CAP 6629: Reinforcement Learning Course - Nerual Network Q-Learning GridWorld

> **Coursse Project 3**

- Shaun Pritchard
- April 7, 2022
- Professor Ni

Pesudo Code

This implementation uses a 5x5 gridworld example, The starting state has the value (0, 0) and the ending state has the value (4, 4). I will be working on an actor-critic approach (ADP) to approximate the Q-value function in a custom grid world problem with an implementation of a neural network to learn the Q-table.

$$e_c(t) = \alpha Q(t) - [Q(t-1) - r(t)]$$

The actor-critic (ADP) uses two networks where the critic network attempts to reduce the error function. Two perceptron networks are assimilated into the ctor-critic architecture using stochastic gradient descent (SGD) as the optimization algorithm.

By reducing $e_a(t)$, the actor network leads the agent system towards U_c, the optimal Q-value function objective:

$$e_a(t) = J(t) - U_c(t)$$

While the critic network reduces the error function $e_c(t)$,

The Q-value function $Q(t)$ repeats the passes of state-action pairs through the critic network while it tunes parameters that reduce $e_c(t)$

With the Actor-critic ADP class to implement the parameters. Define 3 method classes: the policy, actor, critic to instantiate the parameters. Actor networks are multilayer neural networks that take the current state as input, and output a probability distribution of the four possible actions that an agent could take i.e. Up, Down, Left, Right. The actor, critic, and policy networkls implements two fully connected hidden layers, ReLU is used in each of the fully connected layers which varies based on the number of nodes. We will implement a stochastic gradient descent(SGD) algorithm to optimize the input through a neural network. Afterwards, we will update the weights of the network by means of backpropagation.

- **Actor** Actor networks have four nodes corresponding to each action in the action space.With its softmax activation function, it identifies a probability distribution for each action the agent will take.

$E_a(t) = 1/2e_a^2$(t)

- **Critic** As with the actor network, the critic network t shares the same two fully connected layers, uses the same state input as it does for the actor network, is aware of the actions taken when moving between states and updating the weights, and outputs a predicted reward-to-go or Q learning utility value.

$E_c(t) = 1/2e_c^2$(t)

- **Policy** As with the actor and critic networks, the policy network has the same architecture. The NN mainly exists to keep track of the actor network's current plan to reach the goal state, as it outputs a similar probability distribution to the actor network and is not used for training or updating weights.

We will run 75 episodes per test at 6 test with 10 transtions per episode. While this did take longer (48.32 minutes) in google collab using 2BG GPU. I was curious to see the convergance diferential based the current paramters. The wiat was worth it.

Initialize Project

```
# Librarys and Imports

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow.keras import backend as K
from tensorflow.keras.optimizers import SGD, Adam
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Input
tf.__version__
```

NN Actor Critic ADP Agent Class

```
# Define Actor-Centric ADP Agent class which will build policy, actor and cri

class ADP(object):
    def __init__(self, α, β, γ=0.9, num_actions=4,
                layer1_size=2, layer2_size=1, input_dims=2):
        self.α=α # Alpha networks actor learning rate
        self.β=β # Beta networks Critic learning rate
        self.γ=γ # Gamma discount factor
        self.num_actions=num_actions #number of actions
        self.fc1_dims=layer1_size #number of nodes in first hidden layer
        self.fc2_dims=layer2_size #number of nodes in second hidden layer
        self.input_dims=input_dims #dimensionality of state space
        self.actor, self.critic,self.policy=self.actor_critic_NN() # Create a
        self.action_space = [i for i in range(self.num_actions)] # Initialize


# State transition of actor
    def step(self, state, action, reward, done):
        if self.action_space[action]==0: # Move up
            if state[1]<4:
                state[1] +=1
        elif self.action_space[action]==1: # Move down
            if state[1]>0:
                state[1]-=1
        elif self.action_space[action]==2: # Move left
            if state[0]<4:
                state[0]+=1
        elif self.action_space[action]==3: # Move right
            if state[0]>0:
                state[0]-=1
        if state[0]==4 and state[1]==4: # Verify the terminal state
            done=True
        return state, reward, done

# Based on the current predicted policy, this function selects an action from
    def actions(self, observations):
        state=observations[np.newaxis, :] # Initilize state and position
        prob=self.policy.predict(state)[0] # Update actions probability distr
        action=np.random.choice(self.action_space, p=prob) # Choose an action
        return action

# Using Kearas and Tensorflow to implement a NN function that builds an actor
    def actor_critic_NN(self):
        net_input=Input(shape=(self.input_dims,))
```

```
        dense1=Dense(units=self.fc1_dims, activation='relu')(net_input)
        dense2=Dense(units=self.fc2_dims, activation='relu')(dense1)
        probs=Dense(units=self.num_actions, activation='softmax')(dense2)
        values=Dense(units=1, activation='linear')(dense2)
        actor=Model(inputs=[net_input], outputs=[probs]) #  For each action,
        actor.compile(optimizer=Adam(learning_rate=self.α), loss='mse') # Cal
        critic=Model(inputs=[net_input], outputs=[values]) #  Critic network
        critic.compile(optimizer=Adam(learning_rate=self.β), loss='mse') # Ca
        policy=Model(inputs=[net_input], outputs=[probs]) #   he policy netwo
        return actor, critic, policy

# Based on an observations in the environment, this function updates the weig
    def learn(self, state, action, reward, state_, done):
        state=state[np.newaxis, :] # Current state in network
        state_=state_[np.newaxis, :] # Next state in network
        critic_value=self.critic.predict(state) # Current utility value
        critic_value_=self.critic.predict(state_) # Next utility value
        target=reward+self.γ*critic_value_*(1-int(done)) # Utility value afte
        actions=np.zeros((1, self.num_actions)) # Initilize array of number a
        actions[np.arange(1), action]=1.0     # Set positive actions
        self.actor.fit(state, actions, verbose=0) #train actor network
        self.critic.fit(state, target, verbose=0) #train critic network

```

Initialize Paramters

```
from matplotlib import axis
# Implement NN Q-Learning  grid world
rewards=np.zeros((5, 5)) # Grid World size 5 x 5
rewards[4][4]=1 # Reward varaiable
episodes=75 # Number of episode in itteration
steps_per_episodes_output=[] # output for calulation average steps per episo
for i in range(6, 12): # episodes per transition itteratins
    print(i)
    agent=ADP(α=0.00001, β=0.00005, layer1_size=2**i, layer2_size=2**(i-1))
    steps_per_episode=[]
    step_averages=[]
    # df = pd.DataFrame({"Steps in episode data": [i]})
    steps_per_episodes_output.append(steps_per_episode)
    for j in range(episodes):
        done=False
        count=0
        observations=np.array((0, 0))
        while not done:
            action=agent.actions(observations) # Action selected
```

```
            observations_, reward, done=agent.step(observations, action, rew
            agent.learn(observations, action, reward, observations_, done) #
            observations=observations_ # State of update
            count +=1 # Increment count

        steps_per_episode.append(count)
        avg_steps_per_episode=np.mean(steps_per_episode)
        step_averages.append(avg_steps_per_episode)
        # steps_per_episodes_output.append(step_averages)
        print('episoide:', i, 'Number of steps:', steps_per_episodes_output)

```

```
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,

episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
```

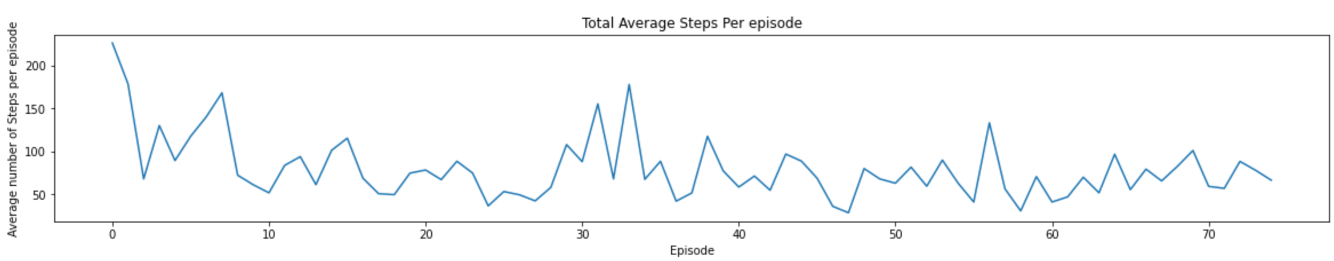
```
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
episoide: 11 Number of steps: [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27,
```

Analysis & Results

```python
print('Average Steps', steps_per_episodes_output)
```

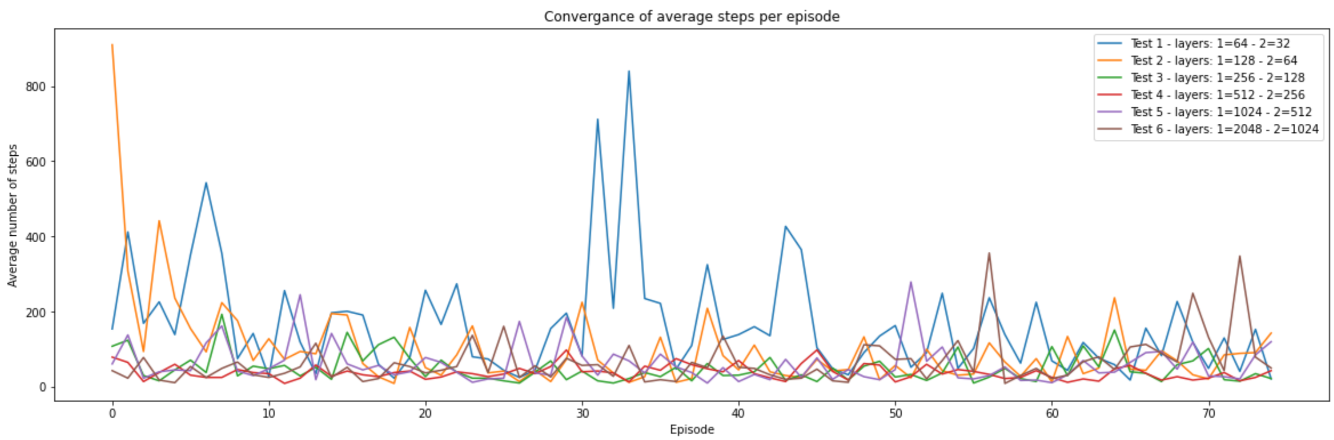
```
Average Steps [[154, 412, 169, 226, 139, 351, 543, 355, 75, 142, 27, 256, 119, 35, 197,
```

```python
average_steps_per_episode=np.mean(steps_per_episodes_output, axis=0)
plt.plot(average_steps_per_episode)
plt.rcParams["figure.figsize"] = (20,6)
plt.title('Total Average Steps Per episode')
plt.xlabel('Episode')
plt.ylabel('Average number of Steps per episode')
plt.show()
```




```python
plt.plot(steps_per_episodes_output[0], label='Test 1 - layers: 1=64 - 2=32')
plt.plot(steps_per_episodes_output[1], label='Test 2 - layers: 1=128 - 2=64'
plt.plot(steps_per_episodes_output[2], label='Test 3 - layers: 1=256 - 2=128
plt.plot(steps_per_episodes_output[3], label='Test 4 - layers: 1=512 - 2=256
```

```
plt.plot(steps_per_episodes_output[4], label='Test 5 - layers: 1=1024 - 2=51
plt.plot(steps_per_episodes_output[5], label='Test 6 - layers: 1=2048 - 2=10
plt.title('Convergance of average steps per episode')
plt.rcParams["figure.figsize"] = (20,3)
plt.xlabel('Episode')
plt.ylabel('Average number of steps')
plt.legend()
plt.show()
```



Results:

The computational time complexity was very high in this initial build. Data show that networks with larger hidden layers converge to optimal policies most efficiently. Comparatively, with 1024 layer 1 nodes and 512 layer 2 nodes, the network configuration began high and then converged to the same number of optimal steps as its previous configuration. Due to the randomness in the probability distributions, it appears the configuration with 512 nodes at layer 1 and 256 nodes at layer 2 had a few outliers in terms of number of steps in the episode.

These results suggest that larger networks can create a more efficient solution than smaller ones. In addition, these results are subpar compared to regular Q-learning. If the networks are trained more, or perhaps a negative outcome occurs during implementation, then perhaps they will be comparable.

1. According to the first implementation, a fully connected layer with 64 nodes and a second layer with 32 nodes converged to an optimal policy involving 150-200 steps towards the goal.

2. According to the second implementation, where 128 nodes have been added to the first fully connected layer and 64 have been added to the second fully connected layer, the optimal policy is around 200-250 steps towards the goal.
3. According to the third implementation, which has 256 nodes for the first and 228 nodes for the second fully connected layers, converged to an optimal policy of 150-200 steps.
4. According to the forth implementation, the first fully connected layer with 512 nodes converges to a policy of around 40-50 steps towards the goal when paired with the second fully connected layer with 256 nodes.
5. According to the fifth implementation, the first fully connected layer with 1024 nodes converges to a policy of around 50-80 steps towards the goal when paired with the second fully connected layer with 512 nodes.
6. According to the sixth implementation, the first fully connected layer with 2048 nodes converges to a policy of around 30-40 steps towards the goal when paired with the second fully connected layer with 1024 nodes.

Compare Project 2:

In project #2 faster computation and complexity with average step to action ratio when meeting the goal based on the algorithm. The algorithms implemented an agent who can enter a new situation with no prior knowledge and a bigger state space. Steps required for the agent to reach its goal are counted in each episode, then averaged over the 1000 iterations. Compared to the other results of the latter algorithms it is uncompritive.

In comparision to the Nerual network with Q-learning using ADP actor critic model. In the neural network variant, the optimal policy is not found at all, with between 20 and 50 steps per episode being an average. It can be challenging to construct a Q-table based on states and actions when the critic network only predicts one utility value in a time. This may change with more training. Nevertheless, due to limited computational resources and time constraints, each experiment was only able to run 75 episodes. In comparison to the algorithms in project 2, the neural networks' initial learning rates were quite small.