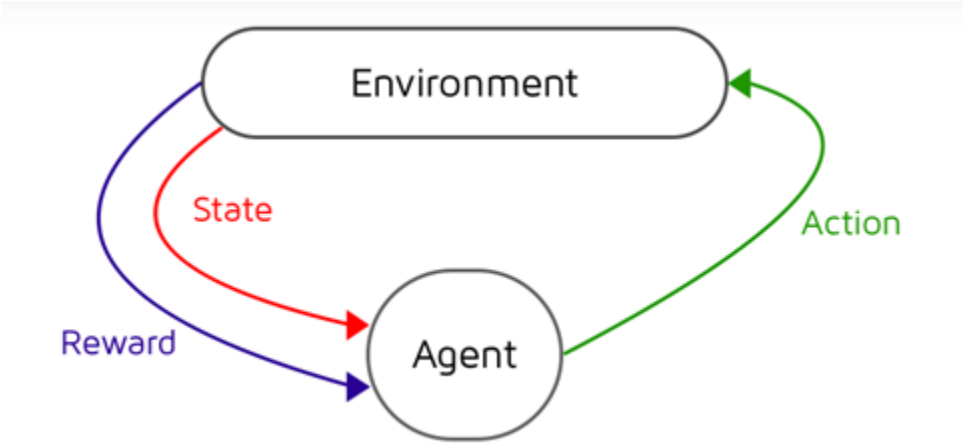
Reinforcement Learning - 1

Kaustab Pal
kaustab21@gmail.com

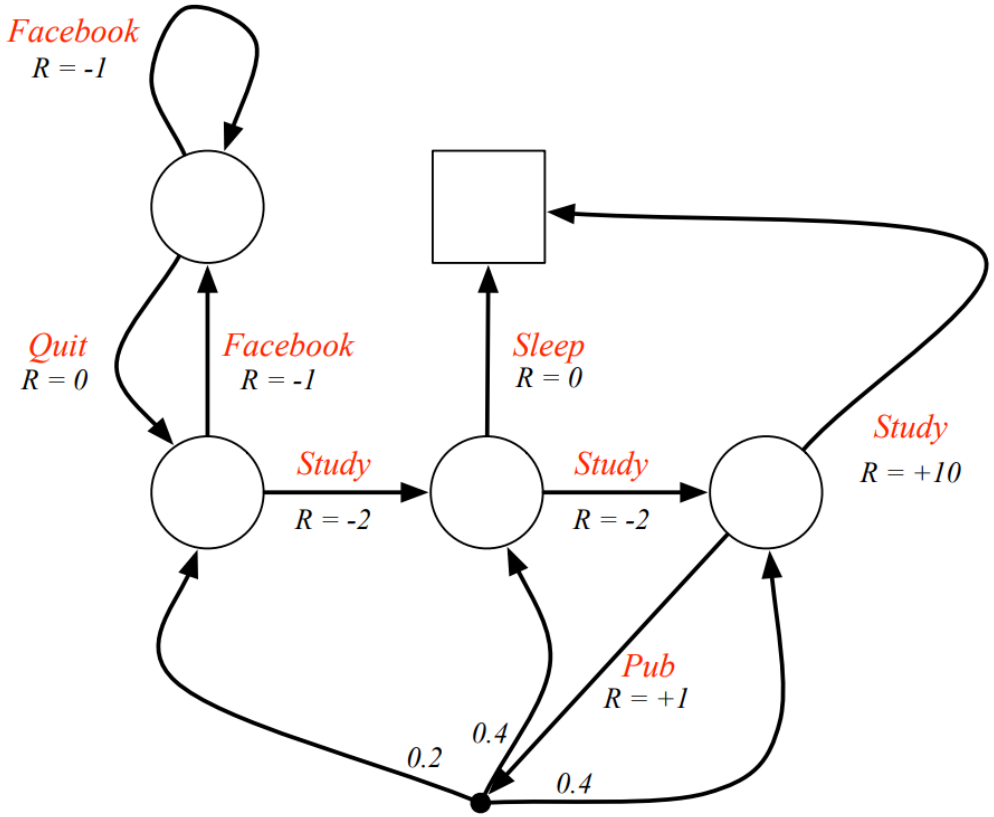
What is Reinforcement Learning?

- RL is that branch of ML where an agent learns how to reach it's goal by interacting with the environment and learning what are the good actions and what are the bad actions.
- Every time the agent interacts with the environment, the state of the environment changes and the environment gives a reward to the agent.
- The agent's job is to take actions that maximizes the cumulative rewards in the long run.



Markov Decision Process

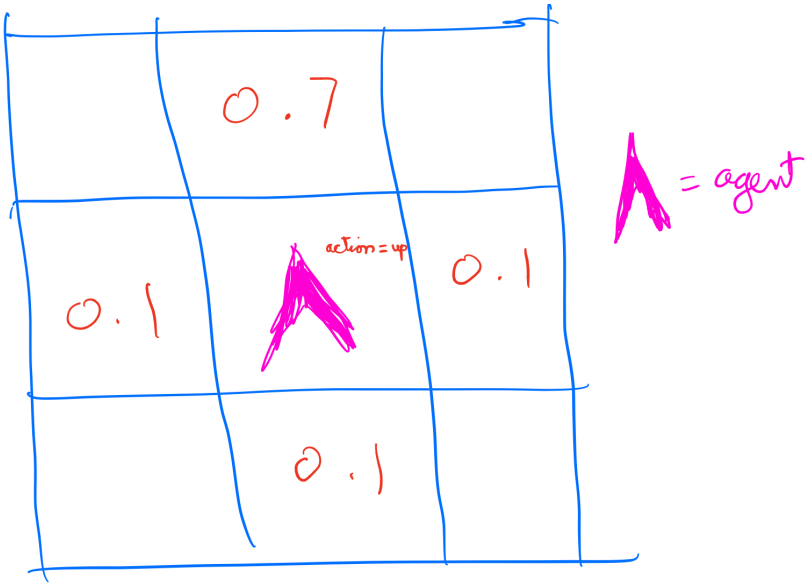
- All RL agents can be defined as a Markov Decision Process.
- At each time step the agent is in state s and chooses an action a. Depending on the action the environment moves to a new state s_{t+1} and gives the agent a reward.
- In a MDP, all states have the markov property, i.e. The future state s_{t+1} depends on the present state S and not the past states.
- An MDP can be defined as a
    - Set of states S
    - Set of Actions A
    - Transition function P(s' | s,a)
    - Reward function R(s, a, s')
    - Start state s_o
    - Discount rate γ
    - Horizon H


Facebook
R = -1
Quit
R = 0
Facebook
R = -1
Sleep
R = 0
Study
R = +10
Study
R = -2
Study
R = -2
Pub
R = +1
0.2
0.4
0.4

source

Transition function P(s' | s,a)

- The transition function gives the probability of going to state s' if we are at state s and take action a.



Reward Function

- Whenever the state of the environment changes due to an agent's action, the environment gives a reward (r_t) to the agent.
- A reward function tells us if we are in state s and take action a, what is the expected reward we will get.
- It is given by:

$$R(s, a) = E[r_t \mid s, a]$$

State Value Function V(s)

- While the reward we get from being in a state tell us how good the state is immediately, value functions of a state tells us how good the state is in the long run.
- The value of a state is the total amount of reward an agent can expect to collect over the future starting from that state.
- The goal of an RL agent is to take actions that takes us to higher valued states.

$$V(s) = E[R_t + \gamma V(s_{t+1}) \mid s_t = s]$$

Action Value Function Q(s,a)

- Given the current state and an action, the action value function tells us how good the action is in the long run.
- An RL agent must only take those actions that have high action values. High valued actions will land us in high valued states.

$$Q(s,a) = E[R_t + \gamma Q(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a]$$

Policy

- A policy is a mapping between states and actions.
- It is a function that takes the current state as input and outputs what actions the agent must take when it is in that state.
- Policies can be deterministic, i.e. given the current state the policy outputs only one action. It can also be stochastic, i.e. given the current state the policy outputs a probability distribution of actions.

Deterministic Policy: $\pi(s) = a$ or $p(a \mid s) = \begin{cases} 1 & \text{if a will be taken} \\ 0 & \text{if a will not be taken} \end{cases}$

Stochastic Policy: $\pi(s) = a$ or $p(a \mid s) =$ some value between 0 and 1

- RL algos that use a policy are called policy based algorithms.

Model of the Environment

- The model of the environment mimics the behaviour of the environment.
- Given the present state and action, the model of the environment will predict the next state of the environment and the reward that will be generated by the environment.
- Not all reinforcement learning algorithms depend on the model of the environment.
- RL algos that depend on the model of the environment are called model based RL.
- RL algos that does not depend on the model of the environment are called model free RL.

Value Iteration Methods

- Value based algorithms that gives us the optimal value functions.
- The optimal value functions can be either **state value function** or **action value function**.

Optimal state value function V*(s)

- The optimal value of a state can be written as:

$$V^*(s) = max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V^*(s')$$

- We have to find the optimal state value function for all the states.

How to find V*(s) for all states

- We iteratively update V*(s) from t=0 to H.
- At timestep t, $V^*_t(s)$ is given by:

$$V_t^*(s) = max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{t-1}^*(s')$$

- Algorithm:

Start with $V^*_o(s) = 0$ for all states.

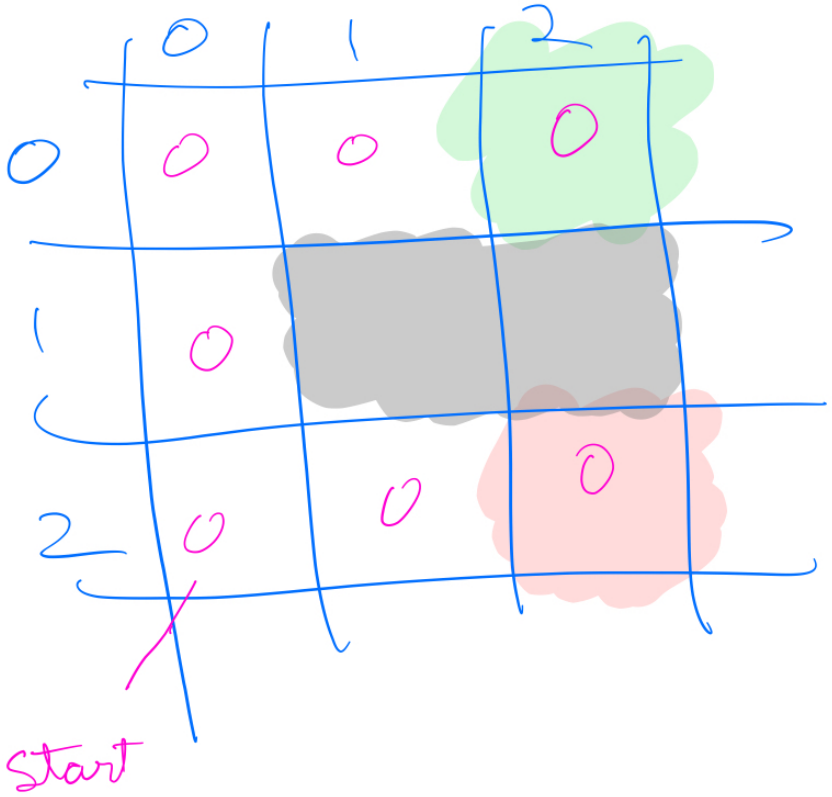
For t = 1, ... , H:

For all states s in S:

$$V_t^*(s) = max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{t-1}^*(s')$$

$$\pi_t^*(s) = argmax_a V_t^*(s)$$

At $t = 0$



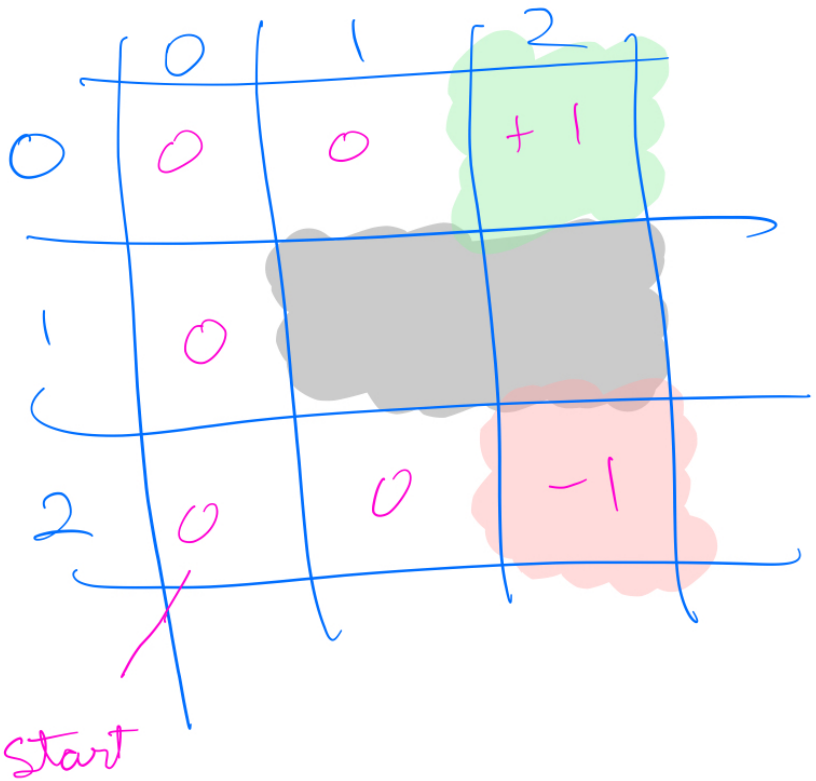
= +1

= -1

$\gamma = 0.9$, $P(\text{green} \mid s) = 0.9$

$V_0(s) = 0$

At $t = 1$



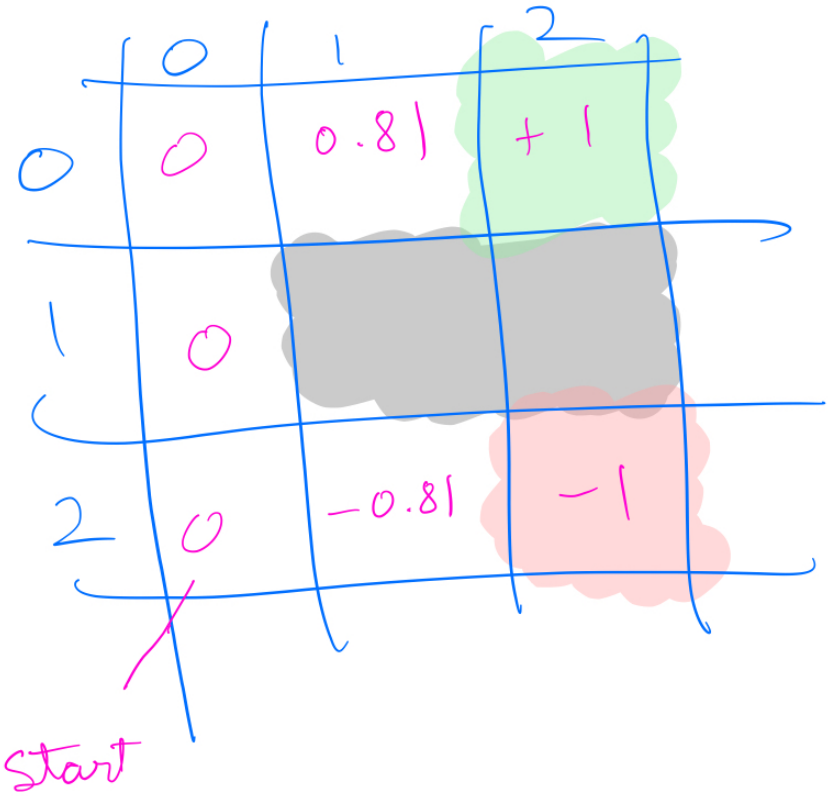
= +1

= −1

$\gamma = 0.9, \quad P(\text{green} \mid s) = 0.9$

$V_1(s) = r + \gamma \sum_{s'} P_{s'} V_0(s')$

At $t = 2$



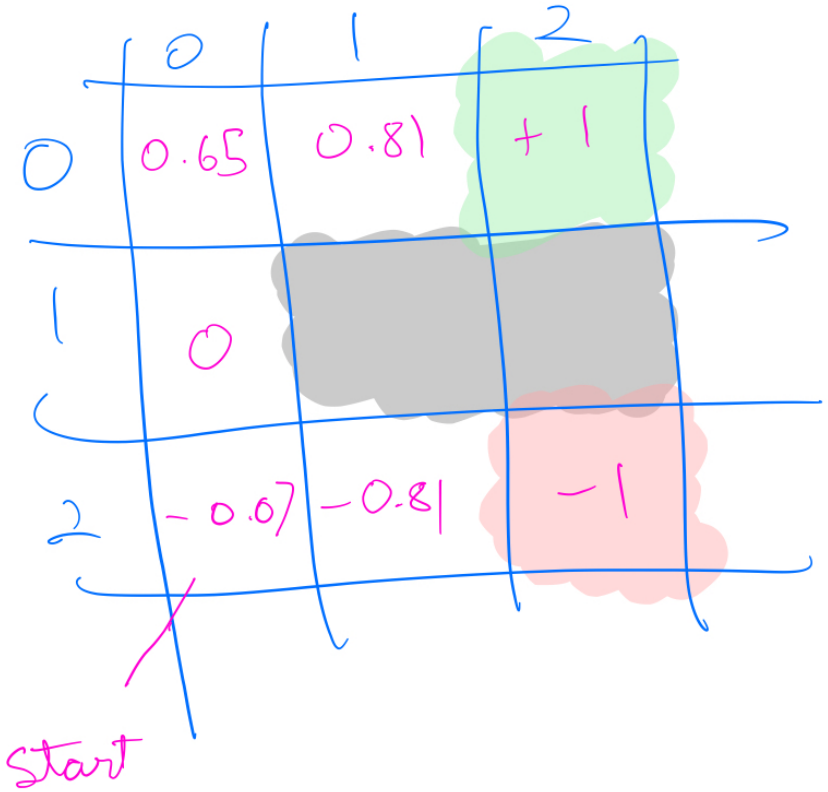
= +1

= -1

$\gamma = 0.9, \quad P(\text{green} \mid s) = 0.9$

$$V_2(s) = r + \gamma \sum_{s'} P_{s'} V_1(s')$$

At $t = 3$



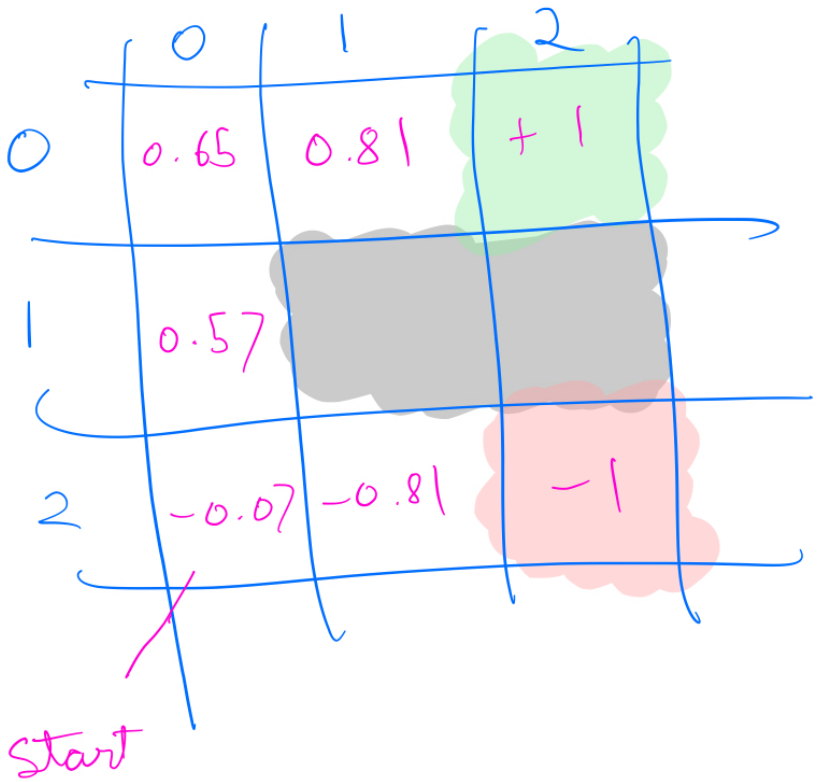
= +1

= -1

$\gamma = 0.9, \quad P(\text{●} \mid s) = 0.9$

$$V_3(s) = r + \gamma \sum_{s'} P_{s'} V_2(s')$$

At $t = 4$



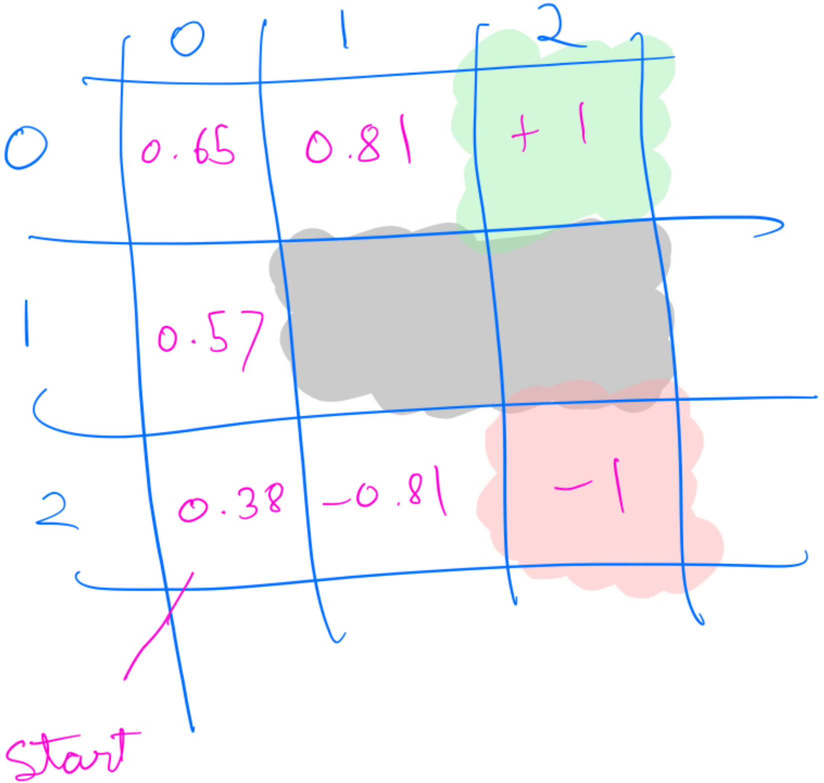
= +1

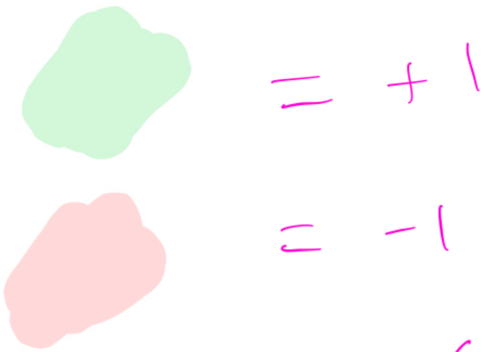
= -1

$\gamma = 0.9, \quad P(\text{green} \mid s) = 0.9$

$$V_4(s) = r + \gamma \sum_{s'} P_{s'} V_3(s')$$

At $t = 5$





$\gamma = 0.9, \quad P(\ \ | s) = 0.9$

$$V_5(s) = r + \gamma \sum_{s'} P_{s'} V_5(s')$$

Optimal action value function Q*(s,a)

- The optimal value of an action can be written as:

$$Q^*(s,a) = \sum_{s'} p(s' \mid s,a) R(s,a,s') + \gamma max_a Q^*(s',a')$$

- We have to find the optimal action value function for all the actions possible in every states.

How to find Q*(s,a) for all states

- We iteratively update Q*(s,a) from t=0 to H.
- At timestep t, $Q^*_t(s,a)$ is given by:

$$Q^*_t(s,a) = \sum_{s'} p(s' \mid s,a) R(s,a,s') + \gamma max_{a'} Q^*_{t-1}(s',a')$$

- Algorithm:

Start with $Q^*_o(s,a) = 0$ for all states.

For t = 1, ... , H:

For all states s in S:

$$Q^*_t(s,a) = \sum_{s'} p(s' \mid s,a) R(s,a,s') + \gamma max_{a'} Q^*_{t-1}(s',a')$$

We can use Q-value iteration instead of V-value iteration because if our Q-values converge, it will also tell us the optimal policy and therefore we don’t have to calculate it ourselves.

Limitations of value based methods

- We need access to the dynamics model of the environment.
- For large state and action spaces these methods becomes hugely inefficient.

Tabular Q-Learning

- Sampling based approximations are used so that we aren't dependent on the dynamics of the model.
- We take an action a and consider the state s' where we land.
- We calculate the target Q-value for reaching state s' from state s by taking action a as: $target(s') = R(s, a, s') + \gamma max_{a'} Q_k(s', a')$
- The new Q value Q_{k+1} thus becomes:
  $Q_{k+1}(s, a) = (1 - \alpha) Q_k(s, a) + \alpha(target(s'))$

Exploration vs exploitation

- How do we choose actions? 🤷
- We can choose actions randomly.
- We can choose action greedily from the Q table.

ϵ-greedy approach

- Here we choose a random action with probability ϵ and a greedy action with probability (1 - ϵ)
- At the start of the training the value of ϵ is high, i.e we want to explore more but as the q table converges ϵ Is lowered so that we choose more of the greedy actions and less of the random actions.

Limitations of tabular Q learning

- As the number of states keeps on increasing tabular Q learning fails to scale.
- To solve for this we use DQN.

DQN

- In DQN we use a neural network to approximate the Q-values.
- Similar states will give us similar Q-values.
- Here instead of updating the Q-table we update the parameters of the neural net by performing gradient descent on $(target_j - Q(\phi_j, a_j, \theta))^2$

Problems of using NN

- The target of the loss function is always changing.
- If we have data for simultaneous actions in the training set, the neural net will take that into account and generalize amongst them.

Old parameters for target Q values

- To solve the problem of a constantly changing target value we use two sets of parameters, the main parameters which are constantly updated and the target value parameters which are an old set of parameters. The target parameters are synced with the updated parameters after certain number of iterations.

Experience Replay

- For training a neural net the training data needs to be independent and identically distributed.
- We store an agent’s experience(state, action, reward) in an experience buffer.
- After a lot of experiences have been stored in the experience buffer we randomly sample a mini batch of experiences and use that to update the main parameters.

DQN Algorithm

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D

    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

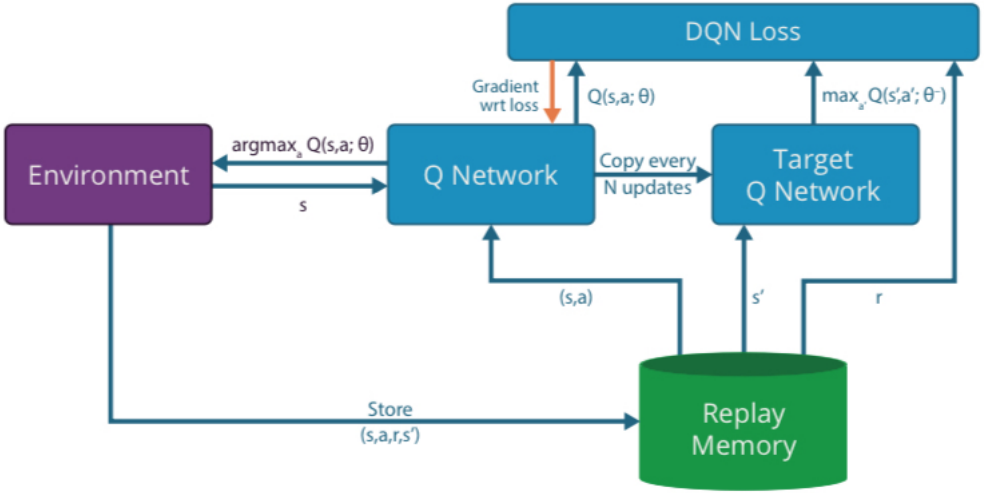


Figure 1. The DQN algorithm is composed of three main components, the Q-network $(Q(s, a; \theta))$ that defines the behavior policy, the target Q-network $(Q(s, a; \theta^-))$ that is used to generate target Q values for the DQN loss term and the replay memory that the agent uses to sample random transitions for training the Q-network.

GORILA architecture for distributed DQN

- Solves single agent problem more efficiently by exploiting parallel computation.
- Consists of parallel actors to generate new behaviors.
- Parallel learners to learn from stored experience.
- A distributed neural network for the value function.
- A central experience replay buffer.

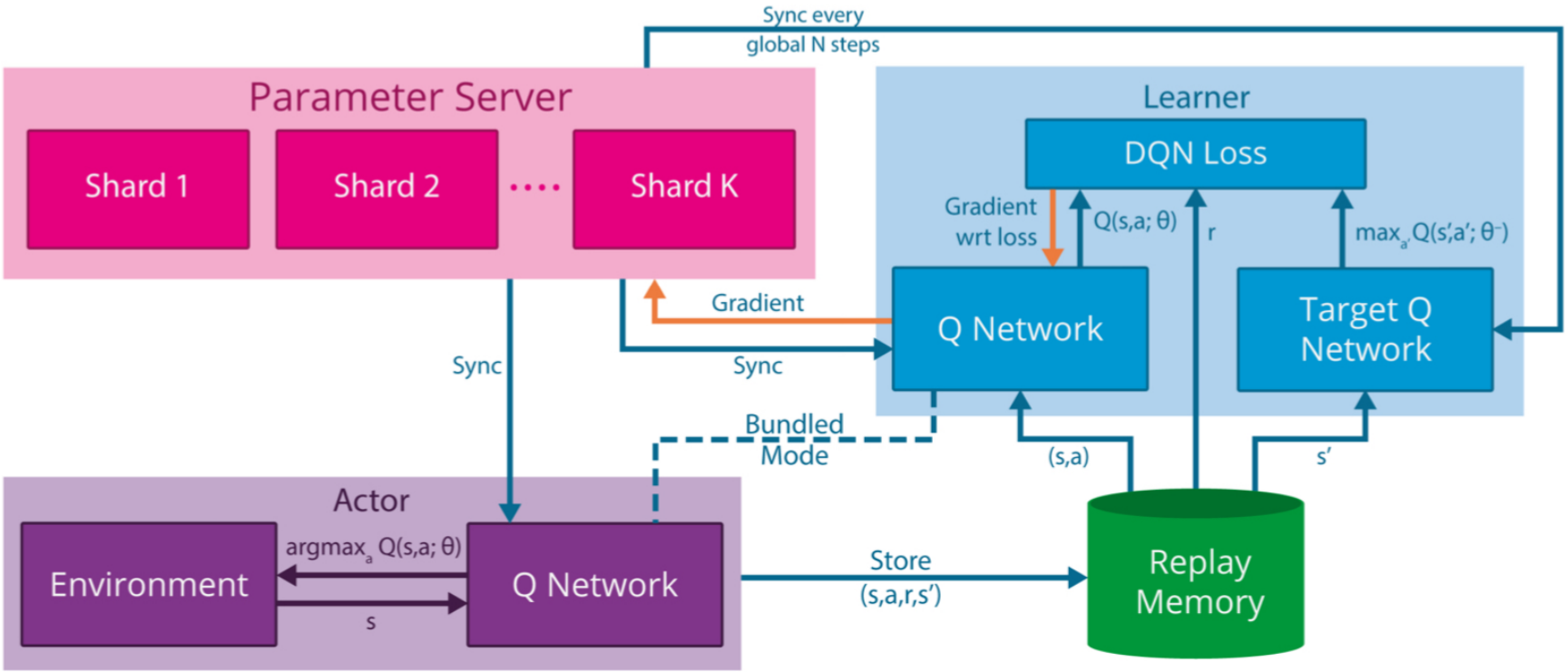


Figure 2. The Gorila agent parallelises the training procedure by separating out learners, actors and parameter server. In a single experiment, several learner processes exist and they continuously send the gradients to parameter server and receive updated parameters. At the same time, independent actors can also in parallel accumulate experience and update their Q-networks from the parameter server.

Resources

- DQN paper: https://arxiv.org/abs/1312.5602 and https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf
- GORILA paper: https://arxiv.org/pdf/1507.04296.pdf
- Deep RL bootcamp: https://sites.google.com/view/deep-rl-bootcamp/lectures
- Meha Kaushik: https://youtu.be/4RfotZC_Ulk
- Phaniteja S: https://sphanit.github.io/publications

Thank you 😇