6.
Deep Q-Learning

Q-Learning Algorithm

- The Q-learning algorithm is an adaptation of the Q-value iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown.
  - Assuming $a \underset{\alpha}{\leftarrow} b$ means $a_{k+1} \leftarrow (1-\alpha) \cdot a_k + \alpha . b_k$, we have:

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma . \max_{a'} Q(s',a')$$

- Q-learning works by watching an agent play (e.g., randomly) and gradually improving its estimates of the Q-values.
- Once it has accurate Q-value estimates, then the optimal policy is just choosing the action that has the highest Q-value (the greedy policy).

Implementing the Q-Learning Algorithm

- Exploring the environment: we need a `step` function so that the agent can execute one action and get the resulting state and reward:

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

- Exploration policy: since the state space is small, a simple random policy will be sufficient.
  - If we run the algorithm for long enough, the agent will visit every state many times, and it will also try every possible action many times.

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

Implementing the Q-Learning Algorithm

- Initialize the Q-values:

```
np.random.seed(42)
Q_values = np.full((3, 3), -np.inf)
for state, actions in enumerate(possible_actions):
    Q_values[state][actions] = 0
```

- Run the Q-learning algorithm with learning rate decay:

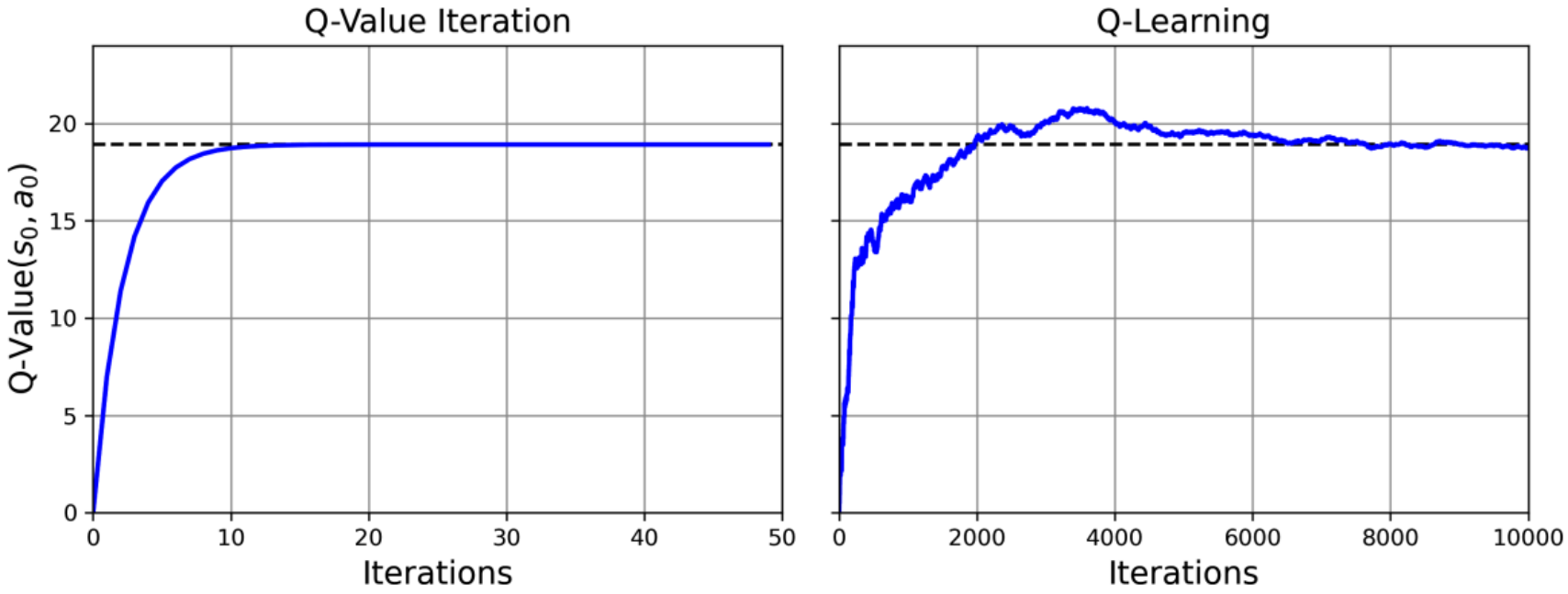
```
alpha0 = 0.05  # initial learning rate
decay = 0.005  # learning rate decay
gamma = 0.90  # discount factor
state = 0  # initial state

for iteration in range(10_000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = Q_values[next_state].max()  # greedy policy at the next step
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state
```

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma . \max_{a'} Q(s',a')$$

Learning Curve

- The Q-learning algorithm will converge to the optimal Q-values, but it will take many iterations, and possibly quite a lot of hyperparameter tuning.



Off-policy Algorithm

- The Q-learning algorithm is called an *off-policy* algorithm because the policy being trained is not necessarily the one used during training.
  - In the code, the policy being executed (the exploration policy) was random, while the policy being trained was never used.
  - After training, the optimal policy corresponds to systematically choosing the action with the highest Q-value.
- The policy gradients algorithm is an *on-policy* algorithm: it explores the world using the policy being trained.

ε-Greedy Policy

- *ε-greedy policy*: at each step, act randomly with probability ε, or greedily with probability $1-\varepsilon$ (i.e., choosing the action with the highest Q-value).
- The advantage of the ε-greedy policy: it will spend more and more time exploring the interesting parts of the environment, as the Q-value estimates get better and better, while still spending some time visiting unknown regions of the MDP.
- Start with a high value for ε (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05).

Exploration Function

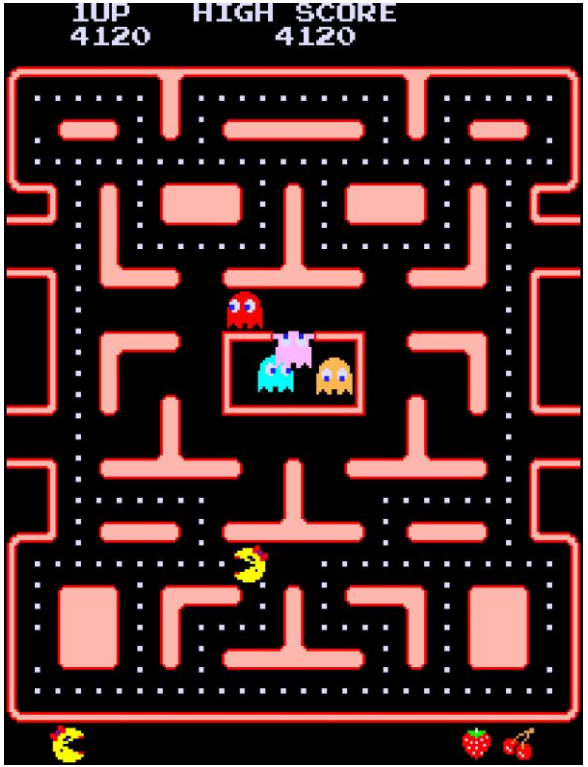
- Instead of relying on chance, encourage the exploration policy to select less-tried actions.
- This can be implemented as a bonus added to the Q-value estimates:

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma . \max_{a'} f\big(Q(s',a'), N(s',a')\big)$$

  - $N(s',a')$: the number of times the action a' was chosen in state s'.
  - $f(Q,N)$ is an *exploration function*, e.g. $f(Q,N) = Q + \frac{\kappa}{1+N}$ where κ is a curiosity hyperparameter that measures how much the agent is attracted to the unknown.

Scalability Problem of Q-learning

- The main problem with Q-learning is that it does not scale well to large (or even medium) MDPs with many states and actions.
- If you add all the possible combinations of positions for all the ghosts and Ms. Pac-Man and presence or absence of each pellet, the number of possible states becomes larger than the number of atoms on earth!



Deep Q-Learning

- The solution to the scalability problem is to find a function $Q_{\boldsymbol{\theta}}(s, a)$ that approximates the Q-value of state-action pair (s, a) using a manageable number of parameters (given by the parameter vector $\boldsymbol{\theta}$).
  - This is called *approximate Q-learning*.
- For years, we used linear combinations of handcrafted features extracted from the state to estimate Q-values.
  - In 2013, DeepMind showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering.
- A DNN used to estimate Q-values is called a *deep Q-network* (DQN), and using a DQN for approximate Q-learning is called *deep Q-learning*.

Training a DQN

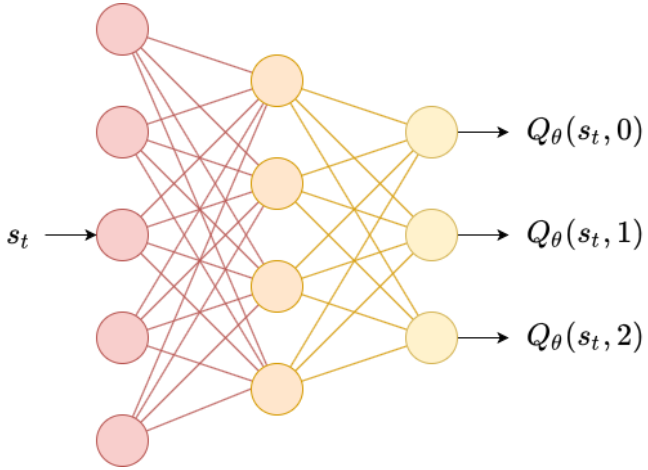
- $Q_{\boldsymbol{\theta}}(s, a)$ = the approximate Q-value computed by the DQN for a given state-action pair (s, a).
  - **Bellman:** $Q_{\boldsymbol{\theta}}(s, a)$ should be as close as possible to the reward r that we actually observe after playing action a in state s, plus the discounted value of playing optimally from then on.
  - To estimate this sum of future discounted rewards, we can just execute the DQN on the next state s', for all possible actions a' and pick the highest and discount it.
- By summing the reward r and the future discounted value estimate, we get a target Q-value $y(s, a)$ for the state-action pair (s, a):

$$y(s, a) = r + \gamma . \max_{a'} Q_{\boldsymbol{\theta}}(s', a')$$

7.
Implementing Deep Q-Learning

Make a DQN

- First we need a deep Q-network: a neural net that takes a state as input, and outputs one approximate Q-value for each possible action.
  - For the CartPole environment, a simple neural net with a couple of hidden layers is enough:




```
input_shape = [4]  # == env.observation_space.shape
n_outputs = 2  # == env.action_space.n

model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    tf.keras.layers.Dense(32, activation="elu"),
    tf.keras.layers.Dense(n_outputs)
])
```

Select an Action

- To select an action using the DQN, we pick the action with the largest predicted Q-value.
- To ensure that the agent explores the environment, we use an ε-greedy policy:

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs)  # random action
    else:
        Q_values = model.predict(state[np.newaxis], verbose=0)[0]
        return Q_values.argmax()  # optimal action according to the DQN
```

Replay Buffer

- Instead of training the DQN based only on the latest experiences, we store all experiences in a *replay buffer*, and we sample a random training batch from it at each training iteration.
  - This helps reduce the correlations between the experiences in a training batch, which significantly helps training.
- For this, we use a double-ended queue (deque):

```
from collections import deque

replay_buffer = deque(maxlen=2000)
```

- A *deque* is a queue elements can be efficiently added to or removed from on both ends.

Sample Experiences

- Each experience is composed of six elements:
  1. a state s
  2. the action a that the agent took
  3. the resulting reward r
  4. the next state s' it reached
  5. a Boolean indicating whether the episode ended at that point ($done$)
  6. a Boolean indicating whether the episode was truncated at that point ($truncated$)
- A function to sample a random batch of experiences from the replay buffer:

```
def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
    return [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(6)
    ]  # [states, actions, rewards, next_states, dones, truncateds]
```

Play One Step

- Create a function that will play a single step using the ε-greedy policy, then store the resulting experience in the replay buffer:

```
def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, truncated, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done, truncated))
    return next_state, reward, done, truncated, info
```

- Create a function that samples a batch of experiences from the replay buffer and train the DQN by performing a single gradient descent step:

```
batch_size = 32
discount_factor = 0.95
optimizer = tf.keras.optimizers.Nadam(learning_rate=1e-2)
loss_fn = tf.keras.losses.mean_squared_error
```

Training Step

$$y(s,a) = r + \gamma . \max_{a'} Q_{\boldsymbol{\theta}}(s', a')$$

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones, truncateds = experiences
    next_Q_values = model.predict(next_states, verbose=0)
    max_next_Q_values = next_Q_values.max(axis=1)
    runs = 1.0 - (dones | truncateds)  # episode is not done or truncated
    target_Q_values = rewards + runs * discount_factor * max_next_Q_values
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

→ e.g., if the first 3 experiences contain actions 1, 1, 0, then the mask will start with [[0, 1], [0, 1], [1, 0],...].

Training the Model

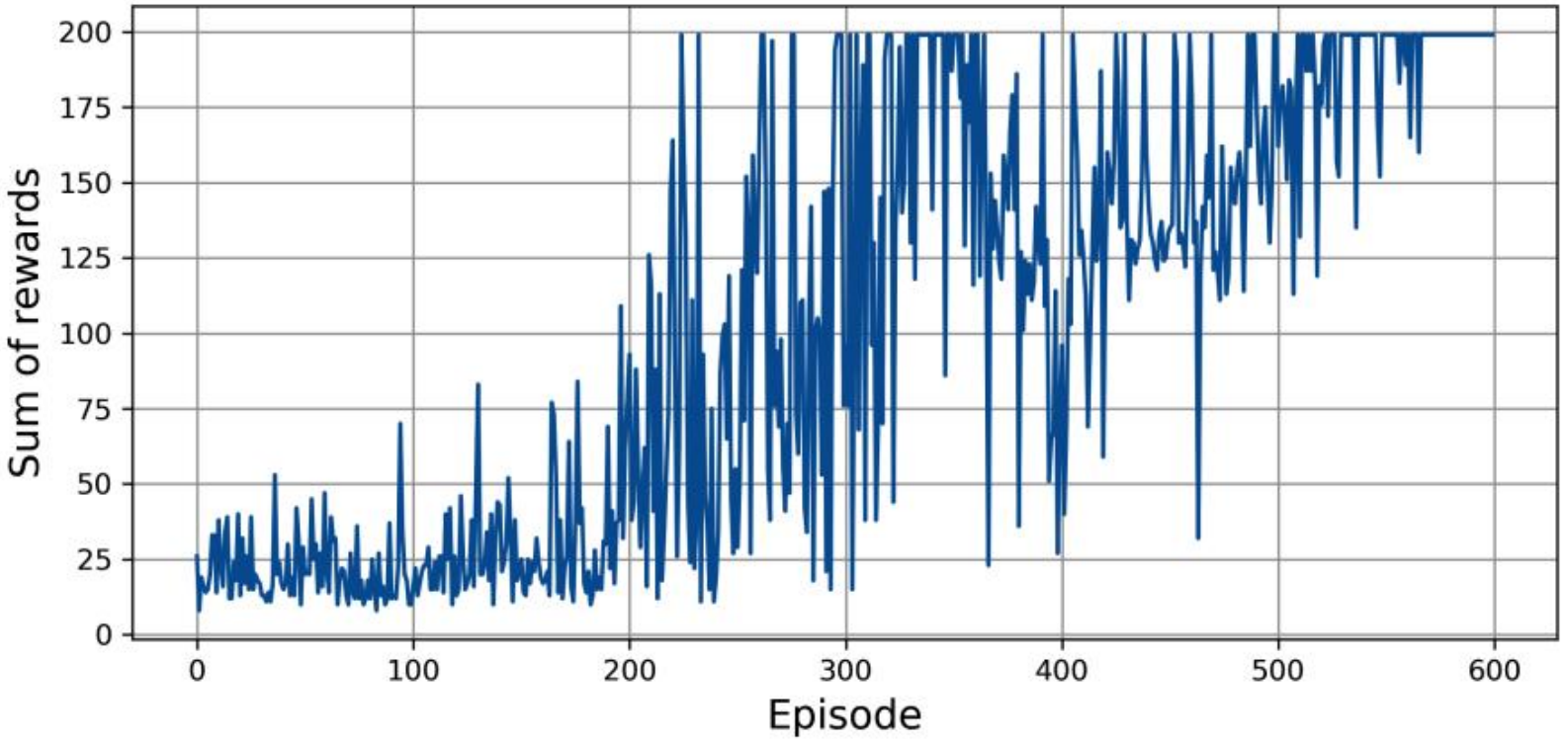
- We run 600 episodes, each for a maximum of 200 steps:

```
for episode in range(600):
    obs, info = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, truncated, info = play_one_step(env, obs, epsilon)
        if done or truncated:
            break
```

- If we are past episode 50, we call the `training_step`() function to train the model on one batch sampled from the replay buffer:

```
if episode > 50:
    training_step(batch_size)
```

Learning Curve



Catastrophic Forgetting

- *Catastrophic forgetting*: as the agent explores the environment, it updates its policy, but what it learns in one part of the environment may break what it learned earlier in other parts of the environment.
  - The experiences are quite correlated, and the learning environment keeps changing—this is not ideal for gradient descent!
    - If you increase the size of the replay buffer, the algorithm will be less subject to this problem.
  - Tuning the learning rate may also help.
- Reinforcement learning is hard: training is often unstable, and you may need to try many hyperparameter values and random seeds before you find a combination that works well.
  - For example, if you try changing the activation function from "elu" to "relu", the performance will be much lower.

Loss Function

- Loss is a poor indicator of the DQN model's performance.
  - The loss might go down, yet the agent might perform worse:
    - This can happen when the agent gets stuck in one small region of the environment, and the DQN starts overfitting this region.
  - The loss could go up, yet the agent might perform better:
    - If the DQN was underestimating the Q-values and it starts correctly increasing its predictions, the agent will likely perform better, getting more rewards, but the loss might increase because the DQN also sets the targets, which will be larger too.
- It is preferable to plot the rewards.