming. We had many hiccups in our code that were due to not fully comprehending the theory of what we wanted to implement. This which required us to make assumptions that were ultimately weak. A key example is failing to use a different set of shared weights to connect layer X to each feature map in layer X+1. Finally, we also learned that unexpected bugs tend to arise with even the most scrupulous planning.

4.3 Progress report

Our first plan was to implement a graph algorithm. However, on the advice of TFs, we decided that a graph algorithm lacked algorithmic complexity and decided to pursue machine learning instead. Machine learning excited all of us due to its applicability in modern technology.

We came up with a plan of action and timeline by the second submission. We adhered well to the timeline up to the completion of our simple network. Unfortunately, unexpected bugs in creating our convolutional network caused us to fall behind. We implemented the multi-layer perceptron very well. We read through all the literature we could get our hands on and made sure we understood every aspect of the implementation and the support code.

We ran into a plethora of issues when trying to implement the convolutional version of our network. Our set-back was more of a function of the sheer difficulty of implementing that algorithm perfectly than of the quality of our planning. We gave ourselves the best chance of successfully implementing every aspect of our project by allocating a lot of time consistently throughout the duration of our project to gathering information from different sources (code, literature, etc...), consulting with our TF and different qualified sources, as well as coding/conceptualizing in every combination of group members. To this end, we met nearly every day for the penultimate week preceding our deadline.

4.4 Future Work and Possible Modifications

There are a number of potential variations for our network that given more time, we would be interested in pursuing. First, an optimal way to train the network is to initially set the learning rate to a large number and then reduce the learning rate in subsequent epochs. This allows for error minimization in large steps initially and then fine tuning to a minimum error thereafter. This can reduce training time. Second, we chose to use a logistic function as our sigmoid function, but it is not symmetric. Literature recommends using a hyperbolic tangent instead of a logistic function to optimize the gradient descent process. Third, we determined a number of hidden and convolutional layers based on what we came across in literature but they are by no means the only option. One may consider using more or fewer layers with more or less connectivity to optimize the network accuracy. Further, one may consider changing the kernel size, the kernel iteration pattern over a feature map (skipping 1, 3, or more instead of 2 nodes on each movement), and increasing or decreasing feature map size. Finally, we implemented convolution using a global shared weights array

4.5 Other applications

A neural network is extensible and can be applied to nearly all situations. A network like ours can be adapted to recognize other visual inputs like letters, drawings, and perhaps even photographs, a prospect that has very excited!

5 Group Member Contributions

- Todd Kawakita
  - Multilayer Neural Network

  - Convolutional Neural Network
  - UI
  - Documentation
- Lisa Yao
  - Multilayer Neural Network
  - Convolutional Neural Network
  - Documentation
- Lucas Hure-Maclaurin
  - Multilayer Neural Network
  - Convolutional Neural Network
  - Documentation
  - Data conversion
- Xinhe Shen
  - Convolutional Neural Network
  - UI
  - Data conversion
  - Documentation

We all contributed heavily and met nearly every day for the 2 weeks preceding the project deadline! It was fantastic working with a highly motivated group tackling a neural networking challenge. We're not sure if we are worthy of the "neural nerds" moniker but we've definitely made great strides by working on this project!