Interpretability in Reinforcement Learning: A Report

Sarthak Das (2018/UG/026)
School of Mathematical and Computational Sciences, IACS
Submitted to : Prof. Rajarshi Ray

January 10, 2022

Contents

1 Introduction

1.1 Interpretable Machine Learning (IML)

This report is a literature and tools survey for the interpretability of reinforcement learning algorithms. Therefore it is important to understand at first what **Interpretable Machine Learning (IML)** means. Interpretability may be defined non-mathematically as the degree to which a human can understand the cause of a decision. Therefore, IML can be described as an attempt to understand the behaviour of machine learning algorithms and the rationale behind why a model in question makes a particular prediction for the given input. The following are some questions IML is concerned with [1]:

- *How does the algorithm create the model?*
- *How does the trained model make predictions?*
- *How do parts of the model affect predictions?*
- *Why did the model make a certain prediction for an instance?*
- *Why did the model make specific predictions for a group of instances?*

In this context, a question might arise that if a model performs well, why should there be a need to understand why the model makes a certain prediction? It would of course be easier to simply trust the model. There are many reasons why interpretability is useful in certain cases:

1. When a machine's decisions affect a person's life deeply (which is increasingly becoming the case in this age) it is important to understand the basis for why a machine behaves a certain way. This helps build a sense of **privacy** and **trust**.
2. Machine learning is often used in safety critical systems, and as such the decisions made by an algorithm demand explanation. This helps ensure that the algorithm is **reliable**, **robust** and **fair**.

Interpretability could be particularly valuable in **Reinforcement Learning (RL)**. Interpretability might help to reduce the RL search space and make RL easier to troubleshoot and use (if we understand the choices and intent of the RL system, we can remove actions that might lead to harm) [2]. However it is also to be noted that when the problem in question is well studied, or when interpretability would enable people to manipulate the system, IML is not preferred [1].

Techniques for IML can be categorized based on certain factors [1]:

1. **Point of application.** *Intrinsic* interpretability refers to machine learning models that are considered interpretable due to their simple structure, such as short decision trees or sparse linear models. *Post hoc* interpretability refers to the application of interpretation methods after model training. Post hoc methods can also be applied to intrinsically interpretable models.

2. **Scope of interpretability.** As mentioned previously, an interpretation method may explain an individual prediction (*local*) or the entire model behaviour (*global*).

3. **Specificity of model.** *Model-specific* interpretation tools are limited to specific model classes. For example, tools that only work for the interpretation of reinforcement learning agents are model-specific. *Model-agnostic* tools can be used on any machine learning model and are applied after the model has been trained (post hoc). These agnostic methods usually work by analyzing feature input and output pairs and do not have access to model internals.

An example of a model-agnostic interpretation tool is **AI Explainability 360** (also known as AIX360) which is an open-source Python library developed by IBM Research that supports the interpretability and explainability of datasets and machine learning models [3].

AIX360 supports various model-agnostic IML techniques such as Boolean Decision Rules via Column Generation (*BRCG*), Generalized Linear Rule Models (*GLRM*), *ProtoDash*, *ProfWeight*, Teaching Explanations for Decisions (*TED*), Contrastive Explanations Method (*CEM*), Contrastive Explanations Method with Monotonic Attribute Functions (*CEM-MAF*), Disentangled Inferred Prior Variational Autoencoder (*DIP-VAE*), and Local Interpretable Model-Agnostic Explanations (*LIME*) [4].

1.2 Reinforcement Learning (RL)

1.2.1 Introduction

Before proceeding further, it is also important to understand what **Reinforcement Learning (RL)** means. Reinforcement learning (RL) is a paradigm of machine learning concerned with how intelligent agents ought to take actions in an uncertain, potentially complex environment in order to maximize the notion of cumulative reward. RL differs from supervised learning in not needing labelled input/output pairs be presented, and in not needing sub-optimal actions to be explicitly corrected. Instead the focus is on finding a balance between *exploration* (of uncharted territory) and *exploitation* (of current knowledge). Thus, reinforcement learning is particularly well-suited to problems that include a long-term versus short-term reward trade-off, such as robot automation, elevator scheduling, as well as games such as backgammon, checkers, chess, Go and Ms.

Pac-Man [5].

In RL, an **intelligent agent** faces a game-like situation in an **environment** modelled after the problem statement. The agent interacts with this environment, employing trial and error to come up with an **optimal policy** on its own, without human interference. To push it in the right direction, the designer sets a **reward policy** based on which the agent is either rewarded or penalized for the actions it performs. Its goal is to maximize the cumulative reward. **RL algorithms** employed by the agents to achieve this goal are classified into two categories [6]:

1. **Model-based RL** uses experience to construct an internal model of the transitions and immediate outcomes in the environment. Appropriate actions are then chosen by searching or planning in this world model. Examples include *policy optimization* and *Q-learning*.

2. **Model-free RL**, on the other hand, uses experience to learn directly but without estimation or use of a world model. Model-free methods are statistically less efficient because information from the environment is combined with previous (possibly erroneous) estimates about state values, rather than being used directly. Examples include Imagination-Augmented Agents (*I2A*), Model-Based Priors for Model-Free Reinforcement Learning (*MBMF*), and Model-Based Value Expansion (*MBVE*).

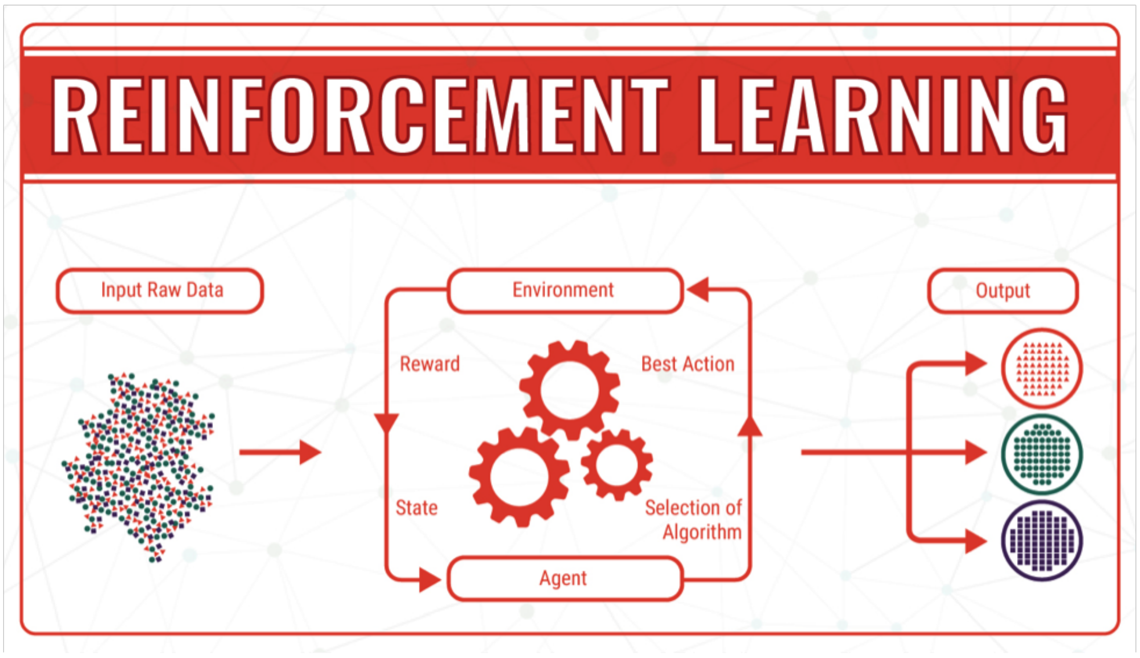


Figure 1.1: A schematic representation of RL.

1.2.2 RL Terminology

This subsection has been adapted from an article published in Towards Data Science [7].

- **Reinforcement Learning (RL).** One of the main areas of machine learning, concerned with the learning process of an arbitrary being, formally known as an *agent*, in the world surrounding it, known as the *environment*. The agent seeks to maximize the *rewards* it receives from the environment, and performs different actions in order to learn how the environment responds and gain more rewards.

- **Agent.** The learning and acting part of an RL problem, which tries to maximize the rewards it is given by the environment.

- **Environment.** Everything the agent can interact with, directly or indirectly. The environment changes as the agent performs actions; every such change is considered a *state transition.*

- **State.** Every scenario the agent encounters in the Environment is formally called a state. The agent transitions between different states by performing actions. There are no possible states after a terminal state has been reached, following which a new episode begins.

- **Actions.** Agent's methods which allow it to interact and change its environment, thus transition between states. Every action performed by the agent yields a reward from the environment. The decision of which action to choose is made by the policy.

- **Reward.** A numerical value received by the agent from the environment as a direct response to the agent's actions. All actions yield reward. Positive rewards emphasize a desired action, negative rewards emphasize an action the agent should stray away from. Zero rewards imply that the agent did not do anything significant enough to be rewarded or penalized. For a given environment, the action-reward mapping is defined by a *reward policy.*

- **Policy.** A mapping π from some state $s \in S$ to the probabilities of selecting each possible action $a \in A$, given s. An *optimal policy* π^* is a policy which seeks to maximize the cumulative reward of the agent over an episode.

- **On-Policy and Off-Policy.** Every RL algorithm must follow some policy in order to decide which actions to perform at each state. Algorithms which concern about the policy which yielded past state-action decisions are referred to as on-policy algorithms, while those ignoring it are known as off-policy.

- **Episodes.** All states that come in between an initial state and a terminal state. The agent's goal it to maximize the cumulative reward it receives during an episode. In situations where there is no terminal state, we assume an infinite episode. Two different episodes are completely independent of each other.

- **Expected Return.** The expected reward over an entire episode. Also known as *cumulative reward.*
- **Episodic Tasks.** RL tasks which are made of multiple episodes, each episode having a terminal state.
- **Continuous Tasks.** RL tasks which are not made of episodes, but rather last forever. Such tasks have no terminal states. For simplicity, they are usually assumed to be made of one never-ending episode.
- **Deep Reinforcement Learning (DRL).** The use of an RL algorithm with a deep neural network as an approximator for the learning part. This is usually done in order to cope with problems where the number of possible states and actions scales fast, and an exact solution in no longer feasible.
- **Discount Factor.** A factor γ multiplying the future expected reward, whose value lies in the range of [0,1]. The lower the discount factor is, the less important future rewards are, and the agent will tend to focus on actions which will yield immediate rewards only.
- **Future Discounted Reward.** In many RL problems, there are valid reasons to discount future rewards. One of them is the actual impact that decisions have on long term performance. Given a discount factor γ, immediate reward received after performing action a in state s r_0, and future reward after n steps r_n, the future discounted reward is calculated as:

$$r = \Sigma_{n=0}^{\infty} \gamma^n r_n$$

- **Q-value.** A measure of the future expected reward assuming the agent is in state $s \in S$ and performs action $a \in A$, and then continues playing until the end of the episode following some policy π. It is mathematically defined as:

$$Q(s,a) = \mathbf{E}[\Sigma_{n=0}^{N} \gamma^n r_n]$$

  where N is the number of states from state s up to the terminal state, γ is the discount factor, r_0 is the immediate reward received after performing action a in state s, and r_n is the future reward after n steps.
- **Q-learning.** An off-policy model-based RL algorithm that uses a table to store all Q-values of all state-action pairs possible. The table is updated using the *Bellman equation for Q-values*, while action selection is made using an *exploration policy.*
- **Bellman equation for Q-values.** Given a set of states $s \in S$, set of actions $a \in A$, and an immediate reward for executing an action a in a state s, the expected value (*Q-value*) of performing an action in a state can be learned over

time by experimenting with actions in states, observing the reward, and updating the Q-value estimate via the equation:

$$Q_{t+1}(s, a) \leftarrow (1-\alpha)Q_t(s, a) + \alpha(r + \gamma.max_{a'}Q_t(s', a'))$$

where Q_t is the current Q-value estimate for a state-action pair, s and a are the current state and chosen action, α is the learning rate, r is the immediate reward for choosing action a when in state s (as experienced immediately by interacting with the environment), s' is the next state the agent is in after executing a, and a' is the best possible next action, such that the maximum expected future reward from being in state s' is added to r after being discounted by the factor γ.

The Bellman equation can be solved by backwards induction, either analytically in a few special cases, or numerically on a computer. An exact solution is infeasible when there are many state variables. In such cases, an approximate solution may be obtained by approximate dynamic programming with the use of artificial neural networks.

1.2.3 An Example of Q-Learning

This example was taken from Prof. Pallab Dasgupta's lecture on RL (CS60045, IIT Kharagpur). Consider the RL environment in Figure 1.2. There are six possible states, one of which is the terminal state marked as G. The RL agent begins from a random state and its goal is to reach G, while maximizing the cumulative reward. The possible actions are marked by arrows. Given, $\gamma = 0.9$.

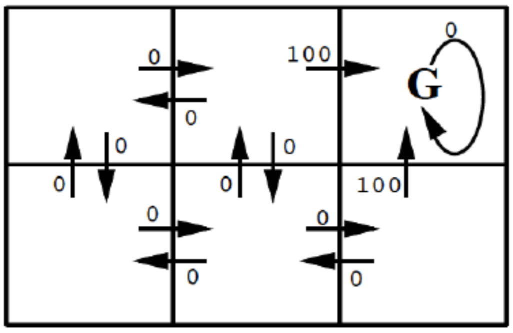


Figure 1.2: The RL environment. G is the terminal state, immediate rewards for each transition is shown.

For every state, the Q-value can be calculated. For example, if we consider the bottom left square, the optimal path to G is defined by the sequence of actions up, right, right. Therefore, the Q-value can be calculated as:

$$Q = 0 + 0.9 * 0 + 0.9^2 * 100 = 81$$

Consider Figure 1.3, where the Q-value for each state has been computed likewise.

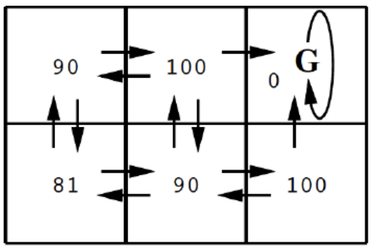
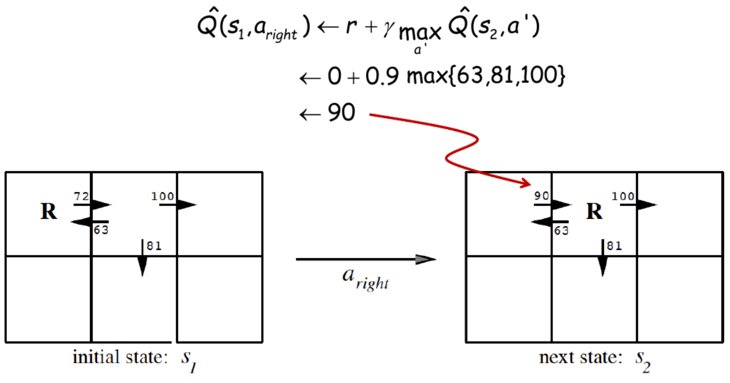


Figure 1.3: The RL environment. Q-value for each state is shown.

Figure 1.4 shows one step in the Q-learning process. The RL agent moves from one state to another, and the Q-estimate is updated using the given Q-values and the Bellman equation. This step is repeated until R reaches the terminal state G, upon which the episode is terminated.



$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$
$\leftarrow 0 + 0.9 \max\{63, 81, 100\}$
$\leftarrow 90$
R
72
63
100
81
a_{right}
90
63
100
81
initial state: s_1
next state: s_2

Figure 1.4: One step of Q-learning.

At the end of an episode, we have an optimal policy which describes the optimal action to be taken, when in a given state, in order to maximize the cumulative reward at the end of the episode. This is shown in Figure 1.5.

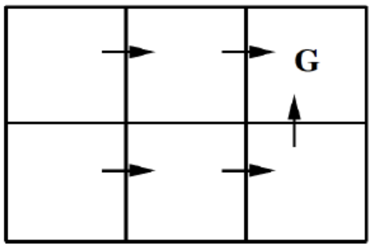


Figure 1.5: An optimal policy, as obtained by Q-learning. The optimal action for each state is marked by arrows.

1.2.4 Deep Q-Network (DQN)

Deep Q-Network (DQN) is an off-policy, value-based, model-free DRL algorithm, that learns to act in discrete action space. DQN was first introduced in 2013, but the so-called "Nature-variant" published in 2015 by Volodymyr Mnih et al. is the most used. DQN is a *Q-learning algorithm.*

The strategy for Q-learning is to transform the problem of reinforcement learning into that of supervised learning. Although there is no labelled data, the recursive nature of the Bellman equation is useful. The right hand side of the Bellman equation is used as the true label, while the left hand side is used as the predicted label. Similar to any supervised learning approach then, the aim is to bring predicted label as close as possible to the true label. DQN makes use of an artificial neural network to perform Q-learning. This neural network is called the *Q-network.* The "Nature-variant" is noted for introducing two key components to DQN [8]:

1. **Replay Memory.** Each interaction that the agent has with the environment is stored in a buffer in the form of a 5-tuple of ⟨state, action, next_state, reward, done⟩. Therefore, at each time step, a mini-batch of experiences are sampled uniformly at random for computing the gradient updates to allow the neural network to learn. The *Replay Buffer* is a finite-sized array, in which past experiences are over-written with newer ones after a certain amount of time. It does not store the entire history of experiences across the agent's lifetime. If the buffer is too large, it would over-replay early experiences that showed poor performance. On the other hand, if the memory is too small, the agent would over-index for recent experiences.

2. **Target Network.** The true label (*TD-Target*) and the predicted label are both estimated from the Q-network. As a result, the TD-target shifts as the network improves. To counter this, the concept of *Target Networks* is introduced. The algorithm keeps two copies of the Q-network — one that is updated at every time step and another that is kept frozen for a set number of steps. This ensures that the target is not continuously moving, thereby improving the stability during training

by reducing oscillations and divergence of the policy. Update of the target network may be *hard* or *soft.* In hard update, after every n gradient update steps ($n >= 1$), the target network is updated by copying the parameters of the current network into the target. In soft update, the update is gradual.

While exploring the environment during training, the RL agent creates an action distribution which describes the agent's belief about the optimal action. Since DQN is off-policy in nature, exploration policies such as the *annealing epsilon-greedy policy* or the *Boltzmann Q-policy* are used by the agent. In the widely used **Boltzmann Q-policy**, the original action distribution gets divided by a *temperature parameter* because of which, the agent's exploration behaviour oscillates between picking an action randomly and always picking the most optimal action.

For loss calculation, metrics such as *mean-squared error (mse)* or *mean absolute error (mae)* are applied to the TD-Target and the predicted output on a mini-batch of past experiences sampled from the Replay Buffer.

This report will concern itself with interpretation techniques and tools that are directed towards explaining decisions taken by trained RL agents in benchmark environments. In literature, this is called **RL Interpretability**. The aim of this project is to explore RL interpretability. For this purpose, it is first necessary to establish a framework for creating RL environments and executing various RL algorithms on these environments. This is covered in the upcoming chapters.

2 Simulating RL in Python3

2.1 Creating an RL environment

OpenAI Gym is a Python toolkit for testing RL algorithms that provides an implementation for different standard RL environments. (A comprehensive list of Gym environments can be checked out here.) It is compatible with any numerical computation library, such as TensorFlow, which can be used to create RL agents for these environments. The standard environments provided for by OpenAI Gym include, among others, several benchmarks such as:

1. **Cart-Pole Problem.** A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center [9].

2. **Lunar Lander (Discrete).** This environment has been designed by OpenAI Gym developers. The goal is to land a Lunar Lander as close between two given flag poles as possible, making sure that both side boosters are touching the ground. Four discrete actions are available: do nothing, fire left orientation engine, fire main engine, and fire right orientation engine. Landing pad is always at coordinates (0,0). [100, 140] points is rewarded for moving to the landing pad and zero speed. If lander then moves away from landing pad it loses the reward. If lander crashes or comes to rest it gets -100 or +100 respectively, and the episode ends. Each leg with ground contact gets +10 reward. Firing the main engine results in -0.3 per frame and firing the side engines results in -0.03 per frame. If the goal state is reached, 200 points are rewarded. However, landing outside landing pad is also possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt.

3. **Ms. Pac-Man.** The Atari 2600 game of Ms. Pac-Man is simulated in OpenAI Gym through the *Arcade Learning Environment* (ALE), which uses the *Stella Atari emulator*. The goal is to maximize the score. In this environment, the observation is the RAM of the Atari machine, consisting of 128 bytes. Each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from {2,3,4}.

4. **Mountain Car.** A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the

car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum [10].

5. **Frozen Lake.** This environment has been designed by OpenAI Gym developers. The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.

The first step in RL is to create an RL environment for the agent to explore. For this, the OpenAI Gym package must be installed first (Python 3.5+ required). This can be done using the following bash commands (tested in an Ubuntu 21.04 system):

```
$ sudo apt install build-essential python3-dev swig \
python3-pygame git libosmesa6-dev libgl1-mesa-glx libglfw3

$ sudo pip3 install ale-py atari-py AutoROM.accept-rom-license \
lz4 opencv-python pyvirtualdisplay pyglet importlib-resources \
Cython cffi glfw imageio lockfile pycparser pillow zipp gym
```

Once OpenAI Gym has been successfully installed, Python code can be written for creating and simulating environments. The following code snippet creates the classic Cart-Pole Problem environment, and then opens a window pop-up simulating it.

Example 2.1: Simulating Cart-Pole in OpenAI Gym

```python
import gym

# Creating the environment
env = gym.make("CartPole-v1")

# Simulating the environment
episodes = int(input("Enter the number of episodes to run: "))
for i in range(episodes):
        observation = env.reset()
        done = False
        t = 0
        while not done:
                env.render()
                action = env.action_space.sample() # take a random action
                observation, reward, done, info = env.step(action)
                t = t + 1
        print(f"Episode {i+1} finished after {t} timesteps.")

# Closing the environment
env.close()
```

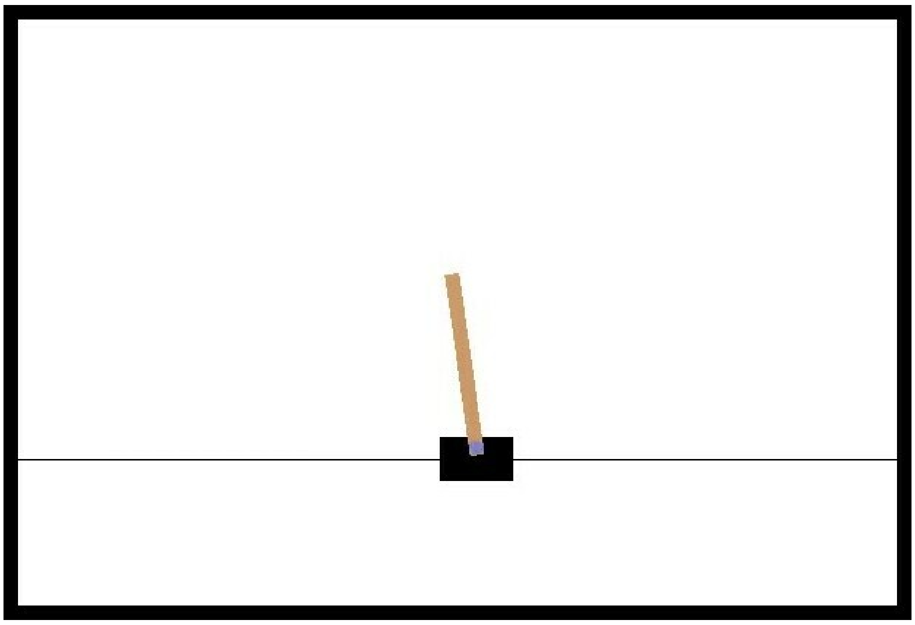


Figure 2.1: A screenshot from the OpenAI Gym Cart-Pole simulation.

Note that the installation guide given here does not cover the installation of Box2D and MuJoCo environments. To install **Box2D**, a 2D Game Physics for Python which runs environments such as *Lunar Lander* and *Mountain Car*, one can visit their github repository and follow the installation guide. However, **MuJoCo** is a licensed physics engine, and the installation of MuJoCo environments for OpenAI Gym does not work properly in many systems and therefore has been omitted.

Also note that in order to import ROMs for **Atari 2600** environments, one needs to download Roms.rar from the Atari 2600 VCS ROM Collection. The archive must then be extracted, followed by the bash command:

```
$ python3 -m atari_py.import_roms <path to folder>
```

2.2 Training and Testing an RL agent

Once an environment has been created for RL, the next step is to train and test an RL agent on it. **Keras-RL** is a deep RL library for Keras that has implementations of state-of-the-art RL algorithms such as Deep Q-Network (*DQN*), Deep Deterministic Policy Gradient (*DDPG*), Normalized Advantage Function (*NAF*), Cross-Entropy Method (*CEM*) and State-action-reward-state-action (*SARSA*). Keras-RL integrates with OpenAI Gym out of the box, hence enabling working with the Gym environments seamlessly. **Keras-RL2** is a fork of Keras-RL with support for TensorFlow 2. It does not have its own documentation, but the existing documentation of Keras-RL can be referred.

To install Keras-RL2 (Python 3.5+ required), the following bash command can be used (tested in an Ubuntu 21.04 system):

```
$ sudo pip3 install keras-rl2
```

Once Keras-RL2 has been successfully installed, Python code can be written for training and testing agents. The following code snippet takes the classic Cart-Pole Problem environment, and executes the Deep Q-Network (DQN) RL algorithm on it.

Example 2.2: Training and Testing Agent via DQN in Keras-RL2

```python
import gym
import numpy as np

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Flatten
from tensorflow.keras.optimizers import Adam

from rl.agents.dqn import DQNAgent
from rl.policy import BoltzmannQPolicy
from rl.memory import SequentialMemory

# Creating the environment in OpenAI Gym
env = gym.make("CartPole-v1")
np.random.seed(123)
env.seed(123)
nb_actions = env.action_space.n # extracting the number of actions

# Building a DQN model
model = Sequential()
model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(nb_actions))
model.add(Activation('linear'))

# Configuring and compiling the agent
memory = SequentialMemory(limit=50000, window_length=1)
policy = BoltzmannQPolicy()
dqn = DQNAgent(model=model, nb_actions=nb_actions, memory=memory,
nb_steps_warmup=100, target_model_update=1e-2, policy=policy)
dqn.compile(Adam(learning_rate=1e-3), metrics=['mae'])
```

```
36
37 # Training the agent via DQN, can press Ctrl + C to skip
38 dqn.fit(env, nb_steps=50000, visualize=False, verbose=2)
39 dqn.save_weights('dqn_CartPole-v1_weights.h5f', overwrite=True)
40
41 # Testing the trained agent for 10 episodes
42 dqn.test(env, nb_episodes=10, visualize=True)
43
44 # Closing the environment
45 env.close()
```

In order to understand the above code snippet, it is necessary to understand the DQN algorithm. A concise introduction to DQN can be found here.

Between lines 13 to 16 of the code, the OpenAI Gym Cart-Pole environment is created and the number of actions in its action space is extracted. Then, using Keras, a neural network is constructed. The input layer has one node corresponding to each feature in the observation space feature vector, which equals 4 in the case of the Cart-Pole environment. There are three hidden layers of 16 nodes each, which use Rectified Linear Unit ($ReLu$) activation function. The output layer has one node corresponding to each action in the action space, which equals 2 in the case of the Cart-Pole environment, and the *Linear* activation function is used. Between lines 19 to 28, this neural network model is defined. This is our Q-network which will be used in training the RL agent.

Between lines 31 to 34, the RL agent is configured with a replay memory of size 50,000 timesteps and a Boltzmann Q-Policy approach to exploration. Training of each episode requires the Q-estimates of the previous episodes. Therefore, before the first episode begins, 100 timesteps are set aside for warming up the agent by initializing the Q-estimates. The parameter target_model_update controls how often the target network is updated. Since the target_model_update is $n = e^{-2}$, a soft update takes place:

$$target\_network = n * target\_network + (1 - n) * model$$

Here, model is the copy of the Q-network that is continually updated, and target_network is the target network. In line 35, the RL agent is compiled to run the DQN algorithm initiated with a learning rate value $\alpha = e^{-3}$ using Keras' inbuilt Adam optimizer (which essentially implements the stochastic gradient descent method), and using mae as the metric for loss calculation. The training is done in line 38 for 50,000 timesteps (each timestep corresponds to choosing an action) and the trained Q-network is saved in the Keras .h5f format in line 39. Finally, in line 42, the trained neural network is tested in the Cart-Pole environment for 10 episodes and the results are simulated.

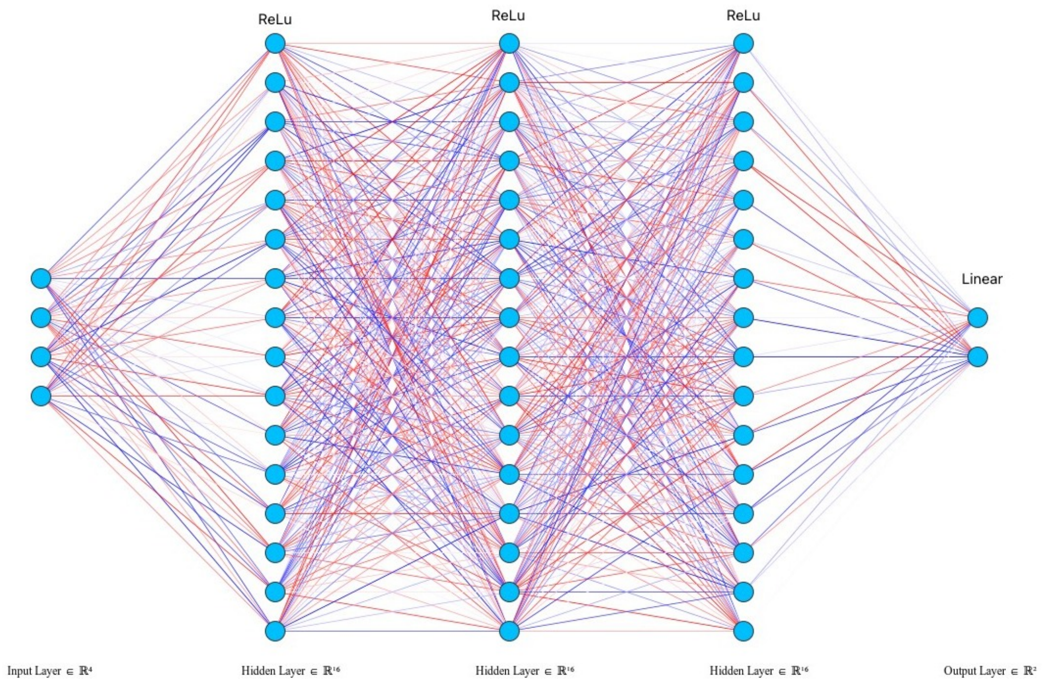
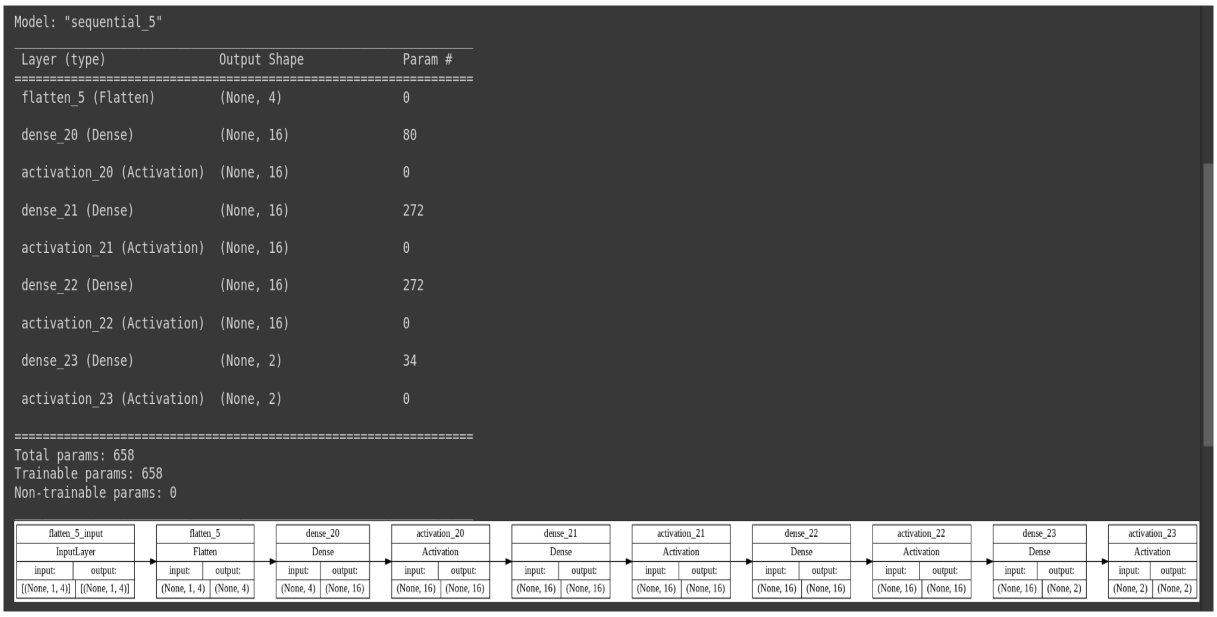
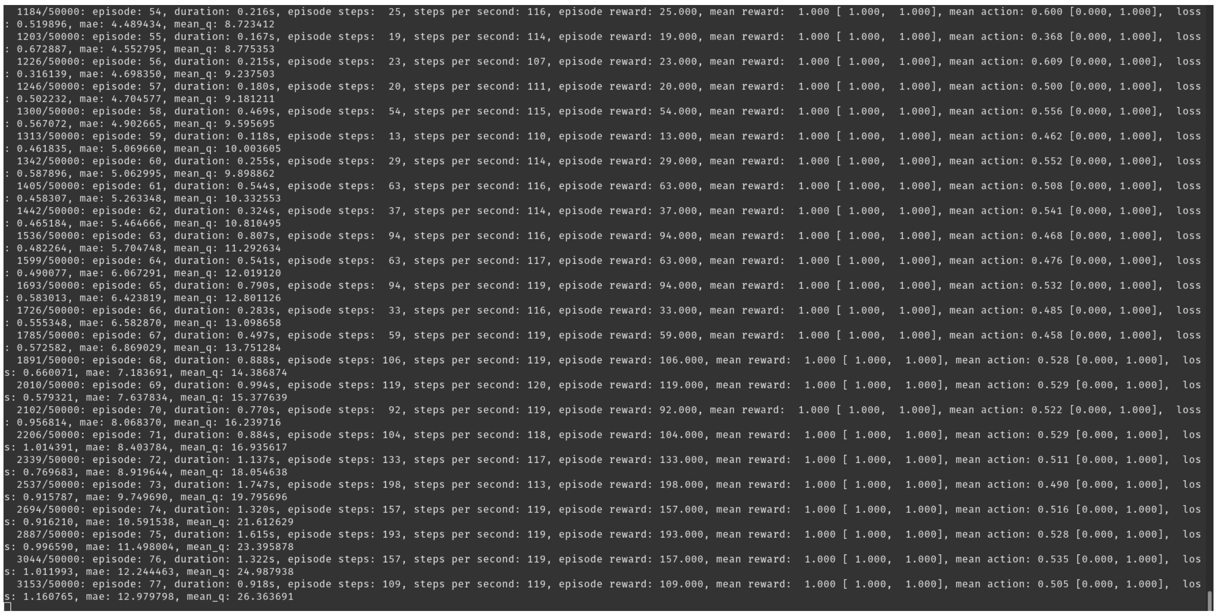


Figure 2.2: A schematic of the Q-network used.



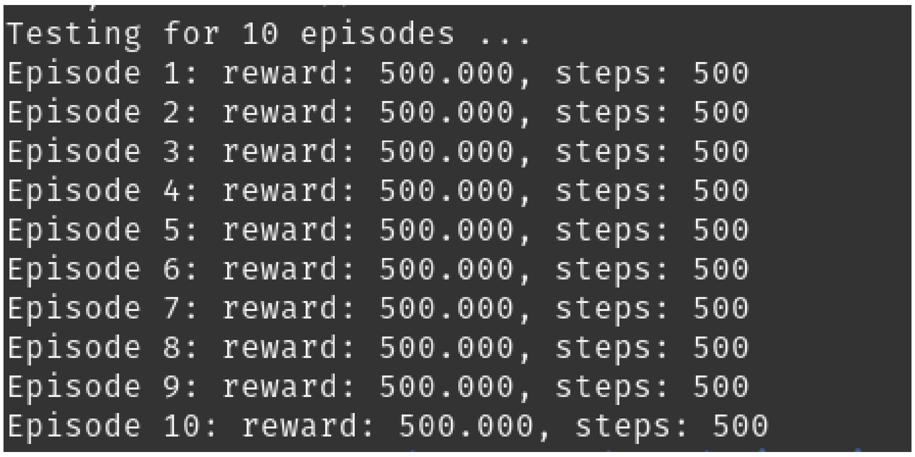
```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_5 (Flatten)         (None, 4)                 0

 dense_20 (Dense)            (None, 16)                80

 activation_20 (Activation)  (None, 16)                0

 dense_21 (Dense)            (None, 16)                272

 activation_21 (Activation)  (None, 16)                0

 dense_22 (Dense)            (None, 16)                272

 activation_22 (Activation)  (None, 16)                0

 dense_23 (Dense)            (None, 2)                 34

 activation_23 (Activation)  (None, 2)                 0

=================================================================
Total params: 658
Trainable params: 658
Non-trainable params: 0
_________________________________________________________________
```

Figure 2.3: A summary of the Q-network using Keras' model.summary() function.



```
  1184/50000: episode: 54, duration: 0.216s, episode steps:  25, steps per second: 116, episode reward: 25.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.600 [0.000, 1.000],  loss
: 0.519896, mae: 4.489434, mean_q: 8.723412
  1203/50000: episode: 55, duration: 0.167s, episode steps:  19, steps per second: 114, episode reward: 19.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.368 [0.000, 1.000],  loss
: 0.672887, mae: 4.552795, mean_q: 8.775353
  1226/50000: episode: 56, duration: 0.215s, episode steps:  23, steps per second: 107, episode reward: 23.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.609 [0.000, 1.000],  loss
: 0.316139, mae: 4.698350, mean_q: 9.237503
  1246/50000: episode: 57, duration: 0.180s, episode steps:  20, steps per second: 111, episode reward: 20.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.500 [0.000, 1.000],  loss
: 0.502232, mae: 4.704577, mean_q: 9.181211
  1300/50000: episode: 58, duration: 0.469s, episode steps:  54, steps per second: 115, episode reward: 54.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.556 [0.000, 1.000],  loss
: 0.567072, mae: 4.902665, mean_q: 9.595695
  1313/50000: episode: 59, duration: 0.118s, episode steps:  13, steps per second: 110, episode reward: 13.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.462 [0.000, 1.000],  loss
: 0.461835, mae: 5.069660, mean_q: 10.003605
  1342/50000: episode: 60, duration: 0.255s, episode steps:  29, steps per second: 114, episode reward: 29.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.552 [0.000, 1.000],  loss
: 0.587896, mae: 5.062995, mean_q: 9.898862
  1405/50000: episode: 61, duration: 0.544s, episode steps:  63, steps per second: 116, episode reward: 63.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.508 [0.000, 1.000],  loss
: 0.458307, mae: 5.263348, mean_q: 10.332553
  1442/50000: episode: 62, duration: 0.324s, episode steps:  37, steps per second: 114, episode reward: 37.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.541 [0.000, 1.000],  loss
: 0.465184, mae: 5.464666, mean_q: 10.810495
  1536/50000: episode: 63, duration: 0.807s, episode steps:  94, steps per second: 116, episode reward: 94.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.468 [0.000, 1.000],  loss
: 0.482264, mae: 5.704748, mean_q: 11.292634
  1599/50000: episode: 64, duration: 0.541s, episode steps:  63, steps per second: 117, episode reward: 63.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.476 [0.000, 1.000],  loss
: 0.490077, mae: 6.067291, mean_q: 12.019120
  1693/50000: episode: 65, duration: 0.790s, episode steps:  94, steps per second: 119, episode reward: 94.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.532 [0.000, 1.000],  loss
: 0.583013, mae: 6.423819, mean_q: 12.801126
  1726/50000: episode: 66, duration: 0.283s, episode steps:  33, steps per second: 116, episode reward: 33.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.485 [0.000, 1.000],  loss
: 0.555348, mae: 6.582870, mean_q: 13.098658
  1785/50000: episode: 67, duration: 0.497s, episode steps:  59, steps per second: 119, episode reward: 59.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.458 [0.000, 1.000],  loss
: 0.572582, mae: 6.869029, mean_q: 13.751284
  1891/50000: episode: 68, duration: 0.888s, episode steps: 106, steps per second: 119, episode reward: 106.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.528 [0.000, 1.000],  los
s: 0.660071, mae: 7.183691, mean_q: 14.386874
  2010/50000: episode: 69, duration: 0.994s, episode steps: 119, steps per second: 120, episode reward: 119.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.529 [0.000, 1.000],  los
s: 0.579321, mae: 7.637834, mean_q: 15.377639
  2102/50000: episode: 70, duration: 0.770s, episode steps:  92, steps per second: 119, episode reward: 92.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.522 [0.000, 1.000],  loss
: 0.956814, mae: 8.068370, mean_q: 16.239716
  2206/50000: episode: 71, duration: 0.884s, episode steps: 104, steps per second: 118, episode reward: 104.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.529 [0.000, 1.000],  los
s: 1.014391, mae: 8.403784, mean_q: 16.935617
  2339/50000: episode: 72, duration: 1.137s, episode steps: 133, steps per second: 117, episode reward: 133.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.511 [0.000, 1.000],  los
s: 0.769683, mae: 8.919644, mean_q: 18.054638
  2537/50000: episode: 73, duration: 1.747s, episode steps: 198, steps per second: 113, episode reward: 198.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.490 [0.000, 1.000],  los
s: 0.915787, mae: 9.749690, mean_q: 19.795696
  2694/50000: episode: 74, duration: 1.320s, episode steps: 157, steps per second: 119, episode reward: 157.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.516 [0.000, 1.000],  los
s: 0.916210, mae: 10.591538, mean_q: 21.612629
  2887/50000: episode: 75, duration: 1.615s, episode steps: 193, steps per second: 119, episode reward: 193.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.528 [0.000, 1.000],  los
s: 0.996590, mae: 11.498004, mean_q: 23.395878
  3044/50000: episode: 76, duration: 1.322s, episode steps: 157, steps per second: 119, episode reward: 157.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.535 [0.000, 1.000],  los
s: 1.011993, mae: 12.244463, mean_q: 24.987938
  3153/50000: episode: 77, duration: 0.918s, episode steps: 109, steps per second: 119, episode reward: 109.000, mean reward:  1.000 [ 1.000,  1.000], mean action: 0.505 [0.000, 1.000],  los
s: 1.160765, mae: 12.979798, mean_q: 26.363691
```

Figure 2.4: Training of the Q-network in progress.



```
Testing for 10 episodes ...
Episode 1: reward: 500.000, steps: 500
Episode 2: reward: 500.000, steps: 500
Episode 3: reward: 500.000, steps: 500
Episode 4: reward: 500.000, steps: 500
Episode 5: reward: 500.000, steps: 500
Episode 6: reward: 500.000, steps: 500
Episode 7: reward: 500.000, steps: 500
Episode 8: reward: 500.000, steps: 500
Episode 9: reward: 500.000, steps: 500
Episode 10: reward: 500.000, steps: 500
```

Figure 2.5: Testing the trained Q-network for 10 episodes.

3 Creating a Custom RL Environment

Other than being able to reproduce benchmark environments, (as demonstrated in Section 2.1) OpenAI Gym also provides tools to allow creation of custom RL environments [11]. The following code snippet creates a simple one-room maze environment of dimension m by n, with a start state, a goal state and a certain number of flags for the RL agent to collect.

Example 3.1: SimpleMaze.py: Creating a Custom Maze Environment in OpenAI Gym

```python
import gym
import numpy as np
import random
from gym import spaces

ACT_DICT = {'0':'Move up', '1':'Move down', '2':'Move left', '3':'Move right'}
DICT = {'0':'O', '1':'*', '2':'S', '3':'G', '4':'X'}

class SimpleMaze(gym.Env):

    def __init__(self, m, n, deterministic, steps=-1):
        super(SimpleMaze, self).__init__()
        self.action_space = spaces.Discrete(4)
        low = np.array([0,0],dtype=np.int_,)
        high = np.array([m-1,n-1],dtype=np.int_,)
        self.observation_space = spaces.Box(low, high, dtype=np.int)
        self.m = m
        self.n = n
        self.deterministic = deterministic
        self.max_step = steps
        self.reset()

    def reset(self):
        maze = []
        for i in range(self.m):
            maze_row = []
            for j in range(self.n):
                maze_row.append(0)
            maze.append(maze_row)
        self.maze = maze
        self.nb_step = 0
        self.nb_flags = 0
        arr = []
        for i in range(int(np.sqrt(self.m*self.n))):
            arr.append(1)
        arr.append(2)
        arr.append(3)
        while len(arr) > 0:
            x = random.choice(arr)
            i = random.choice(np.arange(self.m))
            j = random.choice(np.arange(self.n))
            if self.maze[i][j] == 0:
                self.maze[i][j] = x
                arr.remove(x)
                if x == 2:
                    self.x = i
                    self.y = j
        self.maze[self.x][self.y] = 4
        self.loc = 2
        return [self.x,self.y]

    def render(self, mode='human', close=False):
        print(f"\nNext action:{ACT_DICT[str(self.action)]}")
        print("Map:")
        for i in self.maze:
            string = ''
            for j in i:
                string = string + DICT[str(j)] + ' '
            print(string)

    def step(self, action):

        done = False
        if self.nb_flags == int(np.sqrt(self.m*self.n)):
            reward = -5
        else:
            reward = -1
        self.maze[self.x][self.y] = self.loc
        if self.deterministic == True:
            self.action = action
        else:
            self.action = self.probability_matrix(action)

        if self.action == 0 and self.x != 0:
                self.x -= 1
            if self.action == 1 and self.x != self.m-1:
                self.x += 1
            if self.action == 2 and self.y != 0:
                self.y -= 1
            if self.action == 3 and self.y != self.n-1:
                self.y += 1

            self.loc = self.maze[self.x][self.y]
            self.maze[self.x][self.y] = 4

            self.nb_step += 1

            if self.loc == 1:
                self.loc = 0
                self.nb_flags += 1
                reward += 10

            if self.loc == 3:
                done = True
                reward += 20

            if self.nb_step == self.max_step:
                done = True

            return [self.x,self.y], reward, done, {}

        def probability_matrix(self, action):
            arr = [0,1,2,3]
            arr.remove(action)
            x = random.choice([1,2,3,4,5,6,7,8,9,10])
            if x < 8:
                return action
            else:
                return random.choice(arr)
```

In order to understand the above code snippet, it is necessary to look at the class *SimpleMaze* function by function. Any *gym.Env* class is required to have four member functions:

1. **The Constructor (__init__).** The constructor defines the basic data members of the environment class, including the action space and the observation space. Here, the action space is a discrete space consisting of the four possible actions the RL agent can take: 0 (*Move up*), 1 (*Move down*), 2 (*Move left*), and 3 (*Move right*).

On the other hand, the observation space is a box space of size 2, meaning that it consists of two variables, each taking values over a continuous interval. These two variables are $x \in [0, m-1]$ and $y \in [0, n-1]$, and are sufficient to understand the state of the environment at any given point. Other variables which will be required to define the environment completely are the dimensions of the maze m and n, a Boolean flag *deterministic* to indicate whether the system is deterministic in terms of actions, and an upper bound *max_step* for the number of timesteps per episode. The variables m, n, *deterministic* and *max_step* are passed to the constructor as arguments. The default value of *max_step* is set to -1, which implies that there is no upper bound on the number of timesteps per episode. In this code, the constructor calls the reset function to ensure that every time an environment of this type is created, it is reset to its initial state.

2. **The Reset Function.** The reset function defines the initial state of the environment. For this particular environment, the initial state is defined as a one-room maze of dimension m by n represented using a two-dimensional Python list, with the start cell S, the goal cell G, and $\lfloor \sqrt{m.n} \rfloor$ flags randomly placed. Therefore, with each reset, the positions of S, G and the flags will change. Initially, the number of timesteps covered in the episode (*nb_step*) and the number of flags collected (nb_flags) are both set to zero. The variable *loc*, which is used in the step function to understand whether a given position is the goal state or contains a flag, is set to 2, which indicates the start state. The reset function returns an observation vector which defines the initial state of the environment.

3. **The Render Function.** The render function is responsible for visualizing the environment after each timestep in an episode. For this particular environment, the rendering is a simple printing of the action taken, along with the new state of the environment after taking that action. The action taken is printed by mapping the integers in the action space to their corresponding actions, as defined by the dictionary ACT_DICT. As described earlier, there are four possible actions for every timestep: 0 (*Move up*), 1 (*Move down*), 2 (*Move left*), and 3 (*Move right*). The state of the environment is printed by mapping every character of the two-dimensional Python list to the corresponding character, as defined by the dictionary DICT. '0' represents a normal cell without any flag (represented by 0 in the encoding), '*' indicates a cell with a flag (represented by 1 in the encoding), 'S' indicates the start cell (represented by 2 in the encoding), 'G' indicates the goal cell (represented by 3 in the encoding), and 'X' indicates the current position of the RL agent (represented by 4 in the encoding).

4. **The Step Function.** The step function takes an action in the action space as its argument and describes how the state of the environment changes after taking that action. For this particular environment, the possible actions an agent can take are *Move up*, *Move down*, *Move left* and *Move right*. A penalty of -1 is awarded for every step taken, -5 if all of the $\lfloor \sqrt{m.n} \rfloor$ flags have been already collected. This is done to encourage the RL agent to minimize the number of steps and stop

exploration after all the flags have been collected. If the flag *deterministic* is set to True, the action passed as the argument to the function is set as the action which is to be taken. Otherwise, the function probability_matrix(self, action) is called which introduces the element of non-determinism by returning the correct action with 0.7 probability, and the other three actions with 0.1 probability each. After the action to be performed has been chosen, the two-dimensional Python list which encodes the maze is updated by updating the current x and y, storing the information of the cell in *loc*, and replacing it with 4 which, as described earlier, indicates the current position. Then, the variable *nb_step* is incremented to reflect that the action has been taken. If the current cell stores a flag, the flag is collected by replacing the 1 in the cell with a 0, the variable *nb_flags* is incremented, and a reward of +10 is awarded consequently. The episode terminates when either goal state or the maximum number of steps (if any) is reached, whichever earlier. In either case, a Boolean flag *done* is set to True to indicate that the episode has been completed. A reward of +20 is awarded if the episode terminates by reaching the goal state. The step function returns four variables: an observation vector *observation* which defines the new state of the environment, the net immediate reward *reward* obtained in this timestep, the Boolean flag *done*, and a dictionary *info* for debugging purposes which, in this case, is empty.

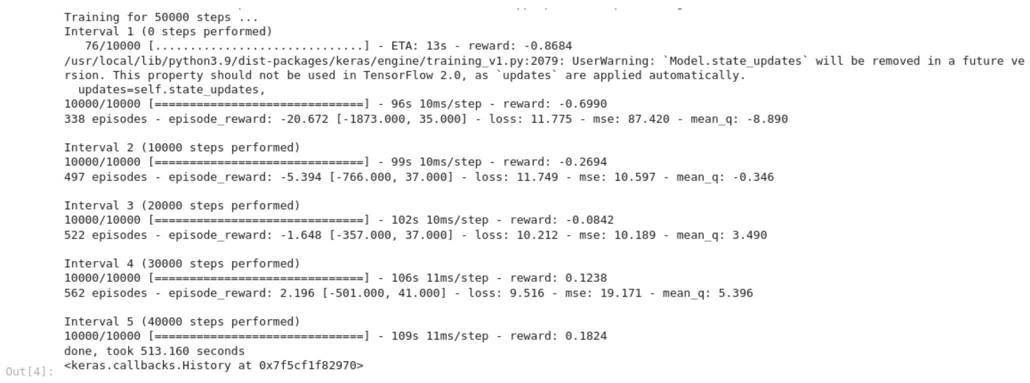
Other than these four functions, a *gym.Env* class may also have a **close function**, which when called, closes the rendered window. This function is useful in cases where the render function creates a separate window for visualization of the environment. However, since the render function for this particular environment prints the output in the console itself, there is no need for a close function.

The following code snippet creates a SimpleMaze environment *env* using the SimpleMaze.py of Example 3.1. The environment *env* is a 3 by 5 one-room maze with a random start cell, a random goal state, and three flags randomly distributed across the maze. *env* is deterministic in terms of action taken, and has no upper bound on the number of timesteps per episode.

Example 3.2: Creating a SimpleMaze environment using SimpleMaze.py of Example 3.1

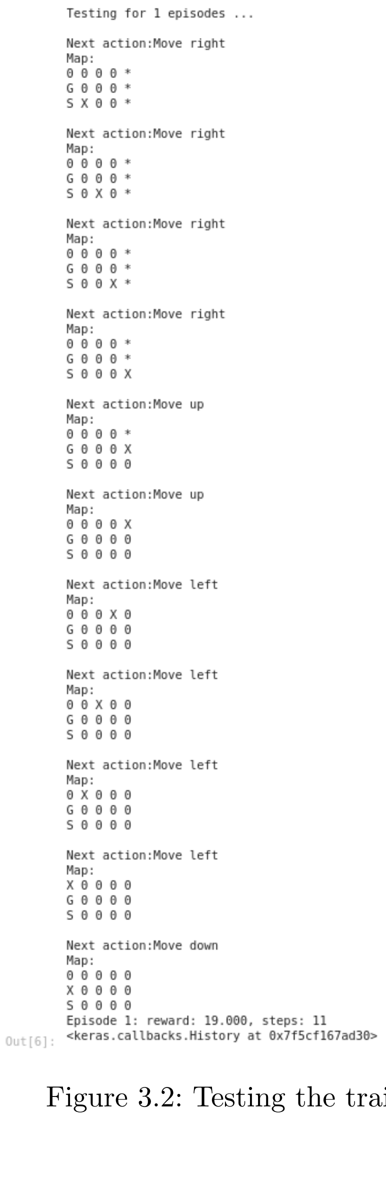
```
from SimpleMaze import SimpleMaze
env = SimpleMaze(3,5,True)
```

Once the environment has been created, it can be used for RL simulation like any benchmark environment. The upcoming chapters discuss the various tools and techniques for RL interpretability and how these can be applied in context of the present framework.



```
Training for 50000 steps ...
Interval 1 (0 steps performed)
   76/10000 [..............................] - ETA: 13s - reward: -0.8684
/usr/local/lib/python3.9/dist-packages/keras/engine/training_v1.py:2079: UserWarning: `Model.state_updates` will be removed in a future ve
rsion. This property should not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
10000/10000 [==============================] - 96s 10ms/step - reward: -0.6990
338 episodes - episode_reward: -20.672 [-1873.000, 35.000] - loss: 11.775 - mse: 87.420 - mean_q: -8.890

Interval 2 (10000 steps performed)
10000/10000 [==============================] - 99s 10ms/step - reward: -0.2694
497 episodes - episode_reward: -5.394 [-766.000, 37.000] - loss: 11.749 - mse: 10.597 - mean_q: -0.346

Interval 3 (20000 steps performed)
10000/10000 [==============================] - 102s 10ms/step - reward: -0.0842
522 episodes - episode_reward: -1.648 [-357.000, 37.000] - loss: 10.212 - mse: 10.189 - mean_q: 3.490

Interval 4 (30000 steps performed)
10000/10000 [==============================] - 106s 11ms/step - reward: 0.1238
562 episodes - episode_reward: 2.196 [-501.000, 41.000] - loss: 9.516 - mse: 19.171 - mean_q: 5.396

Interval 5 (40000 steps performed)
10000/10000 [==============================] - 109s 11ms/step - reward: 0.1824
done, took 513.160 seconds
Out[4]: <keras.callbacks.History at 0x7f5cf1f82970>
```

Figure 3.1: Training an RL agent on *env* using DQN.



```
Testing for 1 episodes ...

Next action:Move right
Map:
0 0 0 0 *
G 0 0 0 *
S X 0 0 *

Next action:Move right
Map:
0 0 0 0 *
G 0 0 0 *
S 0 X 0 *

Next action:Move right
Map:
0 0 0 0 *
G 0 0 0 *
S 0 0 X *

Next action:Move right
Map:
0 0 0 0 *
G 0 0 0 *
S 0 0 0 X

Next action:Move up
Map:
0 0 0 0 *
G 0 0 0 X
S 0 0 0 0

Next action:Move up
Map:
0 0 0 0 X
G 0 0 0 0
S 0 0 0 0

Next action:Move left
Map:
0 0 0 X 0
G 0 0 0 0
S 0 0 0 0

Next action:Move left
Map:
0 0 X 0 0
G 0 0 0 0
S 0 0 0 0

Next action:Move left
Map:
0 X 0 0 0
G 0 0 0 0
S 0 0 0 0

Next action:Move left
Map:
X 0 0 0 0
G 0 0 0 0
S 0 0 0 0

Next action:Move down
Map:
0 0 0 0 0
X 0 0 0 0
S 0 0 0 0
Episode 1: reward: 19.000, steps: 11
Out[6]: <keras.callbacks.History at 0x7f5cf167ad30>
```

Figure 3.2: Testing the trained RL agent for one episode.

4 References

1. Christoph Molnar. "Interpretable Machine Learning: A Guide for Making Black Box Models Explainable". (Leanpub, 2019)

2. Cynthia Rudin et al. "Interpretable Machine Learning: Fundamental Principles and 10 Grand Challenges". (arXiv, 2021)

3. Namita Agarwal and Saikat Das. "Interpretable Machine Learning Tools: A Survey". (IEEE Symposium Series on Computational Intelligence, 2020)

4. Vijay Arya et al. "One Explanation Does Not Fit All: A Toolkit and Taxonomy of AI Explainability Techniques". (arXiv, 2019)

5. Richard S. Sutton and Andrew G. Barto. "Reinforcement Learning: An Introduction (2nd Edition)". (MIT Press, 2018)

6. Robert Moni. "Reinforcement Learning algorithms — an intuitive overview". (Medium, 2019)

7. Shaked Zychlinski. "The Complete Reinforcement Learning Dictionary". (Towards Data Science, 2019)

8. Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". (Nature, 2015)

9. Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem". (IEEE Transactions on Systems, Man, and Cybernetics, 1983)

10. Andrew W. Moore. "Efficient Memory-based Learning for Robot Control". (University of Cambridge, 1990)

11. Adam King. "Create custom gym environments from scratch — A stock market example". (Towards Data Science, 2019)