Solving the CartPole problem with a Deep Q-Network

Author: Louis Miranda-Smedley

Intended as a guide for future years or those interested being introduced to machine learning problems.

CartPole-v0

OpenAI is a non-profit organisation with the mission to ensure artificial general intelligence benefits humanity. They have created Gym, which is a toolkit for developing and testing reinforcement learning algorithms. They make available many different environments such as Atari games, classic control systems (such as CartPole-v0) and robotics environments. This is a great place to start with getting to grips with machine learning.

Perhaps one of the simplest environments is the CartPole-v0 environment, which is the problem involving balancing a rod on a cart which is free to move left and right along an axis. Is it possible to create an algorithm where the agent will learn how to balance the pole through reinforcement learning? We will see that this is indeed the case.

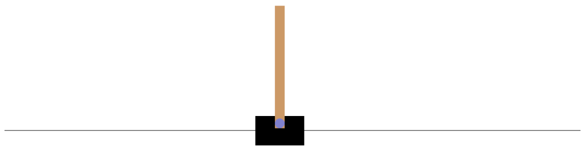


Figure 1: CartPole-v0 environment.

As seen in figure 1, the CartPole system is essentially an inverted pendulum with its centre of mass above its pivot point. The pivot point is the lilac spot on the cart and the cart can apply forces left and right in order to attempt to balance the pole above. The pivot point is un-actuated and the cart moves along a frictionless track. A reward of +1 is given for every timestep the pole remains up right in an episode and the episode finishes when the pole falls by more than 15^0 to the vertical or if the cart moves more than 2.4 units away from the centre.

Basic Installation and Set Up

Before we get started with the reinforcement learning algorithm and our deep neural network, these are the steps for downloading gym and to get the environment up and running as required. Firstly, pip install gym. In order to visualize the environment and get the cart to apply random forces, the simple code below should get you up and running.

```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample()) # take a random action
env.close()
```

Figure 2: Set up code.

Reinforcement Learning

Reinforcement Learning (RL) is a trial-and-error based learning method which focuses on an agent taking actions in an environment and maximise a cumulative reward. The RL algorithm is rewarded based on good actions which improve its performance and penalised based on bad actions which hinder the performance. The longer the agent has to interact with its environment, i.e. longer training, the more information the algorithm accumulates and therefore, can perform the predefined goal more effectively.

Q-Learning

Q-Learning is a model free form of reinforcement learning algorithm, it is based on the Markov decision chain and processes of exploration and exploitation. We start with some initial state S_t of the environment, without an associated reward. For each iteration, the algorithm takes the current state S_t an chooses the best action A_t to take based on the current state, and executes that action on the environment. The environment then returns a reward for the previous action R_{t+1}, the new state of the environment S_{t+1} as well as a done status (if the new state has failed on any of our initial outlined criteria).

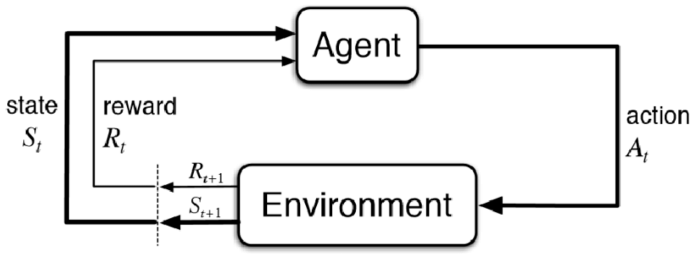
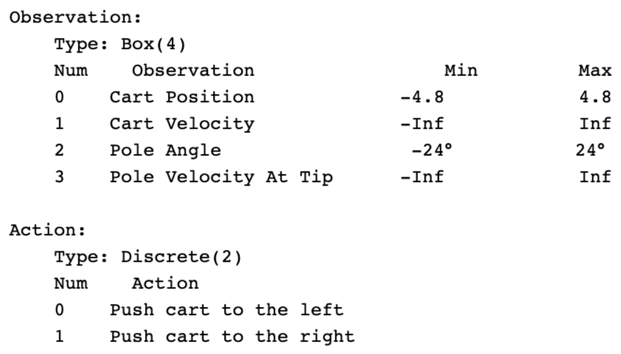


Figure 3: Markov chain.

In our CartPole environment we can see what the observation space and action space looks like. This gives us an understanding of what the states S_t and actions A_t look like for each iteration. We will be using the observations as our inputs and the actions as outputs for our model, where we can see there are four numbers which fully describe the state of the system and only one number (either 0 or 1) which defines the action.



```
Observation:
    Type: Box(4)
    Num    Observation                 Min         Max
    0    Cart Position             -4.8          4.8
    1    Cart Velocity             -Inf          Inf
    2    Pole Angle                 -24°        24°
    3    Pole Velocity At Tip      -Inf          Inf

Action:
    Type: Discrete(2)
    Num    Action
    0    Push cart to the left
    1    Push cart to the right
```

Figure 4: Observation and Action spaces.

The agent therefore tries to come up with the best action given its current state. So, given a current state of the system, the agent needs to pick the best action to maximise rewards. Through trial and error, the agent learns through accumulating knowledge of these (state, action) pairs and can tell if there would be a positive or negative reward associated with a given (state, action) pair. These predictions of rewards based on (state, action) pairs are called q-values. A Q-Table is a very crude way of storing this information and may take the following form.

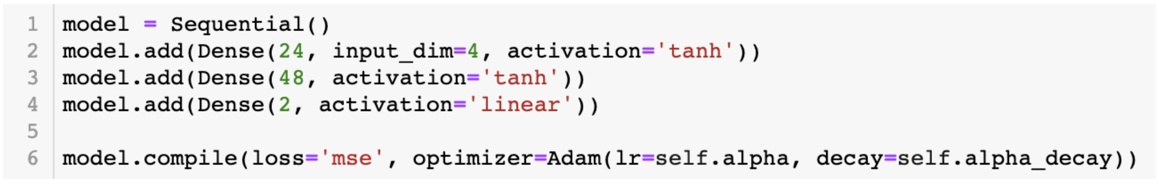
Table 1: Exemplar Q-Table

State	**Action**	**Q(state, action)**
...	...	...
...	...	...

With many entries and a long training time, the (state, action) space soon becomes very large. With problems which concern an environment with a continuous flow between states, it can be no longer possible to store all the information in this way since states which are only slightly different are still distinct states. This would actually be very exhausting to implement in our CartPole problem and certainly wouldn't be the most effective approach. It is possible to instead use something that can generalise the knowledge instead of storing and looking up every state. This can be done using neural networks.

Deep Q-Network

Using a neural network instead of a Q-Table to predict the Q-values of given (state, action) pairs is far more applicable. We need to build a neural network which can be used for this service, which can be done using a sequential model in the Keras package. The architecture of the neural net is designed to match the complexity of the problem, with two hidden layers, this is a deep neural network. The input and output number of nodes are the same dimension of the observation and action spaces they correspond to respectively.

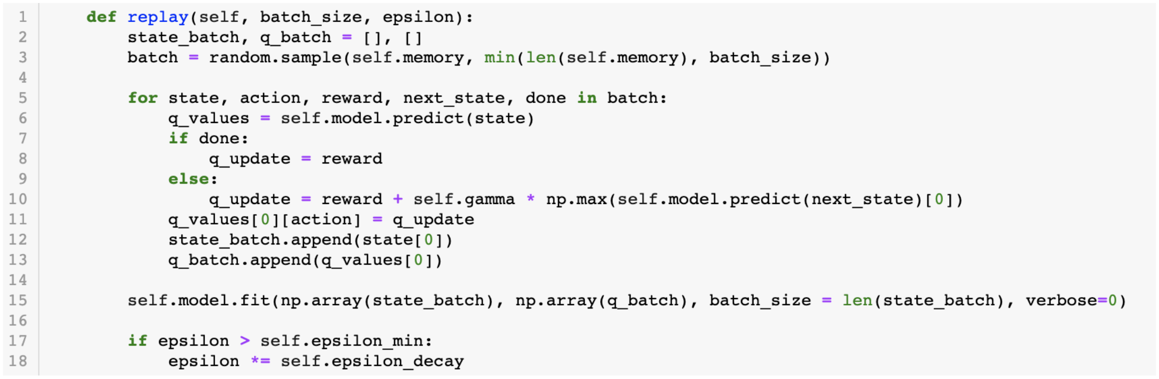


```python
model = Sequential()
model.add(Dense(24, input_dim=4, activation='tanh'))
model.add(Dense(48, activation='tanh'))
model.add(Dense(2, activation='linear'))

model.compile(loss='mse', optimizer=Adam(lr=self.alpha, decay=self.alpha_decay))
```

Figure 5: Deep neural network used for predicting Q-values.

This type of neural network is referred to as a Deep Q-Network (DQN) as we see it is a deep neural network which predicts Q-values. In order to improve the accuracy of the DQN it must be updated and fit to training data first before we use it to make predictions. In order to do this, we first want to remember the agent's states for each environment step and perform an experience replay at the end of an episode. Experience replay is where we sample from the memory and updates the Q-values for each entry. Let us see this in action and explain how this works in the code.



```python
1       def replay(self, batch_size, epsilon):
2           state_batch, q_batch = [], []
3           batch = random.sample(self.memory, min(len(self.memory), batch_size))
4   
5           for state, action, reward, next_state, done in batch:
6               q_values = self.model.predict(state)
7               if done:
8                   q_update = reward
9               else:
10                  q_update = reward + self.gamma * np.max(self.model.predict(next_state)[0])
11              q_values[0][action] = q_update
12              state_batch.append(state[0])
13              q_batch.append(q_values[0])
14  
15          self.model.fit(np.array(state_batch), np.array(q_batch), batch_size = len(state_batch), verbose=0)
16  
17          if epsilon > self.epsilon_min:
18              epsilon *= self.epsilon_decay
```

Figure 6: Experience replay, reinforcement learning in action.

Code Analysis:

Line 2: Define two empty lists, one to store the states and one to store the Q-values that belong to that state
Line 3: Sampling in this case 64 data entries from memory to examine
Line 5: Goes through each of the data entries from the random sample
Line 6: Uses model to predict the Q-value it would calculate for given past state
Line 7/8: Updates Q-value if it was the last state (i.e. no next state)
Line 9/10: Key equation for Q-Learning namely, the Bellman equation
Line 11: Replaces model's predicted Q-values for given state with calculated Q-values based on Bellman equation
Line 12/13: Adds state and new Q-values for each entrant to their respective lists
Line 15: Fits model to information about given inputs and required outputs, to make the model a better predictor of future Q-values
Line 17/18: Incrementally reduces epsilon by a scale-factor which is the reinforcement learning parameter.

Line 10 in the code for the replay function in figure 6, is vital in Q-Learning and is essentially the Bellman equation. We calculate the new Q-value by taking the maximum Q-value for a given action (predicted value of a best next state), multiplying by the discount factor γ and adding to the current state reward. This can be abbreviated as, updating our Q-value with the cumulative discounted future rewards.

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Figure 7: Bellman equation.

Run function

Now that we have designed the architecture for our neural network and created the mechanism which will drive the Q-learning in the network and improve our model, we are now ready to set up a run function. We can now define a function which chooses an action, steps the environment, remembers old state and turns next state into current state, then repeats.

Choose action

We really want to choose an action based on random sampling to begin with, this provides lots of training data early on for the model to fit to and improve with, then slowly turn to using the model's predictions to make decisions on which action to take. It is just a case of writing extra functions which control how you start by taking random actions to begin with to then using the model more, which was done under a choose action function and used the variable epsilon to control this transition.

Choose epsilon

In order to have the transition in the first place, the value of epsilon must be changing and this can be described in its own function which is a logarithmic decay down to a minimum value. Epsilon starts at a maximum value where initially the actions taken are random. As epsilon become smaller, the agent bases its decisions on which actions to take using the model more.

Remember

This is called every time the environment is stepped, recording the current state, action, reward and next state, and adding this information to memory.

Training parameters

All of the hyper-parameters have been chosen to be what they are on the basis of theoretical expectations. In reality, these parameters may not be perfect in giving a solution which converges the fastest and in practise, finding optimal hyper-parameters comes from testing different values through trial-and-error. The table below contains information for the values of the hyper-parameters used in this solution which proved to work as intended fairly effectively, but they are not necessarily optimal.

Table 2: Hyper-parameter values

Symbol	**Name**	**Value**
α	Learning rate	1.0
α_{decay}	Decay factor for Learning rate	0.01
γ	Discount factor	1.0
ϵ	Greedy factor	1.0
ϵ_{min}	Greedy factor minimum	0.01
ϵ_{decay}	Greedy factor decay	0.995

Learning performance

The code referenced throughout this document is available on https://github.com/JamesNunns/Robotics-Group-Studies and can be found from Robotics-Group-Studies → Machine Learning → Deep Q → keras → Deep_CartPole.py. Using the training parameters in Table 2, the following results were found through varying the activation functions.

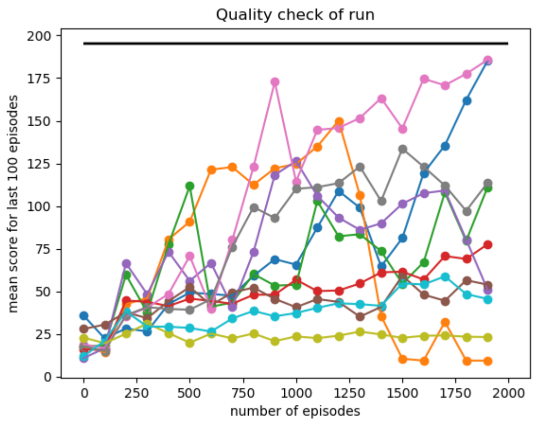


Figure 8: 'relu' for each hidden layer, 'relu' in output layer.

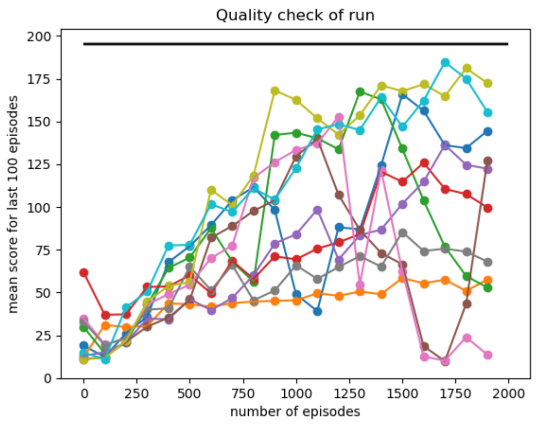


Figure 9: 'relu' for each hidden layer, 'linear' in output layer.

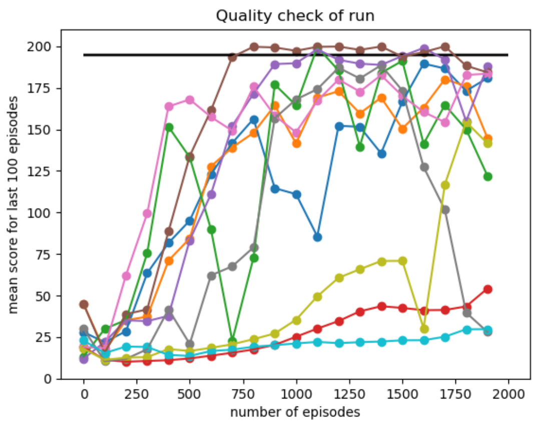


Figure 10: 'tanh' for each hidden layer, 'linear' in output layer.