ECE 448: Fall 2017
Intro to Artificial Intelligence

3 Credit Hour MP4 Report

Perceptrons and Reinforcement Learning

Ben Li, Harsh Modhera, Mihir Sherlekar
cli91, hmodhe2, sherlkr2
December 10, 2017

Part 1: Digit Classification

Part 1.1: Digital Classification with Perceptrons

Introduction

The first part of the MP tasked us with implementing a Perceptron classifier in order to distinguish between the visual patterns of numerical digits 0 through 9. In order to create the classifier, we were given a set of 5000 training exemplars and 1000 testing exemplars along with their respective ground truth tables. The exemplars were roughly evenly distributed among the 10 target classes.

Code Breakdown — File Structure and Program Flow

For this entire MP, please see attached zip file for our code and file structure.

All of our functionality code for this part is placed in `4.1_digit_classification_perceptrons.py`. To run the program, a user would type in the command line the following:

```
python 4.1_digit_classification_perceptrons.py
```

Once the user hits enter, the program initiates.

In order to implement bias, we created a separate file for ease of testing. That file is `4.1_with_bias.py` and it can be run in the same manner as the original implementation above.

Implementation

In order to generate the most accurate classifier using the multi-class (non-differentiable) perceptron learning rule, we had to experiment with various parameters, all of which are listed in the sections below. We decided to test as thoroughly as possible, running our script with different values and different combinations of these parameters. The confusion matrix and classification rate matrix we generated was our primary method of evaluating how accurate our perceptron classifier model. was.

Inputs

The inputs for this part of the MP are the 4 files in the `digitdata` folder. The `trainingimages.txt` file contains 5000 training samples while the `testimages.txt` file contains 1000 testing samples. The training and testing samples were roughly evenly distributed among each of the 10 target classes. Two additional files, `traininglabels.txt` and `testlabels.txt`, were used for training and evaluation of the given inputs.

Each digit sample was a 28x28 image represented as an ASCII text string. All the samples were concatenated together vertically. Three distinct characters were used to determine whether a particular pixel was part of the background (white), a gray pixel, or part of the foreground (black). These pixels were represented by the following characters, respectively — ‘ ‘, ‘+’, and ‘#’.

Results

Before we move to presenting the results for the accuracies of the test images, we want to present the training curve that we obtained. This training curve expresses the overall accuracy on the training set as a function of the epoch. The learning rate decay function used to generate this training curve is 1/(Epoch+1).

Epoch	Average Classification Rate for the 10 Digit Classes (Rounded to 5 decimal points)
0	0.77764
1	0.88933
2	0.91340
3	0.92818
4	0.94018
5	0.94833
6	0.95224
7	0.95807
8	0.96399
9	0.96297
10	0.96384
11	0.96901

12	0.96925
13	0.97213
14	0.97216
15	0.97288
16	0.97213
17	0.97509
18	0.97922
19	0.97820
20	0.97902
21	0.97940
22	0.97890
23	0.98038
24	0.98202

As expected, the classifier approaches 100% accuracy on the training set. However, as we have learned in lecture, over-training your classifier might not improve your overall classification accuracy when testing out of sample.

The following is a list of parameters that we were allowed to play around with in order to obtain a higher classification accuracy through this perceptron learning model.

- Learning rate decay function
- Bias vs. no bias
- Initialization of weights (zeros vs. random)
- Ordering of training examples (fixed vs. random)
- Number of epochs

The following table summarizes some of our results when trying to experiment with these parameters. Each row reports the average classification rate for the 10 digit classes given the combination of parameters for a specific Epoch value.

		Learning Rate Decay Function	**Bias**	**Weight Initialization**	**Ordering of Training Samples**	**Average Classification Rate**
Epochs	5	1/(t+1)	W/O	0.0	Fixed	0.79176
	5	1000/(t+1000)	W/O	0.0	Fixed	0.77392
	5	1/(t+1)	1	0.0	Fixed	0.80439
	5	1/(t+1)	0.5	0.0	Fixed	0.80737
	5	1/(t+1)	0.1	0.0	Fixed	0.8101
	5	1/(t+1)	2	0.0	Fixed	0.76112
	5	1/(t+1)	5	0.0	Fixed	0.63257
	5	1/(t+1)	0	0.0	Fixed	0.8078
	5	1/(t+1)	0.1	Random	Fixed	0.81171

	5	1/(t+1)	0.1	Random	Random	0.77457

After playing around with several combinations of the parameters, we found the following settings to be optimal.

Learning Rate Decay Function	Bias v/s No Bias	Initialization of Weights	Ordering of Training Examples	Number of Epochs
1/(t+1)	0.1	Random	Fixed	5

These settings provided the following results. Below are the classification rates for each digit. The only class that were correctly classified for less than or equal to 75% of the exemplars were 3 and 8 . The best classification rate was 83.081%. The classification rate varied as a result of the randomly initialized weights. After running our best configuration several times, we achieved an average of 81.171%.

Classification Rates for each Digit									
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0.978	0.981	0.874	0.75	0.822	0.783	0.78	0.821	0.699	0.82

The following table represents the confusion matrix for the digit classification problem in MP 4.1. Note that for both confusion matrices, R = real is the column of bolded digits & G = guess is the row of bolded digits.

Confusion Matrix										
R \ G	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0.978	0.0	0.022	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.981	0.009	0.0	0.0	0.0	0.009	0.0	0.0	0.0
2	0.0	0.019	0.874	0.0	0.019	0.0	0.029	0.039	0.019	0.0
3	0.0	0.02	0.07	0.75	0.0	0.03	0.0	0.11	0.01	0.01
4	0.0	0.019	0.0	0.0	0.822	0.0	0.037	0.0	0.009	0.112
5	0.011	0.011	0.022	0.065	0.022	0.783	0.0	0.033	0.054	0.0
6	0.011	0.044	0.066	0.0	0.044	0.044	0.78	0.011	0.0	0.0

7	0.0	0.057	0.066	0.0	0.009	0.0	0.0	0.821	0.009	0.038
8	0.019	0.049	0.039	0.049	0.029	0.078	0.01	0.029	0.699	0.0
9	0.0	0.0	0.01	0.03	0.04	0.02	0.0	0.08	0.0	0.82

The following are the best results we were able to obtain in MP 3.1 with a smoothing constant of K = 0.1. The only classes that were correctly classified for less than 70% of the exemplars were 5 and 8. The average for the classification rate was 77.01%.

Classification Rates for each Digit									
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0.844	0.963	0.786	0.8	0.748	0.685	0.769	0.726	0.602	0.8

The following table represents the confusion matrix for the digit classification problem in MP 3.1.

Confusion Matrix										
R \ G	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0.844	0.0	0.011	0.0	0.011	0.056	0.033	0.0	0.044	0.0
1	0.0	0.963	0.009	0.0	0.0	0.019	0.009	0.0	0.0	0.0
2	0.01	0.029	0.786	0.039	0.019	0.0	0.058	0.01	0.049	0.0
3	0.0	0.01	0.0	0.8	0.0	0.03	0.02	0.07	0.01	0.06
4	0.0	0.0	0.009	0.0	0.748	0.009	0.037	0.009	0.019	0.168
5	0.022	0.011	0.011	0.13	0.033	0.685	0.011	0.011	0.022	0.065
6	0.011	0.044	0.044	0.0	0.044	0.066	0.769	0.0	0.022	0.0
7	0.0	0.047	0.038	0.0	0.028	0.0	0.0	0.726	0.028	0.132
8	0.01	0.01	0.029	0.136	0.029	0.078	0.0	0.01	0.602	0.097
9	0.01	0.01	0.0	0.03	0.1	0.02	0.0	0.02	0.01	0.8

The results clearly indicate that the perceptron approach achieved higher classification accuracies than the naives bayes model.

Part 2: Q-Learning

Part 2.1: Single-Player Pong

Introduction

The second part of the MP tasked us with implementing a single-player pong game, where the agent had to learn how to bounce the ball off the paddle as many times as possible. The Q-learning algorithm was used to train the Markov Decision Process that was highlighted in MP4 documentation.

Code Breakdown — File Structure and Program Flow

For this entire MP, please see attached zip file for our code and file structure.

All of our functionality code for this part is placed in `4.2_q-learning_pong.py`. To run the program, a user would type in the command line the following:

```
python 4.2_q-learning_pong.py
```

Once the user hits enter, the program initiates.

Implementation

The Q-learning agent we implemented follows the Markov Decision Process described in the MP doc. It operates in a simulated, discretized environment to simplify the game which would normally be played in a continuous environment otherwise. We trained our agent over the course of 1000 game simulations and determined the average number of times the ball rebounded.

In order to generate the most accurate agent using the Q-learning, we had to experiment with various parameters, all of which are listed in the sections below. We used the average number of times the ball was rebounded by our agent as the primary method of evaluating how well it performed.

<u>Results</u>

We used α = 0.3 after using the suggested changing α didn't work well for us. γ = 0.9. The learning rate controls the Q-value learning or update: alpha closer to 0 means learning is slow while value closer to 1 means very fastly changing rate. We chose a slow learning rate over a high number of testing games to stabilize the model. The gamma models the fact that future rewards are worth less than immediate rewards. So we chose a value for that closer to 1 to give more weight to current state-action pair value. For the exploration function, we used ε - greedy and set it equal to 0.05, which is a good arbitrary value we picked so that the model can sometimes explore other states. We run approximately 75-100 thousand games to achieve an acceptable policy.

After determining the optimal values for α and γ, we ran our algorithm for 50,000 iterations and determined the average number of times per game the ball bounced back to be 10.

For the ball_x and ball_y states, we simply multiplied the float value by the grid dimension (minus 1) to map to the [0,11] range. For the velocity_x and velocity_y, we used a sign function that returns -1, 0 or 1 to map them to {-1, 1} and {-1, 0, 1}, respectively. Lastly, the paddle_y we used the suggested discrete formula given in the MP doc. So the time impact of discretization was mostly just several floating point calculations and a few function calls to numpy's sign function. The overhead should not be very significant, since we also only discretize the states minimally. One interesting observation is that it reduces the actual space of our state from continuous space to discrete space, the former of which is much greater than the latter. This makes our hashing dictionary for discrete state to Q-value pairs work better than if we hashed continuous states. We found that letting the training model run about 140-150 thousand games let the Q values converge relatively to the point that there are minimal changes to it. We use this model to run a set of 1000 games, and we got a new rebound of 12.

<u>Team Contributions</u>

Ben: 2.1, report

Harsh: 1.1, report

Mihir: 1.1, report