Companion to Reinforcement Learning

Rohan Kumar

Contents

0 Notation

0.1 Data

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_M \end{pmatrix} : \text{ data point corresponding to a column vector of } M \text{ features}$$

$$\overline{\boldsymbol{x}} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ ... \\ x_M \end{pmatrix} : \text{ concatenation of 1 with the vector}\boldsymbol{x}$$

$$\boldsymbol{X} = \begin{pmatrix} x_{1,1} & ... & x_{1,N} \\ ... & ... & ... \\ x_{M,1} & ... & x_{M,N} \end{pmatrix} : \text{ dataset consisting of } N \text{ data points and } M \text{ features}$$

$$\overline{\boldsymbol{X}} = \begin{pmatrix} 1 & ... & 1 \\ x_{1,1} & ... & x_{1,N} \\ ... & ... & ... \\ x_{M,1} & ... & x_{M,N} \end{pmatrix} : \text{ concatenation of a vector of 1's with the matrix } \boldsymbol{X}$$

$$y = : \text{ output target (regression) or label (classification)}$$

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ ... \\ y_N \end{pmatrix} : \text{ vector of outputs for a dataset of } N \text{ points}$$

$$\boldsymbol{x}_* = : \text{ test input / unknown input}$$

$$\boldsymbol{y}_* = : \text{ predicted output}$$

$$N = : \text{ Number of data points in the dataset}$$

$$M = : \text{ Number of a features in a data point}$$

$$\boldsymbol{w} = \begin{pmatrix} w_1 \\ w_2 \\ ... \\ w_M \end{pmatrix}$$

$$\boldsymbol{w}^T = (w_1, w_2, ..., w_M) \text{ or } (w_0, w_1, w_2, ..., w_M) \ w_0 \text{ multiplies the first entry of } \overline{\boldsymbol{x}} \text{ (bias)}$$

$$* = \text{optimal policy and or function}$$

Note: bold symbols represents a vector

1 Introduction

Reinforcement Learning is an area of machine learning inspired by behavioural psychology, concerned with how software **agents** ought to take **actions** in an **environment** so as to maximize some notion of cumulative **reward**.

2 Markov Processes

Markov processes are important to model environment dynamics and to model this we will use stochastic processes and the two important assumptions we will consider are the Markovian Assumption and the Stationary Assumption.

2.1 Unrolling the Problem

If we consider the problem of reinforcement learning where we take an action based on the state and receive a reward then we can unroll the loop into a sequence of states, actions and rewards:

$$s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2$$

This sequence forms a stochastic process (due to some uncertainty in the dynamics of the process). This is because we don't know the underlying state of s_0 and therefore we don't know the correct action a_0 that will lead to the next optimal state.

2.2 Stochastic Processes

Processes are rarely arbitrary and very often exhibit structure. The laws of the process do not change and the short history is sufficient to predict the future.

2.2.1 Definition

We will define a stochastic process with respect to states. We let S be our set of states and we define the stochastic dynamics to be $P(s_t|s_{t-1}, ..., s_0)$ which in general expresses some conditional distribution over the current state given the past states.

2.2.2 Problem

The problem that arises with this is that if we have a process that is very long then this conditional distribution that produces the state at time step t could depend on a number of states before. Expressing a large number of previous states results in an intractable problem. If the process was infinitely long and every state depended on everything that happened before then the problem cannot be expressed.

The solution to this is to assume that the process is stationary: the dynamics do not change over time. The other assumption we need to make is the Markov assumption which is that the current state depends only on a finite history of past states.

2.3 K-order Markov Process

Here we use the Markov Assumption and assume that the last k states is sufficient. In a first-order markov process we assume that only the last state is relevant and sufficient.

By default a Markov Process refers to a first order process

$$P(s_t|s_{t-1}, s_{t-2}, ..., s_0) = P(s_t|s_{t-1}) \forall t$$

and expresses the stationary assumption that

$$P(s_t|s_{t-1}) = P(s_{t'}|s_{t'-1}) \forall t'$$

The stationary assumption means that the conditional distribution is going to be the same regardless of which time step we consider.

The advantage of this is we can now specify an entire process with a single concise conditional distribution

$$P(s'|s)$$

2.4 Non-Markovian and Non-Stationary Processes

If the process is not Markovian and/or not stationary then we can add new state components until the dynamics are Markovian and stationary. For example considering the dynamics in robotics we have state $< x, y, z, \theta >$, this is not stationary when velocity varies. The solution is to add velocity to the state description as $< x, y, z, \theta, \dot{x}, \dot{y}, \dot{z}, \dot{\theta} >$

The problem with this solution is that adding components to the state description to force a process to be Markovian and stationary may significantly increase computational complexity. The solution to this would be to try to find the smallest state description that is self-sufficient (Markovian and stationary).

2.5 Inference in Markov Processes

Assuming we have a Markovian Process that is stationary, we want to do inference to predict what will be the value of some future state and the reason this is important is because in a Markov decision process and more generally in reinforcement learning the goal is to select actions that will influence future states and get us into states that will have high rewards. Therefore if we can predict what the future state is going to be then we can select some good actions.

Predicting a state k time steps into the future:

$$P(s_{t+k}|s_t)$$

To compute this we perform the following which takes advantage of the chain rule in probability theory to expand this into a product to sum out all the intermediate states:

$$P(s_{t+k}|s_t) = \sum_{s_{t+1}\ldots s_{t+k-1}} \prod_{i=1}^{k} P(s_{t+i}|s_{t+i-1})$$

If we have discrete states (finitely many states) then we can represent the conditional distribution as a matrix and here we use the letter T to indicate a transition matrix. So we let T be a $|S| \times |S|$ matrix representing $P(s_{t+1}|s_t)$. Then $P(s_{t+k}|s_t) = T^k$. What we're doing here is that we are essentially taking the product of that matrix k times. Complexity: $O(k|S|^3)$.

3 Markov Decision Processes

For a Markov Decision Process we will augment the Markov process with Actions and Rewards. Our goal is to select actions that will influence the future states in a good way. The choice of actions depends on the current state. The reward depends on the current state and action, this quantifies how good it is to be in a certain state and execute a certain action.

The current assumptions we need to maintain are it being a stochastic process, sequential process, fully observable states, complete model and the process to be discrete.

3.1 Definition

We define the set of states to be S and the set of actions to be A. We consider a transition model $P(s_t, s_{t-1}, a_{t-1})$ that corresponds to the stochastic process model with the addition of an action. We have the reward model $R(s_t, a_t)$ and its discount factor γ. Finally we have a horizon h, which is the time horizon/number of time steps. The goal is to find an optimal policy, a mapping from states to actions.

3.2 Rewards

The reward is a real number ($r_t \in \mathbb{R}$) that we are going to try and maximize. It is essentially a numerical signal that indicates how good the state and action is at every time step. We can express the reward function as $R(s_t, a_t) = r_t$.

Another common assumption we make here is that the reward function is stationary where $R(s_t, a_t)$ is the same at every time step ($\forall t$). This does not mean the reward is the same at every time step just the function is the same. The exception to this is the terminal reward function is often different (e.g. in a game, 0 reward at each turn but +1/-1 at the end for winning/losing).

The goal is to maximize the sum of the rewards $\sum_t R(s_t, a_t)$.

3.2.1 Discounted/Average Rewards

If process is infinite and my rewards are always positive then $\sum_t R(s_t, a_t)$ approaches infinity. This becomes an issue.

The first solution here is to consider discounted rewards. We introduce a discount factor $0 \leq \gamma \leq 1$. The idea here is that we want to discount the reward function by γ^t where t is the time step. We can represent the finite utility as $\sum_t \gamma^t R(s_t, a_t)$. The rewards further in the future will be discounted more. We can think of γ as an inflation rate of $\frac{1}{\gamma-1}$. The intuition here is that we prefer utility sooner than later.

The second solution is to average the rewards. in some problems we don't want to discount earning a reward thousands of time steps into the future rather it should have the same reward as earning the utility now. In this case we can average the reward. This is computationally more complicated and will not currently be considered in this literature.

4 Dynamic Programming Algorithms for Solving MDPs

4.1 Policy and Policy Optimization

A policy is a formal definition for how to select actions. We will denote a policy using π. We are mapping states to actions so we can represent this as $\pi(s_t) = a_t$. The goal in reinforcement learning and markov processes is to come up with this policy π. We are going to make the assumption that we have fully observable states, so therefore we can condition the choice of action basely on the current state s_t.

We now define the expected utility as:

$$V^\pi(s_0) = \sum_{t=0}^{h} \gamma^t \sum_{s_t} P(s_t|s_0, \pi) R(s_t, \pi(s_t))$$

Here we have a discounted cumulative reward where we have a sum with respect to all time steps from 0 to h (horizon). Then our reward at each time step is $R(s_t, \pi(s_t))$ which depends on the current state. That state has some uncertainty in it and it really depends on the transition dynamics that are stochastic and therefore we have some expectation $P(s_t|s_0, \pi)$.

We represent the optimal policy as π^* which is the policy with the highest expected utility where the following holds:

$$V^{\pi^*}(s_0) \geq V^\pi(s_0) \forall \pi$$

There are several classes of algorithms for policy optimizations:

- Value iteration
- Policy iteration
- Linear programming

- Search techniques

Computation may be done offline: before the process starts or online: as the process evolves.

4.2 Value Iteration

The way this algorithm works is by optimizing / selecting actions in reverse order. To find what the best action at every step we can use dynamic programming that will work backwards and optimize the last decision then the second last decision then so on.

We can demonstrate this mathematically as follows:

We start at the last time step h.

$$V(s_h) = max_{a_h} R(s_h, a_h)$$

To optimize the last decision is easy we look at all possible actions and pick the one that yields the highest reward.

Value with one time step left:

$$V\left(s_{h-1}\right) = \max_{a_{h-1}} R\left(s_{h-1}, a_{h-1}\right) + \gamma \sum_{s_h} P(s_h \mid s_{h-1}, a_{h-1}) V\left(s_h\right)$$

Here we are taking into account the reward for the second last time step as well as the last time step and sum the rewards. We have the second last reward $\max_{a_{h-1}} R\left(s_{h-1}, a_{h-1}\right)$ plus the discount factor γ multiplied into the expectation $P(s_h \mid s_{h-1}, a_{h-1})$ with respect to what the last state will be times the value that I can earn in the last state $V(s_h)$. Now we simply select a_{h-1} to maximize this where $V(s_h)$ is already maximized.

Similarly value with two time steps left:

$$V\left(s_{h-2}\right) = \max_{a_{h-2}} R\left(s_{h-2}, a_{h-2}\right) + \gamma \sum_{s_{h-1}} P(s_{h-1} \mid s_{h-2}, a_{h-2}) V\left(s_{h-1}\right)$$

If we do this for the entire planning horizon we can generalize to Bellman's equation:

$$V(s_t) = max_{a_t} R(s_t, a_t) + \gamma \sum_{s_t+1} P(s_{t+1} \mid s_t, a_t) V(s_t + 1)$$

Now we can represent the equation to get the optimal action to construct the optimal policy is:

$$a_t^* = argmax_{a_t} R(s_t, a_t) + \gamma \sum_{s_t+1} P(s_{t+1} \mid s_t, a_t) V(s_t + 1)$$

Finite Horizon When h is finite we have a non-stationary optimal policy where the best action is different at every time step. The intuition here is that the best action varies with the amount of time left.

Infinite Horizon When h is infinite, stationary is the optimal policy. We pick the same best action at each time step. The intuition here is that the same amount of time (infinite) at each time step, therefore we pick the same best action. The main problem with this is that value iteration will do an infinite number of iterations.

The solution is that we can perform value iteration using a finite n since we are using a discount factor γ. After n time steps the rewards are scaled down by γ^n and for a large enough n, the rewards become insignificant since $\gamma^n \to 0$. For this we need to pick a large enough n, run value iteration for n steps and execute policy found at the n^{th} iteration.

4.2.1 Algorithm

Algorithm 1: Value Iteration MDP

```
$V_0^* \leftarrow max_a R(s,a) \forall s$;
for $t \leftarrow 1$ to $h$ do
    $V_t^*(s) \leftarrow max_a R(s,a) + \gamma \sum_{s'} P(s'|s,a) V_{t-1}^* \forall s$
end
return $V^*$
```

We get the optimal policy π^*:

- $t = 0$; $\pi_0^*(s) \leftarrow argmax_a R(s,a) \forall s$
- $t > 0$; $\pi_t^*(s) \leftarrow argmax_a R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_{t-1}^*(s') \forall s$

In this case t indicates the number of time steps to go till the end of process and π^* is non stationary.

We can express this in matrix form as follows. Let:

- R^a: $|S| \times 1$ column vector of rewards for a.
- V_t^*: $|S| \times 1$ column vector of state values.
- T^a: $|S| \times |S|$ matrix of transition probabilities for a.

Algorithm 2: Value Iteration MDP

```
$V_0^* \leftarrow max_a R^a$;
for $t \leftarrow 1$ to $h$ do
    $V_t^* \leftarrow max_a R^a + \gamma T^a V_{t-1}^*$
end
return $V^*$
```

Even when the horizon is infinite we can perform finitely many iterations using the principle of contraction and convergence. We can modify the algorithm to stop when $||V_n - V_{n-1}|| \leq \epsilon$.

Algorithm 3: Value Iteration MDP

```
$V_0^* \leftarrow max_a R^a$;
$n \leftarrow 0$;
do
    $n \leftarrow n + 1$;
    $V_n^* \leftarrow max_a R^a + \gamma T^a V_{n-1}^*$;
while $||V_n - V_{n-1}||_\infty \leq \epsilon||$;
return $V^*$
```

The complexity of the value iteration algorithm is that each iteration is $O(|S|^2|A|)$.

4.3 Policy Evaluation

Policy evaluation computes the value functions for a policy π. If we consider policy evaluation for an infinite horizon where we let $h \to \infty$ then $V_h^\pi \to V_\infty^\pi$ and $V_{h-1}^\pi \to V_\infty^\pi$

Then the equation for policy evaluation becomes:

$$V_\infty^\pi(s) = R\left(s, \pi_\infty(s)\right) + \gamma \sum_{s'} \mathrm{P}\left(s' \mid s, \pi_\infty(s)\right) V_\infty^\pi\left(s'\right) \forall s$$

Bellman's equation:

$$V_\infty^*(s) = \max_a R(s,a) + \gamma \sum_{s'} \mathrm{P}\left(s' \mid s, a\right) V_\infty^*\left(s'\right)$$

Representing the policy evaluation in matrix form where:

- R: $|S| \times 1$ column vector of state rewards for π.
- V: $|S| \times 1$ column vector of state rewards for π.
- T: $|S| \times |S|$ column vector of state rewards for π.

then we get:

$$V = R + \gamma T V$$

If we consider Bellman's equation for a set of non-linear equations then we get:

$$V^* = max_a R^a + \gamma T^a V^*$$

To find the value of the policy we solve the system. We can do it using the following methods:

- Gaussian Elimination: $(I - \gamma T)V = R$
- Compute Inverse: $V = (I - \gamma T)^{-1} R$
- Iterative Methods

– Value Iteration (Richardson Iteration)

– Repeat $V \leftarrow R + \gamma TV$

4.3.1 Induced Policy

Using the infinite value iteration algorithm for an infinite horizon we can derive the station policy $\pi_n(s)$ based on the computed optimal V_n using:

$$\pi_n(s) = argmax_a R(S,a) + \gamma \sum_{s'} P(s' \mid s,a) V_n(s')$$

We can further prove that $||V^{\pi_n} - V^*||_\infty \leq \frac{2\epsilon}{1-\gamma}$. That is how far V^{π_n} (value of the policy) is from V^*.

4.4 Policy Iteration

Instead of doing value iteration where we optimize the value function and extract the induced policy, we can directly optimize the policy using **policy iteration**. An advantage of policy iteration is that the number of iterations required tends to be small in practice.

4.4.1 Algorithm

The algorithm for policy iteration alternates between 2 steps

1. Policy Evaluation

$$V^\pi(s) = R(s,\pi(s)) + \gamma \sum_{s'} P(s' \mid s,\pi(s)) V^\pi(s') \forall s$$

2. Policy Improvement

$$\pi(s) \leftarrow argmax_a R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^\pi(s') \forall s$$

Algorithm 4: Policy Iteration MDP

```
Initialize $\pi_0$ to any policy. ;
$n \leftarrow 0$ ;
do
    Eval: $V_n = R^{\pi_n} + \gamma T^{\pi_n} V_n$ ;
    Improve: $\pi_{n+1} \leftarrow argmax_a R^a + \gamma T^a V_n$ ;
    $n \leftarrow n + 1$
while $\pi_{n+1} = \pi_n$;
return $\pi_n$
```

The complexity of each iteration in policy iteration is $O(|S|^3 + |S|^2|A|)$, however, it has linear-quadratic convergence which requires fewer iterations than value iteration.

4.4.2 Modified Policy Iteration Algorithm

We want to find a modified algorithm that has a complexity similar to that of value iteration and a convergence similar to that of the original policy iteration algorithm. This modified algorithm follows the same principle as policy iteration, however, in order to reduce the cost per iteration it will do a partial policy evaluation.

The algorithm for policy iteration alternates between 2 steps

1. Partial Policy Evaluation. Repeat k times:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) V^\pi(s') \forall s$$

2. Policy Improvement

$$\pi(s) \leftarrow argmax_a R(s, a) + \gamma \sum_{s'} P(s' \mid s, a) V^\pi(s') \forall s$$

Algorithm 5: Modified Policy Iteration MDP

```
Initialize $\pi_0$ and $V_0$ to anything. ;
$n \leftarrow 0$ ;
do
    Eval: Repeat $k$ times ;
        $V_n = R^{\pi_n} + \gamma T^{\pi_n} V_n$ ;
    Improve: $\pi_{n+1} \leftarrow argmax_a R^a + \gamma T^a V_n$ ;
        $V_{n+1} \leftarrow max_a R^a + \gamma T^a V_n$ ;
    $n \leftarrow n + 1$ ;
while $||V_n - V_{n-1}||_\infty \leq \epsilon$;
return $\pi_n$
```

The complexity of each iteration is now $O(k|S|^2 + |S|^2|A|)$ which is less than the cost for the original policy iteration and number of iterations for convergence will be linear-quadratic convergence (more iterations than the original policy iteration but less than that of value iteration).

5 Reinforcement Learning

5.1 Introduction

We can represent the problem of reinforcement learning as a MDP. We are going to relax the assumption of the reward model from being stationary to being stochastic and modify the goal. If we consider the Reinforcement learning problem the policy decides what action to commit and the transition and reward model determine the state and reward. The main difference is that we don't necessarily know what the state model is, or what rewards the environment uses. So formally we can define reinforcement learning as follows:

- States: $s \in S$
- Actions: $a \in A$
- Rewards: $r \in \mathbb{R}$
- Transition Model: $P(s_t \mid s_{t-1}, a_{t-1})$ (unknown)
- Reward Model: $P(r_t \mid s_t, a_t)$
- Discount Factor: $0 \leq \gamma \leq 1$
- Horizon: h

Goal: Find optimal policy π^* such that

$$\pi^* = argmax_\pi \sum_{t=0}^{h} \gamma^t E_\pi[r_t]$$

We will now go in depth about algorithms to achieve this policy with the constraints. The idea is to learn an optimal policy while interacting with the environment. The intuition behind this is that we are going to observe the state's actions and rewards at every step and then this gives us samples effectively from the unknown models. When we have a large enough sample there is presumably enough information to recover the model or come up with policy that would be optimal with respect to these unknown models.

5.2 RL Agent Architectures

Most agents are a combination of 3 important components.

Model: $P(s' \mid s, a)$, $P(r \mid s, a)$ - Some agents are going to learn a model explicitly so they are considered model based agents. This involves transition dynamics and reward distribution.

Policy: $\pi(s)$ - Some agents are also explicitly going to model a policy, so that's the choices that the agents make in different states.

Value Function: $V(s)$ - Some agents will explicitly write the value function which is simply the expected total sum of the rewards.

5.2.1 Categorizing RL Agents

- **Value Based**: No policy, value function
- **Policy Based**: Policy, no value function
- **Actor Critic**: Policy, value function
- **Model Based**: Transition and Reward Model
- **Model Free**: No transition and no reward model

5.3 Model Free Evaluation

Given a policy π, estimate $V^\pi(s)$ without any transition or reward model.

5.3.1 Monte Carlo Evaluation

We know that in theory we can estimate the value at state s by taking the expectation with respect to π of the discounted sum of the rewards obtained at every time step. Since we don't have the transition and reward model we can instead obtain sample approximations where I can replace the expectation by an average.

$$\begin{aligned} V^\pi(s) &= E_\pi[\sum_t \gamma^t r_t] \\ &\approx \frac{1}{n(s)} \sum_{k=1}^{n(s)} \left[\sum_t \gamma^t r_t^{(k)} \right] \end{aligned}$$

Algorithm:

Let G_k be a one-trajectory Monte Carlo target (execute the policy once).

$$G_k = \sum_t \gamma^t r_t^{(k)}$$

Now we want to have a running average, that I can update gradually one sample at a time we can rewrite the approximate value function. This leads to the following incremental update:

$$V_n^\pi(s) \leftarrow V_{n-1}^\pi(s) + \alpha_n \left(G_n - V_{n-1}^\pi(s) \right)$$

where α_n is the learning rate $\frac{1}{n(s)}$.

5.3.2 Temporal Difference Evaluation

For temporal difference evaluation we will consider the equation for policy evaluation where the value of policy π is the expected immediate reward plus the discount factor times an expectation with respect to future rewards. Now considering the reinforcement learning problem, we no longer have distributions to compute this expectation. What instead we can do is take a one sample approximation and replace the expectation with one sample approximation for immediate reward and future reward.

$$\begin{aligned} V^\pi(s) &= E[r \mid s, \pi(s)] + \gamma \sum_{s'} \mathrm{P}\left(s' \mid s, \pi(s)\right) V^\pi\left(s'\right) \\ &\approx r + \gamma V^\pi\left(s'\right) \end{aligned}$$

Using the one sample update we can represent the incremental update as follows:

$$V_n^\pi(s) \leftarrow V_{n-1}^\pi(s) + \alpha_n \left(r + \gamma V_{n-1}^\pi\left(s'\right) - V_{n-1}^\pi(s) \right)$$

Theorem: If α_n is appropriately decreased with number of times a state is visited then $V_n^\pi(s)$ converges to correct value.

Sufficient conditions for α_n

1. $\sum_n \alpha_n \to \infty$
2. $\sum_n (\alpha_n)^2 < \infty$

Often $\alpha_n(s) = 1/n(s)$ where $n(s) = \#$ of times s is visited

Algorithm 6: TD Evaluation (π, V^π)

```
do
    Execute $\pi(s)$ ;
    Observe $s'$ and $r$;
    Update Counts: $n(s) \leftarrow n(s) + 1$ ;
    Learning Rate: $\alpha \leftarrow 1/n(s)$ ;
    Update value: $V_n^\pi(s) \leftarrow V_{n-1}^\pi(s) + \alpha_n \left(r + \gamma V_{n-1}^\pi(s') - V_{n-1}^\pi(s)\right)$ ;
    $s \leftarrow s'$ ;
while Until convergence of $V^\pi$;
return $V^\pi$
```

Comparing Monte Carlo and Temporal Difference evaluation we see the following:

Monte Carlo Evaluation:

- Unbiased estimate
- High Variance
- Needs many trajectories

Temporal Difference Evaluation

- Biased estimate
- Lower Variance
- Needs less trajectories

5.4 Model Free Control

Now we will consider how to update a policy. We still do not know what the model is but we have some samples that will guide us. Instead of evaluating the state value function $V^\pi(s)$, evaluate the state-action value function $Q^\pi(s, a)$.

We define $Q^\pi(s, a)$ as the value of executing a followed by π. The difference here is that the value function was the value of just executing the policy at every step but now with the Q function we are going to allow the first action to be different than the one prescribed by policy π.

$$Q^\pi(s, a) = E[r \mid s, a] + \gamma \sum_{s'} P(s' \mid s, a) V^\pi(s')$$

In the Q function, the reward is conditioned on the action and we are going to consider all the possible actions. We do not necessarily take the maximal action or the action prescribed by a policy; we want to consider all possible actions. Now based on the Q function we can compare Q values to find good ways to improve or select actions.

Greedy policy π':

$$\pi'(s) = argmax_a Q^{\pi}(s,a)$$

5.4.1 Bellman's Equation

For the optimal state-action value function $Q^*(s,a)$

We consider all possible actions $E[r \mid s,a]$ and then follow the optimal policy by taking the expectation with respect to what will be the best in the future (maximizing with respect to a').

$$Q^*(s,a) = E[r \mid s,a] + \gamma \sum_{s'} P(s' \mid s,a) max_{a'} Q^*(s',a')$$

5.4.2 Monte Carlo Control

Let G_n^a be a one-trajectory Monte Carlo target.

$$G_n^a = r_0^n + \sum_{t=1} \gamma^t r_t^n$$

where r_0^n is the initial action and $\sum_{t=1} \gamma^t r_t^n$ is the policy π.

We can now alternate between:

Policy evaluation

$$Q_n^{\pi}(s,a) \leftarrow Q_{n-1}^{\pi}(s,a) + \alpha_n \left(G_n^a - Q_{n-1}^{\pi}(s,a) \right)$$

Policy improvement

$$\pi'(s) \leftarrow \operatorname{argmax}_a Q^{\pi}(s,a)$$

The idea here is that we are evaluating our policy by retrieving a new sample at every time step n. We compare the sample to the current estimate, take the difference and then make a step that will correct based on that difference with learning rate α_n. So each time a new trajectory is obtained a new G_n^a is obtained and this update allows us to gradually evaluate the policy. Now that we have a $Q_n^{\pi}(s,a)$ estimate, we know the value of different actions and we can simply change the policy to select the best action using policy improvement.

5.4.3 Temporal Difference Control

Temporal Difference control is more efficient and generally used in practice. We can approximate the Q-function with a one sample approximation as follows:

$$\begin{aligned} Q^*(s,a) &= E[r \mid s,a] + \gamma \sum_{s'} \mathrm{P}(s' \mid s,a) \max_{a'} Q^*(s',a') \\ &\approx r + \gamma \max_{a'} Q^*(s',a') \end{aligned}$$

We will get many samples over time so each time we are going to gradually update the Q function using the following incremental update:

$$Q_n^*(s,a) \leftarrow Q_{n-1}^*(s,a) + \alpha_n \left(r + \gamma \max_{a'} Q_{n-1}^*(s',a') - Q_{n-1}^*(s,a) \right)$$

5.4.4 Q-Learning Algorithm

We can leverage the incremental update concept using in temporal difference control to derive the algorithm for Q-Learning.

```
Algorithm 7: QLearning($s, Q^*$)
do
    Select and execute $a$ ;
    Observe $s'$ and $r$;
    Update Counts: $n(s,a) \leftarrow n(s,a) + 1$ ;
    Learning Rate: $\alpha \leftarrow 1/n(s,a)$ ;
    Update Q value: $Q_n^*(s,a) \leftarrow Q_{n-1}^*(s,a) + \alpha_n \left( r + \gamma \max_{a'} Q_{n-1}^*(s',a') - Q_{n-1}^*(s,a) \right)$ ;
    $s \leftarrow s'$ ;
while Until convergence of $Q^*$;
return $Q^*$
```

How do we select a?

- If an agent always chooses the action with the highest value then it is exploiting. The learned model is not the real model and may lead to suboptimal results. (higher rewards in the short term)
- By taking random actions (pure exploration) an agent may learn the model.

Common Exploration Methods:

- ϵ-greedy: With a probability ϵ execute random action or otherwise execute best action.

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

- Boltzmann Exploration: probability of choosing any action according to the following formula which tends to prefer actions with high Q values.

$$\mathrm{P}(a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_a e^{\frac{Q(s,a)}{T}}}$$

6 Deep Q Networks

The objective of Deep Q Networks is to estimate the Q function using a neural network. We have seen that we can train a neural network using gradient descent. So the objective for a deep Q network is to train the Q Learning algorithm using gradient descent for linear and non-linear objectives.

6.1 Gradient Q-Learning

We have seen the following:

- Q-value estimate: $Q_{\boldsymbol{w}}(s, a)$
- Target: $r + \gamma \; max_{a'} Q_{\bar{\boldsymbol{w}}}(s', a')$

We want to minimize the squared error between the Q-value estimate and target. We use the squared error function as our loss function that we want to minimize where we vary $\boldsymbol{w}$ to minimize our error and $\bar{\boldsymbol{w}}$ remains fixed.

Squared Error:

$$Err(\boldsymbol{w}) = \frac{1}{2}[Q_{\boldsymbol{w}}(s, a) - r - \gamma \; max_{a'} Q_{\bar{\boldsymbol{w}}}(s', a')]^2$$

Gradient

$$\frac{\partial Err}{\partial \boldsymbol{w}} = \left[Q_{\boldsymbol{w}}(s, a) - r - \gamma \max_{a'} Q_{\bar{\boldsymbol{w}}}\left(s', a'\right)\right] \frac{\partial Q_{\boldsymbol{w}}(s, a)}{\partial \boldsymbol{w}}$$

The gradient is intuitively the gradient of the Q function times the temporal difference.

6.1.1 Gradient Q-Learning Algorithm

Algorithm 8: Gradient QLearning(s)

```
Initialize weights $\boldsymbol{w}$ uniformly at random in $[-1, 1]$ Observe current state $s$. do
    Select and execute action $a$ ;
    Receive immediate reward $r$ ;
    Observer new state $s'$ ;
    Gradient: $\frac{\partial Err}{\partial \boldsymbol{w}} = [Q_{\boldsymbol{w}}(s, a) - r - \gamma \max_{a'} Q_{\boldsymbol{w}}(s', a')] \frac{\partial Q_{\boldsymbol{w}}(s,a)}{\partial \boldsymbol{w}}$ ;
    Update weights: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \frac{\partial Err}{\partial \boldsymbol{w}}$ Update state: $s \leftarrow s'$
while Until number of epochs;
```

In this case we notice that the target value does not have fixed weights. This can lead to variable divergence. If we consider linear gradient Q learning where $Q_{\boldsymbol{w}}(s, a) = \sum_i w_i x_i$ then the Q learning algorithm converges under the conditions of $\sum_{t=0}^{\infty} \alpha_t = \infty$ and $\sum_{t=0}^{\infty} \alpha_t^2 < \infty$. However, under these conditions a non-linear gradient Q learning algorithm may diverge. This is because we haven't fixed $\boldsymbol{w}$ for the target. We are effectively changing the target and the estimate.

6.1.2 Mitigating Divergence

To mitigate the divergence issue discussed in the gradient Q learning algorithm we can use 2 tricks.

1. Experience Replay
2. Use 2 networks:
    - Q-Network
    - Target Network

Experience Replay: The idea is to store previous experiences (s, a, s', r) into a buffer and sample a mini-batch of previous experiences at each step to learn by Q-learning. The intuition behind this is that if we introduce some errors elsewhere, a simple way of correcting this is to remember all of our previous tuples and fixing the errors by replaying this experience and performing further updates.

The advantages of this are:

- Break correlations between successive updates (more stable learning)
- Fewer interactions with environment needed to converge (greater data efficiency)

Target Network: The idea is the use a separate target network that is updated only periodically. We are going to update our estimate network while keeping the target network fixed and then once in a while we are going to update the target network in the direction of the estimation network.

Repeat for each (s, a, s', r) in mini-batch (similar to value iteration):

$$\begin{aligned} &\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha_t \left[Q_{\boldsymbol{w}}(s,a) - r - \gamma \max_{a'} Q_{\bar{w}}\left(s', a'\right) \right] \frac{\partial Q_{\boldsymbol{w}}(s,a)}{\partial \boldsymbol{w}} \\ &\bar{\boldsymbol{w}} \leftarrow \boldsymbol{w} \end{aligned}$$

6.2 Deep Q-Network Algorithm

This algorithm is similar to the Gradient Q Learning Algo but no incorporates Experience Relay and Target Networks to mitigate the chance of divergence.

Algorithm 9: Gradient QLearning(s)

```
Initialize weights $\boldsymbol{w}$ uniformly at random in $[-1, 1]$ Observe current state $s$. do
    Select and execute action $a$ ;
    Receive immediate reward $r$ ;
    Observer new state $s'$ ;
    Add $(s, a, s', r)$ to experience buffer ;
    Sample mini-batch to experiences from buffer ;
    For each experience $(\hat{s}, \hat{a}, \hat{s}', \hat{r})$ ;
        Gradient: $\frac{\partial Err}{\partial \boldsymbol{w}} = [Q_{\boldsymbol{w}}(\hat{s}, \hat{a}) - \hat{r} - \gamma \max_{\hat{a}'} Q_{\bar{\boldsymbol{w}}}(\hat{s}', \hat{a}')] \frac{\partial Q_{\boldsymbol{w}}(\hat{s}, \hat{a})}{\partial \boldsymbol{w}}$ ;
        Update weights: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \frac{\partial Err}{\partial \boldsymbol{w}}$ ;
    Update state: $s \leftarrow s'$ Every $c$ steps, update target: $\bar{\boldsymbol{w}} \leftarrow \boldsymbol{w}$
while Until number of epochs;
```