University of Roehampton London

Machine Learning

Dr Changjiang He, Dr Kuo-Ming Chao
Computer Science| School of Art
University of Roehampton

University of Roehampton London

Lesson 10.1

Deep Q-Learning

Why Deep Q-learning?

University of Roehampton London

- Q-learning is a simple yet quite powerful algorithm to create a cheat sheet for our agent.
- This helps the agent figure out exactly which action to perform.
- But what if this cheat sheet (Q-table) is too long? Imagine an environment with 10,000 states and 1,000 actions per state.
- This would create a table of 10 million cells. Things will quickly get out of control!

Why Deep Q-learning?

It is pretty clear that we can't infer the Q-value of new states from already explored states. This presents two problems:

- First, the amount of memory required to save and update that table would increase as the number of states increases
- Second, the amount of time required to explore each state to create the required Q-table would be unrealistic

Why Deep Q-learning?

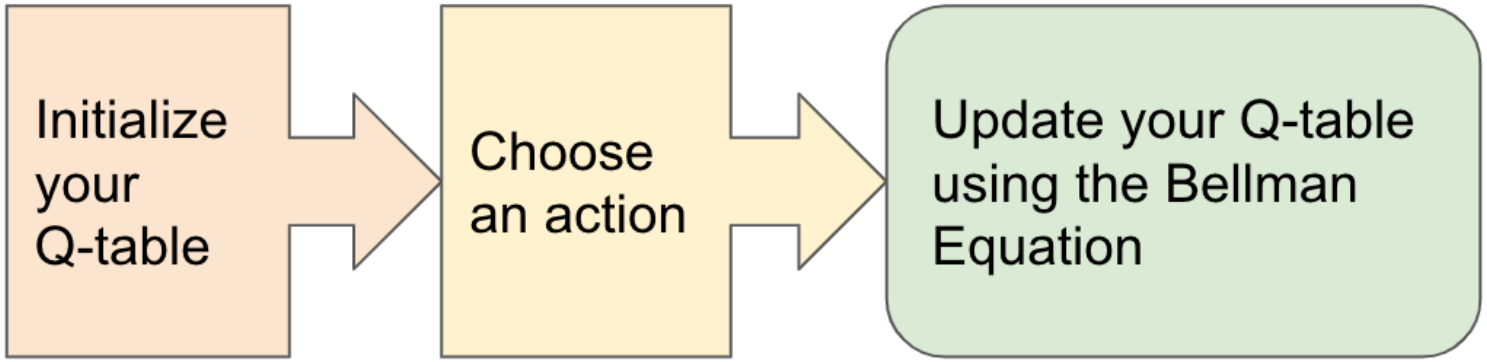
University of Roehampton London

- Here's a thought – what if we approximate these Q-values with machine learning models such as a neural network?
- Well, this was the idea behind DeepMind's algorithm that led to its acquisition by Google for 500 million dollars!

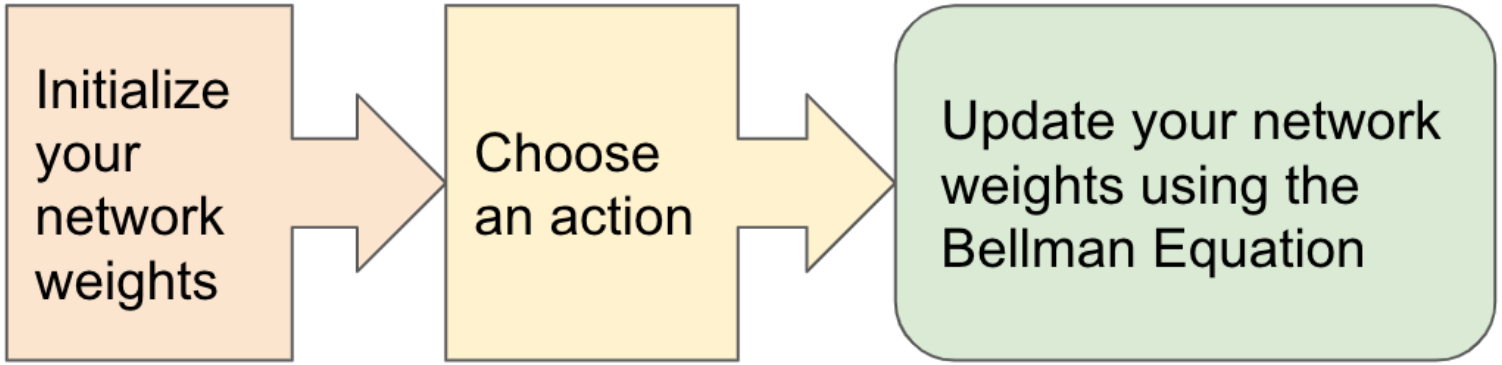
Basic Idea

University of Roehampton London

Vanilla Q-Learning: A table maps each state-action pair to its corresponding Q-value

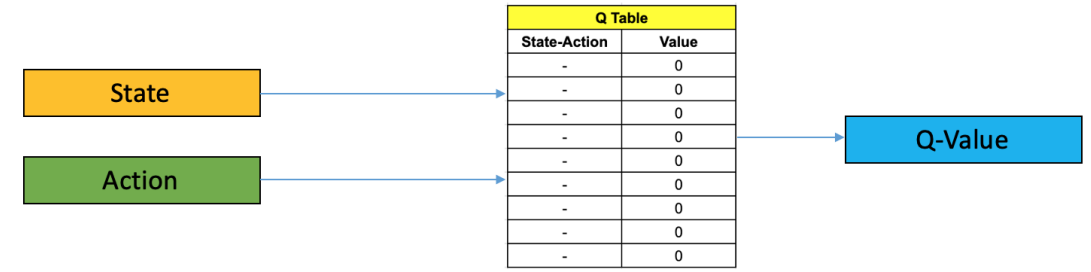


Deep Q-Learning: A Neural Network maps input states to (action, Q-value) pairs

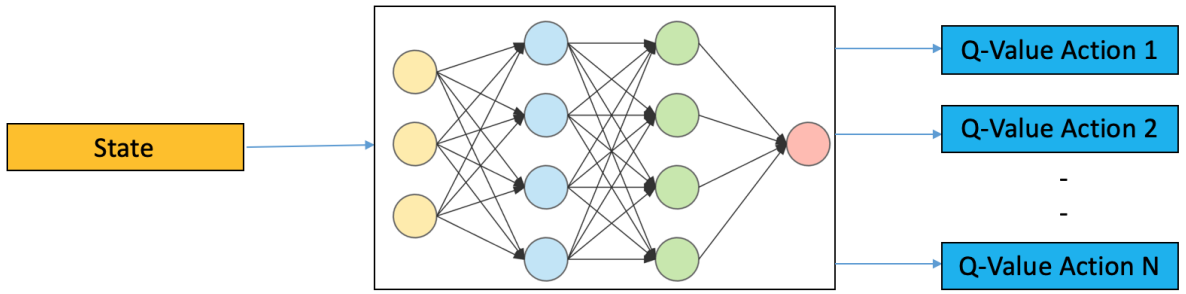


Deep Q-Network

- The state is given as the input and the Q-value of all possible actions is generated as the output.



Q Learning



Deep Q Learning

Initialize your Target and Main neural networks

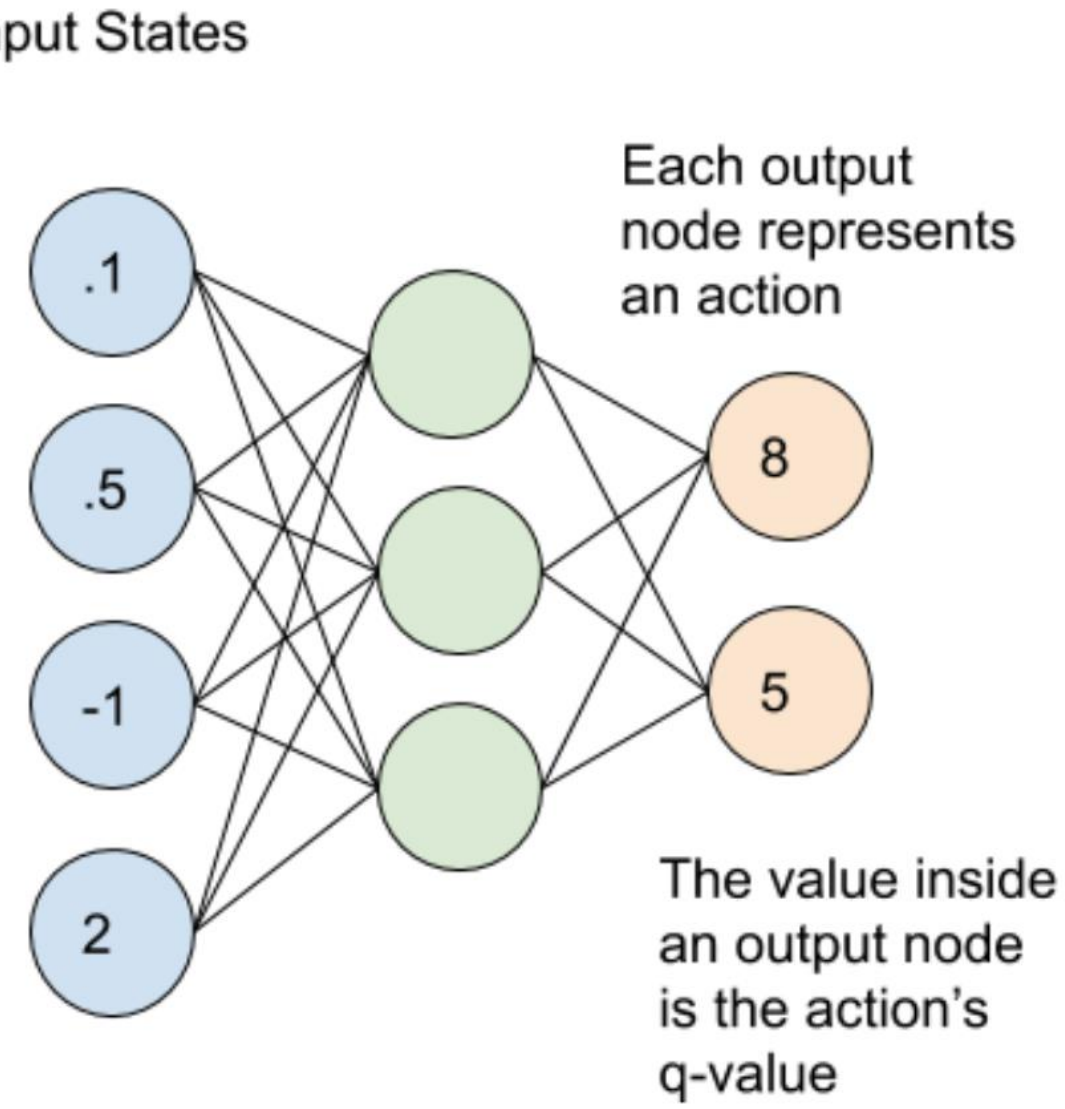
University of Roehampton London

- One of the interesting things about Deep Q-Learning is that the learning process uses 2 neural networks.
- These networks have the same architecture but different weights. Every N steps, the weights from the **main network** are copied to the **target network**.
- Using both of these networks leads to more stability in the learning process and helps the algorithm to learn more effectively.

Initialize your Target and Main neural networks

University of Roehampton London

- The main and target neural networks map input states to an (action, q-value) pair.
- In this case, each output node (representing an action) contains the action's q-value as a floating point number.
- Note that the output nodes do not represent a probability distribution so they will not add up to 1.



An Example of Neural Network in Deep Q-Learning

University of Roehampton London

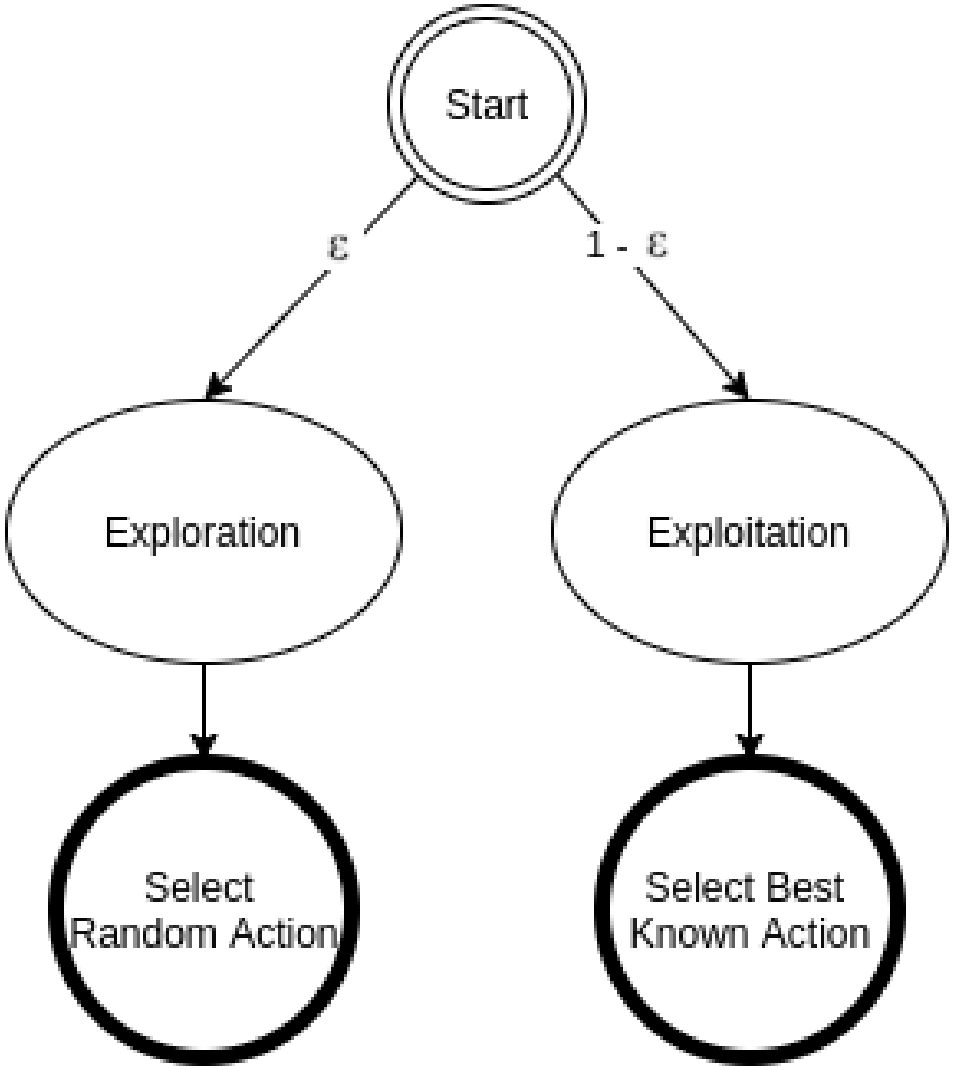
```
def agent(state_shape, action_shape):
    learning_rate = 0.001
    init = tf.keras.initializers.HeUniform()
    model = keras.Sequential()
    model.add(keras.layers.Dense(24, input_shape=state_shape,
activation='relu', kernel_initializer=init))
    model.add(keras.layers.Dense(12, activation='relu',
kernel_initializer=init))
    model.add(keras.layers.Dense(action_shape, activation='linear',
kernel_initializer=init))
    model.compile(loss=tf.keras.losses.Huber(),
optimizer=tf.keras.optimizers.Adam(lr=learning_rate), metrics=
['accuracy'])
    return model
```

Choose action using Epsilon-Greedy Exploration Strategy

University of Roehampton London

- Epsilon-Greedy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly.
- *Exploration* allows an agent to improve its current knowledge about each action, hopefully leading to long-term benefit.
- *Exploitation* on the other hand, chooses the greedy action to get the most reward by exploiting the agent's current action-value estimates.

Choose action using Epsilon-Greedy Exploration Strategy



Update your network weights using the Bellman Equation

University of Roehampton London

- After choosing an action, it's time for the agent to perform the action and update the Main and Target networks according to the Bellman equation.

- Deep Q-Learning agents use Experience Replay to learn about their environment and update the Main and Target networks.

- To summarize, the **main network** samples and trains on a batch of past experiences every 4 steps. The main network weights are then copied to the **target network** weights every 100 steps.

Update your network weights using the Bellman Equation

University of Roehampton London

- **Experience Replay** is the act of storing and replaying game states (the state, action, reward, next state) that the RL algorithm is able to learn from.

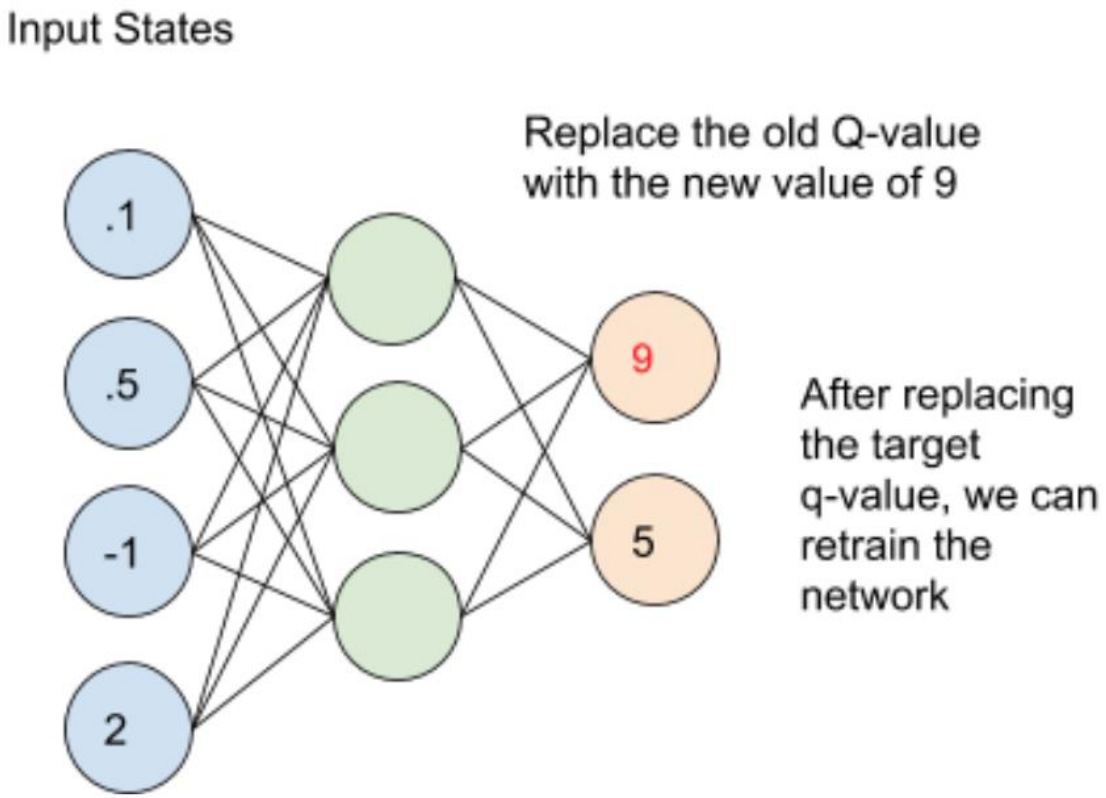
- Deep Q-Learning uses Experience Replay to learn in small **batches** in order to avoid skewing the dataset distribution of different states, actions, rewards, and next states that the neural network will see.

- Importantly, the agent doesn't need to train after each step.

Update your network weights using the Bellman Equation

University of Roehampton London

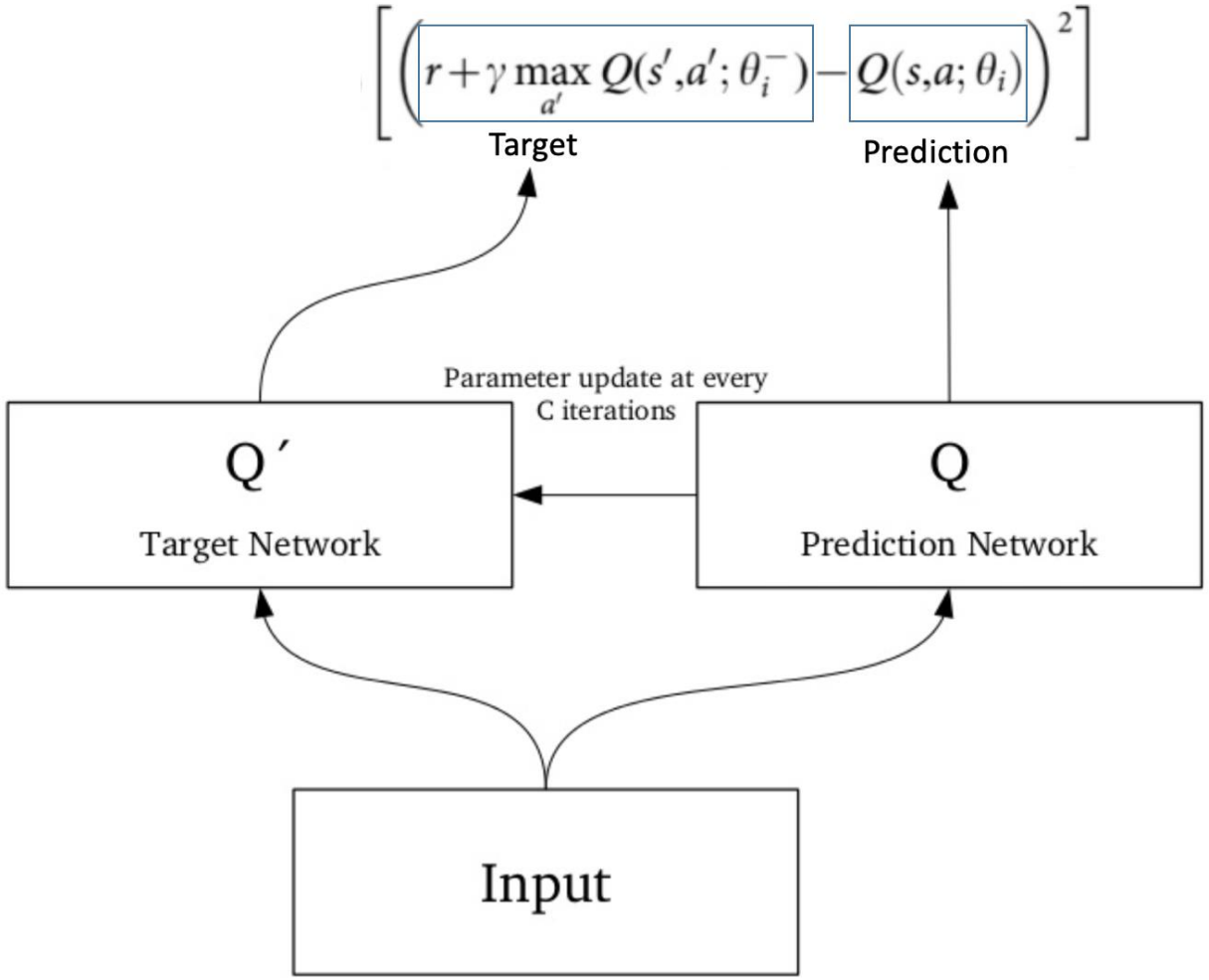
- Just like with vanilla Q-Learning, the agent still needs to update our model weights according to the Bellman Equation.



Note that the **target** network and not the main network is used to calculate the target.

Summary

- We use two separate networks with the same architecture to estimate the target and prediction.
- For every C iterations the parameters from the prediction network are copied to the target network.
- This leads to more stable training because it keeps the target function fixed (for a while).



Basic Idea

University of Roehampton London

Our Deep Q-Network implementation needed a few tricks before the agent started to learn to solve the CartPole problem effectively. Here are some of the tips and tricks that really helped.

1.Having the right model parameter update frequency is important. If you update model weights too often (e.g. after every step), the algorithm will learn very slowly when not much has changed. In this case, we perform main model weight updates every 4 steps which helps the algorithm to run significantly faster.

2.Setting the correct frequency to copy weights from the Main Network to the Target Network also helps improve learning performance. We initially tried to update the Target Network every N episodes which turned out to be less stable because the episodes can have a different number of steps. Sometimes there would be 10 steps in an episode and other times there could be 200 steps. We found that updating the Target Network every 100 steps seemed to work relatively well.

3.Using the Huber loss function rather than the Mean Squared Error loss function also helps the agent to learn. The Huber loss function weighs outliers less than the Mean Squared Error loss function.

4.The right initialization strategy seems to help. In this case, we use He Initialization for initializing network weights. He Initialization is a good initialization strategy for networks that use the Relu activation function.