Reinforcement Learning

contact: *cursifrancesco@gmail.com*

INTRODUCTION

Reinforcement Learning (RL) is a branch of Machine Learning that considers an **Agent** and and **Environment**.

In a RL scenario, the agent is not told what to do but it must discover itself the best actions to take in order to maximize a certain *reward*.

RL differs from *Supervised Machine Learning* because it does not try to find any relationship between some input and some labelled output data. In fact, the goal of RL is to find possible optimal actions, but these actions might have never been identified before, therefore no appropriate label might exist. RL is also different from *Unsupervised Machine Learning*, where patterns among data are sought, yet without any labelled output. As mentioned, RL does not try to find patterns or structures in collected data, but actions to maximize a certain reward by observing how these actions affect the environment.

RL is linked to the concepts of *Exploitation* and *Exploration*. The agent needs to explore as many possible actions to maximize the reward, but it needs to exploit its knowledge about past actions that resulted in high reward too.

How does RL Work?

RL are typically composed of:

- *Agent*: it is the learner (the controller or software program) that takes and learns actions.
- *Environment*: is the system interacting with the agent. In model-based RL, a model of the environment is built and used for planning; in model-free RL there is no model and the agent only relies on trial-and-error strategies.
- *Policy*: defines the agent's behaviour in the environment. the policy defines the way actions will be taken by the agent. It can be considered as a mapping from the perceived state of the environment to the agent's actions in those states.
- *Reward*: defines the goal in the learning problem at each time step. It is the function to maximize and it depends on the agent's current action and the environment's current state.
- *Value Function*: similar to the reward, but considers the overall reward in the long run.

In RL the algorithm generally follows the following steps:

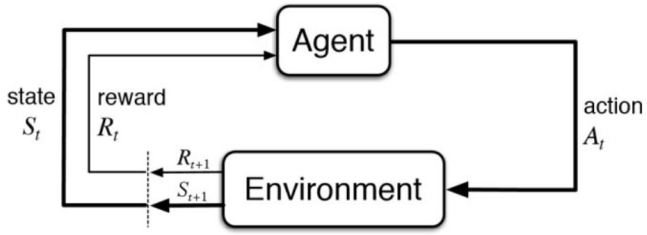

Fig. 1: RL framework

1) the agent takes an action that affects the environment;
2) the agent senses the environment, getting its state and the corresponding reward for the action taken;
3) based on the reward and on the accumulation of past actions and rewards, the agent understands if the action was good or not.

The process terminates when the total reward is maximized and the specified goal achieved. This means that the agent has learned the actions to take to complete the task.

This part will cover:

- RL as Markov Decision Process and learning methods such as Q-learning;
- Deep Reinforcement Learning and methods such as Deep Q-Network, Policy Gradients.

RL AS MARKOV DECISION PROCESS

The **Markov Decision Process** (MDP) provides a framework to solve RL problems, as almost every RL problem can be formulated as an MDP.

An MDP is represented by:

- a set of states where the agent can actually be;
- a set of actions the agent can take;
- a transition probability, namely the probability of moving from a state s to the next one s' by taking action a;
- a reward probability $R^a_{ss'}$, namely the probability of a reward acquired when moving from s to s' by means of a;
- a discount factor γ to penalize past rewards.

Given a set of rewards r_t for each time step t, the *Return* is defined as the sum of the rewards at each time step. In order to give different importance to rewards at closer time steps over those at very past time steps, the discount factor is used as a multiplying coefficient, e.g:

$$R_t = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^n r_{t+n+1} \tag{1}$$

An important element to consider in RL is the *Policy Function* $\pi(s)$ that maps states to actions, thus identifying the action to take give a particular state. The goal of RL is to find the optimal policy in any given state in order to maximize the reward.

A *State Value Function* or *Value Function* $V(s)$ is a function that defines the value of a state following a chose policy. It therefore estimates how good for the agent is to be in that state, given a the policy π. This function can be considered as the expected return when starting from state s and following the policy π thereafter:

$$V^\pi(s) = \mathbb{E}_\pi[R_t | s_t = s] \ . \tag{2}$$

The *Action Value Function* $Q(s)$ specifies how good it is for the agent to perform an action a when in a given state s and following a policy π. It can be computed by considering the expecting return starting from a state s and given the action a, following the policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi[R_t | s_t = s, a_t = a] \ . \tag{3}$$

Solving a MDP means finding the optimal policies and corresponding value function. For a MDP with a known environment, meaning knowing the transition probabilities to pass form a state to another and the rewards, it is possible to obtain the optimal value function and policies by solving the **Bellman Equation**.

Dynamic Programming

One way of solving the Bellman equation with known model of the environment is by **Dynamic Programming** (DP). There exist two approaches:

- Value iteration: starting from a random value function, the Q function for all states and actions pair is computed and then updated.
- Policy iteration: starting from a random policy, the value function for that policy is computed and the policy optimized.

Monte Carlo Methods

With **Monte Carlo Methods**, the goal is to find approximate solutions by random sampling.

Monte Carlo Methods do not require any model of the environment, but just samples of states, actions, and rewards. Their goal is to approximate the value function $V(s)$ of any given policy.

In a basic form the steps are:

- initialize a random value for the value function;
- for each state in the episode, compute the return;
- append the return to a list of returns;
- approximate the value function to the average of the episode returns.

Temporal Difference

Temporal Difference (TD) learning is a model-free approach, that, unlike Monte Carlo methods, does not have the limitation to wait until an episode ends (it is thus suitable for non-episodic tasks too).

This techniques takes the advantages both of Monte Carlo and DP, since it requires no model and no wait until end of episode to estimate the value function. The value function is instead approximated by bootstrapping, namely by considering previously learned estimates. The value function is updated as:

$$V(s) = V(s) + \alpha(r + \gamma V(s') - V(s)) \ , \tag{4}$$

where r is the current reward, γ the discount factor, s the previous state and s' the current state, and α a learning rate. This shows that the value function is being updated by adding the difference between the current actual reward $r + \gamma V(s')$ and the expected one $V(s)$. This difference is called *TD Error*, that the algorithm tries to minimize over the time steps.

The steps are generally:

- initialize $V(s)$ to zero for all states;
- for every step in the episode, take an action starting from state s and obtain the reward r, and move t next state s';
- update $V(s)$ for the previous state;
- repeat the steps until all the states have been explored.

Two known TD algorithms are **Q Learning** and **State-Action-reward-State-Action** (SARSA). The work in a similar way and approximate the state-action function $Q(s, a)$ as:

- initialize random Q for each state and action;
- choose an action a in state s using epsilon-greedy policy*;
- perform the action and move to new state s', and receive reward r;
- update the state-action function Q for the past state given that action;
- repeat the steps until reaching final state.

* an epsilon-greedy policy takes an action a based on a probability ϵ or take the best action (the one with highest Q in that state) with probability $1 - \epsilon$.

In Q-Learning the Q function is updated as:

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_a Q(s', a) - Q(s, a)) \tag{5}$$

A. Example

Let's consider the Taxi Driver example. The agent is a taxi car; the goal is for it to learn how to pick and drop off customers in the best way.

There are 500 discrete states since there are 25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is in the taxi), and 4 destination locations. Note that there are 400 states that can actually be reached during an episode. The missing states correspond to situations in which the passenger is at the same location as their destination, as this typically signals the end of an episode. Four additional states can be observed right after a successful episodes, when both the passenger and the taxi are at the destination. This gives a total of 404 reachable discrete states.

As for the actions in each state, they correspond to: move south (0), move north (1), move east (2), move west (3), pickup (4), dropoff (5).

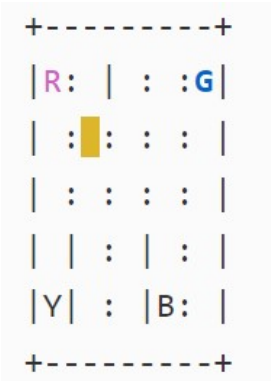

Fig. 2: The taxi driver environment

A Q-Learning algorithm in Python is implemented as:

```
import random
import gym
import time
import numpy as np

#Instantiate environment from gym
env = gym.make("Taxi-v3")

#Initialize Q values for each state and action to 0
q_table = np.zeros([env.observation_space.n, env.action_space.n])

#Parameters for updating Q values
alpha = 0.1
gamma = 0.6
epsilon = 0.1

MaxEpisodes = 100000

#This defines the epsilon-greedy policy
def greedy_policy(state,epsilon):
    if random.uniform(0, 1) < epsilon:
        action = env.action_space.sample()  # Explore action space
    else:
        action = np.argmax(q_table[state])  # Exploit learned values

    return action

#Train the RL agent to populate the q_table
for i in range(1, MaxEpisodes):
    state = env.reset() #start from a random state

    done = False

    r = 0 #total episodic reward

    #the episode ends when driver drops off passenger
    while not done:

        action = greedy_policy(state,epsilon) #get the action for the current state

        next_state, reward, done, info = env.step(action)

        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state]) #get maximum q value for the current state among all actions

        #update q value for state s and action a
        new_value = old_value + alpha * (reward + gamma * next_max-old_value)
        q_table[state, action] = new_value

        #update total reward
        r = r+reward

        state = next_state

    print("Episode ",i," reward:", reward)
print("Training finished.\n")
```

Once the RL agent has been trained, it is possible to exploit the populated Q value table to obtain the optimal path, starting from any random state as:

```
episodes = 10 #test on 10 episodes

for _ in range(episodes):
    state = env.reset() #set random initial state

    r = 0

    done = False

    while not done:

        env.render() #show the simulation

        #in this scenario the action is chosen among the optimal ones
        action = np.argmax(q_table[state])
        state, reward, done, info = env.step(action)

        r = r+reward

    print("episode:", i, "reward: ",r)
```

DEEP RL

Deep Reinforcement Learning is a combination of Machine Learning (ML) and Reinforcement Learning.

This has gained a lot of popularity because of difficulties in estimating value functions and optimal policies when there are many states and possible actions in a certain setting.

The use of ML is to build such approximations only from the events that have been seen and thus generalize to other possible scenarios.

Popular methods for Deep RL are:

- Deep Q-Learning;
- Actor-Critic Method;
- Policy Gradient methods such as Deep Deterministic Policy Gradient and Proximal Policy Optimization.

Deep Q Network

Deep Q Network (DQN) is a method to approximate the Q value function by using neural networks for all possible actions in each state.

The loss for the network is the update error:

$$L = r + \gamma \max_a \hat{Q}(s', a) - \hat{Q}(s', a) \tag{6}$$

where $\hat{Q}(s', a)$ is the network predicted state-action value function. The network input is the current state and the output is the q value for each possible action in that state.

Generally, in order to improve the learning performance of the network an *experience replay buffer* is used in order to train the network not only on the current experience, but also on some past ones.

Another improvement consists in using two different networks: a *target network* which is frozen and reanimated only at some intervals;the *actual network* which is the actual one used for estimating the q value.

As an example, let's consider a cart-pole agent that needs to learn how to be stable in the vertical position. the actions are either 0 or 1 (move left or right) and the state is the cart position and velocity, and the pole angle and angular velocity.

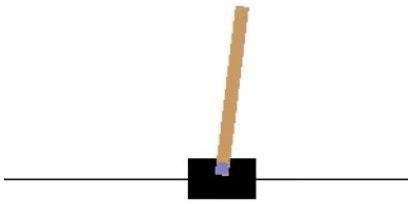

Fig. 3: The cart-pole environment

```
import gym
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import random

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

#Create Deep Q Network
class DQNet():
        def __init__(self, input_dim, h_sizes, output_dim):
            super(DQNet, self).__init__()
```

```
            inputs = tf.keras.layers.Input(shape=input_dim)
            x = inputs
            for k in range(len(h_sizes)):
                x = tf.keras.layers.Dense(h_sizes[k], activation="relu")(x)

            output = tf.keras.layers.Dense(output_dim)(x)

            self.model = tf.keras.Model(inputs=inputs, outputs=output)

            self.loss = tf.keras.losses.MeanSquaredError()
            self.optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

#function to update network weights
        def update(self,state,y):
            with tf.GradientTape() as tape:

                y_pred = self.predict(state)

                loss_value = self.loss(y_pred,y)

                grads = tape.gradient(loss_value, self.model.trainable_variables)
                self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))
#predict q values
        def predict(self,state):
            return self.model(state)

#This function is used if memory replay is enabled
        def replay(self,memory,size,gamma=0.9):
            if len(memory) >= size:
                states = np.empty((0,4))
                targets = np.empty((0,2))

                #take a random set of memory of size=size
                batch = random.sample(memory,size)

                for state, action, next_state,reward,done in batch:
                    states = np.vstack((states,state))

                    q_values = self.predict(state).numpy()

                    if done:
                        q_values[0,action] = reward
                    else:
                        q_values_next = self.predict(next_state).numpy()
                        q_values[0, action] = reward + gamma * np.max(q_values_next)

                    targets = np.vstack((targets, q_values))

                self.update(states,targets)

#Create target network
class DQNet_double(DQNet):
    def __init__(self, input_dim, h_sizes, output_dim):
        super().__init__(input_dim, h_sizes, output_dim)

        self.target = tf.keras.models.clone_model(self.model)
```

```
    def target_predict(self,state):
        return self.target(state)

    def target_update(self):
        self.target.set_weights(self.model.get_weights())

    def replay(self,memory,size,gamma=0.9):
        if len(memory) >= size:
            states = np.empty((0,4))
            targets = np.empty((0,2))

            #take a random set of memory of size=size
            batch = random.sample(memory,size)

            for state, action, next_state,reward,done in batch:
                states = np.vstack((states,state))

                q_values = self.predict(state).numpy()

                if done:
                    q_values[0,action] = reward
                else:
                    #the update values come from target network
                    q_values_next = self.target_predict(next_state).numpy()
                    q_values[0, action] = reward + gamma * np.max(q_values_next)

                targets = np.vstack((targets, q_values))

            self.update(states,targets)

#Function to train the learner
def TrainLearner(model,replay,double_net):

  final = []
  memory = []

  gamma = 0.9
  epsilon = 0.1
  eps_decay=0.99
  MaxEpisodes = 1000
  replay_size = 4 #size of the memory replay
  n_update = 10 #when to update target network

  for episode in range(MaxEpisodes):
    if double_net:
        # Update target network every n_update steps
        if episode % n_update == 0:
            model.target_update()

    # Reset state
    state = env.reset()
    done = False
    total = 0

    state = tf.expand_dims(state, 0)
```

```
    while not done:
        # env.render()

        # Implement greedy search policy
        if random.random() < epsilon:
            action = env.action_space.sample()
        else:
            q_values = model.predict(state).numpy()
            action = np.argmax(q_values[0,:])

        # Take action and add reward to total
        next_state, reward, done, _ = env.step(action)
        next_state = tf.expand_dims(next_state, 0)

        # Update total and memory
        total += reward
        memory.append((state.numpy(), action, next_state.numpy(), reward, done))

        q_values = model.predict(state).numpy()

        if done:
            if not replay:
                q_values[0,action] = reward
                # Update network weights
                model.update(state, q_values)
            break

        if replay:
            # Update network weights using replay memory
            model.replay(memory, replay_size, gamma)
        else:
            # Update network weights using the last step only
            q_values_next = model.predict(next_state).numpy()
            q_values[0,action] = reward + gamma * np.max(q_values_next)

            model.update(state, q_values)

        state = next_state

        # Update epsilon
        epsilon = max(epsilon * eps_decay, 0.01)
        # final.append(total)
        # plot_res(final, title)

    print("Episode: ",episode," reward: ",total)

#Run the training
replay = True
double_net = False

if not double_net:
    model = DQNet(4,[64, 64*2],2)
else:
    model = DQNet_double(4,[64, 64*2],2)

TrainLearner(model,replay,double_net)
```

Actor-Critic

Actor-critic methods are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function.

The actor learns the optimal action, and the critic learns the expected value function V by criticizing the action taken by the actor.

The actor and critic are two neural networks (with weights θ) that generally share the same input and hidden layers. Only the output layer is different.

The critic is trained on the TD error $r + \gamma v(s') - V(s)$. The value function can be computed as the return (total rewards) over the episode, that can be computed as:

$$R_t = \sum_{n}^{N} \gamma^t r_{t+n+1} \ , \tag{7}$$

where N is the number of time steps in the episode. The critic loss i then defined as:

$$L_{critic} = L_{\delta}(R_t, V_{\theta}^{\pi}) \ , \tag{8}$$

where L_{δ} is the Huber loss (generally preferred over mean squared error) and V_{θ}^{π} is the critic network output.

With regards to the actor, instead, the loss can be defined as:

$$L_{actor} = -\sum_{t=1}^{T} \log \pi_{\theta}(R_t - V_{\theta}^{\pi}) \ , \tag{9}$$

where π_{θ} is the expected output policy from the actor network. The term $(R_t - V_{\theta}^{\pi})$ is called *advantage* which indicates how much better an action is given a particular state over a random action selected according to the policy π for that state.

Let's consider the Cart-pole example. In this scenario the policy is given by choosing the cation to move left or right depending on their probabilities, obtained through a softmax function (which is the output of the actor network).

```
import gym
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

from tqdm import tqdm

seed = 42
gamma = 0.99 #discount factor past rewards
max_steps_per_episode = 1000

env = gym.make("CartPole-v0")

env.seed(seed)
eps = np.finfo(np.float32).eps.item()

num_inputs = 4 #The observations
num_actions = 2 #0,1
num_hidden = 128 #hidden layers

#Create Actor-Critic Network
class ActorCritic():
    def __init__(self,input_dim,num_hidden, output_size):
        super(ActorCritic,self).__init__()

        inputs = tf.keras.layers.Input(shape=(num_inputs,))
        hidden = tf.keras.layers.Dense(num_hidden, activation="relu")(inputs)
        action = tf.keras.layers.Dense(num_actions, activation="softmax")(hidden)
```

```
        critic = tf.keras.layers.Dense(1)(hidden)

        self.model = tf.keras.Model(inputs=inputs,outputs=[action,critic])

        self.optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)
        self.huber_loss = tf.keras.losses.Huber()

#Function to update network weights
    def update(self,history):
        actor_losses = []
        critic_losses = []
        for log_prob, value, ret in history:
            # At this point in history, the critic estimated that we would get a
            # total reward = `value` in the future. We took an action with log probability
            # of `log_prob` and ended up receiving a total reward = `ret`.
            # The actor must be updated so that it predicts an action that leads to
            # high rewards (compared to critic's estimate) with high probability.
            diff = ret - value
            actor_losses.append(-log_prob * diff)  # actor loss

            # The critic must be updated so that it predicts a better estimate of
            # the future rewards.
            critic_losses.append(
                self.huber_loss(tf.expand_dims(value, 0), tf.expand_dims(ret, 0))
            )

        # Backpropagation
        loss_value = sum(actor_losses) + sum(critic_losses)
        grads = tape.gradient(loss_value, self.model.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))

    def predict(self,state):
        return self.model(state)


#some initilizations
action_probs_history = []
critic_value_history = []
rewards_history = []
running_reward = 0
episode_count = 0

model = ActorCritic(num_inputs,num_hidden, num_actions)

#Train the learner
while True:  # Run until solved
    state = env.reset()
    episode_reward = 0
    with tf.GradientTape() as tape:
        for timestep,tqdm_iter in zip(range(1, max_steps_per_episode),tqdm(range(1,max_steps_per_episode-1))):
            #env.render(); #Adding this line would show the attempts
            # of the agent in a pop up window.

            state = tf.convert_to_tensor(state)
            state = tf.expand_dims(state, 0) #reshpes to (1,-1)

            # Predict action probabilities and estimated future rewards
```

```
        # from environment state
        action_probs, critic_value = model.predict(state)
        critic_value_history.append(critic_value[0, 0])

        # Sample action from action probability distribution
        # returns a single value for the actions. The possible actions are [0,1]
        #  given their probabilities, the function returns a random value weighted by their probability
        action = np.random.choice(num_actions, p=np.squeeze(action_probs))
        action_probs_history.append(tf.math.log(action_probs[0, action]))


        # Apply the sampled action in our environment
        state, reward, done, _ = env.step(action)
        rewards_history.append(reward)
        episode_reward += reward

        if done:
            break

    running_reward = episode_reward

    # Calculate expected value from rewards
    # - These are the labels for our critic
    returns = []
    discounted_sum = 0
    for r in rewards_history[::-1]: #loop list of rewards from end (mst recent reward)
        discounted_sum = r + gamma * discounted_sum
        returns.insert(0, discounted_sum)

    # Normalize
    returns = np.array(returns)
    returns = (returns - np.mean(returns)) / (np.std(returns) + eps)
    returns = returns.tolist()

    # Calculating loss values to update our network
    history = zip(action_probs_history, critic_value_history, returns)
    model.update(history)

    # Clear the loss and reward history
    action_probs_history.clear()
    critic_value_history.clear()
    rewards_history.clear()

    # Log details
    episode_count += 1
    if episode_count % 10 == 0:
        template = "running reward: {:.2f} at episode {}"
        print(template.format(running_reward, episode_count))

    if running_reward > 195:  # Condition to consider the task solved
        print("Solved at episode {}!".format(episode_count))
        print("running reward!", running_reward)

        break
```

Policy Gradient

DQN and Actor-critic method allow finding the best actions, given a certain policy (whether epsilon-greedy or specified as softmax). **Policy Gradient** (PG) methods aim at directly identifying the optimal policy.

In PG the policy is directly parametrized by some parameters as $\pi(a|s;\theta)$ and neural networks can be used for learning those parameters. PG methods thus allow solving problems with continuous action spaces (unlike DQN and Actor-critic). The policy model's parameters are updated in such a way to receive only a positive reward. Generally, the input of the policy network is the system's state and the output is each action's probability for that state. Given the action probability, the action itself is obtained by sampling from that distribution. The policy model is then updated in such a way to ensure that policies leading to good rewards will have higher probability than those with ow reward.

```
import gym
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_probability as tfp

#Define your policy network
#The input is the environment's state
class PolicyNet():
    def __init__(self,input_dim,hidden_sizes,out_dim=1,discrete=False):

        inputs = tf.keras.layers.Input(shape=input_dim)  # input dimension
        x = inputs
        for n in hidden_sizes:
            x = tf.keras.layers.Dense(n, activation="relu")(x)

#if the action space is not discrete
#output mean and variance of prob distribution
        if not discrete:
            mu = tf.keras.layers.Dense(out_dim, activation="linear")(x)
            sigma = tf.keras.layers.Dense(out_dim, activation="softplus")(x)

            self.model = tf.keras.Model(inputs=inputs, outputs=[mu, sigma])

        else:
#if discrete, the probability of each action is given by softmax
            prob = tf.keras.layers.Dense(out_dim, activation="softmax")(x)
            self.model = tf.keras.Model(inputs=inputs, outputs=prob)


        self.optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

#function to update model parameters
    def update(self,state_history,action_history,reward_history,
               discrete,gamma):

        discounted_rewards = []
        sum_rewards = 0

        for reward in reward_history[::-1]:
            sum_rewards = reward + gamma * sum_rewards
            discounted_rewards.append(sum_rewards)

        discounted_rewards.reverse()
        history = zip(state_hist, action_hist, discounted_rewards)
```

```
        for state,action,reward in history:
            with tf.GradientTape() as tape:

                state = tf.expand_dims(state, 0)

#get action prob distribution
                if not discrete:
                    mu, sigma = self.model(state)
                    # get gaussian distribution
                    distr = tfp.distributions.Normal(mu, sigma)

                else:
                    prob = self.model(state)
                    distr = tfp.distributions.Categorical(probs=prob, dtype=tf.float32)

                # get probability of perfomed action, given the learnt distribution
                log_prob = distr.log_prob(action)
                loss_value = -abs(reward)*log_prob

                grads = tape.gradient(loss_value, self.model.trainable_variables)
                self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))

#Initialize environment
# env = gym.make("Pendulum-v1")
# state_dim = 3 #cos angle, sin angle, angular vel
# out_dim = 1
# discrete=False

env = gym.make("CartPole-v0")
discrete=True
state_dim = 4 #cart posiiton,veolicty, pole position, velocity
out_dim = 2

#Instatntiate policy network
hidden_sizes = [64,64]
PNet = PolicyNet(state_dim,hidden_sizes,out_dim=out_dim,discrete=discrete)

MaxEpisodes = 10000
gamma = 0.95

#Run training
for episode in range(MaxEpisodes):
    state = env.reset()
    done = False
    r = 0
    steps = 0

    reward_hist = []
    state_hist = []
    action_hist = []

    while not done:
        env.render()

        curr_state = state
        state = tf.expand_dims(state, 0)
```

```
#sample action from probability ditribution
        if not discrete:
            mu, sigma = PNet.model(state)
            action = np.random.normal(mu[0].numpy(), sigma[0].numpy(), 1)
            action = np.clip(action,-2,2)

        else:
            probs = PNet.model(state).numpy()
            action = np.random.choice(a=[0, 1],size = 1,p = probs[0,:])
            action = action[0]

        action_hist.append(action)
        state_hist.append(curr_state)

        state, reward, done, _ = env.step(action)

        reward_hist.append(reward)

        r = r+reward
        steps = steps+1

    PNet.update(state_hist,action_hist,reward_hist, discrete,gamma)
    print("eisode:", episode," reward:", r)
```

1) Deep Deterministic Policy Gradient: **Deep Deterministic Policy Gradient** (DDPG) leverages the benefits of DQN and PG to better improve learning performance.

It uses a structure like Actor-critic, with an actor network to propose a possible action, and the critic to predict the goodness of that action. Just like Actor-critic, a replay memory and a target (auxiliary) network can be used to improve the learning process. In this scenario, the critic directly learns the state-action function $Q(s, a)$.

The Actor network takes as input the current system's state and outputs a deterministic action $\alpha(s, \theta)$.

The Critic network takes both the system's state and actions as inputs. Two parallel networks can be used and concatenated, one taking only the states as input and one only the actions. The output of the Critic is the predicted state-action value function $\hat{Q}(s, a, \theta) \in \mathbb{R}$.

A target network consisting (with parameters θ') both of an actor and critic is used to improve the learning. The outputs of the target network are $\alpha'(s, \theta')$ and $\hat{Q}'(s, a, \theta')$.

The Actor weight are updated such to maximize the Critic's output:

$$L_{actor} = -\hat{Q}(s, \alpha(s, \theta), \theta) \ , \tag{10}$$

where $\alpha(s, \theta)$ is the Actor's predicted action.

The Critic weights are instead updated by using the target network as:

$$\begin{aligned} &L_{critic} = ||y - \hat{Q}(s, a, \theta)||^2 \\ &\text{where} \quad y = r + \gamma \hat{Q}'(s', \alpha'(s', \theta'), \theta') \ , \end{aligned} \tag{11}$$

where r is the current reward, s' is the next system's state, α' the target actor network's output, and $\hat{Q}'$ the target critic network's output.

The target network's weights are then updated in a soft manner as:

$$\theta' = \tau\theta + (1 - \tau)\theta' \ , \tag{12}$$

with τ being an updating parameter.

```
import gym
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt
```

```
problem = "Pendulum-v1"
env = gym.make(problem)

num_states = env.observation_space.shape[0]
print("Size of State Space ->  {}".format(num_states))
num_actions = env.action_space.shape[0]
print("Size of Action Space ->  {}".format(num_actions))

upper_bound = env.action_space.high[0]
lower_bound = env.action_space.low[0]

#Buffer class for replay memory
class Buffer:
    def __init__(self, buffer_capacity=100000, batch_size=64):
        # Number of "experiences" to store at max
        self.buffer_capacity = buffer_capacity
        # Num of tuples to train on.
        self.batch_size = batch_size

        # Its tells us num of times record() was called.
        self.buffer_counter = 0

        # Instead of list of tuples as the exp.replay concept go
        # We use different np.arrays for each tuple element
        self.state_buffer = np.zeros((self.buffer_capacity, num_states))
        self.action_buffer = np.zeros((self.buffer_capacity, num_actions))
        self.reward_buffer = np.zeros((self.buffer_capacity, 1))
        self.next_state_buffer = np.zeros((self.buffer_capacity, num_states))

    # Takes (s,a,r,s') obervation tuple as input
    def record(self, obs_tuple):
        # Set index to zero if buffer_capacity is exceeded,
        # replacing old records
        index = self.buffer_counter % self.buffer_capacity

        self.state_buffer[index] = obs_tuple[0]
        self.action_buffer[index] = obs_tuple[1]
        self.reward_buffer[index] = obs_tuple[2]
        self.next_state_buffer[index] = obs_tuple[3]

        self.buffer_counter += 1

#Actor-critic class with target network
class ActorCritic():
    def __init__(self,num_states,num_actions,hidden_actor=[256,256]):

        self.actor = self.getActor(num_states,hidden_actor)
        self.critic = self.getCritic(num_states, num_actions)

        self.target_actor = self.getActor(num_states,hidden_actor)
        self.target_critic = self.getCritic(num_states, num_actions)

        self.setTargetWeights()

        self.critic_loss = tf.keras.losses.MeanSquaredError()
```

```
        critic_lr = 0.002
        actor_lr = 0.001

        self.critic_optimizer = tf.keras.optimizers.Adam(critic_lr)
        self.actor_optimizer = tf.keras.optimizers.Adam(actor_lr)

    def setTargetWeights(self):
        self.target_actor.set_weights(self.actor.get_weights())
        self.target_critic.set_weights(self.critic.get_weights())

    def getActor(self,num_states,hidden_sizes=[256,256]):
        inputs = layers.Input(shape=(num_states,))

        x = inputs
        for n in hidden_sizes:
            x = layers.Dense(n, activation="relu")(x)

        outputs = layers.Dense(1, activation="tanh")(x)
        # Our upper bound is 2.0 for Pendulum.
        outputs = outputs * upper_bound
        model = tf.keras.Model(inputs, outputs)

        return model

    def getCritic(self,num_states,num_actions):

        # State as input
        state_input = layers.Input(shape=(num_states))
        state_out = layers.Dense(16, activation="relu")(state_input)
        state_out = layers.Dense(32, activation="relu")(state_out)

        # Action as input
        action_input = layers.Input(shape=(num_actions))
        action_out = layers.Dense(32, activation="relu")(action_input)

        # Both are passed through seperate layer before concatenating
        concat = layers.Concatenate()([state_out, action_out])

        out = layers.Dense(256, activation="relu")(concat)
        out = layers.Dense(256, activation="relu")(out)
        outputs = layers.Dense(1)(out)

        # Outputs single value for give state-action
        model = tf.keras.Model([state_input, action_input], outputs)

        return model

    @tf.function
    def update_target(self, tau):
        #update weights of target actor
        for (a, b) in zip(self.target_actor.variables, self.actor.variables):
            a.assign(b * tau + a * (1 - tau)) #assign value to reference
        #update weights of critic
        for (a, b) in zip(self.target_critic.variables, self.critic.variables):
            a.assign(b * tau + a * (1 - tau)) #assign value to reference

    def GetActionPolicy(self,state):
```

```
        #it is a deteministic policy
        action = self.actor(state)
        action = action[0].numpy()

        # We make sure action is within bounds
        legal_action = np.clip(action, lower_bound, upper_bound)

        return legal_action

    @tf.function
    def update(
        self, state_batch, action_batch, reward_batch, next_state_batch,
    ):
        #update critic
        with tf.GradientTape() as tape:
            target_actions = self.target_actor(next_state_batch, training=True)
            y = reward_batch + gamma * self.target_critic(
                [next_state_batch, target_actions], training=True
            )
            critic_value = self.critic([state_batch, action_batch], training=True)
            critic_loss = self.critic_loss(y,critic_value)

        critic_grad = tape.gradient(critic_loss, self.critic.trainable_variables)
        self.critic_optimizer.apply_gradients(
            zip(critic_grad, self.critic.trainable_variables)
        )

        #update actor
        with tf.GradientTape() as tape:
            actions = self.actor(state_batch, training=True)
            critic_value = self.critic([state_batch, actions], training=True)
            # Used `-value` as we want to maximize the value given
            # by the critic for our actions
            actor_loss = -tf.math.reduce_mean(critic_value)

        actor_grad = tape.gradient(actor_loss, self.actor.trainable_variables)
        self.actor_optimizer.apply_gradients(
            zip(actor_grad, self.actor.trainable_variables)
        )

    # We compute the loss and update parameters
    def learn(self,buffer):
        # Get sampling range
        record_range = min(buffer.buffer_counter, buffer.buffer_capacity)
        # Randomly sample indices
        batch_indices = np.random.choice(record_range, buffer.batch_size)

        # Convert to tensors
        state_batch = tf.convert_to_tensor(buffer.state_buffer[batch_indices])
        action_batch = tf.convert_to_tensor(buffer.action_buffer[batch_indices])
        reward_batch = tf.convert_to_tensor(buffer.reward_buffer[batch_indices])
        reward_batch = tf.cast(reward_batch, dtype=tf.float32)
        next_state_batch = tf.convert_to_tensor(buffer.next_state_buffer[batch_indices])

        self.update(state_batch, action_batch, reward_batch, next_state_batch)

#——Initialize Actor-Critic network
```

```
ACNet = ActorCritic(num_states,num_actions,hidden_actor=[256,256])

total_episodes = 100
# Discount factor for future rewards
gamma = 0.99
# Used to update target networks
tau = 0.005

buffer = Buffer(50000, 64)

# To store reward history of each episode
ep_reward_list = []
# To store average reward history of last few episodes
avg_reward_list = []

# Takes about 4 min to train
for ep in range(total_episodes):

    prev_state = env.reset()
    episodic_reward = 0

    while True:
        # Uncomment this to see the Actor in action
        # But not in a python notebook.
        env.render()

        tf_prev_state = tf.expand_dims(tf.convert_to_tensor(prev_state), 0)

        action = ACNet.GetActionPolicy(tf_prev_state)
        # Receive state and reward from environment.
        state, reward, done, info = env.step(action)

        buffer.record((prev_state, action, reward, state))
        episodic_reward += reward

        ACNet.learn(buffer)
        ACNet.update_target(tau)

        # End this episode when `done` is True
        if done:
            break

        prev_state = state

    ep_reward_list.append(episodic_reward)

    # Mean of last 40 episodes
    avg_reward = np.mean(ep_reward_list)
    print("Episode * {} * Avg Reward is ==> {}".format(ep, avg_reward))
    avg_reward_list.append(avg_reward)

# Plotting graph
# Episodes versus Avg. Rewards
plt.plot(avg_reward_list)
plt.xlabel("Episode")
plt.ylabel("Avg. Epsiodic Reward")
plt.show()
```

2) Proximal Policy Optimization: **Proximal Policy Optimization** (PPO) is a PG method that can be used both with continuous and discrete actions.

It still uses and Actor-Critic method, but the policy is stochastic. The Actor predicts a possible action, and the Critic predicts the possible reward (thus learning the state value function $V(s)$).

A set of trajectories by running the actor's policy is first collected, computing the rewards and the $advantage$. The $advantage$ can be computed as the TD error $r_t + \gamma V(s') - V(s)$. The actor's model is then updated by using the **PPO-Clip** objective, and the Critic's model updated by minimizing the advantage.

In the Cart-pole example, the Actor takes as inputs the system's current state and outputs the logits (or un-normalized log probabilities) of the possible actions. Since the action space is discrete, the output is a two dimensional vector. If the action space was continuous, the output of the Actor could be the mean and variance of a Gaussian distribution, from which the action would be sampled.

The critic takes as input the system's current state and outputs a single-valued function $V(s)$.

A buffer is used to store the system's states, actions, rewards, predicted value function, and log probabilities of the actions. Once the trajectory is over (for instance because of the pole falling or because of the maximum number of iterations reached), the buffer is used to compute the advantages for each time step and the discounted cumulative rewards.

At the end of the trajectory, the values stored in the buffer are used to update the policy of the Actor and the model of the critic.

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import gym
import scipy.signal
import time

#----------------#
# Actor-Critic Network model
class ActorCritic():
    def __init__(self,num_states,hidden_sizes,num_actions):

        self.actor = self.GetNet(num_states,hidden_sizes,num_actions)
        self.critic = self.GetNet(num_states, hidden_sizes, 1)

        self.critic_loss = tf.keras.losses.MeanSquaredError()

        policy_learning_rate = 3e-4
        value_function_learning_rate = 1e-3

        # Initialize the policy and the value function optimizers
        self.policy_optimizer = keras.optimizers.Adam(learning_rate=policy_learning_rate)
        self.value_optimizer = keras.optimizers.Adam(learning_rate=value_function_learning_rate)

    def GetNet(self,input_dim,hidden_sizes,output_dim):
        input = keras.Input(shape=input_dim, dtype=tf.float32)
        x = input
        for size in hidden_sizes:
            x = layers.Dense(units=size, activation=tf.tanh)(x)
        output = layers.Dense(units=output_dim)(x)
        return keras.Model(inputs=input, outputs=output)

    # Train the value function by regression on mean-squared error
    @tf.function
    def train_value_function(self,observation_buffer, return_buffer):
```

```
        with tf.GradientTape() as tape:  # Record operations for automatic differentiation.
            value_loss = tf.reduce_mean((return_buffer - self.critic(observation_buffer)) ** 2)
        value_grads = tape.gradient(value_loss, self.critic.trainable_variables)
        self.value_optimizer.apply_gradients(zip(value_grads, self.critic.trainable_variables))

    # Train the policy by maxizing the PPO-Clip objective
    @tf.function
    def train_policy(self,
            observation_buffer, action_buffer, logprobability_buffer, advantage_buffer,clip_ratio
    ):
        with tf.GradientTape() as tape:  # Record operations for automatic differentiation.
            ratio = tf.exp(
                logprobabilities(self.actor(observation_buffer), action_buffer)
                - logprobability_buffer
            )
            min_advantage = tf.where(
                advantage_buffer > 0,
                (1 + clip_ratio) * advantage_buffer,
                (1 - clip_ratio) * advantage_buffer,
            )

            policy_loss = -tf.reduce_mean(
                tf.minimum(ratio * advantage_buffer, min_advantage)
            )
        policy_grads = tape.gradient(policy_loss, self.actor.trainable_variables)
        self.policy_optimizer.apply_gradients(zip(policy_grads, self.actor.trainable_variables))

        kl = tf.reduce_mean(
            logprobability_buffer
            - logprobabilities(self.actor(observation_buffer), action_buffer)
        )
        kl = tf.reduce_sum(kl)
        return kl

    # Sample action from actor
    @tf.function
    def sample_action(self,observation):
        logits = self.actor(observation)
        # samples = tf.random.categorical(tf.math.log([[0.5, 0.5]]), 5)
        action = tf.squeeze(tf.random.categorical(logits, 1), axis=1)
        return logits, action

    def logprobabilities(self,logits, a):
        # Compute the log-probabilities of taking actions a by using the logits (i.e. the output of the actor)
        logprobabilities_all = tf.nn.log_softmax(logits)
        act = a[0].numpy()
        return logprobabilities_all[0, act]

#---- Buffer Class
class Buffer:
    # Buffer for storing trajectories
    def __init__(self, observation_dimensions, size, gamma=0.99, lam=0.95):
        # Buffer initialization
        self.observation_buffer = np.zeros(
            (size, observation_dimensions), dtype=np.float32
        )
        self.action_buffer = np.zeros(size, dtype=np.int32)
```

```
        self.advantage_buffer = np.zeros(size, dtype=np.float32)
        self.reward_buffer = np.zeros(size, dtype=np.float32)
        self.return_buffer = np.zeros(size, dtype=np.float32)
        self.value_buffer = np.zeros(size, dtype=np.float32)
        self.logprobability_buffer = np.zeros(size, dtype=np.float32)
        self.gamma, self.lam = gamma, lam
        self.pointer, self.trajectory_start_index = 0, 0

    def store(self, observation, action, reward, value, logprobability):
        # Append one step of agent-environment interaction
        self.observation_buffer[self.pointer] = observation
        self.action_buffer[self.pointer] = action
        self.reward_buffer[self.pointer] = reward
        self.value_buffer[self.pointer] = value
        self.logprobability_buffer[self.pointer] = logprobability
        self.pointer += 1

    def finish_trajectory(self, last_value=0):
        # Finish the trajectory by computing advantage estimates and rewards-to-go
        path_slice = slice(self.trajectory_start_index, self.pointer)
        rewards = np.append(self.reward_buffer[path_slice], last_value)
        values = np.append(self.value_buffer[path_slice], last_value)

        deltas = rewards[:-1] + self.gamma * values[1:] - values[:-1]

        self.advantage_buffer[path_slice] = discounted_cumulative_sums(
            deltas, self.gamma * self.lam
        )
        self.return_buffer[path_slice] = discounted_cumulative_sums(
            rewards, self.gamma
        )[:-1]

        self.trajectory_start_index = self.pointer

    def get(self):
        # Get all data of the buffer and normalize the advantages
        self.pointer, self.trajectory_start_index = 0, 0
        advantage_mean, advantage_std = (
            np.mean(self.advantage_buffer),
            np.std(self.advantage_buffer),
        )
        self.advantage_buffer = (self.advantage_buffer - advantage_mean) / advantage_std
        return (
            self.observation_buffer,
            self.action_buffer,
            self.advantage_buffer,
            self.return_buffer,
            self.logprobability_buffer,
        )

def discounted_cumulative_sums(x, discount):
    # Discounted cumulative sums of vectors for computing rewards-to-go and advantage estimates
    discounted_rewards = []
    sum_rewards = 0
    for reward in x[::-1]:
        sum_rewards = reward + discount * sum_rewards
        discounted_rewards.append(sum_rewards)
```

```
    discounted_rewards.reverse()
    return np.array(discounted_rewards)

#-------------------------------------------------------------------#
# Hyperparameters of the PPO algorithm
steps_per_epoch = 4000
epochs = 30
gamma = 0.99
clip_ratio = 0.2
policy_learning_rate = 3e-4
value_function_learning_rate = 1e-3
train_policy_iterations = 80
train_value_iterations = 80
lam = 0.97
target_kl = 0.01
hidden_sizes = (64, 64)

# Initialize the environment and get the dimensionality of the
# observation space and the number of possible actions
env = gym.make("CartPole-v0")
observation_dimensions = env.observation_space.shape[0]
num_actions = env.action_space.n

# Initialize the buffer
buffer = Buffer(observation_dimensions, steps_per_epoch)

# Initialize the actor and the critic as keras models
ACNet = ActorCritic(observation_dimensions,hidden_sizes,num_actions)

# Initialize the observation, episode return and episode length
state, episode_return, episode_length = env.reset(), 0, 0

for epoch in range(epochs):
    # Initialize the sum of the returns, lengths and number of episodes for each epoch
    sum_return = 0
    sum_length = 0
    num_episodes = 0

    # Iterate over the steps of each epoch
    for t in range(steps_per_epoch):
        env.render()

        # Get the logits, action, and take one step in the environment
        state = state.reshape(1, -1)
        logits, action = ACNet.sample_action(state)
        state_new, reward, done, _ = env.step(action[0].numpy())
        episode_return += reward
        episode_length += 1

        # Get the value and log-probability of the action
        value_t = ACNet.critic(state)
        logprobability_t = logprobabilities(logits, action)

        # Store obs, act, rew, v_t, logp_pi_t
        buffer.store(state, action, reward, value_t, logprobability_t)
```

```
        # Update the observation
        state = state_new

        # Finish trajectory if reached to a terminal state
        if done or (t == steps_per_epoch - 1):
            last_value = 0 if done else ACNet.critic(state.reshape(1, -1))
            buffer.finish_trajectory(last_value)
            sum_return += episode_return
            sum_length += episode_length
            num_episodes += 1
            state, episode_return, episode_length = env.reset(), 0, 0


    # Get values from the buffer after whole trajectories
    (
        state_buffer,
        action_buffer,
        advantage_buffer,
        return_buffer,
        logprobability_buffer,
    ) = buffer.get()

    # Update the policy and implement early stopping using KL divergence
    for _ in range(train_policy_iterations):
        kl = ACNet.train_policy(
            state_buffer, action_buffer, logprobability_buffer, advantage_buffer,clip_ratio
        )
        if kl > 1.5 * target_kl:
            # Early Stopping
            break

    # Update the value function
    for _ in range(train_value_iterations):
        ACNet.train_value_function(state_buffer, return_buffer)

    # Print mean return and length for each epoch
    print(
        f" Epoch: {epoch + 1}. Mean Return: {sum_return / num_episodes}. Mean Length: {sum_length / num_episodes}"
    )
```