Balancing a CartPole System with Reinforcement Learning - A Tutorial

Ankith Gottigundala

***Abstract*— In this paper, we provide the details of implementing various reinforcement learning (RL) algorithms for controlling a Cart-Pole system. In particular, we describe various RL concepts such as Q-learning, Deep Q Networks (DQN), Double DQN, Dueling networks, (prioritized) experience replay and show their effect on the learning performance. In the process, the readers will be introduced to OpenAI/Gym and Keras utilities used for implementing the above concepts. It is observed that DQN with PER provides best performance among all other architectures being able to solve the problem within 150 episodes.**

I. Introduction

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns the optimal action for a given task through its repeated interaction with a dynamic environment that either rewards or punishes the agent's action. Reinforcement learning could be considered as a *semi-supervised* learning approach where the supervision signal required for training the model is made available indirectly in the form of rewards provided by the environment. Reinforcement learning is more suitable for learning dynamic behaviour of an agent interacting with an environment rather than learning static mappings between two sets of input and output variables. Over the years, a number of reinforcement learning methods and architectures have been proposed with varying success. However, the recent success of deep learning algorithms has revived the field of reinforcement learning finding renewed interest among researchers who are now successfully applying this to solve very complex problems which were considered intractable earlier [1]. Events such as artificial agents like AlphaGo beating world chapmpion Lee Sedol [3] [9] or IBM Watson winning the game of Jeopardy [5] [14] has attracted worldwide attention towards the rise of artificial intelligence which may surpass human intelligence in the near future [11] [4]. Reinforcement learning is a key paradigm to build such intelligent systems which can learn from its experience over time. Reinforcement algorithms are now being increasingly applied to Robotics, healthcare, recommender system, data centres, smart grids, stock markets and transportation [13].

In this paper, we will provide the implementation details of two well known reinforcement learning methods, namely, Q-learning [19] and Deep Q network (DQN) [15] for controlling a CartPole system. The objective is to provide a practical guide for implementing several reinforcement learning concepts by using using Python, OpenAI/Gym [16] and Keras [7]. Some of these concepts are ϵ-greedy policy, Q-learning algorithm, Deep Q-learning, experience replay, Dueling networks etc. It will be useful to students and researchers willing to venture into this field. Theoretical details and mathematical analysis of these concepts have omitted to maintain the brevity of this paper. Readers are, instead, referred to the relevant literature for more in-depth understanding of these concepts.

(a) (b)

Fig. 1: A Cart-Pole System: (a) Balanced state, (b) Unbalanced state

The rest of this paper is organized as follows. Various deep learning concepts with their implementation details are provided in next section. The results of applying these concepts to solve the CartPole problem is discussed in Section III. The conclusion is provided in Section IV.

II. Methods

A. The System

We use OpenAI Gym [16] to simulate the Cart-Pole system. Few snapshots of Cart-Pole states are shown in Figure 1. The left image shows the balanced state while the right image shows an imbalanced state. It consists of a cart (shown in black color) and a vertical bar attached to the cart using passive pivot joint. The cart can move left or right. The problem is to prevent the vertical bar from falling by moving the car left or right. One can see the animation of system behaviour under random action policy by executing the code given in Listing 1. The state vector for this system $\mathbf{x}$ is a four dimensional vector having components $\{x, \dot{x}, \vartheta, \dot{\vartheta}\}$. The action has two states: left (0) and right (1). The episode terminates if (1) the pole angle is more than $\pm 12^\circ$ from the vertical axis, or (2) the cart position is more than ± 2.4 cm from the centre, or (3) the episode length is greater than 200. The agent receives a reward of 1 for every step taken including the termination step. The problem is considered solved, if the average reward is greater than or equal to 195 over 100 consecutive episodes.

```
import gym
env = gym.make("CartPole-v0")
env.reset()
for i_episode in range(100):
  obs = env.reset()
  t = 0
  while not done:
    env.render()
    action = env.action_space.sample()
    obs,reward,done,info = env.step(action)
    t += 1
    if done:
      print("Done after {} steps".format(t))
      break;
env.close()
```

Listing 1: Simple Code to visualize Cartpole Animation

B. Q-Learning Algorithm

Q-learning algorithm uses Bellman Equation to form a Q-function to quantify the expected discounted future rewards that can be obtained by taking an action a_t for a given state s_t at any time t. Mathematically, it can be written as:

$$Q^{\pi}(s_t, a_t) = E\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | (s_t, a_t)\right] \quad (1)$$

where $R_i, \quad i = t+1, \ldots$ is the future rewards and γ is the discount factor. The objective is to update these Q function values through an iterative process by exploring all possible combinations of state and actions. Q-learning assumes a discrete observation space. Hence, the continuous state values are first discretized into fixed number of buckets by using `bucketize()` function as shown below:

```
def bucketize(state_value):
  bucket_indices = []
  for i in range(len(state_value)):
    if state_value[i] <= state_value_bounds[i][0]:
        # violates lower bound
        bucket_index = 0
    elif state_value[i] >= state_value_bounds[i][1]:
        # violates upper bound
        # put in the last bucket
        bucket_index = no_buckets[i] - 1
    else:
      bound_width = state_value_bounds[i][1] - \
              state_value_bounds[i][0]
      offset = (no_buckets[i]-1) * \
              state_value_bounds[i][0] / bound_width
      scaling = (no_buckets[i]-1) / bound_width
      bucket_index = int(round(scaling*state_value[i] -offset))
    bucket_indices.append(bucket_index)
  return(tuple(bucket_indices))
```

Listing 2: Discretizing continuous states into discrete states

The method involves creating a Q-table that stores rewards for all possible combinations of state and action choices. With a bucket size $(1, 1, 6, 3)$ for states and two dimensional action vector, the dimension of Q-table is $1 \times 1 \times 6 \times 3 \times 2$. The Q-learning algorithm is shown in Listing 4. It consists of the following four major steps:

1) Select an action as per the ϵ-greedy policy where ϵ controls the balance between exploration and exploitation. A random action is selected during exploration. During exploitation however, an action is selected based on agent's past experience. This is achieved by selecting an action that has maximum reward in the Q-table for the current state. Mathematically, we can write

$$a(s) = \arg\max_{a'} Q(s, a') \quad (2)$$

The exploration rate ϵ starts with a value of 1.0 at the beginning of the training and is reduced gradually over time. The corresponding code for selecting action is shown below:

```
class DQNAgent:
    def select_action(state_value, explore_rate):
        if random.random() < explore_rate:
            action = env.action_space.sample()     # explore
        else: # exploit
            action = np.argmax(q_value_table[state_value])
        return action
```

2) Obtain new observations with the above action and collect reward from the environment.
3) Update the Q-table using the following formulation:

$$Q(s, a) = Q(s, a) + \alpha[R(s, a) + \gamma \max_{a} Q(s', a) - Q(s, a)] \quad (3)$$

where α is the learning rate which is reduce monotonically from 1.0 to 0.1 as the training progresses.
4) Update the current state and repeat the above steps in the the next iteration.

The initial configurations and user-defined parameters for Q-learning algorithm is shown in Code Listing 3. The actual steps involved in the update of Q-table during each iteration is provided in Code Listing 4. These two parts could be executed together as a single python program.

```
import gym
import numpy as np
import random, math
import matplotlib.pyplot as plt

env= gym.make('CartPole-v0')
no_buckets = (1,1,6,3)
no_actions = env.action_space.n
state_value_bounds = list(zip(env.observation_space.low, env.observation_space.high))
state_value_bounds[1] = (-0.5, 0.5)
state_value_bounds[3] = (-math.radians(50), math.radians(50))
# define q_value_table - it has a dimension of 1 x 1 x 6 x 3 x 2
q_value_table = np.zeros(no_buckets + (no_actions,))
# user-defined parameters
min_explore_rate = 0.1; min_learning_rate = 0.1; max_episodes = 1000
max_time_steps = 250; streak_to_end = 120; solved_time = 199; discount = 0.99
no_streaks = 0

# Select an action using epsilon-greedy policy
def select_action(state_value, explore_rate): # omitted

# change the exploration rate over time.
def select_explore_rate(x):
  return max(min_learning_rate, min(1.0, 1.0 - math.log10((x+1)/25)))

# Change learning rate over time
def select_learning_rate(x):
  return max(min_learning_rate, min(1.0, 1.0 - math.log10((x+1)/25)))

# Bucketize the state_value
def bucketize(state_value):# omitted
```

Listing 3: Initial Configurations for Q-learning algorithm

```
# train the system
totaltime = 0
for episode_no in range(max_episodes):
  #learning rate and explore rate diminishes
  # monotonically over time
  explore_rate = select_explore_rate(episode_no)
  learning_rate = select_learning_rate(episode_no)
  # initialize  the environment
  observation = env.reset()
  start_state_value = bucketize_state_value(observation)
  previous_state_value = start_state_value
  done = False
  time_step = 0
  while not done:
    #env.render()
    # select action using epsilon-greedy policy
    action = select_action(previous_state_value, explore_rate)
    # record new observations
    observation, reward_gain, done, info = env.step(action)
    #update q_value_table
    best_q_value = np.max(q_value_table[state_value])
    q_value_table[previous_state_value][action] += learning_rate * (
        reward_gain + discount * best_q_value -
        q_value_table[previous_state_value][action])
    # update the states for next iteration
    state_value = bucketize_state_value(observation)
    previous_state_value = state_value
    time_step += 1
    # while loop ends here

  if time_step >= solved_time:
    no_streaks += 1
  else:
    no_streaks = 0
  if no_streaks > streak_to_end:
    print('CartPole problem is solved after {} episodes.', episode_no)
    break
env.close()
```

Listing 4: Q-learning algorithm

C. Deep Q Network (DQN) Algorithm

Q-learning algorithm suffers from the *Curse-of-Dimensionality* problem as it requires discrete states to form the Q-table. The computational complexity of Q-learning increases exponentially with increasing dimension of the state and action vector. Deep Q learning solves this problem by approximating the Q-value function $Q(s, a)$ with an artificial neural network. This is achieved by the function `build_model()` that uses Keras APIs to build a deep Q-network as shown below:

```
from keras.layers import Dense
from keras.optimizers import Adam
from keras.models import Sequential
class DQNAgent:
    # approximate Q-function with a Neural Network
    def build_model(self):
        model = Sequential()
        model.add(Dense(24, input_dim=self.state_size, activation='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.summary()
        model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
        return model
```

Listing 5: Creating a DQN using Keras APIs

It is a 4-24-24-2 feed-forward network with 4 inputs, 2 outputs and two hidden layers each having 24 nodes. Hidden nodes use a RELU activation function while the output layer nodes use a linear activation function. Having a deep network to estimate Q values allows us to work directly with continuous state and action values. The Q network needs to be trained to estimate Q-values for a given state and action pair. This is done by using the following loss function:

$$L_i(\vartheta_i) = E_{(s,a)\sim P(s,a)}[Q^*(s, a) - Q(s, a; \vartheta_i)]^2$$

where the target Q value $Q^*(s, a)$ for each iteration is given by

$$Q^*(s, a) = E_{s' \in S}\left[R(s, a) + \gamma \max_{a'} Q(s', a'; \vartheta_{i-1}) | s, a\right] \quad (4)$$

where $R(s, a)$ is the reward for the current state-action pair (s, a) obtained from the environment and $Q(s', a', \vartheta_{i-1})$ is the Q-value for the next state obtained using the Q-network weights from the last iteration. This is implemented using the code provided in the code listing 6. It also shows the code for computing Q targets for DDQN architecture which will be explained later in this paper.

[H]

```
class DQNAgent:
    def get_target_q_value(self, next_state, reward):
        # max Q value among the next state's action
        if self.ddqn:
            # DDQN
            # Current Q network selects the action
            # a'_max = argmax_a' Q(s',a')
            action = np.argmax(self.model.predict(next_state)[0])
            # target Q network evaluates the action
            # Q_max = Q_target(s', a'_max)
            max_q_value = self.target_model.predict(next_state)[0][action]
        else:
            # DQN chooses the max Q value among next actions
            # Selection and evaluation of action is on the target Q network
            # Q_max = max_a' Q_target(s', a')
            max_q_value = np.amax(self.target_model.predict(next_state)[0])
        return max_q_value
```

Listing 6: Obtaining the target Q values required for training DQN and DDQN

Sometimes it is convenient to have a separate network to obtain target Q values. It is called a target Q network Q' having same architecture as that of the original Q network. The weights for the target network is copied from the original network at regular intervals. This is shown in the code listing 7 where the `ddqn` flag needs to be set to `false`.

```
class DQNAgent:
    def update_target_model(self, tau = 0.1):
        '''Apply Polyak Averaging during weight update
        make tau = 1.0 for normal update
        '''
        if self.ddqn: # for DDQN
            weights = self.model.get_weights()
            target_weights = self.target_model.get_weights()
            for i in range(len(target_weights)):
                target_weights[i] = weights[i] * tau + target_weights[i] * (1-tau)
            # end of for loop
            self.target_model.set_weights(target_weights)
        else:  # for DQN
            self.target_model.set_weights(self.model.get_weights())
```

Listing 7: Weight update for the target network at regular intervals. Polyak Averaging can be implementing by setting ddqn flag.

D. Experience Replay

It has been shown that the network trains faster with a batch update rather than with an incremental weight update method. In a batch update, the network weights are updated after applying a number of samples to the network whereas in incremental update, the network is updated after applying each sample to the network. In this context, DQN uses a concept called *experience replay* where a random sample of past experiences of the agent is used for training the Q network. The experiences are stored in a fixed size replay memory in the form of tuples (s, a, r, s') containing current state, current action, reward and next state after each iteration. Once a sufficient number of entries are stored in the replay memory, we can train the DQN by using a batch of samples selected randomly from the replay memory. The exploration rate ϵ is reduced monotonically after each iteration of training.

```
class DQNAgent:
    def experience_replay(self):
        if len(self.memory) < self.train_start:
            return
        batch_size = min(self.batch_size, len(self.memory))
        mini_batch = random.sample(self.memory, batch_size)
        state_batch, q_values_batch = [], []
        for state, action, reward, next_state, done in mini_batch:
            # q-value prediction for a given state
            q_values_cs = self.model.predict(state)
            # target q-value
            max_q_value_ns = self.get_target_q_value(next_state, reward)
            # correction on the Q value for the action used
            if done:
                q_values_cs[0][action] = reward
            else:
                q_values_cs[0][action] = reward + \
                           self.discount_factor * max_q_value_ns
            state_batch.append(state[0])
            q_values_batch.append(q_values_cs[0])

        # train the Q network
        self.model.fit(np.array(state_batch),
                       np.array(q_values_batch),
                       batch_size = batch_size,
                       epochs = 1, verbose = 0)
        self.update_epsilon()
```

Listing 8: Training DQN using Experience Replay

E. Double DQN

Taking the maximum of estimated Q value as the target value for training a DQN as per equation 4 may introduce a maximization bias in learning. Since Q learning involves *bootstrapping*, i.e., learning estimates from estimates, such overestimation may become problematic over time. This can be solved by using double Q learning [8] [18] which uses two Q-value estimators, each of which is used to update the other. In this paper, we implement the version proposed in [18] that uses two models Q and Q' sharing weights at regular intervals. The network Q' is used for action selection while the network Q is used for action evaluation. That is, the target value for network training is obtained by using the following equation:

$$Q^{*}(s, a) \approx r_t + \gamma Q(s_{t+1}, \arg\max_{a} Q^{'}(s_t, a_t)) \tag{5}$$

We minimize the error between Q and Q^*, but have $Q^{'}$ slowly copy the parameters of Q through Polyak averaging: $\vartheta^{'} = \tau\vartheta + (1 - \tau)\vartheta^{'}$. The code for computing target Q value and weight update is shown in code listings 6 and 7 respectively where the `ddqn` flag needs to be set to `true`.

F. Dueling DQN

The Q-value $Q(s, a)$ tells us how good it is to take an action a being at state s. This Q-value can be decomposed as the sum of $V(s)$, the value of being at that state, and $A(s, a)$, the advantage of taking that action at the state (from all other possible actions). Mathematically, we can write this as

$$Q(s, a) = V(s) + A(s, a) \tag{6}$$

Dueling DQN uses two separate estimators for these two components which are then combined together through a special aggregation layer to get an estimate of $Q(s, a)$. By decoupling the estimation, intuitively the Dueling DQN can learn which states are (or are not) valuable without having to learn the effect of each action at each state. This is particularly useful for states where actions do not affect the environment in a meaningful way. In these cases, it is unnecessary to evaluate each action for such states and could be skipped to speed up the learning process.

Rather than directly adding individual components as shown in (6), the q-value estimate can be obtained by using the following two forms of aggregation:

$$Q(s, a) = V(s, \beta) + A(s, a, \alpha) - \max_{a^{'}} A(s, a^{'}, \alpha) \tag{7}$$

$$Q(s, a) = V(s, \beta) + A(s, a, \alpha) - \frac{1}{|A|}\sum_{a^{'}} A(s, a^{'}, \alpha) \tag{8}$$

where β and α are the weights for the networks $V(s)$ and $A(s, a)$ respectively. The first equation (7) uses max advantage value and the second equation (8) uses the average advantage value to estimate $Q(s, a)$ from $V(s)$. This form of aggregation apparently solves the *issue of identifiability*, that is - given $Q(s, a)$, it is difficult to find $A(s, a)$ and $V(s)$.

The implementation of Dueling DQN architecture involves replacing the `build_model()` function provided in Code Listing 5 with the function provided in the listing 9. A block-diagram visualization of the dueling architecture is shown in Figure 2. It uses Keras' Lambda function utility to implement the final aggregation layer.

```
class DQNAgent:
        def build_model(self):
         # Advantage network
         network_input = Input(shape=(self.state_size,), name='network_input')
         A1 = Dense(24, activation='relu', name='A1')(network_input)
         A2 = Dense(24, activation='relu', name ='A2')(A1)
         A3 = Dense(self.action_size, activation='linear', name='A3')(A2)
         # Value network
         V3 = Dense(1, activation='linear', name='V3')(A2)
         # Final aggregation layer to compute Q(s,a)
         if self.dueling_option == 'avg':
             network_output = Lambda(lambda x: x[0] - K.mean(x[0]) + x[1],\
                         output_shape=(self.action_size,))([A3,V3])
         elif self.dueling_option == 'max':
             network_output = Lambda(lambda x: x[0] - K.max(x[0]) + x[1],\
                         output_shape=(self.action_size,))([A3,V3])
         elif self.dueling_option == 'naive':
             network_output = Lambda(lambda x: x[0] + [1],\
                                   output_shape=(self.action_size,))([A3,V3])
         else:
             raise Exception('Invalid Dueling Option')

         model = Model(network_input, network_output)
         model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
         model.summary()
         plot_model(model, to_file='model.png', show_shapes=True,\
                                                  show_layer_names=True)
         return model
```

Listing 9: Implementing Dueling DQN Architecture with Keras

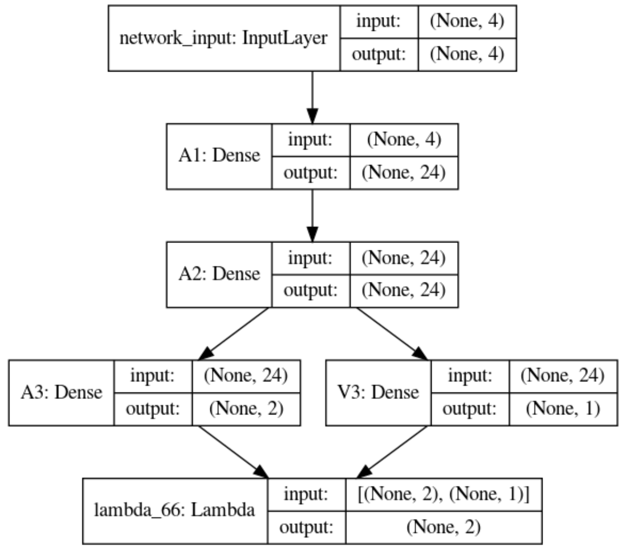

Fig. 2: Dueling Model architecture for DQN

G. The DQN Agent

The final DQN Agent class that implements both DQN, DDQN and Dueling versions of these architectures will appear something as shown in the code listing 10. The implementation details of functions which have been discussed earlier have been omitted here. Please remember to change your `build_model()` if you are implementing a dueling architecture. The main body of the program that uses this `DQNAgent` class to control the cart-pole system is provided in Code Listing 11. It is important to set the reward to -100 when the episode ends (or the `done` flag is set to `true`). This penalizes actions that prematurely terminates the episode.

```
import numpy as np
import random
from collections import deque
class DQNAgent:
    def init (self, state_size, action_size, ddqn_flag=False):
        self.state_size = state_size
        self.action_size = action_size
        # hyper parameters for DQN
        self.discount_factor =  0.9
        self.learning_rate = 0.001
        self.epsilon = 1.0          # explore rate
        self.epsilon_decay = 0.99
        self.epsilon_min = 0.01
        self.batch_size = 24
        self.train_start = 1000
        self.dueling_option = 'avg'
        # create replay memory using deque
        self.memory = deque(maxlen=2000)
        # create main model and target model
        self.model = self.build_model()
        self.target_model = self.build_model()
        # initialize target model
        self.target_model.set_weights(self.model.get_weights())

    # approximate Q-function with a Neural Network
    def build_model(self): # omitted

    # update target model at regular interval to match the main model
    def update_target_model(self): # omitted
```

```python
    # get action from the main model using epsilon-greedy policy
    def select_action(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        else:
            q_value = self.model.predict(state)
            return np.argmax(q_value[0])

    # save sample <s, a, r, s'>. into replay memory
    def add_experience(self, state, action, reward, next_state, done):
        self.memory.append((state,action,reward,next_state,done))

    # Compute target Q value
    def get_target_q_value(self, next_state, reward):

    # Train the model
    def experience_replay(self):  # omitted

    # decrease exploration, increase exploitation
    def update_epsilon(self):
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

Listing 10: The DQNAgent class implementation

```python
if __name__ == "__main__":
    # create Gym Environment
    env = gym.make('CartPole-v0')
    env.seed(0)
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    # create a DQN model
    agent = DQNAgent(state_size, action_size)
    score = []
    for e in range(EPISODES):
        done = False
        t = 0
        state = env.reset()
        state = np.reshape(state, [1, state_size])
        while not done:
            action = agent.get_action(state)
            next_state, reward, done, info = env.step(action)
            next_state = np.reshape(next_state, [1, state_size])
            reward = reward if not done else -100 #important step
            # add <s,a,r,s'> to replay memory
            agent.append_sample(state, action, reward, next_state, done)
            # Train through experience replay
            agent.experience_replay()
            t += 1
            state = next_state
            if done:
                # update target model for each episode
                agent.update_target_model()
                score.append(t)
                break

        # if mean score for last 100 episode bigger than 195, stop training
        if np.mean(score[-min(100, len(score)):]) >= (env.spec.max_episode_steps-5):
            print('Problem is solved in {} episodes.'.format(e))
            break
    env.close()
```

Listing 11: The main code for using DQNAgent for balancing the CartPole System.

H. Prioritized Experience Replay

Prioritize experience replay (PER) [17] is based on the idea that some experiences may be more important than others for training, but might occur less frequently. Hence, it will make more sense to change the sampling distribution by using a criterion to define the priority of each tuple of experience. PER will be more useful in cases where there is a big difference between the predicted Q-value and its TD target value, since it means that there is a lot to learn about it. The priority of an experience is therefore defined as:

$$p_t = |\delta_t| + e \tag{9}$$

where $|\delta_t|$ is the magnitude of TD error and e is a constant that ensures that no experience has zero probability of getting selected. Hence, the experiences are stored in the replay memory along with their priorities as a tuple $< s_t, a_t, r_t, s_{t+1}, p_t >$. However, one can not simply do a greedy prioritization as it will lead to training with the same experiences (having bigger priority) and hence over-fitting. Hence, this priority is converted into stochastic probability given by

$$P(i) = \frac{p_i^a}{\sum_k p_k^a} \tag{10}$$

where a is a hyperparameter used to reintroduce some randomness in the experience selection for the replay buffer. $a = 0$ will lead to pure uniform randomness while $a = 1$ will select the experiences with highest priorities. Priority sampling, in general, will introduce a bias towards high-priority samples and may lead to over-fitting. To correct this bias, we use important sampling (S) weights that will adjust the updating by reducing the weights of the often seen samples. The weights for each sample is given by:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^b \tag{11}$$

The role of hyperparameter b is to control how much these importance sampling weights affect the learning. In practice, b is selected to be 0 in the beginning and is annealed upto 1 over the duration of training, because these weights are more important in the end of learning when the Q-values begin to converge. To reduce the computational burden a `sumTree` data structure is used which provides $O(\log n)$ time complexity for sampling experiences and updating their priorities. A `sumTree` is a Binary Tree, that is a tree with a maximum of two children for each node. The leaves contain the priority values and a data array containing experiences. The code for creating `sumTree` data structure and the corresponding replay memory is shown in the code block listing 12 and 13 respectively. The training function `experience_replay()` will be slightly different from the one given in code listing 8 as it will make use of sumtree data structure and require updating priorities with each iteration. The code for the modified version of `experience_replay()` function is provided in code listing 14. The main changes in the `DQNAgent` class definition is shown in the code listing 15. The main program required for solving the Cartpole problem remains the same as given in code listing 11. The effect of PER is discussed later in the experiment section.

```python
import numpy as np
class SumTree(object):
    data_pointer = 0
    def __init__(self, capacity):
        # Number of leaf nodes (final nodes) that contains experiences
        self.capacity = capacity    #
        self.tree = np.zeros(2 * capacity - 1)
        self.data = np.zeros(capacity, dtype=object)

    def add(self, priority, data):
        # Look at what index we want to put the experience
        tree_index = self.data_pointer + self.capacity - 1
        self.data[self.data_pointer] = data # Update data frame
        self.update (tree_index, priority) # Update the leaf
        self.data_pointer += 1 # Add 1 to data_pointer
        if self.data_pointer >= self.capacity:  # If we're above the capacity
            self.data_pointer = 0 # we go back to first index (overwrite)

    def update(self, tree_index, priority):
        # Change = new priority score - former priority score
        change = priority - self.tree[tree_index]
        self.tree[tree_index] = priority
        while tree_index != 0: # propagate changes through the tree
            tree_index = (tree_index - 1) // 2
            self.tree[tree_index] += change

    def get_leaf(self, v):
        parent_index = 0
        while True:
            left_child_index = 2 * parent_index + 1
            right_child_index = left_child_index + 1
            # If we reach bottom, end the search
            if left_child_index >= len(self.tree):
                leaf_index = parent_index
                break
            else: # downward search, always search for a higher priority node
                if v <= self.tree[left_child_index]:
                    parent_index = left_child_index
                else:
                    v -= self.tree[left_child_index]
                    parent_index = right_child_index
```

```python
        data_index = leaf_index - self.capacity + 1
        return leaf_index, self.tree[leaf_index], self.data[data_index]

    @property
    def total_priority(self):
        return self.tree[0] # Returns the root node
```

Listing 12: The sum tree data structure for creating replay memory.

```python
import numpy as np
class Memory(object):
    # stored as ( state, action, reward, next_state ) in SumTree
    PER_e = 0.01 # hyper parameter
    PER_a = 0.6  # hyper parameter
    PER_b = 0.4  # importance-sampling, from initial value increasing to 1
    PER_b_increment_per_sampling = 0.001
    absolute_error_upper = 1. # clipped abs error

    def __init__(self, capacity):
        self.tree = SumTree(capacity) # Making the tree

    def store(self, experience): # Find the max priority
        max_priority = np.max(self.tree.tree[-self.tree.capacity:])
        if max_priority == 0:
            max_priority = self.absolute_error_upper
        self.tree.add(max_priority, experience)

    def sample(self, n):
        minibatch = []
        b_idx = np.empty((n,), dtype=np.int32)
        priority_segment = self.tree.total_priority / n  # priority segment
        for i in range(n):
            # A value is uniformly sample from each range
            a, b = priority_segment * i, priority_segment * (i + 1)
            value = np.random.uniform(a, b)
            # Experience that correspond to each value is retrieved
            index, priority, data = self.tree.get_leaf(value)
            b_idx[i]= index
            minibatch.append([data[0],data[1],data[2],data[3],data[4]])
        return b_idx, minibatch

    def batch_update(self, tree_idx, abs_errors):
        abs_errors += self.PER_e # convert to abs and avoid 0
        clipped_errors = np.minimum(abs_errors, self.absolute_error_upper)
        ps = np.power(clipped_errors, self.PER_a)
        for ti, p in zip(tree_idx, ps):
            self.tree.update(ti, p)
```

Listing 13: The replay memory using sum tree data structure.

```python
class DQNAgent:
    def experience_replay(self):
        ''' Training on Mini-Batch with Prioritized Experience Replay  '''
        # create a minibatch through prioritized sampling
        tree_idx, mini_batch = self.memory.sample(self.batch_size)
        current_state = np.zeros((self.batch_size, self.state_size))
        next_state = np.zeros((self.batch_size, self.state_size))
        qValues = np.zeros((self.batch_size, self.action_size))
        #action, reward, done = [], [], []
        action = np.zeros(self.batch_size, dtype=int)
        reward = np.zeros(self.batch_size)
        done = np.zeros(self.batch_size, dtype=bool)
        for i in range(self.batch_size):
            current_state[i] = mini_batch[i][0]    # current_state
            action[i] = mini_batch[i][1]
            reward[i] = mini_batch[i][2]
            next_state[i] = mini_batch[i][3]   # next_state
            done[i] = mini_batch[i][4]
            qValues[i] = self.model.predict(current_state[i]\
                                .reshape(1,self.state_size))[0]
            max_qvalue_ns =
            self.get_maxQvalue_nextstate(next_state[i]\
                              .reshape(1,self.state_size))
            if done[i]:
                qValues[i][action[i]] = reward[i]
            else:
                qValues[i][action[i]] = reward[i] + \
                        self.discount_factor * max_qvalue_ns
        # update  priority in the replay memory
        target_old = np.array(self.model.predict(current_state))
        target = qValues
        indices = np.arange(self.batch_size, dtype=np.int32)
        absolute_errors = np.abs(target_old[indices, \
                np.array(action)]- target[indices, np.array(action)])
        self.memory.batch_update(tree_idx, absolute_errors)
        # train the model
        self.model.fit(current_state, qValues,
                       batch_size = self.batch_size,
                       epochs=1, verbose=0)
        # update epsilon with each training step
        self.update_epsilon()
```

Listing 14: The code for training with prioritized experience replay.

```python
from sumtree import sumTree and Memory
class DQNAgent:
    def __init__(self):
        self.memory = Memory(memory_size)

    def add_experience(self, state, action, reward, next_state, done):
        experience = [state, action, reward, next_state, done]
        self.memory.store(experience)

    def experience_replay(self): # provided separately

    # rest of functions remain same as before
```

Listing 15: Main changes to the DQNAgent class definition provided in code listing 10

S.No.	Parameter	Value
1	discount factor, γ	0.9
2	learning rate	0.001
3	Exploration rate, ϵ	1
4	ϵ_{min}	0.01
5	polyak averaging factor, τ	0.1

TABLE I: Values of user-defined parameters used for simulation

III. Experiments and Results

This section provides the details of experiments carried out to evaluate the performance of various reinforcement learning models described in the previous sections. This is described next in the following subsections.

A. *Software and Hardware Setup*

The complete implementation code for this paper is available on GitHub [12]. The program is written using Python and Keras APIs [7]. It takes about a couple of hours (2-3 hours) for running about 1000 episodes on a HP Omen laptop with a Nvidia GeForce RTX 2060 GPU card with 6 GB of video ram. It is also possible to make use of freely available GPU cloud such as Google Colab [6] [2] or Kaggle [10] if you don't own a GPU machine.

B. *Performance of various RL models*

The performance of Q-learning algorithm is shown in Figure 3. As one can see, Q-learning algorithm is able to solve the problem within 300 episodes. It also shows the learning rate that decreases monotonically with training iterations. The performance of DQN Algorithm with experience replay is shown in Figure 4. It is clearly faster than the standard Q-learning algorithm and is found to solve the problem with 200 episodes. The performance comparison for DQN, Double DQN (DDQN) and DDQN with Polyak Averaging (PA) is shown in Figure 5. While all of them are able to solve the problem within 300 episodes, DQN is clearly the fastest. DDQN and DDQN-PA do not provide any perceptible advantage over DQN. This could be because the problem is itself too simple and does not require these complex architectures. The replay memory size of 2000 and batch size of 24 is used for producing the result shown in 5. Polyak Averaging (PA) tends to slow down the learning process and it is more commonly known as the soft method for updating target model. Similarly, the dueling versions of DQN or DDQN architectures fail to provide convergence within 300 episodes as shown in Figure 6. The problem might be too simple to make use of these complex architectures. Dueling architectures with Prioritized Experience Replay (PER) has been shown to provide remarkable improvement in ATARI games. It can be seen that Dueling-DQN is faster than Dueling-DDQN as it uses less number of parameters. The performance of DQN algorithms is also affected by changing the values of parameters such as the replay memory size (MS) and batch size (BS) selected for experience replay.

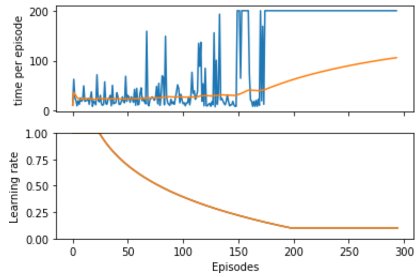

Fig. 3: Performance of Q-learning Algorithm. The standard Q-learning solve the problem within 300 episodes. The problem is considered solved if the average of last 100 scores is >= 195. Learning rate is decreased monotonically with increasing training episodes.

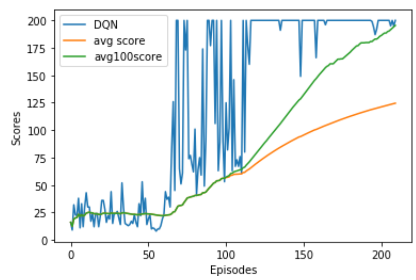

Fig. 4: Performance of DQN Algorithm. Avg100score is the average of last 100 episodes. The problem is considered to be solved when average of last 100 scores is >= 195 for CartPole-V0.

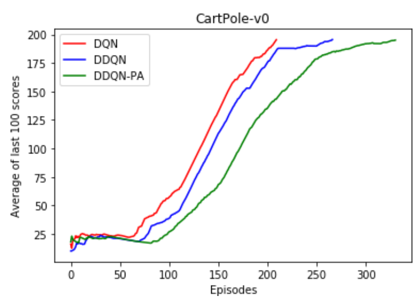

Fig. 5: Performance Comparison of DQN and DDQN architectures. DQN is found to solve the problem faster compared to DDQN and DDQN-PA architectures.

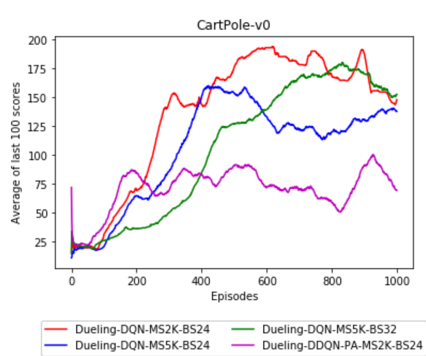

Fig. 6: Performance of Dueling DQN and DDQN architectures. Dueling architectures fail to solve the problem within 1000 episodes.

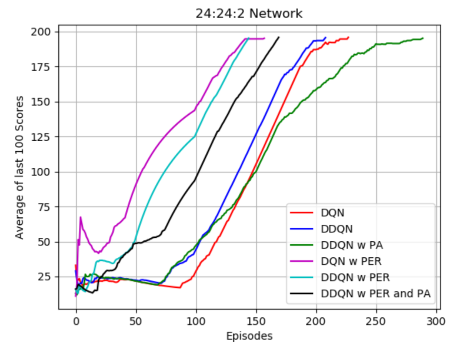

Fig. 7: Effect of Prioritized Experience Replay (PER) on DQN and DDQN network models. The program is terminated when the average of last 100 scores exceed 195.

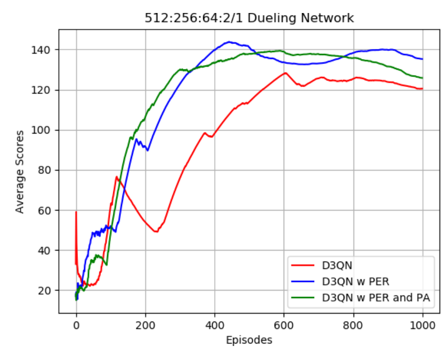

Fig. 8: Effect of PER on Dueling-DDQN (D3QN) Model Architecture

C. Effect of Prioritized Experience Replay

The effect of prioritized experience replay (PER) on DQN and DDQN architectures is shown in Figure 7. These results are produced using a 3 layer network (24-24-2) architecture with about 770 parameters, sampling batch size of 24 and a replay memory size of 2000. The values of various hyper-parameters are as shown in Table I. The best performance out of 2-3 independent runs are shown in this plot. As one can see, PER provides clear improvement over the normal DQN and DDQN implementations. The same is seen in case of Dueling-DDQN (D3QN) architecture as shown in Figure 8. This result for this figure is produced by using a 512-256-64-2/1 network architecture with about 150,531 parameters, a sampling batch size of 32 and a replay memory capacity of 10,000. As one can see, D3QN with PER provides higher average scores compared to that obtained using only D3QN. Soft target update using Polyak Averaging (PA) does not necessarily provide any significant advantage over PER. It has the effect of slowing down the learning process as is evident from these two figures. The best performance is obtained using DQN with PER that learns to solve the problem in about 50 episodes (average of last 100 is take as the termination criterion).

IV. CONCLUSIONS

This is a tutorial paper that provides implementation details of a few reinforcement learning algorithms used for solving the Cart-Pole problem. The implementation code is written in Python and makes use of OpenAI/Gym simulation framework and Keras deep learning tools. It is observed that DQN is considerably faster compared to the standard Q-learning algorithms and allows the use of continuous state values. DDQN and Dueling architectures do not provide any significant improvement over DQN as the problem is too simple to warrant such complex architectures. Further improvement in performance is obtained by using Prioritized Experience Replay (PER). DQN with PER is shown to provide the best performance so far. The codes provided could be executed on Google Colab which provides free access to a GPU cloud. We believe that these details will be of interest to students and novice practitioners and will motivate them to explore further and make novel contributions to this field.

REFERENCES

1 K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

2 E. Bisong. Google colaboratory. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pages 59–64. Springer, 2019.

3 S. Borowiec. Alphago seals 4-1 victory over go grandmaster lee sedol. *The Guardian*, 15, 2016.

4 J. Fang, H. Su, and Y. Xiao. Will artificial intelligence surpass human intelligence? *Available at SSRN 3173876*, 2018.

5 D. A. Ferrucci. Introduction to this is watson. *IBM Journal of Research and Development*, 56(3.4):1–1, 2012.

6 Google Colaboratory. Online gpu cloud by google. `https://colab.research.google.com/`.

7 A. Gulli and S. Pal. *Deep learning with Keras.* Packt Publishing Ltd, 2017.

8 H. V. Hasselt. Double q-learning. In *Advances in neural information processing systems*, pages 2613–2621, 2010.

9 S. D. Holcomb, W. K. Porter, S. V. Ault, G. Mao, and J. Wang. Overview on deepmind and its alphago zero ai. In *Proceedings of the 2018 international conference on big data and education*, pages 67–71, 2018.

10 Kaggle. Online gpu cloud with datasets. `https://www.kaggle.com/`.

11 P. Kraikivski. Seeding the singularity for ai. *arXiv preprint arXiv:1908.01766*, 2019.

12 S. Kumar. Reinforcement learning code for cartpole system. `https://github.com/swagatk/RL-Projects-SK.git`, 2020.

13 Y. Li. Reinforcement learning applications. *arXiv preprint arXiv:1908.06973*, 2019.

14 J. Markoff. Computer wins on jeopardy!: trivial, its not. *New York Times*, 16, 2011.

15 V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

16 OpenAI Gym. Toolkit for developing and comparing reinforcement learning algorithms. `https://gym.openai.com/`.

17 T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

18 H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

19 C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.