NUI Galway
OÉ Gaillimh

Arts with Data Science Final Year Project

# A Practical Introduction to Reinforcement Learning

Based on Sutton and Barto's" Reinforcement Learning: An introduction"
& David Silver's "Introduction to Reinforcement Learning"

Labhaoise Barrett
18321786

Word count (excluding code): 8544
Page count (total): 42

Table of Contents

## Abstract

This project sets out a practical guide to reinforcement learning supported by a Python coding of a blackjack like game.
The intended audience is an undergraduate student comfortable with Python and mathematics but without prior knowledge of reinforcement learning.

This guide is built on the Sutton and Barto book, "Reinforcement Learning: An introduction" as well as David Silver's "Introduction to Reinforcement Learning" course

Throughout this guide examples will focus on gridworld problems and the blackjack game 'Easy21'.
Coding examples will be applied to the Easy21 game.

## 1. Introduction to Reinforcement Learning

Reinforcement learning is a growing space in machine learning. RL involves teaching an agent to choose actions that maximise its rewards in its environment.

A classic introductory idea is teaching a puppy a new trick. The puppy does not understand the higher purpose of putting his paw in your hand but he does understand that that action gets him a treat when his trainer says 'paw' but doing another learnt action such as sitting does not (in this case). The puppy is rewarded when he takes the 'correct' action and so overtime it is reinforced in the puppy's mind that the action of giving his paw will result in a reward and he will learn to follow that pattern when presented with the 'paw' command.

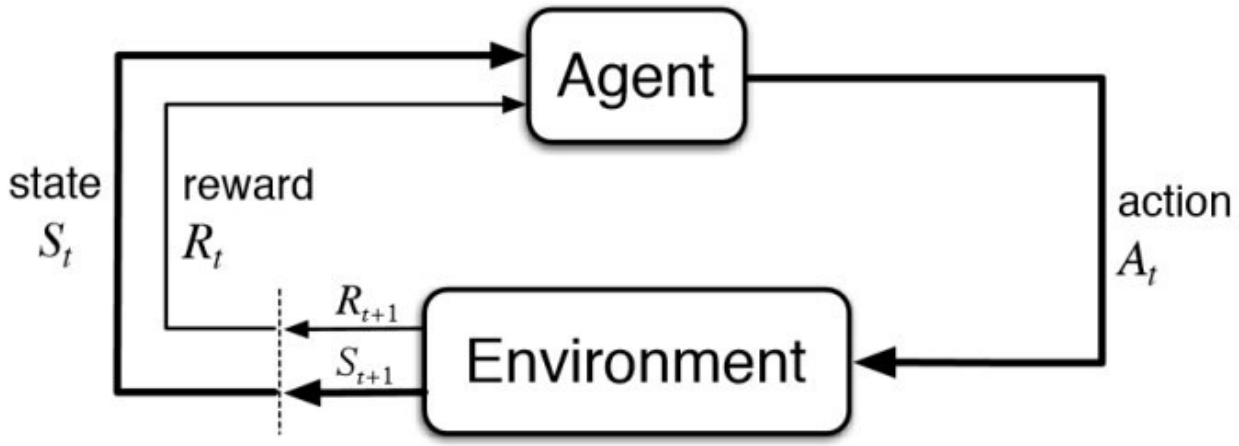
How does the idea of reinforcement learning fit in with the rest of machine learning? It seems clear that reinforcement learning does not fit in exactly with supervised learning - the puppy (or the agent) does not know beforehand what the right answer is, he can only learn by observing the results of actions.
So perhaps it is closer to unsupervised learning? Unsupervised learning searches for patterns in data but does not take action or make changes to those patterns. But reinforcement learning is all about taking actions and learning from those actions. It is a much more dynamic and exploratory process than unsupervised (or supervised) learning.

The history of reinforcement learning is an interesting one. The concept of RL was solidified in the 1980s but before that there were two separate threads of ideas. The first was about learning through trial and error and studying how animals learn. The second was more mathematical and centred round optimal control and how it find

optimal solutions. There wasn't much learning in this second thread so it hadn't quite reached reinforcement learning.

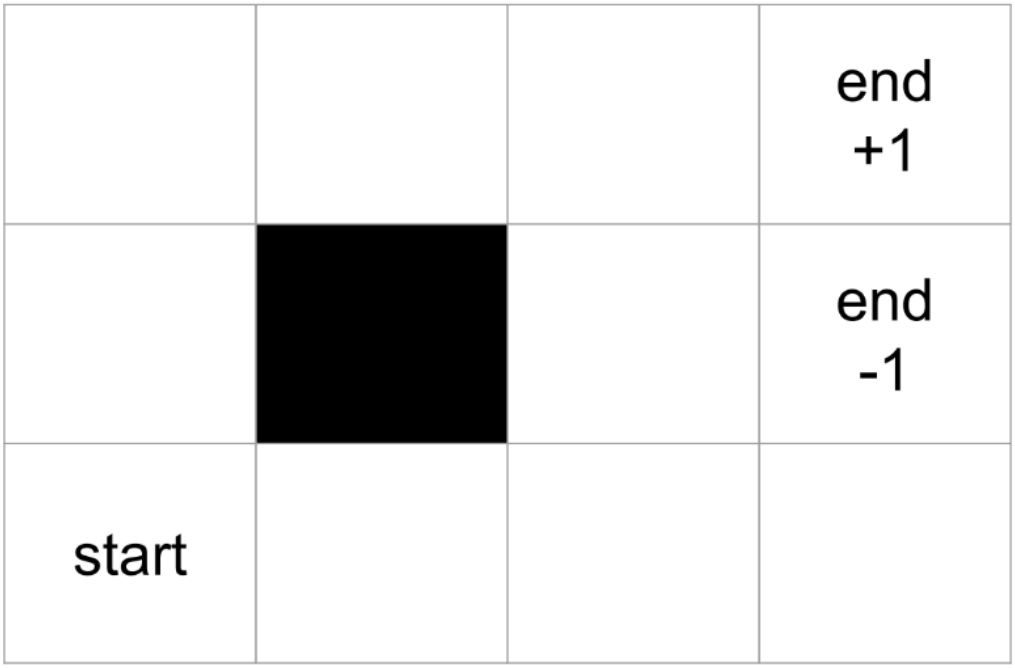
In the broadest terms, reinforcement learning involves an agent, an environment and a reward. The agent interacts with the environment with the hopes of receiving a reward. It learns from its interactions, reinforcing the best way to receive a reward.

There are a few very important building blocks to reinforcement learning:
Agent: an agent performs action in an environment to receive rewards, i.e. the puppy
Environment: the 'world' in which the reinforcement learning problem takes place.
Set of states: this a list of all the possible states of the environment. The set of states is represented as S and a given state is s.
Set of actions: list of all possible actions that the agent can undertake. The set of actions is represented as A and a given action is a.
Reward: the reward is what the agent receives after undertaking an action.
Episode: episodes are a sequence of states and actions at time t where t can be an infinite number of time steps.




*Source: Towards Data Science*

One of the examples we will refer to in this guide is Gridworld. Gridworld is an excellent visual way to understand reinforcement learning techniques. Here is a Gridworld:




*Source: Towards Data Science*

The grid presented above is the environment with which our agent will interact. There are two end points, one with a positive reward and one with a negative. Also present is a black box which represents a space that the agent cannot walk through. To begin with we will assume that the agent's actions are deterministic, meaning the agent will go where it intends to (later we will look at a version of gridworld where every so often the agent is pushed off course by external forces). The agent begins in the start box in the bottom lefthand corner. It initially knows nothing about the environment and so has no idea whether it would be better to move up or to the right. Let's imagine we tell our agent to simply choose randomly for its first travel through gridworld.

With a 50-50 chance the agent chooses up or right, let's say right. Now the agent has three choices up, left or right. It will choose each of these with a 1 in 3 chance. And so this travel continues until the agent arrives at one of the end points. It's important to see that it arrives at this end point completely randomly, it does not yet understand anything about his environment.

Let's posit that in this first path ends with the agent receiving a positive reward. Now the agent knows something about its environment, that this sequence of decisions has

resulted in a reward. Next time the agent starts in its start position, it will remember that moving to the right eventually resulted in him getting a reward. Is the correct thing for the agent to always choose this right step from this first box? Perhaps it is, perhaps not. The agent knows nothing about the rest of its environment yet. The agent should use its learning of course but it should combine it with trying some unknown paths in case there's a larger reward along that way.

## Easy21 Code Part 1

We can build the environment of the game using just what we have learnt in this first section.

First we need some imports.

```
1 import numpy as np
2 import pandas as pd
3 import random
4 import matplotlib.pyplot as plt
5 from matplotlib import cm
6 from matplotlib.ticker import LinearLocator, FormatStrFormatter
7 from mpl_toolkits.mplot3d import Axes3D
8 %matplotlib inline
```

Instructions for the game from David Silver's assignment:

- The game is played with an infinite deck of cards (i.e. cards are sampled with replacement)
- Each draw from the deck results in a value between 1 and 10 (uniformly distributed) with a colour of red (probability 1/3) or black (probability 2/3).
- There are no aces or picture (face) cards in this game
- At the start of the game both the player and the dealer draw one black card (fully observed)
- Each turn the player may either stick or hit
- If the player hits then she draws another card from the deck • If the player sticks she receives no further cards
- The values of the player's cards are added (black cards) or subtracted (red cards)
- If the player's sum exceeds 21, or becomes less than 1, then she "goes bust" and loses the game (reward -1)
- If the player sticks then the dealer starts taking turns. The dealer always sticks on any sum of 17 or greater, and hits otherwise. If the dealer goes bust, then the player wins; otherwise, the outcome – win (reward +1), lose (reward -1), or draw (reward 0) – is the player with the largest sum.

Let's understand the basic elements of this game. We are playing from the player's perspective, meaning that if the player wins we get a positive reward and if the dealer wins we get a negative reward.

Step 1 to building the game is to build a function that will simulate the player drawing a card.

```
def pull_card(score):
    addition_to_score = 0
    # first what number will the card be?
    card_number = random.randint(1,10)
    #secondly what colour will the card be?
    card_colour = np.random.choice(a = ['red', 'black'], p = [1/3,
2/3])
    if card_colour == 'black':
        addition_to_score += card_number
    else:
        addition_to_score -= card_number
    return score + addition_to_score
```

The `pull_card` function implements the rules of the game we saw above, once the game has begun. After the first cards are drawn, if a player or the dealer decides to 'hit' then this function is called. They have a 2 out of 3 probability of getting a black card which would mean that the number of the card is added from their score. Otherwise they will pull a red card meaning its number will be subtracted from their score. Because the cards are drawn with replacement, there is no change to this probability distribution throughout the game.

Next the step function described in the assignment has to be built.

```
def step(action, dealer_card, player_sum):
    if action == 1:
        player_sum = pull_card(player_sum)
        if player_sum < 1 or player_sum > 21:
            return dealer_card, player_sum,'terminal', -1
        else:
            # there is no reward yet since dealer has not played
            return dealer_card, player_sum, 'not terminal', 0
    # else condition if action == stick -> dealer plays -> reward
    else:
        while dealer_card < 17:
            dealer_card = pull_card(dealer_card)
            if dealer_card < 1 or dealer_card > 21:
                return dealer_card, player_sum, 'terminal', 1
        if dealer_card > player_sum:
            return dealer_card, player_sum, 'terminal', -1
        elif player_sum > dealer_card:
            return dealer_card, player_sum, 'terminal', 1
        else:
            return dealer_card, player_sum,'terminal', 0

```

The goal is to move from the policy shown below, where there is a random probability of hit or stick being chosen, to a dataframe with definitive hit or stick policy to maximise reward in each state.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 2 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 3 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 4 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 5 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 6 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 7 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 8 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 9 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 10 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 11 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 12 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 13 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 14 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 15 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 16 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 17 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 18 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 19 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 20 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |
| 21 | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick | hit/stick |

The first step is to check whether the player has chosen to hit or stick. If the player chooses to hit (i.e. `action == 1`) then we define the player's new sum by pulling a card.
An important part of the game that happens in this step function is checking whether the player or the dealer has gone bust from their latest card draw. If the player or dealer's sum is less than 1 or greater than 21, they have lost. If the player loses then the reward returned is -1, if the dealer loses, the player's reward is +1.

If the player does not go bust the current state, the player sum and the dealer card is returned but no reward because the game has not finished yet. The step function is then called again to continue the game.

When the player decides to stick, then it is the dealer's turn. The dealer's turn is fully played within this step function. The dealer's policy is pre-defined. He will always hit if his sum is below 17 and will always stick above that. If the dealer goes bust, then the player has won the game and the reward +1 is returned. If the dealer reaches 17 or higher, he sticks and then the two scores are compared. If the player has a higher sum then the player has won and receives +1 reward, if the player has a lower sum then the dealer has won, and the player receives -1 reward and if the sums are equal, it's a draw and no reward is given.

## 2: Markov Decision Processes

Markov decision processes (MDPs) are reinforcement learning problems that satisfy the Markov property. A Markov decision process is a way to represent reinforcement learning problems mathematically.

### 2.1 Markov Property and Process

In a reinforcement learning problem when the agent is in a state, this state should summarise all the relevant information about the states that came before. A state that does this is called a Markov state and is said to have the Markov property. This is also referred to as the 'independence of path' property because the current state is not dependent on the path it took to get there.

Mathematically a state is Markov if and only if $P[S_{t+1}|S_t] = P[S_{t+1}|S_1, \ldots, S_t]$.

In other words, the probability of the agent choosing to move to state, $S_{t+1}$ given that the agent has information from only the current state, $S_t$, is the same as if the agent knows everything about its past states and current state. The state is a sufficient statistic of the future.

> A Markov process is a memoryless random process: a sequence of random states $S_1, S_2 \ldots$with the Markov property

The states found in the Gridworld and Easy21 examples are Markov states and thus the sequences are Markov processes:
Gridworld: once the agent is in a square it does not need to know how it got there to decide where it goes next, it doesn't matter where the agent started only that this square is where it is now. It uses only the information it knows about this square to decide how to move towards the reward.

Easy21: when a player is playing blackjack they do not need to know their previous card totals, they only need to know the total they hold now. Since the cards are sampled with replacement, the current state is all the information they need to decide whether to hit or stick.

### 2.2 Markov Reward Process

A Markov reward process is a Markov process that has rewards attached to it. Mathematically, this reward is a function: $R_S = E[R_{t+1}|S_t = s]$, i.e. the reward associated with a state is the expected value of the reward an agent will receive by performing an action in state $S$. The reward will often be discounted by a factor of $\gamma$

(gamma). This is a value between 0 and 1 and it tells the agent how much to value

The return $G_t$ is the total discounted reward from time-step $t$.

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

current rewards against future rewards.

When $\gamma$ is close to 0, the agent will care only about immediate rewards.
When $\gamma$ is close to 1, the agent will value future rewards more.

Discounting is an important and useful concept in reinforcement learning, there are several reasons for this including:

- in the real world, immediate rewards are often preferred to future rewards
- In the case of continuous (never-ending) reinforcement learning tasks, discounting future rewards prevents the returns from extending to infinity.

Later we will see that the Easy21 game does not use a discounting factor. This is because all of the sequences in the Easy21 game are guaranteed to terminate and we don't need to worry about infinite returns.

### 2.3 Markov Decision Processes

Markov decision processes are reinforcement learning problems that satisfy the Markov property.

If a reinforcement learning problem can be defined as a MDP then it is ready to be solved by an agent taking actions and receiving rewards. Markov Decision Process are different from a Markov chain because it is not just the current state that factors into the agent's decision making process but the action as well. The agent does not want to randomly choose an action with every step just based on its current state, instead it wants to learn which actions in its current state will lead to the most rewards.

The gridworld agent does not want to keep picking up, down, left, right with a 0.25 probability each if he knows that last time he went up he got a reward. Instead he is more likely to choose the up path. So he factors in information about the current state and his possible actions.

Defining a finite MDP (i.e. the number of possible states and actions are finite):
Given any state s and action a, the probability of each possible pair of next state, s' and reward, r is given as:

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\}$$

This equation actually provides everything needs to represent a MDP. We can see from this equation that the probability of the next state and the reward is dependent both on the current state and the action taken.

### 2.4 Policies

Now that we know how an agent makes decisions in a MDP and how value is assigned to these decisions, we would like to ensure that the agent is making the best choices and maximising its reward. We do this by finding its optimal policy.

A policy $\pi$ is a distribution over actions given states,

$$\pi(a|s) = P[A_t = a]|S_t = s]$$

A policy is essentially an instruction manual for an agent. It tells the agent exactly how to behave in every state.For reinforcement learning a policy must have the Markov property - the policy relies on the current state to tell the agent what the next step is.

A policy does not have to be optimal it is just an arbitrary path through the environment. For a policy to be optimal the agent needs to be able to assign values to states and actions.

The Bellman Optimality Equation is used to estimate the optimal value of each state, $v_*(s)$.

With some rearranging our bellman equation $v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$

can be represented as $\sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_\pi(s')]$.

Let's have a look at this equation:

1. $\pi(a \mid s)$ : this is the policy, the probability of the agent performing action $a$ in state $s$.
2. $p(s', r \mid s, a)$ : this is the probability of an agent ending up in a certain state-reward pair from the current state-action.
3. $[r + \gamma v_\pi(s')]$ : this last part is the sum of the current reward plus the discounted future rewards.

### 2.5 Value functions

Value functions measure how good it is for an agent to be in a given state, or to perform a given action in that state. "How good" is a little vague so to be precise, a state's "goodness" is measured through the rewards an agent expects to receive, i.e. the expected returns. In the same way goodness of a state-action pair is a measure of the future expected rewards associated with performing a given action in a given state. To establish a value function, the agent needs a policy to follow.

The value of a state $S$ within a policy $\pi$ is represented as $v_\pi(s)$.

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right]$$

*Sutton & Barto Equation 3.10*

In English this is essentially the returns that an agent expects when it starts in state $S$ and thereafter follows the policy $\pi$.

In general, $v_\pi$ is known as the state-value function of the policy $\pi$.

We can also define the value of taking an action $a$ in a state $S$, when following $\pi$. This is called $q_\pi(s, a)$.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]$$

In English, this the returns that an agent expects when he takes action $a$ from state $S$ and thereafter follows the policy $\pi$.

# 3. Dynamic Programming

Solving reinforcement learning problems by finding a solution to the Bellman optimality equation is not possible with linear algebra. This is because it is a recursive equation that needs future values. The solution to solve the equation is to use iterative methods, such as dynamic programming.

Dynamic programming is an important theoretical step in understanding reinforcement learning. Dynamic programming describes a group of algorithms that are used to find optimal policies given a perfect Markov Decision Process environment.
This assumption of a perfect environment makes dynamic programming less practical but it is a starting point for the rest of the algorithms we will examine in this guide. Dynamic programming also takes a lot of computational power.

Dynamic programming is a way to solve complicated problems by breaking them down into subproblems. Dynamic programming problems have two properties:
1. The optimal solution can be built from the solutions to subproblems
2. The subproblems must overlap

A MDP will fulfil these properties through the Bellman equation. For dynamic programming to work it needs full knowledge of the MDP, meaning it needs to know the transition probability matrix and how rewards are given. Dynamic programming can evaluate an MDP by finding the value function and it can control a problem to find the optimal value function and its corresponding policy.

Evaluating a policy $\pi$ can be done with an iterative application of the Bellman expectation backup. It is known that the value functions found through this method will eventually converge to the value function associated with policy $\pi$.

## 3.1 Policy improvement

We can value a policy using a value function. The next step is to be able to improve this policy.

We begin with a random value at each state and an arbitrary policy $v_\pi(s)$ to follow (the agent must have instructions to follow, even if they are not optimal, since we wish to compute the state-value function). An example of this initial policy would be to set each action an equal probability of occurring given any state. So in a gridworld, the agent can move up, down, left, right, each with a probability of 0.25.
If the agent is in state s, should it take the action indicated by the established policy or try a new deterministically chosen action? One way to do this is for the agent to

check what would happen if they chose a different action from this state and thereafter followed the established policy.

Mathematically this behaviour is:

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a]$$
$$= \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_\pi(s')]$$

If this new action results in a value greater than $v_\pi(s)$ then it can be concluded that the agent should follow this new action every time it comes to this state s.

### 3.2 Policy iteration

Now we know how to find a value function and then find an improved value function. But we can't stop there. Instead, the new and improved value function becomes our baseline and we iteratively improve policies step by step.

We begin with a random policy $\pi$ and we improve it using the value function $v_\pi$ to find a better policy $\pi'$. We can improve this policy again by computing $v_{\pi'}$. This results in an again improved policy, $\pi''$. This can be repeated to create a sequence of monotonically improving policies and their corresponding value functions.

> A monotonic sequence is one where the values strictly increase or strictly decrease.

We know that each policy will be a strict improvement over the one before until the optimal policy is reached (an optimal policy is guaranteed to be reached in a finite MDP).

### 3.3 Value iteration

Each step of policy iteration requires policy evaluation which can take some time and computational expense. This process can be shortened without losing the assumption of optimality by introducing a stopping condition. One way to do this is to just stop after k iterations of the policy evaluation. If we were to stop at k=1 this is called value iteration. Value iteration doesn't have a defined policy, instead to find the optimal policy the Bellman optimality backup process is applied iteratively.

To apply value iteration a very slightly modified Bellman Optimality Equation is used.

$$v_{k+1}(s) \doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a]$$
$$= \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_k(s')]$$

*Sutton and Barto equation 4.10*

The difference in the equation is the max part. This tells the algorithm to directly choose the action that will maximise its reward. The algorithm still uses the previous value function at step $k$ to find the states of the value function of $k + 1$.

Formally, value iteration needs an infinite amount of iterations to converge to the exact $v_*$ but practically the iteration can be stopped once each sweep is only changing the value function a tiny amount. In each of the sweeps that value iteration performs it effectively performs one sweep of policy evaluation and one policy improvement.

To understand dynamic programming in a simple way, imagine a a 3 x 3 gridworld. The bottom right state is where the reward is achieved, let's call this the end state. The gridworld problem can be divided into subproblems and to start we find the states closest to our end state. Then the best action can be chosen here: if the agent is in grid space (3,2) then the best action (with the highest reward) is 'down'. Then the next subproblem is to find the best actions from states that point towards the states of the first iteration. When this is done, the solutions to these subproblems are combined to find the overall optimal solution. The fact that this is possible is called the Principle of Optimality.

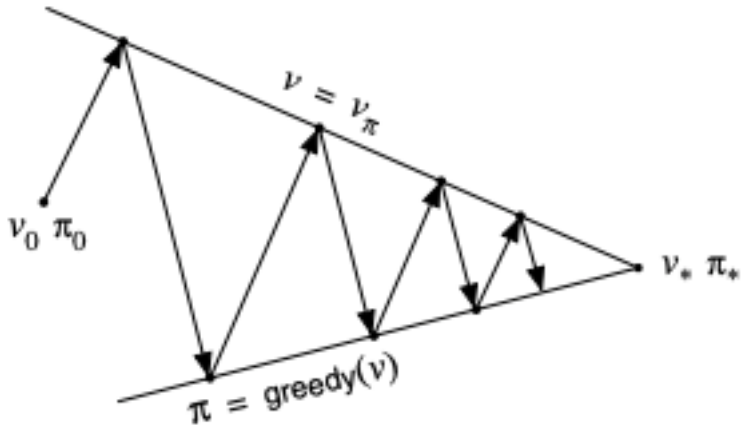
### 3.4 Asynchronous Dynamic Programming

Asynchronous dynamic programming back up states individually without a specific order, instead of sweeping the entire state set. If all states are visited then the optimality assumption will still hold and this process might use less computational power. These algorithms can choose the values of any state to back up and they use whatever values of other states they currently know. The assumption of convergence will still hold so long as the asynchronous algorithm continues to backup the values of all the states through the whole process.

A dynamic programming method could be applied to Easy21 because the environment is entirely known but it would not be easy since all of the transition probabilities and rewards would have to be computed beforehand.

### 3.5 Generalised Policy Iteration

Generalised policy iteration (GPI) refers to the idea of policy evaluation and policy improvement processes interacting with each other. The concept applies to most reinforcement learning methods. In the broadest of terms each method has policies and value functions, and the policy is improved with respect to the value function and the policy is driven toward the value function for the policy. Once both the evaluation and the improvement processes are no longer making changes then the policy and value function are optimal. We can represent the evaluation and improvement processes working in tandem to find the optimal value function and the optimal policy with a simple diagram.



## 4. Monte Carlo Methods

As established above, dynamic programming needs a perfect environment and lots of computational power to work. The goal of looking at Monte Carlo (MC) methods (and later Temporal Difference learning) is to achieve the effect of dynamic programming without these requirements.

Monte Carlo methods don't need to know everything about their environment to start. Instead these methods learn from experience. The agent interacts with the environment, learns from its interaction and uses this information to find an optimal behaviour.

Monte Carlo methods are based on averaging sample returns. MC reinforcement learning problems are episodic, meaning that its sequences terminate. Monte Carlo methods are split into two groups, on-policy and off-policy. On-policy methods directly improve the policy being used to instruct the agents but off-policy methods improve a different policy than the instructions being used by the agent.

Because all action selections are undergoing learning, i.e. returns from a given action-state is dependent on the choices made in the following states, the MC problem is non stationary. The generalised policy iteration concept must be adapted to handle this. This is done by learning value functions from samples instead of computing value functions using complete knowledge of the MDP.

We begin understanding Monte Carlo methods by looking at on-policy Monte Carlo prediction. In MC prediction the state-value function is learnt for a given policy.

> The value of a state is the expected return – expected cumulative future discounted reward – starting from that state

Monte Carlo methods estimate the value of a state from experience. They do this by averaging the returns seen after that state has been visited. The more returns that are collected, the closer the average will be to the true expected value.

Let's use some of the symbols we've become accustomed to. If we want to estimate $v_\pi(s)$ : the value of a state s under policy π, given a set of episodes obtained by following π and passing through s. Each time the state $s$ appears in an episode, this is called a visit. There can be multiple visits to a state in one episode so let's highlight the first visit to $s$. One MC method is called first-visit MC which, as the name

suggests, only uses the returns of the first visit to $S$ to calculate the value. On the other hand, every-visit MC will average the returns of all the visits to $S$. We will continue with a focus on first-visit MC.

Initialize:
$\pi \leftarrow$ policy to be evaluated
$V \leftarrow$ an arbitrary state-value function
$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
Generate an episode using $\pi$
For each state $s$ appearing in the episode:
$G \leftarrow$ return following the first occurrence of $s$
Append $G$ to $Returns(s)$
$V(s) \leftarrow$ average($Returns(s)$)

Figure 5.1: The first-visit MC method for estimating $v_\pi$. Note that we use a capital letter $V$ for the approximate value function because, after initialization, it soon becomes a random variable.

First-visit MC (and indeed every-visit MC) is known to converge to the true state-value function as the number of first (or all visits) to state $S$ approaches infinity.

Estimating action values:
Action values refer the values of state-action pairs. Estimating these are useful when a model is not available. Without a model, we need to know the values of each possible action to decide what the optimal policy (based on current information) should be.

The goal is to estimate $q_\pi(s, a)$.

$q_\pi(s, a)$ represents the expected return when starting in state s, taking action a, and thereafter following policy π.

This problem is the same as just using the state values except that visits are now to state-action pairs instead of a state. A state-action pair has been visited once the state has been arrived at and the action of interest has been chosen from it. This method will also converge to the true expected values as the number of visits approach infinity.

One problem that could be encountered with these Monte Carlo methods is that once an agent is following a policy, only the state-action pairs associated with the chosen actions are visited and there is no information about the other possible choices. This problem can be solved by "maintaining exploration" so that information about all possible actions in a state is collected. This can be done through exploring starts which means that episodes will start at a randomly selected state-action pair (with every state-action pair having a non-zero chance of being chosen). This means that

through an infinite number of episodes every single state-action pair will be visited an infinite number of times.

## 5. Monte Carlo Control

The term 'control' in reinforcement learning means finding optimal policies. Monte Carlo control involves using Monte Carlo episodes to find an optimal policy.

The concept is essentially the same as the generalised policy iteration we looked at in the dynamic programming section. We alternate between policy evaluation and policy improvement to bring the policy and value function closer to optimal.

The simplest method of policy improvement is a greedy method with respect to the current value function. For Monte Carlo control we have an action-value function which means we can find the best action without needing a model. Defining a greedy policy for any action value function, $q$, for each $s \in S$:

$$\pi(s) = \underset{a}{argmax} q(s, a)$$

Put simply, in each state the agent will deterministically choose the action with the maximum action-value.

One of the assumptions we have made so far is that there will be an infinite number of episodes in a Monte Carlo method. This of course is not feasible in practice but it is also not a very difficult solve! An alternative approach is to mix policy evaluation and policy improvement: with each step the value function is slightly adjusted towards the optimal. In Monte Carlo it makes sense to switch between evaluation and improvement on an episode-by-episode basis. When an episode is complete, policy evaluation is performed and it is used to update all the steps in that episode. An example of this implementation is Monte Carlo with Exploring Starts. This algorithm will converge to the optimal policy and value function.

## Easy21 Code Part 2

Sutton and Barto pseudocode: Monte Carlo control with exploring starts:

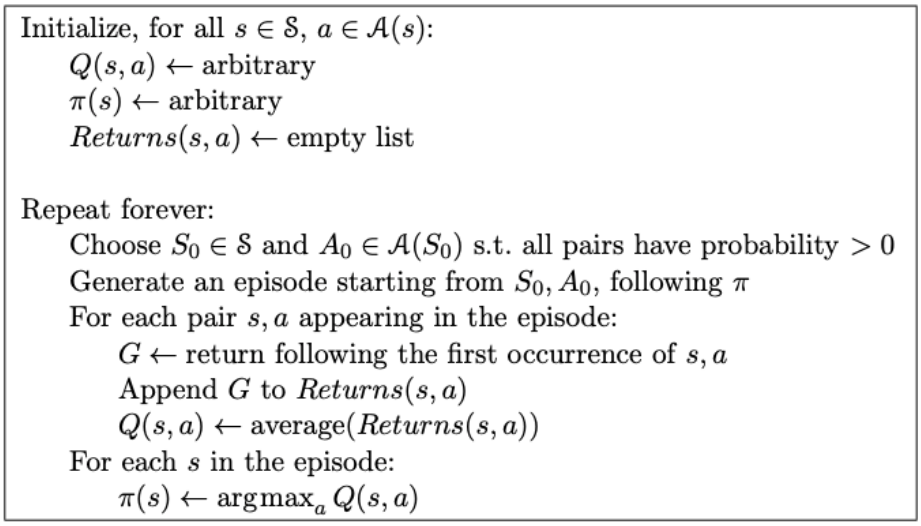



Figure 5.4: Monte Carlo ES: A Monte Carlo control algorithm assuming exploring starts and that episodes always terminate for all policies.

Exploring starts are suitable for Easy21 because the episodes are all simulated games. This means every state can be chosen just by choosing the starting state randomly.

Building Blocks:
Value function: $Q(s,a)$. This represents what the agent knows about how good an action, $a$, is when at state $s$. This will be initialised to zero.
Time varying scalar step size - the step size is also called the learning rate. It tells the agent how much to value new information. It is a weight given to the reward the agent would get by taking action a in state s.
N(s) - number of times that state s has been visited.
N(s, a) number of times that action a has been selected from state s.
$\epsilon$-greedy policy: this defines how the agent will play the game. This greedy policy allows for both exploration and exploitation, weighted by the number of times state s has been visited.

Let's start with the value function. For the value function we need to represent every possible situation that the agent could be in (excluding going bust), taking into account the dealer cards, player sum and the action.
There are 2 possible actions.
There are 10 possible dealer cards.
There are 21 possible player sums.

We need a 2 x 10 x 21 matrix of zeroes. To apply this practically in Python we will create a 2 x 11 x 22 matrix. Why? Because of Python's indexing rules. If we were to try to update the case where the dealer has the 10 card, that would be index 9 in a 2 x 10 x 21 matrix. This isn't very useful because every time we come across a card index we would have to subtract one. This clutters the code unnecessarily. By creating a 2 x 11 x 22 matrix the code is a little easier to read: when the dealer has card 10, we can just index using 10. Note that this results in a row and column of zeroes that won't be updated but since this is such a small space it won't cause any trouble.
The action space doesn't not need to be expanded because the actions 'hit 'and 'stick ' can be represented intuitively and cleanly as 0 and 1.

```
# intitalise value function to 0
Qsa = np.zeros((2,11,22))
# number of times a state has been visited - it doesn't matter what
action has been chosen from that state
Ns = np.zeros((11,22))
# number of times action a has been chosen from state s
Nsa = np.zeros((2,11,22))
```

We should build a function that describes how the agent will choose its action from a state s. Because we want to maintain a balance between exploration and exploitation an $\epsilon$-greedy exploration strategy is used:

$$\epsilon_t = \frac{N_0}{(N_0 + N(s_t))}$$

$N_0$ is a constant, specified in the assignment to equal 100.

```
def exploration_greedy(dealer_card, player_sum):
    # epsilon = 100/(100 + number of times state [player_sum,
dealer_card] has been visited)
    e = 100/(100 + Ns[dealer_card, player_sum])
    if np.random.random() < e:
        # choose randomly next action
        action = np.random.choice([0,1])
    else:
        # choose greedily next action by looking up the best value in the
value function out of the two possible actions
        action = np.argmax(Qsa[action, dealer_card, player_sum] for
action in [0,1])
    return action
```

**The next step is to build the Monte Carlo control function.**

```
def monte_carlo_control(Qsa, Ns, Nsa):
    total_reward = 0
    terminal = 'not terminal'
    visited_states = []
    #initialise by drawing first cards - this is why the exploring starts
are assumed to visit all states in this game
    player_sum = random.randint(1,10)
    dealer_card = random.randint(1,10)
    while terminal != 'terminal':
        # increase the number of visits to this state by 1
        Ns[dealer_card, player_sum] += 1
        # choose action using defined policy
        action = exploration_greedy(dealer_card, player_sum)
        # increase the number of times this action has been chosen in this
state
        Nsa[action, dealer_card, player_sum] += 1
        # record the states visited
        visited_states.append([action, dealer_card, player_sum])
        # now the information has been collected from episode starting
state it's time to play the game
        dealer_card, player_sum, terminal, reward = list(step(action,
dealer_card, player_sum))
        # add the reward to the total reward
        total_reward += reward
    for action, dealer_card, player_sum in visited_states:
        # could write a function for this
        # the value function is updated with the reward associated with
the states the agent has visited
        # the time varying scalar step weights the reward
        Qsa[action, dealer_card, player_sum] += (1 / Ns[dealer_card,
player_sum])*\
        (total_reward - Qsa[action, dealer_card, player_sum])
    # return the updated value function, number of times a state has been
visited,
    # number of times an action has been chosen in a state
    return Qsa, Ns, Nsa
```

Let's walk through this function step by step.

1. First we initialise the total reward. This will be the total of the reward received in each episode.
2. Then we set our starting terminal condition as not terminal. This is how the game will be stopped, when the terminal condition is changed, the loop ends.
3. Then we create an empty list visited_states. This hold the list of states that have been visited in this episode.
4. Choosing the first cards is a fully observed random process. This is the beginning of the Monte Carlo control. The fact that the start of the game is fully observed means that the player also knows what card the dealer has started with.

Enter the while loop:

5. First we have to do some record keeping. We have to note that this state has been visited by updating the N (s) matrix.
6. Then we note that the state-action pair has been visited by updating N( s, a).
7. Then we record that this state has been visited in this episode by adding it to the list of visited states.
8. Now we can actually start playing the game. We can do this just by calling the step function. The step functions returns the updated state, information about whether the game has ended or not and finally the reward that we have received.
9. Then we add the reward to the total reward. If the game isn't finished there is no reward.

Exit the while loop

10. The next step is to update the value function Q(s, a). We can do this by going through the list of visited states and apply the time-varying scalar function to calculate how much of the total reward should be applied to each state that we have visited.
11. The function returns the updated value function, the number of times a state has been visited and the number of times an action has been chosen in a state. This is returned to that the information is passed onto and used by the next episode.

The monte Carlo control function has been built so now we can run it for a number of episodes.

```
for i in range(1000000):
    # run the monte carlo control for a set number of episodes, updating
the value function etc as it goes
    Qsa, Ns, Nsa = monte_carlo_control(Qsa, Ns, Nsa)
```

Now that we have run the episodes and populated the value function we can create an optimal policy. The value function tells us which action is associated with a higher return in any given state. Using this information we can now create a definitive table

with hit or stick recommendations rather than just the random chance table we created above.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | stick | stick | stick | stick | stick | stick | stick | stick | stick | stick |
| 2 | stick | stick | stick | stick | stick | stick | stick | stick | stick | stick |
| 3 | stick | stick | stick | stick | stick | stick | stick | stick | stick | stick |
| 4 | stick | stick | stick | stick | stick | stick | stick | stick | stick | stick |
| 5 | stick | stick | stick | stick | stick | stick | stick | stick | stick | stick |
| 6 | stick | stick | stick | stick | stick | stick | stick | stick | hit | stick |
| 7 | stick | stick | stick | stick | stick | stick | stick | stick | stick | stick |
| 8 | stick | stick | hit | stick | stick | stick | stick | stick | stick | stick |
| 9 | stick | stick | stick | stick | stick | stick | stick | hit | stick | stick |
| 10 | stick | stick | stick | stick | stick | stick | stick | stick | stick | hit |
| 11 | hit | stick | hit | stick | stick | hit | hit | hit | stick | stick |
| 12 | stick | stick | hit | stick | stick | stick | stick | stick | hit | hit |
| 13 | stick | hit | hit | stick | stick | hit | stick | stick | stick | stick |
| 14 | stick | stick | stick | stick | stick | stick | stick | stick | stick | hit |
| 15 | stick | stick | stick | stick | stick | stick | hit | hit | stick | hit |
| 16 | stick | stick | stick | stick | stick | stick | stick | stick | stick | stick |
| 17 | stick | stick | stick | stick | stick | stick | stick | stick | hit | stick |
| 18 | stick | stick | stick | stick | stick | stick | stick | stick | stick | stick |
| 19 | stick | stick | stick | stick | stick | stick | stick | stick | stick | stick |
| 20 | stick | stick | stick | stick | stick | stick | stick | stick | stick | stick |
| 21 | stick | stick | stick | stick | stick | stick | stick | stick | stick | stick |

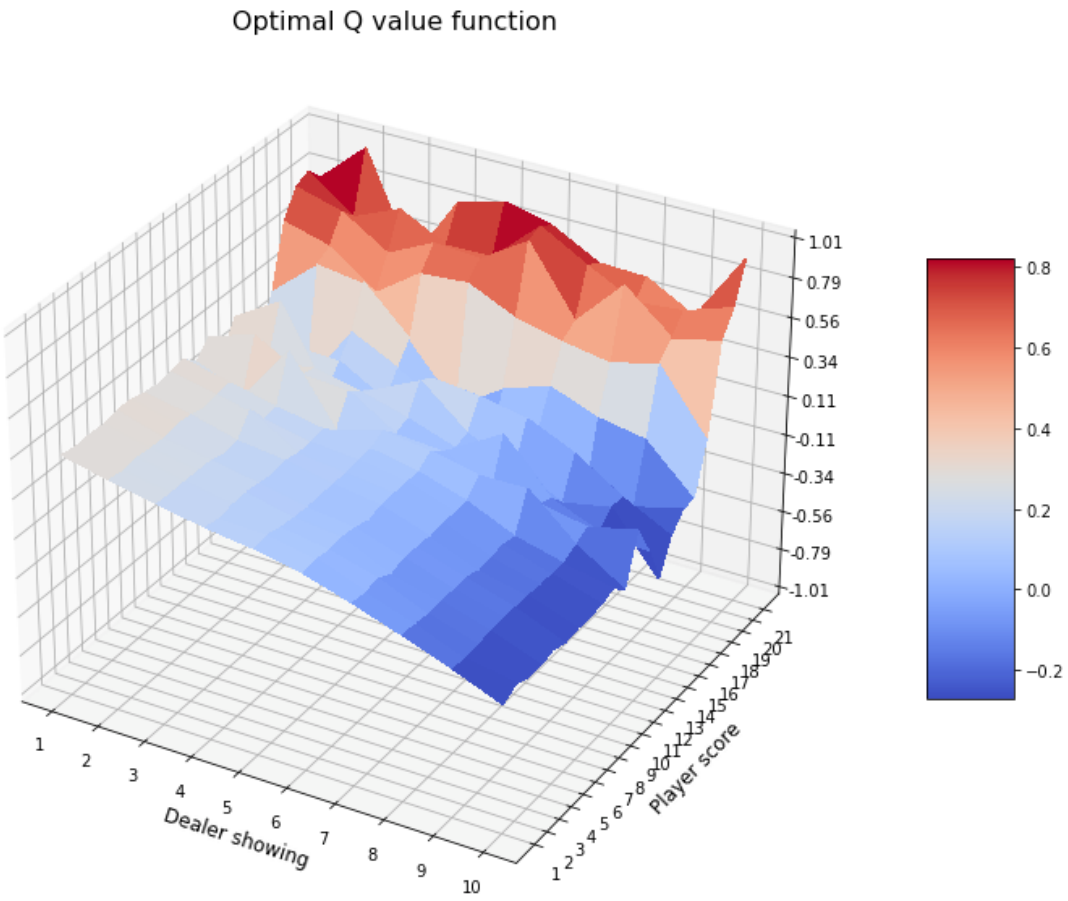
Using this information we can plot the reward associated with following this optimal policy.

```
# numpy array with just the rewards associated with the recommended actions
found above
best_Qsa = np.zeros((11,22))
for dealer_card in range(1,11):
    for player_sum in range(1,22):
        if Qsa[0][dealer_card][player_sum] >
Qsa[1][dealer_card][player_sum]:
            best_Qsa[dealer_card][player_sum] =
Qsa[0][dealer_card][player_sum]
        elif Qsa[0][dealer_card][player_sum] <
Qsa[1][dealer_card][player_sum]:
            best_Qsa[dealer_card][player_sum] =
Qsa[1][dealer_card][player_sum]

#tidy up array
best_Qsa = np.delete(best_Qsa, 0, 0)
value star = np.delete(best Qsa, 0, 1)
```

Now we can plot the value function.
The reference code for this plot is here.

```
fig = plt.figure(figsize=(20,10))
ax = fig.gca(projection='3d')

dealer_showing = np.arange(1, 11)
player_score = np.arange(1, 22)
dealer_showing, player_score = np.meshgrid(dealer_showing, player_score)

surf = ax.plot_surface(dealer_showing, player_score,
np.transpose(value_star), cmap=cm.coolwarm, linewidth=0,
                       antialiased=False, rstride = 1, cstride = 1)
ax.set_zlim(-1.01, 1.01)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))


plt.xlabel('Dealer showing', fontsize=12)
plt.ylabel('Player score', fontsize=12)
plt.title('Optimal Q value function', fontsize=16)

plt.xticks(np.arange(1, 11))
plt.yticks(np.arange(1, 22))

fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```

Here is the optimal value function graphed in 3D.




The more episodes that are run, the smoother the output graph will be.

# 6. Temporal Difference Learning

## 6.1 Introduction to Temporal Difference Learning

Temporal difference learning takes aspects of both dynamic programming and Monte Carlo methods. Sutton and Barto states that TD learning is one of the most "central and novel" ideas in reinforcement learning. TD learning is a model free method like Monte Carlo but unlike Monte Carlo these methods do not need wait for the final outcome to update their estimates, instead they bootstrap (uses estimates to make updates). TD agents will update their knowledge after every action.
It is called temporal difference learning because it makes use of changes (difference) in predictions over time steps (temporal) to learn.

Understanding the concept behind TD learning can be done by thinking about our Easy21 game. Suppose we have just started learning about the environment and our agent doesn't know much yet. It believes that when it has a sum of 20 it should hit. We can see into the future and say that sticking at a sum of 20 is much more likely to lead to a win but the agent does not know this. If we can get the agent to consider the future before it takes an action, to look at its next state and calculate the reward it will get down this path, we essentially get the agent to use some future thinking (that we as humans might see immediately) to make a decision on whether to hit or stick on 20. Without doing exact calculations we can say with confidence that a stick decision here is more likely to lead to a positive reward than a hit. This is essentially temporal difference learning, getting our agent to make decisions without waiting for the game to finish by looking into the future.

Advantages of TD learning over Monte Carlo:
Temporal difference learning has several advantages of Monte Carlo methods:

- TD learns online with every step but MC has to wait for the episode to finish
- TD learning can learn from incomplete sequences but MC cannot
- TD learning works in non-terminating environments but MC needs episodic (terminating) environments

It is a logical next step to replace Monte Carlo with TD in the control loop we have built. This is done in short by applying a TD learning algorithm to the value function, $Q(S, A)$, and using an $\epsilon$-greedy policy to update and improve the value function with each step.

Is the bootstrapping done by TD a good thing? To quote Sutton and Barto TD learning methods "learn a guess from a guess" which at initial consideration might seem unreliable. In truth, the optimality convergence assumption still holds.

## 6.2 Temporal Difference Learning Prediction

Let's begin by understanding how TD learning makes predictions. It is similar to Monte Carlo in that a TD learning agent will also use experience to solve a prediction problem. Both methods will use their experience of following a policy $\pi$ to update their estimate of the value function for that policy, $v_\pi$ for the states (that aren't terminal) that they have experienced. They differ because Monte Carlo methods wait until the end of an episode to update the value function but TD methods only need to wait until the next time step. At this next time step they can create a target and update it using the observed reward, $R_{t+1}$ and an estimate of $V(S_{t+1})$.

The first TD method we will look at is called TD(0). Mathematically this is:
$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$

*Sutton & Barto Equation 6.2*

How does this differ from Monte Carlo? It is helpful to look at what the equivalent equation would be in a Monte Carlo method. The method constant-$\alpha$ MC is a Monte Carlo method with a constant step-size parameter:

$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$

*Sutton & Barto Equation 6.1*

$G_t$ is the actual return following time t.

The difference between the constant-$\alpha$ MC and TD(0) is that the target for the Monte Carlo method is $G_t$ but for TD(0) the target is $R_{t+1} + \gamma V(S_{t+1})$.

Monte Carlo and TD updates are sample backups because they use a sample 'next' state/state-action pair to calculate the value and then updating the original state. DP methods use full backups because they look at a complete distribution of the possible next states.

There has been no formal mathematical proof of which method, Monte Carlo or TD learning, will converge faster but in practice it is seen that TD learning often converges more quickly.

Now that we understand how TD learning does prediction, we can move on to control.

## 7. Sarsa

Sarsa is an on policy TD control method. This method begins with learning an action-value function (as opposed to a state-value function). For this on policy method we need to estimate $q_\pi(s, a)$. This learning process is done as the same way as the TD learning we looked at above.

In the TD learning section above we looked at moving from state to state. With Sarsa we are interested moving from state-action pair to state-action pair. The process is formally the same, a Markov chain with a reward.

Here is the algorithm for calculating the action value function:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

*Sutton & Barto Equation 6.5*

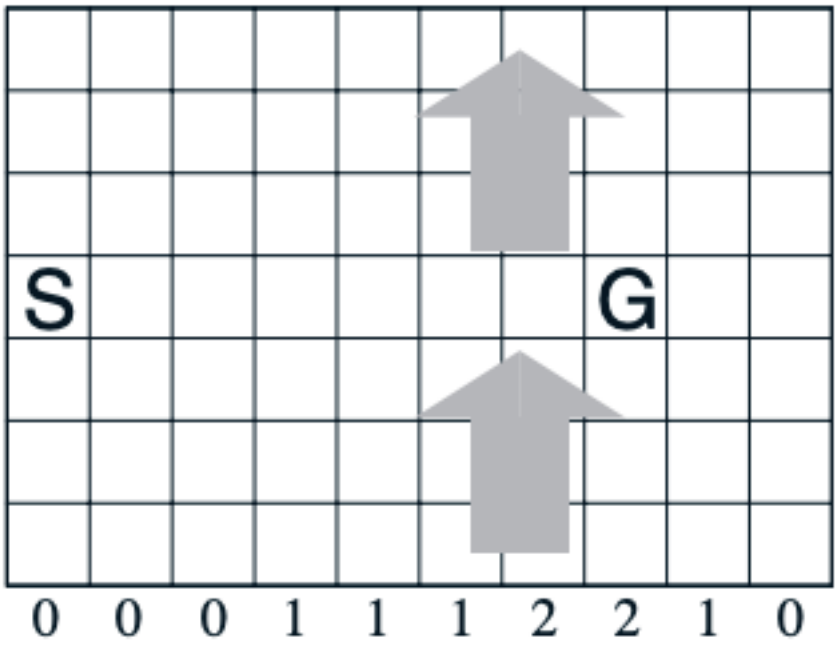
The update defined by this equation is completed after every transition from a non-terminal state. What happens if the next state, $S_{t+1}$ is terminal? We just assign the corresponding value, $Q(S_{t+1}, A_{t+1})$ as zero.

We can see from the update function that each of these five events:

1. **s**tate at current time step, t.
2. **a**ction at time t.
3. **r**eward associated with next state
4. **s**tate at time step t +1.
5. **a**ction at time t +1.

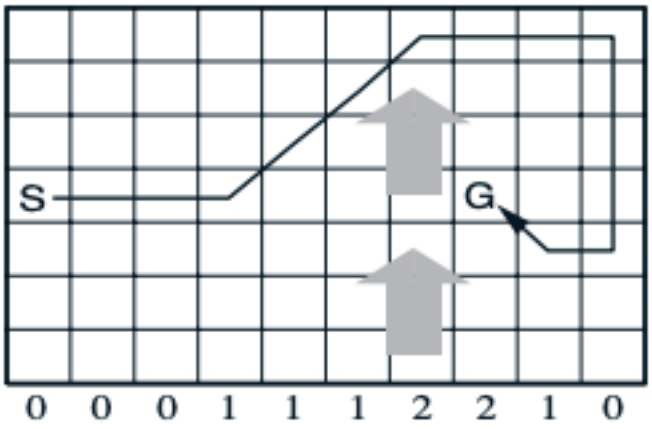
This list, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, gives Sarsa its creatively chosen name!

Windy Gridworld example:
All the way at the start of this guide we looked at our first Gridworld. In this game we made the assumption that every time the agent decides to make a move, it goes exactly where it has decided. But what happens if there is another force acting on the agent, changing where it ends up? This brings us to 'windy gridworld'.




*Sutton and Barto Figure 6.10*

**In the middle of the game there is a wind which will push an agent's next step upward by the number of cells given by the row of numbers along the x-axis. This is the path that Sarsa results in.**




*Sutton and Barto Figure 6.11*

## 8. λ Methods

λ return

Backups can be done over an average of n-step returns. This called a complex backup. This method will also reach convergence. TD(λ) can be thought of as a type of complex backup. Mathematically:

$$L_t = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{t+n}(V_t(S_{t+n})).$$

Comments on the formula:

- The average will contain all n-steps.
- These steps are weighted by $\lambda^{n-1}$, $\lambda \in [0,1]$.
- They are then normalised by a factor of $1-\lambda$ so that the weights are guaranteed to sum to 1.
- This backup is toward a return called the λ-return.

To apply TD(λ) prediction to state-action pairs, we need eligibility traces for each state-action pair. This is denoted as $E_t(s,a)$. There are three different ways to update this trace but the one we will use in our Easy21 code is called accumulating traces.

This is the only difference between the Sarsa(λ) and TD(λ) implementations.

When λ = 0 the λ return becomes

$$G_t^{t+1}(V_t(S_{t+1}))$$

This is the one step return! When λ = 0, the λ-return is the same as TD(0). The backup reduces to just its first component.

On the other end of the scale, when λ = 1, the backup is just its last component. This makes it the same as the Monte Carlo method called "constant-$\alpha$ MC".

Sarsa(λ) will often speed up control algorithms in comparison to one-step Sarsa because of the eligibility traces. This can be seen in the gridworld below. We can see that one-step Sarsa only rewards the immediate action taken to receive the reward *

but when the λ value is set to 0.9 the entire path receives some level of reward. How much reward each step receives is determined by how far away it is from the reward.

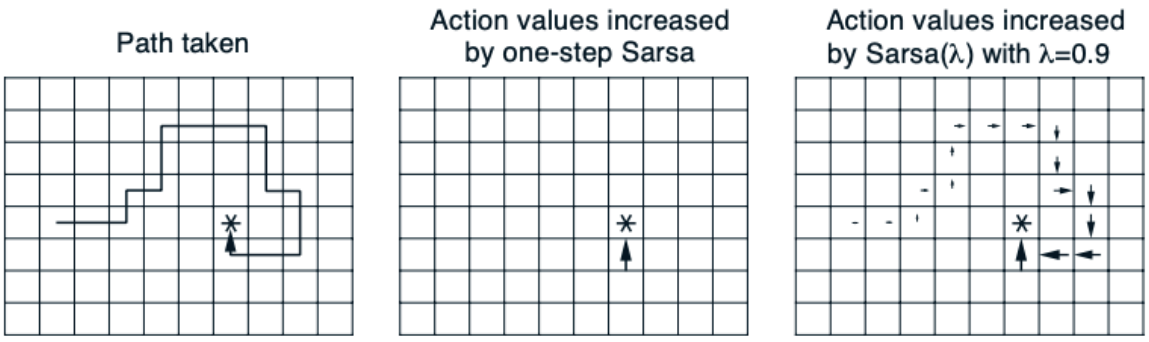



Figure 7.13: Gridworld example of the speedup of policy learning due to the use of eligibility traces.

In this case the reward is reduced by 0.9 with each step away from the reward.

Backward view of TD(λ)
There are two methods of TD learning - backwards view or forwards view. The methods are equivalent but the backwards view is simpler to implement and understand so I will use that.

The backward of TD(λ) will approximately equal the forward view. The benefit of the backward view is that is is simple to understand and easier to code. The backward view is causal.

When the backward view is implemented off policy it will exactly equal to forward view.

The backward view of TD(λ) is where eligibility traces come in. Mathematically:

$$E_t(s) = \gamma\lambda E_{t-1}(s), \qquad \forall s \in \mathcal{S}, s \neq S_t,$$

The eligibility trace for the state that has been visited in that step, $S_t$:

$$E_t(S_t) = \gamma\lambda E_{t-1}(S_t) + 1.$$

Gamma is the discount rate. λ here is called "the trace-decay parameter".
The eligibility trace for the visited state is called an accumulating trace. It will fade each time the state is not visited.

The goal of an eligibility trace is to tell the agent which states should be updated if it learns something new. In gridworld, imagine the agent learns that a path he has taken leads to a reward. The steps in that path should have their value functions updated but it also makes sense that the first step in that path might not be as valuable as the last step in the path. So if an agent keeps coming back to that last step it can weight it more heavily no matter how he reached it but does not get trapped into thinking the very step is as valuable.
Eligibility traces keep a record of recently visited states and tells the agent which states are eligible for a value function update.

At initial consideration, an algorithm that must update the eligibility trace of all steps in its path with every time step seems like it would take a much longer time than simply the one step method. In reality, for common values of λ (and potentially a decay value of $\gamma$) the eligibility trace for almost all states will be very close to zero. The eligibility trace will only change for recently listed states and such only these states need to be updated. Using this information, Sarsa(λ) algorithms implemented in practice on large reinforcement learning problems will only update the few states with greater than zero traces.

## Easy21 Code Part 3

**It's time to build the Sarsa(λ) function for Easy21.**

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize $S$, $A$
  Repeat (for each step of episode):
    Take action $A$, observe $R$, $S'$
    Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
    $E(S, A) \leftarrow E(S, A) + 1$
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha \delta E(s,a)$
      $E(s,a) \leftarrow \gamma \lambda E(s,a)$
    $S \leftarrow S'$; $A \leftarrow A'$
  until $S$ is terminal

*Sutton and Barto Sarsa Pseudocode*

```
def sarsa(Qsa, Ns, Nsa, Esa, Lambda):
    player_sum = random.randint(1,10)
    dealer_card = random.randint(1,10)
    terminal = 'not terminal'
    while terminal != 'terminal':
        # increase the number of visits to this state by 1
        Ns[dealer_card, player_sum] += 1
        # choose action using defined policy
        action = exploration_greedy(dealer_card, player_sum)
        # increase the number of times this action has been chosen in this
state
        Nsa[action, dealer_card, player_sum] += 1

        # find the next state in the game
        dealer_card_prime, player_sum_prime, terminal, reward =
list(step(action, dealer_card, player_sum))

        # if the game isn't finished delta can be calculated using a
combination of [state, action]
        # and [state_prime, action_prime]
        if terminal != 'terminal':
            action_prime = exploration_greedy(dealer_card_prime,
player_sum_prime)
            delta = reward + Qsa[action_prime, dealer_card_prime,
player_sum_prime] - Qsa[action, dealer_card, player_sum]
        else:
            # if the game is finished then the 'next' state cannot be used
to calculate delta
            delta = reward - Qsa[action, dealer_card, player_sum]

        # increase eligibility trace by 1
        Esa[action, dealer_card, player_sum] += 1

        # update value function
        Qsa[action, dealer_card, player_sum] += (1/Ns[dealer_card,
player_sum]) * delta * Esa[action, dealer_card, player_sum]

        # update eligibility trace
        Esa[action, dealer_card, player_sum] = Lambda * Esa[action,
dealer_card, player_sum]

        # redefine next state as current state
        player_sum = player_sum_prime
        dealer_sum = dealer_card_prime

    return Qsa, Ns, Nsa, Esa
```

We can walk through this function as well.

1. The first few lines of code are the same as in the Monte Carlo control function.
2. The difference in this function comes when we move to the next state in the game. We now need to save information about the original state and the next state. We do this by calling the next state 'prime'.
3. If the game isn't finished we can use this new prime state to calculate delta. We pick our next action using the greedy function and use it to calculate delta.

4. If the game is finished then the 'next' state does not exist and we simply calculate delta based on the current state.
5. Next we must increase the eligibility trace by 1.
6. Now we update the value function using the formula defined in the pseudocode.
7. Then we can update the eligibility trace with the discounted λ.
8. Finally we redefine the next state prime as the current state so that the game can carry on.
9. The function returns the updated matrices.

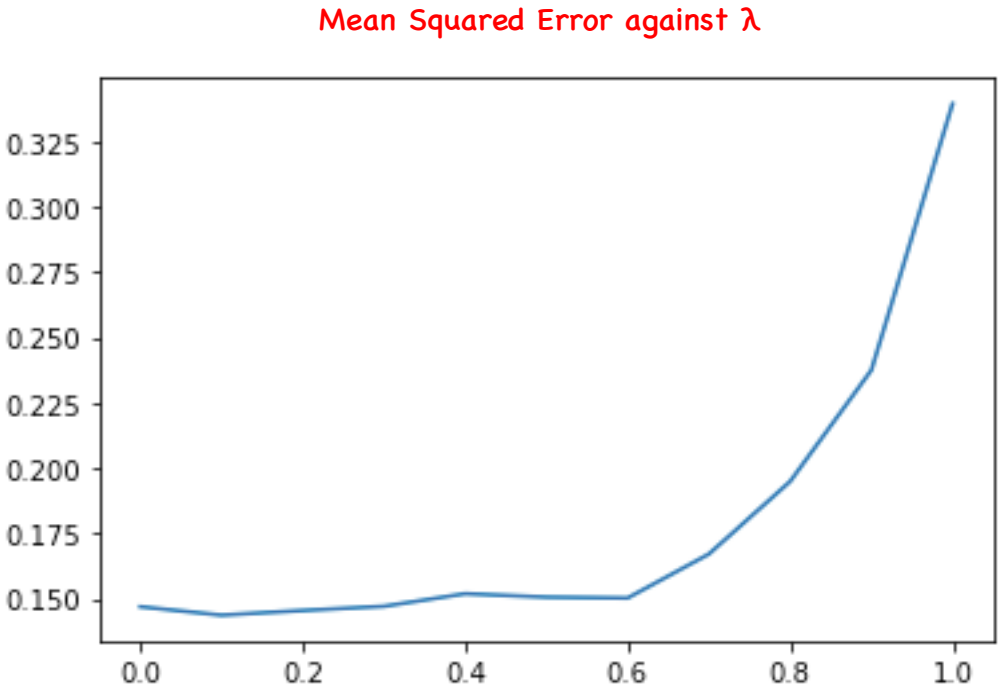
To display the results of this Sarsa(λ) we are instructed as follows:

Run the algorithm with parameter values λ ∈ {0, 0.1, 0.2, ..., 1}. Stop each run after 1000 episodes and report the mean-squared error $\sum_{s,a}\big(Q(s,a) - Q^*(s,a)\big)^2$ over all states $s$ and actions a, comparing the true values $Q\ (s,a)$ computed in the previous section with the estimated values $Q(s,a)$ computed by Sarsa. Plot the mean squared error against λ. For λ = 0 and λ = 1 only, plot the learning curve of mean-squared error against episode number.

### 1.Mean Squared Error against λ

```
mse = []
for Lambda in np.linspace(0,1,11):
    # initalise matrices
    sarsa_Qsa = np.zeros((2,11,22))
    sarsa_Ns = np.zeros((11,22))
    sarsa_Nsa = np.zeros((2,11,22))
    Esa = np.zeros((2,11,22))

    # update matrices
    for i in range(1000):
        sarsa_Qsa, sarsa_Ns, sarsa_Nsa, Esa = sarsa(sarsa_Qsa, sarsa_Ns,
sarsa_Nsa, Esa, Lambda)

    # calculate mean squared error
    # 2 * 10 * 21 = 420
    mse.append(np.sum(np.square(Qsa - sarsa_Qsa)) / 420)
```

```
plt.plot(np.linspace(0,1,11), mse)
```



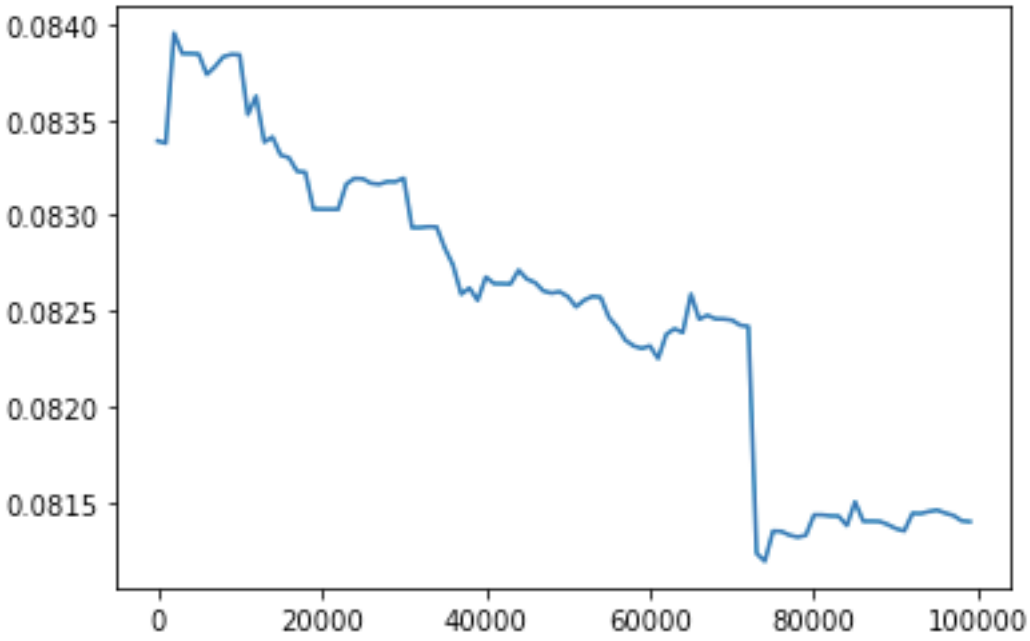

The Sarsa algorithm performs better when λ is closer to zero. This means that an algorithm closer to the TD(0) method performs better.

### 2.Mean Squared Error against Episodes when λ = 0

```
# plot mean squared error against episodes when lambda = 0
episodes_mse_zero = []
for i in range(100000):
    sarsa_Qsa, sarsa_Ns, sarsa_Nsa, Esa = sarsa(sarsa_Qsa, sarsa_Ns,
sarsa_Nsa, Esa, Lambda = 0)
    if i % 1000 ==0:
        episodes_mse_zero.append(np.sum(np.square(Qsa - sarsa_Qsa)) / 420)
```

```
plt.plot(np.arange(100) * 1000, episodes_mse_zero)
```

**Mean Squared Error against episodes when λ = 0**



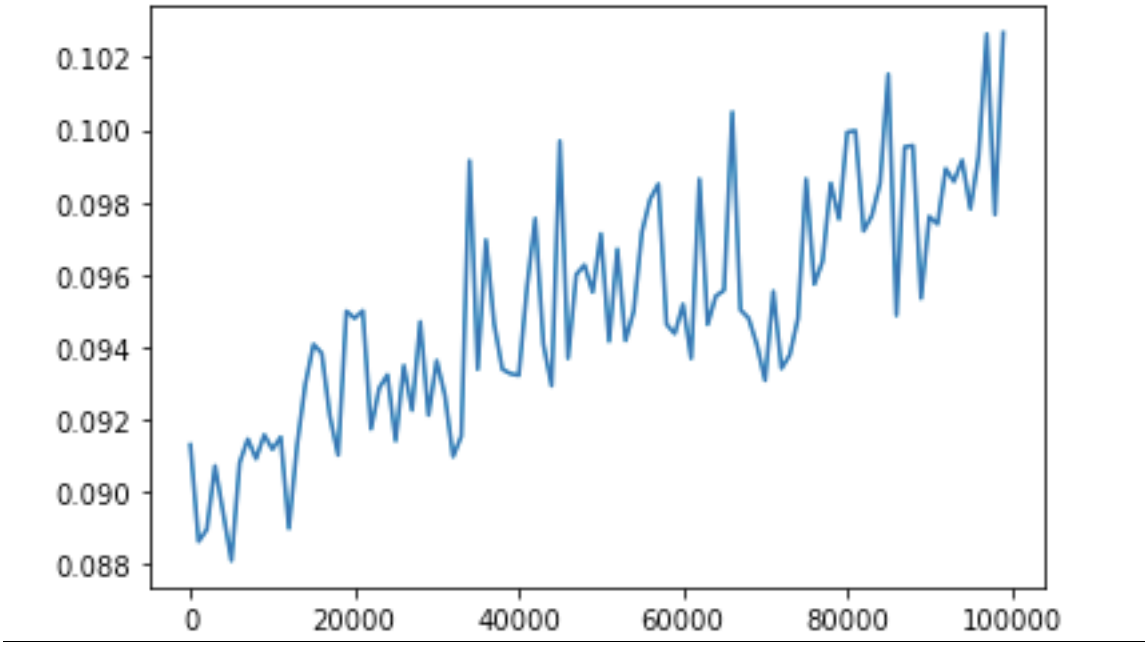

The MSE falls as the number of episodes increase when λ is equal to 0.

### 2.Mean Squared Error against Episodes when λ = 1

```
# plot mean squared error against episodes when lambda = 1
episodes_mse_one = []
for i in range(100000):
    sarsa_Qsa, sarsa_Ns, sarsa_Nsa, Esa = sarsa(sarsa_Qsa, sarsa_Ns,
sarsa_Nsa, Esa, Lambda = 1)
    if i % 1000 ==0:
        episodes_mse_one.append(np.sum(np.square(Qsa - sarsa_Qsa)) / 420)
```

```
plt.plot(np.arange(100) * 1000, episodes_mse_one)
```

## Mean Squared Error against episodes when λ = 1




The MSE rises as the number of episodes increase when $\lambda$ is equal to 1.

## 9. Discussion and Conclusion

What are the pros and cons of bootstrapping in Easy21?
In reinforcement learning, bootstrapping refers to an algorithm updating a value function using previously gathered estimates.
TD learnings methods bootstrap, MC methods do not.

The advantages of bootstrapping in Easy21 is that bootstrapping methods will often converge to the true optimum policy and value function more quickly and will use less computational power.

The main disadvantage of bootstrapping here is that it could introduce bias. In reinforcement learning the bias-variance tradeoff centres around how well the reinforcement signal is actually reflecting the true rewards.

> In the case of RL, variance refers to a noisy, but on average accurate value estimate, whereas bias refers to a stable, but inaccurate value estimate.

The bootstrapping methods could quickly reach what it thinks is an optimum policy but does not have enough information to find a different optimum.

Would you expect bootstrapping to help more in blackjack or Easy21? Why?
Bootstrapping should help more because Easy21 has 'negative' cards meaning that a player/dealer sequence could be much longer, going up and down within the range of 1 and 21. Bootstrapping will learn before these sequences end.

Beyond Easy21

We have seen some of the core reinforcement learning concepts being applied step by step to our card game Easy21 but there is so much more that reinforcement learning can do.

Deep reinforcement learning (DRL) can do much more complicated things. An example is a library called FinRL which is a library for automated trading in finance. DRL balances exploration and exploitation in the stock market and provides algorithmic trading strategies that are difficult for human traders to see.

A large area in reinforcement learning is in games. An example is AlphaGo. It was developed by Google to play the game Go and was the first program to defeat a

human professional. Go is a challenging and complex game that requires humans to think strategically - there are 10 to the power of 170 possible board configurations. AlphaGo had no previous knowledge of the game, it wasn't trained on which moves were good and which weren't, it simply began to play game after game, initially naively, to learn which actions helped it to win and which did not.

AlphaGo was an exercise in proving just how capable a reinforcement learning algorithms were. But there are other more practical applications in the real world. One such example is cooling data centres using the minimum amount of energy. All the information from the cooling system is passed to the deep reinforcement learning algorithm and it predicts how different actions to change the system will effect future energy use. It picks and recommends the optimum policy to save energy.

## Conclusion

Reinforcement learning is an ever growing area of machine learning. In this guide we have understood the basics of reinforcement learning theory, examined the mathematics behind it and applied what we have learnt through Python code. We have seen how to take the theoretical concept of teaching a puppy a trick to defining the optimum way to play a game of modified Blackjack.