Q-learning Approach to Play Pong

By: Michael Campiglia & Anton Paljusevic

Introduction

Reinforcement learning is an increasingly popular branch of machine learning that involves training an agent that can perceive information of a specified environment and then act on that environment [1]. The agent learns over time by receiving a reward, which could be a positive score, or a penalty, which could be a negative score, that in turn serves as feedback for the previous action taken. Similar to how linear regression and other supervised learning algorithms focus on minimizing the cost function during training, the general goal of all reinforcement learning agents is to maximize the overall reward to find the optimal policy, or behavior pattern, to solve a particular task.

Q-learning is a reinforcement learning algorithm that seeks to learn a policy that maximizes the total reward, which means determining the best action to take given a current specified state [2]. Q-learning is generally considered to be an off-policy reinforcement learning technique because the function learns from actions that are outside the current policy, like taking completely random actions, and therefore a strict predetermined set of rules or behaviors isn't required during training.

The goal of our term project is to create a reinforcement learning algorithm that can train an agent to accomplish a goal or improve in a certain skill or task. More specifically, we plan to use Q-learning along with OpenAI's reinforcement learning environment platform for python, *gym* [3]*,* in order to create an AI agent that can potentially perform better than an average human in the Atari video game, Pong.

Data Exploration

To approach most reinforcement learning problems, an environment in which learning can be measured and take place has to be defined or created. The environment we applied our Q-learning algorithm on is the classic Atari game of Pong [4], which happens to be one of the pre-built environments in OpenAI's Gym python package for reinforcement learning. Every Atari game offered by Gym has the same set of 18 actions, called an action space, consisting of every remote button/joystick combination on the original console, such as *No Action, Up, Down, Fire, Fire+Up, Left+Up*. However, Pong only has three applicable actions, *Move Up, Move Down,* and *No Action*, so we redefined our action space to consist of these three maneuvers, mapping them to the respective numeric values of 2, 5, and 0.

Another key component to any reinforcement learning environment is the observation space, which defines all possible states of an environment and provides the details that the agent uses to take in information and learn, comparable to a training dataset for supervised learning techniques. The observation space provided by Gym consists of 210x160x3 RGB images of the game screen, as shown in the figure below.

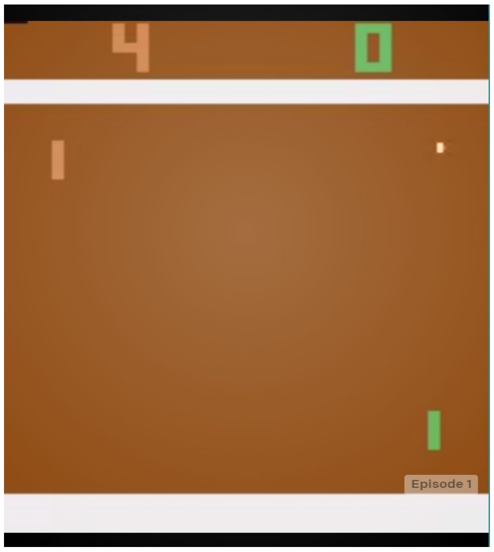

Figure 1: Gym Pong Observation

Model Training

Training a Q-learning model begins with the agent taking a "step", or taking a specific action in a particular state within the observation space, and then updating the associated Q-value for that step in a Q-table. A Q-table is a matrix that contains all the possible combinations of states and actions along with their Q-value, which is updated based on the reward given to the agent for performing a specific action at a specific state. More specifically, the following equation is used to update a Q-value

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma \max Q'(s',a') - Q(s,a)]$$

where $\boldsymbol{\alpha}$ is the learning rate, which defines how much value is given to the new Q-value compared to the current one, $\boldsymbol{\gamma}$ is the discount factor, which usually ranges between 0.8 and 1.0 and determines how much future rewards matter compared to current reward, and $\boldsymbol{R}$ is the reward value for completing a certain action at a specific state.

An agent chooses its action by taking part in either exploration, where the agent acts completely randomly based on the available actions, or exploitation, where the agent references the Q-table and chooses the action with the highest Q-value associated to the current state. In most scenarios, an agent would begin training with a relatively high exploration rate, usually denoted by epsilon, and then decay it to zero or a very small percentage over time, which helps to prevent the model from converging at local minima too early in the training phase.

To summarize the training process, an agent will start at a state in the observation space, choose an exploratory or exploitative action, arrive at a new state with a reward value, and then calculate and fill in the new Q-value for that action-state pair. This marks the completion of a single "step", and once enough steps are made for a player to score 21 points and finish a game of Pong, a single episode has completed. Most of our models trained for around 20,000 episodes, with computation times of up to 12-16 hours.

Data Preprocessing

The observation space and action space along with the possible outputs/inputs are directly related to the size of the q-table required for the q-learning algorithm. Each possible state holds a q-value for each possible action; so the larger the q-table the more complicated the learning process will become for the agent. The observation space returned at each step in Gym's pong environment was an RGB image array with size 210 X 160 X 3 (with 256 possible values each) and the action space was 1 choice each step with 18 available remote button combinations. Reducing this data allows us to shrink the size of the q-table in two ways, by removing unnecessary information and generalizing the observations.

The first step in removing unnecessary information was done by reducing the action space to 3 possible actions (up, down, no movement). The Atari emulation in Gym allows all the original action of the Atari remote, joystick movement and the fire button action. In Pong, a majority of the remote's actions are disabled, the only possible actions are moving the paddle up and down. Theoretically the agent should learn that all other button combinations result in no movement but in order to reduce the q-table size we limited the action space to three options.

The second method used to remove unnecessary information and generalize the observation was done by pre-processing the image array. The top and bottom of the game display a border and score points; since the score is tracked through the reward output, this part of the image could be cropped out since it holds extra information pertaining to the game. Next, the color information of the game was reduced to only one of the three color arrays. The color scheme of the game is simple and each component has a different hue, so storing all three parts of the array is repetitive and unnecessary information. Finally, generalizing the information was done by taking the reduced color array and indexing the location of each of the three major components: the opponent's paddle, the ball, and the agent's paddle. Doing this allows the agent to ignore the background, which shouldn't affect the decision making process of the agent.

After the observation was preprocessed, the result was then discretized; and extra step to further reduce the q-table size. Discretizing each read index reading limited the possible outcomes to 5 each. Normally, discretizing is used to make a continuous reading finite, but in this case it was used to generalize/approximate the location of each of the components. By preprocessing and discretizing our data, we were able to reduce the number of q-table states from 25.8 million to 125. All preprocessing and discretization was repeated after every observation through the functions prepro and get_discretize_state prior to inputting into the q-learning algorithm.

Model Testing and Validation

Before we started training our Q-learning agent for Pong, we wanted to first test the validity of our model and process on a much simpler example called "CartPole" that was provided by Gym. This environment had the goal of balancing a stick on top of a cart using a very small observation space of the velocity, angle, and location of the stick and cart. Points were awarded to the agent for every step taken without dropping the stick, and after around 250 episodes of Q-learning training, the agent was able to solve the scenario and stay alive for over 500 timesteps each episode. This proof of concept gave validation to our algorithm and also gave us a lot of insight towards the importance of parameter tuning for our model. A few important aspects that we learned helped improve training time and efficiency included implementing a decaying learning rate along with epsilon, as well as changing our discount factor to a value of 1.0 to stress maximum importance on future reward.

After we finished training and testing for CartPole, we then began the same process for the Pong environment. We quickly found that Pong was much more complicated and even with parameter tuning, it took thousands of episodes to show progress towards the agent improving in this environment. We ran various models for periods between 12,000 - 30,000 episodes, and tracked the average, minimum, and maximum reward per every 100 episodes and then plotted these results for visualization and validation. The figures below show performance graphs for two of our best agents. Both agents had a decaying learning rate starting at 1.0 and eventually going to 0.01, an exploration rate starting at 0.5 and ending at 0.01 after half the total episodes trained, and a discount factor of $\gamma = 0.995$. We found a significant improvement in our second model after actually implementing a second wave of exploration after a long period of training.

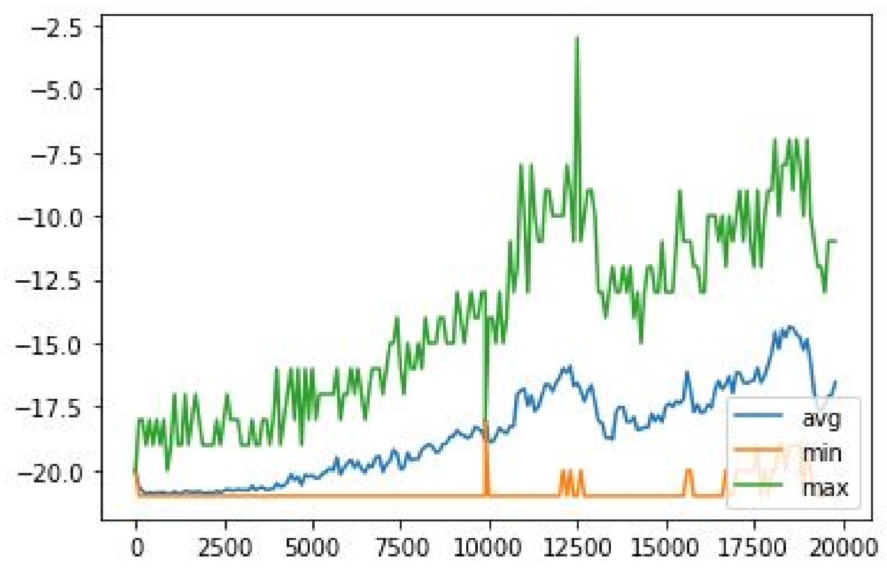

Figure 2 - Performance for agent with observation space of 20x20x20

We noticed that many of our models would reach a plateau point for the average reward. We decided to reset the exploration to half of its original value and resume training on a model that used a Q-table derived from 17k episodes of training. With this readdition of epsilon, the model was able to learn new strategies, and within around 2.5k episodes, it was able to actually solve the environment and win against the other computer pretty significantly.

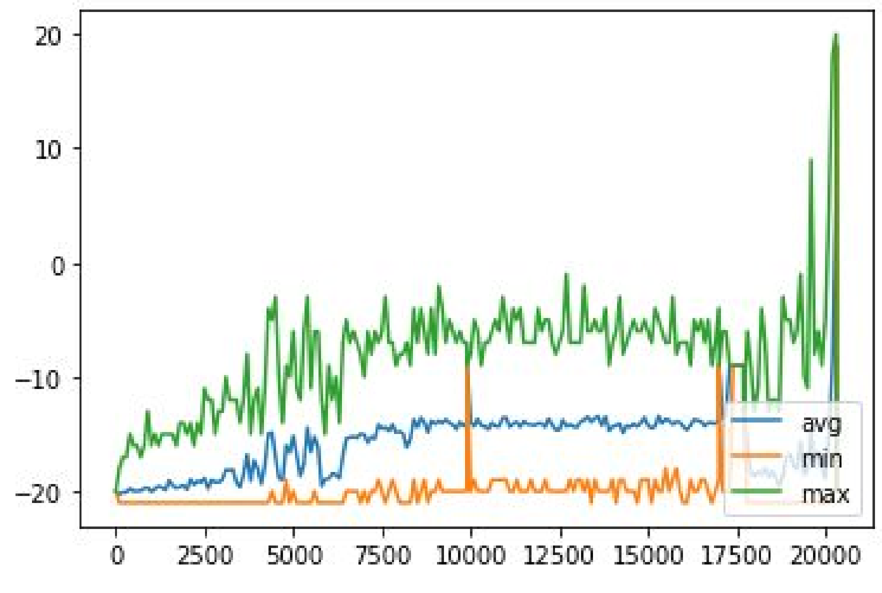

Figure 3 - Performance for agent with observation space of 5x5x5 w/ re-addition of Epsilon

Summary

Creating and running a q-learning algorithm script on the CartPole environment in Gym proved that the reinforcement algorithm could be used to teach an agent how to play a game. After tuning the learning rate and exploration rate parameters, the agent was able to win the game after less than 300 training runs. However, editing the script to train an agent to play Pong showed that the q-learning algorithm alone may not be the best option for teaching an agent. The agents trained to play Pong showed significant improvement when compared to an untrained agent but required a significant amount of training time. After over 12 hours of training on one agent, the parameters were finally tuned to consistently win against the opponent by exploiting a resetting feature after each point is scored.

Although we successfully taught an agent to win a handful of games, the method of q-learning alone is not the optimal method to teach an agent with such a large observation space. Also, with parameter tuning playing an important part in having an agent learn and with training taking thousands of episodes, resetting exploration may not be the best option. With major pre-processing and intense parameter tuning required to learn, another method such as deep q-learning might be a better option. In deep q-learning, a neural network is added on to the model and simultaneously trained prior to each episode. The neural network allows the agent to self-generalize/auto-cluster the observation space; taking in all observation space information and outputting its own clustering assumptions.

References

[1] Dutta, Sayon. *Reinforcement Learning with TensorFlow: A beginner's guide to designing self-learning systems with TensorFlow and OpenAI Gym*. Packt Publishing Ltd, 2018.

[2] Violante, Andre. “Simple Reinforcement Learning: Q-Learning.” *Towards Data Science*, Towards Data Science, 18 Mar. 2019, towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56.

[3] “Getting Started With Gym” *Gym,* OpenAI, 16 April 2020, https://gym.openai.com/docs/

[4] “Pong-v0” *Gym*, OpenAI, 16 April 2020, https://gym.openai.com/envs/Pong-v0/

[5] Dimitrakakis, Christos, and Ronald Ortner. "Decision making under uncertainty and reinforcement learning." (2018).

Appendix

To run the q-learning scripts below, make sure to have the following installed:

- GYM: use the following command in python “pip install gym”
- GYM Atari: use the following command in python “pip install gym[atari]”
- Atari-py: use the following command in python “pip install atari-py”
- C++: (required to render Gym environments) download Visual Studios (https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=BuildTools&rel=16) and install the “C++ build tools”

CartPole_RL_QLearning.py

```
# a #alpha or learning rate
# g #gamma or discount factor
# e #epsilon or exploration
# q_table #q-table
# state #current state
# statep #resulting state after action
# Q #current Q
# Qp #max Q from resulting state after action
# R #reward

import gym
import datetime
import math
import pickle
from sklearn.preprocessing import KBinsDiscretizer
from typing import Tuple
import numpy as np
```

```python
import matplotlib.pyplot as plt

env = gym.make("CartPole-v1")

#initialize variables for discretization
n_bins = ( 6 , 12 )
obsLow = [ env.observation_space.low[2], -math.radians(50) ]
obsHigh = [ env.observation_space.high[2], math.radians(50) ]
# desc_index = [10,30,6,12]
# obsLow = env.observation_space.low
# obsHigh = env.observation_space.high
# desc_size = (obsHigh - obsLow) / desc_index

#define discretization function
def discretizer( _ , __ , angle, pole_velocity ) -> Tuple[int,...]:
    """Convert continues state intro a discrete state"""
    est = KBinsDiscretizer(n_bins=n_bins, encode='ordinal', strategy='uniform')
    est.fit([obsLow, obsHigh ])
    return tuple(map(int,est.transform([[angle, pole_velocity]])[0]))
# def get_discrete_state(state):
#     #if out of upper and lower bounds assign to bounds
#     state = np.minimum(state,obsHigh)
#     state = np.maximum(state,obsLow)
#     #discretize
#     discrete_state = ((state-obsLow)/desc_size)[2:]
#     return tuple(discrete_state.astype(np.int))

#initialized action space
action_space = [0,1]
#initialize q-table
q_table = np.zeros(n_bins + (np.size(action_space),))

#define decay rates for a and e
def learning_rate(n : int , min_rate=0.01 ) -> float  :
    """Decaying learning rate"""
    return max(min_rate, min(1.0, 1.0 - math.log10((n + 1) / 25)))
def exploration_rate(n : int, min_rate= 0.1 ) -> float :
    """Decaying exploration rate"""
    return max(min_rate, min(1, 1.0 - math.log10((n  + 1) / 25)))

#initalize parameters
ep = 30 #number of episodes
g = 1
done = False

for episode in range(ep):
    #reset episode info
    total_reward = 0
    a = learning_rate(episode)
    e = exploration_rate(episode)
    done = False

    #first get the current state
    state = env.reset()
    d_state = discretizer(*state)

    while not done:

        #second choose an action give the current state, get Q value for that
action
        if np.random.random() < e:
            action_index = np.random.choice(np.size(action_space)) #choose a random
choice at rate epsilon
            action = action_space[action_index]
            Q = q_table[d_state + (action_index,)]
        else:
            action_index = np.argmax(q_table[d_state])  #otherwise choose best
likely to lead to a reward
            action = action_space[action_index]
            Q = q_table[d_state + (action_index,)]

        #third take action and observe new state
        statep, R, done, info = env.step(action)
        if episode > 10:
            env.render()
        #env.render()
        d_statep = discretizer(*statep)
        Qp = np.max(q_table[d_statep]) #get max Q value in based on new state
        #update q value based on new state
        #Q = (1-a)*Q + a*(R + g * Qp)
        Q = Q + a * (R + g * Qp - Q)
        q_table[d_state][action_index] = Q

        #assign prime state as current state
        state = statep
        d_state = d_statep
        total_reward += R

    #print final episode results
    print(total_reward)

env.close()
```

Pong_RL_QLearning.py

```
#PONG ENVIRONMENT
#Action space that is relevant
#0 is Do Nothing  #2 is Go UP  #5 is Go DOWN

#Observation space
#a = Opponent paddle position  632 - 710
#b = Ball position   0 - 6319
#c = Agent paddle position   5530 - 5608

import gym
import datetime
import pickle
import math
import numpy as np
import matplotlib.pyplot as plt

resume = True   #Change from True or False depending on if you have a model

env = gym.make("PongNoFrameskip-v4")  #Creates the Pong environment
env.reset()
startTime = datetime.datetime.now()


DISCOUNT = 0.995  #measure of how important we find future actions (Weight of
future reward vs current reward)
EPISODES = 10000  #number of attempts at a full game
SHOW_EVERY = 100  #when to render the game

actionSpace = np.array([0, 2, 5])
obsLow = np.array([632, 0, 5530])     #smallest values for observation (pixel
values)
obsHigh = np.array([711, 6320, 5609]) #largest values

DISCRETE_OS_SIZE = [5] * len(obsHigh) #5x5x5 combinations of paddle and ball
position
discrete_os_win_size = (obsHigh - obsLow) / DISCRETE_OS_SIZE

epsilon = 0.50 # chance to perform a random, exploratory action
START_EPSILON_DECAYING = 1  #over time we want our model to stop exploring
END_EPSILON_DECAYING = EPISODES // 2  #always divide out to integer
epislon_decay_value = epsilon / (END_EPSILON_DECAYING - START_EPSILON_DECAYING)

q_table = np.random.uniform(low=0, high=0, size= (DISCRETE_OS_SIZE +
[np.size(actionSpace)]))
#table of all possible combinations of position and velocity plus 3 possible
```

```
actions for every combination

if resume:
  model = pickle.load(open('pong3v4Winning.pickle', 'rb'))
  q_table = model['Q']
  epsilon = model['Epsilon']
  total_episodes = model['Episode']
  aggr_ep_rewards = model['Stats']
else:
  model = {}
  model['Q'] = q_table
  model['Epsilon'] = epsilon
  model['Episode'] = 0
  total_episodes = 0
  aggr_ep_rewards = {'ep': [], 'avg': [], 'min': [], 'max': []}

ep_rewards = []

def prepro(I):
  """ prepro 210x160x3 uint8 frame into 6000 (75x80) 1D float vector """
  I = I[35:193] # crop - remove 35px from start & 16px from end of image in x, to
reduce redundant parts of image (i.e. after ball passes paddle)
  I = I[::2,::2,0] # downsample by factor of 2.
  I[I == 144] = 0 # erase background (background type 1)
  I[I == 109] = 0 # erase background (background type 2)
  I = I.astype(np.float).transpose().ravel() # ravel flattens an array and
collapses it into a column vector
  a = np.where(I==213)  #213 is opponent paddle index from 632 to 710
  if np.size(a) ==0: #if no paddle yet initalize at obsLow
      a = obsLow[0] + ((obsHigh[0] - obsLow[0]) // 2)
  elif a[0][0] >= obsHigh[0]: #check if pong glitched and shows out of bounds
      a = obsHigh[0] - 1
  elif a[0][0] < obsLow[0]:
      a = obsLow[0]
  else:
      a = a[0][0]
  b = np.where(I==236)  #236 is ball index from 0 to 6319
  if np.size(b) ==0: #if no paddle yet initalize at obsLow
      b = obsLow[1] + ((obsHigh[1] - obsLow[1]) // 2)
  elif b[0][0] >= obsHigh[1]: #check if pong glitched and shows out of bounds
      b = obsHigh[1] - 1
  elif b[0][0] < obsLow[1]:
      b = obsLow[1]
  else:
      b = b[0][0]
  c = np.where(I==92)  #92 is your paddle index from 5530 to 5608
  if np.size(c) ==0: #if no paddle yet initalize at obsLow
```

```
      c = obsLow[2]
  elif c[0][0] >= obsHigh[2]: #check if pong glitched and shows out of bounds
      c = obsHigh[2] - 1
  elif c[0][0] < obsLow[2]:
      c = obsLow[2]
  else:
      c = c[0][0]
  obsv = np.array([a,b,c])
  return obsv

def learning_rate(n : int , min_rate=0.01 ) -> float  :
    #Decaying learning rate
    return max(min_rate, min(1.0, 1.0 - math.log10((n + 1) / 25)))

def get_discrete_state(state):
       discrete_state = (state - obsLow) / discrete_os_win_size
       return tuple(discrete_state.astype(np.int))

for episode in range(EPISODES):
       episode_reward = 0
       total_episodes = total_episodes + 1
       LEARNING_RATE = learning_rate(total_episodes)
       if episode % SHOW_EVERY == 0:
              render = True
       else:
              print('Episode: ', total_episodes)
              render = False

       observation = env.reset()    #get initial state
       obsv = prepro(observation)   #apply preprocessing
       discrete_state = get_discrete_state(obsv)
       done = False

       while not done:

              if np.random.random() > epsilon:
                     action = actionSpace[np.argmax(q_table[discrete_state])]
#exploitation
              else:
                     action = np.random.choice(actionSpace) #exploration

              actionIndex = np.where(actionSpace == action)[0][0]
              new_state, reward, done, info = env.step(action)
              episode_reward += reward
              new_obsv = prepro(new_state)
              new_discrete_state = get_discrete_state(new_obsv)
```

```
            if render:
                  env.render()

            if not done:

                  max_future_q = np.max(q_table[new_discrete_state])
                  current_q = q_table[discrete_state + (actionIndex, )]
                  new_q = (1 - LEARNING_RATE) * current_q + LEARNING_RATE *
(reward + DISCOUNT * max_future_q)
                  q_table[discrete_state + (actionIndex, )] = new_q

            discrete_state = new_discrete_state

      if END_EPSILON_DECAYING >= total_episodes >= START_EPSILON_DECAYING:
            epsilon -= epislon_decay_value

      ep_rewards.append(episode_reward)

      if not episode % SHOW_EVERY:
            model['Q'] = q_table
            model['Epsilon'] = epsilon
            model['Episode'] = total_episodes
            average_reward = sum(ep_rewards[-SHOW_EVERY:]) /
len(ep_rewards[-SHOW_EVERY:])
            aggr_ep_rewards['ep'].append(total_episodes)
            aggr_ep_rewards['avg'].append(average_reward)
            aggr_ep_rewards['min'].append(min(ep_rewards[-SHOW_EVERY:]))
            aggr_ep_rewards['max'].append(max(ep_rewards[-SHOW_EVERY:]))
            model['Stats'] = aggr_ep_rewards
            pickle.dump(model, open('pong3v4Winning.pickle', 'wb'))

            TimePassed = datetime.datetime.now() - startTime
            print("Total time passed - ", TimePassed)
            print(f"Episode: {total_episodes} avg: {average_reward} min:
{min(ep_rewards[-SHOW_EVERY:])} max: {max(ep_rewards[-SHOW_EVERY:])}")

env.close()

print(aggr_ep_rewards['ep'])

plt.plot(aggr_ep_rewards['ep'], aggr_ep_rewards['avg'], label="avg")
plt.plot(aggr_ep_rewards['ep'], aggr_ep_rewards['min'], label="min")
plt.plot(aggr_ep_rewards['ep'], aggr_ep_rewards['max'], label="max")
plt.legend(loc=4)
plt.show()
```