2020

Technical University of Crete

Taxi Driver Agent with Reinforcement Learning

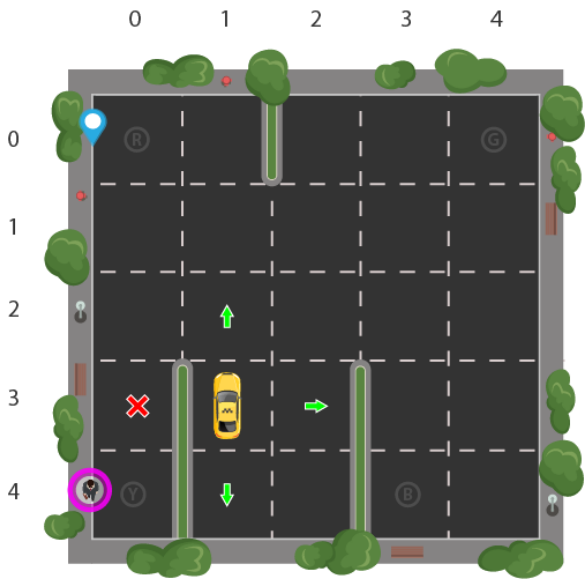
[CS513] AUTONOMOUS AGENTS

PANAGIOTIS GIANNAKOPOULOS

Contents

Problem Description

The goal is to demonstrate, in a simplified environment, how we can teach a software agent to be able to pick up passenger and drop them off to their destination without significant delays.



The taxi is the only vehicle in the parking lot and we can break up the parking lot into a 5x5 grid. There are four locations that we can pick up and drop off a passenger: R, G, Y, B. The task of the agent is to pick up a passenger from one point and drop him off to the other.

We can use Reinforcement Learning algorithms in order to develop an efficient and safe approach for tackling this problem.

<u>Agent's goals:</u>

- Pick up the passenger.
- Drop off the passenger to the right location.
- Save passenger's time by taking minimum time possible to drop off
- Take care of passenger's safety and traffic rules

In order to describe a more concrete model of the problem, we should define rewards, states, and actions.

Rewards

Since the agent is reward-motivated and is going to learn how to control the taxi by trial experiences in the environment, we need to decide the rewards and/or penalties and their magnitude accordingly. We should take into account the following points:

1. The agent should receive a high positive reward for a successful drop-off because this behavior is highly desired.
2. The agent should be penalized if it tries to drop off a passenger in wrong locations.
3. The agent should get a slight negative reward for not making it to the destination after every time-step. "Slight" negative because we would prefer our agent to reach late instead of making wrong moves trying to reach to the destination as fast as possible.

State Space

The agent encounters a state, and then takes action according to the state it's in. The State Space is the set of all possible situations our taxi could inhabit. The state should contain useful information the agent needs to make the right action.

We have the following cases:

1. The 5x5 grid of the parking lot, gives 25 possible taxi locations.
2. There are 4 locations that the agent can pick up and drop off a passenger.
3. The passenger could be in one of the 4 locations (R, G, Y, B) or inside the taxi. In this case, there are 5 possible locations of the passenger.

Eventually, the total number of possible states is $25 \times 4 \times 5 = 500$.

Action Space

The agent encounters one of the 500 states and then it takes an action. The action in our case can be to move in a direction or decide to pickup/drop-off a passenger.

In other words, we have six possible actions:

1. south
2. north
3. east
4. west
5. pickup
6. drop-off

This is the action space: the set of all the actions that our agent can take in a given state.

Depending the given environment, the taxi cannot perform certain actions in certain states due to walls. In case the agent hit a wall, the position of the taxi will not change

and the agent will be penaltized. In this way, the agent will consider going around the wall when he will be in the same situation the next time.

Open AI Gym Taxi v3 API

The simulation of the given problem is made with the use of Open AI gym Taxi v3. The core gym interface is the “env” object, which is the unified environment interface. There are the following methods that can be used:

- observation_space.n: Returns the number of all possible states.
- action_space.n: Returns the number of all possible actions.
- reset(): Resets the environment and returns a random initial state.
- step(action): Step the environment by one timestep. Returns:
    - observation: Observations of the environment
    - reward: If your action was beneficial or not
    - done: Indicates if we have successfully picked up and dropped off a passenger, also called one episode
    - info: Additional info such as performance and latency for debugging purposes
- render(): Renders one frame of the environment

Reinforcement Learning Algorithms

Q Learning

The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.

For any finite Markov decision process (FMDP), Q-learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy.

The algorithm, has a function that calculates the quality of a state-action combination:
$Q : S \times A \rightarrow \mathbb{R}$

Before learning begins, Q is initialized to a possibly arbitrary fixed value. Then, at each time t the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1} (that may depend on both the previous state and the selected action), and Q is updated. The core of the algorithm is a simple value iteration update, using the weighted average of the old value and the new information:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{new value (temporal difference target)}} \overbrace{- \underbrace{Q(s_t, a_t)}_{\text{old value}}}^{\text{temporal difference}}$$

An episode of the algorithm ends when state s_{t+1} is a final state. However, Q-learning can also learn in non-episodic tasks. If the discount factor is lower than 1, the action values are finite even if the problem can contain infinite loops.

Hyperparameters:

Learning Rate

The learning rate $(0 < a \leq 1)$ determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities).

Discount factor

The discount factor $(0 < \gamma \leq 1)$ determines the importance of future rewards. A factor of 0 will make the agent "myopic" by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward.

Exploration Vs Exploitation

There's a tradeoff between exploration (choosing a random action) and exploitation (choosing actions based on already learned Q-values). We want to prevent the action from always taking the same route, and possibly overfitting, so we will use another parameter called ε to take care this issue during training.

SARSA Learning

SARSA algorithm is quite similar to Q learning. Specifically, a SARSA agent interacts with the environment and updates the policy based on actions taken, hence this is known as an on-policy learning algorithm. The Q value for a state-action is updated by an error, adjusted by the learning rate a. Q values represent the possible reward received in the next time step for taking action a_t in state s_t, plus the discounted future reward received from the next state-action observation.

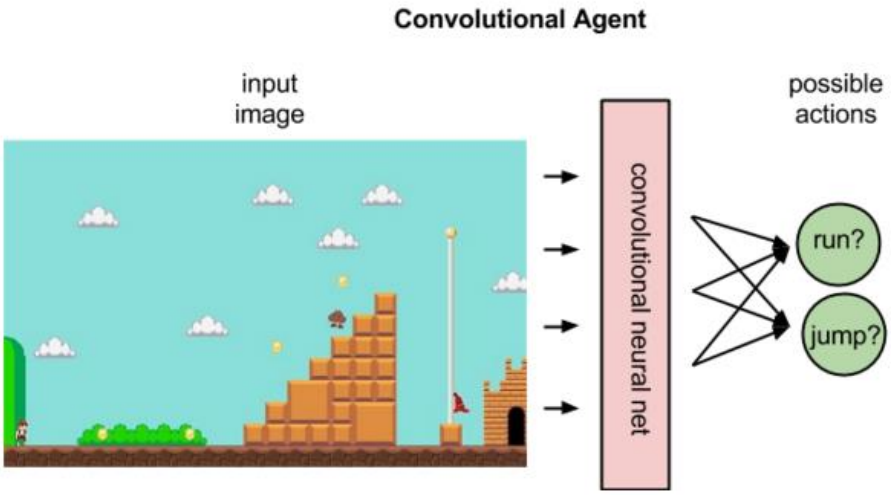
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Deep Q Learning

Neural networks are function approximators, which are particularly useful in reinforcement learning when the state space or action space are too large to be completely known.

A neural network can be used to approximate a value function, or a policy function. That is, neural networks can learn to map states to values, or state-action pairs to Q values. Rather than use a lookup table to store, index and update all possible states and their values, which impossible with very large problems, we can train a neural network on samples from the state or action space to learn to predict how valuable those are relative to our target in reinforcement learning.

Like all neural networks, they use coefficients to approximate the function relating inputs to outputs, and their learning consists to finding the right coefficients, or weights, by iteratively adjusting those weights along gradients that promise less error.



The above image illustrates what a policy agent does, mapping a state to the best action: $a = \pi(s)$. A policy maps a state to an action.

At the beginning of reinforcement learning, the neural network coefficients may be initialized stochastically, or randomly. Using feedback from the environment, the neural networks can use the difference between its expected reward and the ground-truth reward to adjust its weights and improve its interpretation of state-action pairs.

Results

Hyperparameters

In order to tune the hyperparameters and come up with the best set of values, a search function has been created. The function selects the parameters that result in best reward/timesteps ratio. The reason for the selection of reward/timesteps ratio is that we want to choose parameters which enable us to get the maximum reward as fast as possible. For each value, only five values in a specific space are tested, so as to save computational power. Specifically, for each hyperparameter the tested space is:

- $alpha \in [0.5\,, 1.0]$
- $gamma \in [0.9\,, 1.0]$
- $epsilon \in [0.4\,, 0.9]$

After 1000 episodes of training and 100 episodes of simulation, we reach to the following values for each algorithm:

- Q learning - ratio: 0.689
  - α: 0.75
  - γ: 0.9
  - ε: 0.9
- SARSA learning - with ratio: 0.233
  - α: 0.5
  - γ: 0.975
  - ε: 0.9

It is possible to reach better approximation of the hyperparameters by trying a bigger space of values.

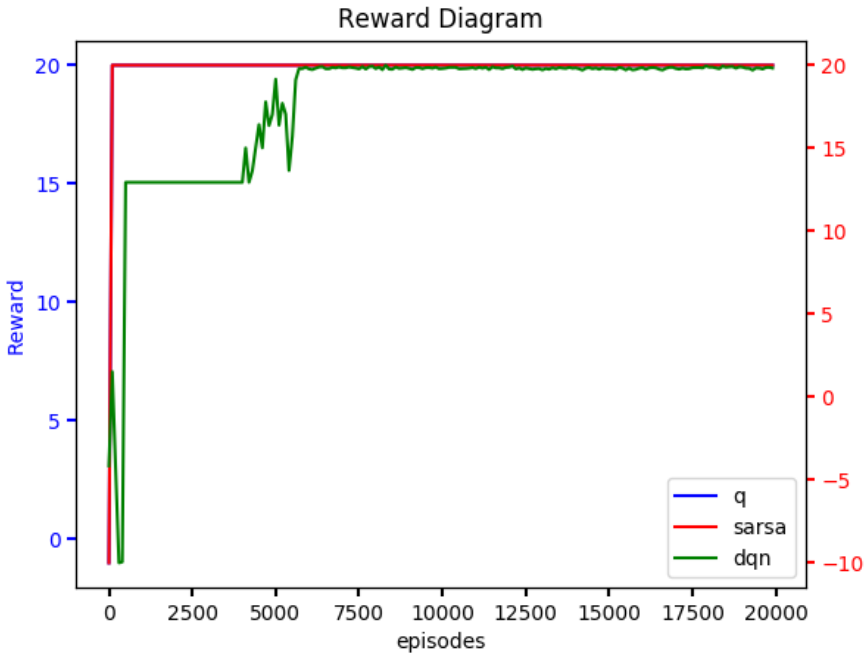
During the training process the aforementioned values for each hyperparameters are set. In addition, the values decrease over time by the factor of 0.0001 due to the that as the agent continues to learn, builds up more confident model of behavior.

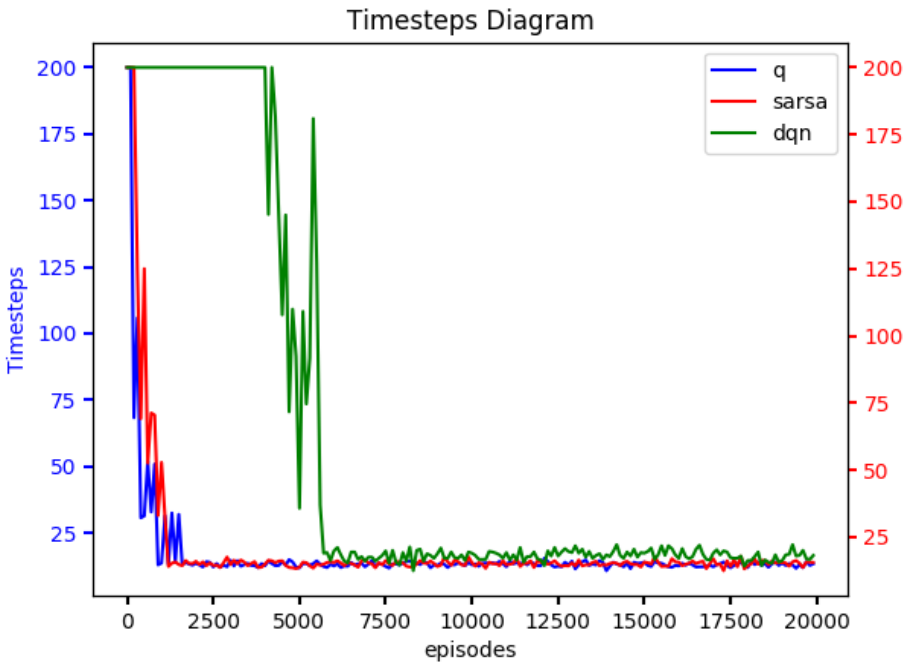
Algorithm comparison

Without learning we get the following results for 20000 episodes

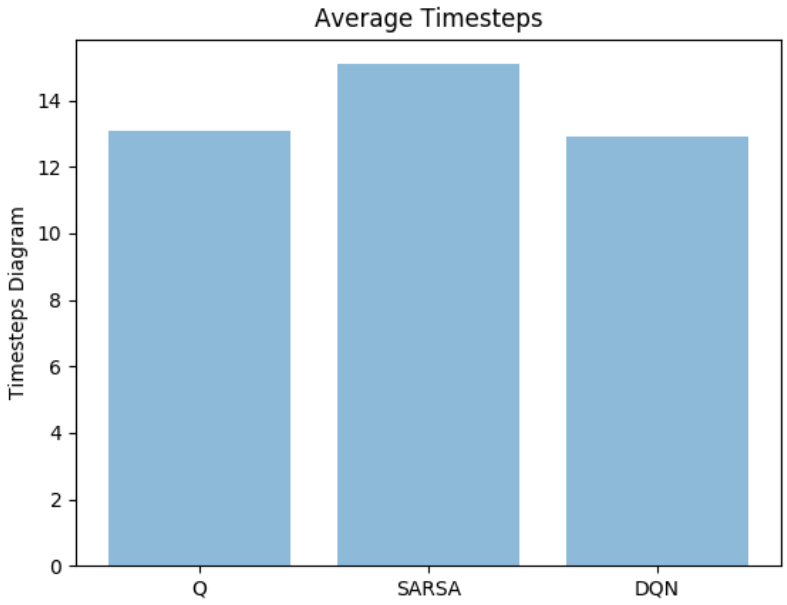
- Average timesteps per episode: 19605.31365
- Average reward per episode: -770.896

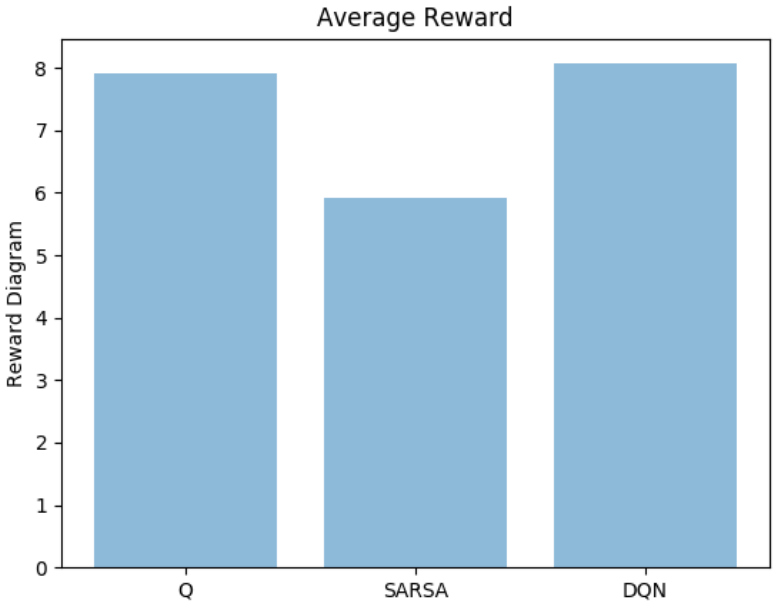
After the application of Q learning, SARSA learning and DQN we get the following results:





After 20.000 episodes of training we can conclude that each algorithm gives us quite good results. Each algorithm goals to minimize the total timesteps and the same time to maximize the total reward of the episode. Specifically, Q learning and SARSA performed similarly while Deep Q Network converges more slowly. The specific problem has 500 possible states which is feasible for the application of Q and SARSA. The number of states could be the reason that DQN did not outperform Q and SARSA.





In this case we compare the total behavior of the algorithms after the training process and take the average of their performance for 20.000 episodes. We can conclude that

Q and DQN perform quite similar while SARSA has lower performance in both timesteps and reward diagrams.

The overall conclusion is that each algorithm has quite good performance in this type of agent. The experiments show that Q learning has the best performance during the training process and at the final simulation.

It is worth mentioned that Deep SARSA algorithm was also tested but it did not perform in a desired way. To be more specific, the loss of the network remained constantly high and as result there wasn't any interesting outcome. The reason of this behavior could be the wrong configuration of the neural network.

References

1. Reinforcement Learning for Taxi-v2 - Anirban Sarkar
2. Reinforcement Q-Learning from Scratch in Python with OpenAI Gym
3. A Beginner's Guide to Deep Reinforcement Learning