Reinforcement Learning

Sheldon Benard

June 19, 2019

1 Concepts and Notation

1.1 Definitions

Reinforcement Learning can be characterized by the following concepts:

- **Agent**: takes actions in an environment
- **Environment**: Markov Decision Process (MDP) is described by $(S,\ A,\ P^a_{ss'},\ R^a_{ss'},\ \gamma)$
  - **State-Space (S)**: States $s \in S$ are the nodes of the MDP - the possible situations that the agent can find itself in; these nodes provide a complete description of the world, with the information usually encoded in arrays, matrices, or tensors. The state-space S is the collection of all the possible states of the MDP. S can be **discrete** (finite number of states) or **continuous** (infinite number of states)
  - **Action-Space (A)**: Actions $a \in A$ are the possible moves that the agent can do. A is the set of all possible actions, and it can also be **discrete** (ex. in Mario, one can move right, left, or jump) or **continuous** (ex. giving a dosage to a patient)
  - **Transition Function ($P^a_{ss'}$)**: Dictates how the agent will move from state to state. $P^a_{ss'}$ can be **stochastic** and defines a probability distribution over the possible next states. Otherwise, $P^a_{ss'}$ is **deterministic**, whereby it will always yield the same s'
  - **Reward Function ($R^a_{ss'}$)**: Reward the agent receives after transitioning from s to s' due to action a. $R^a_{ss'}$ can be **stochastic**, and the reward received is a sample from some distribution. $R^a_{ss'}$ can also be **deterministic**
  - **Discount (γ)**: $\gamma \in [0, 1)$ and quantifies the difference in importance between immediate and future rewards. $\gamma \to 0$, then immediate rewards will be considered more and more. $\gamma \to 1$ will consider future rewards more and more. $\gamma < 1$ ensures convergence of algorithms for continuous tasks.

Environments have many classifications.

- **Stochastic vs. Deterministic**: Stochastic or deterministic reward and transition functions
- **Static vs. Dynamic**: Static environments do not change during the agents trajectory
- **Fully vs. Partially Observable**: Does the agent have a complete picture of the environment (i.e. all the environmental information) or not?
- **Single vs Multi-Agent**: Are there other apathetic, cooperative, or competitive agents in environment?
- **Known vs. Unknown**: Dynamics/Rules/Physics of the environment are known/unknown to the agent

In Reinforcement Learning, the task at hand can be episodic or continuous.

- **Episodic**: Tasks that come to an end (i.e. playing doom, there is an ultimate end to the level)
- **Continuous**: Tasks that do not come to an end (i.e. automated stock trading)

1.2 The Goal of Reinforcement Learning

Formally, the transition function is, given the initial state s and action a at time t, the probability of transitioning to the subsequent state s'

$$P^a_{ss'} = P(s_{t+1} = s' | s_t = s, a_t = a) \ (stochastic)$$

$$P^a_{ss'} = \begin{cases} 1, & \text{if } s' = \text{s(a)} \\ 0, & \text{else} \end{cases} \ (deterministic)$$

The expected reward is, given that we've started at state s and under action a at time t, and that the subsequent state is s' at time $t + 1$, the expectation of the reward feedback from the environment

$$E[R^a_{ss'}] = E[r_{t+1} | s_{t+1} = s', s_t = s, a_t = a]$$

Our goal in Reinforcement Learning is to maximize the expected cumulative reward. As such, we need to define the Return obtained starting at time t:

$$R_t = \sum_{k=0}^{T} r_{t+k+1}$$

where r_{t+k+1} is the reward feedback from the environment and T is the terminal time. However, $T \to \infty$ for continuous tasks and so the return summation could diverge. The agent would learn little in these cases, and thus we need to consider the Discounted Return at time t:

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$$

If $\gamma = 0$, the agent is myopic and takes into account only immediate rewards. As $\gamma \to 1$, immediate and future rewards are valued more equally. $\gamma \in [0, 1)$ ensures the rewards are always bounded. However, γ can equal 1, only if the task is not continuous (i.e. $T \not\to \infty$). NOTE: R_t from now on denotes this **Discounted Return**

Our goal is to maximize the expected cumulative discounted reward:

$$G_t = E[R_t]$$

To do this, the agent needs to learn a policy function which maps a given state to the optimal action:

$$\pi : S \to A$$

Policies can be divided into two types. **Deterministic** policies, given a state s, will simply yield an action, $a = \pi(s)$. **Stochastic** policies give a probability distribution over the different actions $\pi(s) = P(A|s)$.

1.3 Exploration vs. Exploitation

In systems that want to maximize the reward and acquire new knowledge simultaneous, an important trade-off exists between exploration and exploitation. Exploration sacrifices

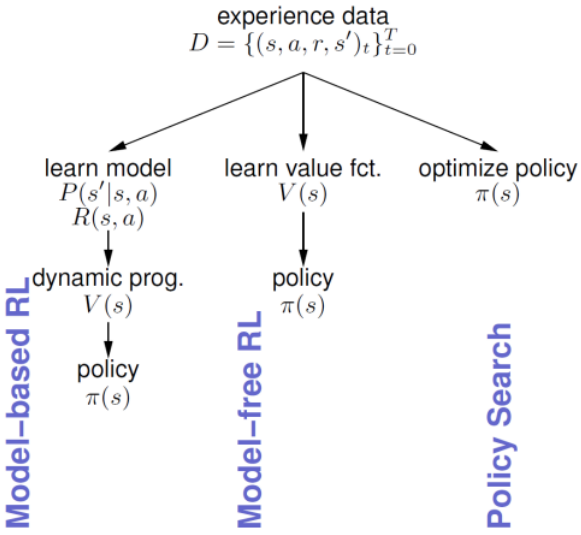
short-term reward for gaining knowledge and potentially maximizing reward in the long-term. Exploitation leverages current knowledge to make the best decision in the current context.

Consider the 2-armed bandit problem. We are presented with 2 casino slot machines, and we want to maximize the reward (i.e. profit). This can be visualized by a single state MDP, with 2 actions (we can pull the left or right arm).

Each slot machine's reward is stochastic and has a different probability of winning. As the agent, we need to figure out which machine has the better odds. However, if we always pick a machine at random, we will learn a lot about each machines probability but we will probably not make the most money we could. If you pick only one machine, you will learn nothing about the other machines probability and you potentially could have chosen the machine with worse odds. Thus, a strategy involving both exploration and exploitation is necessary.

In practice, in most reinforcement learning algorithms, we start with zero knowledge and so exploration is at a maximum. As the algorithm continues, we don't want to waste time exploring sub-optimal trajectories, and so we decrease exploration and increase exploitation.

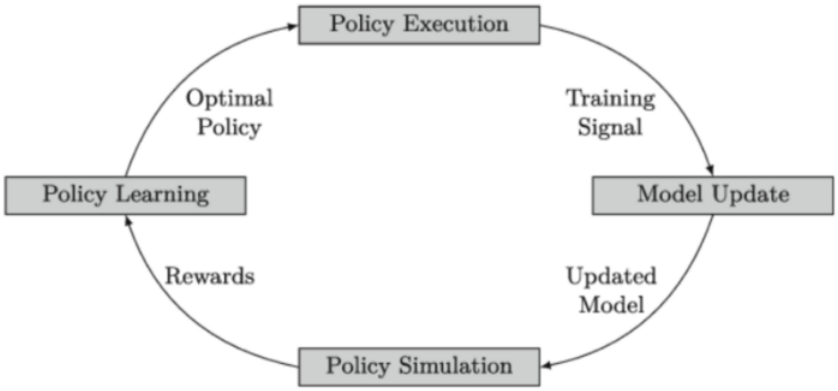
1.4 Reinforcement Learning Approaches



1.4.1 Model-Based Reinforcement Learning

The first approach to achieving the maximum expected cumulative reward is to use the environmental information to construct a model of the environment. With this model of reality, the agent can plan the best course of action.

More formally, in model-based reinforcement learning, given trajectory data $D = (s_t, a_t, r_{t+1}, s_{t+1})$ the algorithm attempts to estimate the transition function $P^a_{ss'}$ and the reward function $R^a_{ss'}$. The general flow of model-based Reinforcement is below.



A flow diagram of model-based RL

Model-based reinforcement learning is often more sample efficient than model-free reinforcement learning. We need less environmental interactions, in general, to learn a policy,

since we can leverage our model of the environment to simulate sequences of actions rather than needing to perform these actions directly on the environment. Moreover, with a model of reality, the potential to transfer what we've learned is higher, as we now can predict and generalize the dynamics of the environment.

However, the complexity of model-based reinforcement learning is significantly higher, since we need to learn a model to simulate the environment and a policy. Thus, this approach tends to be more computationally demanding.

1.4.2 Model-Free Reinforcement Learning

As suggested by the name, in model-free reinforcement learning we do not learn a model for the environment. We simply learn a policy for navigating the environment optimally.

Value-Based: The goal of the agent in value-based reinforcement learning is to optimize a value-function $V_\pi(s)$:

$$V_\pi(s) = E_\pi[r_{t+1} + \gamma r_{t+2} + \gamma r_{t+3} + \ldots | s_t = s] = E[R_t | s_t = s]$$

Given a state s and time t, the value of the state under a fixed policy π is the expected return when starting at s at time t and following the policy. Since we are after an optimal policy, and the optimal policy is a policy which maximizes the expected cumulative reward, we have that the optimal policy π^*, $V_{\pi^*}(s) \geq V_\pi(s)\ \forall \pi, s$. Thus, optimizing $V_\pi(s)$ pushes our policy towards the optimal policy.

Suppose we've optimized the value function, to derive an action for a given state, we simply choose the state with the largest value-function value from the set of next states.

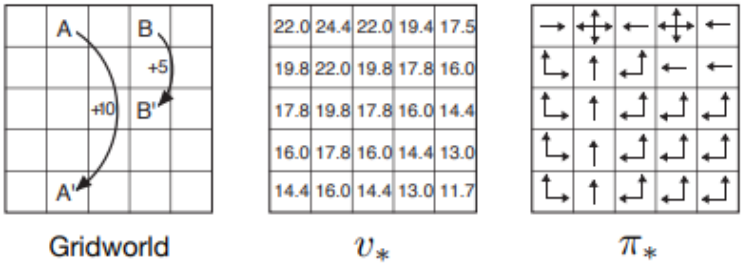


Figure 1: v_* is the optimal value-function values. π_* is the derived optimal policy

Policy-Based: In policy-based reinforcement learning, we parameterize the policy function π with the set of parameters θ, denoted π_θ. With an objective function $J(\theta)$, we update the policy directly via updating the parameters:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$$

Now, a question we must ask ourselves is: **when to learn?**. There are two main approaches as to when to optimize the value function and/or take policy gradients.

In **Monte Carlo** (MC) methods, the algorithm waits until the end of an episode (i.e. starting a state s_0 and continuing until terminating state s_T) and utilizes the entire return of the episode. Thus, the update is global and along full trajectories. As such, MC methods have high variance but low bias. **Temporal Difference** (TD) learns online after every step, not waiting for the end of the episode. TD learning methods have low variance but higher bias.

More details for Model-Free Reinforcement Learning are provided in the next section.

2 Model-Free Reinforcement Learning

2.1 Value-Based

To improve policies and evaluate their performance, many reinforcement learning methods utilize value function, which assign values to each state. As stated previously, the value function is defined as:

$$V_\pi(s) = E_\pi[r_{t+1} + \gamma r_{t+2} + \gamma r_{t+3} + \ldots | s_t = s] = E[R_t | s_t = s]$$

Moreover, we can define a similar function. The action-value function assigns a value for each state-action pair. Given state s and action a at time t, the value of the state-action pair under policy π is the expected return of starting at state s and performing action a and continuing with policy π:

$$Q_\pi(s, a) = E_\pi[\sum_{k=0}^{T} \gamma^k r_{t+k+1} | s_t = s, a_t = a] = E[R_t | s_t = s, a_t = a]$$

From these functions, we can derive the **Bellman Equations**. These equations allows us to define the value of states, using the value of other states. This lays the groundwork for reinforcement learning algorithms like Q-Learning and SARSA. The derivation of the Bellman Equations can be found at (*https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/*).

We can express the value function as:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_\pi(s')]$$

We can express the action-value function as:

$$Q_\pi(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s',a')]$$

Using expectations, we can express this more simply as:

$$Q_\pi(s,a) = E_\pi[r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1})|s_t = s, a_t = a]$$

From this formulation, two important ideas can be explored. First, we can derive an update rule in order to calculate the state-action values for each of the state-action pairs iteratively. Recall, in Temporal Difference learning, we learn after every step. In looking at the above expectation, we know more about the current Q value by using the true reward r_{t+1} and the next Q value. This is the TD-Target, and so the update rule for state s, action a at time t:

$$Q_\pi(s_t,a_t) = Q_\pi(s_t,a_t) + \alpha(\underbrace{r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1})}_{\text{TD-Target}} - Q_\pi(s_t,a_t))$$

Second, we have two policies: one policy for choosing the current action a_t at time t (the π in $Q_\pi(s_t,a_t)$), and one policy for choosing the next action a_{t+1} at time $t+1$ (the π in $Q_\pi(s_{t+1},a_{t+1})$) in the TD-Target. The former policy is called the **Behaviour Policy** and the latter is called the **Target Policy**. If the algorithm that uses this update rule uses the same policy function for the behaviour and the target, the algorithm is said to be **On-Policy**. Otherwise, the algorithm is **Off-Policy**.

2.1.1 Q-Learning

Q-learning is an off-policy algorithm. For the behaviour policy, the ϵ-greedy policy is typically chosen: with probability ϵ the policy chooses a random action and with probability $1-\epsilon$ the policy chooses the action which maximizes the Q value at state s. For the target policy, Q-learning chooses the greedy policy (action which maximizes the Q-value):

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

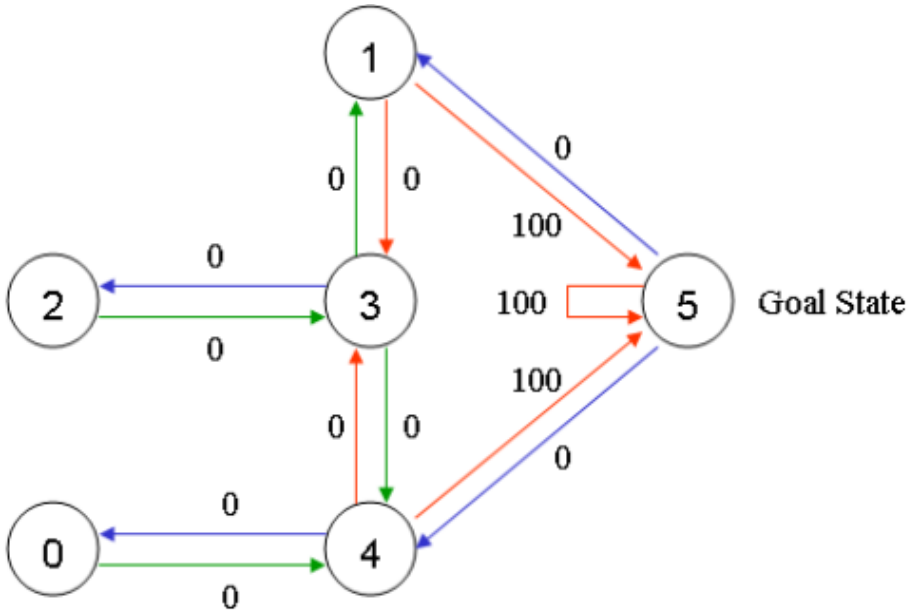
With that, the Bellman Equation and TD Learning update rule become:

$$Q_\pi(s,a) = E_\pi[r_{t+1} + \gamma max_{a'} Q(s_{t+1}, a')|s_t = s, a_t = a]$$

$$Q_\pi(s_t,a_t) = Q_\pi(s_t,a_t) + \alpha(r_{t+1} + \gamma max_{a'} Q(s_{t+1}, a') - Q_\pi(s_t,a_t))$$

An Example

For small action and state spaces, we can maintain a Q-table and iteratively compute the Q-values. Consider the following deterministic example (from *http://mnemstudio.org/path-finding-q-learning-tutorial.htm)*:



Set $\gamma = 0.8$ and we initialize the Q-table as a zero matrix. The Q-table is a matrix with $|S|$ rows and $|A|$ columns:

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

If we start in state 1 and our ϵ-greedy policy chooses action $1 \rightarrow 5$, we can update the Q-value: $Q(1,5) = r + \gamma max[Q(5,1), Q(5,5), Q(5,4)] = 100 + 0 = 100$

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Next, suppose we are in state 3 and by random selection the policy chooses action $3 \rightarrow 1$, then we can update the Q-value

$$Q(3,1) = r + \gamma \, max[Q(1,5), Q(1,3)] = 0 + (0.8)(100) = 80$$

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

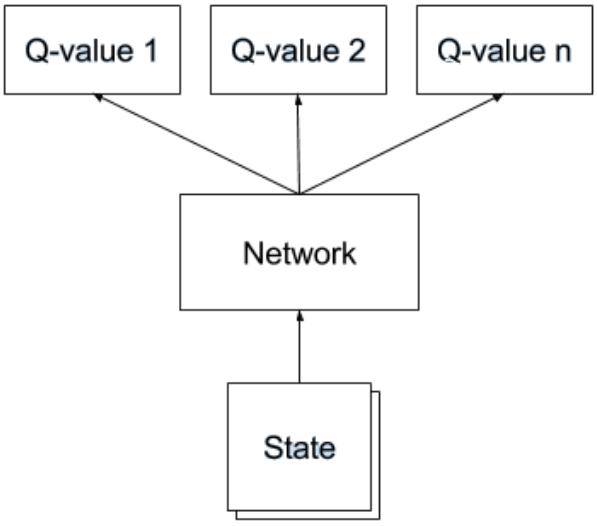
Ultimately, the Q table will converge to:

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 & 400 & 0 \\ 0 & 0 & 0 & 320 & 0 & 500 \\ 0 & 0 & 0 & 320 & 0 & 0 \\ 0 & 400 & 256 & 0 & 400 & 0 \\ 320 & 0 & 0 & 320 & 0 & 500 \\ 0 & 400 & 0 & 0 & 400 & 500 \end{bmatrix}$$

Now, we can derive our optimal policy, by choosing the action which maximizes the Q-value, given the state s.

Deep Q-Learning

However, action and state spaces can get very large (and even continuous) and so, the Q-tables can get very large or even become unrepresentable. Thus, we approximate the Q function using neural networks and avoid using tables altogether. We setup the network such that, given a state, it produces the Q-values for every action:



This is a regression problem, and the L2 loss function is:

$$L = (y_{true} - y_{pred})^2$$

y_{pred} is the predicted value for action a as predicted by the network. But, unlike supervised learning, we don't have true values given to us. However, looking at the Q-learning iteration $Q_\pi(s_t, a_t) = Q_\pi(s_t, a_t) + \alpha(r_{t+1} + \gamma max_{a'} Q(s_{t+1}, a') - Q_\pi(s_t, a_t))$, the TD-target consists of a true value: the reward r_{t+1}. Thus, we can set $y_{true} = r_{t+1} + \gamma max_{a'} NN(s_{t+1}, a')$, where $NN(s, a)$ is the output of the neural network for state s and selecting action a.

The hyper-parameters of the network are the discount γ, the learning rate α, and the exploration rate ϵ (if using ϵ-greedy policy). Typically, ϵ is set close to 1 at the start, since we want to encourage exploration at the start of training (as we don't know anything about the environment yet). As the network trains, ϵ is decayed, to discourage fruitless exploration of sub-optimal trajectories in the environment.

2.1.2 SARSA

Another value-based reinforcement learning algorithm is called **State-action-reward-state-action** (SARSA). This algorithm is very similar to Q-learning, however SARSA is **On-Policy**. The same update rule is followed:

$$Q_\pi(s_t, a_t) = Q_\pi(s_t, a_t) + \alpha(r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) - Q_\pi(s_t, a_t))$$

The only difference: the policy used at the current state to select an action is the same as the policy used at the next state to select an action in each iteration of the environment.

2.2 Policy-Based

A sequence of states, actions, and rewards in an environment is called a **trajectory**, which I'll denoted by τ.

$$\tau = S_0, A_0, R_1, S_1, A_1, R_2, \ldots$$

As stated previously, in policy-based reinforcement learning, we parameterize the policy function π with a parameter set θ, denoted π_θ. If we let $r(\tau)$ be the total reward for trajectory τ, we can define an objective function:

$$J(\theta) = E_{\pi_\theta}[r(\tau)]$$

For an update rule, we need the gradient of $J(\theta)$

$$\nabla J(\theta) = \nabla E_{\pi_\theta}[r(\tau)] = E_{\pi_\theta}[r(\tau)\nabla log(\pi_\theta(\tau))]$$

such that

$$\pi_\theta(\tau) = \underbrace{P(s_0)}_{\text{Ergodic Distribution}} \prod_{t=1}^{T} \pi_\theta(a_t|s_t)p(s_{t+1}, r_{t+1}|s_t, a_t)$$

This can be simplified to the following:

$$E_{\pi_\theta}[r(\tau) \sum_{t=1}^{T} \nabla \pi_\theta(a_t|s_t)]$$

If we replace $r(\tau)$ with the discounted return G_t, we get the REINFORCE algorithm which is an example of a Monte Carlo algorithm:

$$E_{\pi_\theta}[\sum_{t=1}^{T} G_t \nabla \pi_\theta(a_t|s_t)]$$

Now, the update rule is:

$$\theta_{t+1} = \theta_t + \alpha E_{\pi_\theta}[\sum_{t=1}^{T} G_t \nabla \pi_\theta(a_t|s_t)]$$

2.3 Hybrid Models

Instead of only optimizing value-functions or taking policy gradients, we can devise an algorithm that leverages both value- and policy-based methods. In **Actor-Critic** learning models, the agent is comprised of an actor and a critic. The actor learns a policy π using the feedback of the critic. The critic learns the value function $V(s)$. The **Advantage Actor-Critic** (A2C) algorithm introduces an *advantage* function. The advantage, $A(s) = r + \gamma V(s') - V(s)$, determines how advantageous it is to be in a certain state. Thus, we don't need to also learn the Q-function.

In A2C, we have two networks, with two different loss functions. The value loss function uses sum-squared error and the policy loss function uses the log-loss error:

$$L_{value} = \sum (r + \gamma V(s') - V(s))^2$$

$$L_{policy} = -log(\pi(a|s))A(s)$$

3 Optimizations

3.1 Experience Replay

Deep Neural Networks tend to overfit to current episodes quite easily, and thus the agent can fail to behave optimally given various experiences. Introduced in 1993, experience replay proposes that we store $\{s_t, a_t, r_{t+1}, s_{t+1}\}$ and then use mini-batches to update the networks. The number of experiences we store in the buffer (i.e. the buffer size) is a hyper-parameter. This technique aims to reduce correlation between data points when updating the neural network, increase learning speed, and reuse past experiences to protect against forgetting important experiences.

4 References

- https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d
- https://towardsdatascience.com/policy-networks-vs-value-networks-in-reinforcement-learning-da2776056ad2
- https://towardsdatascience.com/self-learning-ai-agents-part-ii-deep-q-learning-b5ac60c3f47

- https://medium.com/deep-math-machine-learning-ai/ch-13-deep-reinforcement-learning-deep-q-learning-and-policy-gradients-towards-agi-a2a0b611617e
- https://katatrepsis.com/2011/08/06/evolution-of-superstition-2/
- https://en.wikipedia.org/wiki/Reinforcement_learning
- https://medium.freecodecamp.org/an-introduction-to-reinforcement-learning-4339519de419
- http://ipvs.informatik.uni-stuttgart.de/mlr/wp-content/uploads/2013/12/06-ModelBased.pdf
- https://medium.com/the-official-integrate-ai-blog/understanding-reinforcement-learning-93d4e34e5698
- https://medium.freecodecamp.org/a-brief-introduction-to-reinforcement-learning-7799af5840db
- https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/
- https://en.wikibooks.org/wiki/Artificial_Intelligence/AI_Agents_and_their_Environments
- http://incompleteideas.net/book/RLbook2018trimmed.pdf
- https://medium.com/@henrymao/reinforcement-learning-using-asynchronous-advantage-actor-critic-704147f91686
- https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b