**Deep Q-Network**

**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

**The Deep Q-Network Algorithm**

Figure 5: The Deep Q-Network Algorithm (Image by Author)

1. Initialize your Main and Target neural networks
2. Choose an action using the Epsilon-Greedy Exploration Strategy
3. Update your network weights using the Bellman Equation

**4a. Initialize your Target and Main neural networks**

A core difference between Deep Q-Learning and Vanilla Q-Learning is the implementation of the Q-table. Critically, Deep Q-Learning replaces the regular Q-table with a neural network. Rather than mapping a state-action pair to a q-value, a neural network maps input states to (action, Q-value) pairs.

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. In our implementation, the main network weights replace the target network weights every 100 steps.

**How to map States to (Action, Q-value) pairs**

Figure 6: A neural network mapping an input state to its corresponding (action, q-value) pair (Image by Author)

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. For the example above, one action has a Q-value of 8 and the other action has a Q-value of 5.

**Defining our network architecture**

**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

In our implementation, the main and target networks are quite simple consisting of 3 densely connected layers with Relu activation functions. The most notable features are that we use He uniform initialization as well as the Huber loss function to achieve better performance.

**4b. Choose an action using the Epsilon-Greedy Exploration Strategy**

In the Epsilon-Greedy Exploration strategy, the agent chooses a random action with probability **epsilon** and exploits the best known action with probability **1 — epsilon**.

**How do you find the best known action from your network?**

Both the Main model and the Target model map input states to output actions. These output actions actually represent the model’s predicted Q-value. In this case, the action that has the largest predicted Q-value is the best known action at that state.

**4c. Update your network weights using the Bellman Equation**

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.

**Experience Replay**

**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. Experience Replay can be used in **Off-Policy** algorithms to learn in an offline fashion. Off-policy methods are able to update the algorithm’s parameters using saved and stored information from previously taken actions. 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. In our implementation, we use Experience Replay to train on small batches once every 4 steps rather than every single step. We found this trick to really help speed up our Deep Q-Learning implementation.

**Bellman Equation**

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

Figure 7: Updating the neural network with the new Temporal Difference target using the Bellman equation (Image by Author)

Figure 8: The Temporal Difference target we want to replicate using our neural network (Image by Author)

From the original Bellman equation in Figure 3, we want to replicate the **Temporal Difference target** operation using our neural network rather than using a Q-table. Note that the **target** network and not the main network is used to calculate the Temporal Difference target. Assuming that the temporal difference target operation produces a value of 9 in the example above, we can update the **main** network weights by assigning 9 to the target q-value and fitting our **main** network weights to the new target values.

**5. Tips and Tricks (**[**source**](https://towardsdatascience.com/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756)**)**

Our Deep Q-Network implementation needed a few tricks before the agent started to learn to solve the CartPole problem effectively. [Here](https://towardsdatascience.com/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756) 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](https://towardsdatascience.com/all-ways-to-initialize-your-neural-network-16a585574b52) for networks that use the Relu activation function.

**6. Deep Q-Network Coding Implementation**

Putting it all together, you can find our minimal Deep Q-Network implementation solving the CartPole problem [here](https://github.com/mswang12/minDQN/blob/main/minDQN.py). This implementation uses Tensorflow and Keras and should generally run in less than 15