Q-Learning

Inventory Control with Poisson Distribution

Introduction

Imagine you own a store selling products. In order to sell the products your store sells, you need to hold onto inventory. Because your store is only so big, there is a limit to how much inventory you can hold. Furthermore, purchasing inventory costs you money. Your goal in this experiment, is to maximize your profit by selling the most amount of stock, while minimizing inventory.

Building the environment

To simulate this environment, we imagine customers arriving at our store, randomly sampled from a poisson distribution. When these customers arrive, they will always buy inventory from your store (given that you have inventory to sell). After customers are done shopping, you are given the opportunity to restock a certain amount before the next wave of customers enters your store.

We define a random variable X such that:

$$X = \text{The number of customers at a given step}$$

$$P(X = k) = \frac{e^{-\lambda} * \lambda^k}{k!}$$

To define a poisson distribution in python, we make parameters λ and n. λ controls how our distribution looks, while n controls the maximum amount of customers that can arrive at a certain step (Note that, if n is too small for your λ, an error will occur).

```
import numpy as np
import math
```

```
class Environment:
    def __init__(self, lam, n):
        self.max_inventory = n
        self.inventory = 3
        self.actions = self.customers = [x for x in range(n)]
        self.lam = lam
        self.distribution = [(1/(math.e ** self.lam)) *
                             (1/(math.factorial(k))) *
                             (self.lam ** k) for k in self.customers]
```

Here we compute P(X = k) from 0 <= k <= n - 1 and store each probability in our list called self.distribution

```
    def get_customer(self):
        return np.random.choice(self.customers, p = self.distribution,
         ↪  size=1)[0]
```

This function returns the amount of customers for any given episode. Here we randomly choose an integer from our self.customers list which contains integers ranging from 0 to n - 1. Self.distribution contains a list of probabilities, where the index of each probability represents the amount of customers.

Reward

The environment receives actions and outputs a reward from those actions. Technically, this reward is a feature of the observer, as the observer observes the outcome of the action the model takes, and decides whether the action had a positive or negative outcome (the observer is discussed in more detail later). Thus, the reward is defined in the environment as:

```
    def action_outcome(self, action):
        num_customers = self.get_customer()
        reward = self.buy_inventory(action) + self.order(num_customers)
        return (reward, num_customers)
```

Here the reward is defined as:

$$reward = \text{amount of stock sold} - \text{amount of inventory bought}$$

When calculating our reward, we need to update our inventory amount, so we pass our action to a helper function that processes our action and updates our inventory amount accordingly

```
    def buy_inventory(self, amount):
        self.inventory =  self.max_inventory if self.inventory + amount
         ↪  > self.max_inventory else self.inventory + amount
        return -amount
```

Finally, we need to process the number of customers at this episode and simulate our stock being sold. We use a function: order(int num_customers), which takes an integer representing the amount of customers coming into our store, updates our inventory, and returns the amount of money we receive from these customers.

```
    def order(self, num_customers):
        orders_sold = self.inventory if self.inventory < num_customers
↪   else num_customers
        orders_missed = num_customers - orders_sold
        self.inventory = 0 if self.inventory < num_customers else
         ↪  self.inventory - orders_sold
        return 2 * (orders_sold  - orders_missed)
```

If the amount of customers exceeds our current inventory, the observer punishes the model for every sale lost, where:

number of customers <= orders_sold - orders_missed <= number of customers

Model

Now we can construct the model. This is the agent that takes a state from the environment as input, and outputs an action depending on a policy.

To update a policy, first we define a lookup-table. This table is responsible for holding an estimated reward prediction for every action/state pair. We will define this lookup table as self.state_actions.

Next, to choose an action depending on the values in our lookup table, we also need to define a probability distribution for each state. In each state, every action will have a certain probability of occurring, such that:

$$\sum_{i=1}^{n} p(a_i|s) = 1$$

We use a lookup table for this probability distribution such that, for every state, there will contain a list of probabilities where the index of the probability represents the action the model can take. We name this lookup table self.state_action_distribution.

Finally, we define some hyperparameters. γ is responsible for controlling the weight in which future rewards have an effect on current rewards. ϵ is responsible for the speed in which the state/action probability distribution is updated. Furthermore, α is responsible for how quickly our state_action lookup table is updated.

```
class Model:
    def __init__(self, actions, gamma, alpha, epsilon):
        self.actions = actions
        self.state_actions = np.zeros((len(actions), len(actions)))
        self.state_action_distribution = np.zeros((len(actions),
         ↪  len(actions)))
        self.state_action_distribution.fill(1/len(self.actions))
        self.current_SA = None
        self.alpha = alpha
        self.action_log = []
        self.gamma = gamma
        self.epsilon = epsilon
```

The model has now one function, to take a state from the environment, and output an action. To do this it needs two tools: the first being a function that updates the self.state_actions, and another function to update self.state_action_distribution.

First, we define a function that our observer calls to get the next action given the current state. Here the agent, takes the state from the observer, and returns an action.

```
    def get_action(self, state):
        current_action = self.choose_action(state.inventory)
        self.action_log.append(current_action)
        self.current_SA = (state.inventory, current_action)
        return current_action
```

self.current_SA stores the current state and action for further use. The choose_action function, chooses an action using our probability distribution lookup table.

```
    def choose_action(self, state):
        return np.random.choice(self.actions, size=1,
         ↪  p=self.state_action_distribution[state])[0]
```

After the observer receives the next action, it then passes the outcome of the action as a reward to the model. The model uses this reward to update both lookup tables. First, the model updates the state, action value table.

```
    def policy_update(self, reward, next_state):
        future_state = next_state.inventory

        optimal_future_action = self.get_max_value(future_state)

        self.state_actions[self.current_SA] += (self.alpha *
                                                (reward + (self.gamma *

                                                 ↪  self.state_actions[future_state][

 ↪  optimal_future_action]) -

                                                 ↪  self.state_actions[self.current_SA]))
        self.update_distribution(self.current_SA[0])
```

Here, the current state action pair in our lookup table, represents a reward prediction for our current state action. We take this prediction, and update its accuracy by finding the difference between the prediction and the actual reward. Added to this, is the predicted reward for the next best action using the current policy. This lets the model use future outcomes to determine the actual reward of an action at a given state.

Next the model updates the probability distribution lookup table.

```
    def update_distribution(self, state):
        best_action = self.get_max_value(state)
        for x in range(len(self.state_action_distribution[state])):
            if x == best_action:
                self.state_action_distribution[state][x] += (1 -
                 ↪  self.state_action_distribution[state][x]) *
                 ↪  self.epsilon
            else:
                self.state_action_distribution[state][x] -=
                 ↪  self.state_action_distribution[state][x] *
                 ↪  self.epsilon
```

The probability distribution is updated using our hyperparameter epsilon. In this function, we find our optimal action given our current state, and we increase the probability of that action, while decreasing the probability of all other actions in the given state.

$$a_k|s = \text{best action given an arbitrary state s} : 1 <= k <= n$$

$$p(a_k|s) += p(a'_k|s) * \epsilon$$

$$a_i|s = \text{any action in state s that is not the best action}$$

$$p(a_i|s) -= p(a_i|s) * \epsilon$$

The smaller epsilon is, the more we let the model explore different actions for every state.

get_max_value(int state) is a helper function that returns the action with the highest predicted reward given a state.

```
    def get_max_value(self, state):
        return np.unravel_index(np.argmax(self.state_actions[state]),
         ↪  self.state_actions.shape)[1]
```

Observer

The observer is responsible for passing information between the model and the environment. It is the observer that creates the environment and the model, takes the state of the environment and gives it to the model, and then passes the action chosen by the model, into the environment and extracts the next state to repeat the process again.

```
env1 = Environment(20, 50)
model = Model(env1.actions, .9, 1, .03)

trial = 200000
customer_l = []

for i in range(trial):
    next_action = model.get_action(env1)
    reward , t= env1.action_outcome(next_action)
    model.policy_update(reward, env1)
    customer_l.append(t)
```

```
AttributeError: 'Model' object has no attribute 'get_action'
```

The process detailed above occurs in the for loop. Until the experiment ends the observer receives states and actions, passing them to the environment and the model accordingly. We create a list that stores how many customers came at each step. We will use this list to track how well our model performed.

Performance

Now we would like to test how well our model did in choosing the best action. To do this, we use numpy to find the average action of the last 30 actions. We then compare this number to the expected amount of customers each step during the last 30 steps of the experiment. Looking at these averages towards the end of the experiment, allows me to see if the model was able to train to an optimal strategy. An optimal strategy being if the model was able to restock at the same rate customers arrived in the store.

```
print(customer_l[len(model.action_log) - 30:])
print(model.action_log[len(model.action_log) - 30:])
average_customer = np.average(customer_l[len(model.action_log) - 30:])
average_action = np.average(model.action_log[len(model.action_log) -
 ↪  30:])
print(f"Average customer per step: {average_customer} with lambda:
 ↪  {env1.lam}")
print(f"Average inventory restock per step: {average_action}")
```

```
[]
[]
Average customer per step: nan with lambda: 20
Average inventory restock per step: nan
```

```
/home/jet08013/anaconda3/envs/torch/lib/python3.11/site-packages/numpy/lib/function_base.py:5
  avg = a.mean(axis, **keepdims_kw)
/home/jet08013/anaconda3/envs/torch/lib/python3.11/site-packages/numpy/core/_methods.py:192:
  ret = ret.dtype.type(ret / rcount)
```