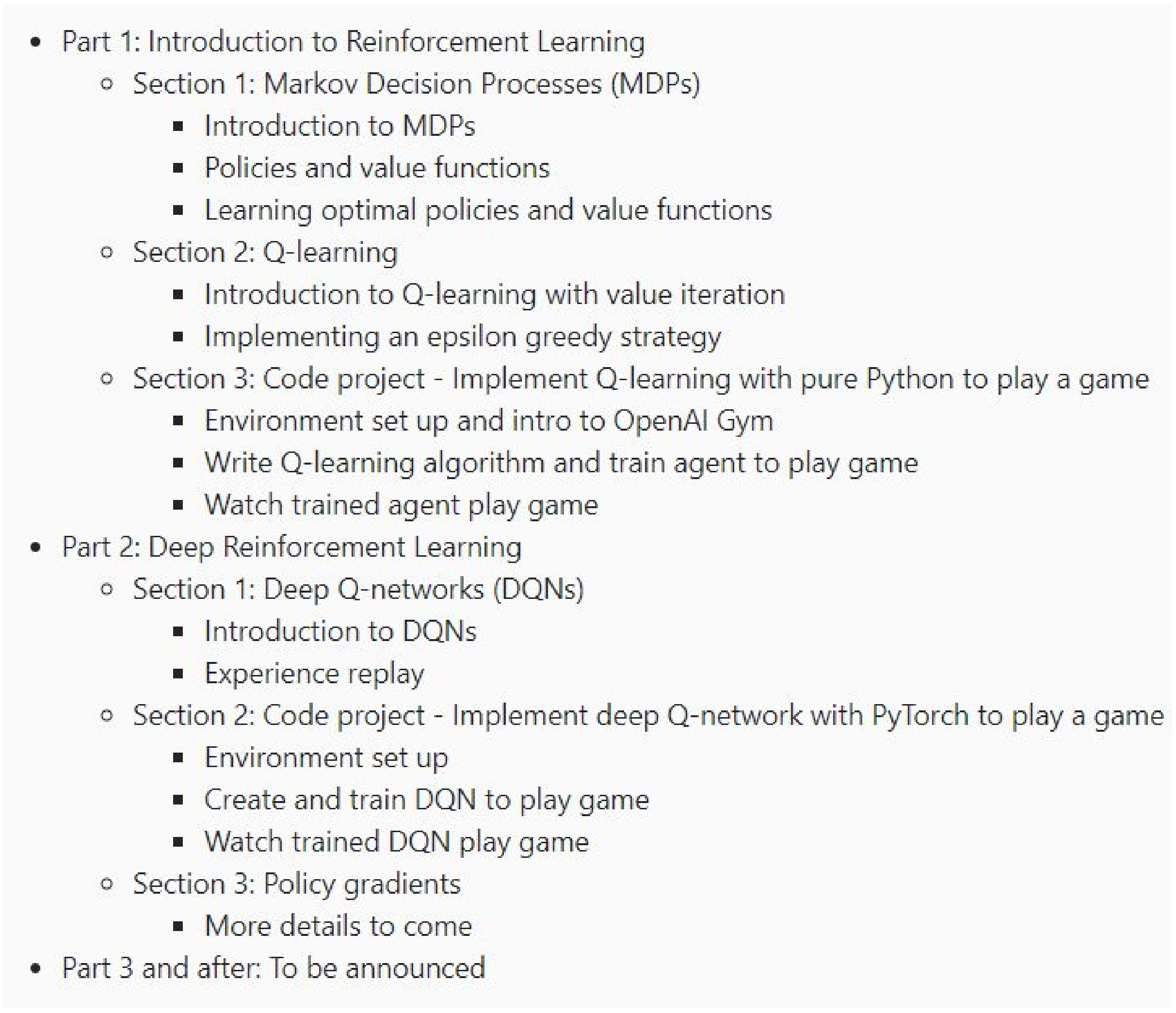
Source: http://deeplizard.com

Tutorial 1: Syllabus overview

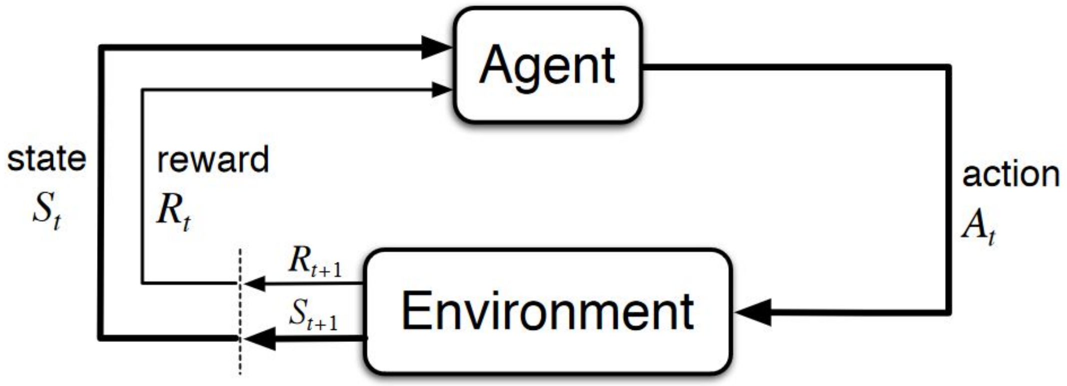


- Part 1: Introduction to Reinforcement Learning
  - Section 1: Markov Decision Processes (MDPs)
    - Introduction to MDPs
    - Policies and value functions
    - Learning optimal policies and value functions
  - Section 2: Q-learning
    - Introduction to Q-learning with value iteration
    - Implementing an epsilon greedy strategy
  - Section 3: Code project - Implement Q-learning with pure Python to play a game
    - Environment set up and intro to OpenAI Gym
    - Write Q-learning algorithm and train agent to play game
    - Watch trained agent play game
- Part 2: Deep Reinforcement Learning
  - Section 1: Deep Q-networks (DQNs)
    - Introduction to DQNs
    - Experience replay
  - Section 2: Code project - Implement deep Q-network with PyTorch to play a game
    - Environment set up
    - Create and train DQN to play game
    - Watch trained DQN play game
  - Section 3: Policy gradients
    - More details to come
- Part 3 and after: To be announced

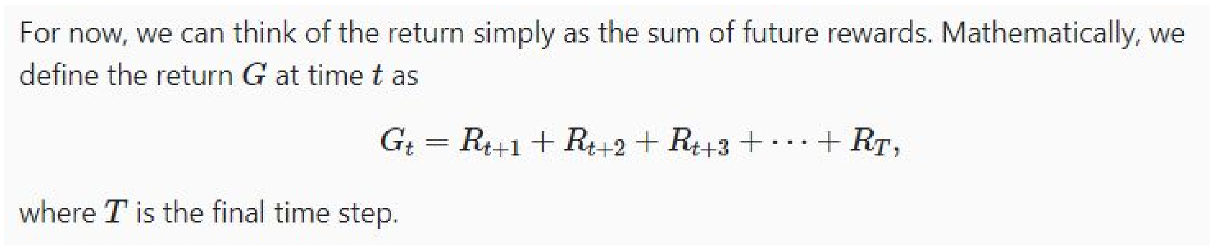
Tutorial 2: Markov Decision Process

Components of an MDP:

- Agent
- Environment
- State
- Action
- Reward



Tutorial 3: Expected Return



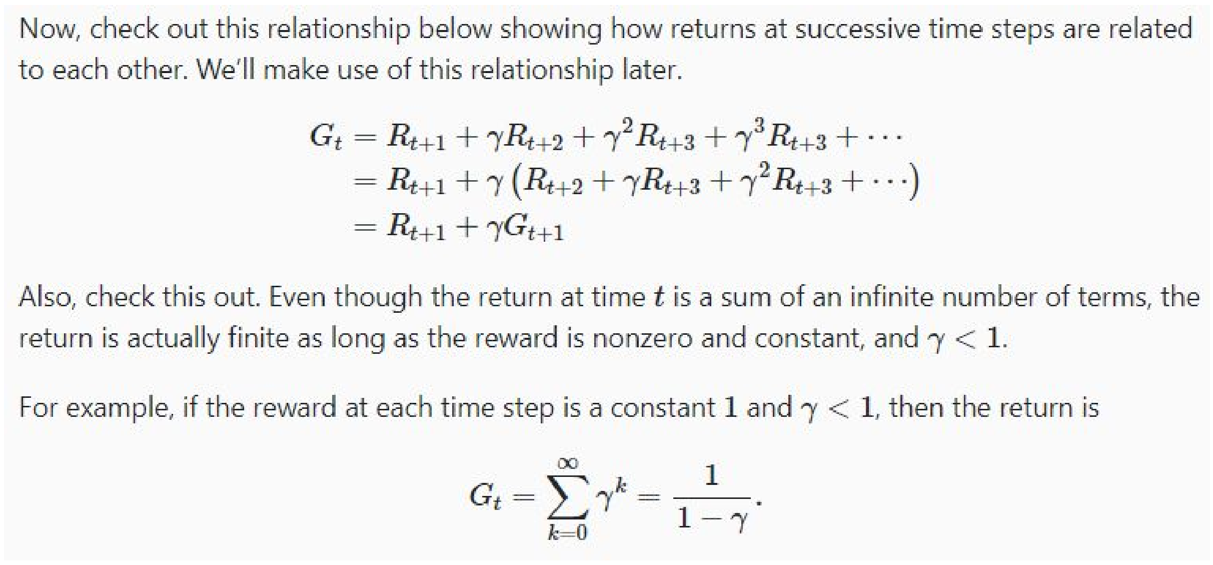
For now, we can think of the return simply as the sum of future rewards. Mathematically, we define the return G at time t as

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$

where T is the final time step.

Episodic vs. continuing tasks

Discounted return



Now, check out this relationship below showing how returns at successive time steps are related to each other. We'll make use of this relationship later.

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+3} + \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+3} + \cdots \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

Also, check this out. Even though the return at time t is a sum of an infinite number of terms, the return is actually finite as long as the reward is nonzero and constant, and $\gamma < 1$.

For example, if the reward at each time step is a constant 1 and $\gamma < 1$, then the return is

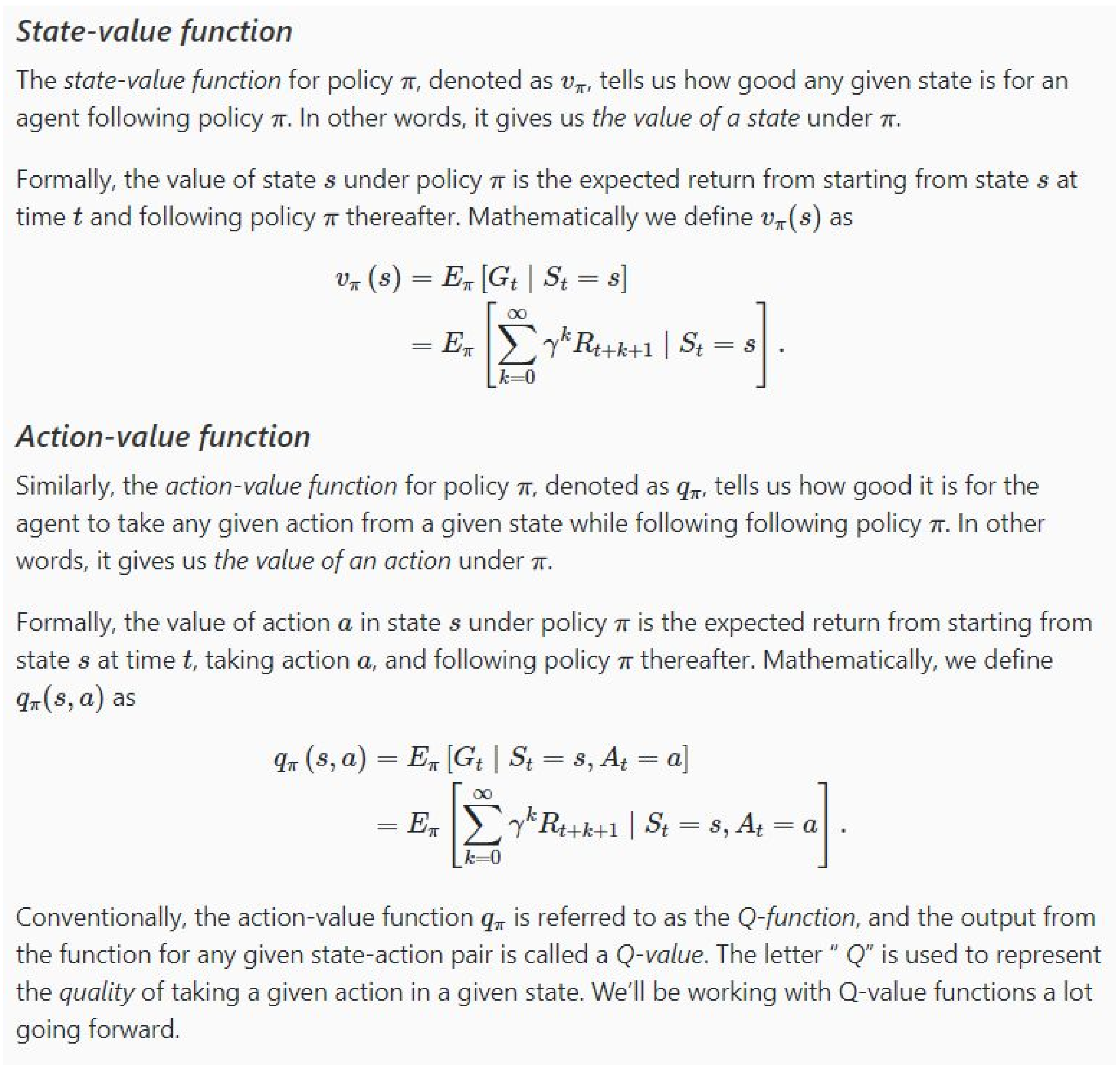
$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}.$$

<u>Tutorial 4:</u> Policies and value functions

Two things;

1. <u>Policies</u> [$\pi(a|s)$]: Whether the agent takes an action or not, compares probability of each action on a state

2. <u>Value functions</u> [$v_\pi(s)$]: How good the action is, compares the expected return for each action on a state



State-value function

The *state-value function* for policy π, denoted as v_π, tells us how good any given state is for an agent following policy π. In other words, it gives us *the value of a state* under π.

Formally, the value of state s under policy π is the expected return from starting from state s at time t and following policy π thereafter. Mathematically we define $v_\pi(s)$ as

$$\begin{aligned} v_\pi(s) &= E_\pi\left[G_t \mid S_t = s\right] \\ &= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right]. \end{aligned}$$

Action-value function

Similarly, the *action-value function* for policy π, denoted as q_π, tells us how good it is for the agent to take any given action from a given state while following following policy π. In other words, it gives us *the value of an action* under π.

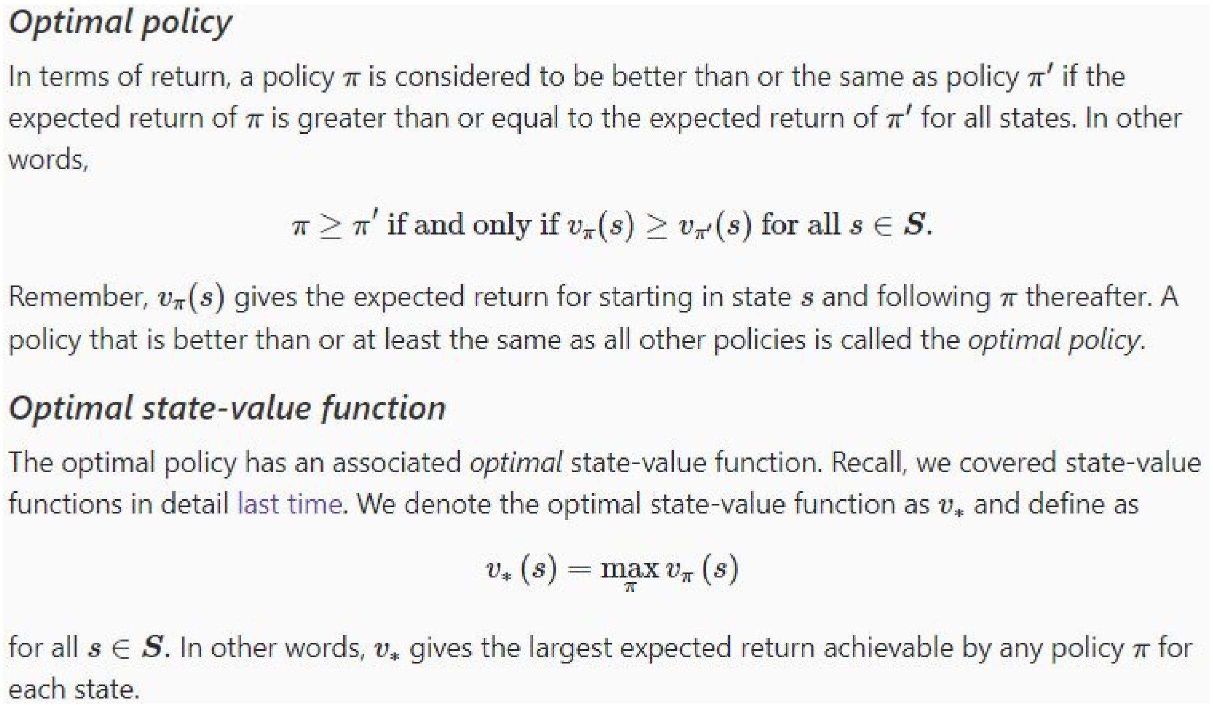
Formally, the value of action a in state s under policy π is the expected return from starting from state s at time t, taking action a, and following policy π thereafter. Mathematically, we define $q_\pi(s,a)$ as

$$\begin{aligned} q_\pi(s,a) &= E_\pi\left[G_t \mid S_t = s, A_t = a\right] \\ &= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]. \end{aligned}$$

Conventionally, the action-value function q_π is referred to as the *Q-function*, and the output from the function for any given state-action pair is called a *Q-value*. The letter " *Q*" is used to represent the *quality* of taking a given action in a given state. We'll be working with Q-value functions a lot going forward.

<u>Tutorial 5:</u> What do reinforcement learning algorithms learn

Optimal policies, optimal value functions



Optimal policy

In terms of return, a policy π is considered to be better than or the same as policy π' if the expected return of π is greater than or equal to the expected return of π' for all states. In other words,

$$\pi \geq \pi' \text{ if and only if } v_{\pi}(s) \geq v_{\pi'}(s) \text{ for all } s \in \boldsymbol{S}.$$

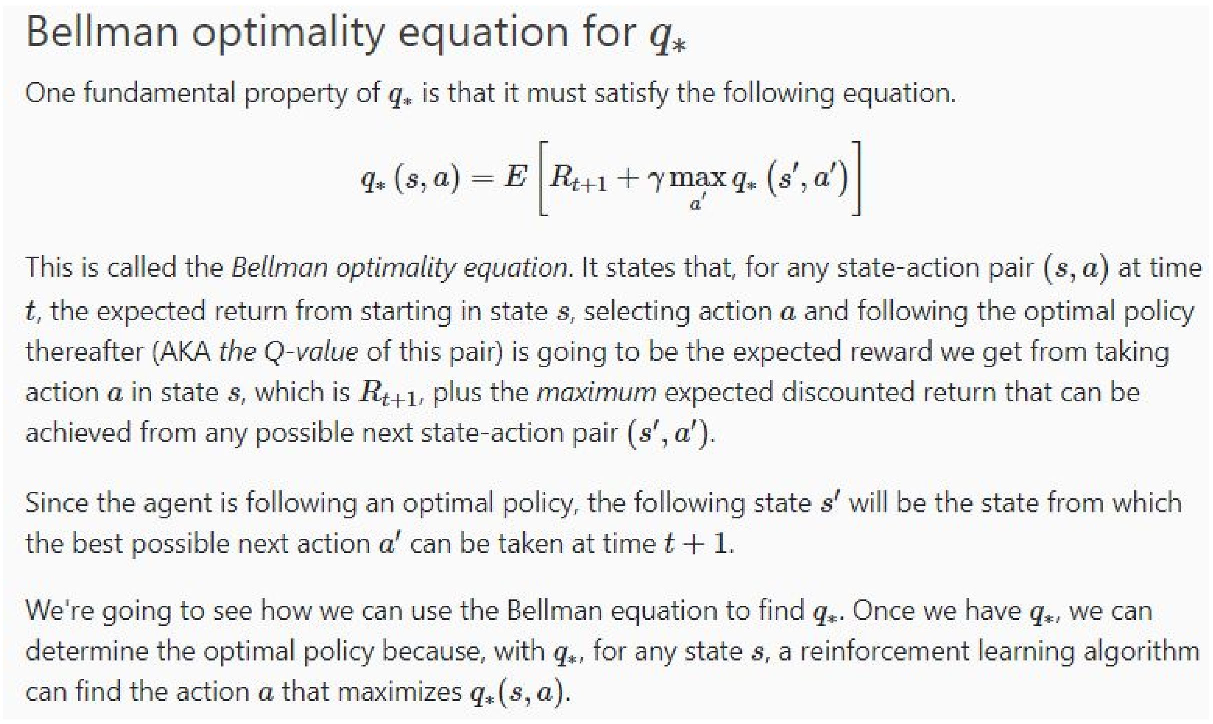
Remember, $v_{\pi}(s)$ gives the expected return for starting in state s and following π thereafter. A policy that is better than or at least the same as all other policies is called the *optimal policy*.

Optimal state-value function

The optimal policy has an associated *optimal* state-value function. Recall, we covered state-value functions in detail last time. We denote the optimal state-value function as v_* and define as

$$v_*(s) = \max_{\pi} v_{\pi}(s)$$

for all $s \in \boldsymbol{S}$. In other words, v_* gives the largest expected return achievable by any policy π for each state.



Bellman optimality equation for q_*

One fundamental property of q_* is that it must satisfy the following equation.

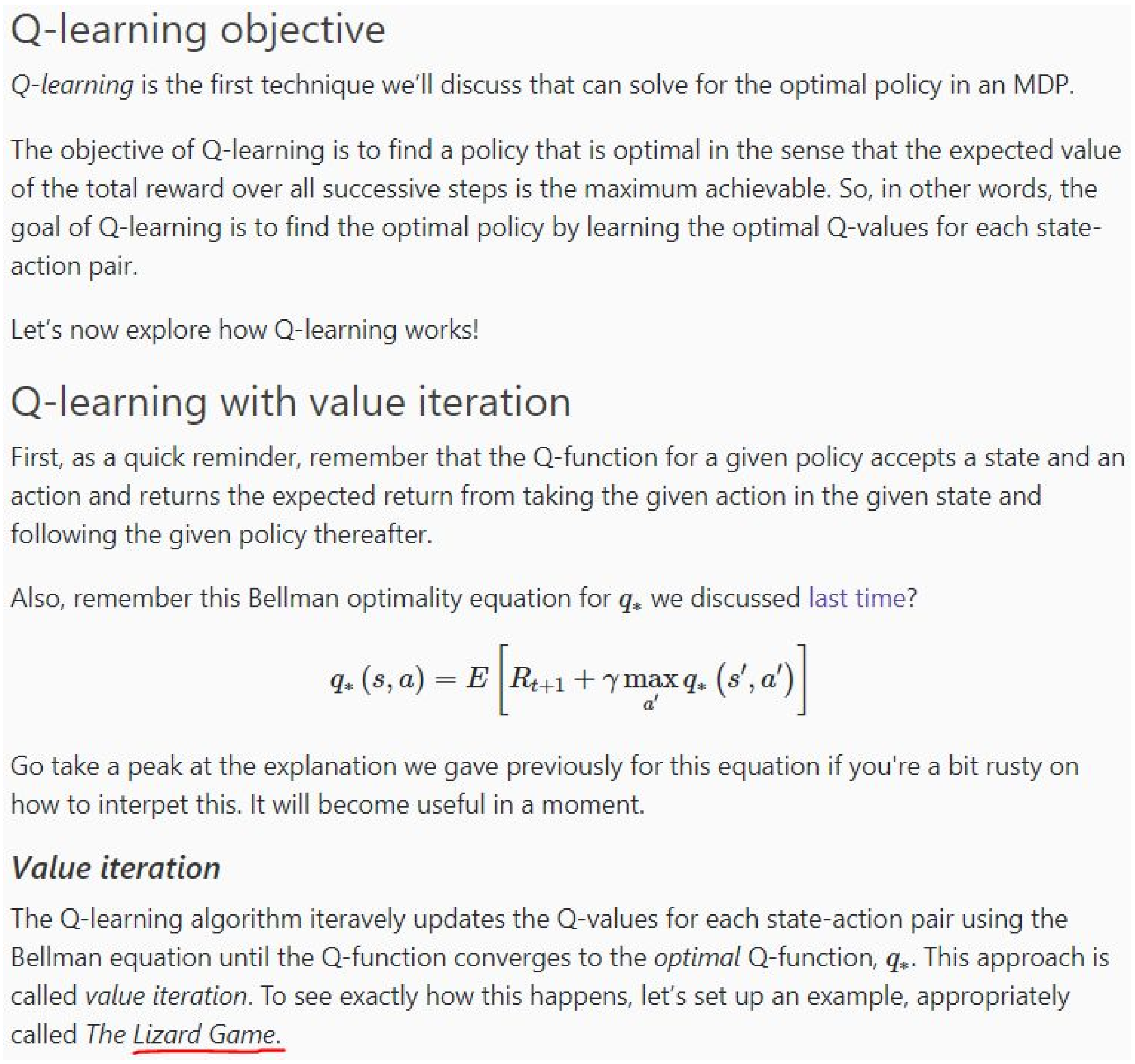
$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right]$$

This is called the *Bellman optimality equation*. It states that, for any state-action pair (s,a) at time t, the expected return from starting in state s, selecting action a and following the optimal policy thereafter (AKA *the Q-value* of this pair) is going to be the expected reward we get from taking action a in state s, which is R_{t+1}, plus the *maximum* expected discounted return that can be achieved from any possible next state-action pair (s',a').

Since the agent is following an optimal policy, the following state s' will be the state from which the best possible next action a' can be taken at time $t+1$.

We're going to see how we can use the Bellman equation to find q_*. Once we have q_*, we can determine the optimal policy because, with q_*, for any state s, a reinforcement learning algorithm can find the action a that maximizes $q_*(s,a)$.

Tutorial 6: Q-learning and Q-table



Q-learning objective

Q-learning is the first technique we'll discuss that can solve for the optimal policy in an MDP.

The objective of Q-learning is to find a policy that is optimal in the sense that the expected value of the total reward over all successive steps is the maximum achievable. So, in other words, the goal of Q-learning is to find the optimal policy by learning the optimal Q-values for each state-action pair.

Let's now explore how Q-learning works!

Q-learning with value iteration

First, as a quick reminder, remember that the Q-function for a given policy accepts a state and an action and returns the expected return from taking the given action in the given state and following the given policy thereafter.

Also, remember this Bellman optimality equation for q_* we discussed last time?

$$q_*\left(s,a\right) = E\left[R_{t+1} + \gamma \max_{a'} q_*\left(s',a'\right)\right]$$

Go take a peak at the explanation we gave previously for this equation if you're a bit rusty on how to interpet this. It will become useful in a moment.

Value iteration

The Q-learning algorithm iteravely updates the Q-values for each state-action pair using the Bellman equation until the Q-function converges to the *optimal* Q-function, q_*. This approach is called *value iteration*. To see exactly how this happens, let's set up an example, appropriately called *The Lizard Game.*

<u>Tutorial 7:</u> Exploration vs exploitation

Make move to maximize q-value

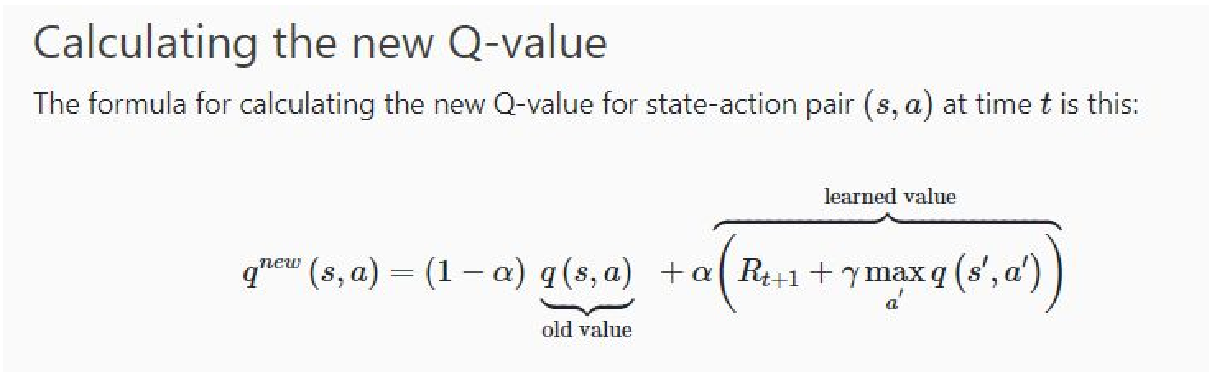
What if two moves have the same q-value?

Epsilon greedy strategy- balance between exploration and exploitation

Choosing an action based on maximum q-val

Updating the q-val

Using learning rate- how much the learner is likely to adopt new result



Calculating the new Q-value

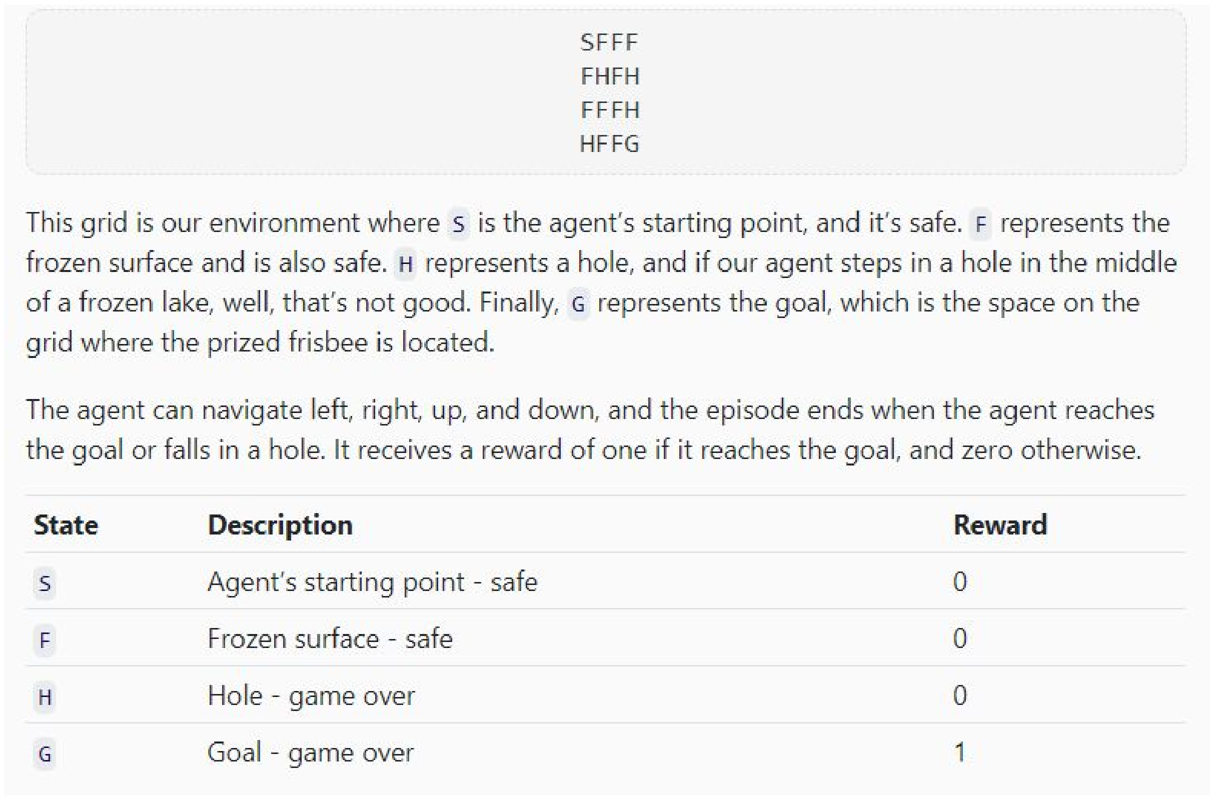
The formula for calculating the new Q-value for state-action pair (s, a) at time t is this:

$$q^{new}(s,a) = (1-\alpha)\ \underbrace{q(s,a)}_{\text{old value}}\ + \alpha \overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{learned value}}$$

We can specify maximum number of steps

<u>Tutorial 8:</u> OpenAI Gym and Python for Q-learning

Frozen lake



```
SFFF
FHFH
FFFH
HFFG
```

This grid is our environment where `S` is the agent's starting point, and it's safe. `F` represents the frozen surface and is also safe. `H` represents a hole, and if our agent steps in a hole in the middle of a frozen lake, well, that's not good. Finally, `G` represents the goal, which is the space on the grid where the prized frisbee is located.

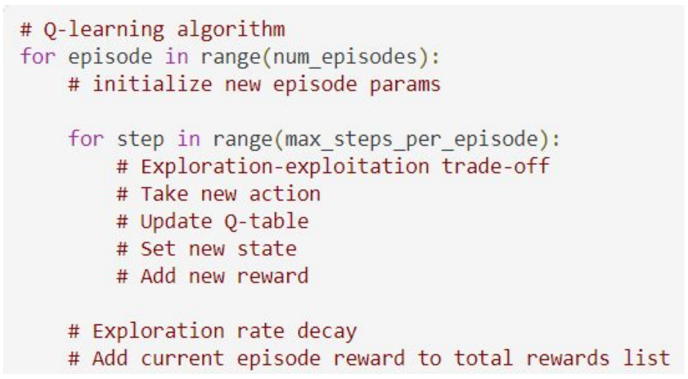
The agent can navigate left, right, up, and down, and the episode ends when the agent reaches the goal or falls in a hole. It receives a reward of one if it reaches the goal, and zero otherwise.

State	Description	Reward
`S`	Agent's starting point - safe	0
`F`	Frozen surface - safe	0
`H`	Hole - game over	0
`G`	Goal - game over	1

Creating the Q-table with numpy

Initializing Q-learning parameters

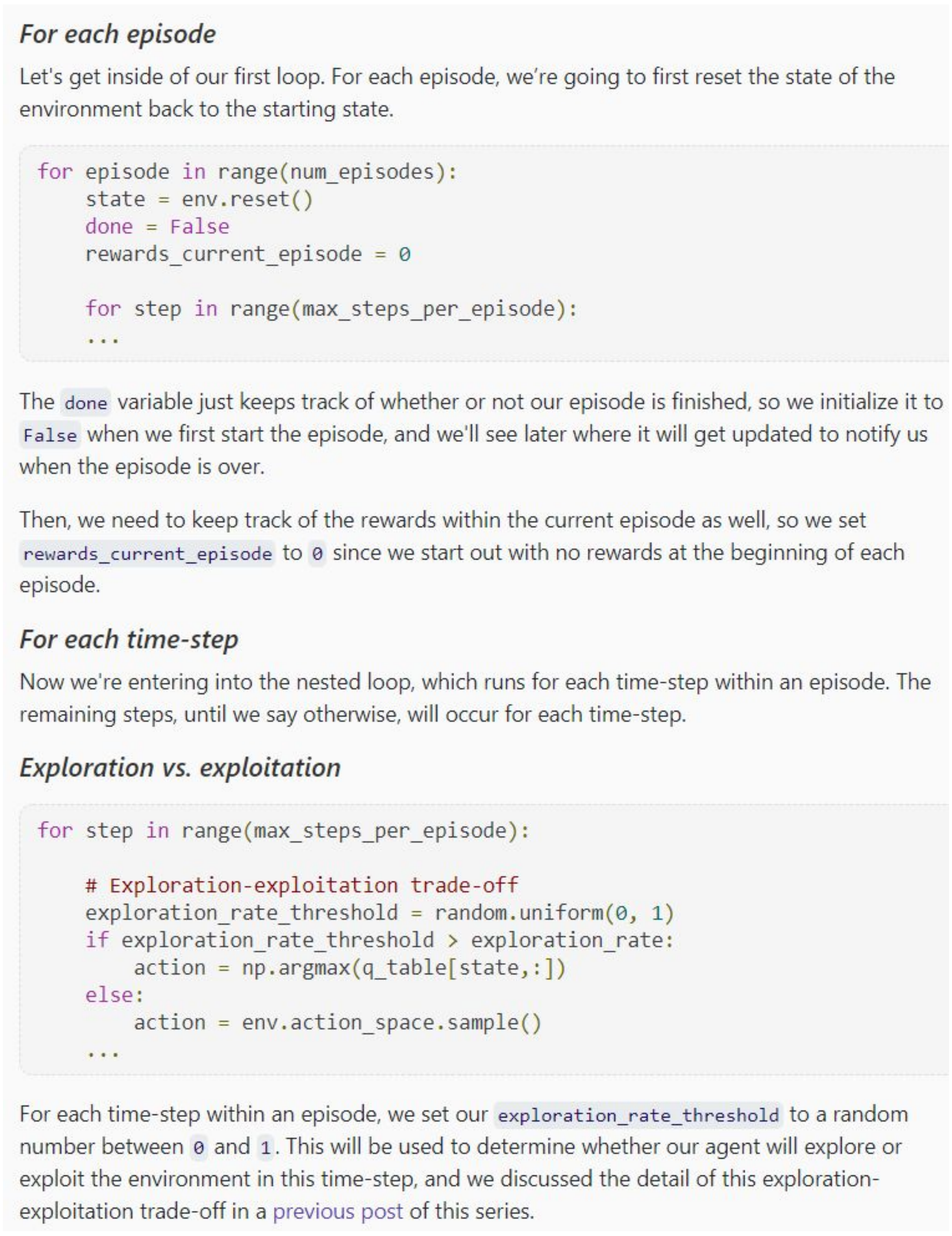
Tutorial 9: Train Q-learning Agent with Python



```
# Q-learning algorithm
for episode in range(num_episodes):
    # initialize new episode params

    for step in range(max_steps_per_episode):
        # Exploration-exploitation trade-off
        # Take new action
        # Update Q-table
        # Set new state
        # Add new reward

    # Exploration rate decay
    # Add current episode reward to total rewards list
```



For each episode

Let's get inside of our first loop. For each episode, we're going to first reset the state of the environment back to the starting state.

```
for episode in range(num_episodes):
    state = env.reset()
    done = False
    rewards_current_episode = 0

    for step in range(max_steps_per_episode):
    ...
```

The `done` variable just keeps track of whether or not our episode is finished, so we initialize it to `False` when we first start the episode, and we'll see later where it will get updated to notify us when the episode is over.

Then, we need to keep track of the rewards within the current episode as well, so we set `rewards_current_episode` to `0` since we start out with no rewards at the beginning of each episode.

For each time-step

Now we're entering into the nested loop, which runs for each time-step within an episode. The remaining steps, until we say otherwise, will occur for each time-step.

Exploration vs. exploitation

```
for step in range(max_steps_per_episode):

    # Exploration-exploitation trade-off
    exploration_rate_threshold = random.uniform(0, 1)
    if exploration_rate_threshold > exploration_rate:
        action = np.argmax(q_table[state,:])
    else:
        action = env.action_space.sample()
    ...
```

For each time-step within an episode, we set our `exploration_rate_threshold` to a random number between `0` and `1`. This will be used to determine whether our agent will explore or exploit the environment in this time-step, and we discussed the detail of this exploration-exploitation trade-off in a previous post of this series.

Tutorial 10: Watch Q-learning Agent Play Game with Python

```python
# Watch our agent play Frozen Lake by playing the best action
# from each state according to the Q-table

for episode in range(3):
    # initialize new episode params

    for step in range(max_steps_per_episode):
        # Show current state of environment on screen
        # Choose action with highest Q-value for current state
        # Take new action

        if done:
            if reward == 1:
                # Agent reached the goal and won episode
            else:
                # Agent stepped in a hole and lost episode

        # Set new state

env.close()
```

<u>Tutorial 11:</u> Deep Q-Learning

Normal Q-network takes a long time to traverse all those states if number of states increase

Deep Q-network: replace Q-network with deep neural network

Deep Q-network

Input- states

Layers- many deep-Q networks are purely just some convolutional layers, followed by some non-linear activation function, and then the convolutional layers are followed by a couple fully connected layers

Output- Q-values for different actions

Tutorial 12: Replay memory explained

Replay memory explained

Experience replay and replay memory

we store the agent’s experiences at each time step in a data set called the*replay memory*

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

Actually we will save a finite number of experiences in replay memory

Why use experience replay?

To break the correlation between consecutive samples

Combining a deep Q-network with experience replay

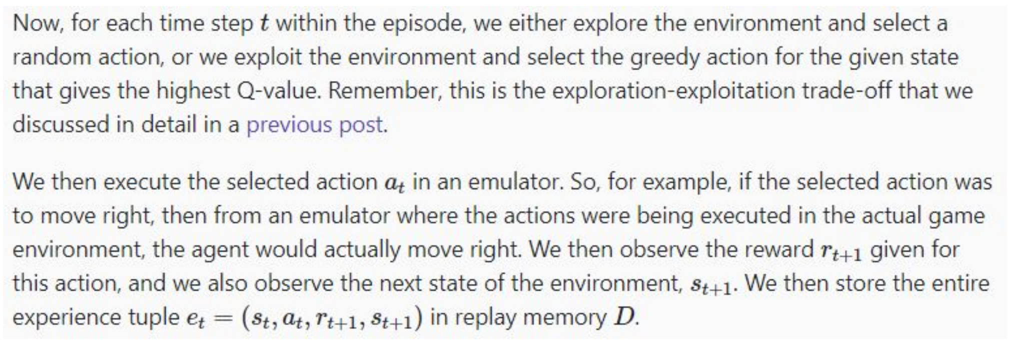
Setting up

Initialize the replay memory data set D to capacity N

Initialize the network with random weights

For each episode, initialize the starting state of the episode.

Gaining experience

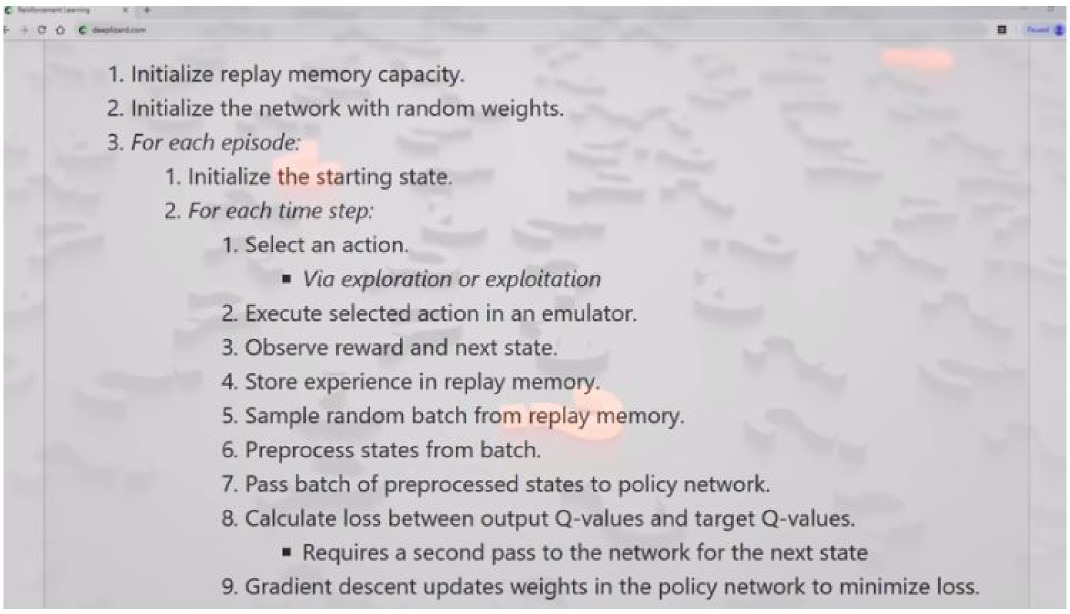


Now, for each time step t within the episode, we either explore the environment and select a random action, or we exploit the environment and select the greedy action for the given state that gives the highest Q-value. Remember, this is the exploration-exploitation trade-off that we discussed in detail in a previous post.

We then execute the selected action a_t in an emulator. So, for example, if the selected action was to move right, then from an emulator where the actions were being executed in the actual game environment, the agent would actually move right. We then observe the reward r_{t+1} given for this action, and we also observe the next state of the environment, s_{t+1}. We then store the entire experience tuple $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D.

<u>Tutorial 13:</u> Training a deep Q-network

Summary of the whole process:



```
1. Initialize replay memory capacity.
2. Initialize the network with random weights.
3. For each episode:
    1. Initialize the starting state.
    2. For each time step:
        1. Select an action.
            ▪ Via exploration or exploitation
        2. Execute selected action in an emulator.
        3. Observe reward and next state.
        4. Store experience in replay memory.
        5. Sample random batch from replay memory.
        6. Preprocess states from batch.
        7. Pass batch of preprocessed states to policy network.
        8. Calculate loss between output Q-values and target Q-values.
            ▪ Requires a second pass to the network for the next state
        9. Gradient descent updates weights in the policy network to minimize loss.
```

Training a Deep Q-Network - Reinforcement Learning

Tutorial 14: Training a deep Q-network with fixed Q-targets

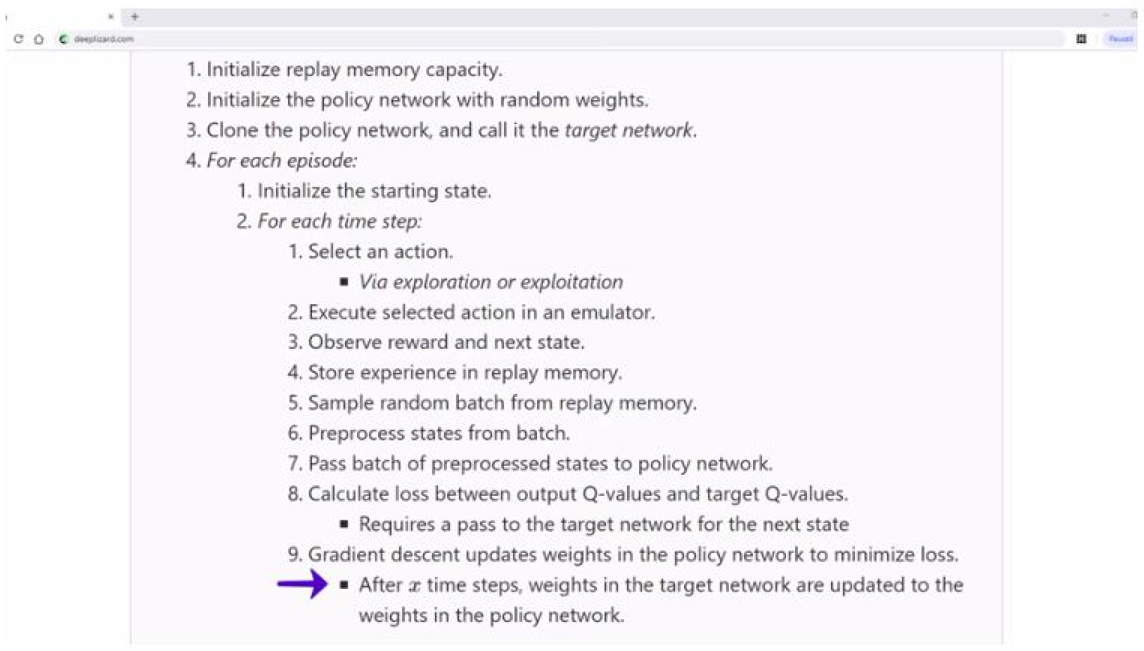
Problem if used the same Q-network: chasing its own tail, so unstability occurs in the optimization

Change from the previous tutorial:

On point 3

On point 8

On point 9



```
1. Initialize replay memory capacity.
2. Initialize the policy network with random weights.
3. Clone the policy network, and call it the *target network*.
4. *For each episode:*
    1. Initialize the starting state.
    2. *For each time step:*
        1. Select an action.
            ▪ *Via exploration or exploitation*
        2. Execute selected action in an emulator.
        3. Observe reward and next state.
        4. Store experience in replay memory.
        5. Sample random batch from replay memory.
        6. Preprocess states from batch.
        7. Pass batch of preprocessed states to policy network.
        8. Calculate loss between output Q-values and target Q-values.
            ▪ Requires a pass to the target network for the next state
        9. Gradient descent updates weights in the policy network to minimize loss.
            ▪ After $x$ time steps, weights in the target network are updated to the
              weights in the policy network.
```