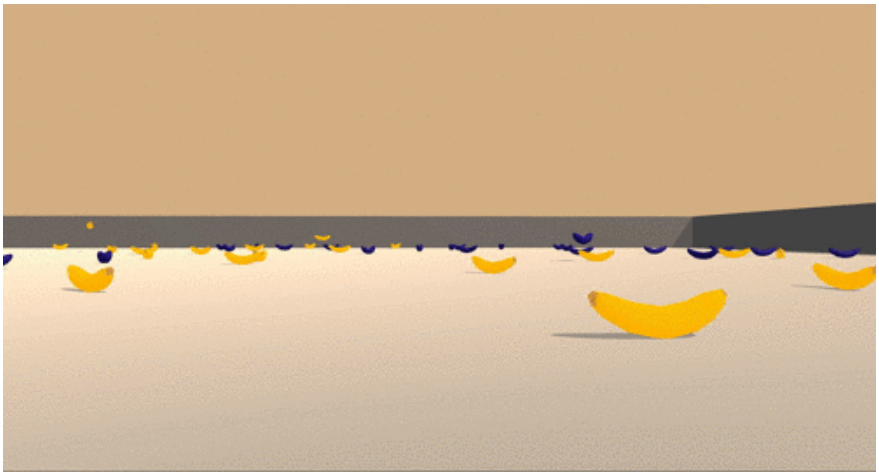
# Udacity Deep Reinforcement Learning —Project 1—Navigation

Mr Nisheed Rama
Apr 29 · 15 min read



## Introduction

There have been several breakthroughs with neural networks in areas like computer vision, machine learning, machine translation, and time series prediction. Combing neural networks with deep reinforcement learning allows us to create astounding computer intelligence. The best and most famous example is AlphaGo. AlphaGo is the first computer program to defeat a Go world champion and arguably the best player in history.

The complexity of Go means it has long been viewed as the most challenging of classic games for artificial intelligence. Despite decades of work, the Go programs were only able to play at the level of human amateurs. Traditional AI methods, which constructed a search tree over all possible positions, are limited due to the sheer number of possible

moves and difficulty of evaluating the strength of each possible position.

AlphaGo took a novel approach, and it combines advanced tree search with deep neural networks to capture the intuitive aspects of the game. The neural networks take a description of the board as inputs and outputs and process it through several different layers containing millions of neuron-like connections. One neural network, the "policy network," selects the next move to play. The other network, the "value network," predicts the winner of the game.

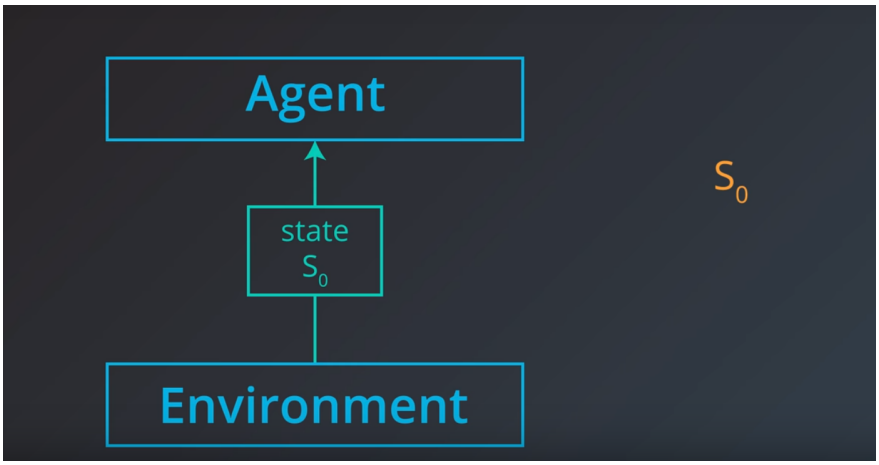
AlphaGo was then trained with a large number of strong amateur games to help it develop its an understanding of game profile for a reasonable player. Then it played against different versions of itself thousands of times, each time it learned from its mistakes and incrementally improved until it became a strong player. This process is known as reinforcement learning.

## Deep Reinforcement Learning Crash Course

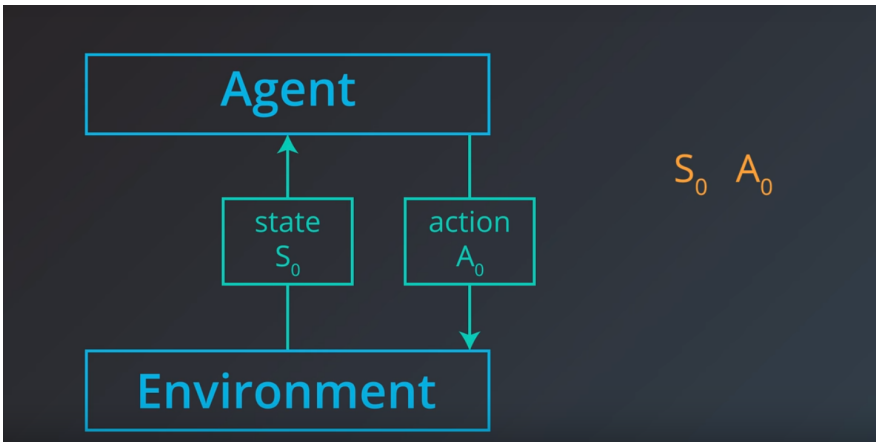
Reinforcement Learning refers to goal-oriented algorithms, which learn how to attain a complex goal. These algorithms can start from a blank slate, and under the right conditions achieve superhuman performance. Just as we teach children with rewards and discipline, these algorithms are rewarded with positive and negative rewards when they take good or bad actions. Rewarding good actions with positive reward is the reinforcement aspect of this approach.

Like humans, these reinforcement learning algorithms have to work in a delayed return environment as they have to wait to see the fruits of their decisions. In this environment, it is difficult to understand which action leads to which outcome over many steps. Deep Reinforcement Learning is a sequence of decision making where the environment model is unknown, and the agent has to learn the optimal behavior in the environment. It does this as described in a generalized way below:

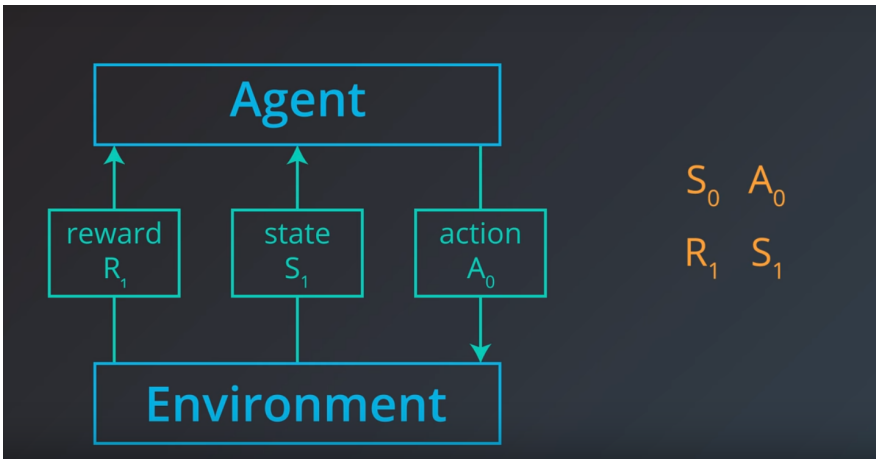
The agents stated with environments and a state S0, 0 is timestep 0.



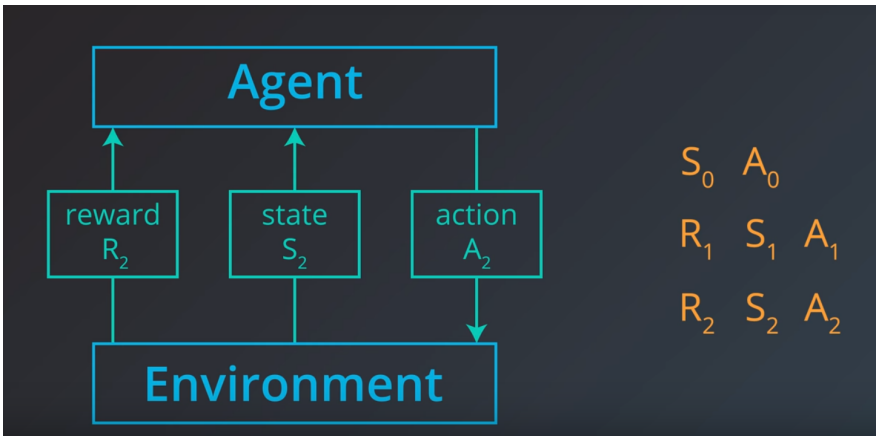

Based on the observation it chooses an action A0



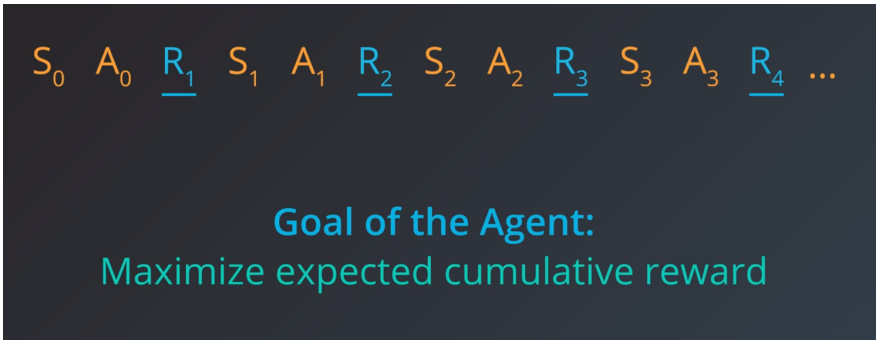

Based on the choice of action A0, the environment transitions to a new state S1 and gives some reward R1 to the agent.



The agent then chooses an action, A1. At time step 2, the process continues where the environment transitions to state S2 and gets the reward R2. The process continues the sequence of state, action, and rewards.




The agent has the goal of maximizing the cumulative reward or the sum of rewards attained over all time steps.




## Episodic vs. Continuing Tasks

1. **Episodic tasks** have a well-defined ending point. Playing chess is an episodic task, where the episode finishes when the game ends.

2. **Continuing Tasks** are tasks that continue forever, without end. An agent that buys and sell stocks could be model as a continuous task. These algorithms are more complicated then episodic algorithms.

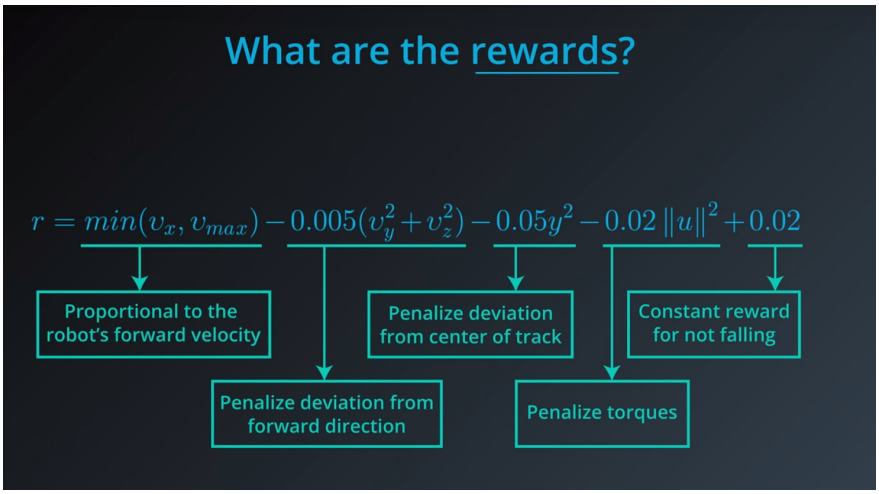
NB! Project 1 is Episodic.

## The Reward Hypothesis

As we discussed above, all agents formulate their goals in terms of maximizing expected cumulative reward. In a context of teaching a robot to walk this could be a trainer that watches the robot walking and rewards it for good walking form. This reward could potentially be subjective and not scientific at all.

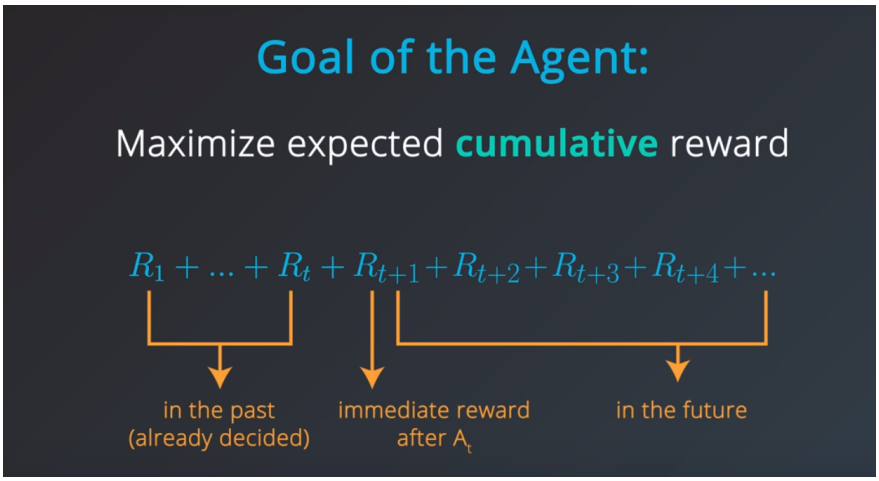
So how do we define reward? Before we address this, it is essential to note that the word "Reinforcement Learning" is a term from behavioral science. It refers to a stimulus that's delivered immediately after behavior to make the behavior more likely to occur in the future. It's an important defining hypothesis in reinforcement learning that we can always formulate an agents goal along the lines of maximizing expected cumulative reward. And we call this hypothesis, the "Reward Hypothesis."

Here is an example of how the reward was defined in DeepMind for teaching a robot how to walk. , Please check out this link. The research paper can be accessed here. Also, check out this cool video!

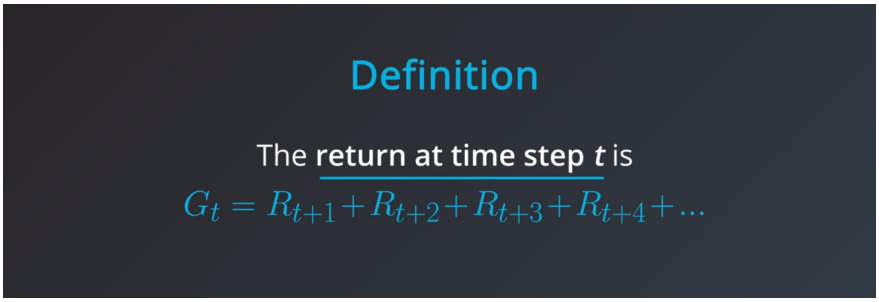



The agent needs to ensure it is maximizing cumulative reward. If the agent maximizes the reward it received in a single time step that would work well in the short term but could destabilize it, so it fell soon after. This way the agent is only learning to maximize initial reward. The cumulative reward will be small. There to maximize cumulative reward the agent needs to keep all time steps in mind. How exactly does it keep all time steps in mind?

It is important to keep in mind that all previous time steps are in the past and only future actions are in the agent's control.

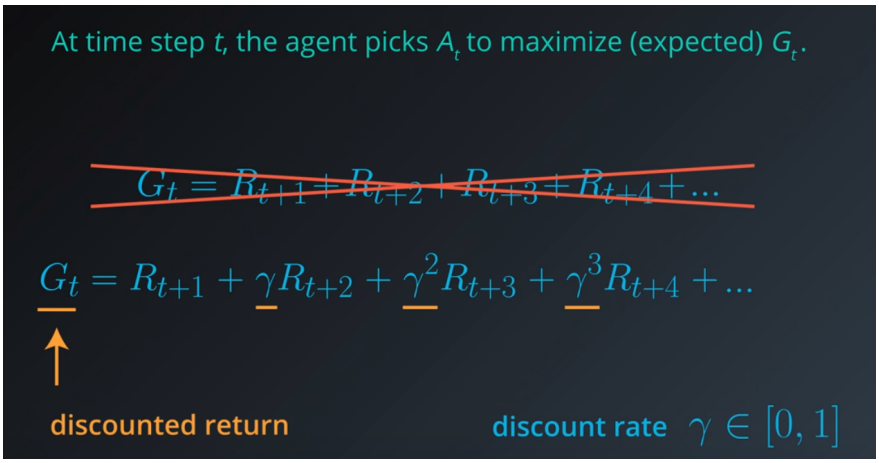



To explain this let's start with a definition. We will refer to the sum of rewards from the next time step onward as the return and denote it with a capital G.



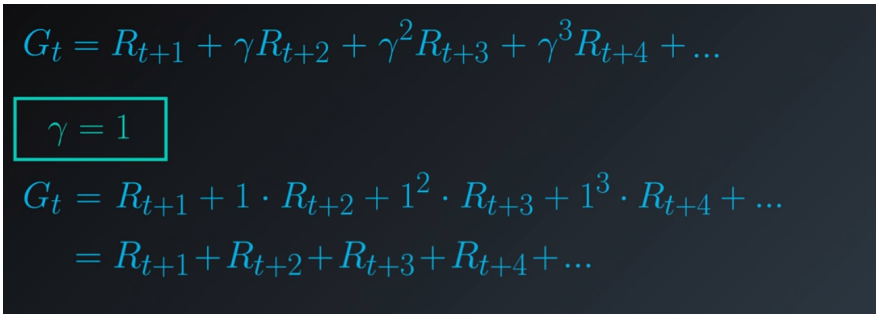

At any arbitrary time step, the agent will choose action towards the goal of maximizing the return. i.e., the agent seeks to maximize expected return.

A discount rate is used in rewards to value rewards that come sooner more highly and rewards that come later.

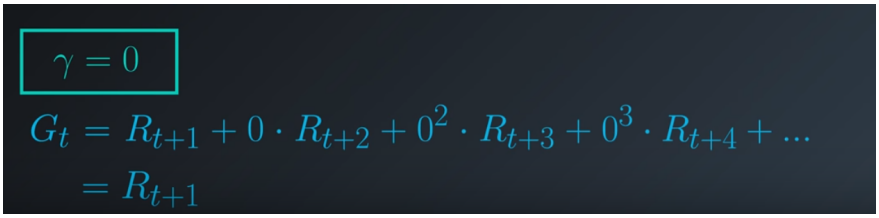



Gamma values are set and not learned by the agent. So how exactly might you set the value of gamma?

- Let's begin by looking at what happens when we set gamma to one. So we plug in one everywhere we see gamma, and we see it yields the original,
  completely un-discounted return from the previous videos.



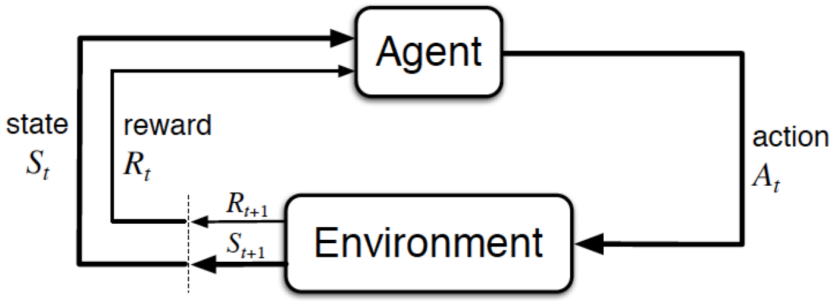

- If we set Gamma to 0, in this way, we see that the larger you make gamma, the more the agent cares about the distant future. And as gamma gets smaller and smaller, we get increasingly extreme discounting, wherein the most extreme case, the agent only cares about the most immediate reward.



## Markov Decision Process

In this section, we will discuss how to define a reinforcement learning problem as a **Markov Decision Process (MDP)**.

Markov decision process (MDP), a method to sample from a complex distribution to infer its properties. In an MDP, the agent has full knowledge about how the environment decides states and reward. The agent interacts with the environment by taking actions. The agent gets rewards and next_state information from the environment. This agent iterates through this process with the goal of maximizing the total reward by finding the optimum policy.




The agent-environment interaction in reinforcement learning. (Source: Sutton and Barto, 2017)

In general, the state space S is the set of **all nonterminal states**.

In continuing tasks (like the recycling task detailed in the video), this is equivalent to the set of **all states**.

In episodic tasks, we use S+ to refer to the set of **all states, including terminal states**.

The action space A is the set of possible actions available to the agent.

In the event that there are some states where only a subset of the actions are available, we can also use A(*s*) to refer to the set of actions available in state $s \in S$.

The **one-step dynamics** of the environment determine how the environment decides the state and reward at every time step. The dynamics can be defined by specifying

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

or each possible s', r, s and $a$.

A **(finite) Markov Decision Process (MDP)** is defined by:

- a (finite) set of states \mathcal{S}S (or \mathcal{S}^+S+, in the case of an episodic task)
- a (finite) set of actions \mathcal{A}A
- a set of rewards \mathcal{R}R
- the one-step dynamics of the environment
- the discount rate \gamma \in [0,1]$\gamma \in [0,1]$

## Policies

- A **deterministic policy** is a mapping $\pi: S \to A$. For each state $s \in S$, it yields the action $a \in A$ that the agent will choose while in state $s$.
- A **stochastic policy** is a mapping $\pi: S \times A \to [0,1]$. For each state $s \in S$ and action $a \in A$, it yields the probability $\pi(a|s)$ that the agent chooses action $a$ while in state $s$.

[source: Udacity course notes]

## State-Value Functions

- The **state-value function** for a policy $\pi$ is denoted $v\pi$. For each state $s \in S$, it yields the expected return if the agent starts in state ss and then uses the policy to choose its actions for all time steps. That is, $v\pi(s) \doteq E\pi[Gt|St=s]$. We refer to $v\pi(s)$ as the **value of state** $s$ **under policy** $\pi$.
- The notation $E\pi[\cdot]$ is borrowed from the suggested textbook, where $E\pi[\cdot]$ is defined as the expected value of a random variable, given that the agent follows policy $\pi$.

[source: Udacity course notes]

## Bellman Equations

- The **Bellman expectation equation for** $v\pi$ is: $v\pi(s) = E\pi[Rt+1 + \gamma v\pi(St+1)|St=s]$.

## Optimality

- A policy $\pi'$ is defined to be better than or equal to a policy $\pi$ if and only if $v\pi'(s) \geq v\pi(s)$for all s\in\mathcal $s \in$ S.
- An **optimal policy** $\pi*$ satisfies $\pi* \geq \pi$ for all policies $\pi$. An optimal policy is guaranteed to exist but may not be unique.
- All optimal policies have the same state-value function $v*$, called the **optimal state-value function**.

[source: Udacity course notes]

## Action-Value Functions

- The **action-value function** for a policy $\pi$ is denoted $\pi$. For each state $s \in$ S and action $a \in$ A, it yields the expected return if the agent starts in state s$s$, takes action $a$, and then follows the policy for all future time steps. That is, $q\pi(s,a) \doteq E\pi[Gt|St=s,At=a]$. We refer to $q\pi(s, a)$ as the **value of taking action** $a$ **in state** $s$ **under a policy** $\pi$ (or alternatively as the **value of the state-action pair** $s$, $a$).
- All optimal policies have the same action-value function $q*$, called the **optimal action-value function**.

## Optimal Policies

- Once the agent determines the optimal action-value function $q*$, it can quickly obtain an optimal policy $\pi*$ by setting $\pi*(s) = \text{argmax}a\in A(s)q*(s, a)$.

## Monte Carlo Methods

1. The Monte Carlo method; tries different actions in each state and given enough samples will have enough information to select the action that will give the maximum reward.

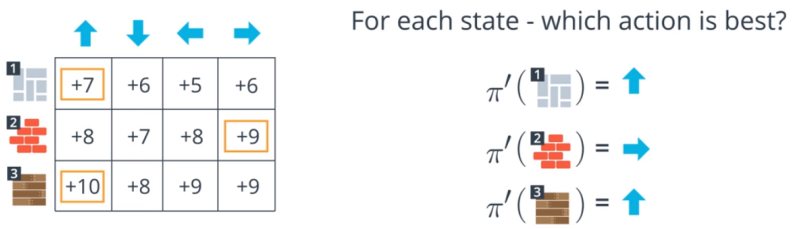
- **Equiprobable random policy**—is a stochastic policy, this involves selecting form each action with equal probability. The agent uses the collected set samples with state-action and the reward received to define a better policy.

## Monte Carlo Predictions

- When working with MDPs we can estimate the action-value function

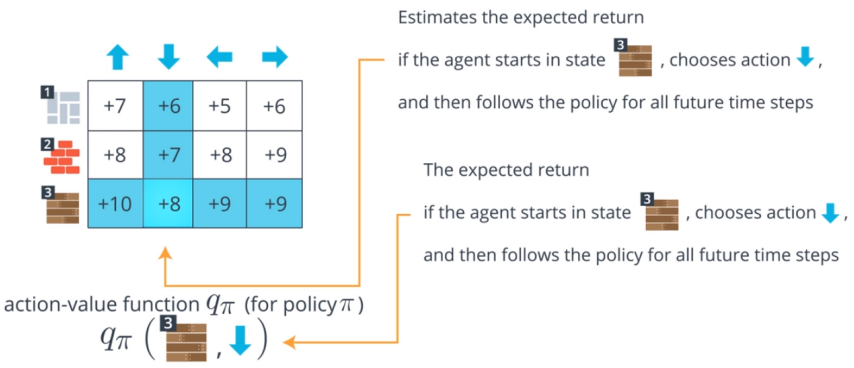
*qπ corresponding to* a policy π in a table known as a **Q-table**.

- Q-functions map the state-action pairs to the expected value. A Q-function takes as its input the visually impaired person's state and the action they tool and assigns it to the probable reward.




Example of a simple Q-table mapping state and action to reward

- A Q-table has one row for each state and one column for each action. The entry in the s*s*-th row and *a*-th column contains the agent's estimate for expected return that is likely to follow, if the agent starts in state *s*, selects action a*a*, and then henceforth follows the policy π.
- Each occurrence of the state-action pair *s*, *a* (*s* ∈ S, *a* ∈ A) in an episode is called a **visit to** *s*,*a*.
- There are two types of MC prediction methods (for estimating q_\piqπ):
- **First-visit MC** estimates *q*π(*s*, *a*) as the average of the returns following *only first* visits to *s*, *a* (that is, it ignores returns that are associated to later visits).
- **Every-visit MC** estimates *q*π(*s*, *a*) as the average of the returns following *all* visits to *s*, *a*.




## Greedy Policies

- A policy is greedy with respect to an action-value function estimate $Q$ *if for every state* $s \in S$, it is guaranteed to select an action $a \in A(s)$ such that it always selects the action with the maximum value $a = \text{argmax}_{a \in A(s)} Q(s,a)$. i.e. we need only select the action (or actions) corresponding to the column(s) that maximize the row.

- If an agent acts in a greedy manner it could be missing out on finding actions that have higher returns that are still to be discovered.

## Epsilon-Greedy Policies

A policy is $\epsilon$-**greedy** with respect to an action-value function estimate $Q$ *if for every state* $s \in S$:

- with probability 1-$\epsilon$, the agent selects the greedy action, and

- with probability $\epsilon$, the agent selects an action *uniformly* at random from the set of available (non-greedy **AND** greedy) actions.

All reinforcement learning agents face the **Exploration-Exploitation Dilemma**, where they must find a way to balance the drive to behave optimally based on their current knowledge (**exploitation**) and the need to acquire knowledge to attain better judgment (**exploration**).

In order for MC control to converge to the optimal policy, the **Greedy in the Limit with Infinite Exploration (GLIE)** conditions must be met:

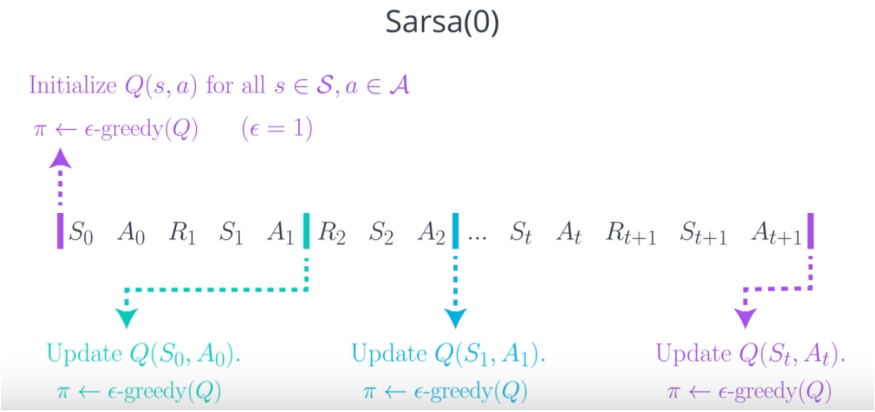
- every state-action pair $s,a$ (for all $s \in S$ and $a \in A(s)$) is visited infinitely many times, and
- the policy converges to a policy that is greedy with respect to the action-value function estimate $Q$.

**Temporal-Difference Methods**

Monte Carlo methods need to wait till the end of the episode to update the value function estimate, temporal difference (TD) methods update the value function after every time step.

**TD Control**

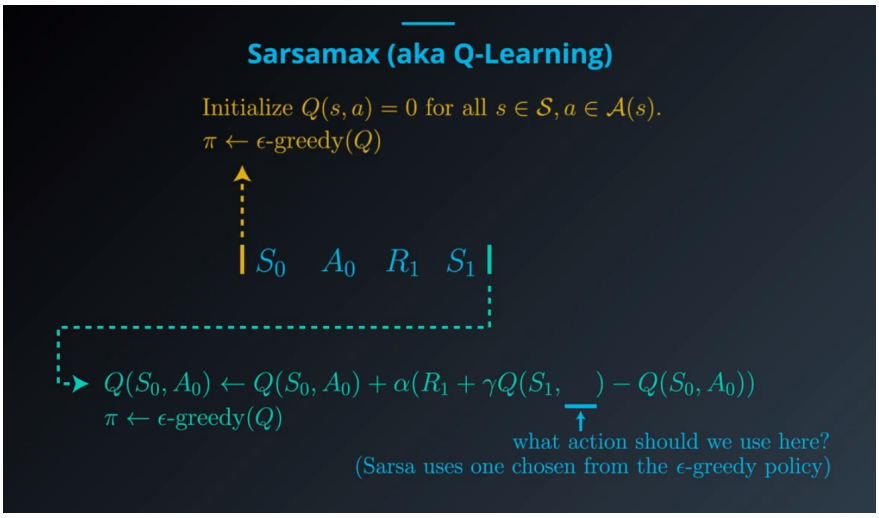
**Sarsa(0)** (or **Sarsa**)



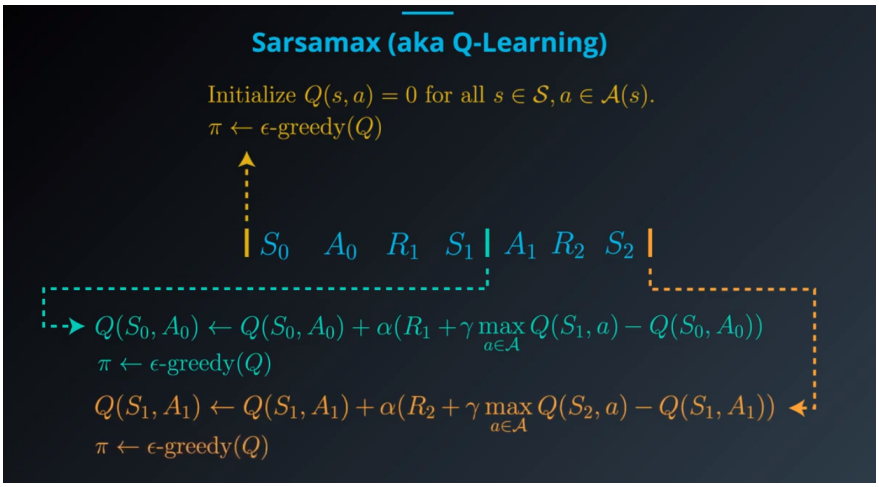

- This is identical to Monte Carlo where we use the $\epsilon$-**greedy** policy to select actions with the only difference being that we update the Q-Table at every time step instead of waiting till the end of the episode. It is guaranteed to converge to the optimal action-value function $q*$, as long as the step-size parameter $\alpha$ is sufficiently small and $\epsilon$ is chosen to satisfy the **Greedy in the Limit with Infinite Exploration (GLIE)**conditions.

**Sarsamax** (or **Q-Learning**)

Q-Learning works in the same way as Sarsa(0) except for one difference. Q-Learning updated the policy before choosing the next action. What action does it choose?




Q-Learning uses the action from the greedy policy as shown below.




Q-Learning is guaranteed to converge to the optimal action-value function $q_*$, under the same conditions that guarantee convergence of the Sarsa control algorithm.

## Deep Q-Networks

Deep Q-Learning algorithm represents the optimal action-value function q_*$q_*$ as a neural network (instead of a table).

Reinforcement learning is notoriously unstable when neural networks are used to represent the action values. Deep Q-Learning algorithm addresses these instabilities by using **two key features**:

- **Experience Replay:** a rolling history of past data via a re-play pool. Using the replay pool the behavior distribution is averaged out over many of its previous states smoothing out learning and avoiding oscillations. This has the advantage that the step of the updates is potentially used in many weight updates.
- **Fixed Q-Targets:** use a target network to represent the old Q-Function, which will be used to calculate the loss of every action during training. At every step, the q-function values change so we can use a single network and the value estimates could spiral out of control.

### Deep Q-Learning Algorithm

Please take the time now to read the research paper that introduces the Deep Q-Learning algorithm.

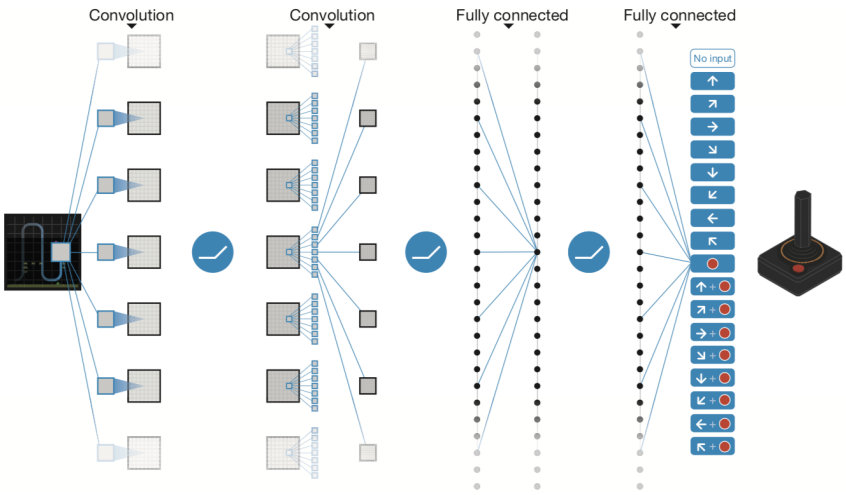



Illustration of DQN Architecture (Source)

### Deep Q-Learning Improvements

Several improvements to the original Deep Q-Learning algorithm have been suggested. Over the next several videos, we'll look at three of the more prominent ones.

- **Double DQN:** Deep Q-Learning tends to overestimate action values. Double Q-Learning has been shown to work well in practice to help with this.

- **Prioritized Experience Re-play:** Deep Q-Learning samples experience transitions *uniformly* from a replay memory. Prioritized experienced replay is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability.
- **Dueling DQN:** Currently, in order to determine which states are (or are not) valuable, we have to estimate the corresponding action values *for each action*. However, by replacing the traditional Deep Q-Network (DQN) architecture with a dueling architecture, we can assess the value of each state, without having to learn the effect of each action.

## Enough Theory: Lets look At My Code Implementation

Here is a link to my git repo if you want to clone this project and look at the code.

To get a better understanding of the project and environment, please refer to my Readme.md

The code I use in my implementation is from the "Lunar Lander" tutorial from the Deep Reinforcement Learning Nanodegree. I made some minor modification to make it work for the Navigation Project.

My code consists of:

model.py: this is the implementation of the Q-Network class. It is a regular fully connected DNN using the PyTorch Framework. Architecture is as follows:

- The input layer which size depends on the state_size parameter passed in the constructor
- 2 hidden fully connected layers of 64 nodes.
- the output layer which size depends on the action_size parameter passed in the constructor

dqn_agent.py: the DQN agent and replay buffer implementation.

**The DQN agent class has the following methods:**

**constructor():**

- Initializes 2 instances of the Neural Network: the *target* network and the *local* network and the memory buffer (*Replay Buffer*)

**step() :**

- Stores state, action, reward, next_state, done in the Replay Buffer/Memory
- Every 4 steps and if there are enough samples in the memory buffer, it updates the *target* network weights with the current weight values from the *local* network.

**act():** which returns actions for the given state as per current policy

**learn():** which update the Neural Network value parameters using a batch of experiences from the Replay Buffer.

**soft_update():** is called during the learning process to update the value from the *target* Neural Network using the *local* network weights.

**The ReplayBuffer class has the following methods:**

**add():** used to add an experience step into memory

**sample():** used to randomly sample a batch of experience steps for learning

**DQN_Navigation_Banana_Project.ipynb**

This is my Jupiter notebook that has the reference project starter code and the code that I've implemented to train the agent.

## DQN Hyperparameters

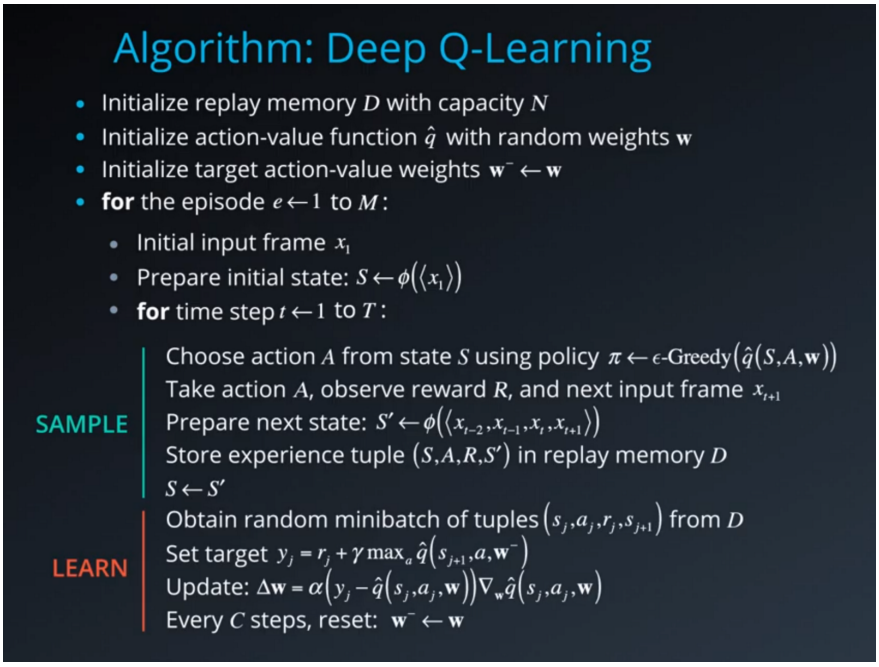
```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size (default: 64)
GAMMA = 0.9965          # discount factor (default: 0.99)
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

## The Architecture

```
Input nodes (37) -> Fully Connected Layer (64 nodes, Relu
activation) -> Fully Connected Layer (64 nodes, Relu
activation) -> Ouput nodes (4)
```

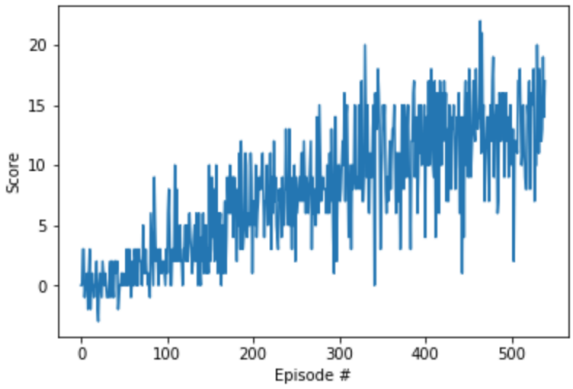
The Neural Networks use Adam optimizer with a learning rate LR=5e-4 and are trained using a BATCH_SIZE=64.

## Learning Algorithm




## Results

```
Episode 100     Average Score: 1.01
Episode 200     Average Score: 4.62
Episode 300     Average Score: 7.74
Episode 400     Average Score: 10.72
Episode 500     Average Score: 12.64
Episode 540     Average Score: 13.00
Environment solved in 440 episodes!     Average Score: 13.00
```




My agent solved the task in 440 episodes where it was able to receive an average reward (over 100 episodes) of at least 13. (in comparison the default solution was benchmarked to be able to solve the project in 1800 episodes.

## Ideas for Future Work

As described in the optional challenge a more challenging task would be to implement the learning directly from pixels. To do this, i would need to add a convolution Neural Network at the input to process the raw pixel values.

Other enhancements to increase performance would be to use:

- **Double DQN:** as Deep Q-Learning tends to overestimate action values. Double Q-Learning has been shown to work well in practice to help with this.
- **Prioritized Experience Re-play:** Deep Q-Learning samples experience transitions *uniformly* from a replay memory. Prioritized experienced replay is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability.
- **Dueling DQN:** Currently, in order to determine which states are (or are not) valuable, we have to estimate the corresponding

action values *for each action*. However, by replacing the traditional Deep Q-Network (DQN) architecture with a dueling architecture, we can assess the value of each state, without having to learn the effect of each action.