ST1504 DEEP LEARNING

Practical 6
Deep Reinforcement Learning

What you will learn / do in this lab

1. *Use Reinforce.js to experiment with the basic principles of Reinforcement Learning on a browser*
2. *Control actions using the OpenAI Gym and Reinforcement Learning*

TABLE OF CONTENTS

1.
OVERVIEW

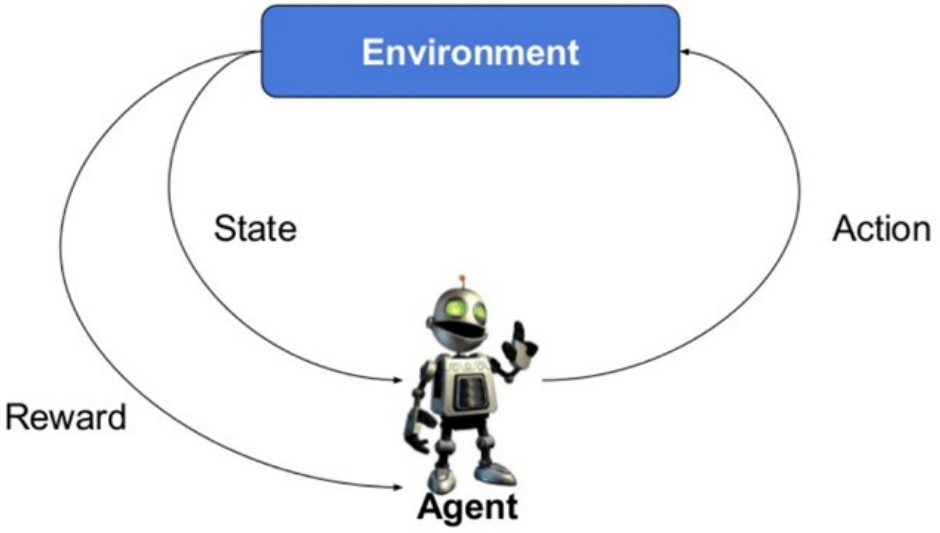
In this practical we will use python to implement Reinforcement Learning. Reinforcement Learning has many applications where the ultimate success depends on a series of steps towards the final goal instead of just optimizing for the immediate future (e.g. classification problems).

INTRODUCTION TO REINFORCEMENT LEARNING (RL)

Reinforcement learning describes a class of problems where an agent operates in an environment and must learn to operate using feedback.

Reinforcement learning is learning what to do — how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them.

Typical RL scenario



Breaking it down, the process of Reinforcement Learning involves these simple steps:

- *Observation of the environment*
- *Deciding how to act using some strategy*
- *Acting accordingly*
- *Receiving a reward or penalty*
- *Learning from the experiences and refining our strategy*
- *Iterate until an optimal strategy is found*

APPLICATIONS OF REINFORCEMENT LEARNING (RL)

Application areas of RL includes:

- *Robotics for industrial automation.*
- *Business strategy planning*
- *Game playing AI (e.g. AlphaGo)*
- *It helps you to create training systems that provide custom instruction and materials according to the requirement of students.*
- *Aircraft control and robot motion control*

2.
REINFORCE.JS

For this lab we would be using the interactive demonstration using the JavaScript library reinforce.js.

https://cs.stanford.edu/people/karpathy/reinforcejs/index.html

Reinforce.js is a Reinforcement Learning library that implements several common RL algorithms supported with fun web demos.

The library currently includes the following algorithms.

Dynamic Programming
For solving finite (and not too large), deterministic Markov Decision Processes (MDPs). The solver uses standard tabular methods will no bells and whistles, and the environment must provide the dynamics.

Tabular Temporal Difference Learning
Both SARSA and Q-Learning are included. The agent still maintains tabular value functions but does not require an environment model and learns from experience. Support for many bells and whistles is also included such as Eligibility Traces and Planning (with priority sweeps).

Deep Q Learning
Reimplementation of Mnih et al. Atari Game Playing model. The approach models the action value function Q(s,a) with a neural network and hence allows continuous input spaces. However, with a fixed number of discrete actions. The implementation includes most of the bells and whistles (e.g. experience replay, TD error clamping for robustness).

Policy Gradients
The implementation includes a stochastic policy gradient Agent that uses REINFORCE and LSTMs that learn both the actor policy and the

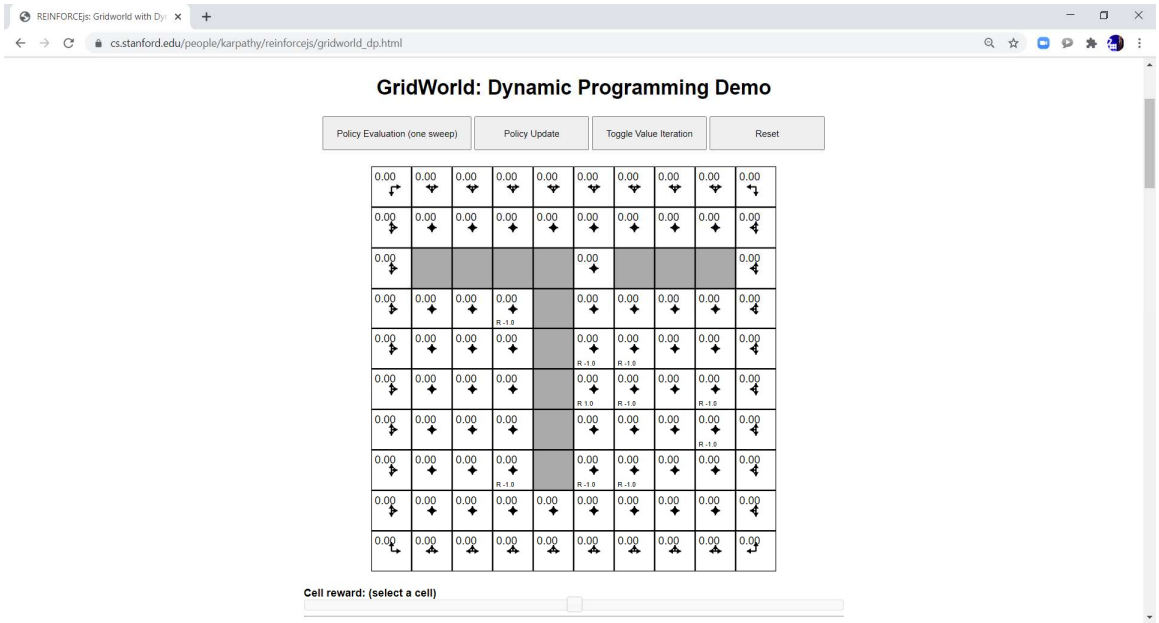
value function baseline, and also an implementation of recent Deterministic Policy Gradients by Silver et al.

GRIDWORLD (DP)

https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html

What is the algorithm implemented in this demo?
Explain what each for the following button does:

- *Policy Evaluation (one sweep)*
- *Policy Update*
- *Toggle Value Iteration*
- *Reset*

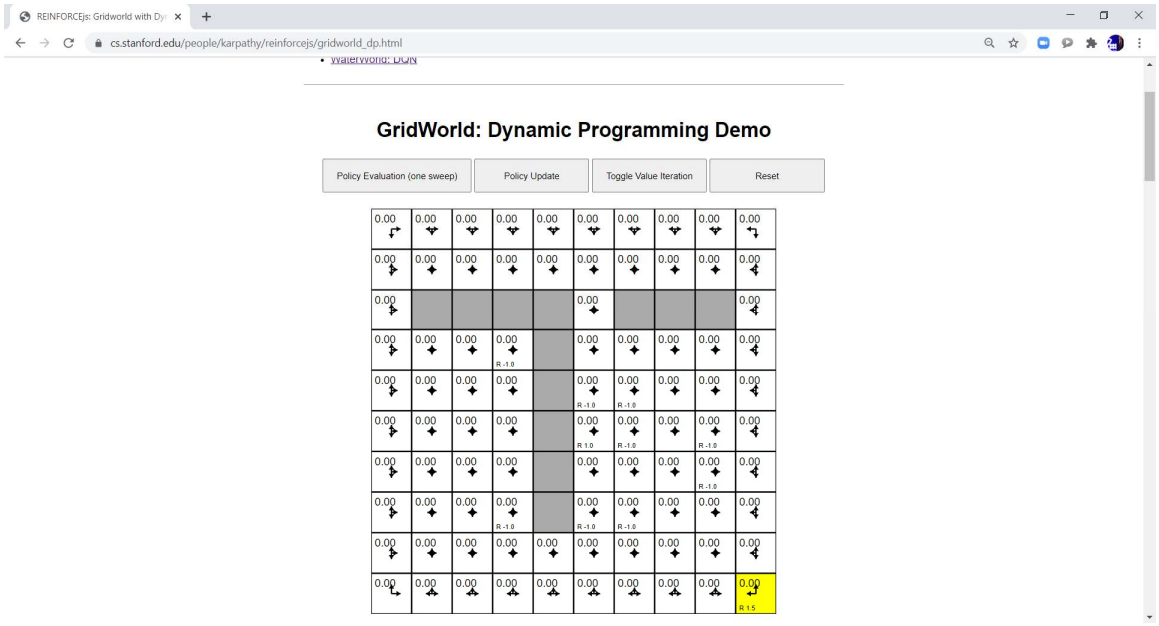


This is a toy environment called **Gridworld** that is often used as a toy model in the Reinforcement Learning literature. In this particular case:

- **State space**: GridWorld has 10x10 = 100 distinct states. The start state is the top left cell. The gray cells are walls and cannot be moved to.
- **Actions**: The agent can choose from up to 4 actions to move around. In this example
- **Environment Dynamics**: GridWorld is deterministic, leading to the same new state given each state and action
- **Rewards**: The agent receives +1 reward when it is in the centre square (the one that shows R 1.0), and -1 reward in a few states

(R -1.0 is shown for these). The state with +1.0 reward is the goal state and resets the agent back to start.

In other words, this is a deterministic, finite Markov Decision Process (MDP) and as always the goal is to find an agent policy (shown here by arrows) that maximizes the future discounted reward.

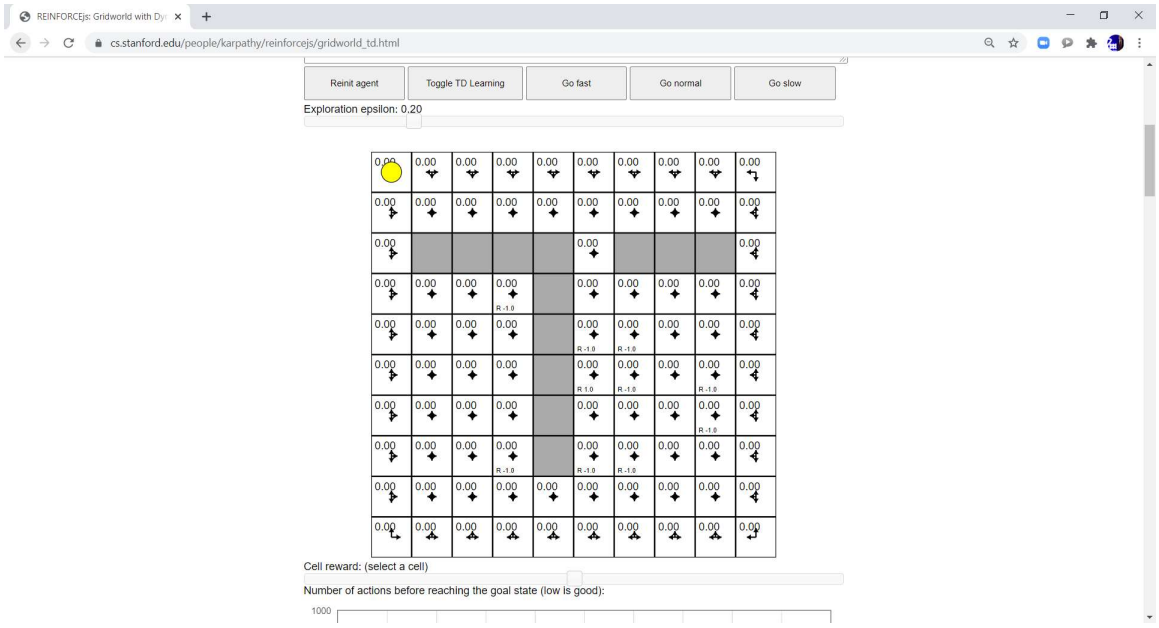


Set the bottom right cell, to a reward of 1.5.
Reset the GridWorld and run it.
Capture the screen.
Explain what happens to the Gridworld

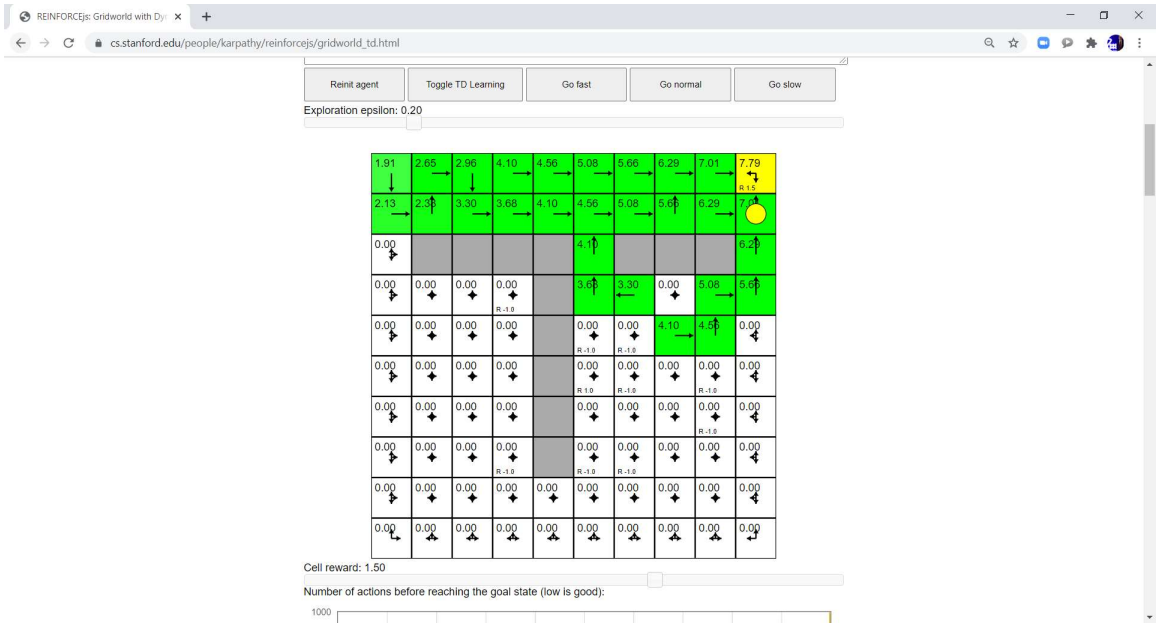
GRIDWORLD (TD)

https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_td.html

What is the algorithm implemented in this demo?



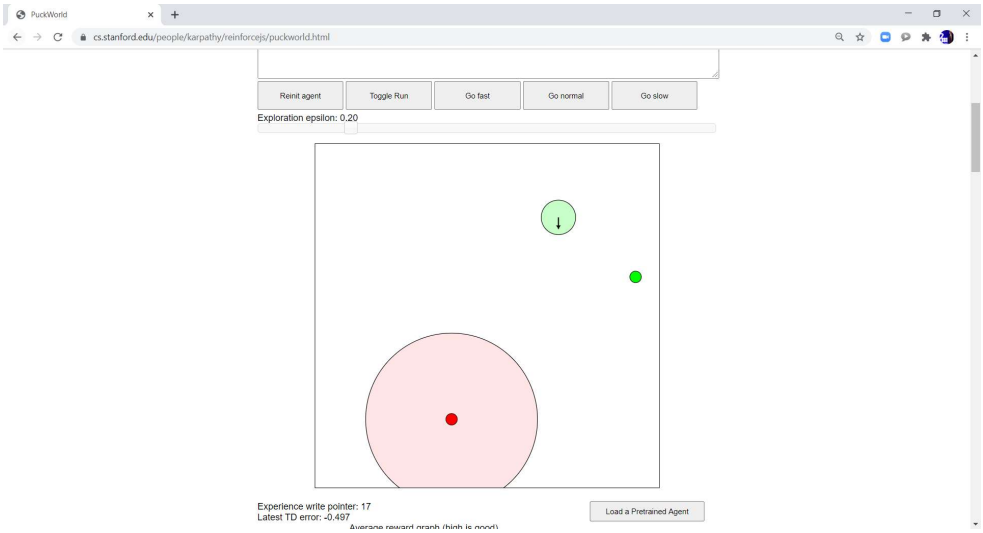
Try to replicate the figure below.
Capture the screen.



PUCKWORLD (DQN)

https://cs.stanford.edu/people/karpathy/reinforcejs/puckworld.html

What is the algorithm implemented in this demo?



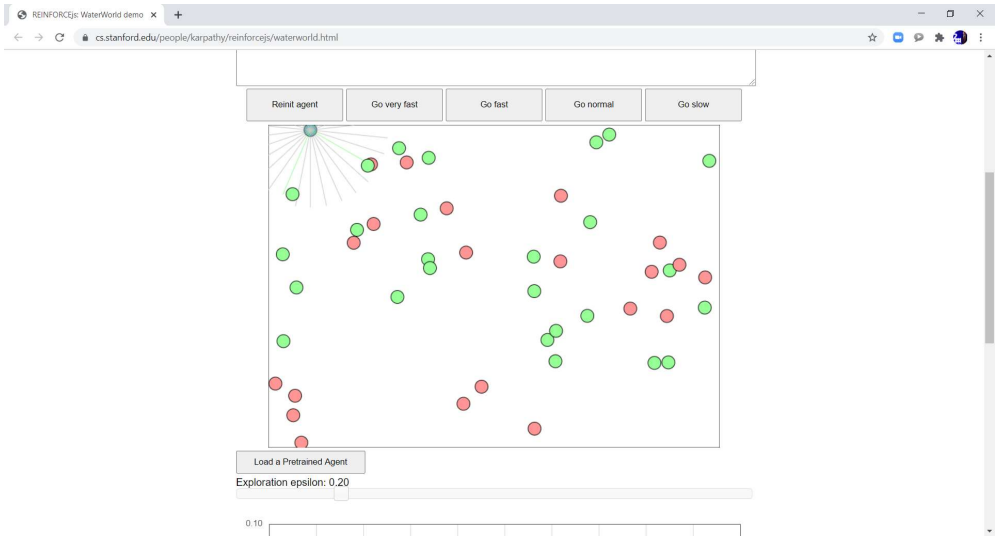
This demo is a modification of **PuckWorld**:

- The **state space** is now large and continuous: The agent observes its own location (x,y), velocity (vx,vy), the locations of the green target, and the red target (total of 8 numbers).
- There are 4 **actions** available to the agent: To apply thrusters to the left, right, up and down. This gives the agent control over its velocity.
- The PuckWorld **dynamics** integrate the velocity of the agent to change its position. The green target moves once in a while to a random location. The red target always slowly follows the agent.
- The **reward** awarded to the agent is based on its distance to the green target (low is good). However, if the agent is in the vicinity of the red target (inside the disk), the agent gets a negative reward proportional to its distance to the red target.

The optimal strategy for the agent is to always go towards the green target (this is the regular PuckWorld), but to also avoid the red target's area of effect. This makes things more interesting because the agent has to learn to avoid it. Also, sometimes it's fun to watch the red target corner the agent. In this case, the optimal thing to do is to temporarily pay the price to zoom by quickly and away, rather than getting cornered and paying much more reward price when that happens.

WATERWORLD (DQN)

https://cs.stanford.edu/people/karpathy/reinforcejs/waterworld.html



This is another Deep Q Learning demo with a more realistic and larger setup:

- The **state space** is even larger and continuous: The agent has 30 eye sensors pointing in all directions and in each direction is observes 5 variables: the range, the type of sensed object (green, red), and the velocity of the sensed object. The agent's proprioception includes two additional sensors for its own speed in both x and y directions. This is a total of 152-dimensional state space.
- There are 4 **actions** available to the agent: To apply thrusters to the left, right, up and down. This gives the agent control over its velocity.
- The **dynamics** integrate the velocity of the agent to change its position. The green and red targets bounce around.
- The **reward** awarded to the agent is +1 for making contact with any red target (these are apples) and -1 for making contact with any green target (this is poison).

Observe what is the optimal strategy for the agent.
Note it down.

3.
REINFORCEMENT LEARNING WITH OPENAI GYM

We will be using OpenAI Gym to provide an environment for our agent. OpenAI Gym is an open source toolkit to develop and compare RL algorithms. It contains a variety of simulated environments that can be used to train agents and develop new RL algorithms.

The first thing to do is install OpenAI Gym, the following command will install the minimal gym package:

```
pip install gym
```

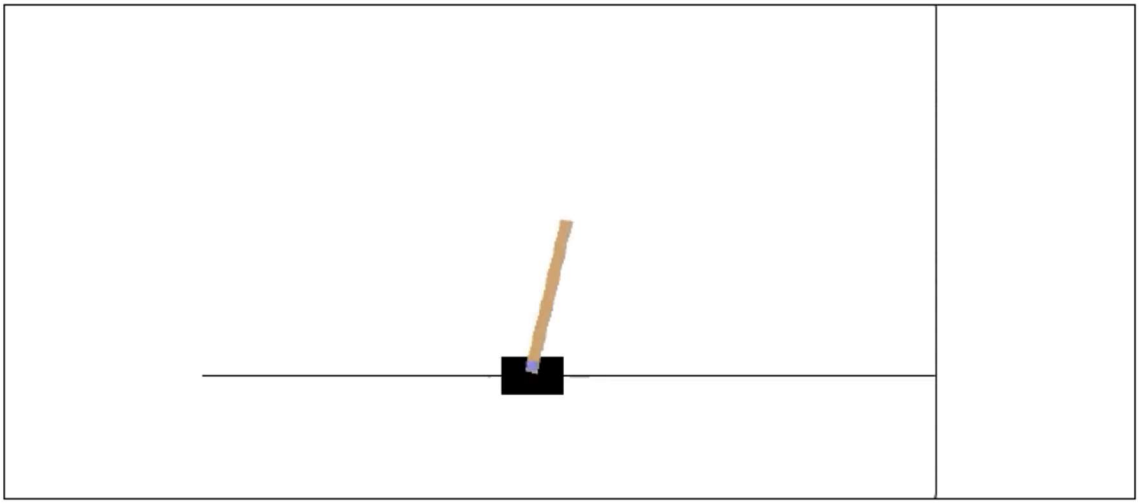
ENVIRONMENT

Unlike supervised learning, we will not be training the RL model for the agent using labelled data or examples. In order words, we do not have a correct answer to use to train the agent.

We need to determine how we will represent the world the agent is interacting by setting up the environment. The OpenAI gym enables us to provide a simulated environment. For this lab, we will be creating an environment emulating the classic pole-on-cart problem.

TRAINING AND EVALUATING THE MODEL

CartPole is a classic OpenAI problem with continuous state space and discrete action space. In it, a pole is attached by an un-actuated joint to a cart; the cart moves along a frictionless track. The goal is to keep the pole standing on the cart by moving the cart left or right. A reward of +1 is given for each time step the pole is standing. Once the pole is more than 15 degrees from the vertical, or the cart moves beyond 2.4 units from the center, the game is over



The code here is adapted from the best entry at OpenAI for the CartPole environment: https://gym.openai.com/envs/CartPole-v0/.

We start with importing the necessary modules. We require gym obviously to provide us with the CartPole environment, and tensorflow to build our DQN network. Besides these we need random and numpy modules.

```
import random
import gym
import math
import numpy as np
from collections import deque
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
```

The code for this lab found in DQNCartPole.ipynb

Now let us build our DQN. We declare a class DQN and in its __init__() function declare all the hyperparameters and our model. We are creating the environment also inside the DQN class. As you can see, the class is quite general, and you can use it to train for any Gym environment whose state space information can be encompassed in a 1D array

```
def __init__(self, env_string, batch_size=64):
    self.memory = deque(maxlen=100000)
    self.env = gym.make(env_string)
    self.input_size = self.env.observation_space.shape[0]
    self.action_size = self.env.action_space.n
    self.batch_size = batch_size
    self.gamma = 1.0
    self.epsilon = 1.0
    self.epsilon_min = 0.01
    self.epsilon_decay = 0.995
    alpha=0.01
    alpha_decay=0.01
    if MONITOR: self.env = gym.wrappers.Monitor(self.env, '../
        data/'+env_string, force=True)
    # Init model
    self.model = Sequential()
    self.model.add(Dense(24, input_dim=self.input_size,
        activation='tanh'))
    self.model.add(Dense(48, activation='tanh'))
    self.model.add(Dense(self.action_size, activation='linear'))
    self.model.compile(loss='mse', optimizer=Adam(lr=alpha,
        decay=alpha_decay))
```

The DQN that we have built is a three-layered perceptron; in the following output you can see the model summary. We use Adam optimizer with learning rate decay.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 24)                120
_________________________________________________________________
dense_1 (Dense)              (None, 48)                1200
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 98
=================================================================
Total params: 1,418
Trainable params: 1,418
Non-trainable params: 0
_________________________________________________________________
```

Next, we write a method to train the agent. We define two lists to keep track of the scores. First, we fill the experience replay buffer and then we choose some samples from it to train the agent and hope that the agent will slowly learn to do better.

```
def train(self):
    scores = deque(maxlen=100)
    avg_scores = []
    for e in range(EPOCHS):
        state = self.env.reset()
        state = self.preprocess_state(state)
    done = False
    i = 0
    while not done:
        action = self.choose_action(state,self.epsilon)
        next_state, reward, done, _ = self.env.step(action)
        next_state = self.preprocess_state(next_state)
        self.remember(state, action, reward, next_state, done)
        state = next_state
        self.epsilon = max(self.epsilon_min, self.epsilon_
        decay*self.epsilon) # decrease epsilon
        i += 1
    scores.append(i)
    mean_score = np.mean(scores)
    avg_scores.append(mean_score)
    if mean_score >= THRESHOLD and e >= 100:
        print('Ran {} episodes. Solved after {} trials✓'.format(e, e - 100))
        return avg_scores
    if e % 100 == 0:
        print('[Episode {}] - Mean survival time over last 100 episodes was {}
ticks.'.format(e, mean_score))
    self.replay(self.batch_size)
    print('Did not solve after {} episodes :('.format(e))
    return avg_scores
```