Brandon Quinones

CS-370-12785-M01

[brandon.quinones@snhu.edu](mailto:brandon.quinones@snhu.edu)

**Design Defense – Module 7 Project Two**

As part of this the development of the Treasure Hunt Game, I was given the task of designing the episode loop for the Pirate Intelligent Agent. The loop I created will be provided below along with a detailed explanation for the reasoning behind each step.

for epoch in range(n\_epoch):

agent\_cell = random.choice(qmaze.free\_cells) # randomly chose a free cell

qmaze.reset(agent\_cell) # reset the maze env

envstate = qmaze.observe() # get state of env after reset

n\_episodes = 0 # set episodes to zero

while True: # start an episode loop

previous\_envstate = envstate # store the env state from the prev step

if np.random.rand() < epsilon: # decide exploration or exploitation

action = np.random.choice(qmaze.valid\_actions()) # exploration

else: # exploitation

q\_values = model.predict(np.array([envstate]).reshape(1, -1),verbose=0)[0]

action = np.argmax(q\_values)

envstate, reward, game\_status = qmaze.act(action) # perform action, get new state, reward, and game status

episode = [previous\_envstate, action, reward, envstate, game\_status] # create episode tuple

experience.remember(episode) # store episode to experince memory

n\_episodes += 1 # increase episode counter

if game\_status in ('win', 'lose'): # check game status

win\_history.append(1 if game\_status == 'win' else 0) # record win or lose

break # break if game is over

inputs, targets = experience.get\_data(data\_size) # sample batch of experiences for training

loss = model.train\_on\_batch(inputs, targets) # train the model on sampled batch

win\_rate = sum(win\_history[-hsize:]) / hsize if len(win\_history) >= hsize else sum(win\_history) / len(win\_history)

Firstly, the episode loop lies within the qtrain function which is the core of this q-learning agent’s training process. It is the interaction between the agent, the environment, and the experience replay mechanism. The training loop I created is epoch-based, meaning that it allows for iterative refinement of the Q-network. Each training cycle is an epoch, and within this loop we cover exploration, exploitation, and model updates.

In order to achieve randomness, which is important for machine learning because we do not want the model to rely on what it knows but also be able to explore when it needs to, we used the line **agent\_cell = random.choice(qmaze.free\_cells)** to accomplish this. After that, **qmaze.reset(agent\_cell)** then resets the environment to this state.

Once the state is set, we begin our episodes, we first set the episode count to zero and then begin our episode loop. The **while True:** loop will simulate an episode. **qmaze.act(action)** applies the chosen action (explore or exploit) and **game\_status** determines to end the game when it is over depending on if the status is won or lost.

The determination of the loop going the explore route or exploit route is based on our epsilon-greedy policy. The idea is to balance exploration and exploitation. When the agent chooses a random action, it is exploring. And when it chooses the action with the highest Q-value it is exploiting. The implementation is done based on, **if np.random.rand() < epsilon:** , which is a conditional statement. **np.random.choice(qmaze.valid\_actions())** will handle exploration. **np.argmax(q\_values)** will handle exploitation. In this line you will also see **verbose=0** which was an attempt to disable the logging of each episode, but it did not work for me.

The experience replay is the next step after this, this is there to enhance learning stability and efficiency by storing past experiences and sampling them for training. This decorrelates training samples and reduces variance. It is implemented with **experience.remember(episode)** which stores each episode in the experience replay buffer. **experience.get\_data(data\_size)** samples a batch of experiences for training.

The model training is then done using the sampled experiences. I could have used **model.fit** but decided to use **model.train\_on\_batch** for efficiency and per-batch updates. **model.train\_on\_batch(inputs, targets)** is the line of code that performs the model updates.

The win rate and completion check happen after this and is there to provide a metric for training progress. The win rate tracks the agent’s success, and the completion check verifies that the agent can solve the maze from all starting positions. The **win\_rate** is calculated based on the **win\_history**. And **completion\_check(model, qmaze)** verifies the agent's ability to solve the maze. Later in the script there is an epsilon decay, **if win\_rate > 0.9 : epsilon = 0.05**, which uses the win rate to reduce the amount of random explorations as the model improves. In the end of each epoch, the epoch summary is given and shows the training progress by proving information such as the loss, win rate, win count, and time elapsed. The win history is also used for early stoppage in case that there is a 100% win rate and can solve the maze from all the starting positions, **if sum(win\_history[-hsize:]) == hsize and completion\_check(model, qmaze).**