CS 440 MP4
Section Q4
4 Credits

Christian Howard
howard28@illinois.edu

Luke Pitstick
pitstck2@illinois.edu

Abstract

Within this assignment, areas of supervised and reinforcement learning were investigated. In the first part of the assignment, digit classification was revisited using Multi-Class perceptrons and k-Nearest Neighbor classification methods. Both methods proved to generate models with great testing accuracy. For the second part of the assignment, Q Learning was used to teach a set of agents to play Pong with a set of differing goals. The first agent was tasked to volley the ball off of a wall on the opposite side of the domain as many times as possible, achieving about 14.4 volleys on average per game with the setup developed. The second agent was tasked to play against and beat an agent with a predefined strategy. After sufficient training, the Q Learning agent was found to have a win rate of just over 97% against the predefined agent. Software was developed to watch the agents play and allow humans to play. It was found that while the Q Learning agent performed well against the predefined agent, human players appeared to find holes in the Q Learning strategy that made the Q Learning agent play sub-optimally against humans.

Contents

1 Digit Classification using Discriminative Machine Learning Methods

1.1 Digit Classification with Perceptrons

First digit classification is done with a multi-class perceptron. Each pixel is considered a feature with 784 in total. First the program reads all the digits into the system in a binary format. It then passes through all the training examples and obtains a result $y_c = w \cdot x$ where y_c is the result for a single class and $w \cdot x$ is the dot product of the weight and feature vectors. Then the maximum result is selected ass the temporary guess for that round. If correct, the weights are left unchanged. If incorrect the expected class is updated as $w_{cc} = w_{cc} + \alpha x$ and the guessed class as $w_{cg} = w_{cg} - \alpha x$. Alpha is the learning rate which decays over time. This process is repeated multiple times (epochs) over the same training set.

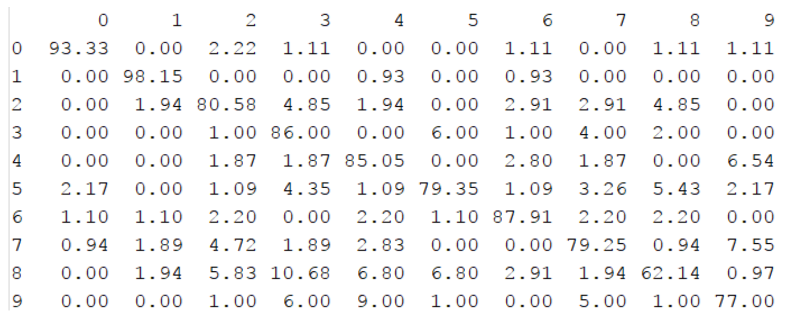
Once the weights have been trained the same guessing procedure is done just once on each example in the testing set, and the results are recorded. The best settings that were selected were with bias, initial weights as random, random training order, and inverse decay. The selection of these is discussed later. These were then tested over a range of epochs to see the effect on overall accuracy. Figure 1 contains these results.

Epochs	Accuracy	Runtime (s)
1	76.9%	0.4
2	71.7%	0.6
3	82.2%	0.8
4	80.3%	1.0
5	82.7%	1.4
10	83.1%	3.9
15	82.5%	3.3
20	80.9%	4.2
30	82.4%	5.8
40	82.1%	8.5
50	82.9%	10.9
100	82.3%	21.1
200	81.9%	42.4
500	82.4%	102.9

Table 1: Classification Accuracy by Number of Epochs

We can see that accuracy improves early on with the increase in number of epochs. We see a leveling off after 30 epochs around the accuracy of 82%. The highest values being at 10 epochs is likely due to a random variation in the weights that made it get some lucky guesses. We decided to use 50 epochs for all further test because it is after the leveling point with good results. This also gives us the confusion in figure 2 which shows how often digits from the rows on the left were identified as the classes from the row on the top.

Runtime scaled linearly to epochs as expected. It took roughly epochs/5 in seconds to run.

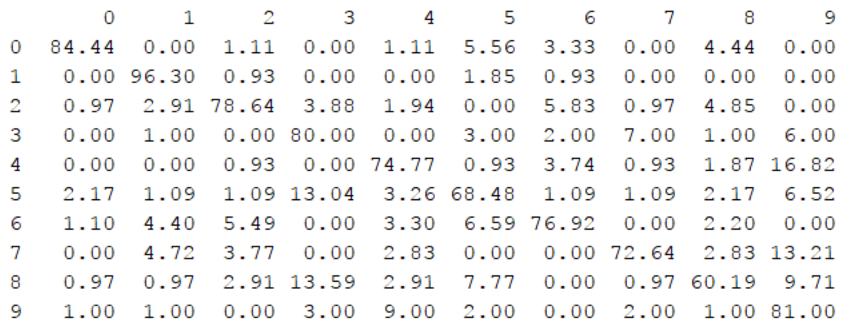


	0	1	2	3	4	5	6	7	8	9
0	93.33	0.00	2.22	1.11	0.00	0.00	1.11	0.00	1.11	1.11
1	0.00	98.15	0.00	0.00	0.93	0.00	0.93	0.00	0.00	0.00
2	0.00	1.94	80.58	4.85	1.94	0.00	2.91	2.91	4.85	0.00
3	0.00	0.00	1.00	86.00	0.00	6.00	1.00	4.00	2.00	0.00
4	0.00	0.00	1.87	1.87	85.05	0.00	2.80	1.87	0.00	6.54
5	2.17	0.00	1.09	4.35	1.09	79.35	1.09	3.26	5.43	2.17
6	1.10	1.10	2.20	0.00	2.20	1.10	87.91	2.20	2.20	0.00
7	0.94	1.89	4.72	1.89	2.83	0.00	0.00	79.25	0.94	7.55
8	0.00	1.94	5.83	10.68	6.80	6.80	2.91	1.94	62.14	0.97
9	0.00	0.00	1.00	6.00	9.00	1.00	0.00	5.00	1.00	77.00

Figure 1: Perceptron Confusion Matrix

As mentioned above, the overall accuracy obtained with optimal settings is 82.9%. The bias was implemented by having a feature dimension that was a constant 1 for all images. This helps account for the overall frequency that a class appears. Without a bias accuracy is 82.5%, so using bias performs better. The inital weights could be set to either 0 or a random range between -10 and 10. When set to 0 overall accuracy was 81.5%, so random weights were used. Before each epoch the training examples could remain in a static order, or be shuffeled each time. Without shuffling the accuracy is 81.7%, so shuffling is used for the best results. Finally the best decay function found was $\alpha = epochs/(epochs + t)$ where t is the current epoch. This is an inverse decay. We also tested exponential decay $\alpha = e^{-t/epochs}$ with an accuracy of 81.9%, and linear decay $\alpha = (-t/epochs) + 1$ with an accuracy of 81.5%.

The single pixel Naive Bayes approach gave an overall accuracy of 77.5%, so clearly the perceptron approach was much more accurate. The combinations with the highest confusion rates were nearly the same. The top 3 for both approaches are (8,3), (7,9), (4,9). But the Naive Bayes had (5,3) in the top four, while the perceptron had (3,5) instead. The perceptron also did better for each individual accuracy, except for 9's where Naive Bayes outperformed it. These previous results are in Figure 2.



	0	1	2	3	4	5	6	7	8	9
0	84.44	0.00	1.11	0.00	1.11	5.56	3.33	0.00	4.44	0.00
1	0.00	96.30	0.93	0.00	0.00	1.85	0.93	0.00	0.00	0.00
2	0.97	2.91	78.64	3.88	1.94	0.00	5.83	0.97	4.85	0.00
3	0.00	1.00	0.00	80.00	0.00	3.00	2.00	7.00	1.00	6.00
4	0.00	0.00	0.93	0.00	74.77	0.93	3.74	0.93	1.87	16.82
5	2.17	1.09	1.09	13.04	3.26	68.48	1.09	1.09	2.17	6.52
6	1.10	4.40	5.49	0.00	3.30	6.59	76.92	0.00	2.20	0.00
7	0.00	4.72	3.77	0.00	2.83	0.00	0.00	72.64	2.83	13.21
8	0.97	0.97	2.91	13.59	2.91	7.77	0.00	0.97	60.19	9.71
9	1.00	1.00	0.00	3.00	9.00	2.00	0.00	2.00	1.00	81.00

Figure 2: Naive Bayes Confusion Matrix

1.2 Digit Classification with Nearest Neighbor

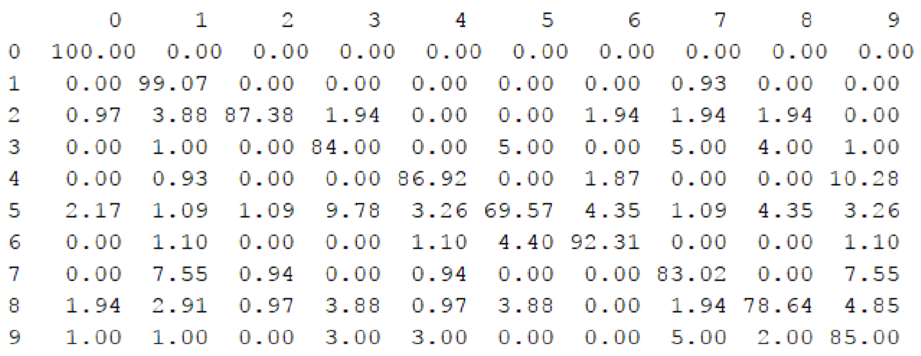
We performed the same image classification but using the k-nearest neighbors approach. This approach uses a distance function to find the k nearest training examples, and is classified as the most frequent one. My distance function is simply $d = abs(FA_{11} - FB_{11}) + abs(FA_{11} - FB_{11}) + ... + abs(FA_{11} - FB_{11})$. This is the sum of the absolute values of the differences between the features at each index on each image.

k	Accuracy	Runtime (s)
1	90.2%	22.2
2	89.3%	22.2
3	89.0%	21.0
4	89.6%	21.4
5	90.1%	22.1
10	88.8%	22.2
20	88.8%	22.2
50	88.6%	22.3
100	88.2%	22.5
500	86.6%	25.9

Table 2: Classification Accuracy by Number of Epochs

We see a slow decline in accuracy as k increases. The best overall accuracy is with k=1 of 90.2%. The confusion matrix of this result is in Figure 3.

The change in runtime is small, it increases it by .1s (less than 1%) for even the larger k=100. It is a bit more significant at 3s for k=500, but that is only 15% for a k which considers 1/10 of all entries as neighbors. This is because k only influences the runtime when adding to and checking the k nearest neighbors. Adding doesn't happen too often once a few good values have been found. And selecting from it is just an O(k) operation. The bulk of computation time is spent comparing each test image to every single training image.



	0	1	2	3	4	5	6	7	8	9
0	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1	0.00	99.07	0.00	0.00	0.00	0.00	0.00	0.93	0.00	0.00
2	0.97	3.88	87.38	1.94	0.00	0.00	1.94	1.94	1.94	0.00
3	0.00	1.00	0.00	84.00	0.00	5.00	0.00	5.00	4.00	1.00
4	0.00	0.93	0.00	0.00	86.92	0.00	1.87	0.00	0.00	10.28
5	2.17	1.09	1.09	9.78	3.26	69.57	4.35	1.09	4.35	3.26
6	0.00	1.10	0.00	0.00	1.10	4.40	92.31	0.00	0.00	1.10
7	0.00	7.55	0.94	0.00	0.94	0.00	0.00	83.02	0.00	7.55
8	1.94	2.91	0.97	3.88	0.97	3.88	0.00	1.94	78.64	4.85
9	1.00	1.00	0.00	3.00	3.00	0.00	0.00	5.00	2.00	85.00

Figure 3: KNN Confusion Matrix

Performance was better than both the perceptron and the Naive Bayes implementations. However, we had different high confusion rates for this approach. KNN had trouble with similar pairs (5,3),(4,9), (7,9) and (3,5).

2 Pong and Q Learning

2.1 General Formulation

To formulate Pong such that it can be used in a Q Learning framework, a Markov Decision Process (MDP) needs to be defined that represents Pong. To this end, we can define $\mathcal{S} \in \mathbb{R}^n$ as the continuous state-space representing the Pong environment and then define $\mathcal{A}$ as a set of finitely many actions that can be taken within the environment.

Next, we need a transition function $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ that can give us a new state given a previous state and action pair. This function maintains the Markov property in that it only needs the last state and some corresponding action to obtain the next state. Note that $T(\cdot)$ can internally have randomness associated with it such that a given state-action pair need not produce the same output state each time they are operated on by $T(\cdot)$.

Another important thing to define, which may greatly impact the policy learned via Q Learning, is a rewards function $R : \mathcal{S} \rightarrow \mathbb{R}$ that maps a given state into a scalar that decides how good it is to be in that state. This function will have specific definitions depending on the goal for Pong, so these will be defined later.

Lastly, we need a way to discretize the continuous space into features that can be used to represent the Q values for a given state. Let us define $\mathcal{S}_D$ as the discrete state-space of our environment, then we must have some function $D : \mathcal{S} \rightarrow \mathcal{S}_D$ that takes a continuous state and spits out a discrete state. For added convenience, we can define bijective hash functions $h_s : \mathcal{S}_D \rightarrow \mathbb{N}$ and $h_a : \mathcal{A} \rightarrow \mathbb{N}$ to go from discrete states and actions to indices. From there, we define the Q values representation as a matrix $Q \in \mathbb{R}^{|\mathcal{S}_D|} \times \mathbb{R}^{|\mathcal{A}|}$ and use the hashes for the discrete state and action to retrieve a Q value for some state and action pair.

The Q Learning update algorithm itself, in the complete continuous representation, is then based on the following recursive relationship:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha\left(R(s) + \gamma \max_{\hat{a}} Q(T(s,a),\hat{a})\right) \tag{1}$$

where (s,a) are some state-action tuple, $\alpha \in (0,1]$ is the learning rate, and $\gamma \in [0,1]$ is the discount factor. Using a discrete analog to the above equation, we can over experience get a better representation for Q and in turn develop a better policy $\pi(s)$ where it is defined as:

$$\pi(s) = \arg\max_{a} Q(s,a) \tag{2}$$

Now in terms of training, there are a few things that are being done. For exploration, ϵ-greedy is being used. This means for some probability ϵ, the agent will use a random action and otherwise will use the greedy policy $\pi(s)$ given the current values for Q. Additionally, α is being computed using the following equation:

$$\alpha(s,a) = \frac{C}{C + N(s,a)} \tag{3}$$

where C is some constant to be tuned, $N(s,a)$ keeps track of how many times the state-action tuple (s,a) has been visited before. The discount factor γ and C are tuned either manually or through a Design of Experiments (DOE).

2.2 General Implementation

The software implemented for this part of the assignment was done in C++. The Q Learning encapsulation was designed as a templated class that used some Template Metaprogramming techniques to allow for defining a Markov Decision Process (MDP) class that can used as a template parameter.

Additionally, the actual learning method for this Q Learning was also templated such that a user can define a function for choosing α as a function of episode number and state-action pairs and can also pass in a callback function for reporting information about the learning process. Inside this training method, the Q Learning algorithm retrieves discrete state indices, gets actions via the ϵ-greedy exploration-exploitation strategy, updates the MDP state by feeding in the chosen action, gets a reward for the MDP state, and then updates the associated Q-values using the reward, the original state-action pair, and the new state. Note that some basic statistical information is taken as an episode is played out, such as the total reward for the current episode and then a moving average of the average reward across all the episodes. After training is complete, the Q-values learned are safed off into a binary file that can be read in later.

Software was also implemented for playing games, either with a GUI or without one, against any AI players or Human players or with wall barriers on any side of the game environment. This software was developed with the help of the C library Allegro5 which helped with the GUI and event handling when playing with the GUI. A sample screenshot of the GUI, annoted with which agent is which on the screen, is shown in Figure 4.

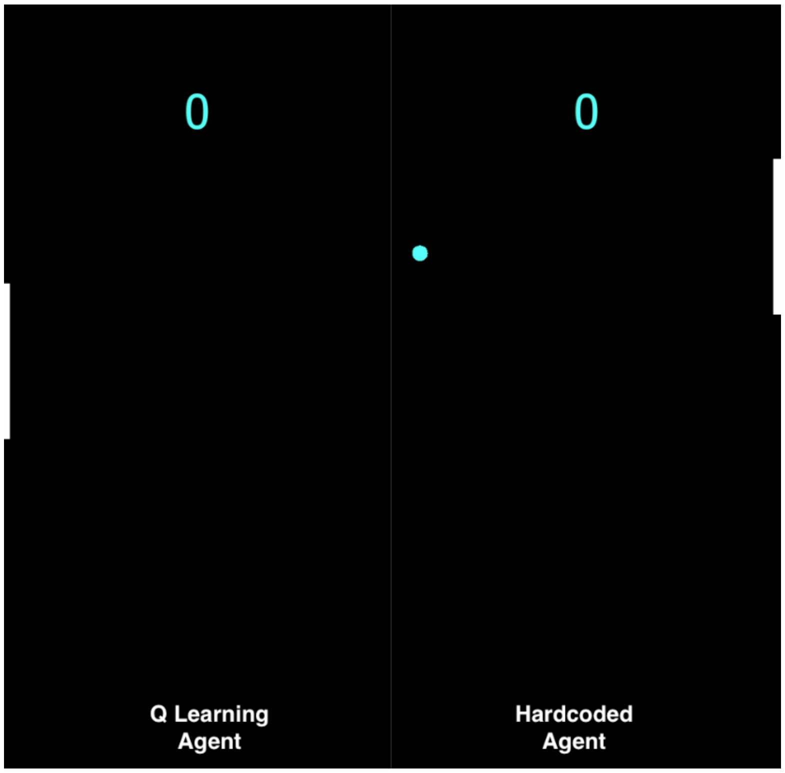


Figure 4: Snapshot of Annotated GUI in middle of Game

2.3 Volleying Pong Agent

2.3.1 Formulation

For this task, the goal is to come up with a Pong agent via Q Learning that can volley the ball against a wall as long as possible. Let us first define the continuous state as $\mathcal{S} = (s_1, s_2, s_3, s_4, s_5)$ and the actions set as $\mathcal{A} = \{a_1, a_2, a_3\}$, where the components are defined in Table 3. Note that h_p is the height of the paddle which is set to $h_p = 0.2$ for this problem. The transition function for this problem has the definition found below:

- The top and bottom of the domain are walls and the ball will bounce off of them without any noise added on top of the reflection

- The left of the domain is a wall as well but a bounce off of this wall can result in random, bounded changes to the velocity dependent on a Uniform distribution
- If the ball gets hit by the paddle on the right side of the domain, the ball will bounce and can have random changes just like those done by the wall
- Ball position vector is otherwise linearly propogated via the ball velocity
- The paddle position is updated directly by the value for an input action a

Variable	Definition
$s_1 \in [0,1]$	Ball x position
$s_2 \in [0,1]$	Ball y position
$s_3 \in [-1,1] \setminus (-0.03, 0.03)$	Ball x speed
$s_4 \in [-1,1]$	Ball y speed
$s_5 \in [0, 1-h_p]$	Paddle top y coordinate
$a_1 = 0$	Do not change paddle position
$a_2 = 0.04$	Increase paddle position
$a_3 = -0.04$	Decrease paddle position

Table 3: State and Action variable definitions for Wall Bouncing Pong Agent

The discrete states $\mathcal{S}_D$ are defined in the following manner, taken directly from the problem statement:

- Treat the entire board as a 12×12 grid, and let two states be considered the same if the ball lies within the same cell in this table. Therefore there are 144 possible ball locations.
- Discretize the x speed of the ball to have only two possible values, 1 or -1, which represent the sign of the x speed
- Discretize the y speed of the ball to have only three possible values: 1, 0, or -1. It should map to 0 if $|s_4| < 0.015$
- To convert your paddle's location into a discrete value, use the following equation:

$$\text{discrete\_paddle} = \left\lfloor \frac{12 s_5}{1-h_p} \right\rfloor$$

In cases where $s_5 = 1-h_p$, set discrete_paddle $= 11$. As can be seen, this discrete paddle location can take on 12 possible values

- Add a special state whenever the ball makes it past the paddle and leaves the game domain, call it s_e

The last thing to do is decide how the rewards are going to be generated. How the rewards function $R(\cdot)$ works is defined below:

- If the discrete state is special state s_e, the reward is -1
- If the discrete state represents a bounce off of the paddle, the reward is 1
- Otherwise, return reward of 0

With these definitions, it is pretty trivial to put together the necessary MDP and state-space information needed to solve Pong using Q Learning.

2.3.2 Nominal Results

To create a good agent capable of volleying the ball against a wall in Pong using Q Learning, it was important to come up with choices for γ, ϵ, and C. Choices for γ, and C were found by doing a small manual Design of Experiments (DOE), specifically a Factorial Experiment. The experiment was based on $\log_{10}(C) \in [0, 4]$ and $\gamma \in [0.1, 0.99]$. For ϵ, a function was chosen as the following:

$$\epsilon(n) = \frac{k\epsilon_0}{k + n}$$

where n represents the episode number in the training, k is some constant that was chosen as $k = 10^3$, and ϵ_0 is a new tuning parameter. This parameter was experimented with values $\log_{10}(\epsilon_0) \in [-2, 0]$. Through experiments with the DOE, it was found that choices for γ and ϵ_0 certainly affected convergence to a good policy but that the choice for C was surprisingly vital in getting a good policy quickly. Through experimenting, it was found $\gamma \geq 0.7$, $\log_{10}(C) \geq 3$, and $\epsilon_0 \geq 0.5$ tended to generate quicker convergence to strong policies.

To obtaining a policy that performed sufficiently well for this volleying Pong agent, the tuning parameters were chosen as $\gamma = 0.75$, $C = 10^4$, and $\epsilon_0 = 0.99$ and the training was over 10^6 volleying games. After running this volleying Pong agent across 10^5 games to test its performance, the average number of times per game that the ball bounced off the agent's paddle before the ball escaped past the paddle was 14.43 bounces per game.

2.3.3 Results after MDP Modification

To see if improved performance could be obtained, the MDP and discrete state representation were slightly modified. First, the rewards were modified such that the agent received a reward of 1 for bouncing the ball and received a reward of -10, instead of the original value of -1, if the ball made it past them. The idea to do this is that the agent may discover a bad action quicker with such a heavy penalty.

Additionally, the y ball position was discretized to have 24 discrete values instead of 12. The reason only the y ball position was refined, instead of both the x and y ball position values, was because the paddle's decision of where to be is more dependent on the ball's y position than the x position. This theory stems from the fact that given the paddle is lined up vertically with the ball, the paddle really only needs to move of the ball has a nonzero y velocity, so it could be useful to keep track of the ball's y position more accurately than the x position.

One should note the training time was really no different since the number of training episodes were set, but the Q-values table was twice as large after the above change. After doing the training using the same tuning parameter values and number of episodes for the optimal case described in Section 2.3.2, the average number of bounces per game for this new Q Learning agent was found to be 15.07 bounces per game. While not some enormous improvement on the baseline performance of 14.43 bounces per game, it does show some improvement.

2.4 Dualing Pong Agents

2.4.1 Formulation

Within this task, the goal was to come up with a Pong agent that could beat another Pong agent hardcoded to essentially track the vertical position of the ball at all times but move with a velocity half that of the Pong agent being trained. To do this, the new Pong agent had a slightly modified state and set of rewards.

For the state, an added state component s_6 was thrown in such that s_6 represents the opponent paddle's top y coordinate. This continuous state was then discretized in the same manner s_5 was. Additionally, a second special discrete state was added for when the ball moves past the opponent's paddle. It should be noted that the added variable s_6 multiplies the original state space by the number of discrete values this variable takes, which is a negative since the space requirements are larger, the training time is longer, and chance of touching some discrete state is much less. However, the benefit is that hopefully knowledge of the opponent paddle's position will help the Q Learning agent's paddle perform more strategically.

As for rewards, now a reward of 1 is given when the ball moves past the opponent's paddle, a reward of -1 is given when the ball moves past the Q Learning agent's paddle, and a reward of 0 is given otherwise. The logic behind this choice is that the agent should only view getting a ball past their opponent as a job well done while having a ball get past them is obviously from doing a poor job. It is expected that this reward system will also require γ to be on the larger side so the agent can maintain the value of some move that might take a little bit to see how useful it was on scoring on the opponent.

2.4.2 Implementation Modifications

Relative to the MDP defined for the Volleying Pong agent, the rewards, state representation, and transition function were slightly modified depending on if the MDP should represent the Volleying Pong agent or the Dualing Pong agent situation. These changes were quite trivial, though, so no major code changes were made here.

2.4.3 Nominal Results

To produce the most optimized Q Learning agent seen in our experiments for the Dualing Pong task, the training was done with $\gamma = 0.8$, $\epsilon_0 = 0.99$, $C = 10^4$, and the number of training episodes were 10^7. With these training parameters in mind, the win rate of the Q Learning agent against the baseline Pong agent can be found to be 97%, as noted in Table 4.

Number of Games Played	Win Rate by Q Learning Agent (%)
10,000	97.04

Table 4: Win Rate of Q Learning Agent for Dualing Pong Task

A video for this Q Learning agent going up against the hardcoded Pong agent can be found here. Note that this video just shows the two AI playing to 50 points.

2.4.4 Results after Human Playing Agent

So to really test the quality of the agent, a human player played against this agent. This test was actually quite interesting because some flaws in the agent's play style became quite apparent. First, it seemed this agent's strategy banked too much on the fact its opponent was going to actively track the vertical position of the ball. Since the human player generally didn't perfectly track the vertical position of the ball as a strategy, sometimes the Q Learning agent would get stumped about what action to take and even stop moving much!

This flaw seemed to be related to the Q Learning agent ending up in a state it had not either seen before or had not seen often enough to get a good idea what action to take while in it. So while the agent performed fairly well, it was not generally good enough to win against the human player when playing the game to a higher number of points.

3 Contributions

- Christian Howard
  - Did Q Learning part of assignment
- Luke Pitstick
  - Did Digit Classification part of assignment

4 List of Potential Extra Credit

- Made a GUI for Pong
- Made the GUI capable of allowing two players to play Pong. The players can be only AI, only Human, or a mix.
- Video of Q Learning-based Agent vs. Baseline Agent found here.

Appendices

A Open Source Software Used

- Standard C++14 Libraries
- Allegro5 C library for GUI and event handling