Reinforcement Learning

Project Report.

EKLAVYA MENTORSHIP PROGRAM
At
Society of Robotics And Automation,
Veermata Jijabai Technological Institute
Mumbai.

October 2021

Acknowledgement

This project would not have been possible without the support of our seniors, and SRA in general. To our seniors, thank you for encouraging us and helping us with all the challenges and doubts we faced throughout this project. A big thanks to SRA for giving us this wonderful opportunity to work on this project.

Special Thanks to our amazing mentors:

- Kush Kothari
- Chaitravi Chalke
- Reshmika Nambhar

Mentees:

- Chirag Shelar
  chiragshelar1428@gmail.com
  +918928427371
- Himanshu Chougule
  himanshuchougule3430@gmail.com
  +919819655699

Table of Contents

1. Project Overview.

To understand Reinforcement Learning and to implement RL agents for OpenAI Gym environments from scratch.

1.1 Major Objectives:

- Learn about Q-Learning and Bandit Learning methods.
- To complete 4-5 gym environments in OpenAI Gym.

1.2 Technologies used:

- Conda environment - Jupyter notebook:
  - Coding for the project was entirely done on Jupyter notebook.
- OpenAI gym:
  - The gym library is a collection of test problems — **environments** — that you can use to work out your reinforcement learning algorithms.
  - These environments have a shared interface, allowing you to write general algorithms.
  - In simple words, Gym provides the environment and we provide the algorithms to solve them.
- Stable-Baselines3:
  - Stable Baselines is a set of improved implementations of Reinforcement Learning (RL) algorithms based on OpenAI Baselines.
  - We are using Stable-Baselines3 which is essentially an up to date version of stable baselines also having tensorboard support.
- Python NumPy and Matplotlib libraries:
  - NumPy is the fundamental package for scientific computing with Python. We have mainly used NumPy for its random number capabilities and operations on arrays.
  - Matplotlib is a cross-platform, data visualization and graphical plotting library for Python and its numerical extension NumPy.

1.3 What to expect?

- A simple solution to the Multi Armed Bandit Problem using Epsilon Greedy Algorithm
- Solution to various Environments in openAI gym having discrete and continuous action spaces.

2. Introduction:

2.1 Reinforcement Learning In a Nutshell:

- Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making.
- It is distinguished from other computational approaches by its emphasis on learning by an agent from direct interaction with its environment, without requiring exemplary supervision or complete models of the environment.
- Reinforcement learning is the first field to seriously address the computational issues that arise when learning from interaction with an environment in order to achieve long-term goals.

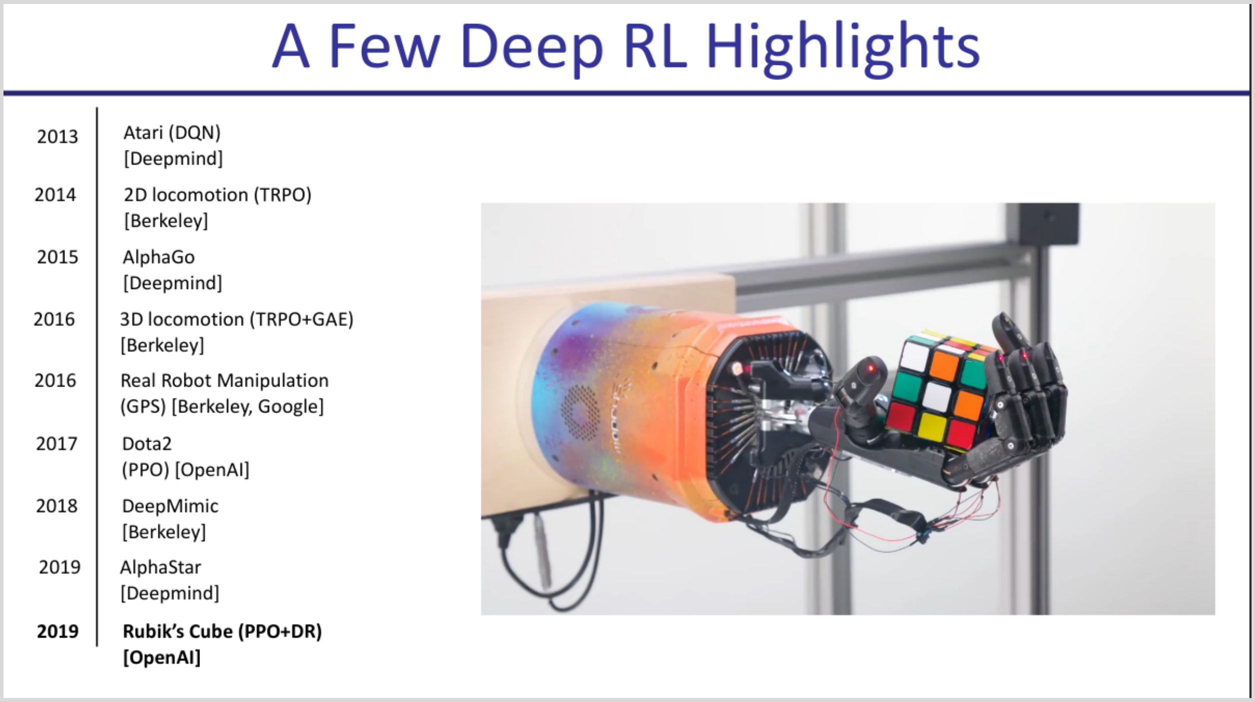
In simpler words,

- Reinforcement learning is a machine learning training method based on rewarding desired behaviors and/or punishing undesired ones.
- In general, a reinforcement learning agent is able to perceive and interpret its environment, take actions and learn through trial and error.

Note:

- Reinforcement learning uses the formal framework of Markov decision processes to define the interaction between a learning agent and its environment in terms of states, actions, and rewards. *More on that later*
- This framework is intended to be a simple way of representing essential features of the artificial intelligence problem.
- These features include a sense of cause and effect, a sense of uncertainty and nondeterminism, and the existence of explicit goals.

2.2 Notable Examples



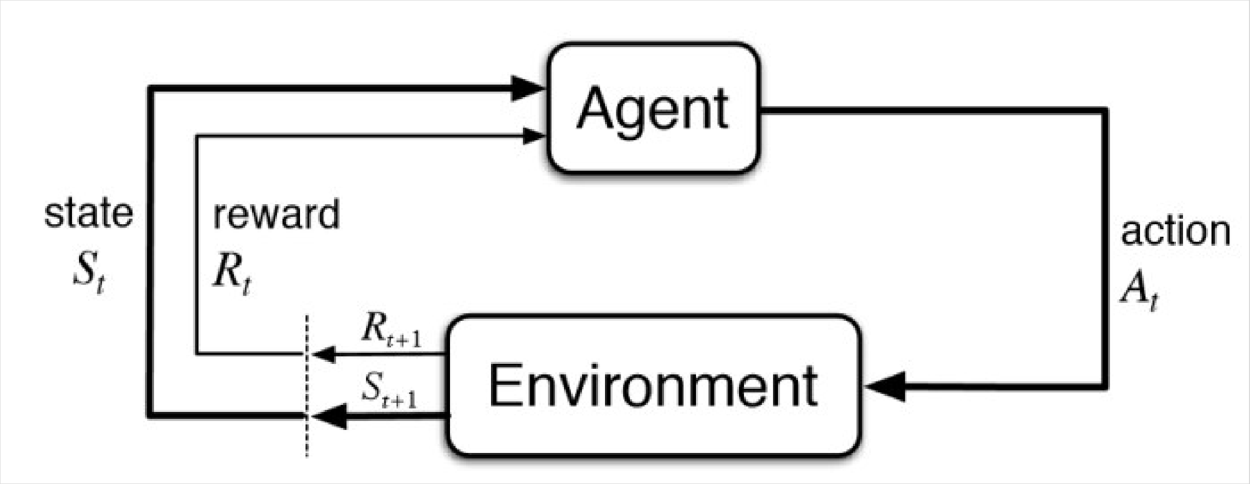
A Few Deep RL Highlights

2013	Atari (DQN) [Deepmind]
2014	2D locomotion (TRPO) [Berkeley]
2015	AlphaGo [Deepmind]
2016	3D locomotion (TRPO+GAE) [Berkeley]
2016	Real Robot Manipulation (GPS) [Berkeley, Google]
2017	Dota2 (PPO) [OpenAI]
2018	DeepMimic [Berkeley]
2019	AlphaStar [Deepmind]
2019	**Rubik's Cube (PPO+DR)** **[OpenAI]**

- Some notable examples of RL in particular Deep RL
- Atari game Breakout took around 36 hours of training with the DQN in order to achieve commendable results!
- The agents created in Dota2 were able to defeat pro players at their own game! And did really well in the 5v5 matchup!!
- As you can see DeepMind by Google and OpenAI are two organisations with insane accomplishments in the field of Reinforcement Learning

3. Tabular Solution Methods

3.1 Markov Decision Process (MDP)



This is a pretty well-known diagram of what actually happens in RL in a simplified way.

- The learner and decision maker is called the **agent.**
- The thing it interacts with, comprising everything outside the agent, is called the **environment**.
- These interact continually, the agent selecting **actions** and the environment responding to these actions and presenting **new** situations to the agent.
- The environment also gives rise to **rewards**, special numerical values that the agent seeks to **maximize** over time through its choice of actions.
- Basically, If you have a problem you want to solve, if you can map it to an MDP, it means you can run a reinforcement algorithm on it

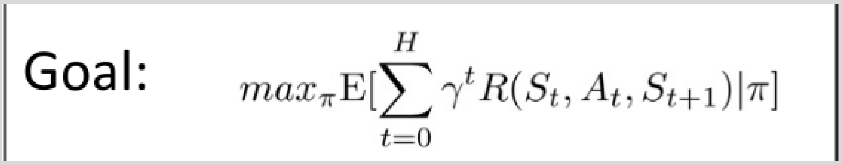
An MDP is defined by:

- Set of states S
- Set of actions A
- Transition function P(s' | s, a)
- Reward function R(s, a, s')
- Start state s0
- Discount factor γ
- Horizon H

To explain these terms:

- There's a set of states, then a set of actions the agent gets to choose from.
- There's a transition function that defines the probability of ending up in state S' at the next time, given at the current time, the agent is in state s and took action a.
- There's a reward function that assigns reward for that transition when you were in a state s, took action a, landed in state s prime.
- There is a start state, and a discount factor γ that essentially captures that things that are further in the future, we might care less about.
- For eg, we care more about getting a reward today than getting a reward, let's say a year from now and so the discounting says we should discount future rewards.
- Horizon H is basically Number of Timesteps

The end goal:

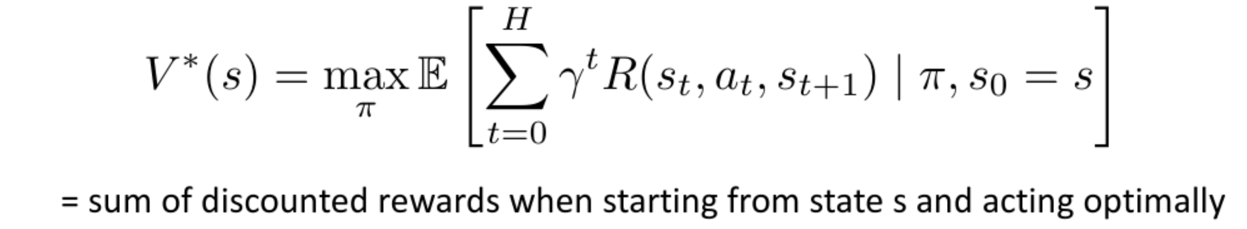


Goal: $$max_\pi \mathrm{E}[\sum_{t=0}^{H} \gamma^t R(S_t, A_t, S_{t+1})|\pi]$$

- Formally the goal is to maximize the expected, discounted sum of rewards accumulated over time. And we want to find a policy that does that.
- The easiest way to explain policy is that it is the agent's strategy or A policy defines the learning agent's way of behaving at a given time
- The actual definition is "A policy defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states."
- Sometimes, the policy can be stochastic instead of deterministic. In such a case, instead of returning a unique action a, the policy returns a probability distribution over a set of actions.
- In general, the goal of any RL algorithm is to learn an optimal policy that achieve a specific goal
- There are two ways to solve an MDP in case of Exact Solution Methods. **Value iteration** and **Policy iteration**

3.2 Value Iteration:

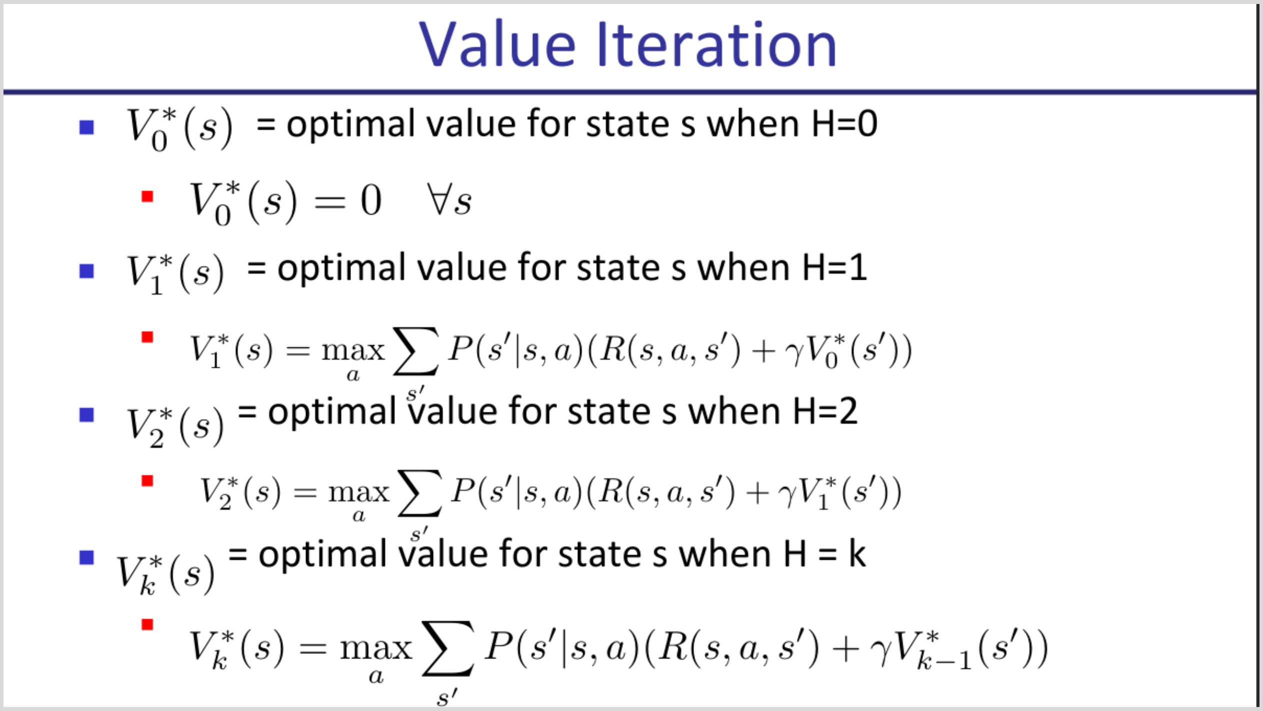
To understand Value Iteration we need to know that an optimal Value function is



$$V^*(s) = \max_\pi \mathbb{E}\left[\sum_{t=0}^{H} \gamma^t R(s_t, a_t, s_{t+1}) \mid \pi, s_0 = s\right]$$

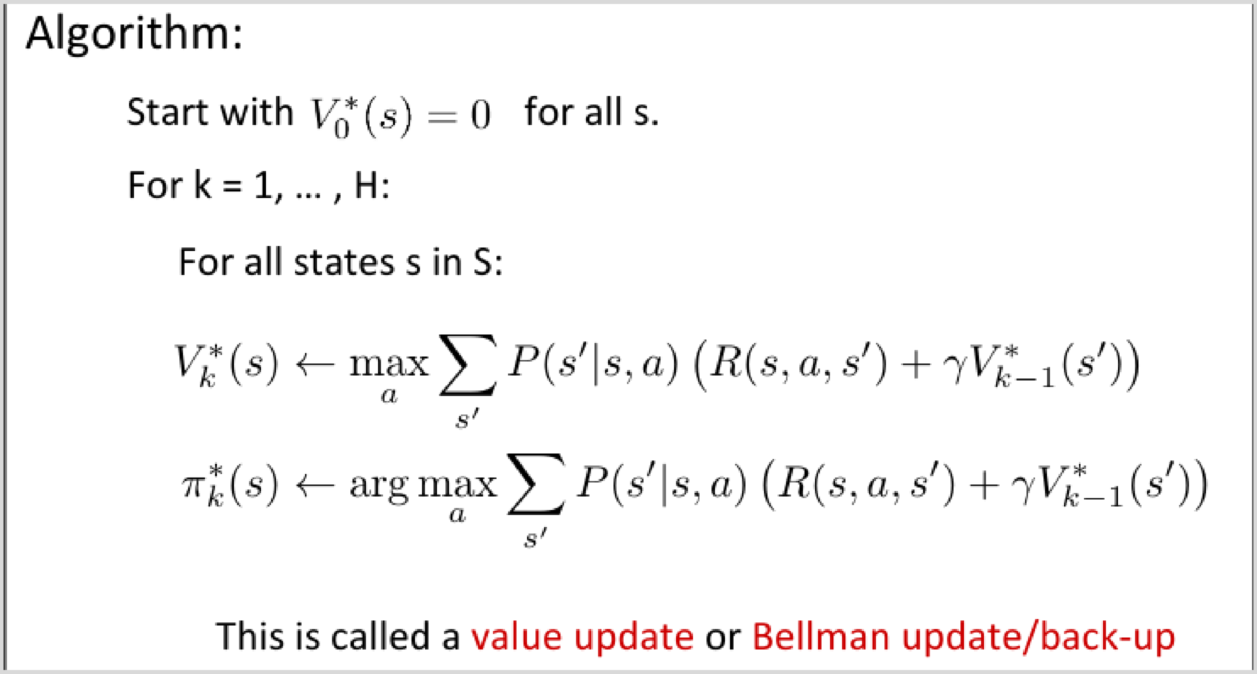
= sum of discounted rewards when starting from state s and acting optimally

- In value iteration, you start with a random value function and then find a new (improved) value function in an iterative process, until reaching the optimal value function.
- Notice that you can easily derive the optimal policy from the optimal value function. This process is based on the optimality Bellman operator.
- Value Iteration works on principle of “ Optimal value function —-> optimal policy”



Value Iteration

- $V_0^*(s)$ = optimal value for state s when H=0
  - $V_0^*(s) = 0 \quad \forall s$
- $V_1^*(s)$ = optimal value for state s when H=1
  - $V_1^*(s) = \max_a \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma V_0^*(s'))$
- $V_2^*(s)$ = optimal value for state s when H=2
  - $V_2^*(s) = \max_a \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma V_1^*(s'))$
- $V_k^*(s)$ = optimal value for state s when H = k
  - $V_k^*(s) = \max_a \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma V_{k-1}^*(s'))$

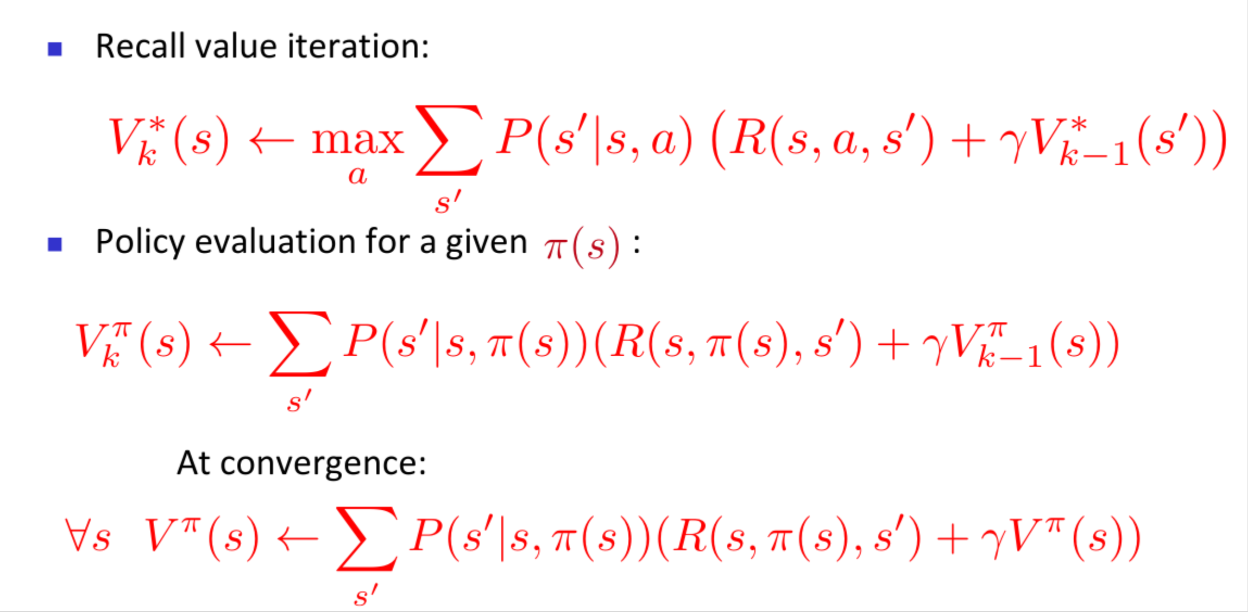


Algorithm:

```
Start with $V_0^*(s) = 0$ for all s.
For k = 1, … , H:
    For all states s in S:
        $V_k^*(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)\left(R(s,a,s') + \gamma V_{k-1}^*(s')\right)$
        $\pi_k^*(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s,a)\left(R(s,a,s') + \gamma V_{k-1}^*(s')\right)$
```

This is called a value update or Bellman update/back-up

3.3 Policy Evaluation:



- Recall value iteration:

$$V_k^*(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)\left(R(s,a,s') + \gamma V_{k-1}^*(s')\right)$$

- Policy evaluation for a given $\pi(s)$:

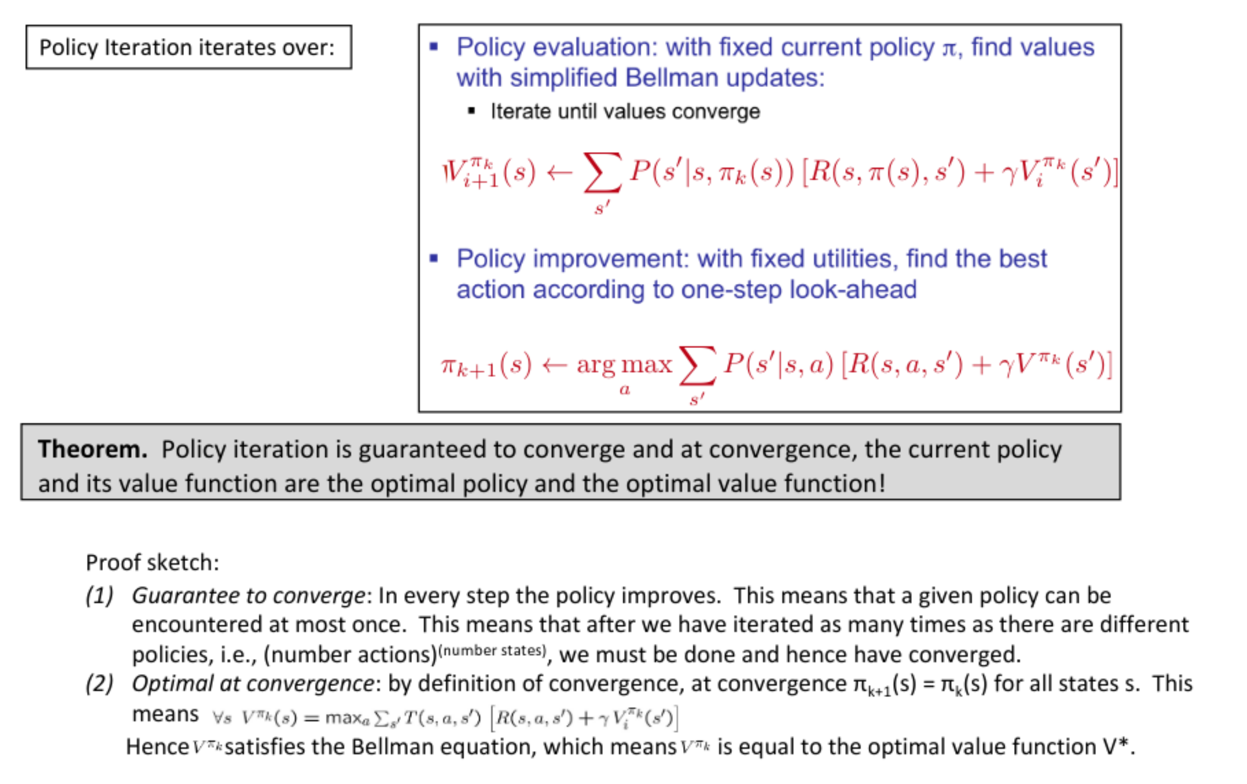
$$V_k^\pi(s) \leftarrow \sum_{s'} P(s'|s,\pi(s))(R(s,\pi(s),s') + \gamma V_{k-1}^\pi(s))$$

At convergence:

$$\forall s \quad V^\pi(s) \leftarrow \sum_{s'} P(s'|s,\pi(s))(R(s,\pi(s),s') + \gamma V^\pi(s))$$

- In policy evaluation, we fix the policy. we don't get to max over actions we're going to fix the policy and then it’s the same equation, but there's no max anymore.
- Once we fixed the policy, we can run value iteration for that fixed policy And that's called **policy evaluation.**

3.4 Policy Iteration:



Policy Iteration iterates over:

- Policy evaluation: with fixed current policy π, find values with simplified Bellman updates:
  - Iterate until values converge

$$V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s'} P(s'|s,\pi_k(s))\left[R(s,\pi(s),s') + \gamma V_i^{\pi_k}(s')\right]$$

- Policy improvement: with fixed utilities, find the best action according to one-step look-ahead

$$\pi_{k+1}(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V^{\pi_k}(s')\right]$$

Theorem. Policy iteration is guaranteed to converge and at convergence, the current policy and its value function are the optimal policy and the optimal value function!

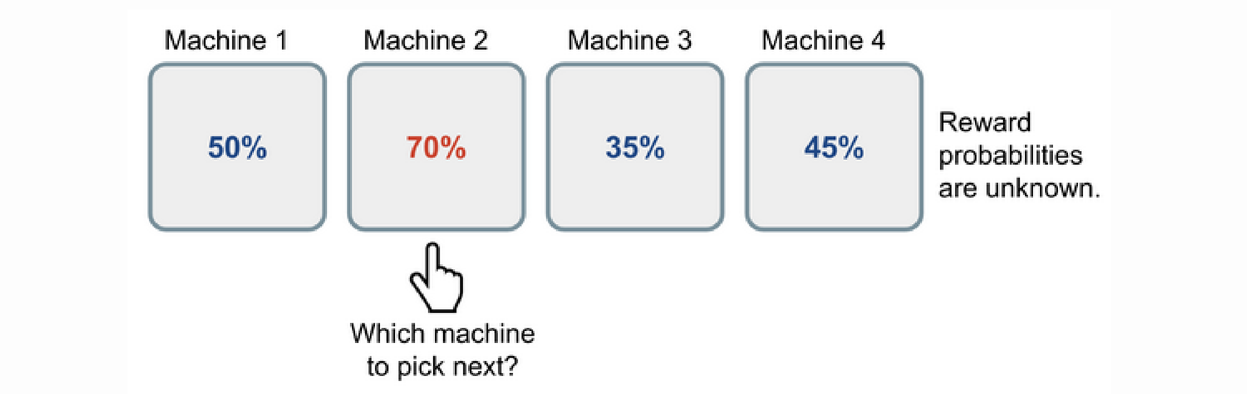
Proof sketch:

(1) *Guarantee to converge*: In every step the policy improves. This means that a given policy can be encountered at most once. This means that after we have iterated as many times as there are different policies, i.e., (number actions)$^{\text{(number states)}}$, we must be done and hence have converged.

(2) *Optimal at convergence*: by definition of convergence, at convergence $\pi_{k+1}(s) = \pi_k(s)$ for all states s. This means $\forall s \;\; V^{\pi_k}(s) = \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V_i^{\pi_k}(s')\right]$
Hence V^{π_k} satisfies the Bellman equation, which means V^{π_k} is equal to the optimal value function V*.

- Policy iteration includes: policy evaluation + policy improvement, and the two are repeated iteratively until policy converges
- The algorithms for **policy evaluation** and **finding optimal value function** are highly similar except for a max operation (as highlighted)
- Similarly, the key step to **policy improvement** and **policy extraction** are identical except the former involves a stability check.
- In **policy iteration** algorithms, you start with a random policy, then find the value function of that policy (policy evaluation step), then find a new (improved) policy based on the previous value function, and so on. In this process, each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Given a policy, its value function can be obtained using the *Bellman operator*.

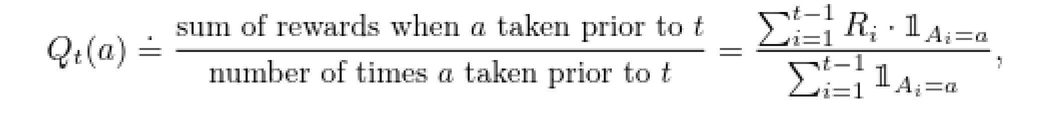
4. Multi armed Bandit Problem:


Machine 1
Machine 2
Machine 3
Machine 4
50%
70%
35%
45%
Reward probabilities are unknown.
Which machine to pick next?

- Those old slot machines with the single lever on the side which always take your money — those are called one-armed bandits.
- Imagine a whole bank of those machines lined up side-by-side, all paying out at different rates and values. This is the idea of a multi-armed bandit.
- If you're a gambler who wants to maximize your winnings, you obviously want to play the machine with the highest payout. But you don't know which machine this is. You need to **explore** the different machines over time to learn what their payouts are, but you simultaneously want to **exploit** the highest paying machine.
- A similar scenario is Richard Feynman's restaurant problem. Whenever he goes to a restaurant, he wants to order the tastiest dish on the menu, but he has to order everything available to find what is that best dish.
- This balance of exploitation, the desire to choose an action which has paid off well in the past, and exploration, the desire to try options which may produce even better results, is what multi-armed bandit algorithms were developed for.
- To solve the multi armed bandit problem we use different algorithms like Greedy, Epsilon Greedy, UCB1. We have used the Epsilon Greedy Algorithm for this.

4.1 The Epsilon Greedy Way.

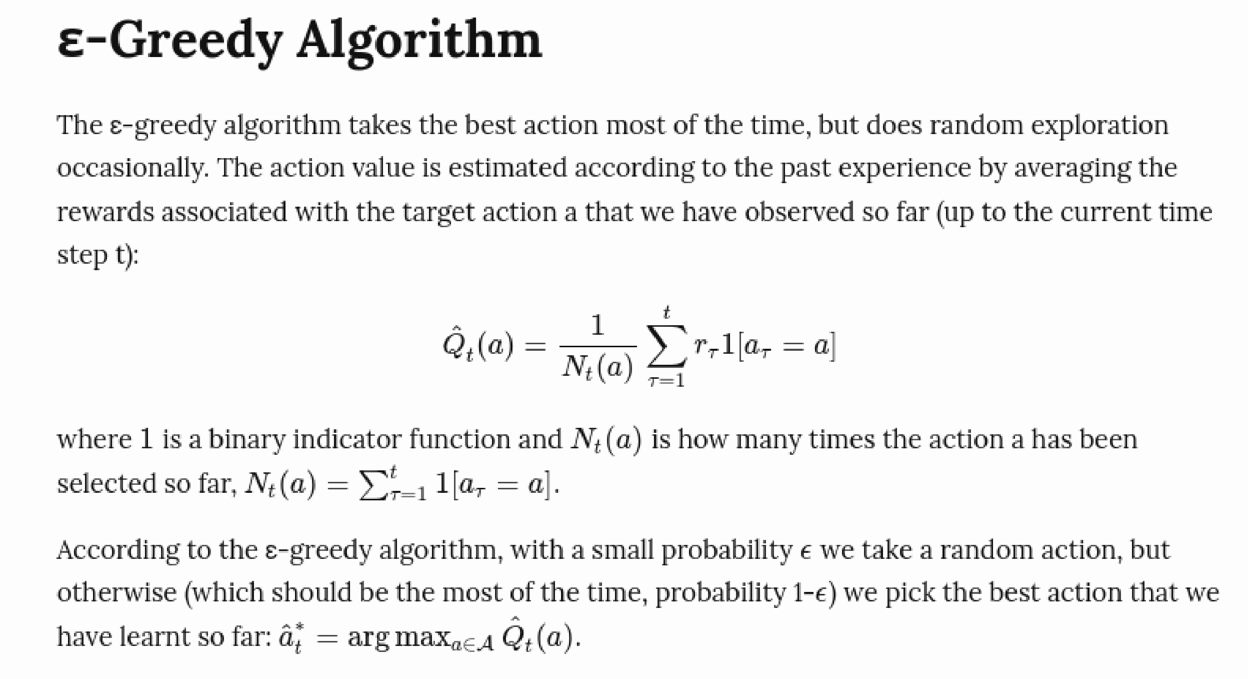
- We begin by looking more closely at methods for estimating the values of actions and for using the estimates to make action selection decisions, which we collectively call **action-value** methods.
- The true value of an action is the mean reward when that action is selected. One natural way to estimate this is by averaging the rewards actually received



$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}},$$

- The Epsilon-Greedy algorithm balances exploitation and exploration fairly basically.
- It takes a parameter, epsilon, between 0 and 1, as the probability of exploring the options (called arms in multi-armed bandit discussions) as opposed to exploiting the current best variant in the test.
- For example, say epsilon is set at 0.1.
- Every time a visitor comes to the website being tested, a number between 0 and 1 is randomly drawn. If that number is greater than 0.1, then that visitor will be shown whichever variant (at first, version A) is performing best.
- If that random number is less than 0.1, then a random arm out of all available options will be chosen and provided to the visitor.
- The visitor's reaction will be recorded (a click or no click, a win or lose, etc.) and the success rate of that arm will be updated accordingly.
- Low values of epsilon correspond to less exploration and more exploitation, therefore it takes the algorithm longer to discover which is the best arm but once found, it exploits it at a higher rate.
- Clearly, choosing the value of epsilon can matter a great deal and is not trivial. Ideally, you would want a high value (high exploration) when the number of trials is low, but would transition to a low value (high exploitation) once learning is complete and the best arm is known. There is a technique called

annealing.



ε-Greedy Algorithm

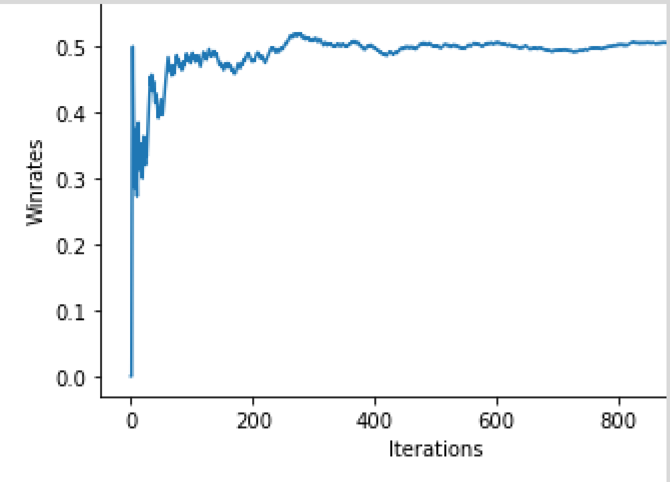
The ε-greedy algorithm takes the best action most of the time, but does random exploration occasionally. The action value is estimated according to the past experience by averaging the rewards associated with the target action a that we have observed so far (up to the current time step t):

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{\tau=1}^{t} r_\tau 1[a_\tau = a]$$

where 1 is a binary indicator function and $N_t(a)$ is how many times the action a has been selected so far, $N_t(a) = \sum_{\tau=1}^{t} 1[a_\tau = a]$.

According to the ε-greedy algorithm, with a small probability ϵ we take a random action, but otherwise (which should be the most of the time, probability 1-ϵ) we pick the best action that we have learnt so far: $\hat{a}_t^* = \arg\max_{a \in \mathcal{A}} \hat{Q}_t(a)$.

- In ε-greedy action selection, for the case of two actions and ε = 0 .5, what is the probability that the greedy action is selected?
- ANS: The probability is 0.75%, which is broken down in the following way:
    - 0.5% probability derived of choosing the greedy action, which is 1-ε 1-0.5 = 0.5
    - 0.25% chance derived of being chosen as the random option = 0.5 x 0,5.
    - So in total 0.5 + 0.25 = 0.75 or 75%
- By implementation we are achieving similar results: (753)



5. OpenAI gym:

OpenAI Gym is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so you can train agents, compare them, or develop new Machine Learning algorithms (Reinforcement Learning).

OpenAI is an artificial intelligence research company, funded in part by Elon Musk. Its stated goal is to promote and develop friendly AIs that will benefit humanity (rather than exterminate it).

Essentially Gym provides the environment where we code the agents using algorithms.

To get started refer the documentation here

Environments used For this project:

- CartPole
- Mountain-Car
- LunarLander
- Self Driving Racing-Car
- Taxi

Basic steps to solve RL problem:
1. Import Dependencies
2. Test Environment
3. Train Model
4. Save Model (optional)
5. Evaluate model
6. Final Test

5.1 CartPole-v0

- Description:

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright, and the goal is to prevent it from falling over by increasing and reducing the cart's Velocity.

- Source:

This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson

Observation:

Type: Box(4)

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-0.418 rad (-24 deg)	0.418 rad (24 deg)
3	Pole Angular Velocity	-Inf	Inf

- Actions:

Type: Discrete(2)

Num	Action
0	Push cart to the left
1	Push cart to the right

Note: The amount the velocity that is reduced or increased is not fixed; it depends on the angle the pole is pointing. This is because the center of gravity of the pole increases the amount of energy needed to move the cart underneath it

- Reward:

Reward is 1 for every step taken, including the termination step

Starting State:

All observations are assigned a uniform random value in [-0.05..0.05]

Episode Termination:

Pole Angle is more than 12 degrees.

Cart Position is more than 2.4 (center of the cart reaches the edge of the display).

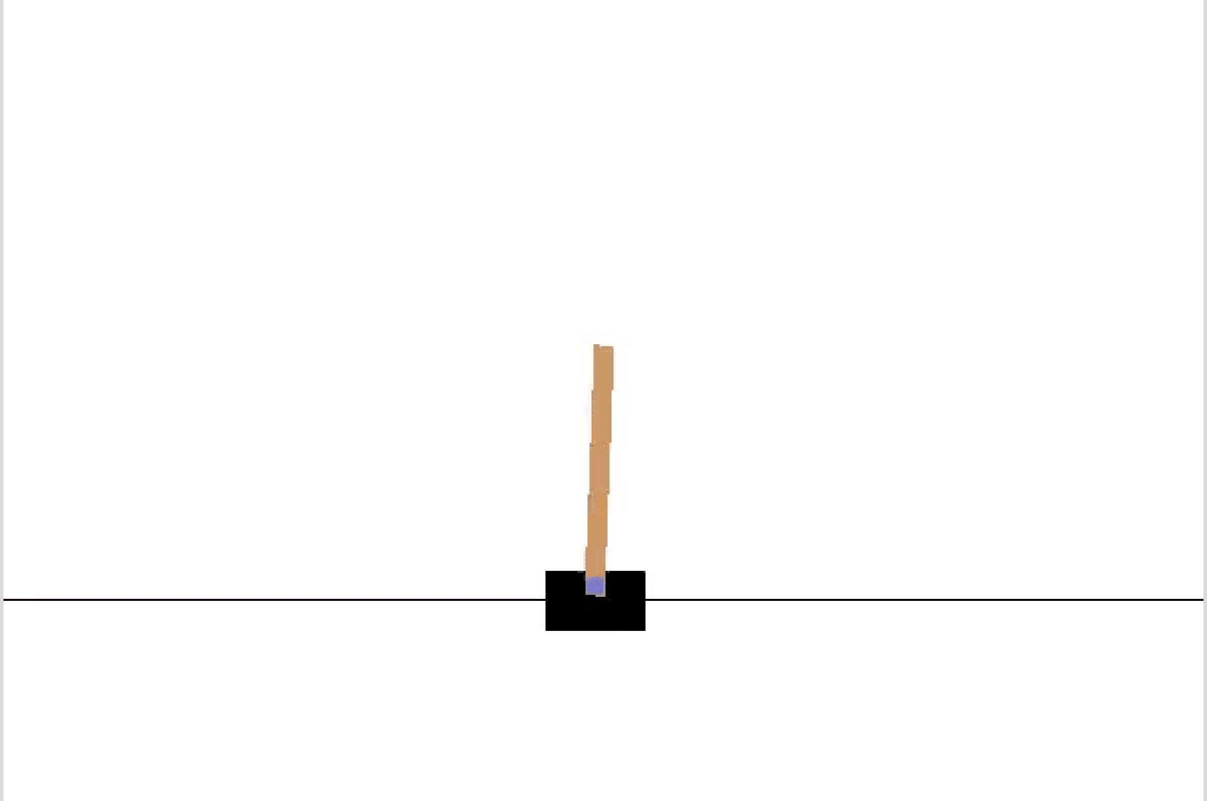
Episode length is greater than 200.

Solved Requirements:

Considered solved when the average return is greater than or equal to 195.0 over 100 consecutive trials.

- Model used : PPO(refer later pages) with mlp policy

- Conclusion:



5.2 MountainCar-v0

- Description:

  The agent (a car) is started at the bottom of a valley. For any given state the agent may choose to accelerate to the left, right or cease any acceleration.
- Source:

  The environment appeared first in Andrew Moore's PhD Thesis (1990).
- Observation:

  Type: Box(2)

Num	Observation	Min	Max
0	Car Position	-1.2	0.6
1	Car Velocity	-0.07	0.07

- Actions:

  Type: Discrete(3)

Num	Action
0	Accelerate to the Left
1	Don't accelerate
2	Accelerate to the Right

  Note: This does not affect the amount of velocity affected by the gravitational pull acting on the car.
- Reward:

  Reward of 0 is awarded if the agent reached the flag (position = 0.5) on top of the mountain.

  Reward of -1 is awarded if the position of the agent is less than 0.5.
- Starting State:

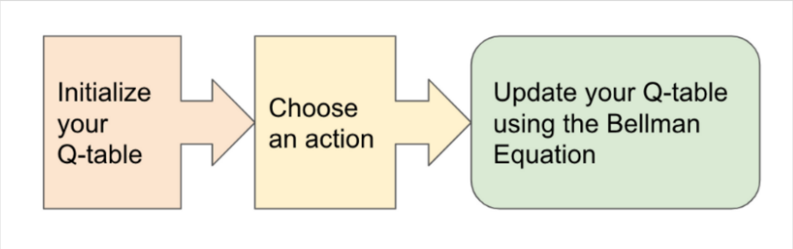
  The position of the car is assigned a uniform random value in [-0.6 , -0.4].

  The starting velocity of the car is always assigned to 0.
- Episode Termination:

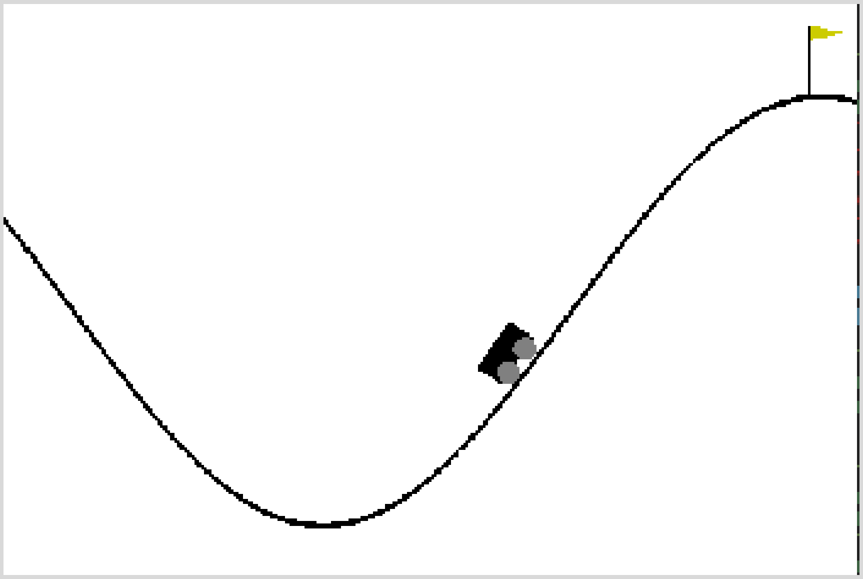
  The car position is more than 0.5

  Episode length is greater than 200

- Model Used : Q-Learning (With a high epsilon value of about 0.9-1.0, the model does its first clear in around 100-200 episodes.)
  - Q-Learning : Q-learning is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward.
  - The 'q' in q-learning stands for quality. Quality in this case represents how useful a given action is in gaining some future reward.
  - When q-learning is performed we create what's called a q-table or matrix that follows the shape of [state, action] and we initialize our values to zero. We then update and store our q-values after an episode. This q-table becomes a reference table for our agent to select the best action based on the q-value.



- Conclusion:



5.3 LunarLander-v2

- Description:

  The Agent(LunarLander) drops from the top of the screen and has to land in between the given flags. For any given state the agent may choose to turn the engine on or off.

- Source:

  Rocket trajectory optimization is a classic topic in Optimal Control.

- Observations:

  The landing pad is always at coordinates (0,0). The coordinates are the first two numbers in the state vector.

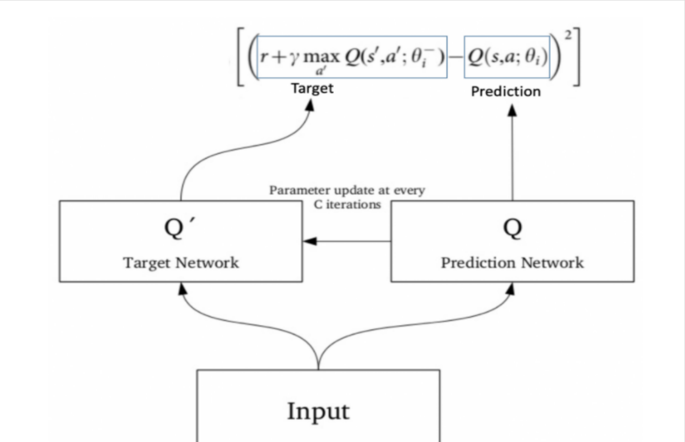
- Reward:

  Reward for moving from the top of the screen to the landing pad and zero speed is about 100..140 points. If the lander moves away from the landing pad it loses reward. The episode finishes if the lander crashes or comes to rest, receiving an additional -100 or +100 points. Each leg with ground contact is +10 points. Firing the main engine is -0.3 points each frame. Firing the side engine is -0.03 points each frame. Solved is 200 points.

- Model Used : Deep Q-Learning network (uses linear layers)
    - Deep Q-Learning : In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output.
    - Steps involved:
        - All the past experience is stored by the user in memory.
        - The next action is determined by the maximum output of the Q-network.
        - The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem. Going back to the Q-value update equation derived from the Bellman equation. we have:

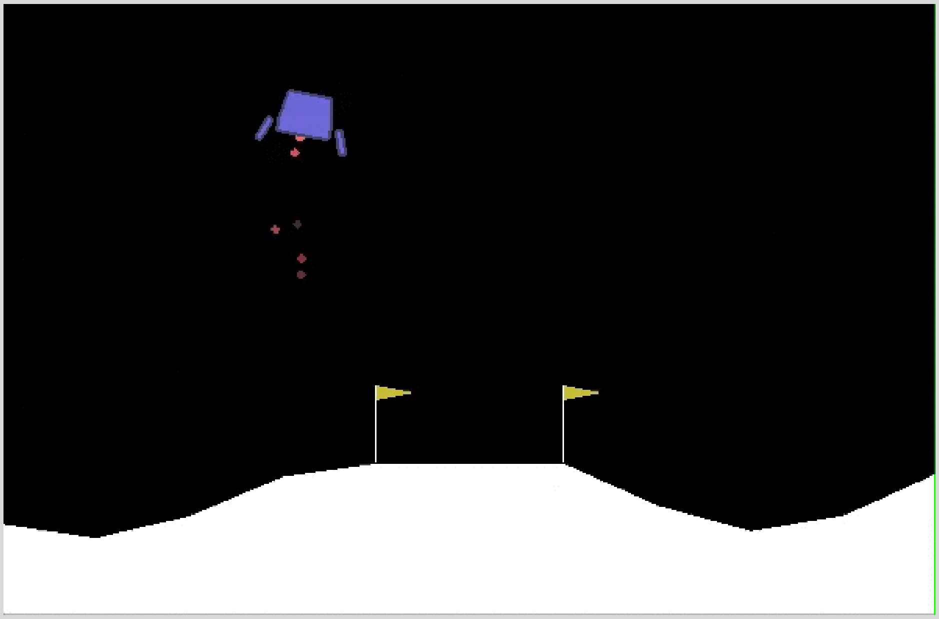
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[\boxed{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)} - Q(S_t, A_t) \right]$$

        - Since the same network is calculating the predicted value and the target value, there could be a lot of divergence between these two. So, instead of using 1one neural network for learning, we can use two.
            - Target Network: Since the same network is calculating the predicted value and the target value, there could be a lot of divergence between these two. So, instead of using 1one neural network for learning, we can use two. We could use a separate network to estimate the target. This target network has the same architecture as the function approximator but with frozen parameters. For every C iterations (a hyperparameter), the parameters from the prediction network are copied to the target network. This leads to more stable training because it keeps the target function fixed (for a while):

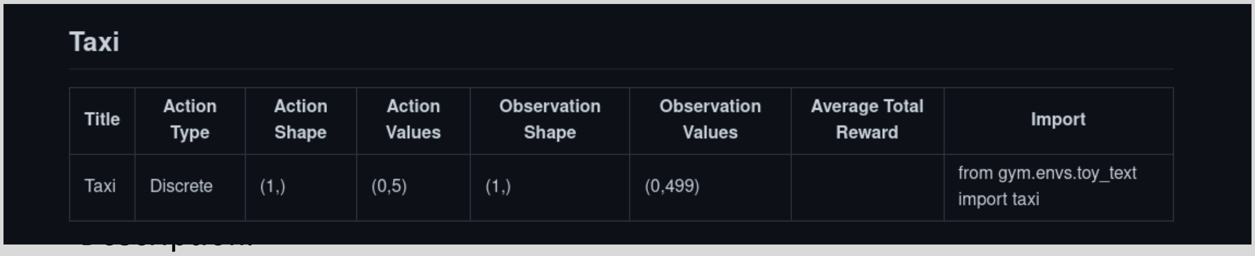

$\left[\left(\boxed{r+\gamma \max_{a'} Q(s',a';\theta_i^-)} - \boxed{Q(s,a;\theta_i)}\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q´
Target Network
Q
Prediction Network
Input

- Experience Replay : Experience Replay is the act of storing and replaying game states (the state, action, reward, next_state) that the RL algorithm is able to learn from. Experience Replay can be used in Off-Policy algorithms to learn in an offline fashion. Off-policy methods are able to update the algorithm's parameters using saved and stored information from previously taken actions. Deep Q-Learning uses Experience Replay to learn in small batches in order to avoid skewing the dataset distribution of different states, actions, rewards, and next_states that the neural network will see. Importantly, the agent doesn't need to train after each step.

- Conclusion:



5.4 Taxi-v3



Taxi

Title	Action Type	Action Shape	Action Values	Observation Shape	Observation Values	Average Total Reward	Import
Taxi	Discrete	(1,)	(0,5)	(1,)	(0,499)		from gym.envs.toy_text import taxi

Description:

There are four designated locations in the grid world indicated by R(ed), G(reen), Y(ellow), and B(lue). When the episode starts, the taxi starts off at a random square and the passenger is at a random location. The taxi drives to the passenger's location, picks up the passenger, drives to the passenger's destination (another one of the four specified locations), and then drops off the passenger. Once the passenger is dropped off, the episode ends.

- MAP:

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

- Actions:

There are 6 discrete deterministic actions:

0: move south
1: move north
2: move east
3: move west
4: pickup passenger
5: drop off passenger

- Observations:

  There are 500 discrete states since there are 25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is in the taxi), and 4 destination locations.

  Note that there are 400 states that can actually be reached during an episode. The missing states correspond to situations in which the passenger is at the same location as their destination, as this typically signals the end of an episode. Four additional states can be observed right after a successful episodes, when both the passenger and the taxi are at the destination. This gives a total of 404 reachable discrete states.

- Passenger locations:

  0: R(ed)
  1: G(reen)
  2: Y(ellow)
  3: B(lue)
  4: in taxi

- Destinations:

  0: R(ed)
  1: G(reen)
  2: Y(ellow)
  3: B(lue)

- Rewards:

  -1 per step reward unless another reward is triggered.
  +20 delivering passengers.
  -10 executing "pickup" and "drop-off" actions illegally.

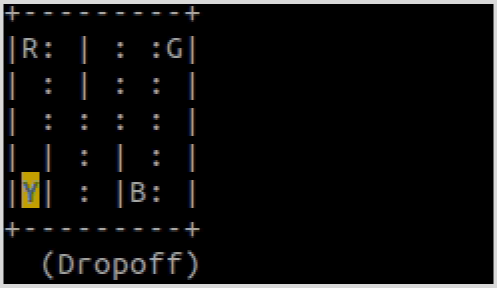
- Rendering:

  blue: passenger
  magenta: destination
  yellow: empty taxi

green: full taxi
other letters (R, G, Y and B): locations for passengers and destinations state space is represented by: (taxi_row, taxi_col, passenger_location, destination)

- Model : Used Q learning to solve the environment

- Conclusion :



```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
```

5.5 CarRacing-v0

- PPDescription:
  Easiest continuous control task to learn from pixels, a top-down racing environment.Discrete control is reasonable in this environment as well, on/off discretization is fine.
  State consists of STATE_W x STATE_H pixels.

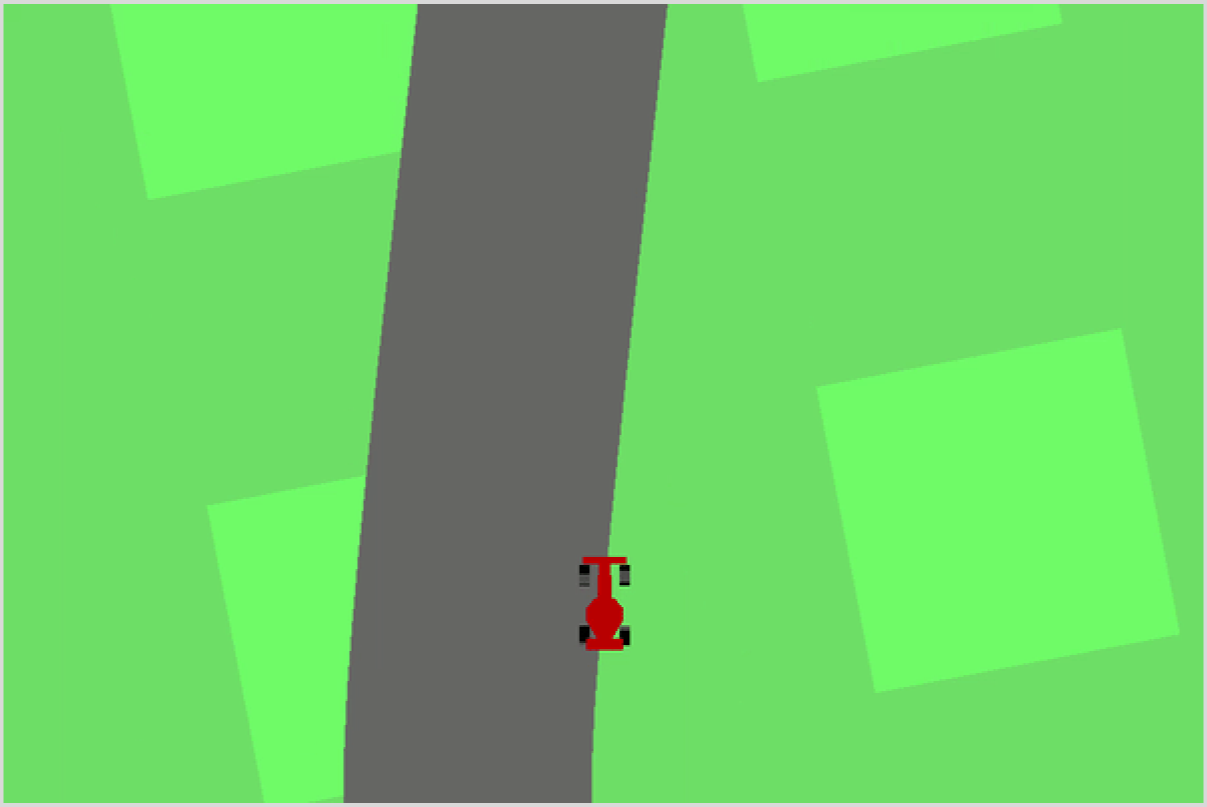
- Rewards:
  The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles visited in the track. For example, if you have finished in 732 frames, your reward is 1000 - 0.1*732 = 926.8 points.
  The game is solved when the agent consistently gets 900+ points. The generated
  the track is random every episode.

- Episode Termination:
  The episode finishes when all the tiles are visited. The car also can go outside of the PLAYFIELD - that is far off the track, then it will get -100 and die.

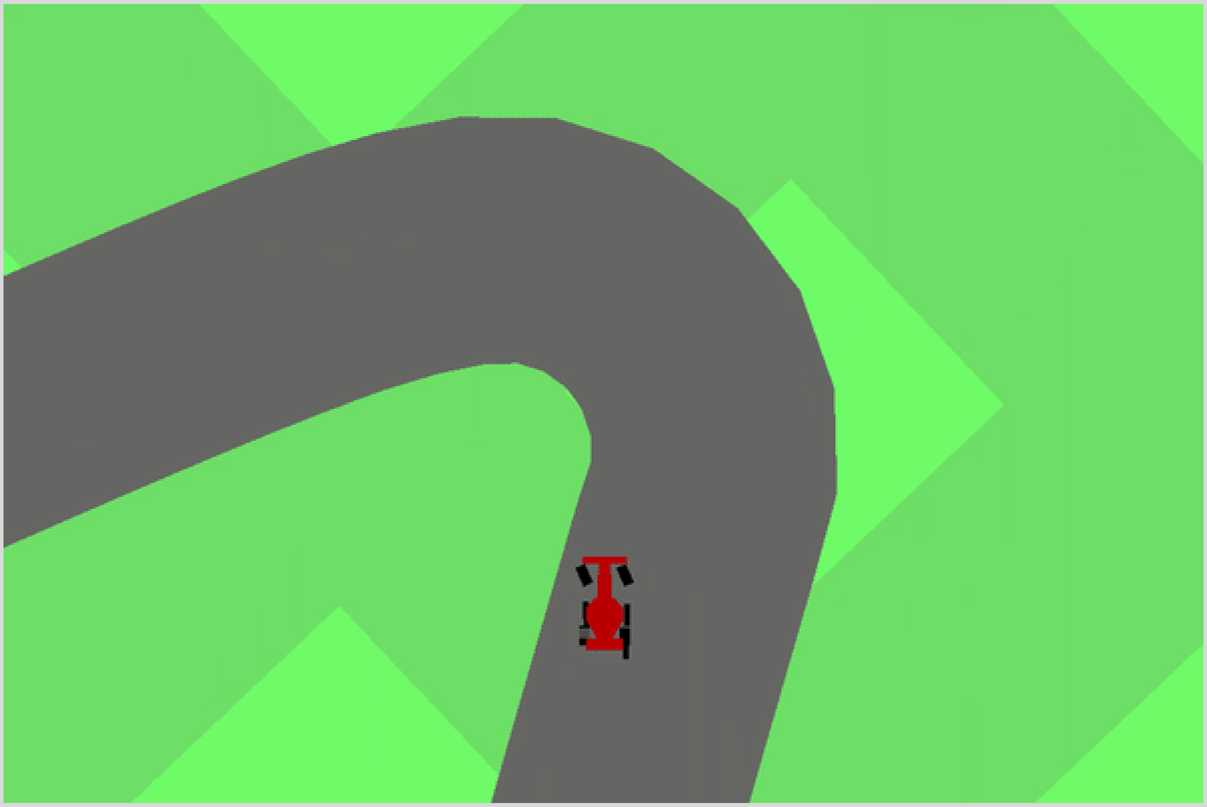
- Note: Some indicators are shown at the bottom of the window along with the state RGB buffer. From left to right: the true speed, four ABS sensors, the steering wheel position and gyroscope. Remember it's a powerful rear-wheel drive car - don't press the accelerator and turn at the same time.

- Model used: PPO

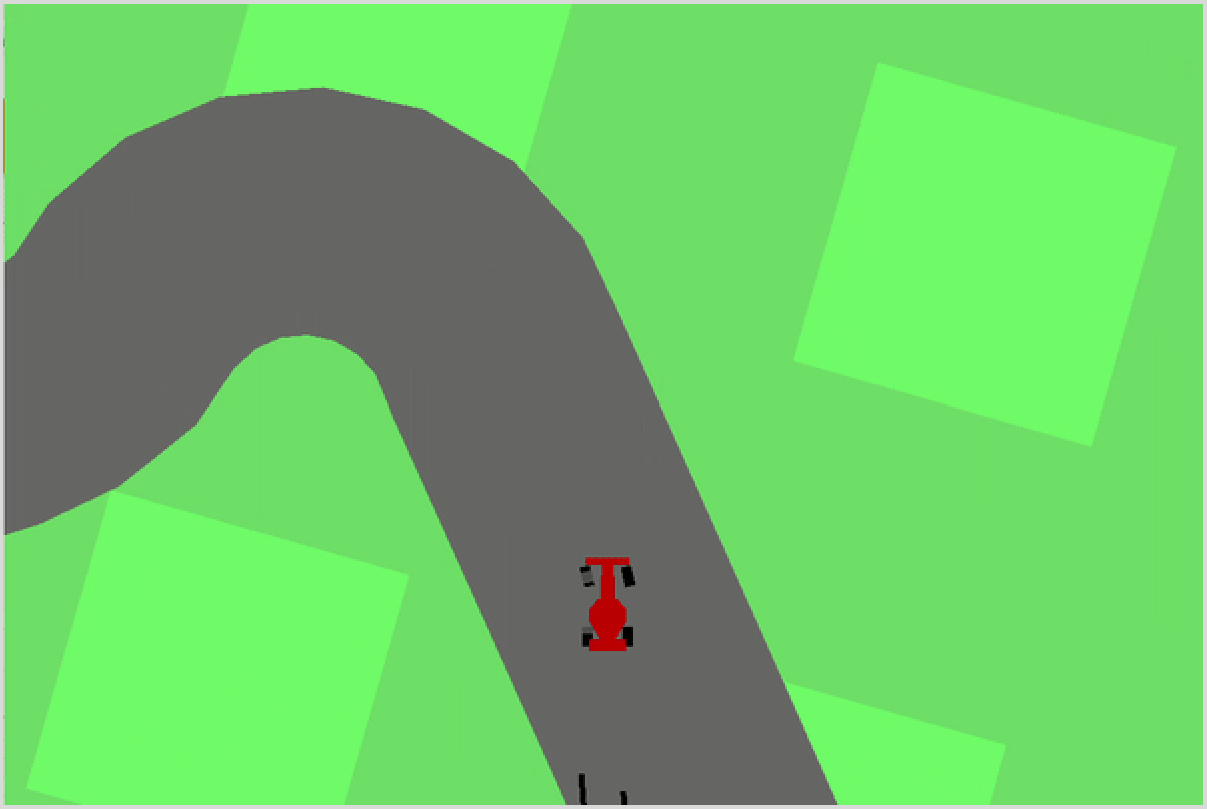
- Conclusion: Some interesting results:
  - Going backwards: Here the agent, after recovering from slipping, returns to the track but starts going in the wrong direction.



- Double slipping: I don’t know how to call this but it is not an easy recovery and an interesting one backwards



- Breaking after recovering: In this clip, the agent avoids going out again by breaking and this time taking the curve slowly.



In the end A reward of around 500-700 out of 900 is achieved

6. Future Aspects:

- Creating a custom environment from scratch.
- Coding Agents using various Models such as Deep Q-Learning, PPO, etc to train the custom environment and compare the results.
- Creating Agents using Deep Q-Learning to train more advanced environments.

References:

- RL Github Links:

  https://github.com/diegoalejogm/Reinforcement-Learning

  https://github.com/dennybritz/reinforcement-learning

  https://github.com/lilianweng/multi-armed-bandit

  https://github.com/bgalbraith/bandits

  https://github.com/alison-carrera/mabalgs

  https://github.com/akhadangi/Multi-armed-Bandits https://github.com/SahanaRamnath/MultiArmedBandit_RL

https://github.com/ali92hm/multi-armed-bandit

- RL StackOverflow links:

  What is ML
  https://stackoverflow.com/questions/2620343/what-is-machine-learning/2620583#2620583

  Policy in RL
  https://stackoverflow.com/questions/46260775/what-is-a-policy-in-reinforcement-learning

  Value iteration vs policy iteration
  https://stackoverflow.com/questions/37370015/what-is-the-difference-between-value-iteration-and-policy-iteration/42493295#42493295

  https://stackoverflow.com/questions/37370015/what-is-the-difference-between-value-iteration-and-policy-iteration

  Good implementations of RL
  https://stackoverflow.com/questions/740389/good-implementations-of-reinforcement-learning

  When to use a certain RL algorithm
  https://stackoverflow.com/questions/22723830/when-to-use-a-certain-reinforcement-learning-algorithm

  Q learning and SARSA
  https://stackoverflow.com/questions/6848828/what-is-the-difference-between-q-learning-and-sarsa