RL Hands-On Session: Solutions Manual

Dr. Balaraman Ravindran,
Department of Computer Science and Engineering,
Indian Institute of Technology, Madras

August 29, 2017

1 Getting started

- You will need the latest version of Mozilla Firefox, which you can download from this link.
- Get the code for the experiments here. Click the "Clone or download" button and select "Download ZIP". Once the download is complete, extract it to a folder of your choice. Any directories/paths referred to will be relative to this folder.

2 TD Learning Parameters

In the following activities, we study the effect of several parameters that appear in TD learning on the agent's learning. The environment used to conduct tests is a grid world, where an agent needs to go from the starting square to the goal square (the only one with a +1 reward near the center). There are obstacles that the agent cannot enter, and other squares that penalize the agent (-1 reward) that it must avoid.

2.1 Discount factor

1. Open `gridworld_td.html` in the `TD` directory with Firefox. If you like, you can review Q-Learning and Sarsa using the material on the same page.
2. Set the `spec.planN` parameter to 0. Do this in all future runs, as we will not be using planning in this series of experiments. You can enter this parameter along with others in the text box at the top.
3. Initially, set the `spec.gamma` parameter to 0.1.
4. Start algorithm execution by first clicking on the red "Reinit agent" button to update the parameters and then the blue "Toggle TD Learning" button.

5. As you observe the execution of the algorithm in the grid world domain (the arrows describe the policy it learns at each square), keep an eye on the corresponding graph being generated below. This graph plots the steps taken to reach the goal for each episode played.

6. You will note that after spiking continually for a while, the graph settles down to a small value on the Y-axis as the algorithm continues execution. *What do the initial spikes in training correspond to?* Exploratory moves that it makes until it can find the optimal path (or something close).

7. Pause algorithm execution (by clicking on the blue "Toggle TD Learning" button) after about 500 episodes (on the X-axis).

8. Now, we will repeat the algorithm execution from the initial state. To do so, click on the red "Reinit agent" button. Do not refresh the browser window as you will loose the results of the previous execution being displayed in the graph.

9. Now, start algorithm execution from the start (by clicking on the blue Toggle TD Learning button).

10. Allow the second instance of algorithm execution until the graph nears about 1000 on the X-axis. This time, to speed things up, click the "Go fast" button.

    You will observe that graph generated on the second run is similar to but not exactly the same as the graph generated on the first run. This is due to the stochasticity inherent in the algorithms. This is a feature of most RL algorithms and when comparing results of RL algorithms, it is a good idea to conduct multiple executions of the same algorithm with the same parameters and environment and calculate the average value of different measures of performance (or using the average values to plot a graph).

11. Next, change the `spec.gamma` parameter to 0.9. Refresh the page to clear the graph.

12. Update the new parameter by clicking on the red "Reinit agent" button and then start algorithm execution by clicking on the blue Toggle TD Learning button.

13. Comparing the execution with the two different values as done above should convince you that convergence happens faster than in the case where `spec.gamma` $= 0.1$. *What is the reason behind the faster convergence?* With higher γ (i.e lower discounting), reward information from states far ahead can effectively reach past states earlier. This is particularly effective here since the reward is in only one place that too far away.

2.2 Learning Rate Parameter

1. We will now focus on the learning rate parameter. Refresh the browser page to clear the graph and obtain default parameters. Remember to set `spec.planN` to 0.

2. Note that the `spec.alpha` or learning rate parameter is set to 0.1. Execute the algorithm until the graph extends to around 300 on the X-axis and then pause execution.

3. Now, change the `spec.alpha` parameter value to 0.9. Re-initialize the agent and execute the algorithm.

4. Repeating the experiment a few times should convince you that convergence is faster with a higher value of the learning rate parameter. *How can you explain this observation?* The higher learning rate means values are updated more toward their recent (and likely better) estimates, thus leading to faster learning.

5. With the same parameters, reinitialize the agent and execute the algorithm, but this time, click the "Go slow" button. Focus on the square diagonally above and to the right of the square with default reward of 1. Notice that the color of the square (you can also consider the actual values displayed) first becomes dark red and then subsequently changes to light red.

   How can you explain this observation? Does the same happen when the ***`spec.alpha`*** *value is 0.1? From these observations, what trade-off do you think exists between a low and a high value of the learning rate parameter?* A low learning rate leads to slow updates, so convergence takes a long time. A high learning rate leads to unstable value estimates for many states since it may cause the updates to overshoot the necessary amount.

2.3 Exploration Parameter

1. Let us now focus on the exploration parameter, `spec.epsilon`. Refresh the browser, set `spec.planN` to 0, re-initialize, and execute the algorithm with default value of `spec.epsilon` = 0.2 till convergence.

2. Now change the value of `spec.epsilon` to 0.8 and re-execute the algorithm.

3. You will notice that the after convergence, the average values (Y-axis) with `spec.epsilon` = 0.8 are higher when compared to the values when `spec.epsilon` = 0.2. How do you explain this observation? Even after convergence, in one case exploration is still high resulting in more steps to reach the goal state whereas in the other case on average less number of steps are required due to lesser explorative steps.

4. Continuing execution with a higher value of `spec.epsilon`, once convergence occurs, slowly (allow 100 units on the X-axis before each 0.1 decrease) decrease the exploration parameter (as the algorithm is executing) by using the slider underneath the buttons.

5. *What effect on the output do you observe on gradually decreasing the value of the exploration parameter?* Lesser number of steps to reach the goal. *Is this a desirable behavior?* Typically, yes. *Are there any situations you can think of when such behavior is not desirable?* Non-stationary environment/problem: it can't learn about the recent behavior of the environment once the exploration has been reduced. *If the exploration parameter value is brought down to zero, would the algorithm always follow an optimal path to the goal state?* Depends upon the optimality of the solution found before reducing parameter value to zero, as well as stochasticity of the environment. It likely will, for deterministic problems such as this one.

2.4 Eligibility Trace Decay Factor

1. Finally, we will focus on the lambda parameter. Refresh the browser page, set spec.planN to 0, reinitialise, and execute the algorithm with default value of spec.lambda = 0 till convergence.

2. Now change the value of spec.lambda to 0.9 and reexecute the algorithm.

3. A few iterations should convince you that execution with a high lambda value results in faster learning. To understand why, scroll down the page to the heading "TD Bells and Whistles"and observe the eligibility traces figure.

   Typically, when we observe a reward, we update the vlaue function of only a single state. In contrast, eligibility traces allow us to update the value functions of many preceding states, with the lambda parameter controlling the magnitude of the updates. For example, in the figure, the first grid world shows a path taken by an agent with the state marked with an astrisk being a positively rewarding state. When we do not use eligibility traces, on reaching the positively rewarding state, the value function of only the previous state is updated (second part of the figure). However, when eligibility traces are used, as can be seen in the third part of the figure, the value functions of all states in the path are updated. This should explain the observation above where training took much less time when eligibility traces were used (i.e., the value of the lambda parameter was set to a value greater than 0).

4. *In the third part of the figure, we observe that the magnitudes of the updates indicated by the size of the arrows decrease the further back the path we go. What do you think is the intuition behind this?* More importance to recent actions and less importance to earlier actions.

3 Q-Learning vs Sarsa

This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. This is a standard episodic task, with a start and goal state, and the usual actions causing movement up, down, right, and left. It is also called a cliffworld, with the grey cells being the 'cliff'. Falling down this cliff gives the agent a large negative reward and the resets it to the starting state. The optimal path is such that the agent needs to do a 'cliff walk' by walking along the edge of the cliff.

3.1 Experiment

1. Open `gridworld_td.html` in the `ql-vs-sarsa` directory.
2. As usual, set `spec.planN` $= 0$. (Scroll to the end of the page to see the 'Planning' sub-section under 'TD Bells and Whistles' to know why. It is quite interesting!)
3. Reinitialise the agent, and toggle learning.
4. Watch the agent learn how to go to the goal state at the bottom-right corner of the gridworld. You can vary the 'Exploration epsilon' by sliding the bar in case there are any unexplored (white) states. Simply raise the value to 0.9, hold it for a while and bring it back down again.
5. Once the agent converges (after about 800 episodes, check the graph), slide the exploration to 0 in order to observe the policy being followed. You'll see that the agent chooses the most optimal (shortest) route to get to the goal.
6. Now, repeat this procedure, but using Sarsa. Change the algorithm to Sarsa by changing `spec.update` to `'sarsa'`.

Now we see that instead of choosing the shortest route to the goal, the agent learns the a roundabout route and moves as far away from the cliff as possible before turning towards the goal. This is a "safe" policy like a human might follow.

Why does SARSA learn this and not the optimal policy in contrast to Q-Learning? Can SARSA be used to find the optimal policy at all? If so, how? Sarsa being on-policy, learns about the same policy that it uses to generate actions for exploration. Thus, it does not learn an optimal greedy policy, but rather an optimal ϵ-greedy policy. Q-Learning on the other hand learns about the optimal greedy policy while following the ϵ-greedy policy. Here, the optimal ϵ-greedy policy would try to put some distance to the cliff due to the possibility of taking a wrong action at each step (going down instead of right).

4 The Necessity of Experience Replay for DQNs

The aim of this experiment is to show how experience replay is an essential component of DQNs. You will do this by disabling experience replay in the DQN agent's code, so that it uses only plain Q-learning updates with the same Neural Network (NN) action-value function approximator and then observing what happens while training.

The DQN agent operates in a continuous, fairly high dimensional state space with a discrete action space. It uses a neural network with two densely connected hidden layers of 100 units each, with tanh activations as its value-function approximator.

4.1 Experiment

Note: Throughout this experiment as well as the next, do not press the arrow keys while focused on the experiment's tab since there seems to be a bug that starts another agent. You can actually control this agent using the arrow keys, but it would skew the results since it competes with the DQN.

1. Open `waterworld.html` in the `DQN` directory with Firefox and *quickly* click on the "Go very fast" button (which appears immediately after the text box with parameter settings). Then click on "Toggle Agent View". Now the vanilla DQN has been set to train.

2. As it trains, familiarize yourself with the environment details in the "Setup" section at the bottom of the page. Let it train for 350,000 iterations. You can check its progress from the Javascript console, which can be accessed in Firefox with the key combination `Ctrl+Shift+K`.

3. In the meantime, open the `rl.js` file in the lib sub-directory with your IDE/text editor. The relevant parts of the code are in the function `learn` in the `DQNAgent` prototype. To easily find this, search for the comment "DQN Learning Function". Study its code to get an overview of the DQN implementation. It is also worth looking at the `learnFromTuple` function which is the very next function in the file. It takes a (state, action, reward, next state, next action) tuple and performs a stochastic gradient descent update on the NN parameters. It also returns the TD error for that update.

4. Identify the relevant parts of the code to comment out so that experience replay is disabled.

5. Set it to train as in step 1, but this time since the training goes much faster, wait until the "Poison" counter under the environment display goes up to 2000.

The graph below shows the percentage of apples in the most recent 100 "digestions" (which could be either apples or poison) smoothed over time. The higher it is, the better the agent has performed. Once the agents finish training, save a screenshot of the graph for future reference. Also note the apples and poison counts that the agent has achieved.

You can observe the behavior of the trained agents by clicking on "Go normal" and "Toggle Agent View" again. If all has gone well, the vanilla agent should have learned some of the requisite behaviors such as avoiding the poisons and gravitating towards the apples when it can see them.

With experience replay disabled on the other hand, the agent is unable to train, as would be evident from random/degenerate behavior like staying in the same place and not responding to the presence of apples/poison.

4.2 Discussion

You might observe that your trained vanilla DQN agent tends to get stuck against walls/corners, and even when many poisons are heading toward it, it makes no move to escape away from the wall. Only when it sees an apple will it do so. However, once it isn't near the wall, it dodges poisons with much better skill than before. *What could be the cause of this?* The agent knows only its velocity, and not its position. It treats all points in the space equally, but quite clearly walls and corners are different, and the agent has no way of distinguishing between them. This is the problem of partial observability. The true state, which would include the position of the agent, is not available to the agent, but rather only some part of the state is revealed in the observations it gets.

4.3 Remarks

It is still possible to train very small NN function approximators without experience replay. In the original code on which this code is based (ReinforceJS), only one hidden layer is used for the NN, and testing showed that it still trained when experience replay was disabled.

It should also be noted that one of the other major innovations used in the original DQN is missing: here, we simply use the same network to compute the target as opposed to freezing the network. According to the original author, Andrej Karpathy, there seemed to be no tangible benefit to freezing the network in this environment, so it was left out for brevity.

5 Prioritized Replay for DQNs

In DQNs, all transitions stored in the replay memory are treated as equal. Obviously, some transitions can help the agent learn more about the environment than others. This experiment demonstrates the effect of accounting for this by adding a feature called *prioritized replay*.

There are several ways to assign preferences to experiences as described in Schaul et. al. (2015). Here, we consider one such way called *proportional prioritization*. To decide which transition is more effective, the magnitude of the TD error δ is used as a metric. This means assigning more importance to transitions that cause surprise. To each experience, a weight of the form

$$w_i = (|\delta_i| + b)^\alpha$$

is assigned. Now, instead of choosing uniformly at random, the experiences are chosen with probability

$$p_i = \frac{w_i}{\sum w_i}$$

Thus, an experience with high TD error is more likely to be chosen. The constant b is added so that no transition is completely ignored (i.e has zero probability).

Each time a particular experience is chosen, its TD error needs to be updated to whatever is computed using the current network. In this implementation, we simply store the corresponding weights in another circular buffer that is updated synchronously with the experience buffer.

5.1 Experiment

The following modifications are to be made. The relevant part of the code is the same as in the previous experiment.

1. Store the weights w_i computed according to the above formula in a circular buffer `this.sampling_weights` which has already been initialized for you. You can update it the same way as the experience buffer `this.exp`, i.e use the same index `this.expi`. The TD error has also been previously computed. For the formula itself, α has been declared as `this.sampling_exponent` and b as `this.sampling_base_weight`. They are set up to take values from the parameter check box. Remember to use the absolute value of the TD error.

2. In the experience replay code, replace the uniform choice with a weighted choice. To do this, you can use the following code

   ```
   chance.weighted(this.exp, this.sampling_weights);
   ```

which will return one experience sampled according to the weights that you can store in a variable (say `e`). Note that the library takes care of normalizing to compute p_i for you. A learning update is done as before, but this time, after the update, the sampling weights need to be updated for the new TD error. You can get the index of the experience e that was sampled as `e[5]`, and update the weight at that index using the same formula as before.

Train this again for 250,000 iterations with the values $\alpha = 0.7$ and $b = 10^{-3}$ that are set by default. Compare the performance graph earlier to this. You will find prioritized replay has caused it to learn faster, as well as attain a higher scoring rate more quickly as also evidenced by the apple and poison counts.

5.2 Discussion

What makes this work? That is, what property of the environment causes the agent to benefit from this modification? Rewards are sparse, and are attained only at those instants where the agent swallows something. There are many more "mundane" transitions where there is no reward. With uniform sampling, the agent is less likely to learn from data that is already scarce. However, with prioritization, the agent can pay more attention to the interesting transitions.

5.3 Remarks

Strictly speaking, prioritization causes the learning experiences to be drawn from a different distribution than what is actually observed (which is the uniform case). So an importance sampling factor is actually necessary to correct for this distribution mismatch, which needs to be multiplied to the gradient update (see the paper referenced earlier for more details). However, this domain doesn't seem to be complex enough to be affected by the omission of this step.