**CPSC6114: Fundamentals of Machine Learning**

**Assignment 5: Reinforcement Learning – Epsilon Optimization Using Decay Approach**

Model

The goal of this assignment is to optimize one of the hyperparameters (alpha, gamma or epsilon) of the **reinforcement learning Q-learning algorithm** for the **Taxi project**. We will focus on **optimizing Epsilon hyperparameter using decay approach**.

**Reinforcement learning** is another machine learning technique that utilizes “learning” models. In the model, an “agent” (e.g. taxi cab) interacts with the “environment”(e.g. streets) via certain “actions” (go east, go north, etc) to achieve a certain “goal” (e.g. pick a passenger, drive to the destination in the least amount of time, and drop the passenger off at the right location). Each “action” gets the agent into a “state” and earns “rewards” or “penalties” as feedback from the environment for its actions. The ideas is that “agent” maximizes its “rewards” (both local rewards for each action and total reward to achieve the goal) by performing the correct actions to achieve its “goal”. The agent uses a “policy” or strategy to achieve the desired goal (i.e. what “action” should be performed when in a given “state”)

**Q-Learning algorithm** is one of the most popular reinforcement learning algorithms. Q-learning is **model-free** (does not require model of the environment), **off-policy** (maximizes rewards regardless of policy) value-based learning algorithm that finds the best course of action given the current state of the agent. To do so, the algorithms starts with a trial and error approach and interactively learns the optimal Q-value function using Bellman Optimality Equation. The algorithm stores the Q-values in a table (Q-table) and updates them after each step. The function used is:

*Diagram, text

Description automatically generated*

Where **α is the learning rate and γ is the value of future reward**

Hyperparameters

Q-learning algorithms uses three main hyperparameters: **alpha, gamma and epsilon**

*Alpha*

Alpha is learning rate of the model and defines the magnitude of steps that is taken towards the solution. The higher alpha value means Q values are updates in big steps. It is recommended that initially alpha uses a higher value is because Q-learning starts at a random point and the algorithm should take big jumps towards the solution. As it closer, it is advisable to minimize – or decay – alpha to get more and more precise in optimizing the function.

*Gamma*

Gamma is the value of future reward. Lower gamma allows the model to learn faster by obtaining higher immediate rewards. However, this can cause our algorithm to get “stuck” in “local maximas” of our space. Higher gamma allows the algorithm to focus more on the “grand prize”, so to speak. But the learning process can be significantly slower. As the algorithm gets closer to the end of the process, the gamma values should decrease

(<https://towardsdatascience.com/practical-reinforcement-learning-02-getting-started-with-q-learning-582f63e4acd9>)

*Epsilon*

**Epsilon** is a hyperpartemeter used to help algorithm to **decide whether to utilize its knowledge** that it already gained when deciding on the next action, **or to do so at random and continue to explore** the environment in hope to find better solution than what it already found/learned. Epsilon helps us manage the **“Exploration vs exploitation” dilemma.**

**Exploration** or taking random action, allows an agent to improve its current knowledge about each action, hopefully leading to long-term benefit. Exploration is more important in the beginning of the process, when the agent does not have enough information about the environment. Improving the accuracy of the estimated action-values, enables an agent to make more informed decisions in the future.

**Exploitation** on the other hand, chooses the **greedy action** to get the most reward by exploiting the agent’s current action-value estimates. Exploitation makes a lot of sense later in the process when “enough” information about the environment was collected and the agent needs to behave in an optimal way. However, until then there is a major drawback of being greedy: The agent acts only on the basis of its knowledge of the environment, which may be incomplete. Being greedy with respect to action-value estimates may not get the most reward and lead to less optimal strategy.

When an agent explores, it gets more accurate estimates of action-values. And when it exploits, it might get more reward. It cannot, however, choose to do both simultaneously, which is also called the exploration-exploitation dilemma.

(<https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/>)

**Without** the **Epsilon hyperparameter** our algorithm is at risk of taking the same sub-optimal approach and possibly **overfitting.**

Epsilon Decay

Epsilon’s value is between 0 and 1. The lower then value of epsilon say (e = 0) then more the algorithm uses its current knowledge to choose the best action for the next decision. Higher values of e (e=.9) make the algorithm to explore a lot more than using Q values it has already stored. In order to optimize “exploration-exploitation” dilemma, we can use a technique called Epsilon Decay.

**Epsilon greedy**

Epsilon greedy approach uses a random function to determine whether the next action should be random/exploratory or knowledge based. In our example, the Self-Driving Cab is using epsilon greedy approach:

if random.uniform(0, 1) < epsilon:

action = env.action\_space.sample() # Explore action space

else:

action = np.argmax(q\_table[state]) # Exploit learned values

**Continuous decay and Epsilon greedy**

Continuous decay is another approach where epsilon value is diminished after each iteration by some constant. In some cases, it is preferrable to start with high values of epsilon, say e = 0.9. We can then add:

epsilon = epsilon \*0.9

at the end of the loop of our code. Epsilon value will gradually diminish as the model learns its environment and develops optimal strategy. In higher steps, the exploration can be less important than during the initial stages. This can speed up the learning process while maintaining high quality. We can **still use the greedy approach**, but the value of **epsilon will not be constant** but gradually reduced.

**Rewards or Knowledge based**

Another approach is to use rewards or knowledge that the algorithm accumulated as a trigger for epsilon decay. Only when an agent has cross some rewards value, the value of epsilon starts to decrease by some amount. For example:

if np.max(Q\_table)>1

epsilon = epsilon \*.9

Epsilon decay optimization

For this analysis, we will keep alpha and gamma constant at alpha = 0.1 and gamma = 0.6, as given by the example.

**Before we analyze various values** of epsilon, lets determine if having e = 0 produces optimal results. Running the model with no epsilon **does not** produce the best outcome with average timesteps per episode of 13.35 which is significantly higher than the base case of

A screenshot of a computer

Description automatically generated with medium confidence

**Greedy Epsilon**

We can start with **greedy approach**. This approach is already implemented in our example. Starting with **e = 0.9** (very high e value that relies heavily on exploration and a lot less on exploitation). The model training time is significantly slower at **10 mins and 22 secs**. The performance is also lower with **13.23 timesteps** per episode.

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

We can manually decrease value of epsilon to **e = 0.5**. Model’s learning time improves substantially (since it has much higher probability of using exploitation vs exploration now). It finished in 1 min and 27 secs. The performance also improved to **13.08 timesteps** per episode.

Text

Description automatically generated

Text

Description automatically generated

We can now try **e = 0.1**, which was the initial value proposed by this example. Learning time is the fastest of all three tries at 50 seconds, since the algorithm does exploration only occasionally now. We also achieve the best performance at **12.8 timesteps** per episode

Text

Description automatically generated



It seems that **e = 0.1** is optimal level for given algorithm settings and using **greedy decay approach**

**Continuous Decay**

We can now add **continuous decay** to our greedy epsilon approach, lowering epsilon by a constant factor after each iteration. Let’s add ***epsilon = epsilon \* 0.9***  at the end of the loop:

*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*

***epsilon = epsilon \* 0.9***

We can begin by trying to use this approach with epsilon = 0.9, 0.5 and 0.1. While keeping the decay factor at 0.9. After three tries it appears that **Epsilon = 0.5, decay factor = 0.9**

Text

Description automatically generated

Text

Description automatically generated with medium confidence

**Rewards or Knowledge based**

We now can experiment with epsilon decay based on a the values of Q-table. As the algorithm learns the environment, the values tend to converge and increase. We can use a minimum Q-value threshold to minimize the value of epsilon so that the algorithm relies mostly on exploitation. We will look for **maximum value of Q to be > then zero to make e = 0:**

*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*

***maxQ = np.max(q\_table)***

***if maxQ >0:***

***epsilon = 0***

We can see that this approach produces best results of all three: although training takes slightly longer, we achieved **best average timesteps per episode at 12.64** while incurring no penalties.

Text

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

Conclusion

The choice of epsilon decay technique heavily depended on the nuances of the problem that Q-algorithm is trying to solve. All three approaches – greedy decay, continuous decay, or reward based decay – have their usefulness and merit. In case of Self Driving Taxicab problem, rewards based decay seem to produce better results than the other two approaches.

Python Code (Model Training Only)

*%%time*

*"""Training the agent"""*

*"""REWARDS BASED DECAY"""*

*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*

*maxQ = np.max(q\_table)*

*if maxQ >0:*

*epsilon = 0*

*if i % 100 == 0:*

*clear\_output(wait=True)*

*print(f"Episode: {i}")*

*print("Training finished.\n")*