Alex Ho
ID: 1001294712
CSE 4334-001

Assignment 3 Report

Problem 1:

I implemented a logistic regressor by creating a single layer neural network. In my code, I created an object called called LRModel which contains a vector of weights, vector of all errors for each iteration, a vector of all gradient norms for each iteration, and the number of iterations. LRModel trains its weights by utilizing the batchTrain() function and an onlineTrain() function. I instantiated two LRModels and used them to get and plot data for each training function.

Part 1: (Figures below or in "./assignment3/figures/Batch Training")

I performed batch training on the training data using my batchTrain() function in the LRModel. The onlineTrain() function first creates a 3 row vector with 1 column (1st row is the bias, 2nd and 3rd row are random weights). I then instantiate a list to keep track of all errors, a list to keep track of gradient norms, and set the iteration to 0. I train the model by doing iterations until the L1 norm of the gradient is < 0.001 or it completes 100,000 iterations.

For each iteration, I create an error and gradient vector and enter a for loop to find their values using the current weights. For each set of training data xi, I first insert a 1 as the first element to multiply with the bias later. I calculate the output, use it to calculate the gradient at i using the formula *(o – y)*xi*, then add to the gradient vector. I then calculate the error using Cross Entropy and add it to the error vector.

Once I get the error and gradient values from the training data, I divide each by the amount of data to normalize it. I then calculate the change in weight using the formula *gradient * (-learning weight)*. I update the weights, add the current error to the list of all errors, and increment the iteration. I also calculate the L1 norm using the gradient, add it to the gradient norms list, and exit if it's < 0.001. Otherwise I set the error and gradient vectors to 0 and repeat the process of finding them for the next iteration.

The results of my batch training are included below, and can also be found in "./assignment3/figures/Batch Training", titled x_y.png. x is the learning rate used which is also displayed in each figure (1 = 1, 2 = 0.1, 3 = 0.01). y is which graph it is (1 = Test Data + Decision Boundary, 2 = Training Error w.r.t. Iterations, 3 = Gradient Norms w.r.t. Iteration)).

NOTE: I was unable to complete the batch training for a learning rate of 0.001. While I was training the model, I accidentally allowed a maximum iteration number of 1,000,000 and did not have enough time to retrain it at 100,000. I'll discuss the expected results in Part 2.

Each training took a long time, as it was able to converge with a learning rate of 1. Each instance of a lowered learning rate simply multiplied the time it took to process by 10, while giving the same results.

Part 2: (Figures below or in "./assignment3/figures/Online Training")

For part 2, performing online training was very similar with the only difference being where the weights were calculated. The onlineTrain() function in the LRModel first creates the same values as batchTrain(). It then enters a for loop to do the training iterations.

In the for loop, I first get the training data i and its label. I insert 1 as the first element of xi to multiply with the bias later. I calculate the output, use it to get the gradient using the formula *(o – y)*xi*, and calculate the change of weights using the formula *gradient * (-learning weight)*. After updating the weights, I increment the iteration number and calculate the error to add to list of all errors. I also calculate the L1 norm using the gradient, add it to the gradient norms list, and exit if it's < 0.001. If the L1 norm is not < 0.001, it loops to complete the next iteration.

The results of my batch training are included below, and can also be found in "./assignment3/figures/Online Training", titled x_y.png. x is the learning rate used which is also displayed in each figure (1 = 1, 2 = 0.1, 3 = 0.01, 4 = 0.001). y is which graph it is (1 = Test Data + Decision Boundary, 2 = Training Error w.r.t. Iterations, 3 = Gradient Norms w.r.t. Iteration)).

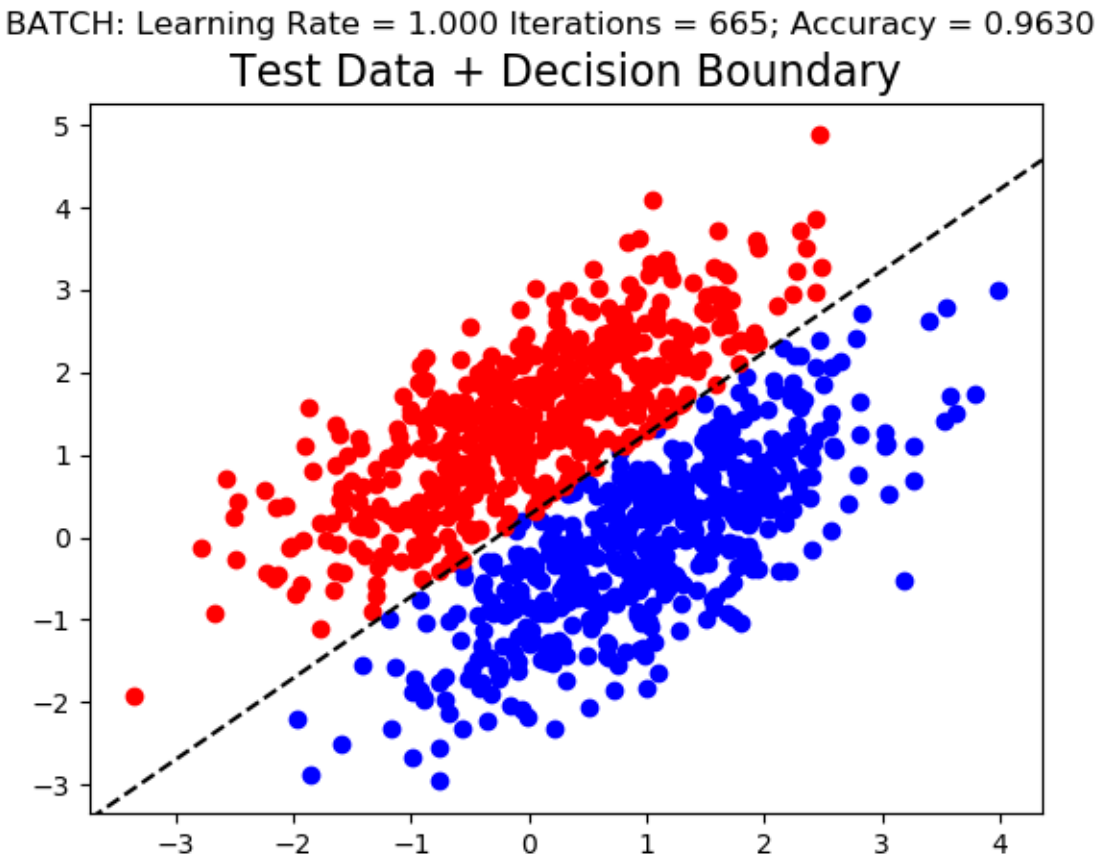
Online training was significantly faster than Batch Training, but its accuracy depended on the learning rate. Error rates and gradient norms also fluctuated a lot, with a large spike when it switches from the class 0 training data to the class 1 training data (First 500 data was class 0, next 500 was class 1). Despite this, it was still able to converge to be as good as the Batch Training while still being significantly quicker. Each iteration is also much faster, since an iteration of Batch Training is the all 1000 sets of training data, while an iteration of Online Training is only one set of data each.

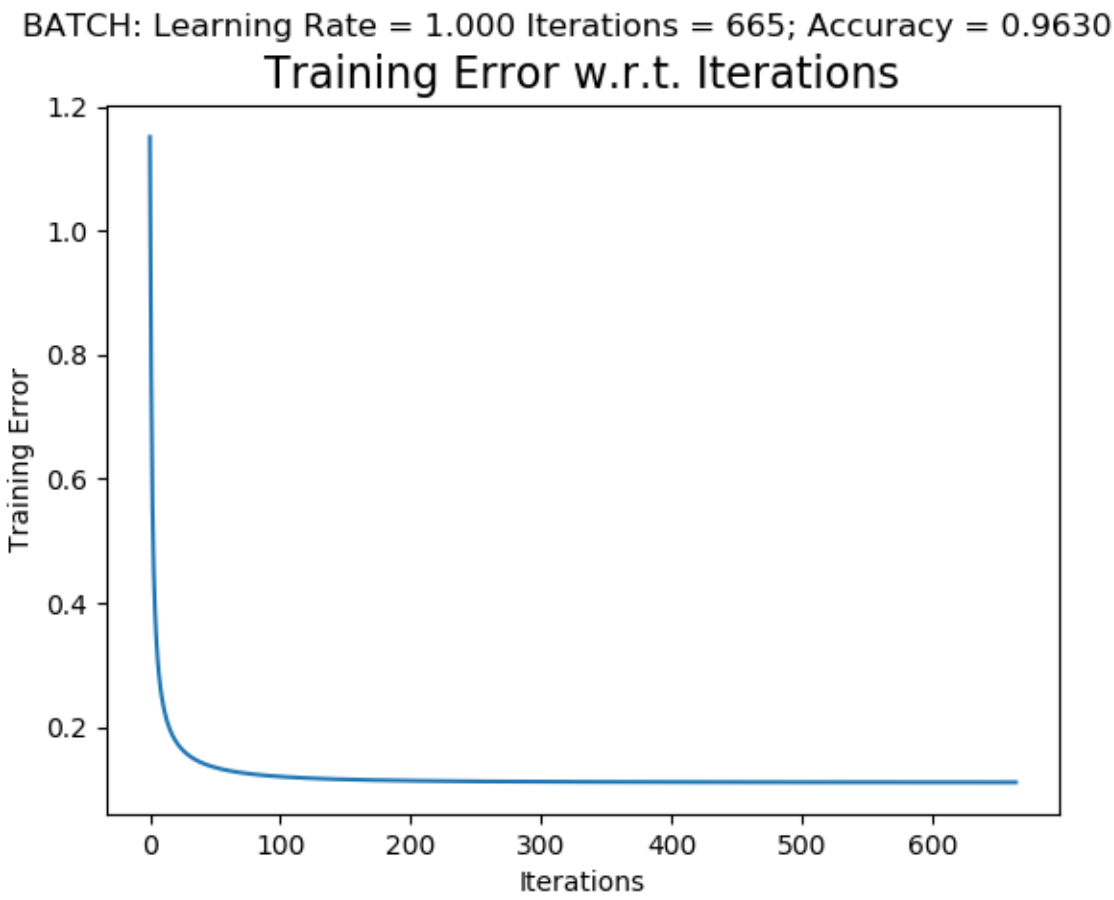
Batch Training vs Online Training

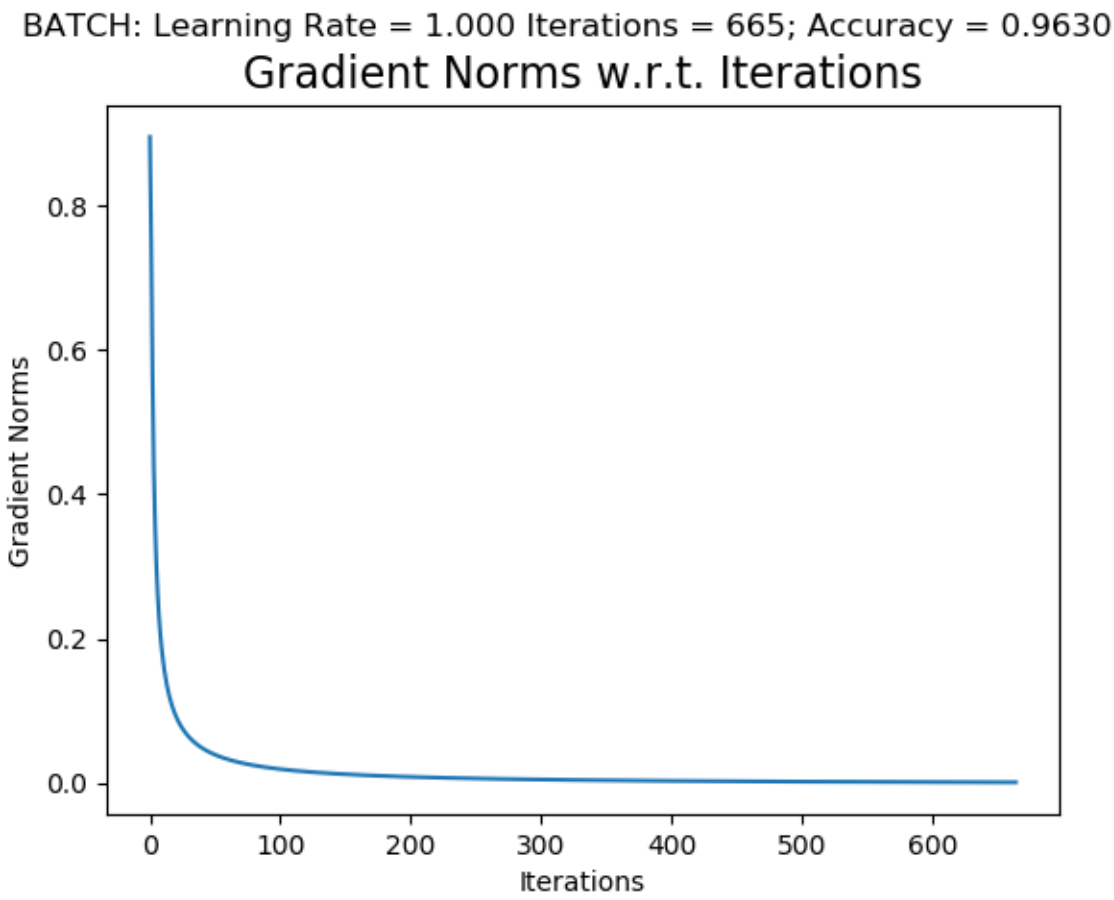
	Batch Training		**Online Training**	
Learning Rate	**Accuracy**	**Iterations**	**Accuracy**	**Iterations**
1	0.96300	665	0.6150	23
0.1	0.96300	6653	0.9050	618
0.01	0.96300	66553	0.9620	4488
0.001	N/A*	N/A*	0.9630	41023

* As stated, I was unable to complete the batch training for a learning rate of 0.001 in time as I had accidentally set the max iterations to 1,000,000. I expected it would have taken ~665,530 iterations for it to converge with no changes in accuracy. If the max iteration was at 100,000 as it should have been, the accuracy may have been lower since it wouldn't have fully converged like the other iterations.

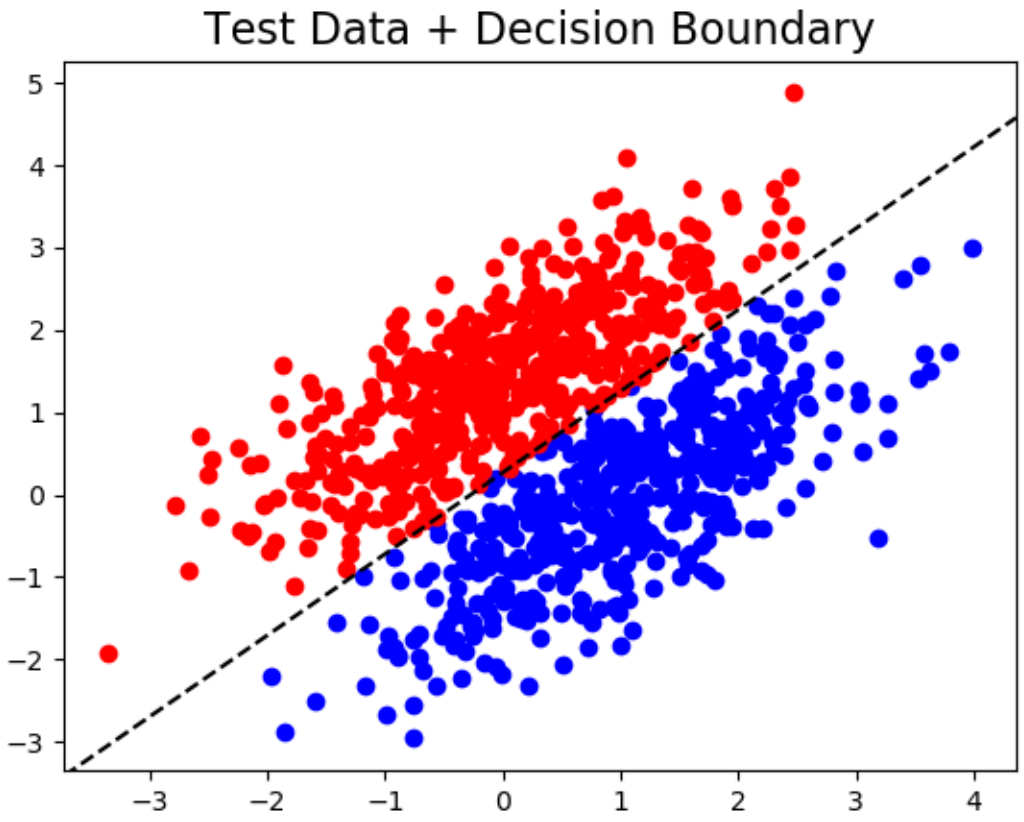
Part 1: Batch Training Figures



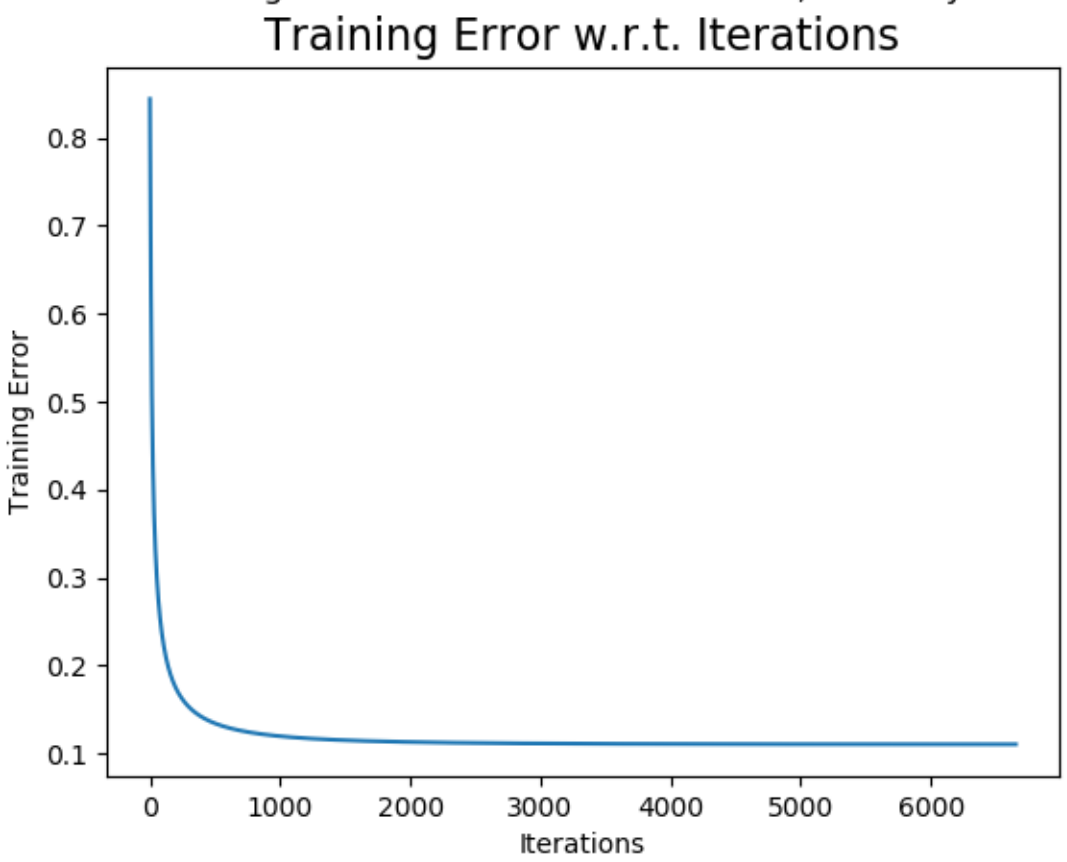

BATCH: Learning Rate = 1.000 Iterations = 665; Accuracy = 0.9630
Training Error w.r.t. Iterations
Training Error
1.2
1.0
0.8
0.6
0.4
0.2
0
100
200
300
400
500
600
Iterations


BATCH: Learning Rate = 1.000 Iterations = 665; Accuracy = 0.9630
Gradient Norms w.r.t. Iterations
Gradient Norms
0.0
0.2
0.4
0.6
0.8
0
100
200
300
400
500
600
Iterations

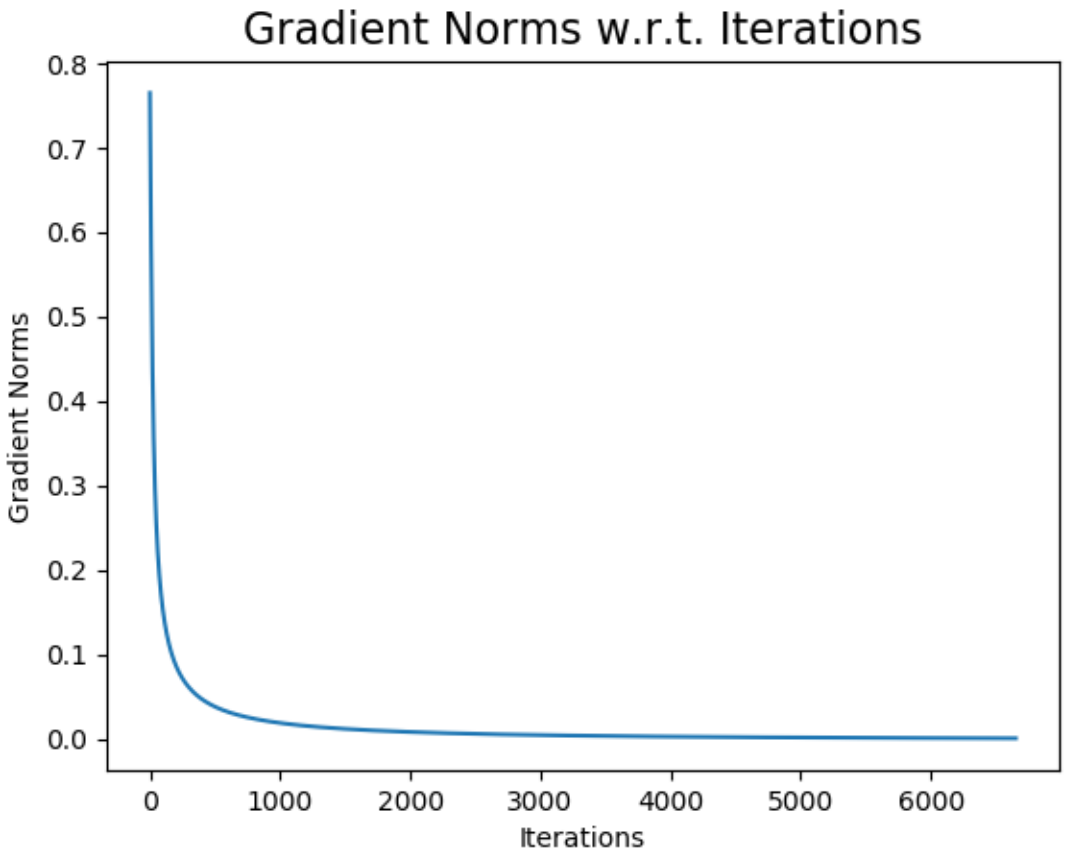
BATCH: Learning Rate = 0.100 Iterations = 6653; Accuracy = 0.9630



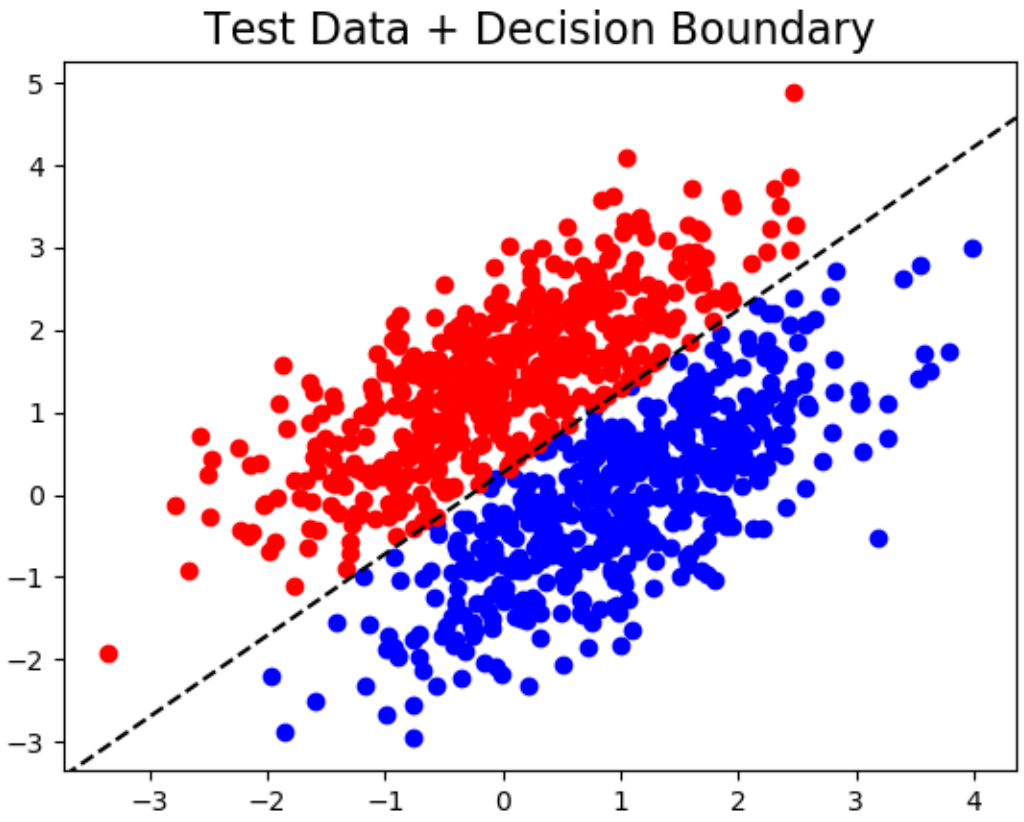
BATCH: Learning Rate = 0.100 Iterations = 6653; Accuracy = 0.9630



BATCH: Learning Rate = 0.100 Iterations = 6653; Accuracy = 0.9630



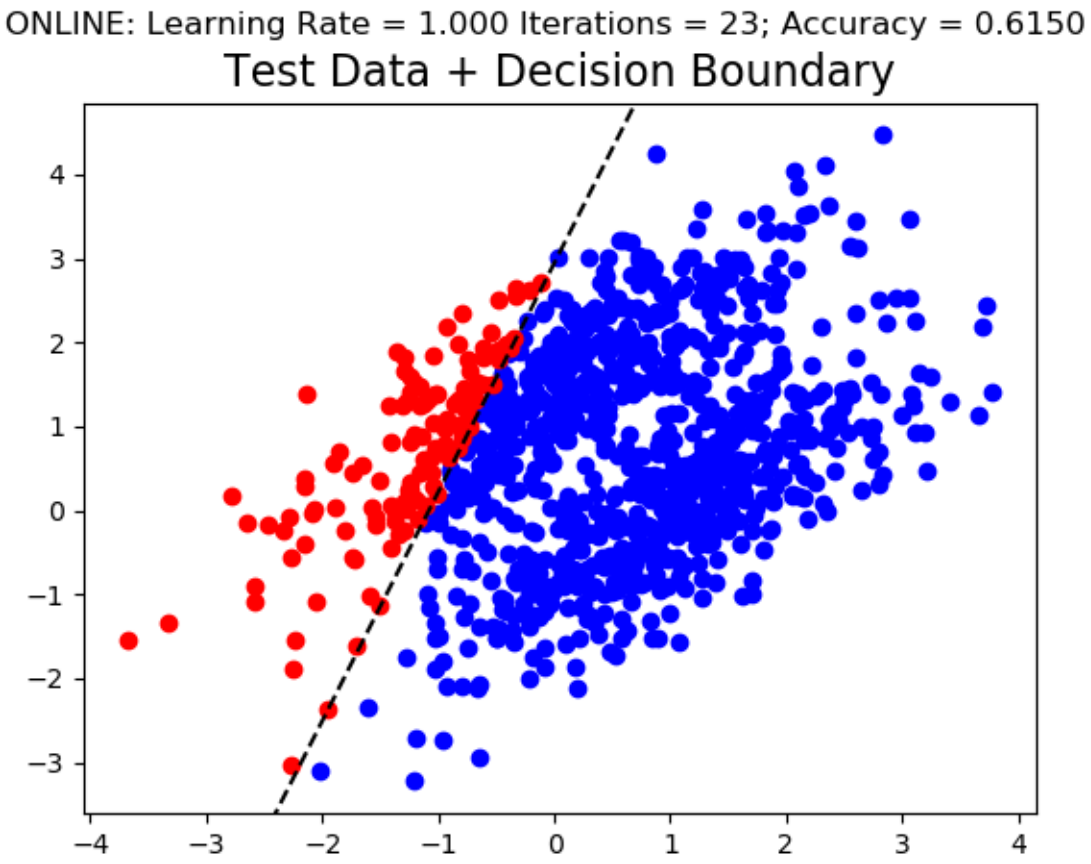
BATCH: Learning Rate = 0.010 Iterations = 66553; Accuracy = 0.9630

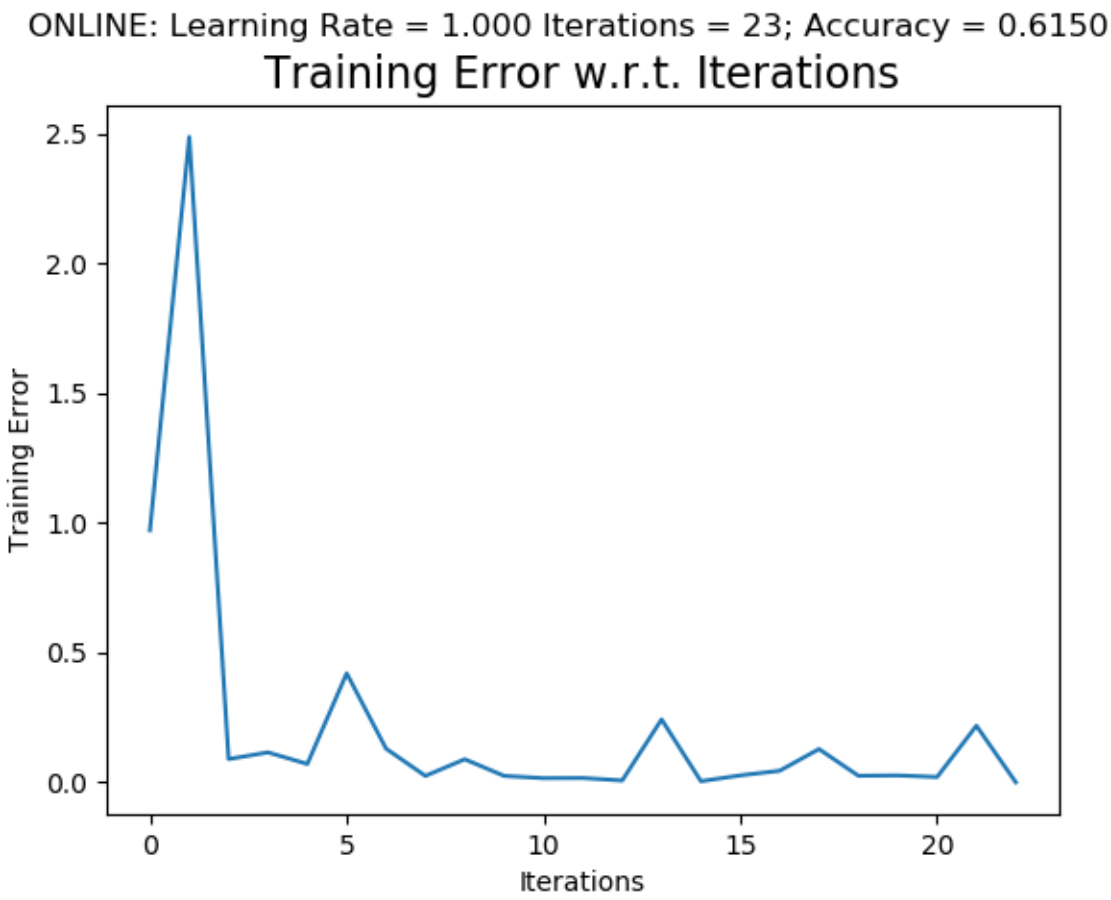


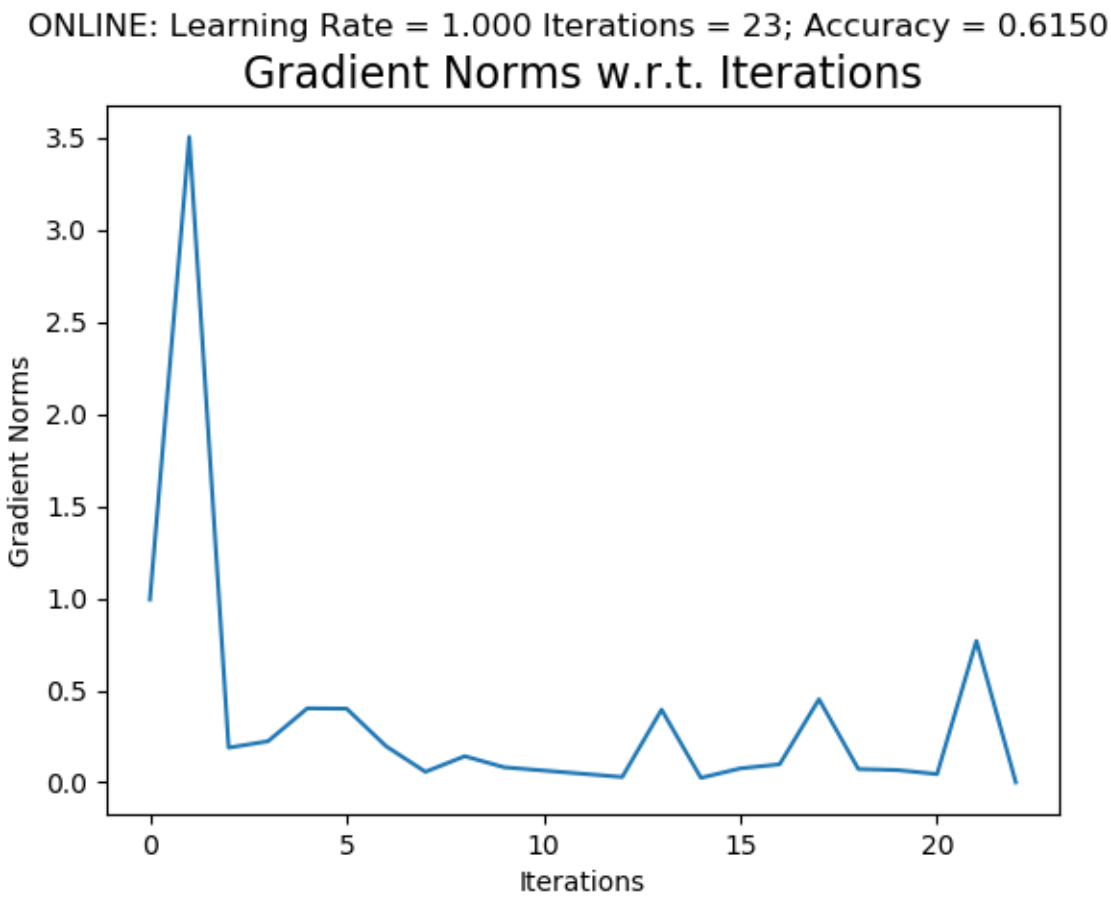
BATCH: Learning Rate = 0.010 Iterations = 66553; Accuracy = 0.9630

BATCH: Learning Rate = 0.010 Iterations = 66553; Accuracy = 0.9630

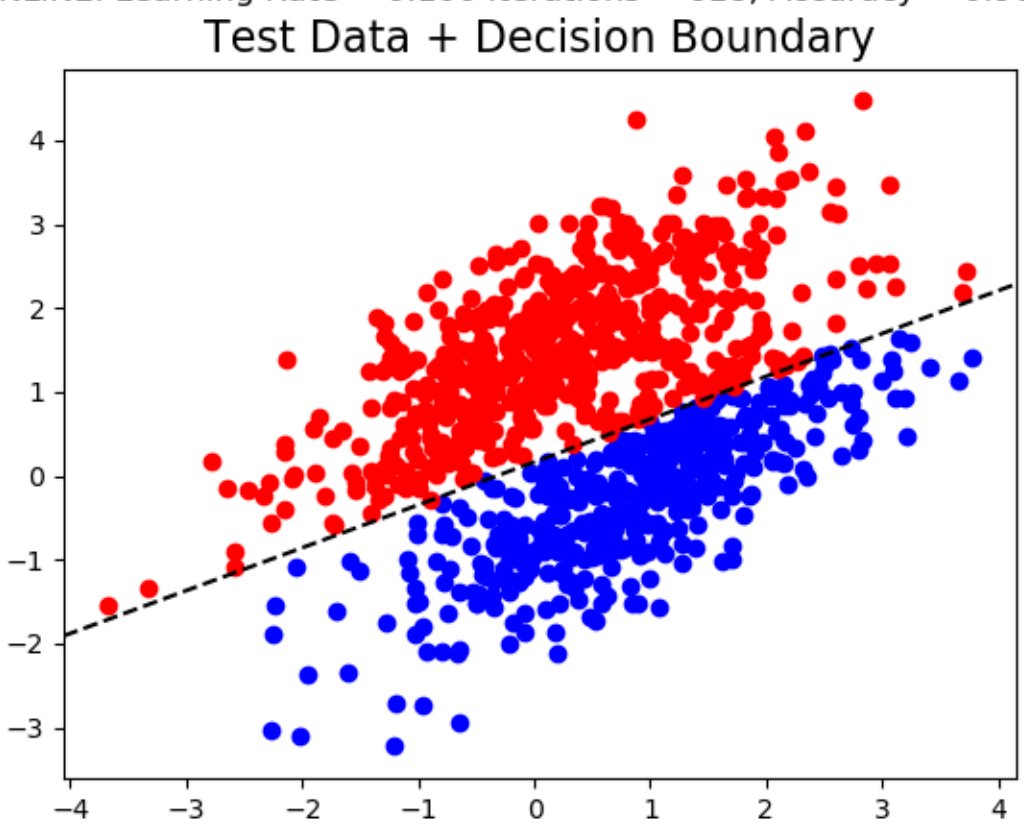
Part 2: Online Training Figures

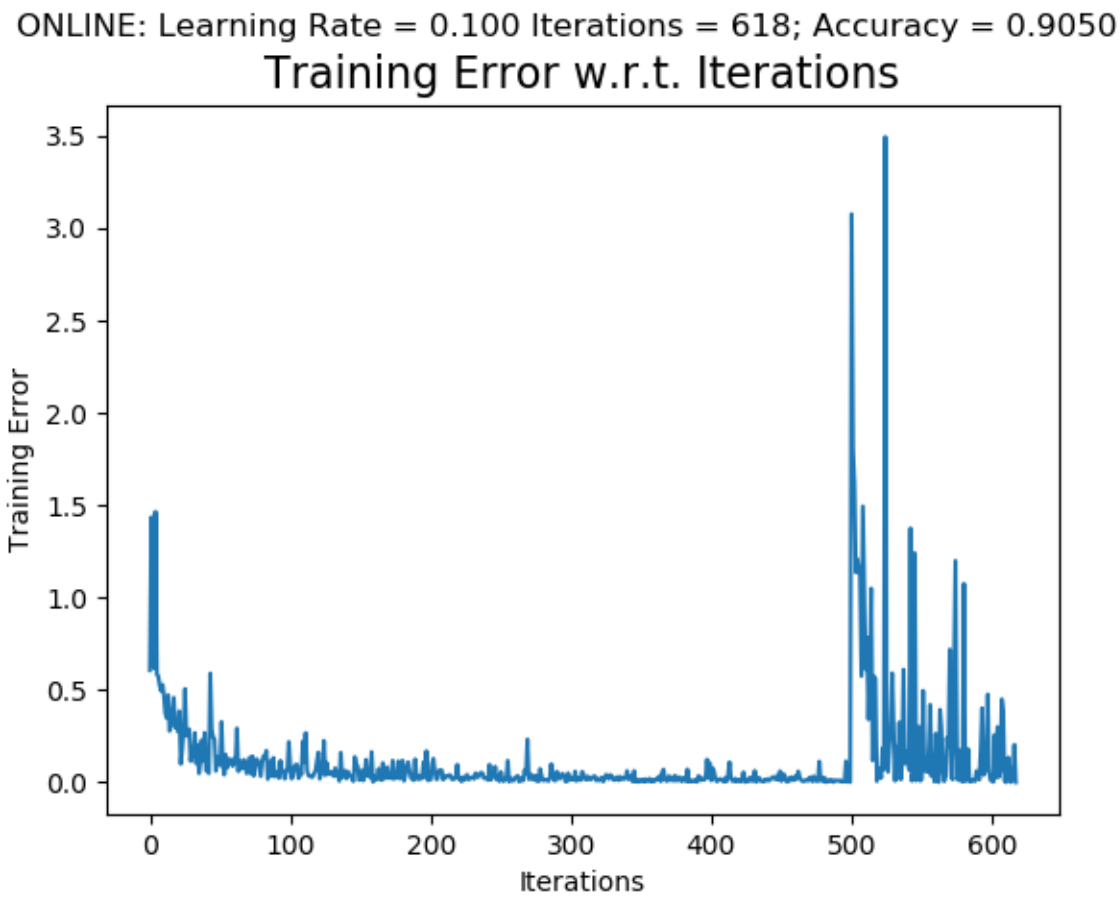



ONLINE: Learning Rate = 1.000 Iterations = 23; Accuracy = 0.6150
Training Error w.r.t. Iterations
Training Error
Iterations
2.5
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20

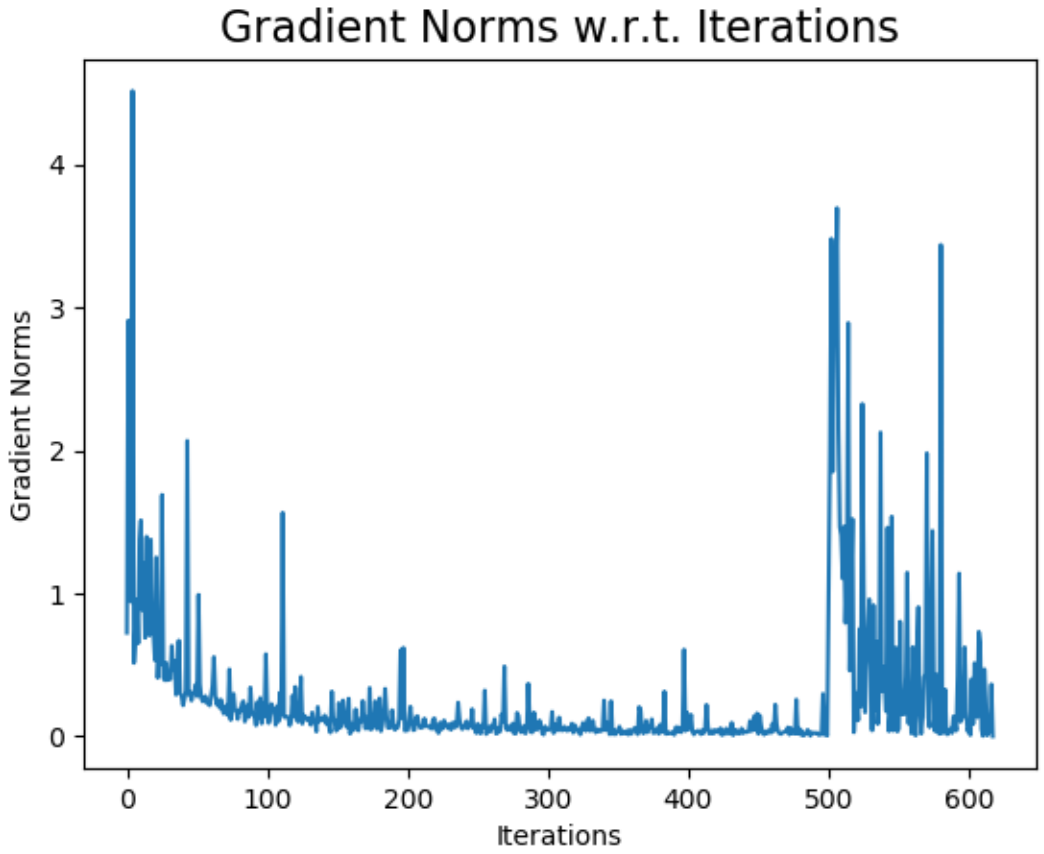

ONLINE: Learning Rate = 1.000 Iterations = 23; Accuracy = 0.6150
Gradient Norms w.r.t. Iterations
Gradient Norms
Iterations
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
5
10
15
20

ONLINE: Learning Rate = 0.100 Iterations = 618; Accuracy = 0.9050

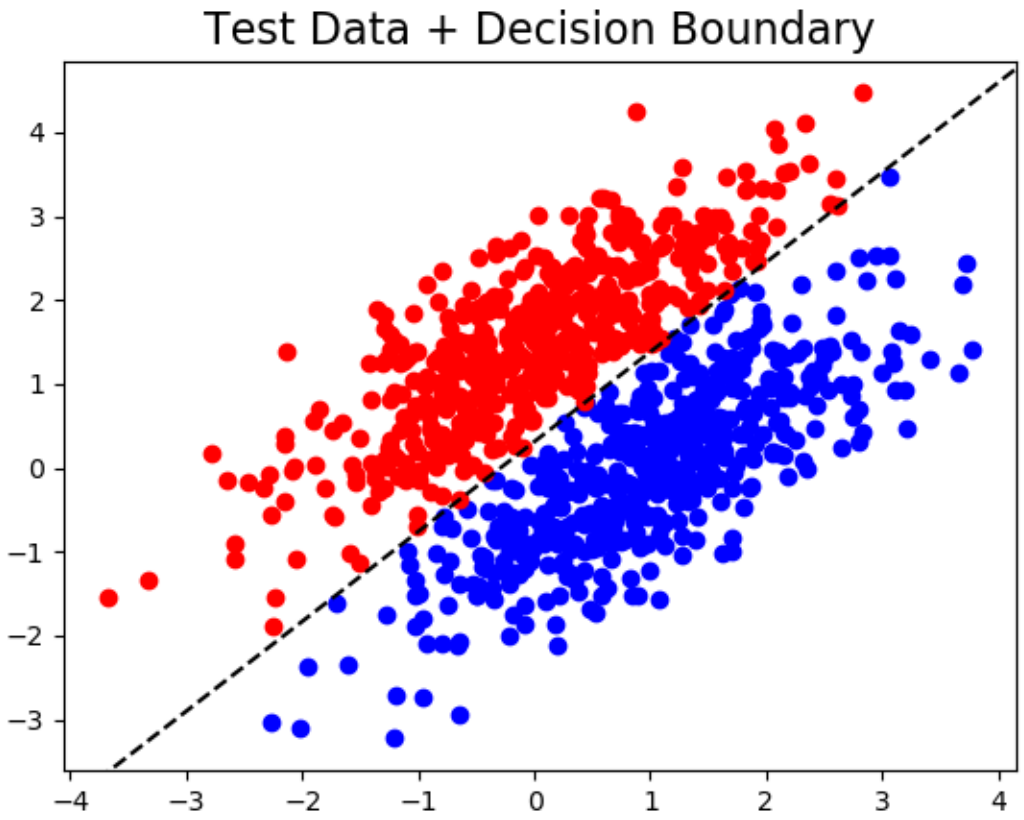



ONLINE: Learning Rate = 0.100 Iterations = 618; Accuracy = 0.9050
Training Error w.r.t. Iterations
Training Error
Iterations
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
100
200
300
400
500
600

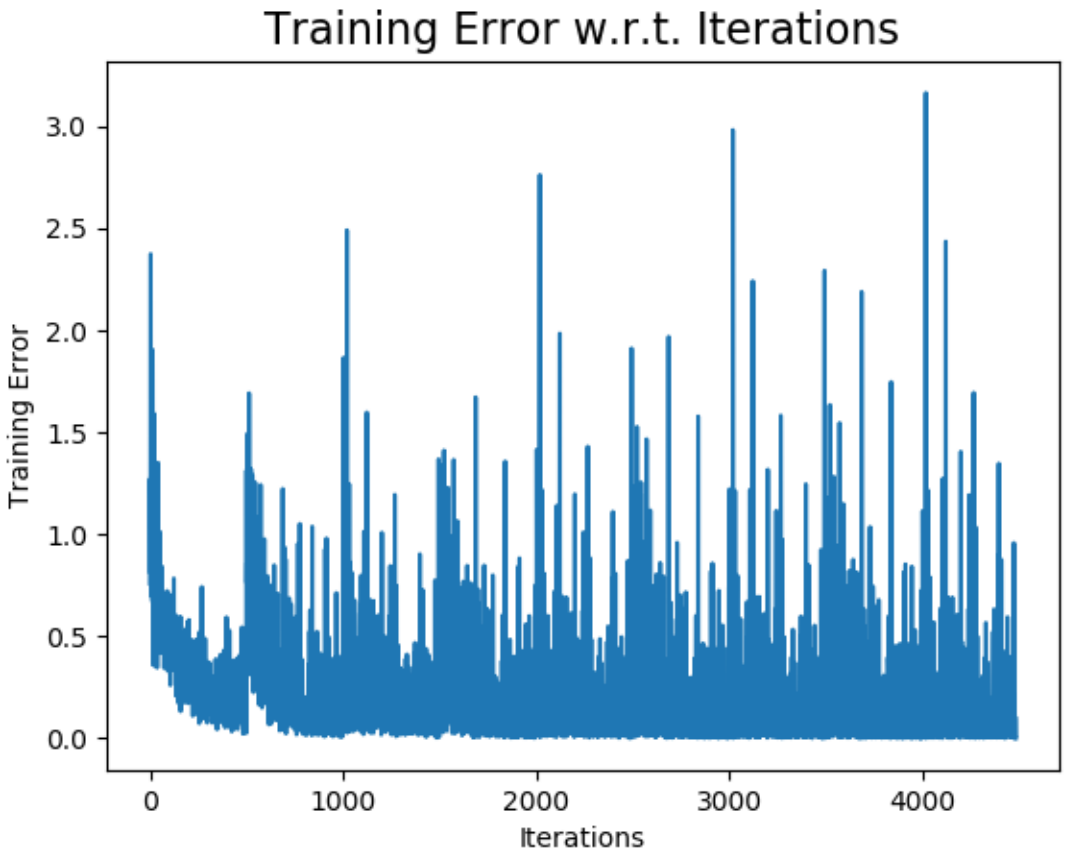
ONLINE: Learning Rate = 0.100 Iterations = 618; Accuracy = 0.9050



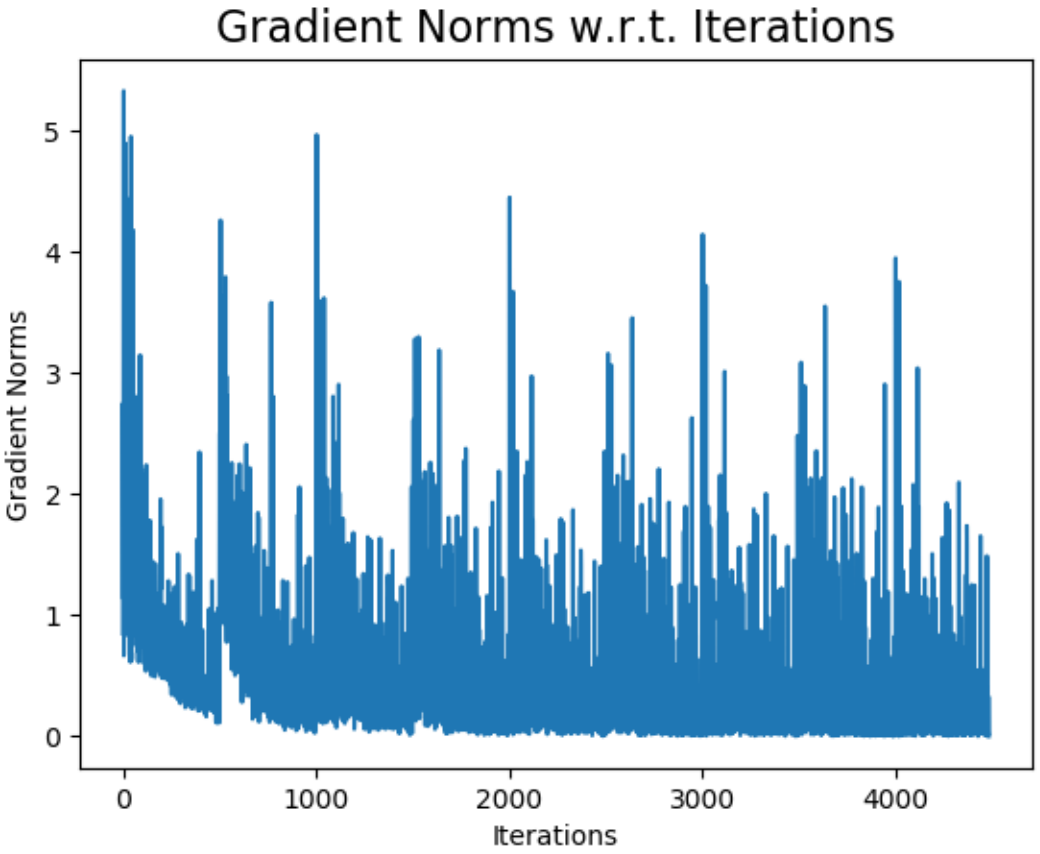
ONLINE: Learning Rate = 0.010 Iterations = 4488; Accuracy = 0.9620



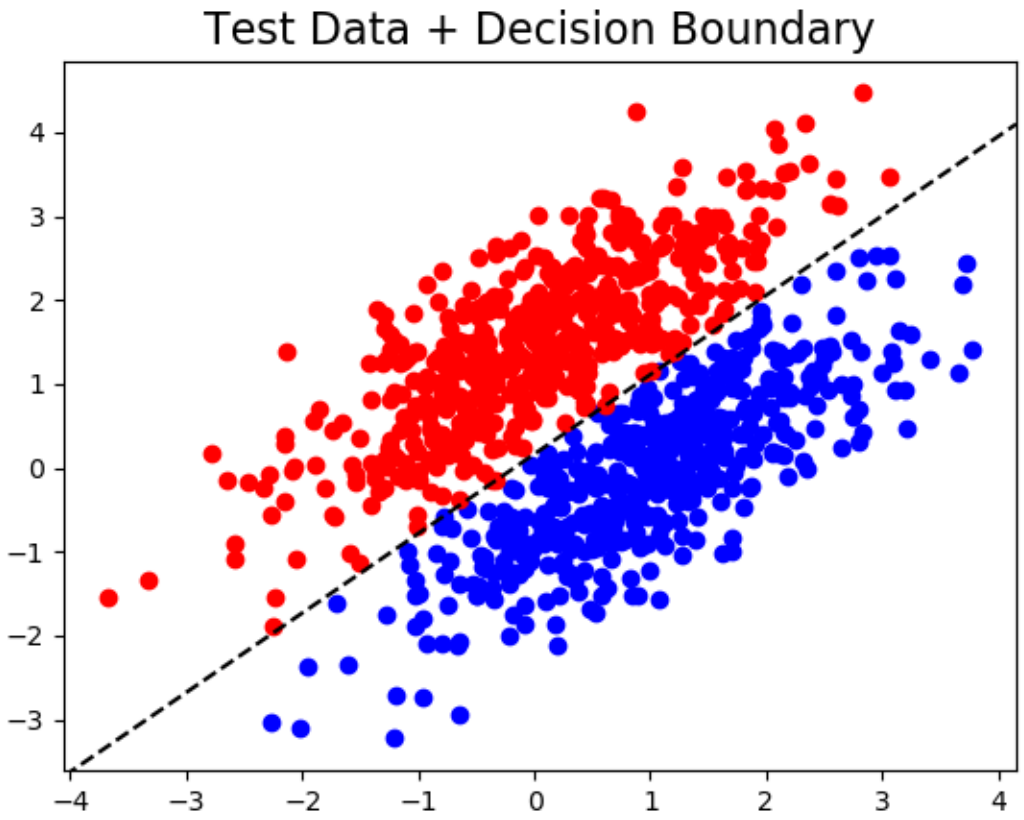
ONLINE: Learning Rate = 0.010 Iterations = 4488; Accuracy = 0.9620



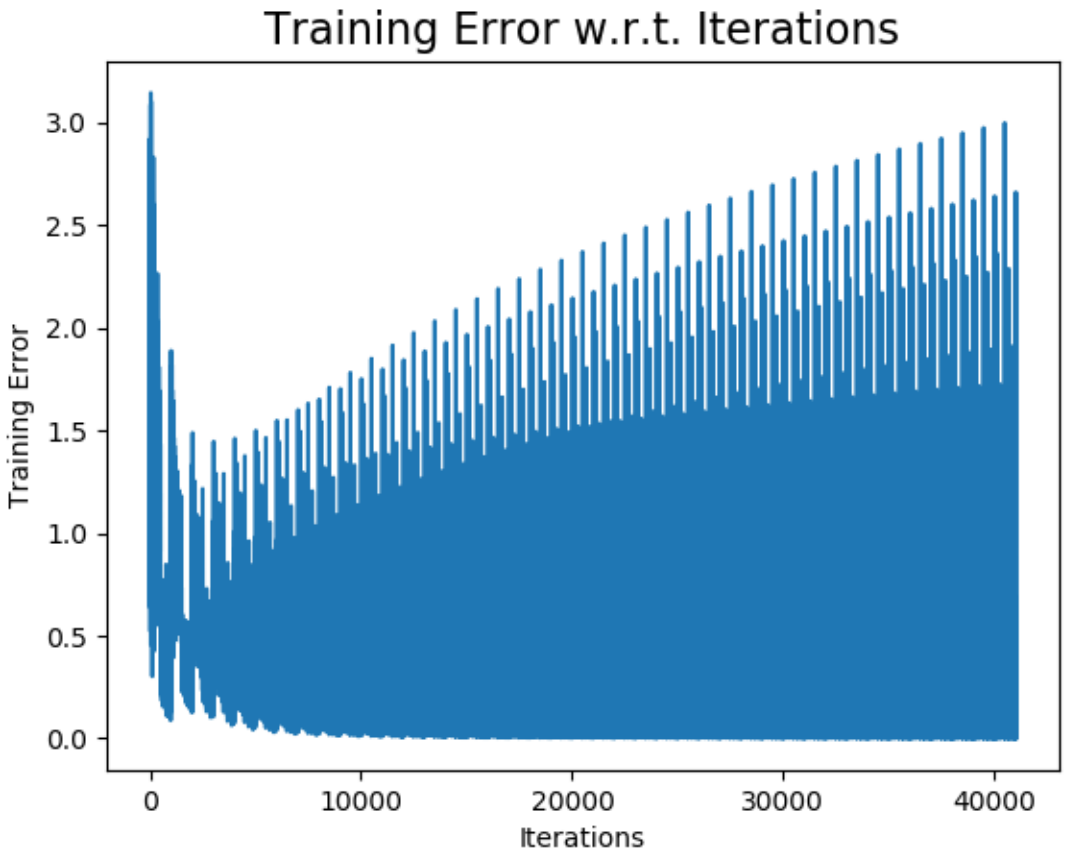
ONLINE: Learning Rate = 0.010 Iterations = 4488; Accuracy = 0.9620



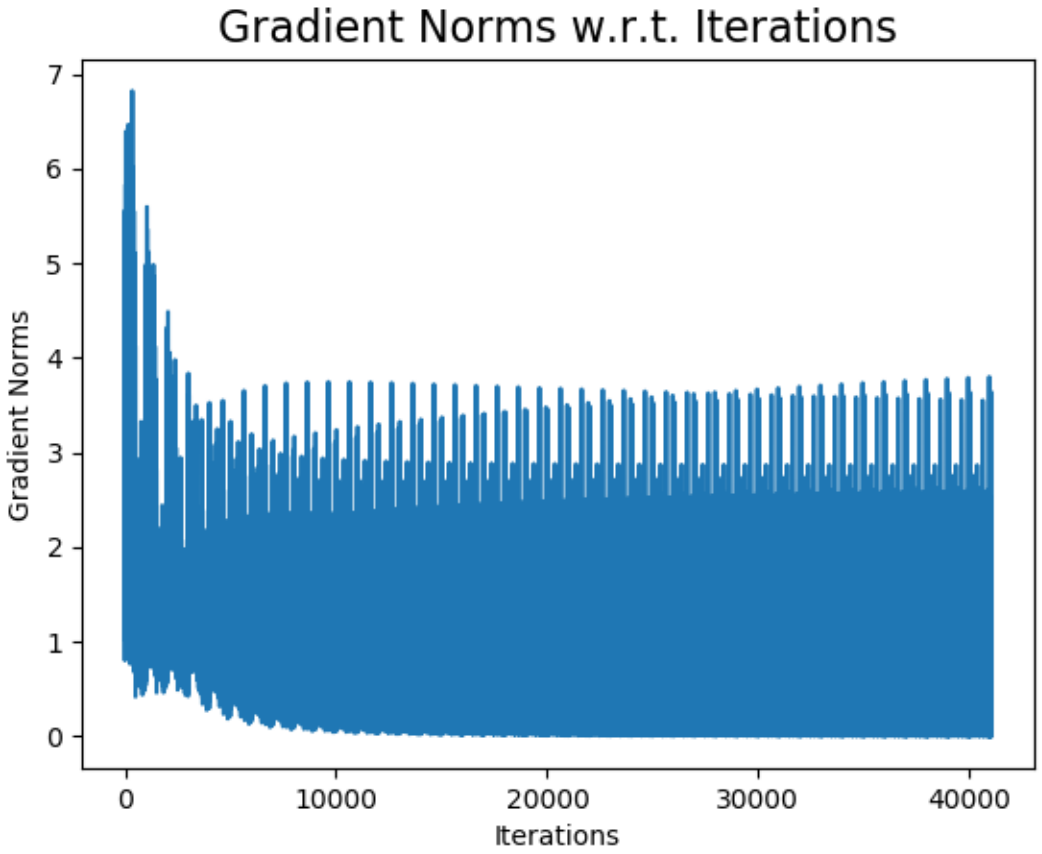
ONLINE: Learning Rate = 0.001 Iterations = 41023; Accuracy = 0.9630



ONLINE: Learning Rate = 0.001 Iterations = 41023; Accuracy = 0.9630



ONLINE: Learning Rate = 0.001 Iterations = 41023; Accuracy = 0.9630



Problem 2:

Part 1:

```
import tensorflow as tf #Import the tensorflow module
mnist = tf.keras.datasets.mnist #Loads the MNIST dataset

(x_train, y_train),(x_test, y_test) = mnist.load_data() #Loads the training and test
data for the MNIST dataset
x_train, x_test = x_train / 255.0, x_test / 255.0 # Pre-processes training and
testing data

model = tf.keras.models.Sequential([    #Instantiates a sequential model
  tf.keras.layers.Flatten(input_shape=(28, 28)), # Flattens data into a column vector
  tf.keras.layers.Dense(512, activation=tf.nn.relu), # Adds a layer with 512 units
using a reLU activation function
  tf.keras.layers.Dropout(0.2), # Applies dropout to the input at a rate of 20%
  tf.keras.layers.Dense(10, activation=tf.nn.softmax) # Adds a layer with 10 units
using a softmax activation function
])
model.compile(optimizer='adam', # Optimizes the model using the Adam algorithm
              loss='sparse_categorical_crossentropy', # Selects the loss function
              metrics=['accuracy']) # Calculates the accuracy of the model

model.fit(x_train, y_train, epochs=5) # Trains the model with 5 iterations
model.evaluate(x_test, y_test) # Returns loss and accuracy values for the model
```

Part 2:

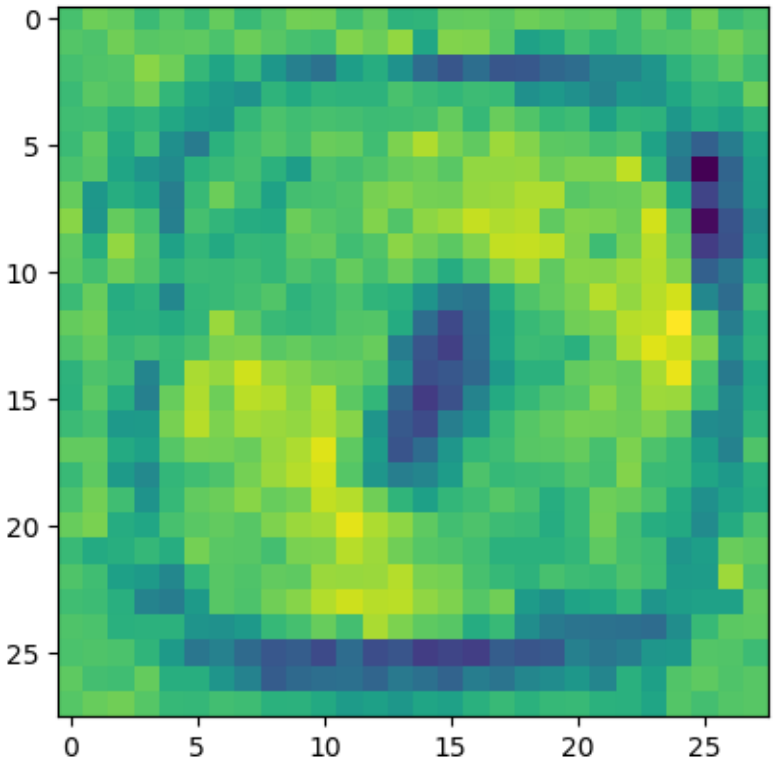
# Hidden Nodes	Testing Accuracy
512	0.9829
128	0.9773
10	0.9148
5	0.8503

Accuracy clearly decreases proportionally with the number of hidden nodes. The increase in accuracy between 5 nodes and 10 nodes (5 difference) is much lower than the increase in accuracy between 128 nodes and 512 nodes (384 difference). The 5-node increase boosted the accuracy by a whole ~6%, whereas the 384-node increase only boosted accuracy by ~1%. This shows that there are diminishing increases in accuracy when you add more nodes.

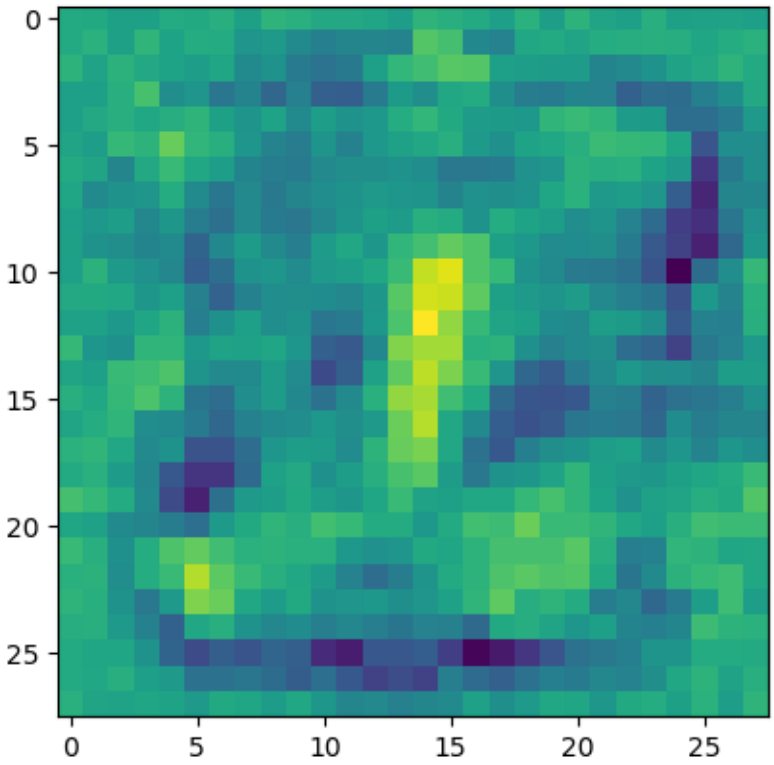
Part 3:

I was able to retrieve the training weights with model.get_weights()[0]. For each output, I the 784 weights that corresponded to it, normalized the data, and displayed it. Here are the results (also included in the "./problem2/figures" folder:

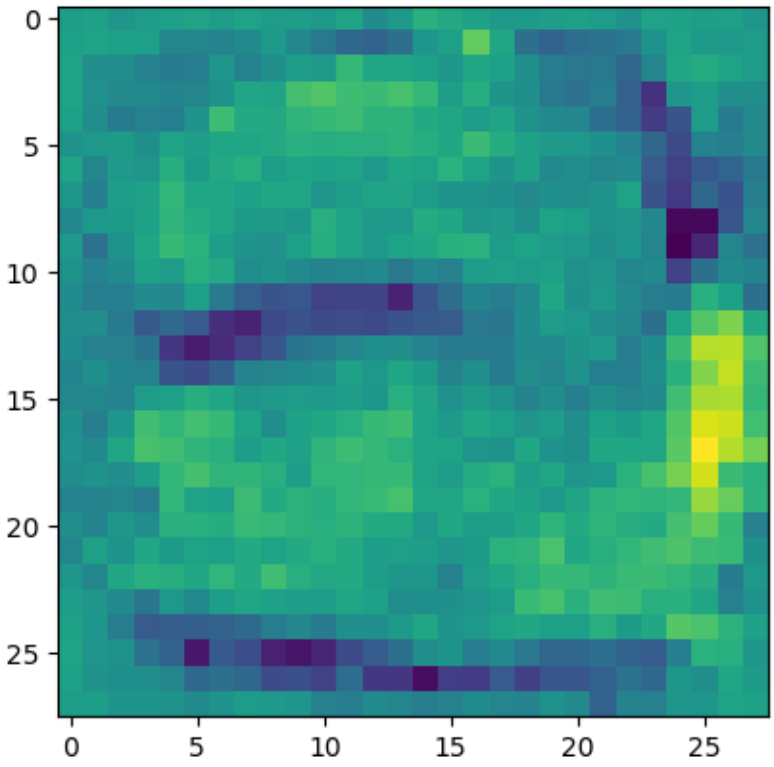
Class 0:



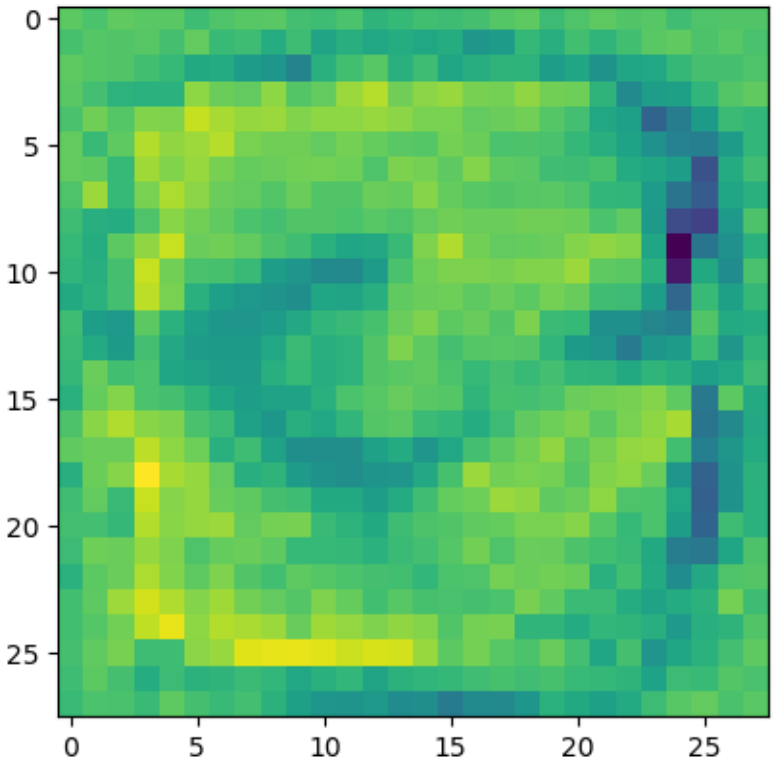
Class 1:



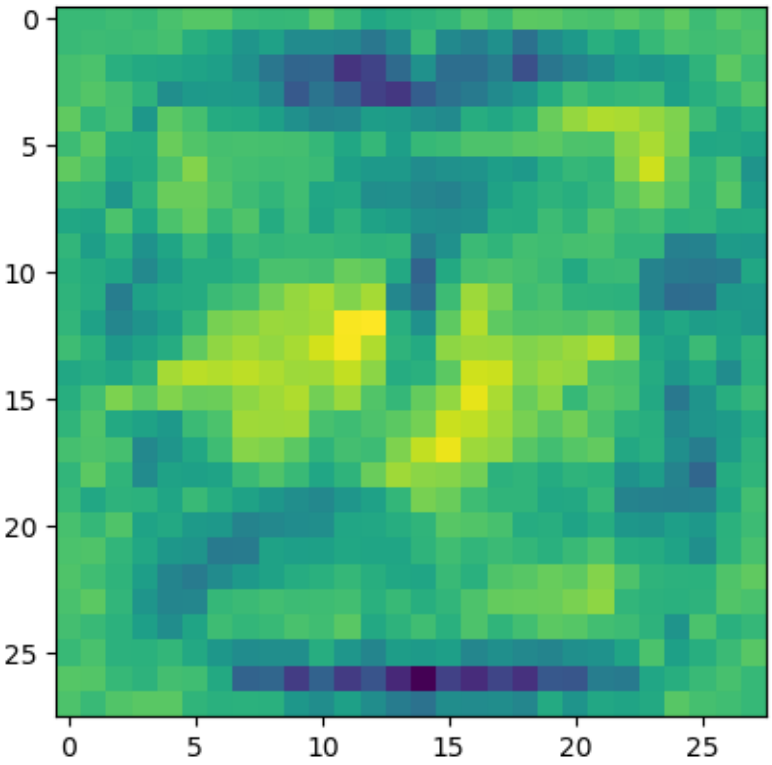
Class 2:



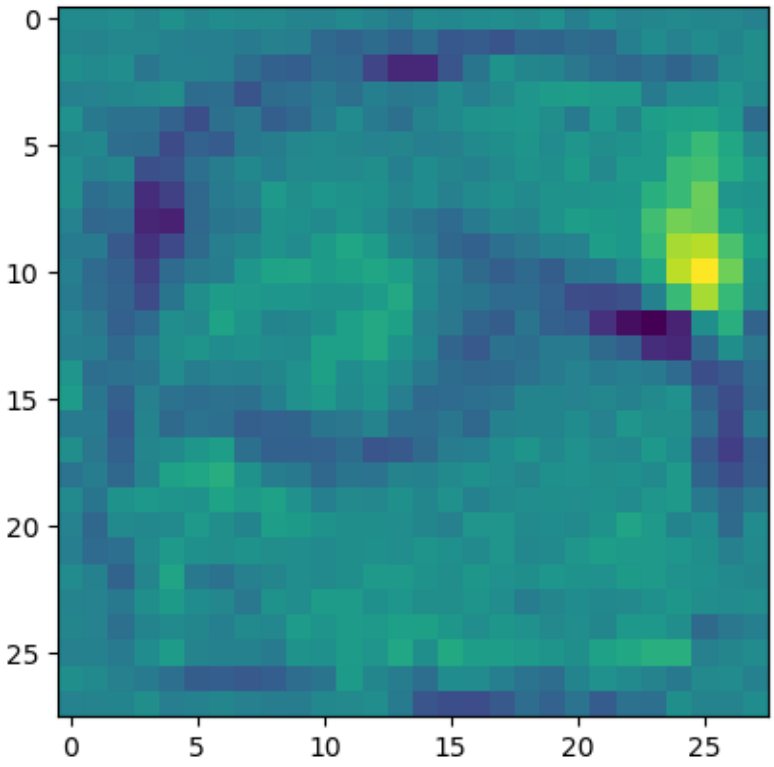
Class 3:



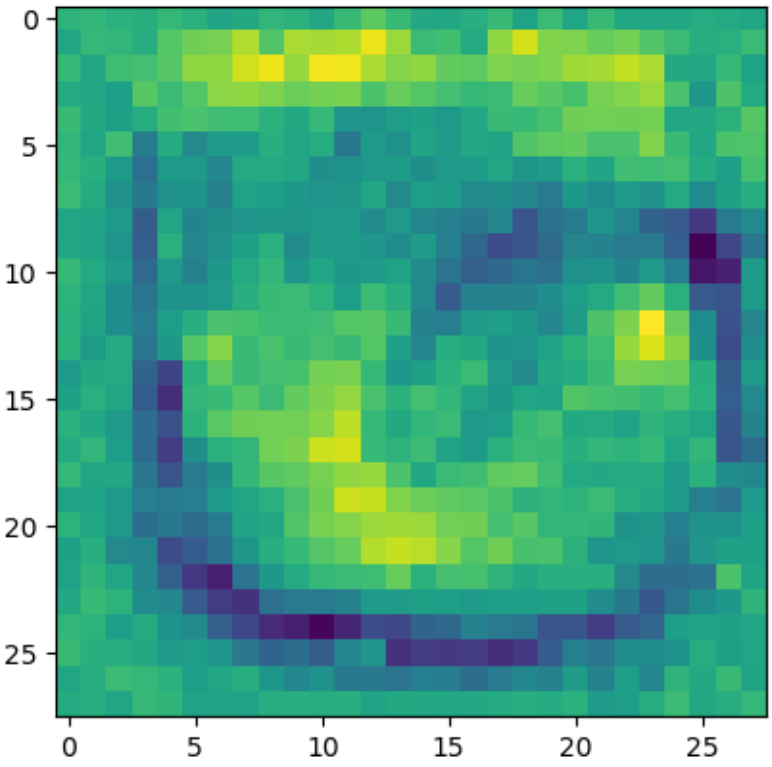
Class 4:



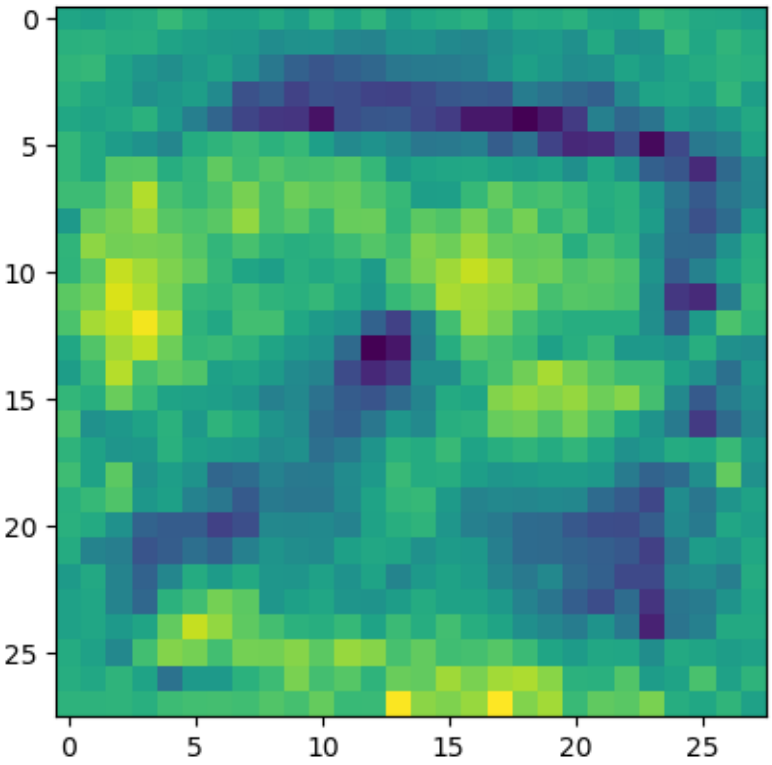
Class 5:



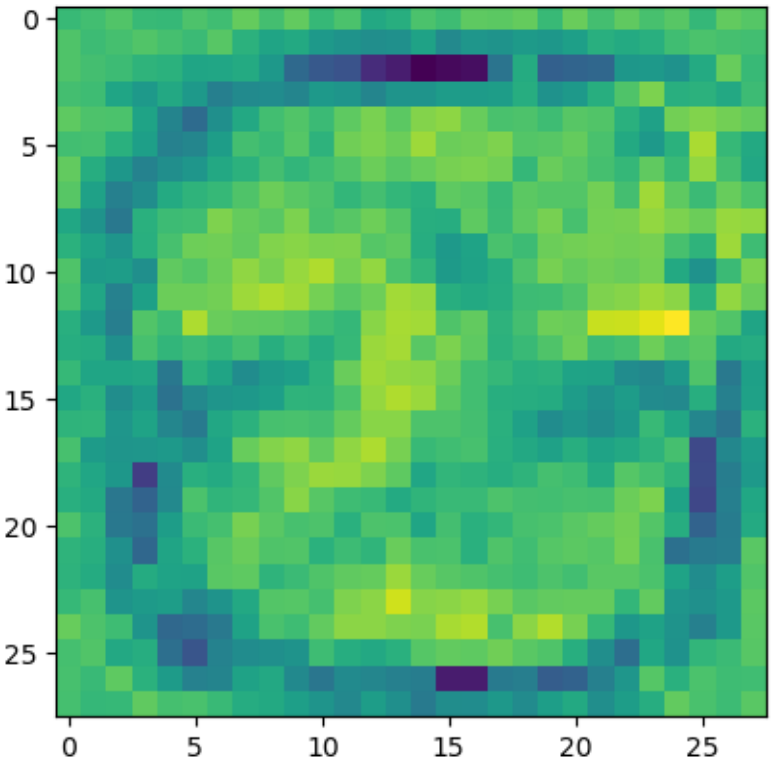
Class 6:



Class 7:



Class 8:



Class 9: