MP4: Reinforcement Learning and Deep Learning

PENGXU ZHENG

Hockenmaier

4/21/2019

# Part 1: Q-learning (Snake)

Part 1

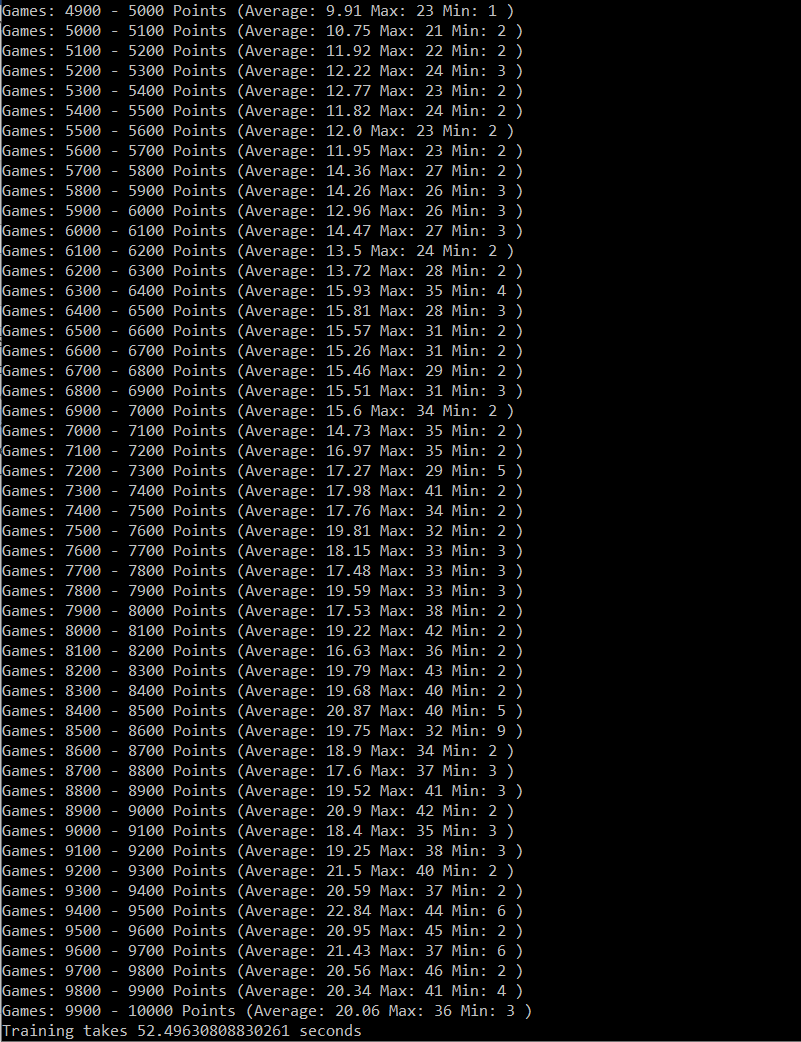
1. Briefly describe the implementation of your agent snake.
   * How does the agent act during train phase?
   * How does the agent act during test phase?
2. Use Ne, C (or fixed alpha?), gamma that you believe to be the best. After training has converged, run your algorithm on 1000 test games and report the average point.
   * Give the value of Ne, C (or fixed alpha) you believe to be the best.
   * Report the training convergence time.
   * Report average point on 1000 test games.
3. Describe the changes you made to your MDP(state configuration, exploration policy and reward model), **at least make changes to state configuration**. Report the performance (the average points on 1000 test games). Notice that training your modified state space should give you at least 10 points in average for 1000 test games. Explain why these changes are reasonable, observe how snake acts after changes and analyze the positive and negative effects they have. **Notice again, make sure your submitted agent.py and q\_agent.npy are without these changes and your changed MDP should not be submitted.**

## **1. Implementation of Snake**

### Training Phase Description:

During the beginning of the training phase the agent seems to be acting arbitrarily. It does not seem to have a good understanding of the best actions to take in the environment as is defaulting to a few poor actions. As the training continues, with in seconds you can observe the game averages increasing as show in the screenshot below. As the agent continues training you can tell it is learning and slowly taken better actions. It’s average is increasing as well as it’s max number of points.

One thing that is interesting to note though is that even though it averages about 20 points in the end, the minimum points is still only around 3. This shows that even though the agent can score well, it is still occasionally choosing paths through the Markov decision process (MDP) that are less beneficial probably because it hasn’t been on those paths very often. This is called exploration and is guaranteed with our exploration function in the code which forces the agent to visit each node in the MDP at least Ne times. This exploration although may not seem beneficial, helps the algorithm converge to a better policy.



### Testing Phase Description:

In the testing phase, the agent performs better than in training phase. In the last training run the agent scored about 20.06 average, however in the test run it scored 21.898 average, as shown in the screenshot below.

This performance improvement is attributed to the fact that while testing there is no exploration function, so the action is chosen simply by a greedy policy that choses the action with the maximum Q value in the Q-table. This results it more consistent actions with a higher average score since the agent is not taking actions it is unsure about, according to its Q table. However, it can still be seen that the minimum points of a game were 2 points. This indicates that the Q table probably has not converged to an optimal policy and the agent should be trained for longer, since some of its Q-values still lead to suboptimal actions.

## **2. Modifying Parameters**

### Testing different parameters:

The table below shows different parameters that were tested. Essentially, I started with the parameter configuration from the checkpoint resulting in the best average points and modified slightly each parameter while keeping the others consistent. This experimental method is slightly arbitrary but allowed me find model with an average of 24.15 points after only 10,00 training runs. This result was further run for 100,000 training games to a Q table that is converged (or very close) to a true optimal policy

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C | Ne | Gamma | Alpha | Average Points after 10,000 training runs |
| 80 | 20 | 0.5 | C/(C+N(s,a)) | 20.723 |
| 70 | 22.148 |
| 60 | 21.898 |
| 50 | 24.149 |
| 40 | 22.38 |
| 30 | 20.129 |
| 20 | 20.192 |
| 10 | 23.3 |
| 50 | 30 | 16.513 |
| 40 | 11.832 |
| 50 | 10.567 |
| 25 | 18.136 |
| 35 | 13.326 |
| 20 | 0.6 | 23.172 |
| 0.7 | 19.906 |
| 0.8 | 20.998 |
| 0.4 | 22.559 |
| 0.3 | 22.096 |
| 0.5 | 0.3 | Infinite loop with snake repeating states  See [Trying Constant Alpha](#_Trying_Constant_Alpha:) |
| 0.5 | 0.4 |
| 0.5 |
| 0.6 |
| 0.7 |
| 0.8 |

### Best Parameters:

C is value used to determine alpha: 50

Ne is maximum number of visits required for exploration: 20

Gamma is discount factor of future rewards in Bellman equations: .5

Alpha is learning rate: C/(C+N(s,a)) (where N(s,a) is the number of times action a was taken in state s)

### Convergence Time:

981.65 seconds = 16.427 minutes

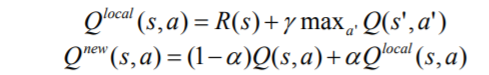
This convergence time was after running 100,000 training games; however, I would say that this policy would actually take less time to converge. After running about 30,000 games its average was consistently around 25, indicating it is near convergence

### Average Points:

Average points: 25.562, after running 100,000 training games

### Trying Constant Alpha:

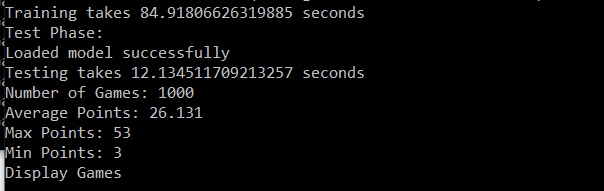
Notice how in the [Testing Different Parameters](#_Testing_different_parameters:) section, when a constant alpha was tested, the result was an infinite loop during the training process with the snake just repeating states. This is troublesome and makes the snake impossible to train, however it kind of makes sense. It seems that it would be much more difficult for the Q learning algorithm to converge if the learning rate was constant. A constant learning rate means regardless of how often you have visited the state-action pair, the Q-value should always be updated by the same percentage of the old Q-value and the new. Whereas, when alpha goes to zero as the n (the number of times you visit the state) goes to infinity, the values in the Q table would begin to converge to the true value. This is apparent in the interpolated form where you can see the Qnew as percentage of the current Q value, Q(s,a), and the new updated value, Qlocal(s,a).



## **3. Changes to the MDP**

The state discretization was changed so that the indices (states) of the Q-table indicating if there is a wall near the snake head was combined with the state indicating if there was a portion of the snake body near the snake head. When observing the games, it seemed to me that regardless of whether there is a snake body part or wall right next to the snake, it should avoid taking that action, so I thought it wasn’t necessary to represent a wall and body part by the snake head as being two different states. We know that if there is a body part to the left of the snake head there cannot be a wall and vice versa, so combing these into one 4 states (indicating if there is an obstacle above, below, to the left, or right of the obstacle), effectively eliminating a redundancy in state representation, made the algorithm converge significantly faster. I do not see any negative sides with this new state configuration, besides potentially the possibility that snake should actually act different when it is one a way from a wall versus when it is one away from a body part, however, this seems unlikely since the snake’s state space can only see one step ahead.

Average Points: 26.131, after 10,000 training rounds and 1000 test runs. This is higher than previous state configuration!



# Part 2: Deep Learning (MNIST Fashion)

Part 2

1. Briefly describe any optimizations you performed on your functions for faster runtime.
2. Report Confusion Matrix, Average Classification Rate, and Runtime of Minibatch gradient for 10, 30, 50 epochs. This means that you should have **3x3=9** total items in this section.
3. Add a graph that plots epochs vs losses at that epoch. (For 50 epochs)
4. Describe any trends that you see. Is this expected or surprising? What do you think is the explanation for the observations?
5. Report Extra Credit section, if any.

Run your minibatch gradient for 10, 30, and 50 epochs. You should start with clean generated weights/biases for each run. For each run, report:

1. Confusion Matrix
2. Average Classification Rate
3. Runtime of your minibatch gradient function

At the end of 50 epochs, include a graph that plots epochs vs losses.  
Also report any interesting observations, and possible explanations for those observations.

## 1. Optimizations:

Most of the optimization were done through the numpy library. A lot of the formulas for the neural network forward propagation, backward propagation, and activation required summing over many different values in a multidimensional array. Instead of writing loops, using the numpy functions below helped run the code better, since numpy has its own low-level optimizations in place.

|  |  |  |
| --- | --- | --- |
| Function | Purpose | Use |
| np.matmul: | Standard matrix multiplication | Useful to transform input to an output through weights of the network |
| np.transpose: | Transposes matrix | Used with mat.mul, when rows need to be summed with other rows |
| np.sum: | Sums array along specified axis | Helped sum exponentials for loss function and sum derivatives in backpropagation |
| np.where: | Changes individual array values based on a condition | Used in activation functions to zero out negative values |

## 2. Results for 10,30 and 50 Epochs

### 10 Epochs

#### Confusion Matrix:

A screenshot of a cell phone

Description automatically generated

#### Average Classification Rate:

.857 correct images/total images

#### Runtime:

78.63 seconds or 1.31 minutes

### 30 Epochs

#### Confusion Matrix:

A screenshot of a cell phone

Description automatically generated

#### Average Classification Rate:

.8859 correct images/total images

#### Runtime:

239.9 seconds or 3.998 minutes

### 50 Epochs

#### Confusion Matrix:

A screenshot of a cell phone

Description automatically generated

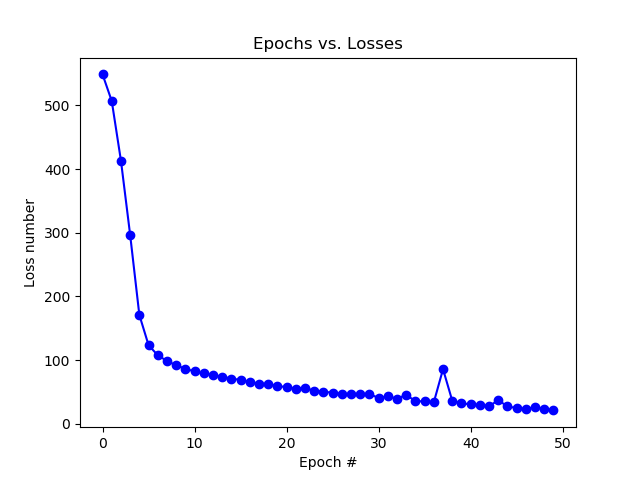
#### Average Classification Rate:

.8908 correct images/total images

#### Runtime:

448.46 seconds or 7.48 minutes

## 3. Graph Epochs vs Losses for 50 Epochs:



## 4. Trends Summary and Explanations

It seems like the more epochs you train neural network for, the more accurate the classifications should get. This is also shown in the graphs for Epoch’s vs Losses. As the epochs increase the loss decreases, which means that the difference between the score the neural net gives a token for classifying it as a certain label and the correct label is decreasing.

One thing that surprised me was the variations when training the neural net with different number of epochs. For example, when training the network for 10 epochs, occasionally it would result in a 58% accuracy and get 0% classification rates for 2 of the classes. Sometimes it runs with a reasonable accuracy like 78% or 85%. A few times it was even more accurate then the 50 epochs run with an accuracy of 89%.

There was less variation for 30 and 50 epochs but still some inconsistencies. A few times 30 epochs scored a higher accuracy than the 50 epochs, which theoretically seems like it shouldn’t happen. More specifically 30 epochs seem to be getting consistently an accuracy of 88-90%, and 50 epochs seems to consistently get an accuracy of 86-89%.

The explanation for this variation in runs is probably due to the random initialization of weights. It is the only factor that is nondeterministic from run to run and could possibly be leading the loss function to different local minimums. The gradient descent method we use essentially uses the derivative of the loss function with respect to the weights to minimize the loss, however the loss function could have different local minima. With different initial conditions the algorithm could be approximating different local minima, and since these are still minima of the loss function the classification accuracy is relatively high yet still inconsistent from run to run.