Alexander Peralta
CMSC 426
Project 3

Section 1: Overview and Background Details

The goal of this project is to classify images of numbers 0-9 using a neural network. Two different types of neural networks were utilized, a mulit-layer perceptron neural network and a convolutional neural network. In order to accomplish this goal a large dataset was required. The MNIST dataset is a collection of images of handwritten numbers with labels correctly identifying the numbers. Each image is originally 28x28 pixels but was vectorized to be a 1x784 vector. These image vectors were used as the inputs for the neural networks. Three different datasets of image vectors were provided in the form of numpy arrays - a (50000, 784) training set, a (5000, 784) validation set, and a (5000, 784) testing set - along with corresponding one dimensional label arrays. The training set was used to train the neural networks, the validation set was used during training to validate / make sure the network was working correctly and not memorizing the training set , and the testing set was used after training to check the accuracy of the networks against images that it had not seen before. To make tuning the networks more manageable, they were first trained only using 2000 images in the training set, and then later using the entire 50,000 images. For a similar reason and for an implicit form a regularization, stochastic gradient descent was used with a batch size of 256.

Section 2: Multi-layer Perceptrons Neural Network

The first neural network was created using multi-layer perceptrons. This network has an input layer of 784 perceptrons corresponding to the 784x1 image vectors (the image vectors were transposed). There is a hidden layer with 64 perceptrons and then an output layer with ten perceptrons. The activations in the hidden layer are compressed using a sigmoid function and the activations in the output layer are normalized using a softmax function. Loss of the network is calculated using cross-entropy loss function. W1 is a 64x784 matrix of weights connecting the input layer and the hidden layer. W2 is a 10x64 matrix of weights connecting the hidden layer and the output layer. b1 is a 61x1 vector of biases corresponding to the hidden layer activations. b2 is a 10x1 vector of biases corresponding to the output layer activations. Numpy arrays are used to store the vectors and matrices- whenever either term is used in this section of the report, the code being referenced is a numpy array. The output of the network is a 10x1 vector where each component of the vector is a value between 0-1, with the index position corresponding to the image number. The index with the largest value is the number that the network thinks the image represents.

The network was initialized by creating W1 and W2 weight matrices and setting all values inside those matrices to random gaussians with a standard deviation of 0.1. Bias vectors b1 and b2 were initialized to zero.

The forward pass of the network takes a 784x1 image vector as input, which we'll call x. It multiples W1*x and adds that to the b1 bias vector to form a 64x1 activation vector a1, where each component of the vector is one of the 64 hidden layer activations. The a1 vector is passed through a sigmoid function with the output being represented by a 64x1 vector named f1. This vector is multiplied by the second weight matrix, W2 * f1, and added to the b2 bias vector the for a 10x1 activation vector a2. The a2 vector is passed through a softmax function with the output is a 10x1 vector named y_hat.

The weights and biases are adjusted using back propagation in order for the network to be able to accurately classify images. The partial derivatives that define this process are:

$$\frac{\partial L}{\partial W2} = \frac{\partial L}{\partial a2}\frac{\partial a2}{\partial W2} \qquad \frac{\partial L}{\partial W1} = \frac{\partial L}{\partial a2}\frac{\partial a2}{\partial f1}\frac{\partial f1}{\partial a1}\frac{\partial a1}{\partial W1}$$

$$\frac{\partial L}{\partial b2} = \frac{\partial L}{\partial a2}\frac{\partial a2}{\partial b2} \qquad \frac{\partial L}{\partial b1} = \frac{\partial L}{\partial a2}\frac{\partial a2}{\partial f1}\frac{\partial f1}{\partial a1}\frac{\partial a1}{\partial b1}$$

The jacobian matrices for the partial derivatives are:

Jacobian 1x10 matrix for $\frac{\partial L}{\partial a_2}$

$$[\hat{y}^1 - y^1, \;\; \hat{y}^2 - y^2, \;\; \ldots \;\; \hat{y}^{10} - y^{10}]$$

Jacobian 64x64 matrix for $\frac{\partial a1}{\partial b1}$

$$I_{64}$$

Jacobian 64x64 matrix for $\frac{\partial f1}{\partial a1}$

$$diagonal \quad matrix \quad with \quad f_1^n(1 - f_1^n)$$
$$along \quad the \quad diagonal$$

Jacobian 10x10 matrix for $\frac{\partial a2}{\partial b2}$

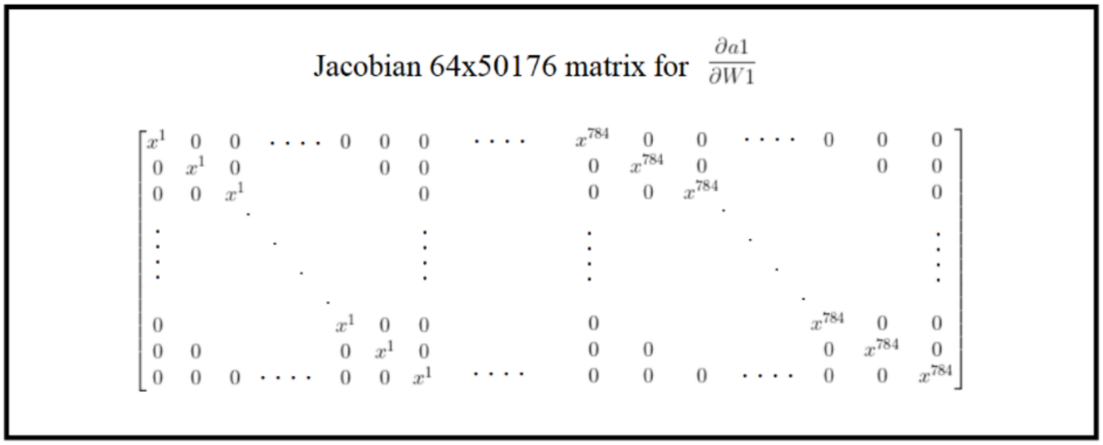
$$I_{10}$$

Jacobian 10x64 matrix for $\frac{\partial a2}{\partial f1}$

$$W2$$

Jacobian 10x640 matrix for $\frac{\partial a2}{\partial W2}$

$$\begin{bmatrix}
f_1^1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & f_1^{64} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & f_1^1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & f_1^{64} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & f_1^1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & f_1^{64} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & f_1^1 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & f_1^{64} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & f_1^1 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & f_1^{64} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & f_1^1 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & f_1^{64} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & f_1^1 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & f_1^{64} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & f_1^1 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & f_1^{64} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & f_1^1 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & f_1^{64} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & f_1^1 & \cdots\cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & f_1^{64}
\end{bmatrix}$$



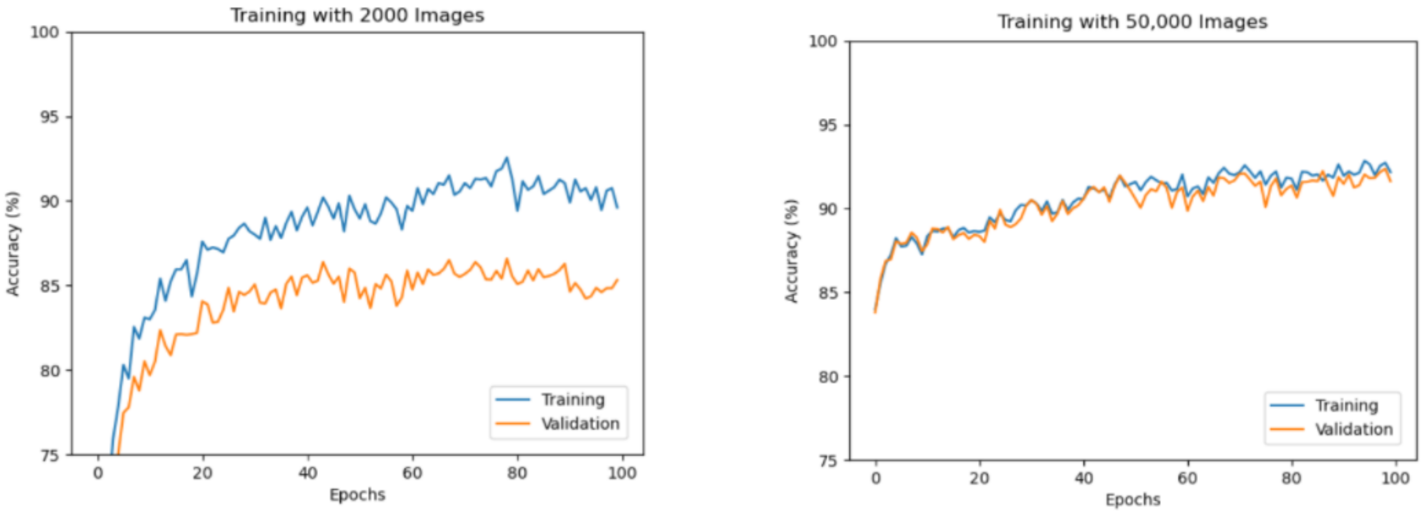
Jacobian 64x50176 matrix for $\frac{\partial a1}{\partial W1}$

$$\begin{bmatrix}
x^1 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & x^{784} & 0 & 0 & \cdots & 0 & 0 & 0 \\
0 & x^1 & 0 & & & 0 & 0 & & 0 & x^{784} & 0 & & & 0 & 0 \\
0 & 0 & x^1 & & & & 0 & & 0 & 0 & x^{784} & & & & 0 \\
\vdots & & & \ddots & & & \vdots & & \vdots & & & \ddots & & & \vdots \\
0 & & & & x^1 & 0 & 0 & & 0 & & & & x^{784} & 0 & 0 \\
0 & 0 & & & 0 & x^1 & 0 & & 0 & 0 & & & 0 & x^{784} & 0 \\
0 & 0 & 0 & \cdots & 0 & 0 & x^1 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & x^{784}
\end{bmatrix}$$

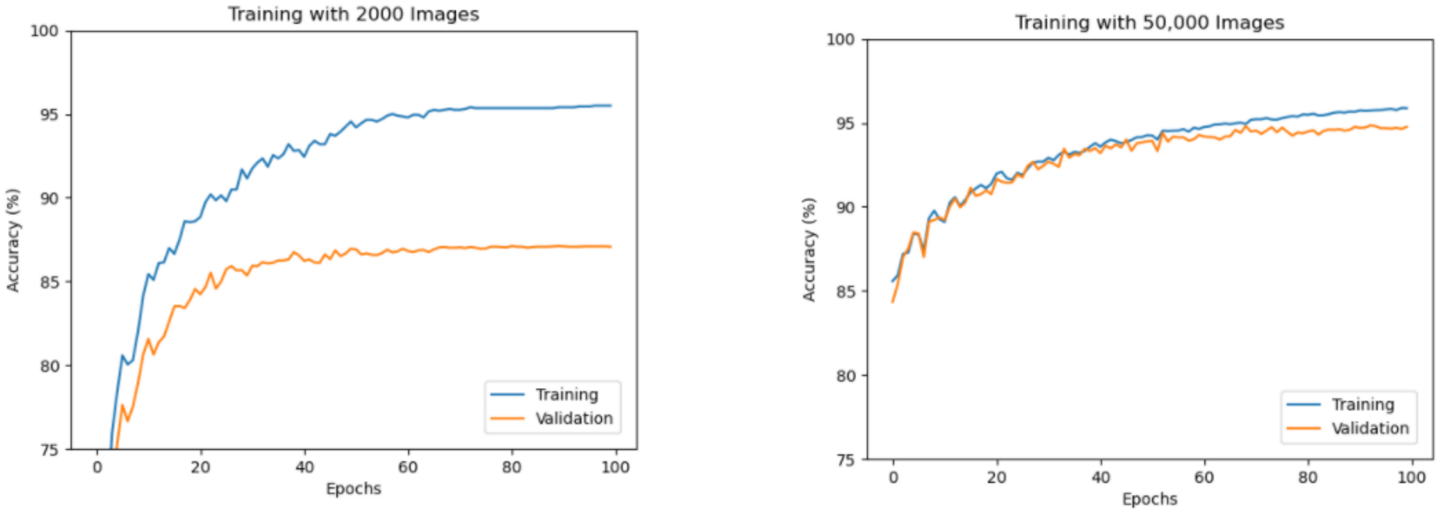
The purpose of back propagation is to find the gradient of the cross entropy loss function with respect to the weights and biases, and subtract these gradients from their weights and biases in order to decrease the outputs of loss function and create a more accurate network. The overarching idea is take steps down to find a local minimum of the loss function. To ease our referencing, we will refer to these different gradients by their differing weights and biases, i.e. the W1 gradient, the W2 gradient, the b1 gradient, and the b2 gradient. Each of these gradients is found using the partial derivative equations referenced on the previous page, substituting in the respective Jacobian matrices. For the W1 and W2 gradients respectively, a 1x50176 vector and a 1x640 vector are the results of multiplication of the Jacobian matrices. This makes sense because W1 is a 64x784 matrix composed of 50176 weights, and W2 is a 10x64 matrix composed of 640 weights. The W1 and W2 gradient vectors correspond to the columns of the W1 and W2 matrices. The 1x50176 W1 gradient vector is reshaped to form a 64x784 matrix. This reshape can take place by transposing the vector and then reshaping by columns. Python however reshapes by rows so this gradient vector can be reshaped by rows to form a 784x64 matrix and then transposed to be 64x784 The 1x640 W2 gradient vector undergoes the same reshape to form a 10x64 matrix. For the b1 and b2 gradients respectively, a 1x64 vector and a 1x10 vector are the results of multiplication of the jacobian matrices. Both are transposed. Now all gradients are in the correct form to be added to the original weight matrices and bias vectors. Note, in the actual code, the matrices were being formed implicitly to reduce computation time, but the spirit of the multiplication of jacobian matrices is being followed.

The network was trained using 100 epochs and a batch size of 256 images. In the code, a for loop iterated 100 times for each epoch and a nested for loop iterated n time, where n equaled either 2000 or 50000, corresponding to the number of training images used. An if statement inside the second for loop along with a counter kept track of batch size and updated the gradients and reset the batch every 256 images. For each epoch, the images were passed into the network in a random order which was accomplished by shuffling a sequence of 1 through n numbers where n is still 2000 or 50000, saving it as an array and accessing training images in that order. To form the validation graphs during the training at the end of each epoch the training dataset and the validation dataset were each separately forward passed through the network checking the accuracy of how the network categorized each image. The first version of the network did not have an adaptive learning rate and used a learning rate of 0.001 for every epoch. The second version did use an adaptive learning rate, multiplying the previous rate by 0.97 at the end of every epoch. The difference between the two is shown in the graphs below, with the second version having a much smoother curve. Since the learning rate is the step size of the gradient, you can see that a decreasing learning helps the gradient step more accurately to that local minimum. When training with 2000 images the accuracy starts off low and increases fast for about the first 20 epochs and then increases at a slower rate before tapering off. When training with 50000 images, the accuracy starts off after only 1 epoch is fairly high around 85% and then increases at a slow rate. For training with 50000 images, the accuracy against the validation set is comparable to the accuracy against the training set, compared to training with 2000 images where there is a distinct separation between the two sets.

MLP without Adaptive Learning Rate



MLP with Adaptive Learning Rate



After training was completed on the network with an adaptive learning rate, the test dataset was forward passed into the network and accuracy was recorded. The MLP neural network had an accuracy of 94.8 or approximately 95%. A confusion matrix breaking down each type of image is below.

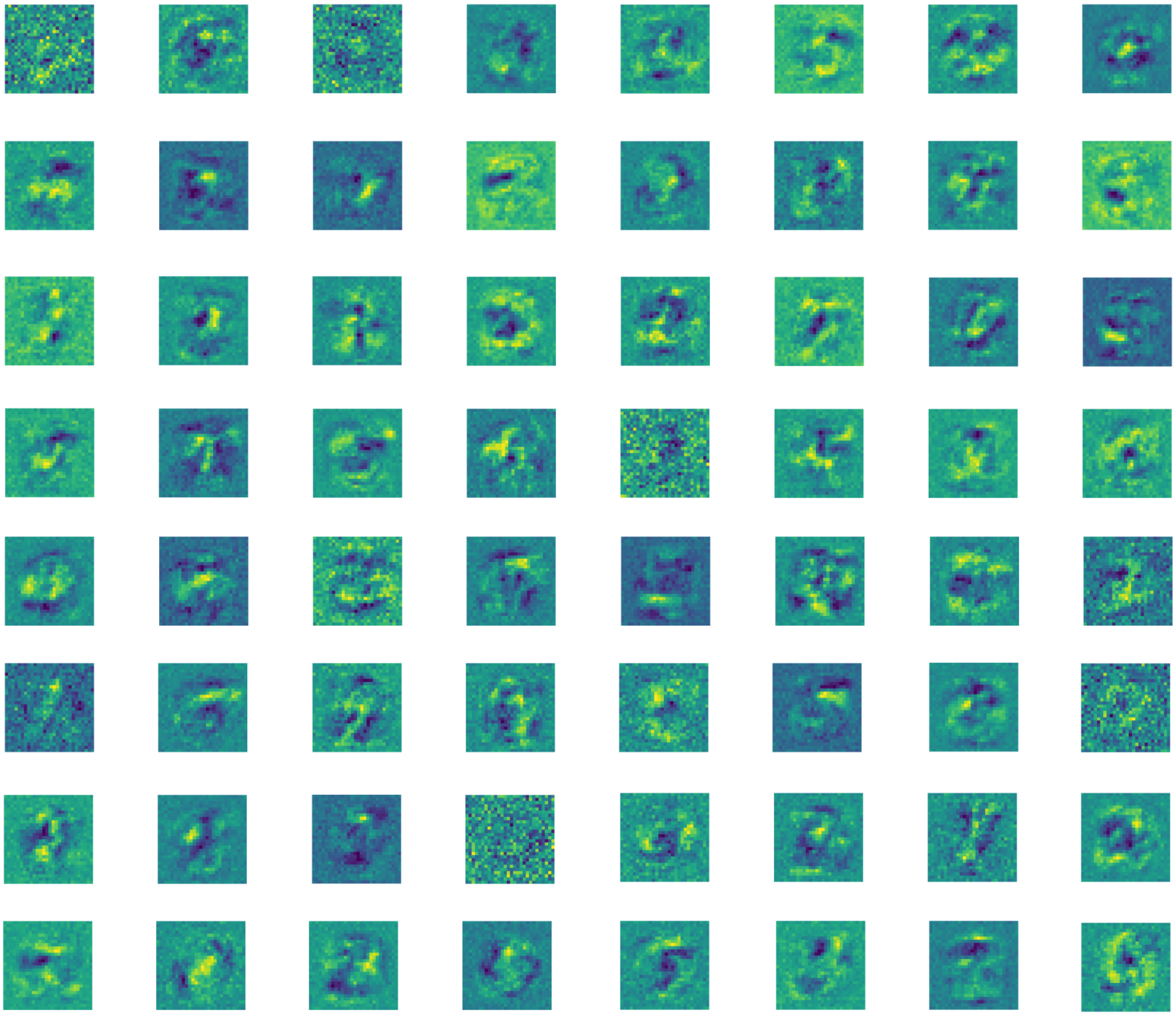
Predicted Number

Actual Number

	0	1	2	3	4	5	6	7	8	9
0	97.95	0.00	0.41	0.00	0.00	0.20	0.20	0.00	1.23	0.00
1	0.00	96.35	0.33	0.83	0.00	0.33	0.50	0.33	0.83	0.50
2	0.63	0.21	95.40	0.63	0.84	0.21	0.21	1.46	0.42	0.00
3	0.38	0.38	1.33	90.34	0.00	3.79	0.19	0.57	2.27	0.76
4	0.21	0.62	0.41	0.21	94.42	0.21	1.24	0.21	0.21	2.27
5	0.46	0.00	0.69	0.93	0.00	96.53	0.46	0.46	0.23	0.23
6	1.20	0.60	0.80	0.20	0.40	0.60	95.78	0.00	0.40	0.00
7	0.39	0.78	0.58	0.19	0.58	0.19	0.00	96.11	0.00	1.17
8	0.00	0.43	0.21	1.28	0.64	0.86	0.21	0.21	95.49	0.64
9	0.59	0.00	0.19	2.75	2.36	0.39	0.20	2.36	0.79	90.37

The weight matrix W1 from the trained network has the shape 64x784. Each of these 64 rows are visualized below in order from left to right, top to bottom. Each row represents the weights that connect to one of the 64 perceptrons of the hidden layer, so these 64 visualizations should be viewed with the 64 hidden layer perceptrons in mind.

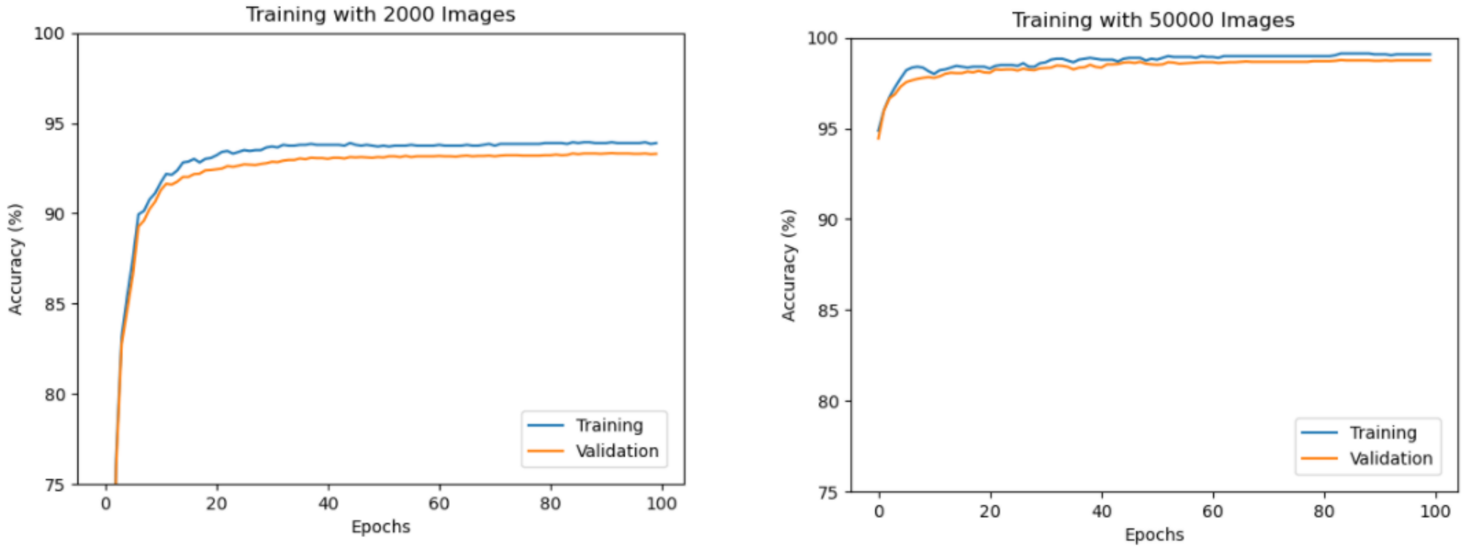
Visualization of Weights forming W1



Section 3: Convolutional Neural Network
For the convolution neural network we largely followed the tutorial and code from here https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html utilizing the pytorch library to define the CNN along with the cross entropy loss and a stochastic gradient descent optimizer. The CNN architecture takes the same 1x784 vector image used by the MLP but as a tensor. It then uses view() to reshape it to (-1,2,28,28), convolves it with 6 distinct 1x5x5 filters, applies a RELU, and then applies a maxpool with a kernel size of 2 and a stride of 2. The output of this is now used as the input and passed through similar operations. The output is not a 6 channel input and is convolved using 16 distinct 6x5x5 filters, passed through a RELU and then downsampled with the same maxpool function. The output from this is then vectorized using view(-1, 16*4*4). To get to the fully connected layer it is then passed through a linear transformation nn.Linear(16*4*4, 120), a RELU, a second linear transformation nn.Linear(120, 84), a RELU, and a final linear transformation nn.Linear(84, 10). This fully defines the network architecture.

Training and testing was done in the same manner as described for the MLP, except with images having to be transformed into tensors and batches being passed into the network the entire batch at a time. Similar graphs comparing the network accuracy during training on the training dataset and validation dataset are below. Unlike the MLP, the network accuracy against the validation set is comparable to the accuracy against the training set for both training with 2000 images and training with 50000 images. After training completed the CNN was tested against the unseen testing dataset and had an accuracy of 98.88 or approximately 99%

Convolutional Neural Network


Training with 2000 Images
Accuracy (%)
Epochs
Training
Validation
Training with 50000 Images
Accuracy (%)
Epochs
Training
Validation