CS440/ECE448

Fall 2016

Assignment 4: Reinforcement Learning and Perceptrons

Section R3

Patrick Wang (3 credit)
Armin Mohammadi (3 credit)
Ryan Chien (3 Credit)

Pwang39, Amohmmd2, Rchien2

Part 1: Q-learning (Pong)

1.1: Single-Player Pong

For this part of the MP, we applied the Q-learning algorithm to train a rational agent to play the game of Pong. We implemented the game using PyGame.

For our learning rate α, which is determined by C/(C+N(s,a)), we used a value of 100 for the constant C. It is clear from the expression above that increasing the value of C will in effect increase the the value of α. We first started with lower values of C and found that increasing the value would improve the consistency of the bot during training. This makes sense because we want the rational agent to be trained by prioritizing more recent information.

For the discount factor γ, we want to make sure that as the ball is approaching the paddle, the reward from the previous hit has been discounted. We noticed that values approaching 0 would have the agent value current rewards more, while values approaching 1 would make the agent value long term rewards more. We used a value of 0.5 for the discount factor because we believe the agent should consider current rewards as well as long-term rewards.

For our exploration function, here is the pseudocode that we tried using:

```
1. For every possible action (stay still, move up, move down):
    a. Determine utility and frequency of the state-action pair
    b. If frequency under a threshold:
        i. Utility = 10000
```

This function forces the agent to take actions that it has not yet explored by setting the utility of those actions to optimistically high values. Our implementation focuses on exploitation, which we believe would be better in the long run. It might take more runs to train the rational agent to achieve an optimal policy, but it will be more accurate. We tried threshold values ranging from 5 to 30 and found that the bot would be stuck repeating the same moves after a certain period of time, regardless of what the threshold value was.

After running around 3000 test games after training for around 110,000 training games, we found our average of ball bounces to be 9.2. The average number of ball bounces undoubtedly increase but fluctuates as the agent is still learning.

In terms of adjusting the discretization settings, there were two major changes that we made. The first adjustment we made was to the resolution of the grid. We scaled everything up, making the grid to be 600x600. This did not make any difference in allowing the agent to to learn a better policy, as all elements were scaled up proportionally as well. Next, we changed the set of possible velocities for the ball after a collision with the paddle. Originally, we scaled up the velocity by multiplying the randomization factor added to the velocity after a collision by 600 to keep it consistent with the rest of the elements. However, we found that the velocities after bouncing off the paddle would increase by too much and the paddle would be unable to move fast enough to keep up as a result. This undoubtedly decreased our average bounces per game by a significant amount; running 3000 test games after training for around 110,000 training games yielded an average of 0.43 ball bounces per game. To fix this, we multiplied the randomization factor by 6 instead of 600, which increased the consistency drastically.

1.2: Two-Player Pong (Extra Credit)

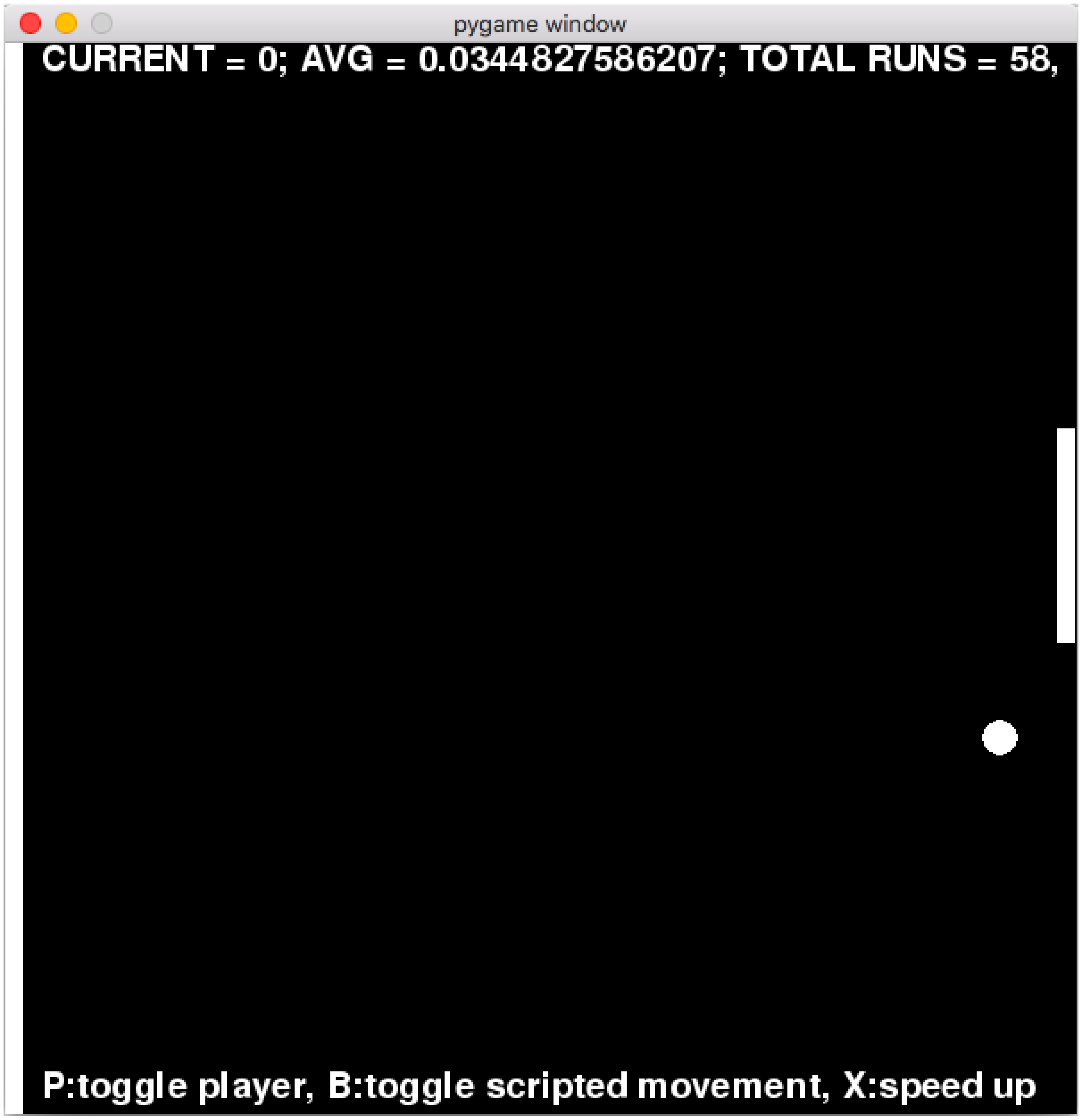
For this part, we had our trained agent play against a hard-coded agent. We did not have to make any changes to the MDP because the trained agent does not care about whether or not the ball passes the hard-coded agent's paddle. Essentially, to the trained agent, the hard-coded agent is just a wall like in part 1.1, except that this wall is smaller and moves according to the current position of the ball.

Implementing this part was rather straightforward. Because in part 1.1, we had the wall on the left coded as a paddle, all we had to do for this part was to change the size of the paddle and have it move according to the position of the ball, which we could determine from the current state.

Due to time constraints, we were unable to have a trained agent with a completely optimal policy to play against a hard-coded agent. However, we noticed that as it first started training, the trained agent was completely unable to beat the hard-coded agent. As more games were played, the trained agent was able to achieve a more optimal policy and was able to bounce the ball back more consistently. As a result, we are confident that after a certain number of test games, that the trained agent will be able to beat the hard-coded agent. One interesting observation we made is that if we made the speed of the hard-coded agent's paddle the same as that of the trained agent, the games tended to be much closer.

Extra Credit:

- **We created a Graphical Interface for the pong game, which lists out features:**



CURRENT = 0; AVG = 0.0344827586207; TOTAL RUNS = 58,

P:toggle player, B:toggle scripted movement, X:speed up

- **We also added the option for a human player to join in. At first, the bot is very easy to beat, but as the training goes on, it progressively becomes harder to beat.**
- **Animations were recorded with the filenames "Stage 1" and "Stage 2" that show our agent improving over time with the average bounces increasing. (Video files attached with source code in zip file)**
- **We also added a speed up button to help you see the training in real time, as well allowing you to play the game at an inhumane frame rate, increasing the complexity of the game**

Part 2: Digit Classification with Machine Learning

2.1: Perceptrons

We applied the multi-class (non-differentiable) perceptron learning rule to try and better classify a set of written numerical digits, 0-9.

Implementation:
To perform the classifications, like in MP3, we first parsed through the training set of images to record the counts of training images (i.e. how many of each label) and get values for the images' features (i.e. 0 for background pixel, 1 for foreground pixel).

Utilizing the multiclass perceptron rule, we trained a set of weight vectors for each digit class, going through the entire training set repeatedly to learn the weight each pixel had for a particular class. We use the decision rule function used for classifying an image of class, **c**, with a value vector, **x**, and some weight vector, **w**, with **c'** as the predicted class.

For every misclassification in the training set, we'd update the weights of both the actual and predicted class weight vectors. This continues until the weights converge in such a way that reduces all training classification errors to zero (note that we also include an extra dummy pixel in the weight vectors to represent the bias for each weight vector, interpreting its corresponding value as 1).

The following page contains our confusion matrix for the results of the perceptron classification:

Analysis of Test Images:
Overall Accuracy: **82.5%**
Confusion Matrix (rates of classification for each label):

		Predicted									
Actual		0	1	2	3	4	5	6	7	8	9
	0	**93.3**	0.0	2.22	0.0	0.0	0.0	2.22	0.0	2.22	1.11
	1	0.0	**98.2**	0.0	0.0	0.9	0.0	0.9	0.0	0.0	0.0
	2	0.0	2.91	**79.61**	3.88	1.94	0.0	2.91	3.88	3.88	0.97
	3	0.0	0.0	2.0	**82.0**	0.0	8.0	3.0	4.0	1.0	0.0
	4	0.0	0.0	1.87	0.93	**83.2**	0.0	2.8	2.9	0.93	7.48
	5	2.17	0.0	2.17	5.43	0.0	**75.0**	1.09	3.26	7.61	3.26
	6	1.1	1.1	2.2	0.0	2.2	1.1	**89.0**	2.2	1.1	0.0
	7	0.94	2.83	3.77	1.98	2.83	0.0	0.0	**78.3**	0.94	8.49
	8	0.0	1.94	4.85	8.74	2.91	4.85	2.91	0.97	**66.99**	5.83
	9	0.0	0.0	1.0	4.0	7.0	1.0	0.0	6.0	1.0	**80.0**

From the above confusion matrix, the classification rates of each label are in **bold** across the diagonal, and the top four "confused" classification pairs are highlighted in red.

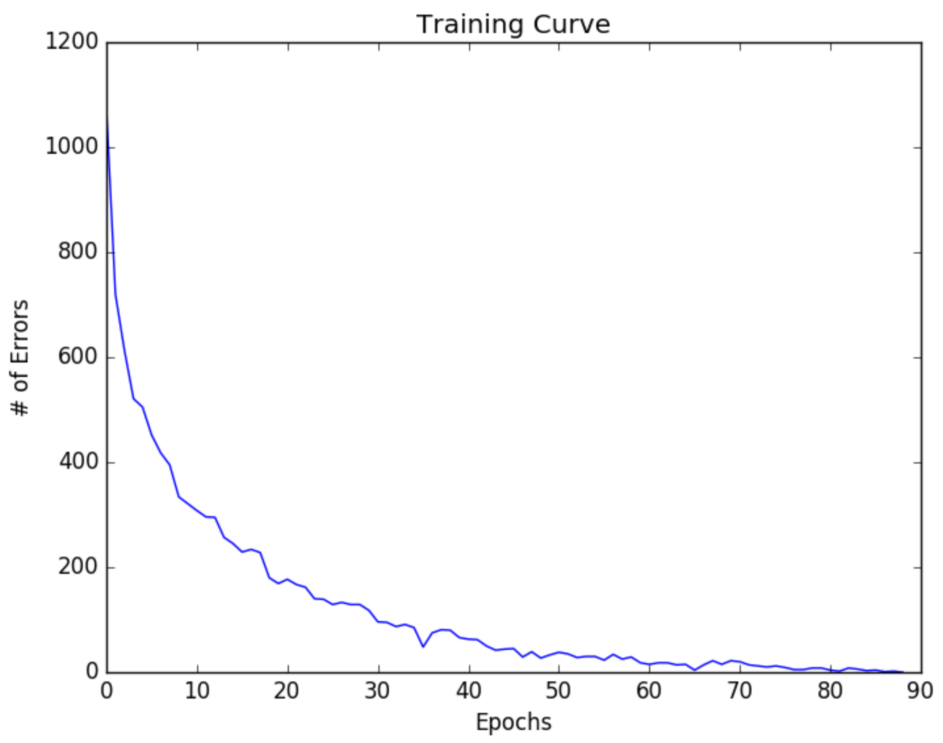
Parameters:
The following are the various values we tried for each parameter, with the best ones in bold:

Learning Decay: α = 20/(20+t), α = 200/(200+t), α = 150/(150+t), **α = 100/(100+t)**
Bias: None, Random, **Zero at Start**
Initial Weights: Random, **Zero at Start**
Training Order: Random, **Fixed**
Number of Epochs (for weight training): 20, 70, 86, **93**

Overall, **α = 100/(100+t)** worked best for our learning decay, since it was a good balance between accuracy and performance (i.e. finishing training before epoch limit), while adding bias greatly increased our accuracy, and randomizing weights, bias, and training orders were somewhat unpredictable, but mainly underperforming.

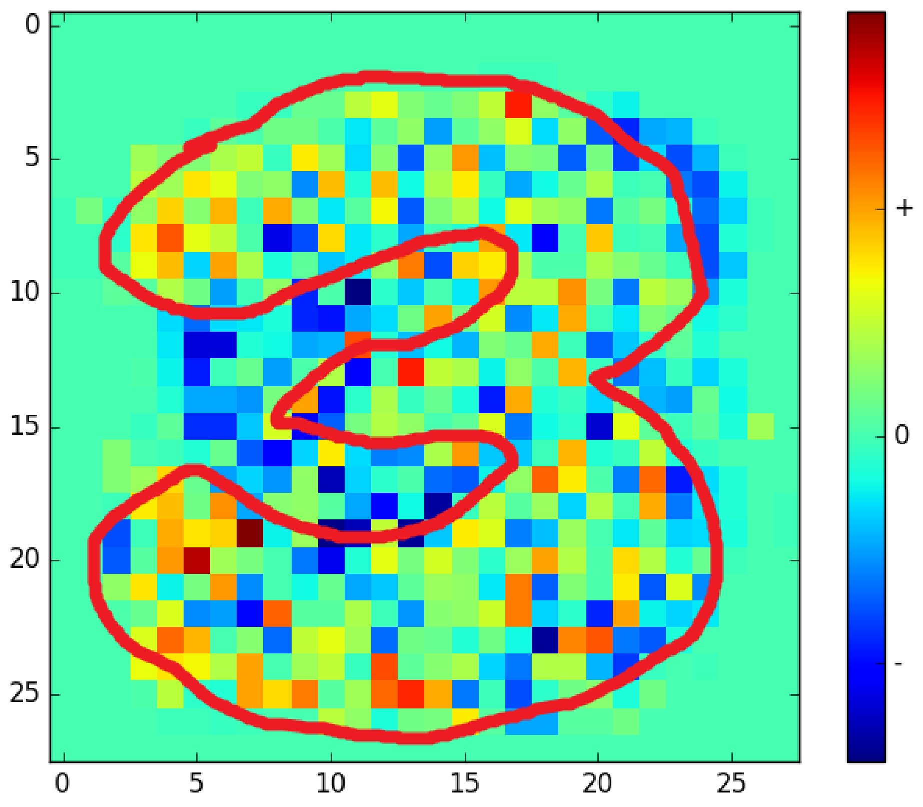
Comparison to Naive Bayes:
The perceptron classification method nicely outperformed the Naive Bayes classifier (from 77.3% to 82.5%), likely because the weights of the pixels for the perceptron method better represent the influence of pixels in classification (i.e. foreground pixels take more precedence). Interestingly enough, the least accurate individual classes were the same as before, although all accuracy magnitudes were raised overall.

Training Curve:
The following is the training curve for our perceptron classifier, showing the improvement of weight training over the number of epochs:



Extra Credit (Weight Visualization):

The following is a heatmap of the weight vector for class 3, which shows that, for the most part, the weight of the foreground pixels are greater than the background pixels (with some variations). The redder the pixel, the higher the weight, the bluer the pixel, the lower the weight:



Extra Credit (Ternary Features):

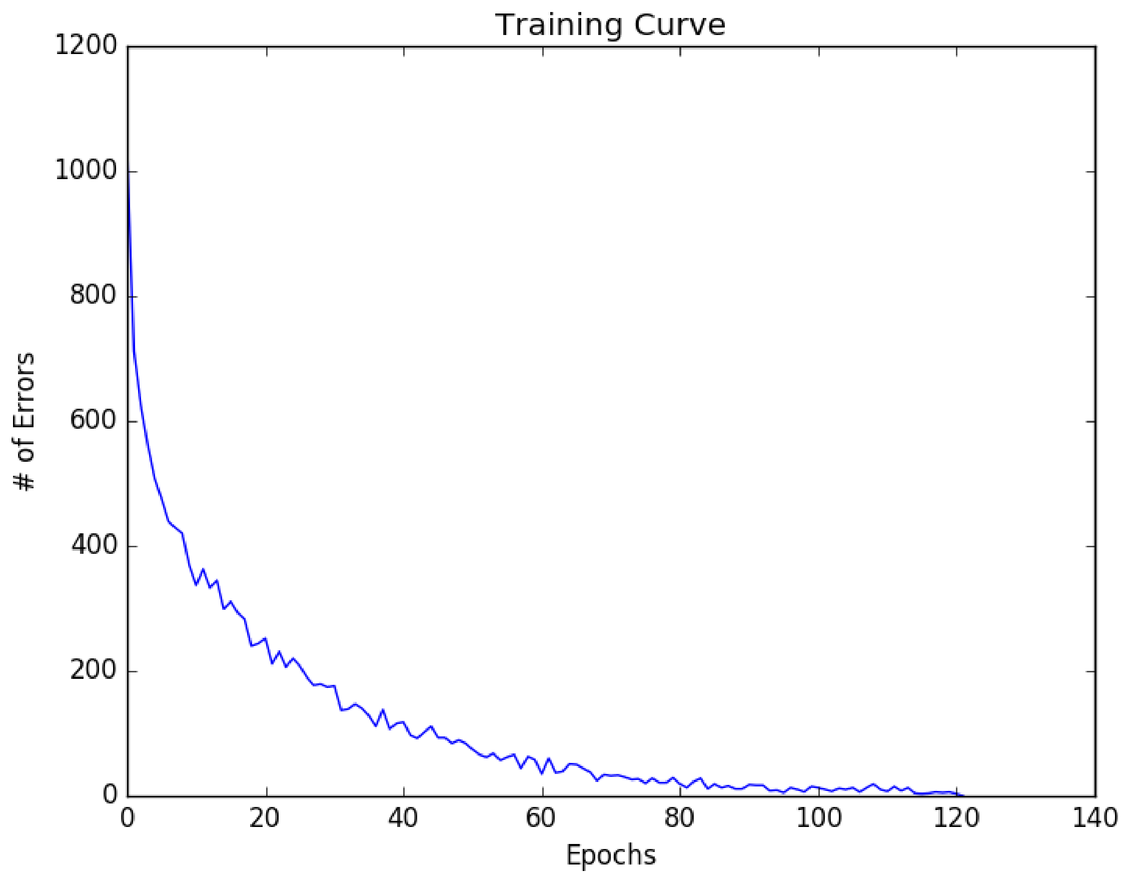
As with the Naive Bayes classifier, we look to now using ternary values (foreground, middleground, and background pixels) in an attempt to better represent the images and hopefully get better classification accuracy. The following is our new confusion matrix:

Analysis of Test Images:
Overall Accuracy: **82.6%**
Confusion Matrix (rates of classification for each label):

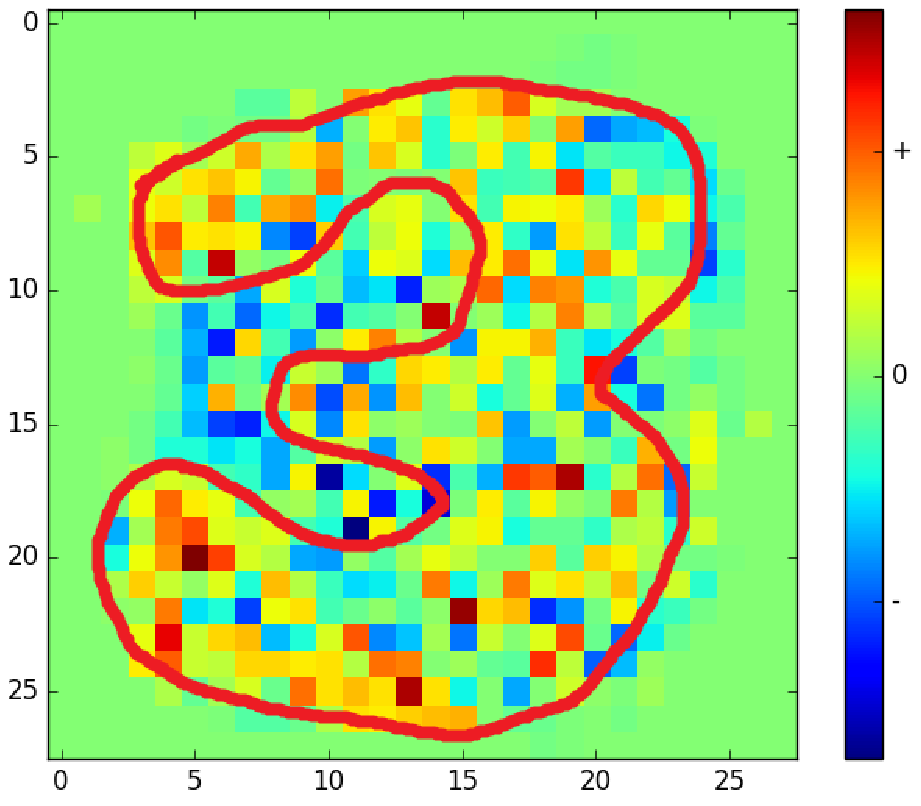
		Predicted									
Actual		0	1	2	3	4	5	6	7	8	9
	0	**93.3**	0.0	3.3	0.0	0.0	0.0	2.2	0.0	0.0	1.1
	1	0.9	**96.3**	0.9	0.0	0.9	0.0	0.9	0.0	0.0	0.0
	2	0.0	1.9	**81.5**	2.9	0.9	0.9	2.9	3.8	4.8	0.0
	3	1.0	1.0	4.0	**78.0**	0.0	6.0	0.0	4.0	4.0	2.0
	4	0.0	0.9	0.0	0.9	**85.1**	0.0	4.6	0.9	0.0	7.4
	5	1.1	0.0	2.1	6.5	1.1	**76.1**	4.3	1.1	6.52	1.1
	6	0.0	1.1	1.1	0.0	2.2	1.1	**91.2**	1.1	2.2	0.0
	7	0.9	2.8	3.7	1.8	0.9	0.9	0.0	**74.5**	0.0	14.1
	8	0.9	2.9	2.9	7.8	3.8	5.8	3.8	0.9	**66.0**	4.8
	9	0.0	0.0	0.0	5.0	4.0	4.0	0.0	3.0	1.0	**83.0**

As with the Naive Bayes classifier, the accuracy does improve overall with ternary features. The only issues that arose here with the perceptrons was that our original training function was not sufficient enough for training to complete before the maximum epoch, so for this scenario, we used **α = 150/(150+t)**, and it took 124 epochs to finish.

Extra Credit (Ternary Training Curve):
The following is the training curve for our ternary perceptron classifier, showing the improvement of weight training over the number of epochs. While the new features are more representative of the images, more training is needed overall to utilize these better results:



Extra Credit (Ternary Weight Visualization):
The following is a heatmap of the weight vector for class 3. It appears to be more defined than the binary weight vector, likely due to the greater specificity in image values with the ternary design:



Extra Credit (Differentiable Classifier):

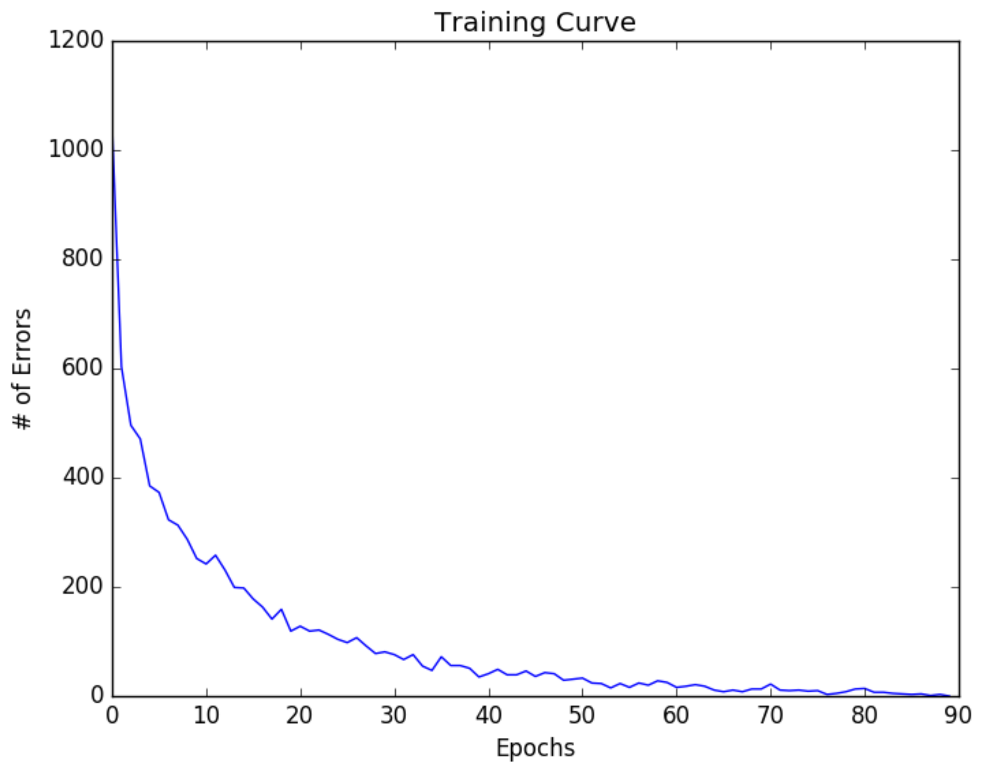
We looked to try implementing a differentiable classifier with its modified update rule that utilizes the sigmoid function in order to see the results on the original binary-value perceptron classification. The following is our new confusion matrix:

Analysis of Test Images:
Overall Accuracy: **81.7%**
Confusion Matrix (rates of classification for each label):

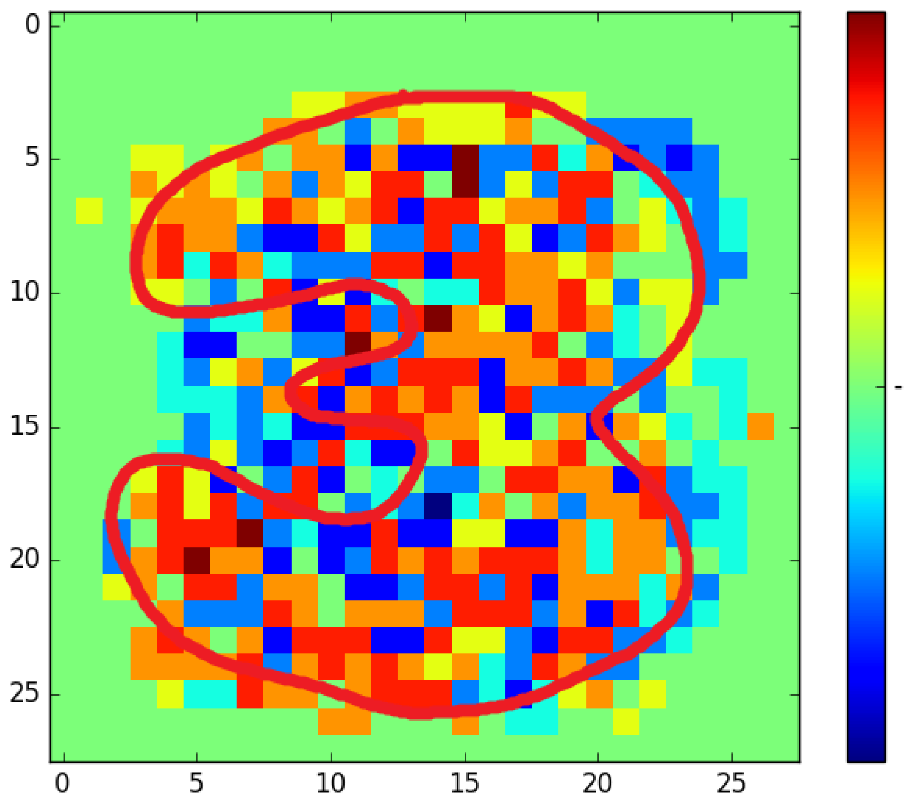
		Predicted									
		0	1	2	3	4	5	6	7	8	9
Actual	0	**93.3**	0.0	1.1	0.0	0.0	2.2	2.2	0.0	1.1	0.0
	1	0.0	**97.2**	0.9	0.0	0.9	0.0	0.9	0.0	0.0	0.0
	2	0.0	1.9	**75.7**	10.7	3.8	0.0	2.9	0.9	3.8	0.0
	3	0.0	0.0	0.0	**80.0**	0.0	9.0	2.0	5.0	2.0	0.0
	4	0.9	0.0	1.8	0.0	**83.2**	0.0	1.8	2.8	0.9	8.4
	5	1.1	0.0	1.1	8.7	0.0	**76.1**	3.3	2.1	4.3	3.2
	6	1.1	1.1	2.2	0.0	2.2	2.2	**89.0**	0.0	2.2	0.0
	7	0.0	2.8	3.7	2.9	1.9	0.9	0.0	**76.4**	0.94	10.4
	8	0.9	2.9	3.8	9.7	3.8	3.8	2.9	2.9	**65.1**	3.8
	9	0.0	0.0	2.0	6.0	7.0	2.0	0.0	3.0	1.0	**79.0**

Our configurations were the same as the original non-differentiable binary-value perceptron, but training finished slightly faster, after only 89 epochs instead of 93. Interestingly, though, overall accuracy decreased a bit, along with some individual accuracies, with even some changes in pairs of misclassifications.

Extra Credit (Differentiable Training Curve):
The following is the training curve for our ternary perceptron classifier, showing the improvement of weight training over the number of epochs. It trains slightly faster than the non-differentiable classifier:



Extra Credit (Differentiable Weight Visualization):
The following is a heatmap of the weight vector for class 3. Some parts are clearer than the non-differentiable parts, but weights seem a bit more messier than before, possibly leading to more inaccuracy:



Work Distribution
Patrick/Ryan/Armin: 33.3%/33.3%/33.4%
Patrick/Ryan-Focused on part 1
Armin-Focused on part 2