ASSIGNMENT 1: TABULAR REINFORCEMENT LEARNING

Van Nguyen
s3726266

1 DYNAMIC PROGRAMMING

In this part we implement the Q-value iteration algorithm in which we sweep through all state-action pairs and update the estimate of a its value based on this equation:

$$Q(s,a) \leftarrow \sum_{s'} \Big[p(s'|s,a) \Big(r(s,a,s') + \gamma \max_{a'} Q(s',a') \Big) \Big] \tag{1}$$

- $Q(s,a)$ Q-value of a state-action pair - indicating how good is it to take action a in state s
- $p(s'|s,a)$ Transition probability - representing the chance of reaching state s' from state s when action a is taken
- $r(s,a,s')$ Immediate reward after transitioning from state s to state s' as result of action a
- γ Discount factor in range $< 0, 1 >$ used to decrease the value of future rewards
- $\max_{a'} Q(s',a')$ Maximum estimated Q-value for the state s' over possible actions a'

1.1 IMPLEMENTATION

We first we finish functions in DynamicProgramming.py

`QValueIterationAgent` In this class we needed to implement three functions `init()`, `select_action()` and `update()`.

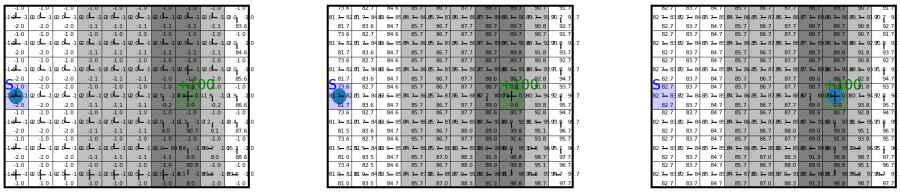
- The `init()` is construct method to initialize variables such as number of action, number of states, gamma discount, threshold and more importantly setting table with means to $Q(s,a)$.
- `select_action` function implements a greedy policy $\pi(s) = \arg\max_a Q(s,a)$ using `np.argmax(q_values)` which returns the index of athe action with the highest Q-value in state s.
- `update()`: In this function, we calculate the new Q-value for the state-action pair (s,a) using the Bellman equation as shown in Equation 1. The implementation in code is as follows:

```
q_value = np.sum(p_sas*(r_sas+self.gamma*np.max(self.Q_sa, axis=1)))
```

Where p_sas is the transition probability and r_sas is the immediate rewards. Maximum absolute error is calculated as well as `np.abs(self.Q_sa[s,a] - q_value)` calculating the absolute difference between the old Q-value and the new Q-value.

1.2 RESULTS

Analysing the progression in Figure 1 from the *End state* to *Beginning state* through *Middle state* we observe the evolution of the state values as it goes from non-optimal policy to the optimal policy.



(a) State of DP at the start state (b) State of DP at the mid state (c) State of DP at the end state

Figure 1: Three states side by side

The cells that are immediately adjacent to the goal(green square) have higher values adjusted by the γ discount factor. As we move further away from the goal each cell's value becomes the maximum of the values of it s adjacent cells, showing the recursive character of Bellman optimally equation 1. This results in decrease of cell's values going outward from the goal state to the start S. Showcasing the principle of future diminishing with distance and time from the goal.

When computing $V*(s=3)$ at the state $s=3, location=(0,3)$ the converged optimal values is **83.678**. This indicates the expected total discounted reward from starting in end state.

To compute **average reward per time step** under policy we assume few things:

1. Optimal value at the start state, $V*(s=3)$ which is as mentioned 83.768.
2. Magnitude of the final reward - gained from environment +100 denoted as R
3. Magnitude of the reward on every other step - gained from environment -1 (from beginning state) denoted as r

First we need to compute number of steps which done as $V*(s=3) = r(n-1)+R$ where n is number of steps. After plotting we get to $83.678 = -1(n-1)+100 \approx 17.322$ since n has to be integer we n ≈ 17.

Average reward per time step is then calculated as *Average reward per time step* = $\frac{R+r(n-1)}{n}$ which is ≈ 4.941. Meaning the average reward per time step under optimal policy is approximately **4.941**.

In my `update()` function I use the Bellman equation to update the Q-values. If the agent reaches terminal state, the p_sas and the r_sas returned by `env.model(s,a)` would yield zero values for Q-value and for any action as well. Q-values and any action being at zero follow the definition of a terminal state.

One way to handle terminal state in dynamic programming would be a usage of a terminal flag. If for example we check if the state is terminal before we update, we can apply the terminal reward without considering future actions.

Pseudo code for the terminal flag:

```
if is_terminal(s):
    Q_sa[s, :] = terminal_reward
else:
    # Proceed with the regular Bellman update
    Q_sa[s, a] = update_using_bellman_equation()
```

When moving the goal state to $[6,2]$ the whole process of calculating the optimal policy takes much longer. In comparasion when the goal state was at $[7,3]$ the calculation took 27 iterations whereas the computation for the new goal state took 146 iterations with mean reward per time step under optimal policy with **6.769**.

This can be explained with this environment being a Stochastic Windy Grid world. Certain paths have higher variance due to the "wind" effect. The new goal state is simply situated in such way where the agent has to navigate with high variation path which slows the process of finding the optimal policy.

2 EXPLORATION

2.1 IMPLEMENTATION

In this section I'll talk about how I implemented `q_learning()` and its `update()` function.

`update()` in Q_learning.py For Q_learning we need to use tabular learning update which is denoted as:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)]$$

Here $Q(s_t, a_t)$ represents the estimated Q-value for taking action a_t in state s_t. G_t as teraget values which is calculated on reward r and maximum expected future reward in s_{t+1} discounted by factor γ, the α denoting learning rate which limits how much of new information overwrites the existing estimates.

This is the implemented as:

Algorithm 1 Q-learning update function

Result: Update Q-value for state-action pair
initialization **while** *not at the end of the episode* **do**
  Calculate the backup estimate/target G_t
  $G_t \leftarrow r + \gamma \cdot \max(Q[s_{next}, :]) \cdot (1 - \text{done})$
  Apply the tabular learning update
  $Q[s, a]\alpha \cdot (G_t - Q[s, a])$
end

The `update()` function has two main steps:

- First we calculate G_t which is the sum of the reward r nad the discounted maximum Q-value of then next state s_{next} Maximum q-value is taken across all possible actions from s_next resulting in best possible reward.
- Application of the Q-value update - for $[s, a]$ which stands for current state-action pair. We update by moving it towards the target value G_t. The learning rate (α controls how much of update we do).

`q_learning()` Q_learning is then implemented as following:

Algorithm 2 Q-learning algorithm

Result: Update Q-value for state-action pair
for *t from 1 to $n\_timesteps$* **do**
  Select action a using the specified policy Obtain next state s_{next}, reward r, and termination signal *done* by taking action a in the environment Update Q-value using the Q-learning update rule $agent.update(s, a, r, s_{next}, \textit{done})$
  if $(t + 1) \mod \textit{eval\_interval} == 0$ **then**
    Perform evaluation of the agent on the evaluation environment Append the timestep $t + 1$ to the list of evaluation timesteps Calculate the mean return from the evaluation and append it to the list of evaluation returns
  end
  Update current state s to s_{next} if the episode has not terminated, otherwise reset the environment
end

The main loop iterates over specificed number of n_timesteps and in each of them:

- Selects action under specified policy

- Takes action a and observe the next state s_next, reward r and end signal *done*
- Q-value update - updates the Q-value pair using `update()`
- Evaluate the agent's performance if the timestep is a multiple of the evaluation interval *eval_interval*. Append the timestep to the list of evaluation timesteps and calculate the mean return appending it to the list of evaluation returns.
- Update the current state s to s_{next} if episode is not terminated otherwise reset the environment.

2.2 Results

After running Experiment.py in particular Assignment 2: Effect of exploration we get this graph as result:

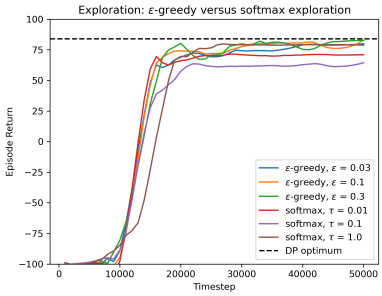


Figure 2: Exploration policies

From the graph we can interpret that the best performing ϵ-greedy policy is the one with $\epsilon = 0.1$ the best softmax policy is the one with $\tau = 0.1$. Both of these converge quickly but the best performing one is the **ϵ-greedy policy with $\epsilon = 0.1$** as it converges the fastest(reaches close to DP optimum in fewer time steps) and is the most stable one of all policies.

The reason this policy does not achieve DP optimum is probably due to not being able to fully converge within given time steps we have.

The overall performance is much better in sense of reaching a peak much faster than one goal scenario. This is due to second goal being added as it increases the chances of receiving reward. After running Experiment.py I compared the two graphs and compared to the one goal the learning rate is much faster but it does not get better after reaching 0 on y-axis as opposed to Figure 2. The second goal also affects the exploration-exploitation trade-off by not giving it incentive to explore as the agent learns to exploit the second goal's rewards. The optimal exploration parameter is probably much lower as the agent learns to exploit the second goal which does have smaller reward goal than the first one (5 as opposed to 100).

3 Back-up: On-policy versus off-policy target

3.1 Implementation

In this part we implement SARSA which is denoted as:

$$G\_t = r\_t + \gamma \cdot \hat{Q}(s_{t+1}, a_{t+1}) \tag{2}$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)] \tag{3}$$

We implement SARSA in two functions, `update()` and `sarsa()`.

`update()` As denoted in Equation 3 we implement the update function with following algorithm:

Algorithm 3 SARSA update()

Result: Update Q-value for state-action pair using SARSA
State s, action a, reward r, next state s_{next}, next action a_{next}, termination signal *done*
Updated Q-value $Q[s, a]$
$G_t \leftarrow r + \gamma \cdot Q[s_{next}, a_{next}] \cdot (1 - \text{done})$

$Q[s, a]\alpha \cdot (G_t - Q[s, a])$

`sarsa()` implemented as:

Algorithm 4 Policy Update and Evaluation Loop

Result: Update policy and evaluate agent periodically
for *timestep in range(n_timesteps)* **do**
  Obtain next state s_{next}, reward r, and termination signal $done$ by taking action a in the environment Select next action a_{next} using policy π Update policy π based on observed transition $(s, a, r, s_{next}, a_{next}, done)$
  if *done* **then**
    Reset environment and select initial action a using policy π
  end
  else
    Update current state s to s_{next} and current action a to a_{next}
  end
  if *timestep % eval_interval == 0* **then**
    Evaluate the agent's performance on an evaluation environment using a greedy policy Append the evaluation return to the list of evaluation returns Append the current timestep to the list of evaluation timesteps
  end
end

The `update()` is standard implementation of Equation 3 for `sarsa()` itself we implement the first equation 2 First we loop over $n\_timesteps$ and at each time step agent selects an action a based on current policy pi. The environment returns s_next (next state), the reward r and if episode ended then termination signal $done$. After transitioning to next state and next action a_next the policy updates using `update()`. If the episode hasn't ended then we update the action and the state. At certain time steps we evaluate the performance.

3.2 RESULTS

Based on Figure 3 we can say that the preferred method is q_learning with a learning rate of α= 0.1. To interpret the results further we can look at SARSA and Q-learning at 0.3 learning rate. They learn much faster than our best chosen method but around return 60 there is big fall off. Whereas our chosen method learn slower but is much more stable. The worst performing learning rates for both methods are at α = 0.03 naturally the learning rate indicated the slow learning.

The case where would prefer SARSA over Q-learning is when the environment is non-stationary. Meaning if the environment changes often then using SARSA would benefit more because it's updating the policy to make next decision.

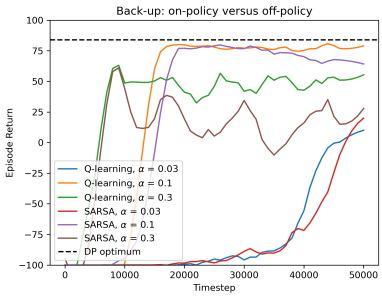


Figure 3: On-policy vs off-policy

4 Depth of Target

4.1 Implementations

First we need to implement n-step update and then Monte Carlo update as well. The n-step update is denoted as:

$$G_t = \sum_{i=0}^{n-1} (\gamma)^i \cdot r_{t+i} + (\gamma)^n \cdot \max_a Q(s_{t+n}, a) \tag{4}$$

Monte Carlo update is then denoted as:

$$G_t = \sum_{i=0}^{\infty} (\gamma^i \cdot r_{t+i}) \tag{5}$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)] \tag{6}$$

N-step update is then implemented as:

Algorithm 5 n-step Q-learning Update

Result: Update Q-values using n-step Q-learning
States $states$, actions $actions$, rewards $rewards$, termination signal $done$, n
$T \leftarrow$ length of rewards

for t *from 0 to* $T-1$ **do**
  $m \leftarrow \min(n, T-t)$

  if $t+m == T$ ***and*** $done$ **then**
    Compute return G for terminal state: $G \leftarrow \sum_{i=0}^{m-1} \gamma^i \cdot rewards[t+i]$
  end
  else
    Compute return G for non-terminal states:
    $G \leftarrow \sum_{i=0}^{m-1} \gamma^i \cdot rewards[t+i] + \gamma^m \cdot \max_{a'} Q(s_{t+m}, a')$
  end
  Update Q-values:
  $updated\_value \leftarrow Q(s_t, a_t) + \alpha \cdot (G - Q(s_t, a_t))$
  $Q(s_t, a_t) \leftarrow \text{clip}(updated\_value, -1e6, 1e6)$
end

Loop over each time step from 0 to $T-1$. At each time step we determine m which is number of steps to take in consideration. Then we calculate the return G if state is terminal then we return G which is computed as sum of rewards up to $T-1$ - discount γ. If state is not terminal then G is calculated as sum to $t+m-1$ plus the discounted value of max Q-value of next state. Then

we update Q-value using the return G using the pair (s_t, a_t). Then we apply clipping to prevent unnecessary overflows.

For Monte Carlo the update looks like following:

Algorithm 6 Monte Carlo Update

Result: Update Q-values for state-action pairs using Monte Carlo method
List of observed states $states$, actions $actions$, and rewards $rewards$ in an episode
$T \leftarrow$ length of $states$ - 1 ; `// Terminal state is not included in updates`
for t *in reversed(range(T))* **do**
  $G_t \leftarrow 0$ **for** k, *reward in enumerate(*$rewards[t:]$*)* **do**
    $G_t(\gamma^k) \times$ reward
  end
  $state \leftarrow states[t]$ $action \leftarrow actions[t]$ $self.Q_sa[state, action]\alpha \times (G_t - self.Q_sa[state, action])$
end

We perform backward iteration through episode. Starting from $T-1$ down to the first time step (0). We calculate G_t (discounted return). At each timestep t the G_t is updated by discounting previous return (G_{t+1} by γ and adding immediate reward rewards[t]. Using this newly aquired G_t and the pair s_t, a_t we update the Q-value.

4.2 Results

In the initial stages we can see 1-step Q-learning to start learning first but then it's overtaken by 3-step Q-learning as it nearly reaches DP optimum by the end of the run. The reason why 1-step learns quickly at the start can be explained by immediate rewards which play major role in the beginning. The 3-step Q-learning is the optimum as it balances short-term and long-term rewards. 10-step Q-learning considers mostly long-term goals only. Monte Carlo performs poorly which is probably the results of high variance within full-episode runs.

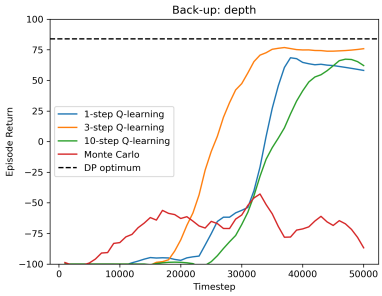


Figure 4: Depth figure

5 Discussion

DP vs RL: One of clear strengths that DP has is its ability to find optimal solution. It's exploiting the fact that optimal policy can be found by finding optimal decision at each step. In contrary DP requires the complete knowledge of the environment, its dynamics, transition probabilities, rewards to find the optimal policy. This is only possible in small scale problems such as this assignment. It would not work if the environment is stochastic.

Exploration: I personally prefer ϵ-greedy approach as it is easy to implement. The randomness of selecting new random state gives it its beauty even if it might produce temporary sub optimal solution. Another exploration strategy could be UCB (Upper Confidence Bound) or Thompson

Sampling (Ranade, 2020).

Exploration 2:

After changing the reward per step to 0 the rewards in each cell are no longer updated. Agent also seems to be stuck for longer periods at each cell. This could be because there is no longer a immediate punishment agent is less likely to try something else hence being stuck is as beneficial as exploring new cell.

Discount factor: With experiment γ - 1.0 the agent considers all future rewards of equal importance. Reward is set to 0 therefore agent will strive to reach the goal as soon as possible therefore reaching it should find the shortest path to the goal. With γ = 0.99 the future rewards are still considered but they are little bit discounted. Since the reward is set to 0 they is also no reason for agent to delay.

In summary I think both methods will reach the goal since the reward is set to 0 therefore making the γ difference very trivial.

Back-up, on-policy versus off-policy: Generally speaking on-policy strategy learns safer and converges faster its disadvantage is that might become trapped in local minima and less likely to find optimal policy. Advantages of off-policy strategy is that its more likely to find optimal policy and less likely to get stuck in local minimum. Off-policy also has ability to experience replay. Its disadvantage is that the policy learned might not be as safe and it might not perform well in online environment (mschrum3 & Gombolay, 2020).

Backup, target depth: Benefits of 1-step method is that it is simple to implement and computationally efficient. Its problem is that it only considers immediate rewards and only looks for next state which can spiral into sub optimal policy. Advantage of n-step method is that it balances benefits of 1-step and benefit of Monte Carlo, meaning they can take in rewards over n step which leads into faster learning and better value estimates. Benefit of Monte Carlo that it only returns after complete episodes, which gives unbiased estimate of action or state. Its disadvantage is that it has very high variance and if the episodes are long they learn slow as well.

Bias-variance trade-off is a concept in machine learning that talks about balance between bias and variance in predictive models. Bias measures the error which comes from simplifying a model. High bias is not good as it tends to oversimplify the model and underfit the data. Variance measures the sensitivity to small deviations in training data. High variance captures the noise and results in over fitting. The trade-off happens when decreasing one increases the other (bias and variance). Finding the optimum balance is crucial to avoiding either under fitting or over fitting (Singh, 2018).

For this task I'd say using the 3-step Q-learning suffice as it is relatively simple environment. The 1-step method propagates the information the fastest since they update values at every step. The only method that from Figure 4 that is still on the rise is the method of 3-step Q-learning so I assume this method to eventually converge given enough time.

Curse of dimensionality: One of the biggest advantages of RL algorithms is that they are relatively simple to implement and easy to understand. Another one is that since the policy and observed metrics are so explicit we can easily interpret the models and analyse the results. RL algorithms are applicable only when the complexity is low. It will run into problems specifically due to curse of dimensionality. RL methods (tabular) do not scale in large environments since the bigger the action space the bigger the table (exponentially) with dimensions. The memory required to store the table is also immense.

The curse of dimensionality refers to a problem in ML where high-dimensional data create challenges such as increased computational complexity, sparse data distribution and overall higher data requirements. This leads in harder data analysis and cases of underfitting and overfitting. This can be solved by dimension reduction methods or regularization (Shetty, 2022).

REFERENCES

mschrum3 and Matthew Gombolay. On-policy vs off-policy reinforcement learning. 2020. URL https://core-robotics.gatech.edu/2022/02/28/bootcamp-summer-2020-week-4-on-policy-vs-off-policy-reinforcement-learning/.

Gireeja Ranade. Lecture notes on reinforcement learning. https://data102.org/sp20/assets/notes/notes21.pdf, 2020.

Badreesh Shetty. What is the curse of dimensionality? *BuiltIn*, 2022. URL https://builtin.com/data-science/curse-dimensionality.

Seema Singh. Understanding the bias-variance tradeoff. *Towards Data Science*, May 2018. URL https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229.