CS 440/ECE 448: Artificial Intelligence
Assignment 4: Perceptrons and Reinforcement Learning
Three Credit Hours

Rohan Kasiviswanathan (kasivis2)
Nicholas Wheeler (nrwheel2)
Joseph Vande Vusse (vandevu2)

Part 1: Digit Classification using Discriminative Machine Learning Methods

1.1 Introduction and Observations:

For implementing the perceptron algorithm, we initialize the weights to random values (between 0 and 1). While initializing the weights to zero was more consistent and sometimes better than random weights, the best performance of random weights did better than not choosing random weights. One point of interest is that choosing fixed initial weights from 0 - 10 resulted in the same overall accuracy. The overall accuracy of random weights ranged from 84 - 86%. After initializing the weights, we made a run through the training samples using a fixed ordering, as that generally performed better than a random ordering. After doing that, we go through the training set a specific number of times (epochs) up until we converge on a specific accuracy for the training data. For each epoch, we make predictions on the training data and then accordingly update the weight vectors using a specific learning rate decay function. The overall number of epochs required to converge was 10. The average number ranged from 5 - 7, and 10 is a maximum as it never required more than 10 epochs to converge. The learning rate decay function that worked best for us was: 1 / (1 + exp(currentEpoch * x)), where currentEpoch is the current run (epoch). The best accuracy and corresponding convergence rate was gotten when x = 2. When viewed on a graph which goes from 0 to infinity, 2 would be a maximum with performance results not as great for values of x both smaller and larger than 2. Lastly, we found that having no bias when updating the weights generally led to better performance. Once finishing going through the epochs, we would then use our current updated weight vector on the testing data and then figure out the overall accuracy, which we then displayed. We also had code which displayed the confusion matrix, and the other needed output information. Now, we will move on to displaying the results.

1.1 Results:

The **training curve** is as follows. Here, we display the training curve as a list of statements with each line indicating the accuracy for that specific epoch (goes from 1 - 10):

```
Training accuracy after 1 epochs is 0.78
Training accuracy after 2 epochs is 0.9066
Training accuracy after 3 epochs is 0.925
```

Training accuracy after 4 epochs is 0.9294
Training accuracy after 5 epochs is 0.9294
Training accuracy after 6 epochs is 0.9298
Training accuracy after 7 epochs is 0.93
Training accuracy after 8 epochs is 0.93
Training accuracy after 9 epochs is 0.93
Training accuracy after 10 epochs is 0.93
Epochs greater than 10 have an accuracy of 0.93

Here is the overall **accuracy** for both the training and test data:
Training set accuracy was: **0.93**.
Test set accuracy was: **0.847**.
Here are the accuracies for the individual digits within the test data:
Correct rate of classification for digit 0 is 0.956
Correct rate of classification for digit 1 is 0.972
Correct rate of classification for digit 2 is 0.874
Correct rate of classification for digit 3 is 0.840
Correct rate of classification for digit 4 is 0.879
Correct rate of classification for digit 5 is 0.772
Correct rate of classification for digit 6 is 0.857
Correct rate of classification for digit 7 is 0.755
Correct rate of classification for digit 8 is 0.738
Correct rate of classification for digit 9 is 0.830

Displayed below is the overall **confusion matrix** for the output.
The confusion matrix is as follows.
Rows correspond to the label value.
Columns are the predicted values.
The diagonal represent correct classifications.

0.956	0.000	0.000	0.000	0.000	0.000	0.011	0.000	0.022	0.011
0.000	0.972	0.009	0.000	0.009	0.000	0.009	0.000	0.000	0.000
0.000	0.010	0.874	0.019	0.010	0.000	0.049	0.010	0.010	0.019
0.000	0.000	0.020	0.840	0.000	0.050	0.030	0.040	0.010	0.010
0.000	0.000	0.009	0.000	0.879	0.000	0.037	0.009	0.009	0.056
0.033	0.000	0.011	0.065	0.011	0.772	0.011	0.011	0.087	0.000
0.011	0.011	0.033	0.000	0.044	0.022	0.857	0.011	0.011	0.000
0.000	0.028	0.057	0.028	0.000	0.000	0.000	0.755	0.009	0.123
0.010	0.019	0.019	0.097	0.010	0.058	0.019	0.010	0.738	0.019

0.000 0.000 0.000 0.040 0.070 0.020 0.000 0.040 0.000 0.830

1.1 Comparison to Naive Bayes Classifier and Conclusion:

Using perceptron with a specific number of epochs performs considerably better than the naive bayes classifier used for assignment 3, part 1. The naive bayes classifier had an accuracy of around 77.5%, whereas the perceptron based classifier has an average accuracy of 84.5% (considering the accuracy varies slightly when using random initial weights). Here are some interesting points. The naive bayes classifier only makes one run through the training data due to the nature of the algorithm, compared to the perceptron which makes multiple runs. A few other differences are that the naive bayes is a probabilistic classifier, while Perceptron is a supervised learning algorithm. Lastly, for perceptron, the training data is also used for testing, whereas in naive bayes it is not. Overall, neural network models of learning generally perform better than probabilistic models.

Part 1 Extra Credit (Part 1.2)

1.2 Introduction:

We implemented a K-Nearest Neighbor algorithm to classify the Digits that were provided. Multiple different functions were considered for the similarity function of the classifier. Ranges of values for k were tried for each different similarity function. We tried using the Manhattan distance between test and training image pixels, the Euclidean distance, and a modified Euclidean distance that only included pixels in the center of the image.

Our K-nearest neighbors algorithm scales linearly with the choice of k. We simply have to grab the k-nearest neighbors and increment an array that contains the 10 possible labels based on what the neighbor's label value is in the training set. The running time for one K-NN classifier the way we implemented it is around 10 seconds. Increasing k on this dataset does not significantly increase the running time, but on a larger dataset with much larger k values, the difference would be noticeable.

1.2 Results:

Results for our best similarity function are shown below. The Euclidean distance proved to be the best similarity function. For that, we would take the difference between every pixel value for a given test image and a training image, square the values, add them up, and ultimately take the square root of the acquired similarity. As is shown below, the accuracy was 89% at its best, decreasing slightly as the value of k continued to get larger. This is a better performance than both our Bayes and Perceptron classifiers.

While the Euclidean distance provided a better similarity than Bayes and Perceptron, Manhattan distance and a Euclidean distance over a smaller portion of the features in the center both underperformed these other classifiers. Euclidean distance might not be the optimal similarity function, but it certainly does a good job on this dataset.

```
Accuracy for k= 1 is 0.896
Accuracy for k= 4 is 0.895
Accuracy for k= 7 is 0.884
Accuracy for k= 10 is 0.89
Accuracy for k= 13 is 0.879
Accuracy for k= 16 is 0.881
Accuracy for k= 19 is 0.867
Accuracy for k= 22 is 0.859
Accuracy for k= 25 is 0.86
Accuracy for k= 28 is 0.858
Accuracy for k= 31 is 0.852
Accuracy for k= 34 is 0.845
Accuracy for k= 37 is 0.839
Accuracy for k= 40 is 0.833
Accuracy for k= 43 is 0.828
Accuracy for k= 46 is 0.817

The confusion matrix is as follows.
Rows correspond to the label value.
Columns are the predicted values
The diagonal represent correct classifications


1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000

0.000   1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000

0.009   0.058   0.903   0.000   0.000   0.000   0.000   0.019   0.010   0.000

0.000   0.010   0.000   0.870   0.000   0.050   0.010   0.050   0.010   0.000

0.000   0.009   0.000   0.000   0.869   0.000   0.019   0.000   0.000   0.103
```

```
0.033   0.000   0.011   0.065   0.000   0.815   0.033   0.000   0.011   0.033

0.022   0.011   0.000   0.000   0.000   0.022   0.945   0.000   0.000   0.000

0.000   0.094   0.000   0.000   0.009   0.000   0.000   0.821   0.000   0.076

0.010   0.039   0.010   0.049   0.049   0.019   0.010   0.029   0.777   0.010

0.010   0.000   0.010   0.030   0.020   0.010   0.000   0.010   0.000   0.910

Correct rate of classification for digit 0 is 1.00
Correct rate of classification for digit 1 is 1.00
Correct rate of classification for digit 2 is 0.903
Correct rate of classification for digit 3 is 0.870
Correct rate of classification for digit 4 is 0.869
Correct rate of classification for digit 5 is 0.815
Correct rate of classification for digit 6 is 0.945
Correct rate of classification for digit 7 is 0.821
Correct rate of classification for digit 8 is 0.777
Correct rate of classification for digit 9 is 0.910
```

Part 2: Q-Learning (Pong)

The general reinforcement learning in Pong revolves around developing a set of states based on the state of the paddle, the state of the ball, and the x and y velocity of the ball. During the simulation, we run our ball and paddle values through function to get discrete value and see what state we are in. Then, we make a random move until we have made 100 of each move in each state. Eventually, we make moves according to the best Q value (once we have a lot of data). We then apply the velocities to the ball and move it. Its movement dictates our next state. The current state, next state, and Q values associated with both are used in conjunction with constants to add or subtract from the Q value of the current move for the current state. The game ends when the ball gets past the paddle. We perform 100,000 training games. After this, we perform 1000 testing games, during which we record the paddle rebounds, and divide them all by 1000 to determine the average rebound rate for the paddle (our mark of success for reinforcement learning).

The choice of reinforcement learning constants was essentially arbitrary. The ones up to our discretion were gamma (multiplier on optimal next-state move) and C (the decay constant in the alpha decay equation). Based on some research and inference from the types of equations these constants were affecting, we concluded that

C should range somewhere from 25-100 and gamma should be somewhere from 0-1, more specifically from 0.5-1. After this, it was purely up to training and testing with different parameters to choose the ones that achieved the highest average rebound rate. The process was somewhat scientific in that we would vary one constant while keeping the other stable to detect changes, but there existed some randomness from the paddle movement and ball velocity.

Ultimately, we achieved an average rebound rate of 12.869 with C = 50 and gamma = 0.9, so we adopted these values as our best. This rate was based on 1000 testing games after 100,000 training games. The problem statement suggests that we should rebound the ball roughly 9 times per trial for our implementation to be considered "good". After 50,000 training games, we only achieved an average bounce rate of 4.141, but after 60,000 games, it jumped all the way up to 9.222. Therefore, 60,000 games appears to be the necessary amount of training for a "good" result. The quick jump is likely due to the number of moves made from each state. We require that at least 100 of each move must be made from each state before we begin making moves based on the highest Q value. Each state likely has over 100 of each move attempted for each state at around 60,000 games. After 75,000 games, the average rebound rate is up to 11.925, but after 250,000 games, it only rises to 12.914. Therefore, we would argue that the saturation or training convergence occurs around 80,000 training games. It may still increase slightly forever, but with diminishing marginal returns.

In changing the MDP/constants to achieve a more desirable result, we first tried to change the number of states for the ball rows, columns, and the discrete paddle (currently 12x12 and 12, respectively). My naïve assumption was that more states would result in more precision and, therefore, a higher average rebound rate. It turned out that 12 appears to be a peak because no other numbers (higher or lower) had a better average rebound rate. For fewer states, the 6 and 10 state values yielded rebound rates of 5.473 and 10.741, respectively. For more states, the 15 and 24 state values yielded rebound rates of 2.390 and 1.311, respectively. I would argue that if we could train these higher values with many more games (proportional to the size of their state spaces) they may achieve much higher average rebound rates than even the initial 12 state model. We would have tested this, but doing enough extra games to make a difference was too time-intensive. I experimented with changing several other constants as well, such as the maximum velocity, change in velocity, and paddle height. Whether or not this was the spirit of the question, each had the expected outcome with lower velocity, less change in velocity, and a larger paddle achieving increasingly better average rebound rates with how much they were changed.

Results:

```
Rebound Test Rate = 12.869 per trial
```

Part 2 Extra Credit (Part 2.2)

In developing the two-player Pong version, we decided to not make any changes to the MDP or the state space. We simply wanted the AI agent to be the same agent as in Part 2.1. In this way, it was easier to compare our extra credit results with other groups based on the similarity (or dissimilarity) of our Part 2.1 results. All of the changes occurred in developing similar functions for the hard-coded player “Player 2”. Player 2 had its own decision function on how to move the paddle, its own actual movement function, and a move in each trial of the simulation.

For 100,000 trial games (like the published results of Part 2.1), the AI beat the slow, hard-coded Player 2 97.4% of the time. This seems quite high when first looking at it; however, the AI returns the ball almost 13 times each trial on average. Because Player 2 is so much slower, it seems logical that it would almost always be too slow to also return the ball 13 times. It was stated in the 2.1 problem statement that returning the ball on average 9 times per trial would be a “good” implementation, so we ran the two-player game again with only 60,000 training games for Player 1 (the stated threshold for a “good” agent from Part 2.1). In this case, the win rate for Player 1 was only 81.5%. This seems logical because the problem statement suggests the AI should be able to win about 75% of the time (close to 81.5%), and this is the minimum threshold on a “good” agent.

Results:

```
AI Win Rate = 0.974
AI Win Rate = 0.815
```

Statement of Individual Contributions:

Rohan - Part 1 and report Part 1
Nick - Part 1 and report Part 1
Joe - Part 2 and report Part 2