Fundamentals of Machine Learning

Assignment 5

Joshua Koilpillai

Question

Re-create the provided example to implement an intelligent agent in a game using Q-Learning. You will choose one of the questions that are listed below to address conceptually in order to improve the model. Implement your idea into the provided code.

Turn this code into a module of functions that can use multiple environments
or
Tune alpha, gamma, and/or epsilon using a decay over episodes
or
Implement a grid search to discover the best hyperparameters

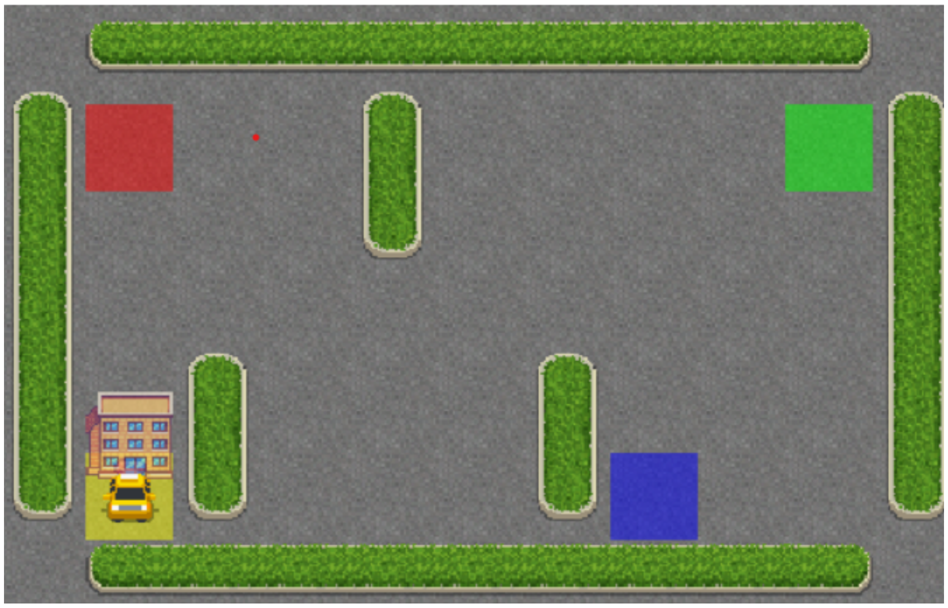
Code explanation and analysis for implementing taxi V3

- **Performing reinforcement learning**

Gym provides different game environments which we can plug into our code and test an agent. The library takes care of API for providing all the information that our agent would require, like possible actions, score, and current state

```
import gym

env = gym.make("Taxi-v3", render_mode="human")
env.reset()  # reset environment to a new, random state
env.render()

print("Action Space {}".format(env.action_space))
print("State Space {}".format(env.observation_space))

state = env.encode(3, 1, 2, 0)  # (taxi row, taxi column, passenger
index, destination index)
print("State:", state)
```

```
env.s = state
env.render()
env.s = 328  # set environment to illustration's state

epochs = 0
penalties, reward = 0, 0

frames = []  # for animation

done = False
```



The core gym interface is env, which is the unified environment interface. The following are the env methods that would be quite helpful to us:
env.reset: Resets the environment and returns a random initial state.
env.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
env.render: Renders one frame of the environment (helpful in visualizing the environment)

"There are 4 locations (labeled by different letters), and our job is to pick up the passenger at one location and drop him off at another. We receive +20 points for a successful drop-off and lose 1 point for every time-step it takes. There is also a 10 point penalty for illegal pick-up and drop-off actions."

```
while not done:
    action = env.action_space.sample()
    state, reward, done, info = env.step(action)
    if reward == -10:
        penalties += 1

    # Put each rendered frame into dict for animation
    frames.append({'frame': env.render(mode='ansi'), 'state': state,
'action': action, 'reward': reward})

    epochs += 1

print("Timesteps taken: {}".format(epochs))
print("Penalties incurred: {}".format(penalties))
from time import sleep


def print_frames(frames):
    for i, frame in enumerate(frames):
        clear_output(wait=True)
        print(frame['frame'].getvalue())
        print(f"Timestep: {i + 1}")
        print(f"State: {frame['state']}")
        print(f"Action: {frame['action']}")
        print(f"Reward: {frame['reward']}")
        sleep(.1)


print_frames(frames)

import numpy as np

q_table = np.zeros([env.observation_space.n, env.action_space.n])
```

Reinforcement Learning will learn a mapping of states to the optimal action to perform in that state by exploration, i.e. the agent explores the environment and takes actions based off rewards defined in the environment.

The optimal action for each state is the action that has the highest cumulative long-term reward.

- ## Using Q learning

```
"""Training the agent"""

import random
from IPython.display import clear_output

# Hyperparameters
alpha = 0.1
```

```
gamma = 0.6
epsilon = 0.1

# For plotting metrics
all_epochs = []
all_penalties = []

for i in range(1, 100001):
    state = env.reset()

    epochs, penalties, reward, = 0, 0, 0
    done = False

    while not done:
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample()  # Explore action
space
        else:
            action = np.argmax(q_table[state, :])  # Exploit learned
values

        next_state, reward, done, info = env.step(action)

        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state, :])

        new_value = (1 - alpha) * old_value + alpha * (reward + gamma
* next_max)
        q_table[state, action] = new_value

        if reward == -10:
            penalties += 1

        state = next_state
        epochs += 1

    if i % 100 == 0:
        clear_output(wait=True)
        print(f"Episode: {i}")

print("Training finished.\n")
```
With Q-learning agent commits errors initially during exploration but once it has explored enough (seen most of the states), it can act wisely maximizing the rewards making smart moves. Let's see how much better our Q-learning solution is when compared to the agent making just random moves.

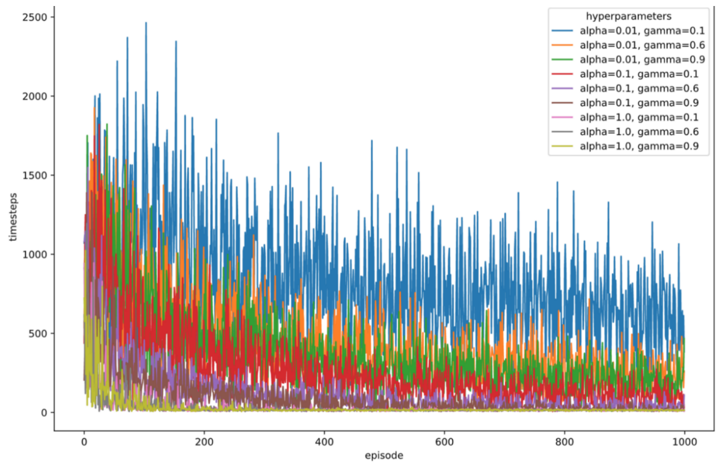
We evaluate our agents according to the following metrics,

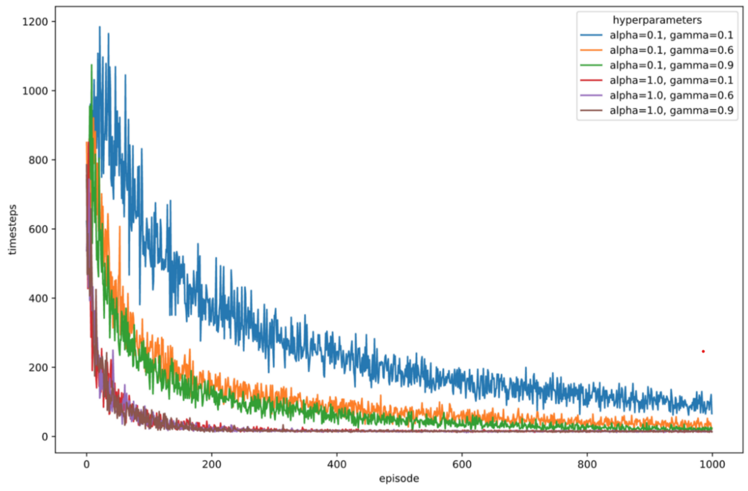
Average number of penalties per episode: The smaller the number, the better the performance of our agent. Ideally, we would like this metric to be zero or very close to zero.
Average number of timesteps per trip: We want a small number of timesteps per episode as well since we want our agent to take minimum steps(i.e. the shortest path) to reach the destination.
Average rewards per move: The larger the reward means the agent is doing the right thing. That's why deciding rewards is a crucial part of Reinforcement Learning. In our case, as both timesteps and penalties are negatively rewarded, a higher average reward would mean that the agent reaches the destination as fast as possible with the least penalties"

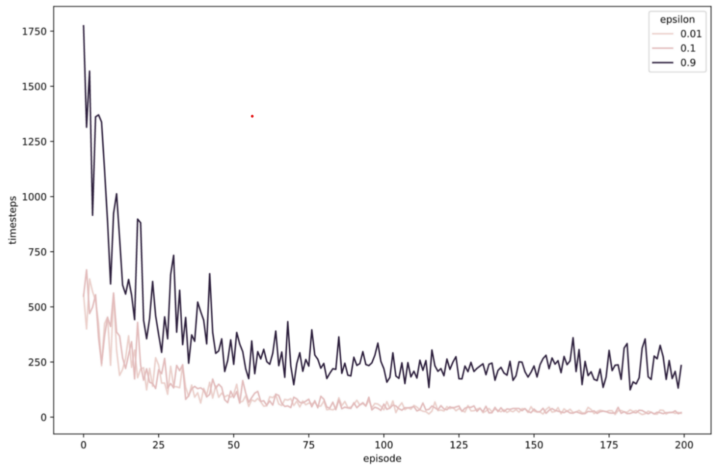
Tuning the hyperparameters
A simple way to programmatically come up with the best set of values of the hyperparameter is to create a comprehensive search function (similar to grid search) that selects the parameters that would result in best reward/time_steps ratio. The reason for reward/time_steps is that we want to choose parameters which enable us to get the maximum reward as fast as possible. We may want to track the number of penalties corresponding to the hyperparameter value combination as well because this can also be a deciding factor (we don't want our smart agent to violate rules at the cost of reaching faster). A more fancy way to get the right combination of hyperparameter values would be to use Genetic Algorithms.

- **Tune alpha, gamma, and/or epsilon using a decay over episodes**

```
import gym
env = gym.make("Taxi-v3").env

from src.q_agent import QAgent

alphas = [0.01, 0.1, 1]
gammas = [0.1, 0.6, 0.9]
import pandas as pd

from src.loops import train

epsilon = 0.1
n_episodes = 1000

results = pd.DataFrame()
for alpha in alphas:
    for gamma in gammas:

        print(f'alpha: {alpha}, gamma: {gamma}')
        agent = QAgent(env, alpha, gamma)

        _, timesteps, penalties = train(agent,
                                        env,
                                        n_episodes,
                                        epsilon)


        results_ = pd.DataFrame()
        results_['timesteps'] = timesteps
        results_['penalties'] = penalties
        results_['alpha'] = alpha
        results_['gamma'] = gamma
        results = pd.concat([results, results_])


results = results.reset_index().rename(
    columns={'index': 'episode'})

results['hyperparameters'] = [
    f'alpha={a}, gamma={g}'
    for (a, g) in zip(results['alpha'], results['gamma'])
]
import seaborn as sns
import matplotlib.pyplot as plt

fig = plt.gcf()
fig.set_size_inches(12, 8)
sns.lineplot('episode', 'timesteps',
             hue='hyperparameters', data=results)
from src.loops import train_many_runs

alphas = [0.1, 1]
gammas = [0.1, 0.6, 0.9]

epsilon = 0.1
n_episodes = 1000
n_runs = 10

results = pd.DataFrame()
for alpha in alphas:
    for gamma in gammas:

        print(f'alpha: {alpha}, gamma: {gamma}')
        agent = QAgent(env, alpha, gamma)

        timesteps, penalties = train_many_runs(agent,
                                               env,
                                               n_episodes,
                                               epsilon,
                                               n_runs)


        results_ = pd.DataFrame()
        results_['timesteps'] = timesteps
        results_['penalties'] = penalties
        results_['alpha'] = alpha
        results_['gamma'] = gamma
        results = pd.concat([results, results_])

results = results.reset_index().rename(
    columns={'index': 'episode'})

results['hyperparameters'] = [
    f'alpha={a}, gamma={g}'
    for (a, g) in zip(results['alpha'], results['gamma'])]
fig = plt.gcf()
fig.set_size_inches(12, 8)
sns.lineplot('episode', 'timesteps', hue='hyperparameters',
data=results)

alpha = 1.0
gamma = 0.9

epsilons = [0.01, 0.10, 0.9]
n_runs = 10
n_episodes = 200

results = pd.DataFrame()
for epsilon in epsilons:

    print(f'epsilon: {epsilon}')
    agent = QAgent(env, alpha, gamma)

    timesteps, penalties = train_many_runs(agent,
                                           env,
                                           n_episodes,
                                           epsilon,
                                           n_runs)



    results_ = pd.DataFrame()
    results_['timesteps'] = timesteps
    results_['penalties'] = penalties
    results_['epsilon'] = epsilon
    results = pd.concat([results, results_])


results = results.reset_index().rename(columns={'index': 'episode'})
fig = plt.gcf()
fig.set_size_inches(12, 8)
sns.lineplot('episode', 'timesteps', hue='epsilon', data=results)
plt.show()

fig = plt.gcf()
fig.set_size_inches(12, 8)
sns.lineplot('episode', 'penalties', hue='epsilon', data=results
```







The graphs help us understand the best hyperparameter values we ca use by having some fixed and varying alpha gamma and epsilon separately