Tutorial 11: Deep Reinforcement Learning

ESE546

November 26, 2020

1 The Reinforcement Learning Problem

Reinforcement Learning (RL) is a framework that deals with sequential decision making problems. An *agent* is the component that one has complete control over. An agent is responsible for assimilating information from the *environment*, which is used as the basis for *action*. For example, a self-driving car is an example of an agent with the rest of the world being the environment.

The environment also presents the agent with some feedback after every step. The feedback is called the *reward*. The effect of an action can often be delayed or could have implications far into the future. For example, a chess agent may have to sacrifice a piece and receive negative reward in order to win the game. This is referred to as the credit assignment problem. The optimal action should not only depend on the immediate reward, but also its effect on future states and rewards.

The goal of optimal control problem aims to maximize the sum of the rewards, accrued over the lifetime of the agent. This is determined by how the agent behaves in the environment. A *policy* is a function that maps states to actions in the environment and is a complete description of how an agent behaves in an environment. This tutorial will focus on techniques to find "optimal" policies under the Markov-decision process framework.

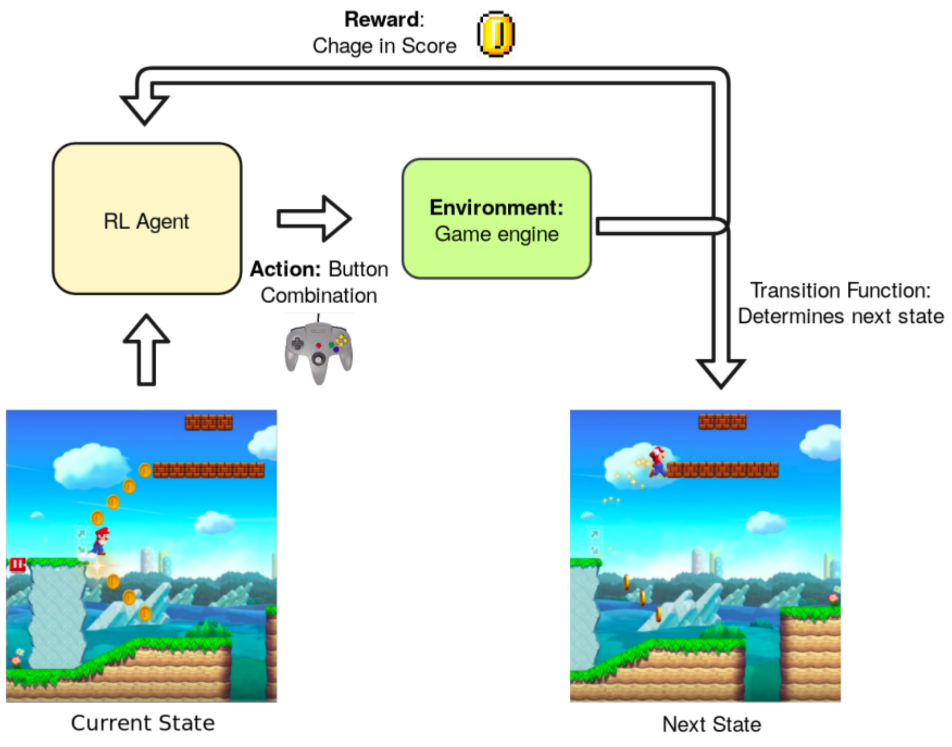


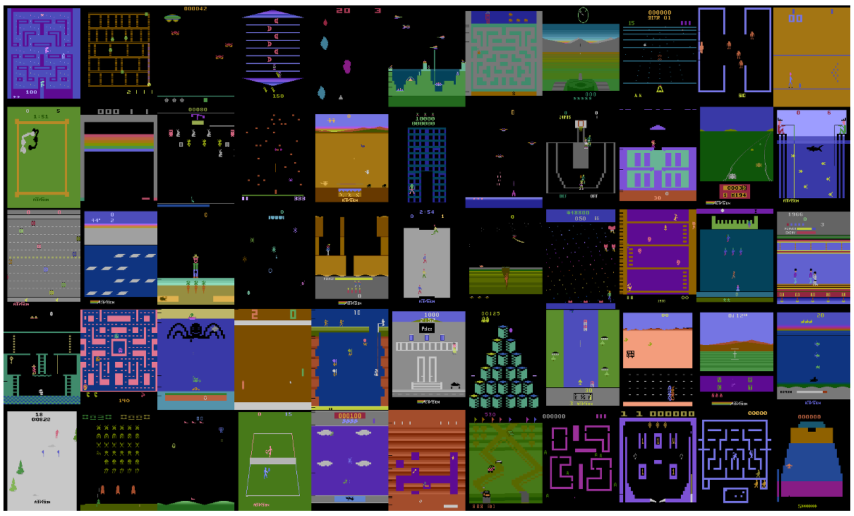
Figure 1: Agent and Environment

Note that this setup is very different from the supervised learning problem. In the supervised learning problem, we have a mapping from each input, to the desired output. In RL, we are left to infer the same from the rewards and adopt a trial and error based strategy.

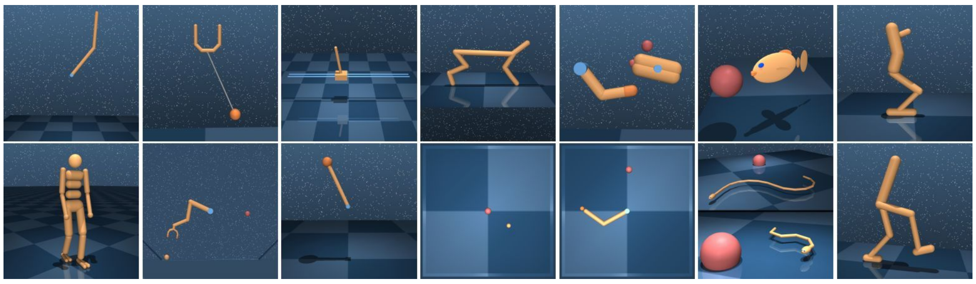
2 Deep Reinforcement Learning

Deep reinforcement learning agents makes use of neural networks to find a mapping from states to actions. This allows us to generalize to unseen scenarios using the power of neural networks. Video games provide a very natural testbed for reinforcement learning problems. The change in score is used as the reward while the RGB input is used as the "state" of the environment. Some success stories of RL include:

- **Atari Games**: The first biggest success in RL is the super-human performance of RL agents on 52 Atari games using Deep Q-learning (DQN).



- **Continuous control tasks in Robotic**: Algorithms like DDPG/PPO have demonstrated state of art performances tasks like Mujoco continuous control tasks and Deepmind-lab.



- **Alpha-Go**: Go/chess agents with the highest every Elo ratings, demonstrating super-human performance.
- **DoTA 2**: OpenAI's DoTA agent was capable of defeating professional DoTA players with ease

Does this mean that major problems in reinforcement learning have been solved? This is far from the case. Agents often require an enormous amount of data and hence, most success stories have been limited to video games where the process of gathering samples is not an expensive process. One can imagine that the same is not true for self-driving cars and robots. In fact, self-driving cars have a large cost associated with every "crash" and hence a trial-and-error based solution will not work.

Another difficult challenge is to determine a suitable reward function. Self-driving and robotic manipulation tasks don't have natural reward functions like video-games. Often, incorrectly designed reward functions lead to undesired artifacts in the policy. For example, training a Coastrunner agent using the game-score as the reward, results in the speedboat drifting in circles. This behavior is commonly observed in RL agents.

3 Exploration and Exploitation

The primary mechanism for learning in RL is trial-and-error. An RL agent is required to trade-off between exploitation and exploration. The exploitation phase executes the optimal strategy according to the current estimates of the agent. The exploration phase involves executing random actions, in order to build an estimate of the optimal policy.

We understand this balancing act through the lens of the *Multi-armed bandit problem*. The problem setup is as follows:

Definition 3.1. Multi-armed bandit problem: Consider k different slot machines. Each slot machine yields a reward that is sampled from a Gaussian. The agent is unaware about the distribution of each Gaussian. Given that we use the slot machines for infinitely many steps (or a very large number of steps), what is the optimal strategy that allows the agent to accumulate the maximum reward?

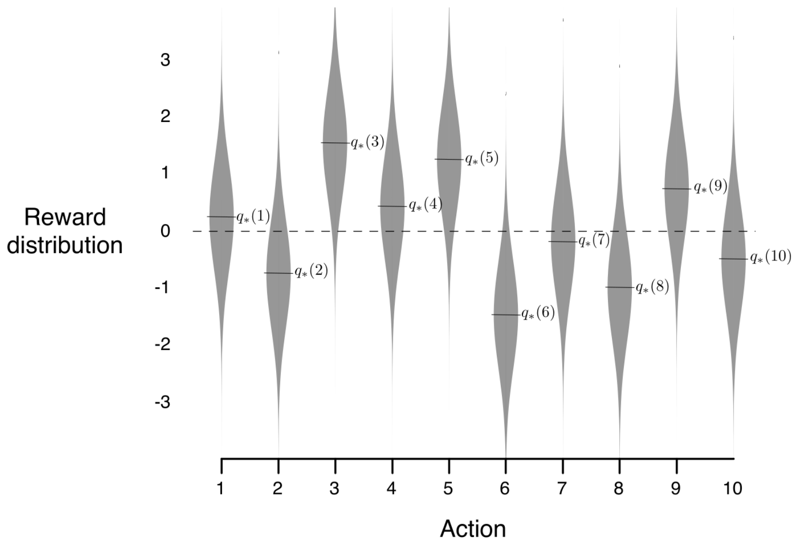


Figure 2: Figure 2.1 from Sutton & Barto

The multi-armed bandit problem is the simplest instantiation of the full RL problem. Actions from earlier steps have no bearing on future steps, resulting in a vastly simplified problem.

A simple strategy would involve trying each bandit arm once. The arm with the largest reward is used by the agent in all consequent steps. Such a strategy has a minimal amount of exploration and a large amount of exploitation.

Action-Value estimation: Estimate the means of each arm. Following which, execute the arm with the largest mean for the remaining steps. An improvement to this algorithm is called UCB and attempts an arm if its variance is large.

```
def action_value_bandit():
    # Measure of performance / sum of rewards
    return = 0.0

    # Estimate mean of actions in first k * 100 steps
    Q_value = zeros((k, 1))
    for arm_num in range(k):
        for _ in range(100):
            reward = env.step(action=arm_num)
            return += reward
            Q_value[k] += reward
    Q_value = Q_value / 100.0

    # Execute optimal action based on above
    best_arm = argmax(Q_value)
    for _ in range(TOTAL_STEPS - k * 100):
        reward = env.step(action=best_arm)
        return += reward

    reward = env.step(action)
```

Epsilon Greedy: Uses the parameter ϵ to trade-off exploration and exploitation. A common strategy is to anneal ϵ down to zero, while starting with a large value initially. This allows you to converge to the optimal arm in the limit.

```
def epsilon_greedy(epsilon):
    # Measure of performance / sum of rewards
    return = 0.0

    Q_value = random.uniform((k, 1))
    num_tries = random.uniform((k, 1))

    for _ in range(TOTAL_STEPS):
        with probability epsilon:
            arm = random_integer((1, k))
        with probability 1 - epsilon
            arm = argmax(Q_value)

        reward = env.step(action=arm)
        return += reward

        # Re-estimate mean
        num_tries[k] += 1
        Q_value[k] = [(num_tries[k] - 1)Q_value[k] + reward] /
            num_tries[k]
```

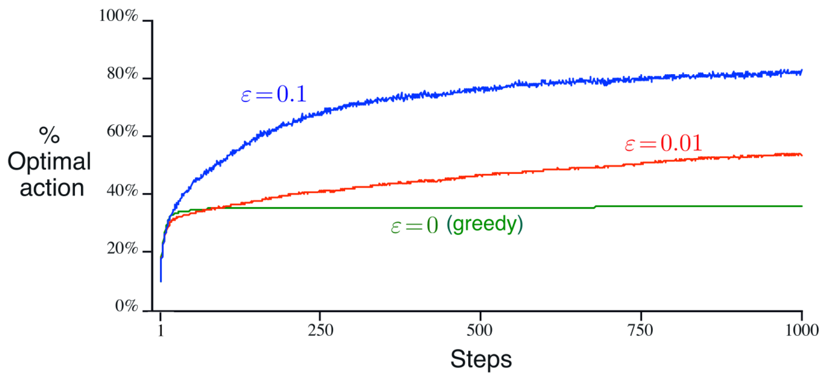


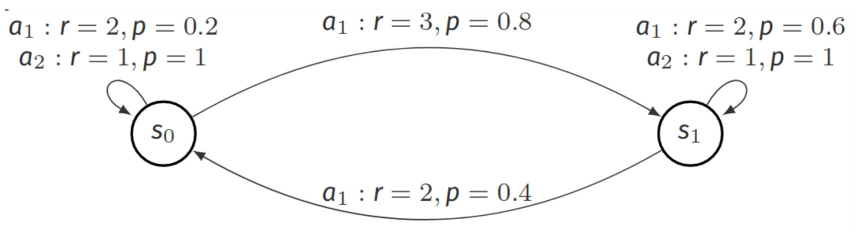
Figure 3: Figure from Sutton & Barto.

4 Markov-Decision Processes

We now formalize the Markov-decision processes (MDP), which is the underlying model for any RL problem. Every MDP is represented by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$. They represent:

- $\mathcal{S}$: The current "state of the agent.
- $\mathcal{A}$: The set of actions available to the agent
- $\mathcal{T}$: The transition function $\mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$, which is a mapping from (state, action) to a probability distribution over the states.
- $\mathcal{R}$: The reward function $\mathcal{S}_t \times \mathcal{S}_{t+1} \times \mathcal{A} \to \mathbb{R}$ is a mapping from the current state, next state and action, to the reward.

For example, a game of chess would have the chess board configuration to be the state. The transition function is deterministic and is determined by the rules of the game. There are different reward function for this task. One reward function would assign a positive reward for capturing a piece and a large positive reward for winning the game.



A policy is a mapping from each state, to a probability distribution over actions, i.e. $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$. The MDP satisfies the Markov property, implying that future transitions and rewards are solely dependent on the current state and is independent of the previous history. This implies that an optimal policy also satisfies the Markov property. The objective of reinforcement learning is to maximize the expected return defined as:

$$V_\pi = \mathbb{E}_{s_t \sim \mathcal{T}, a \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right]$$

γ is the discount factor and ensures that the above objective is bounded. The optimal policy is defined as the $\pi^* = \text{argmax}_\pi V_\pi$.

5 Value functions

Value functions are perhaps the most fundamental objects in all of RL. The value function for a particular policy is defined as the expected return starting from state s, executing policy π.

$$V_\pi(s_t) = \mathbb{E}\left[\sum_{i=t}^{\infty} \gamma^{i-t} r_i\right]$$

Note that $V_\pi(s_0)$ is direct measure of the performance of policy π. Another related object is the action-value function defined by $Q_\pi(s, a)$.

$$\begin{aligned} Q_\pi(s_t, a) &= \mathbb{E}\left[r(s_t, a, s_{t+1})\right] + \mathbb{E}\left[\sum_{i=t+1}^{\infty} \gamma^{i-t} r_i\right] \\ &= \mathbb{E}\left[r(s_t, a, s_{t+1})\right] + \gamma V_\pi(s_{t+1}) \end{aligned}$$

The value function V is the expected return when sampling solely from policy π. The Q-value function on the other hand is the return obtained by executing action a on the first step, followed by executing policy π for all consequent steps. Another relationship between Q_π and V_π is that:

$$V_\pi(s_t) = \sum_{a \in \mathcal{A}} \pi(a|s_t) Q(s_t, a)$$

Using the above relationship, we can build our first RL algorithm. The core idea is to alternate between evaluating the value of Q_π followed by finding a better policy π. This process is known as policy iteration where we alternate between policy evaluation and policy improvement. The algorithm is as follows:

```
def policy_iteration():
    Start with random policy pi
    repeat until convergence:
        Find Q_pi # Using Monte-carlo or Q-iteration

        # Find a better policy using Q-values
        for all states s:
            best_action = argmax(Q_pi(s))
            Set pi(s) = one_hot_vector(best_action)
```

Why does the above update work? Let the old policy be π. Consider the change in policy to the state s as described in the above algorithm. Let $a^* = \text{argmax}_a Q_\pi(s, a)$. Define π_{new} such that it differs only in state s, i.e., $\pi_{new}(s) = \text{one\_hot}(a^*)$. Observe that:

$$\begin{aligned} V_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a) \\ &\leq \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a^*) \\ &\leq Q_\pi(s, a^*) \\ &\leq \sum_{a \in \mathcal{A}} \pi_{new}(a|s) Q_\pi(s, a^*) = V_{\pi_{new}}(s) \end{aligned}$$

Hence, policy improvement strictly improves the value function from that state. If, the policy improvement stage results in no improvement in the resultant policy, then we have converged to the optimal policy. The optimal policy is related to the optimal Q-value function in the following manner:

$$\pi^*(s) = \text{argmax}_a Q^*(s, a) \tag{1}$$

Another interesting observation is that the final optimal policy is deterministic. In general, any MDP has atleast one deterministic optimal policy. The optimal Q-value function is unique while the optimal policy need not be unique.

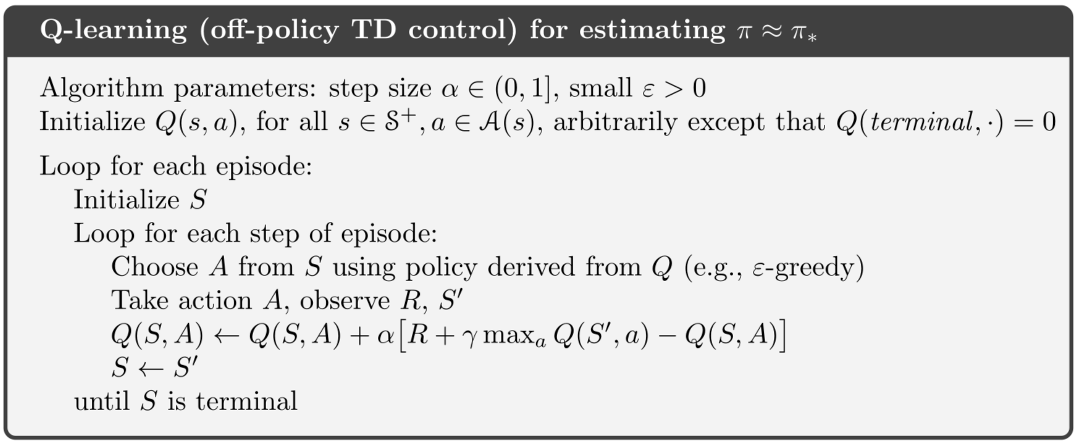
6 Q-learning

The Q-learning algorithm combines the policy improvement and policy evaluation steps into a single equation. Note that the Q-values of intermediate points in the algorithm need not correspond to Q-values of realizable policies. However on convergence, the Q-values are guaranteed to find the values of Q_{π^*}.

The underling strategy for Q-learning is to estimate the value of Q^* and finally use Equation 1 to derive the policy. The update for Q-learning is given by:

$$Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a) \right)$$

The update incorporates a learning rate α to account for the stochasticity in the environment. A slowly decaying function for α will result in asymptotic convergence. Repeatedly applying this algorithm results in the converge of $Q \to Q^*$. A typical instantiation of Q-learning looks like the below algorithm:



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

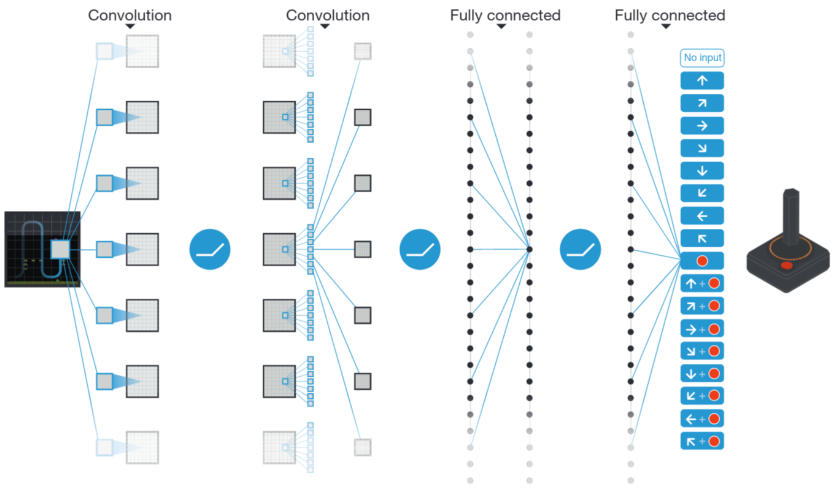
```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Figure 4: Q-learning from Sutton & Barto

7 Deep Q-learning

Maintaining tables of size $\mathcal{S} \times \mathcal{A}$ becomes infeasible for very large states spaces. This is common in robotic control tasks or in any vision based task. DQN is an application of deep-learning to the Q-learning algorithm. The Q-value function is realized using a neural network $Q_\theta(s)$. The Atari suite of games consists of 18 actions in total, which implies that the neural network outputs 18 linear outputs, corresponding to the 18 Q-value functions for each action in that state.



The loss function is given by the equation:

$$L_\theta = \Big(r_t + \gamma \max_a Q_\theta(s_{t+1}, a) - Q_\theta(s, a)\Big)^2$$

The gradient update for Q-learning is given by:

$$\nabla L_\theta = \nabla_\theta Q_\theta(s_t, a_t)\Big(r_t + \gamma \max_a Q_\theta(s_{t+1}, a) - Q_\theta(s, a)\Big)$$

However, this equation has been observed to be unstable in the past. In particular there are two problems with the above update. The first cause of instability is the unstable target $r_t + \gamma \max_a Q_\theta(s_{t+1}, a)$ for the Q-value. Change in the parameters dramatically affects the target. In order to ensure that the targets do not change rapidly, DQNs make use of a delayed target. The parameters used by the target is an older version of the parameters that is updated only every 10,000 steps. Hence, the new update is given by (note the θ^{-1}:

$$\nabla L_\theta = \nabla_\theta Q_\theta(s_t, a_t)\Big(r_t + \gamma \max_a Q_{\theta^-}(s_{t+1}, a) - Q_\theta(s, a)\Big)$$

The second issue is the correlated nature of every batch. Every batch is consists of samples from a contiguous part of the trajectory. This is addressed using an experience replay. The experience replay stores the 50,000 most recent samples. Every gradient update uses 32 (batch size) random samples from the experience replay. This dramatically slows down the code, but is critical to the stability of DQNs.

There have been a number of papers that have improved upon DQNs. Different methods stabilize the DQN (Double DQN, C51, Prioritized Experience replay, Noisy Networks, etc...).

```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t$ = 1,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

Figure 5: DQN algorithm

8 Deep Deterministic Policy Gradients

DQN assume that the output space is a discrete number of actions. DDPG extend the idea of Q-learning to continuous action spaces. A DDPG agent consists of two neural networks. The first outputs the Q-value $Q_\theta(s, a)$. The action a is obtained using the output of a neural network and is a deterministic action $\mu_\phi(s)$.

The loss is given by:

$$\begin{aligned} L_\theta &= \left[r_t + \gamma Q_\theta(s_{t+1}, \mu_\phi(s_{t+1})) - Q_\theta(s_t, \mu_\phi(s_t))\right]^2 \\ \implies \nabla L_\theta &= \nabla_\theta Q_\theta(s_t, a)|_{a=\mu_\phi(s_t)} \left(r_t + \gamma Q_{\theta^-}(s_{t+1}, \mu_\phi(s_{t+1}) - Q_\theta(s_t, \mu_\phi(s_t))\right) \end{aligned}$$

$\mu_\phi(s)$ is updated using policy gradients, which is a gradient update that modifies a parameterized policy in a direction that improves the return.

9 More on DRL

- Policy Gradient based methods: Gradient updates where a neural network models models a policy $\pi_\theta(a|s)$.
- Imitation Learning: Learns from demonstrations (similar to maximum likelihood)
- Hierarchical RL: Build a hierarchy of controllers in order build re-usable components in order to improve sample efficiency.
- Offline-RL: Learning from a fixed dataset
- Model-based RL: Building a model of the environment in order to artificially generate samples. * Exploration in RL