CS 440 ONL: Assignment 4

Team: Asish Balu(asishtb2), Ramya Narayanaswamy(rpn2)
3 Credit Students

Part 1: Digit Classification with Perceptrons

A multi-class classifier was built for digit classification. The fundamental classifier is based upon the principle of differentiable perceptron. The goal of the classifier is to beat the results of accuracy of Naive Bayes classifier from Assignment 3.

There are 10 perceptron-based classifiers in our framework, following the principle of one-vs-others multi-class classification. The classifier 'i' outputs a value of 1 for digit 'i', and 0 for all other digits. The activation function is sgn(w.x + b), where 'w' is the weight vector to be learned and 'x' is the input feature set for each digit. '

Implementation details:

Each digit is represented by a 28x28 pixel image. A full-connected network for perceptron at the input layer gives a structure that needs 28*28 = 784 weights (+1 if there is a bias factor) to be learned for each class (i.e each perceptron). A single perceptron class was implemented in Python and 10 objects or instances of the class were created. Each perceptron object has initializer to set initial weights to 0 or random values and option to incorporate bias. There are three helper functions: one each for incrementing weight, decrementing weight and for calculating weighted sum of w.x+b

Training Phase: At the training phase, In each trial : every training digit is fed into all 10 perceptrons and the classifier output for each digit is evaluated as argmax_c (w_c x + b). If a digit c is misclassified d', the weights of classifier c are increased and weights of classifier d are reduced based on the formulae $w_c = w_c + \alpha x$ and $w_d = w_d - \alpha x$. The above process is repeated N number of times, where each N is called number of epochs. Alpha is the learning rate, which decays with epoch.The learned weights from training are fixed for testing phase.

Testing phase: At testing phase, classifier output for each digit is evaluated as argmax_c (w_c x + b).

Tuning of parameters :
Note: Learning Rate and Number of epochs were tuned in tandem

1. Learning Rate : Learning rate was tuned to get results higher than Naives Bayes classifier. Among the various values of learning rate tried, we achieved better accuracy with Learning Rate as = 10/(10+Epoch Num). Learning rate starts close to 1 and decays with every epoch.

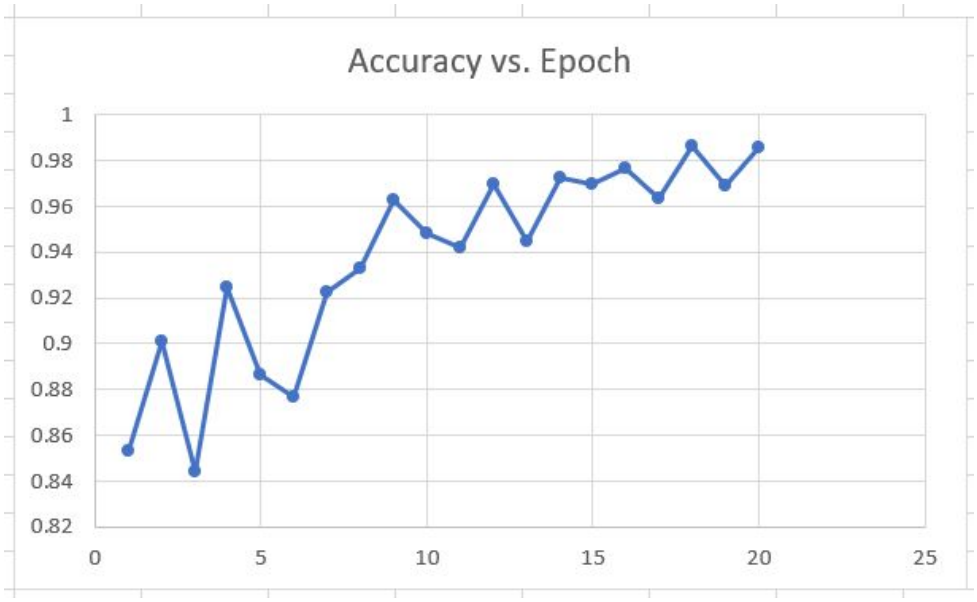
2. Number of epochs: The number of trials need to be at least 10 to get overall accuracy greater than 80%. We ran a few experiments and found there is less run-run variation of accuracy if number of trials were at least 20.
3. Bias vs No Bias: There was no significant influence on bias or no bias once learning rate and number of epochs were fixed
4. Initialization of weights: To all 0's or random value. Once learning rate and epochs were fixed, there was no significant variation of results with either choice.
5. Ordering of training samples : Random ordering in every epoch helped in improving the accuracy by at most 1%. Hence, we chose to use random ordering.

Codes : All the part1 codes are in folder Part1. Perceptron.py has code related to weight modification, and weighted sum calculation. Classifier.py has code for training and testing phase, ReadData.py has code for parsing training and test data. Finally, the top-level is in TestClassifier.py. How to run : python TestClassifier.py

Results:
Training curve. The first number is epoch and second number is accuracy of training set at the end of each epoch

epoch, accuracy 1 0.8532
epoch, accuracy 2 0.9012
epoch, accuracy 3 0.8442
epoch, accuracy 4 0.9248
epoch, accuracy 5 0.8866
epoch, accuracy 6 0.877
epoch, accuracy 7 0.9222
epoch, accuracy 8 0.9332
epoch, accuracy 9 0.963
epoch, accuracy 10 0.9484
epoch, accuracy 11 0.9418
epoch, accuracy 12 0.9696
epoch, accuracy 13 0.9444
epoch, accuracy 14 0.9726
epoch, accuracy 15 0.9696
epoch, accuracy 16 0.9764
epoch, accuracy 17 0.9634
epoch, accuracy 18 0.9866
epoch, accuracy 19 0.9688
epoch, accuracy 20 0.9856

Training Curve (graph):



Overall accuracy for a run is 82.2%. However, there were run-run variations and we were able to get as high as **82.8%** This is higher than Naive Bayes classifier accuracy that we achieved in Assignment 3 (which was 76%). We believe that the limitation of Naive Bayes is the assumption that the individual pixel features are independent for likelihood calculation. We also compared the diagonal entries of confusion matrix. The accuracy rate for digit 8 is pretty much same as Naive Bayes. '8' was the most difficult digit to classify in both the techniques as per our observation. We also observed that accuracy of digit 9 reduced by 5% in perceptron approach. However all other accuracies improved, contributing to an increase in overall accuracy.

The confusion matrix (for 82.2% overall accuracy) is given below. Top row corresponds to digit 0 and last row corresponds to digit 9.

0.900	0.000	0.033	0.011	0.000	0.011	0.011	0.000	0.022	0.011
0.000	0.981	0.000	0.000	0.009	0.000	0.009	0.000	0.000	0.000
0.000	0.019	0.816	0.058	0.000	0.010	0.029	0.029	0.039	0.000
0.000	0.000	0.030	0.860	0.000	0.060	0.000	0.040	0.010	0.000
0.000	0.000	0.028	0.009	0.850	0.000	0.028	0.019	0.009	0.056
0.011	0.000	0.022	0.054	0.011	0.772	0.011	0.033	0.065	0.022
0.011	0.011	0.033	0.000	0.022	0.055	0.835	0.022	0.011	0.000
0.009	0.028	0.038	0.009	0.028	0.000	0.000	0.821	0.000	0.066
0.000	0.029	0.058	0.078	0.039	0.097	0.029	0.019	0.631	0.019
0.000	0.000	0.010	0.040	0.120	0.010	0.000	0.060	0.010	0.750

Part 2: Q-Learning (Pong)

In this assignment, we are tasked with using Q-Learning to learn the optimal policy for an agent (paddle) playing pong. The goal of the agent was to bounce the ball as many times as possible before it missed.

In Q-Learning, the agent keeps track of the Q values for each state/action pair. The Q value for a particular state/action pair gives a rough estimate of the utility of choosing that action in that state. This means that the policy of the agent should be to choose the action associated with the maximum Q value.

Q Values are learned using the following update equation, which is calculated iteratively as the agent chooses actions:

$$Q(s,a) := Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)) \right]$$

Choosing the Parameters:

There are three key parameters that alter the accuracy and speed of the Q learning algorithm: alpha, gamma, and epsilon.

Alpha represents the learning rate. The higher the learning rate, the faster the agent will learn the optimal Q value for a state/action pair. However, if the learning rate is too high, the agent may overshoot the optimal Q value. The learning rate must decay over time so that the solution will eventually converge. The “Learning Rate Constant” we chose for this problem is inversely proportional to alpha.

Gamma represents the discount factor. A higher discount encourages the agent to choose more rewards now rather than think about future rewards. Essentially, a low gamma value (close to 1), makes the agent greedy; it will seek out whatever reward is closest. A high gamma value encourages the agent to seek rewards past the next state. This increases accuracy at the cost of time; it may take longer for the Q values to converge.

We used epsilon-greedy approach. Epsilon represents the exploration vs exploitation factor. More specifically, it represents the probability that the agent will choose to explore rather than exploit. At each time step, the agent can either choose the path that leads to the highest expected utility (given by the maximum Q value), or it can choose a random action. The first option is exploitation, and the second option is exploration. The epsilon value decreases over time, which means that in the beginning the agent will favor exploration, while in the end the agent will favor exploitation. The “Epsilon Constant” that we chose for this problem is inversely proportional to the epsilon value.

We chose the values of **Learning Rate Constant = 10, Gamma = 0.7, and Epsilon Constant = 1000.** The learning rate is defined as **Learning Rate Constant/(Learning Rate Constant + N(s,a)).** The epsilon value is defined as **Epsilon Constant/(Epsilon Constant + EpochNum).** The values of learning Rate Constant tried are (10,100,1000,10000). The values of Gamma tried are [0.1,0.2,0.4,0.7]. The values of Epsilon constant tried are [10,100,1000,10000]. We ran the Q learning algorithm multiple times with all the combinations of parameters, and values that provided the highest number of average hits for the 12x12 case were chosen. The parameters were reused for the 16x16 and 18x18 cases.

Results:

We found that our agent learned a good policy after 100000 training games. After training, we ran the agent for 1000 test games and reported the average number of bounces in those games. In addition to using a grid size of 12x12, we used grid sizes of 16x16 and 18x18 too see how it would affect the results.

When running our program with the parameters given above, we achieved the following results:

12x12 Grid : 11.779 average hits
16x16 Grid : 13.503 average hits
18x18 Grid : 13.687 average hits

Changes to the MDP:

The changes we made to the MDP involved altering the constants used to discretize the continuous state space. In the original problem, the state space was discretized into a 12 by 12 grid. We tested two other grid sizes, 16x16 and 18x18.

The average number of hits for the 16x16 grid size was hits, which was a little better than the 12x12 case. However, the 18x18 discretization was the most successful of all, averaging hits over 1000 games. This make sense, because the finer the grid resolution, the more closely it represents the actual state space.

When using Q learning, the agent will learn to apply the same action regardless of where the ball is within a certain grid cell. For instance, if the ball's coordinates are (.5, .5), this will discretize to the same location as (.5,.56) for a 12x12 grid. This means, given a certain paddle position and ball velocity, the agent will take the same action at (.5, .5) and (.5 ,.56). At first, this might not seem like a big problem, but it can cause dramatic effects when the ball is near the paddle. The ball might barely pass the paddle, which could have been prevented with a higher grid resolution.

Unfortunately, the main drawback of using a higher grid resolution is the training time. The 12x12 case only took 718.034 seconds for 101000 trials. The 16x16 case took ~800

seconds and the 18x18 case took ~814 seconds. With a finer grid resolution, the state space increases from 10369 in the 12x12 case to 18433 and 23329 in the 16x16 and 18x18 cases respectively.

Other changes we could have made would be to the exploration/exploitation factor, learning rate, or discount factor.

Extra-credit for Part2:

A GUI was implemented for Part2 with Python graphics libraries to visualize the agent playing a pong game during testing/evaluation phase. A short video of the agent playing pong is attached in the file: *PongVideo.mp4*.

This video shows the agent trained with the 12x12 state space playing pong after 100000 iterations of training.

Codes : All the part2 codes are in folder Part2. MarkovState.py has state space definition, updation in continuous domain and discretization related functions. GamePlay.py contained Q learning and pong game setup. Display.py and graphics.py have content for GUI. PongVideo.mp4 is the video file of agent playing. TestRL.py is top-level in which hyperparameters are set. How to run : python TestRL.py

Statement of Contribution:
Part1 : Ramya
Part2 : Ramya and Ashish independently implemented the code and shared techniques/results.
Part2 extra credit: Asish