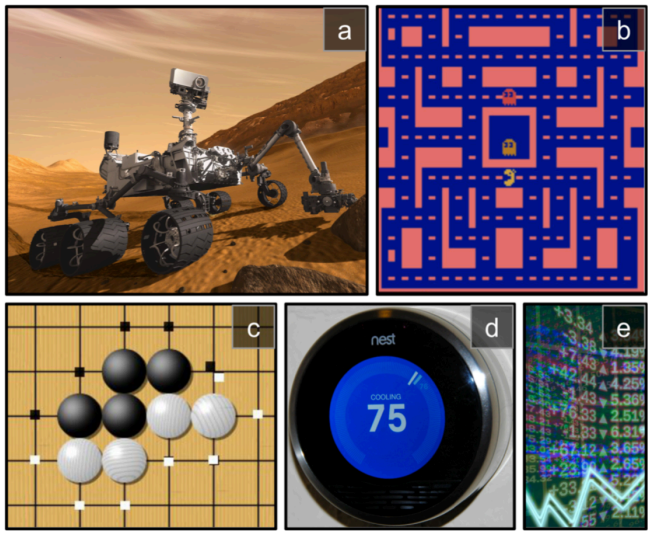
Reinforcement Learning (Chapter 18)

- Reinforcement Learning (RL) is one of the oldest ML fields which took off in 2013 and became a success story

- Specifically, the start-up DeepMind developed algorithms which allowed computers to beat humans in the hardest games and to teach themselves to play arcade games better than men… Some RL applications are



RL Basic Components

The following are RL's main components

- Agent
- Observations
- Actions
- Environment
- Rewards

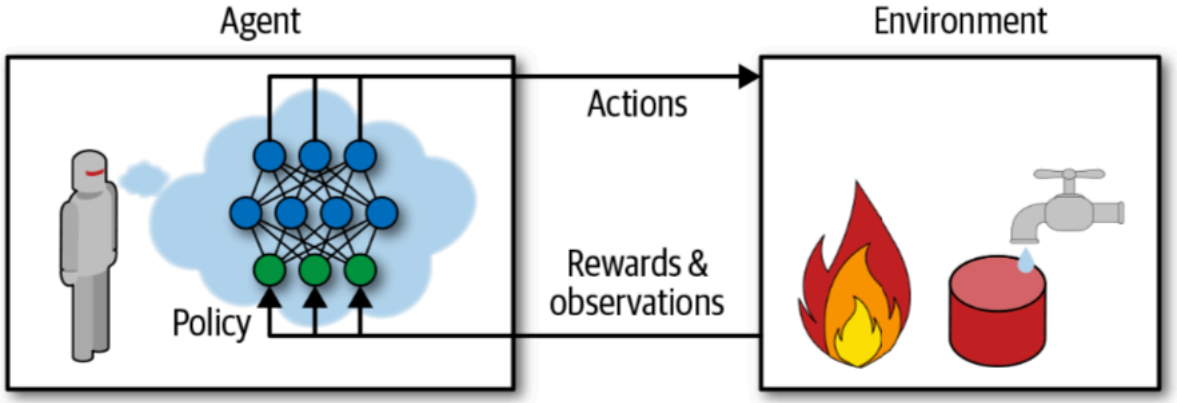
The goal is to learn the actions which will maximize the expected rewards over time. We can identify the components for some of a)-e) in the above picture

a) Agent – the program controlling the robot; environment – the real world; observations – readings from sensors; actions – commands to robot's motors; rewards – positive if robot approaches target and negative otherwise

e) The agent observes stock market prices and decides how much to buy/sell

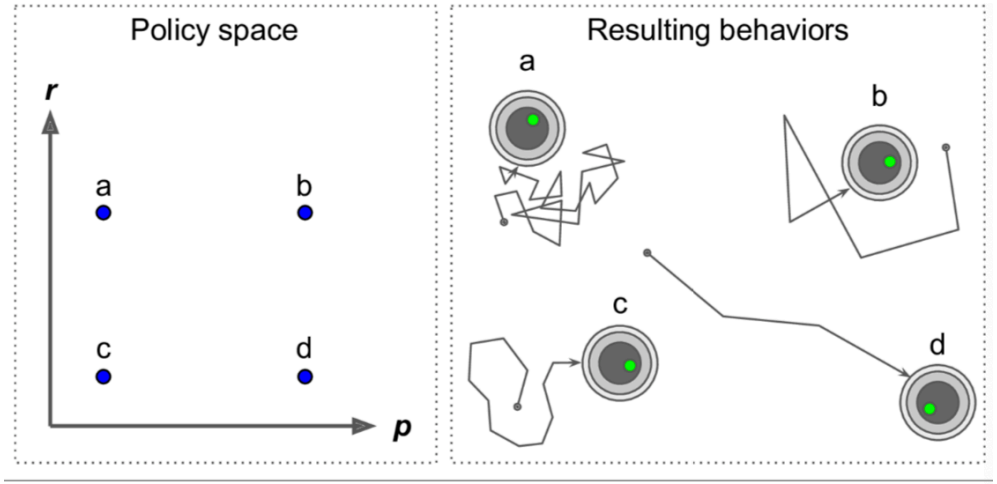
Policy Search

- Policy – the algorithm an agent uses to determine its actions



- Stochastic policy – when decision involves some randomness. In some cases, you may not even need observations – see next example.

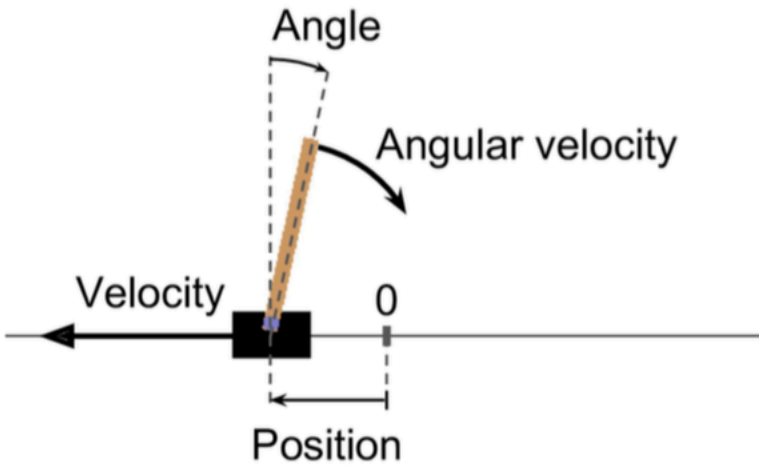
- Example – robotic vacuum cleaner which aims to pick as much dust as possible in 30 mins. There are 2 policy parameters: p – probability to move forward and $1 - p$ to rotate with random angle between $-r$ and r. Next is an illustration of this robot's behaviors.



- Training of such robot can be done by trying many different values for the policy parameters as on the above picture
- Genetic algorithms
- Policy Gradients (PG) - optimization of the gradients of the rewards

CartPole – a Demonstrative Example in OpenAI Gym

- The goal is to balance the pole on top of a cart for as long as possible



- Observations – (horizontal position, velocity, pole angle, angular velocity)
- Actions – accelerate the cart, expressed by an integer: 0 (=left), 1 (=right)
- Environment – a simulator is created in OpenAI gym

After creating the environment, we want to implement and evaluate some policies.

The step() method executes an action and returns 4 values :

```
>>> action = 1  # accelerate right
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.01261699,  0.19292789,  0.04204097, -0.28092127])
>>> reward
1.0
>>> done
False
>>> info
{}
```

obs: This is the new observation.

reward: In this environment, you get a reward of 1.0 at every step, no matter what you do, so the goal is to keep the episode running as long as possible.

done: This value will be True when the episode is over. This will happen when the pole tilts too much, or goes off the screen, or after 200 steps (in this last case, you have won).

Info: Some extra information that you may find useful for debugging or for training.

Let's define a simple policy – accelerate left when pole is leaning toward left and accelerate right otherwise, is implemented below

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

```
>>> import numpy as np

>>> np.mean(totals), np.std(totals), np.min(totals),
np.max(totals) (41.718, 8.858356280936096, 24.0,
68.0)
```

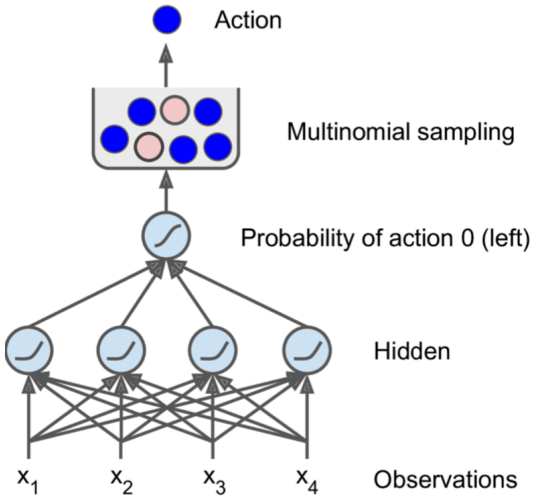
Even with 500 tries, this policy never managed to keep the pole upright for more than 68 consecutive steps. Not great.

(Look at the animation in the notebook.)

Time to try some other policies.

Neural Network Policies

- The NN neural network will take an observation as input, will estimate a probability for each action, and then we will select an action randomly, according to the estimated probabilities.
- There are 2 possible actions (moving left or right) => only need 1 output neuron. It will output the probability p of action 0 (left), and the probability of action 1 (right) will be $1 - p$



Using probability to pick an action allows the agent to

- Explore (new actions)
- Exploit (actions which work well)

In this environment, each observation contains environment's full state.

- The implementation is simple

```
import tensorflow as tf
from tensorflow import keras

n_inputs = 4 # == env.observation_space.shape[0]

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
])
```

- In this case, the output is a single probability (of going left), so 1 output neuron with a sigmoid activation is sufficient.

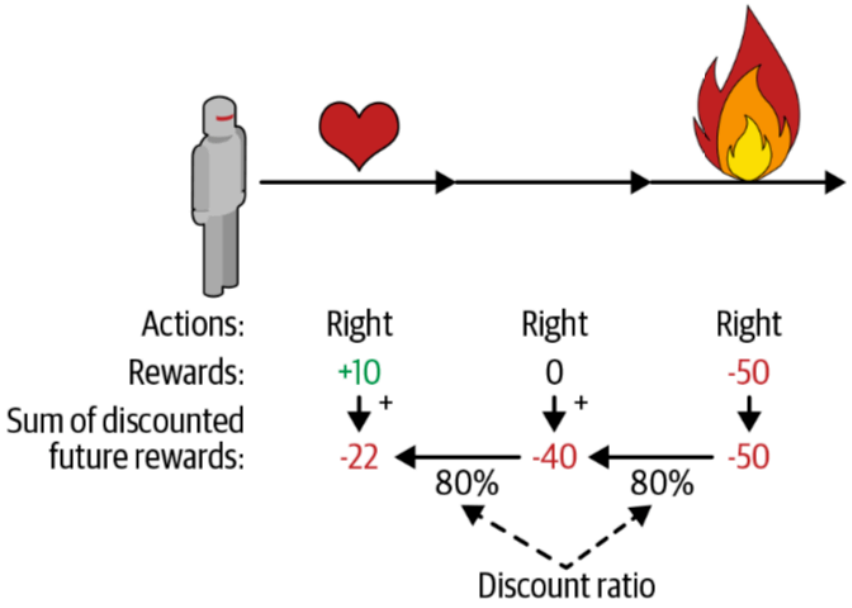
- For > 2 possible actions, use 1 neuron/action and softmax activation.

Evaluating Actions: The Credit Assignment Problem

- If we knew what the best action is at each step, we could consider and train RL as a regular supervised learning
- But the only information how good an action the agent took was is through rewards which are usually sparse and delayed. Consider the balancing pole example.
- This credit assignment problem (how much each action affected positively or negatively the outcome) is usually tacked by applying a discount factor $0 < \gamma < 1$ at each step. After k-steps, the reward would be

$$r_1 + \gamma r_2 + \cdots + \gamma^{k-1} r_k$$

If γ is close to 0, then (discounted) future rewards will not count that much compared to immediate rewards.



The discounted feature rewards corresponding to the picture (3 steps with 3 rewards) and $\gamma = 0.8$ are:

$$10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$$

Typically, γ is 0.9-0.99.

- A good action may be followed by several bad actions that cause the pole to fall quickly, resulting in the good action getting a low return. However, on average good actions will get a higher return than bad ones.
- We want to estimate this action advantage. To do that we must run many episodes and normalize all the action returns

Policy Gradients

- Knowing how to estimate the action advantage, we can train an agent.
- REINFORCE algorithms (1992) are one way to do it. The usual steps are:
  - Let the NN policy play the game several times, and at each step, compute the gradients that would make the chosen action even more likely — but don't apply these gradients yet.
  - After several episodes, compute each action's advantage
  - If an action's advantage is positive => apply the gradients computed earlier to make the action even more likely to be chosen in the future. Act similarly, if action's advantage is negative. Thus, multiply each gradient vector by the corresponding action's advantage.
  - Finally, compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step.

The core of a **keras** implementation is function playing one step:

```
def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))

    grads = tape.gradient(loss, model.trainable_variables)

    obs, reward, done, info = env.step(int(action[0, 0].numpy()))

    return obs, reward, done, grads
```

Review Chapter 12/ Computing Gradients Using Autodiff (p.399) for explanations about GradientTape.

- There are 3 other functions
  - One organizes a loop to play the game multiple times
  - Two more to discount & normalize the rewards

- Let's define the hyperparameters

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

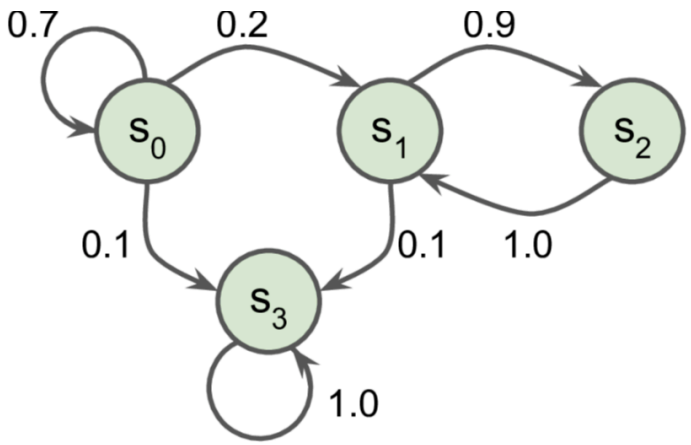
- And an optimizer and the loss function.

```
optimizer = keras.optimizers.Adam(lr=0.01)
loss_fn = keras.losses.binary_crossentropy
```

- There is a final training loop and after training the mean reward per episode will get very close to 200 (which is the maximum in this example).
- This algorithm is very inefficient in terms of samples/time it needs to make progress - it needs many episodes to estimate each action's advantage
- Next types of algorithms don't try to increase the returns by tweaking the policies. The agent first learns to estimate expected return for each state/action and then decides how to act

Markov Decision Processes

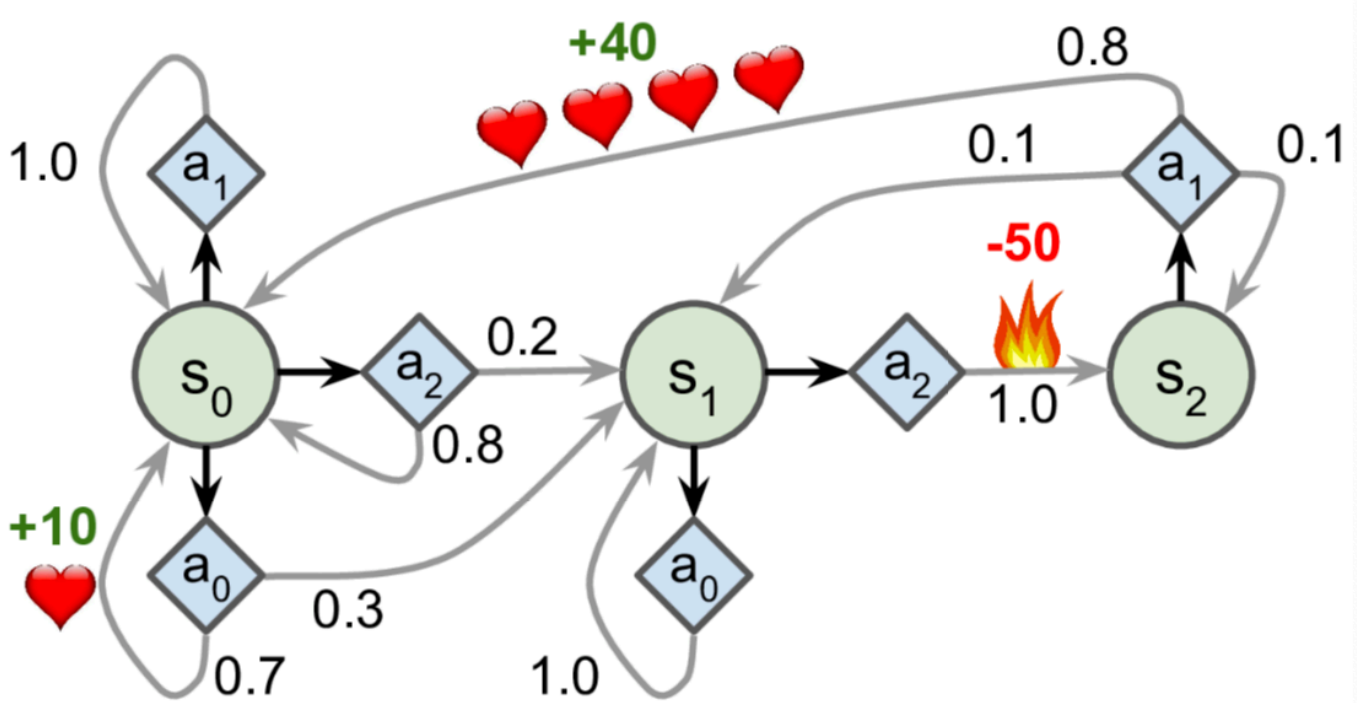
- Markov chains – fixed # of states, randomly goes from state to state with given probability (transition probability) depicted on the arc. An example with 4 states below



State s_3 is called terminal state (once there, the process never leaves)

Markov decision problem (R. Bellman, 1950's)

- Similar to Markov chain but an agent can choose some action which affects the transition probabilities. Also, some transitions bring some reward. The agents is looking for a policy maximizing reward over time.



If the process starts at any of the states, what is the "optimal strategy"?

- Bellman's Optimality Equation (Dynamic Programming), (18.1)

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \text{ for all } s$$

$V^*(s) =$ optimal state value, i.e. the sum of the discounted future rewards an agent can expect on average (at state s), assuming it acts optimally.

$T(s,a,s')$ - transition probability for $s \to s'$, given action a
$R(s,a,s')$ – the reward for the above transition
$\gamma -$ discount factor

- Value Iteration Algorithm, (18.2)

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s)] \text{ for all } s$$

Here $V_k(s)$ is the estimated optimal values at state s at iteration k. Initialize with 0 – the $V_k(s)$ converge to optimal state values when $k \to \infty$.

- Q-Value (Quality Value) Iteration Algorithm – due again to Bellman

$$Q_{k+1}(s,a) \leftarrow \max_{a} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')] \text{ for all } (s,a)$$

$Q_k(s,a)$ is the current approximation of $Q*(s,a)$ defined below:

$Q*(s,a)$ is the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a, but before it sees the outcome of this action, assuming it acts optimally after that action.

- Once the optimal Q-Values are determined, defining the optimal policy $\pi*(s)$ is trivial. Namely, when the agent is in state s, it should choose the action with the highest Q-Value for that state:

$$\pi^* = argmax_a Q^*(s,a)$$

- For the MDP depicted on slide 16:

```
transition_probabilities = [ # shape=[s, a, s']
        [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
        [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
        [None, [0.8, 0.1, 0.1], None]]
rewards = [ # shape=[s, a, s']
        [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
        [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
        [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

- Next, initialize all the Q-Values to 0 (except for the impossible actions, for which we set the Q-Values to $-\infty$):

```
Q_values = np.full((3, 3), -np.inf) # -np.inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0  # for all possible actions
```

- Applying the algorithm is simple

```
gamma = 0.90 # the discount factor

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                    transition_probabilities[s][a][sp]
                    * (rewards[s][a][sp] + gamma * np.max(Q_prev[sp]))
                for sp in range(3)])
```

- The resulting “solution” looks like this

```
>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.        ,         -inf, -4.87971488],
       [       -inf, 50.13365013,         -inf]])
```

E.g. when the agent is in state s_0 and it chooses action a_2, the expected sum of discounted future rewards is approximately 13.62. And this is the list of the best actions (their indices) for each state

```
>>> np.argmax(Q_values, axis=1) # optimal action for each state
array([0, 0, 1])
```

Temporal Difference Learning (TD Learning)

- TD Learning algorithm is very similar to the Value Iteration algorithm, but here we assume that the agent initially knows only the possible states and actions, and nothing more.
- The agent uses an *exploration policy* (e.g. a purely random one) to explore the MDP. As it progresses, the TD Learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed

Equation 18-4. TD Learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

or, equivalently:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$

with $\delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$

- In the above equation, we have

  - α is the learning rate (e.g., 0.01).
  - $r + \gamma \cdot V_k(s')$ is called the *TD target*.
  - $\delta_k(s, r, s')$ is called the *TD error*.

- A short notation is using this convention

$$a \underset{\alpha}{\leftarrow} b, \text{ which means } a_{k+1} \leftarrow (1 - \alpha) \cdot a_k + \alpha \cdot b_k.$$

$$V(s) \underset{\alpha}{\leftarrow} r + \gamma \cdot V(s').$$

- For each state s, this algorithm simply keeps track of a running average of the immediate rewards the agent gets upon leaving that state, plus the rewards it expects to get later

Q-Learning

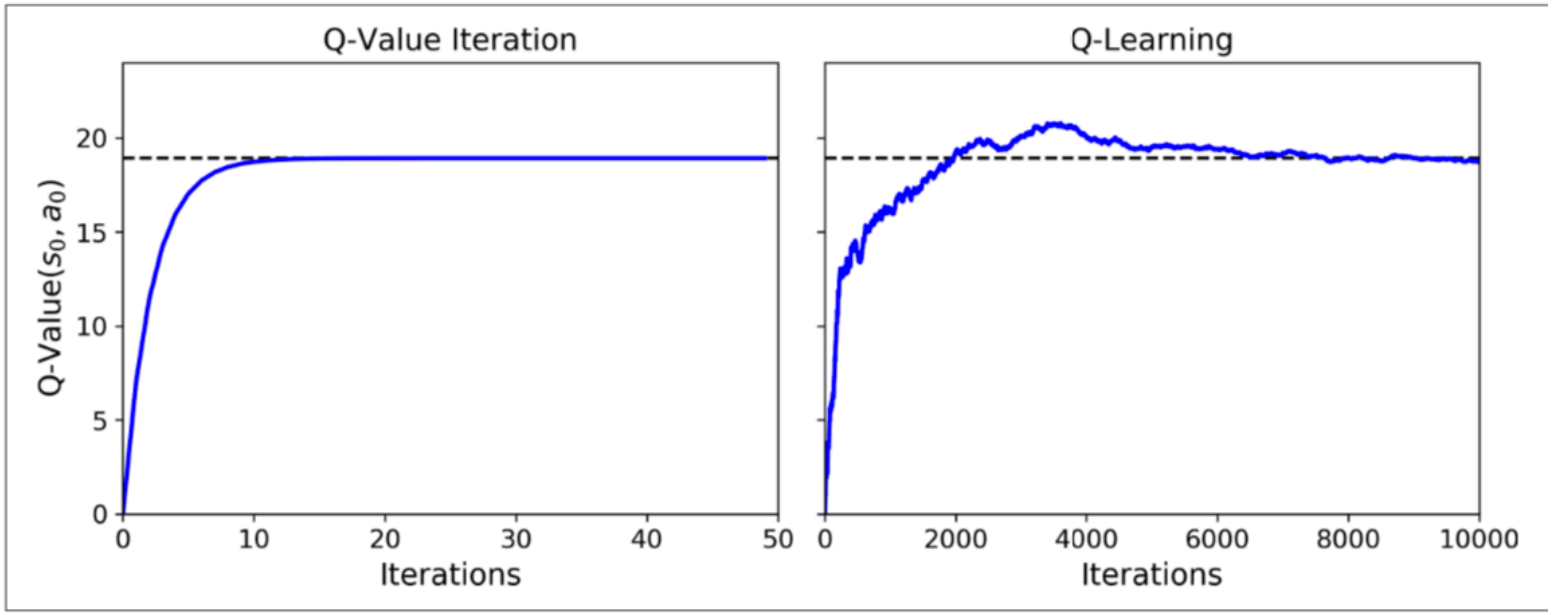
- An adaptation of the Q-Value Iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown

Equation 18-5. Q-Learning algorithm

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

- For each state-action pair (s, a), this algorithm keeps track of a running average of the rewards r received upon leaving the state s with action a, plus the sum of discounted future rewards it expects to get. To estimate this sum, we take the maximum of the Q-Value estimates for the next state s', since we assume that the target policy would act optimally from then on.

- This algorithm will converge to the optimal Q-Values, but it will take many iterations, and possibly quite a lot of hyperparameter tuning.
- On plot below, the Q-Value Iteration algorithm (left) converges very quickly, in fewer than 20 iterations, while the Q-Learning algorithm (right) takes about 8,000 iterations to converge. Obviously, not knowing the transition probabilities or the rewards makes finding the optimal policy significantly harder



Exploration Policies

- Using a random policy to explore the MDP is very slow

- A better option is to use ε-greedy policy at each step
  - Act randomly with probability ε
  - Or greedily (choosing the action with highest Q-value) with probability $1-\varepsilon$

  Usually, $\varepsilon \sim 1$ in the beginning and then with time it goes down. This way the agent will explore the interesting parts of the environment more and more while still visiting unknown parts of it.

- Another option is to use an exploration function which gauges the agent's affinity to explore the unknown through a curiosity parameter.

Approximate Q-Learning and Deep Q-Learning

- The main weakness of Q-Learning is that it doesn't scale well on larger MDPs. E.g. in Pac-Man game, there are > 10^{45} states.
- Very efficient approximations of Q-Learning that work faster are based on DNN. They also don't require feature engineering.
- A DNN used to estimate Q-Values is called a Deep Q-Network (DQN). Using a DQN for Approximate Q-Learning is called Deep Q-Learning.
- Denote by $Q_{\boldsymbol{\theta}}(s,a)$, here $\boldsymbol{\theta}$ is a hyperparameter vector with a relatively low dimension, an efficient approximation of $Q(s,a)$.Then, we can use a Gradient Descent algorithm to minimize the squared error between the estimation and the target Q defined as in Equation 18-7:

$$Q_{\text{target}}\left(s,a\right) = r + \gamma \cdot \max_{a'} \quad Q_{\boldsymbol{\theta}}\left(s',a'\right)$$

Implementing Deep Q-Learning

- Instead of building a DQN that takes a (s, a) pair and outputs an approximate Q-Value, more efficient is to build a DQN taking a state s and outputting an approximate Q-Value for each possible a.

- Consider the above for the cart-pole system & define a simple NN

```
env = gym.make("CartPole-v0")
input_shape = [4] # == env.observation_space.shape
n_outputs = 2 # == env.action_space.n

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

- An ε-greedy policy is used

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(2)
    else:
        Q_values = model.predict(state[np.newaxis])
        return np.argmax(Q_values[0])
```

- To reduce the correlations between the experiences in a training batch, store all experiences in a *replay buffer* (or *replay memory*), and then sample a random training batch from it at each training iteration. It can be achieved through a deque (see text)
- A function is defined to sample a random batch of experiences from the replay buffer. It will return 5 NumPy arrays corresponding to:

  state, action taken, reward, next state reached & "done" indicator

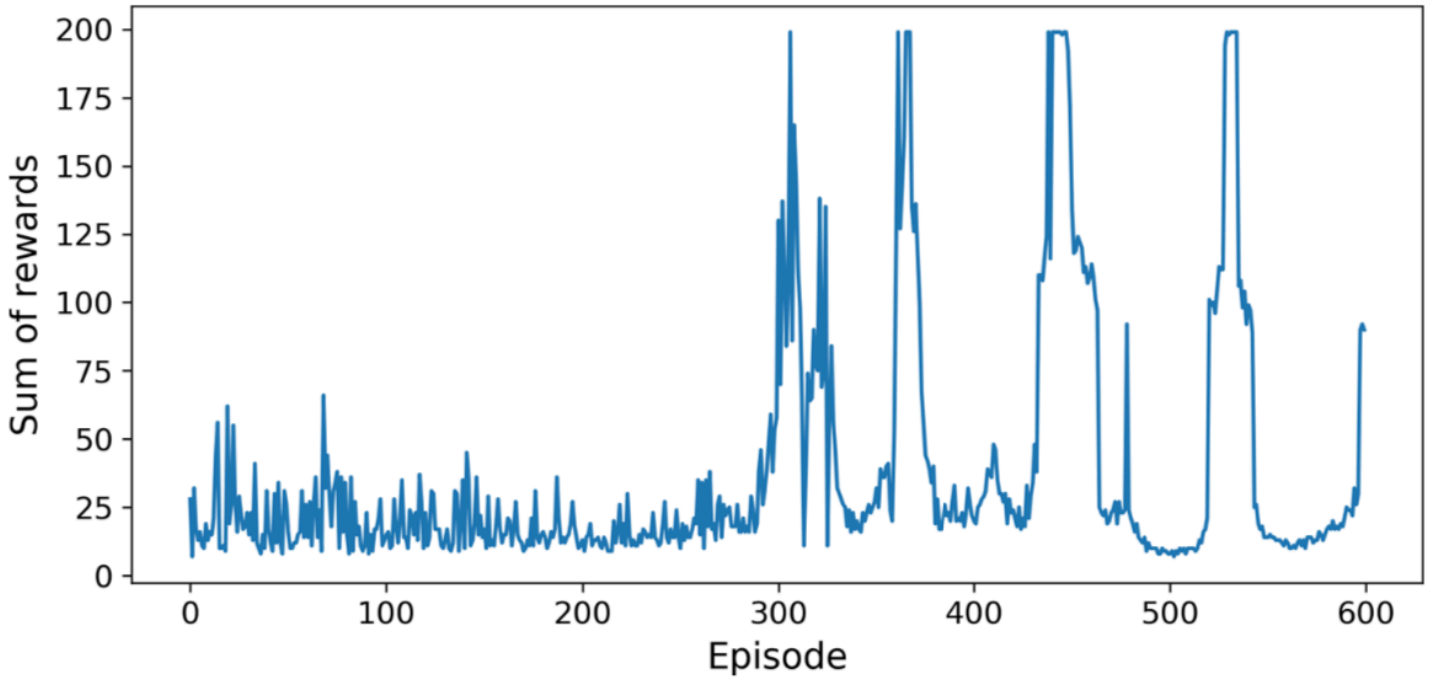
  Then a function making a single step (using ε-greedy policy) and storing the results in the replay buffer is added, as well as the training step is set

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

```python
batch_size = 32
discount_factor = 0.95
optimizer = keras.optimizers.Adam(lr=1e-3)
loss_fn = keras.losses.mean_squared_error
```

```python
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    max_next_Q_values = np.max(next_Q_values, axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * max_next_Q_values)
    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))
```

- The model is trained for 600 episodes with max 200 steps in each and the rewards are plotted



- The spikes on the right illustrated one of the big challenges in RL - catastrophic forgetting. The experiences are very correlated and give a hard time to GD. Partly fixable by reducing LRate. Instability still remains.

Deep Q-Learning Variants

Fixed Q-Value Targets

- The basic Deep Q-Learning is employing a “closed loop” (both predicts and sets its own targets) and this makes the NN to oscillate, diverge etc.
- This problem was solved in 2013 by using two NN models
  - online model – to learn at each step & move the agent
  - target model – only defines the targets (a clone of the online model)
- The above can be achieved by simple model cloning (to define the target model) and copying the weights of the online model to the target mode at regular, preferably long, intervals (e.g. every 10,000 steps)

```
target = keras.models.clone_model(model)
target.set_weights(model.get_weights())

next_Q_values = target.predict(next_states)

if episode % 50 == 0:
    target.set_weights(model.get_weights())
```

Double DQN

- It was observed that the target network overestimates the Q-Values. To fix that, the online model is used instead of the target model when selecting the best actions for the next states. The target model is used only for the estimation of the best actions

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    best_next_actions = np.argmax(next_Q_values, axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    next_best_Q_values = (target.predict(next_states) * next_mask).sum(axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * next_best_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    [...] # the rest is the same as earlier
```

Prioritized Experience Replay (PER)

- In this variation of DQN, importance sampling (IS) is used.
- Importance is defined as the ability to learn faster. E.g. it could be the magnitude of the TD error

$$\delta = r + \gamma\, V(s') - V(s)$$

- Once sampled, the priority of an experience is set to $p = |\delta|$ and the probability of sampling to $P = p^{\zeta}, 0 \leq \zeta \leq 1$ ($\zeta = 0.6$ in DeepMind's paper)

- During training the weights are down-sampled: $w = (nP)^{-\beta}$, n - number of experiences in the replay buffer.

- ζ and β are hyperparameters

Dueling DQN

- For each state-action pair (s, a):

$$Q(s,a) = V(s) + A(s,a),$$

$A(s,a)$ is the *advantage* of taking the action a in state s, compared to all other possible actions in s. Since $V(s) = Q(s, a*) \Rightarrow A(s, a*) = 0$.

- In a Dueling DQN, the model estimates both the value of the state and the advantage of each possible action. The best action should have a 0 advantage, so the model subtracts max(predicted advantage) from all predicted advantages

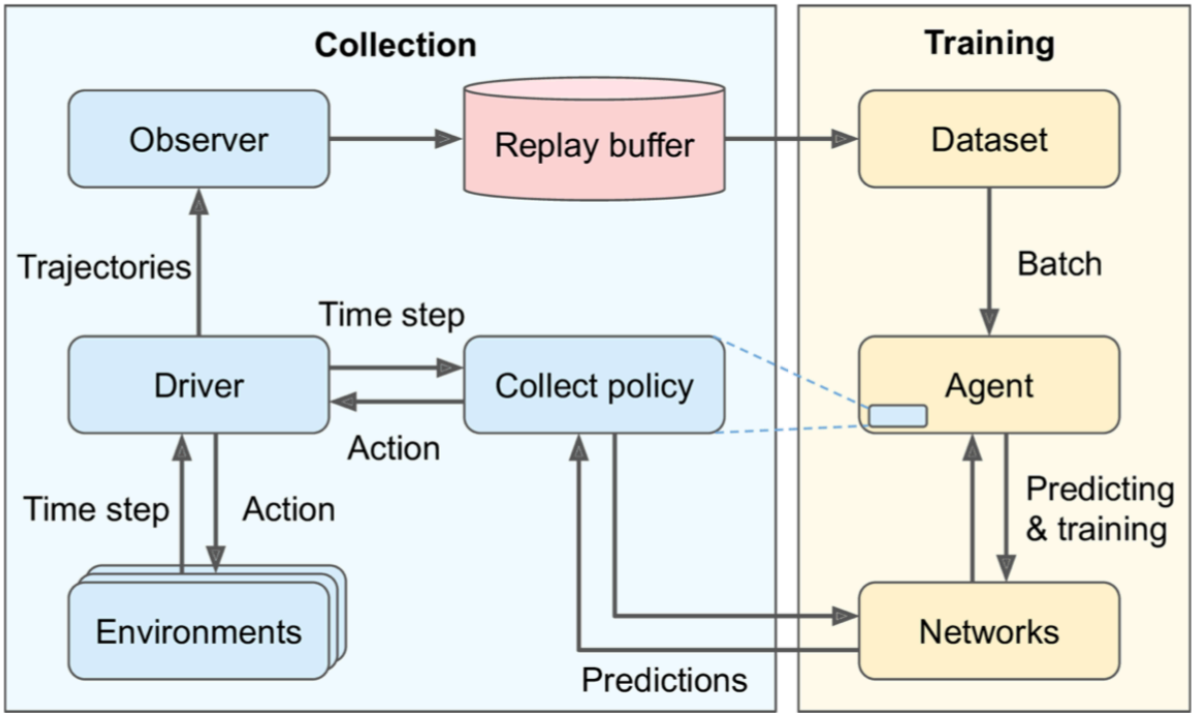
```
K = keras.backend
input_states = keras.layers.Input(shape=[4])
hidden1 = keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = keras.layers.Dense(32, activation="elu")(hidden1)
state_values = keras.layers.Dense(1)(hidden2)
raw_advantages = keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - K.max(raw_advantages, axis=1, keepdims=True)
Q_values = state_values + advantages
model = keras.Model(inputs=[input_states], outputs=[Q_values])
```

The TF-Agents Library

- The RL models and improvements described above are complicated and take a lot of time to make work and tune
- TF-Agents package comes to the rescue with the implementation of many RL algorithms and refinements (REINFORCE, DQN, DDQN etc.)
- This is demonstrated in the notebook where training of an agent to play the game Breakout is done

Training Architecture

- A TF-Agents training program usually has two part running in parallel



- Try running the Breakout training!