Introduction to Reinforcement Learning

Content

- What is Reinforcement learning?
- Markov Process
- Q Learning
- DQN

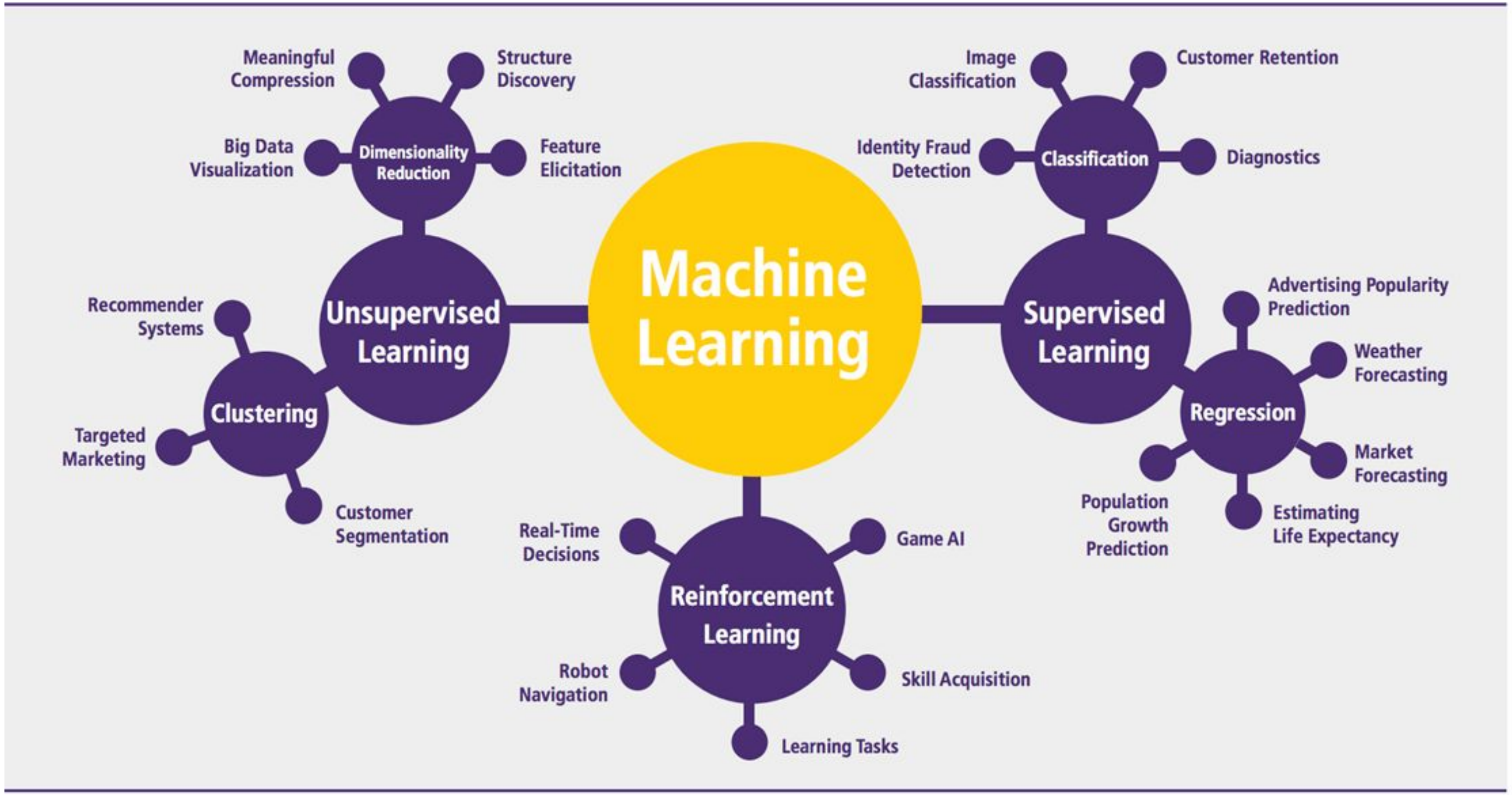
What is Reinforcement Learning?

Reinforcement Learning(RL) is a subfield of machine learning that focuses on the problem of **self-driven** learning of **optimal decisions** over time.

We will dive more deeply into this definition,but before that let’s see how RL differs from other well-known ML problems.

1. **Supervised Learning**
2. **Unsupervised Learning**

The 3 Machine Learning paradigms



Supervised Learning problem

Model is trained on **labelled** datasets to predict output.

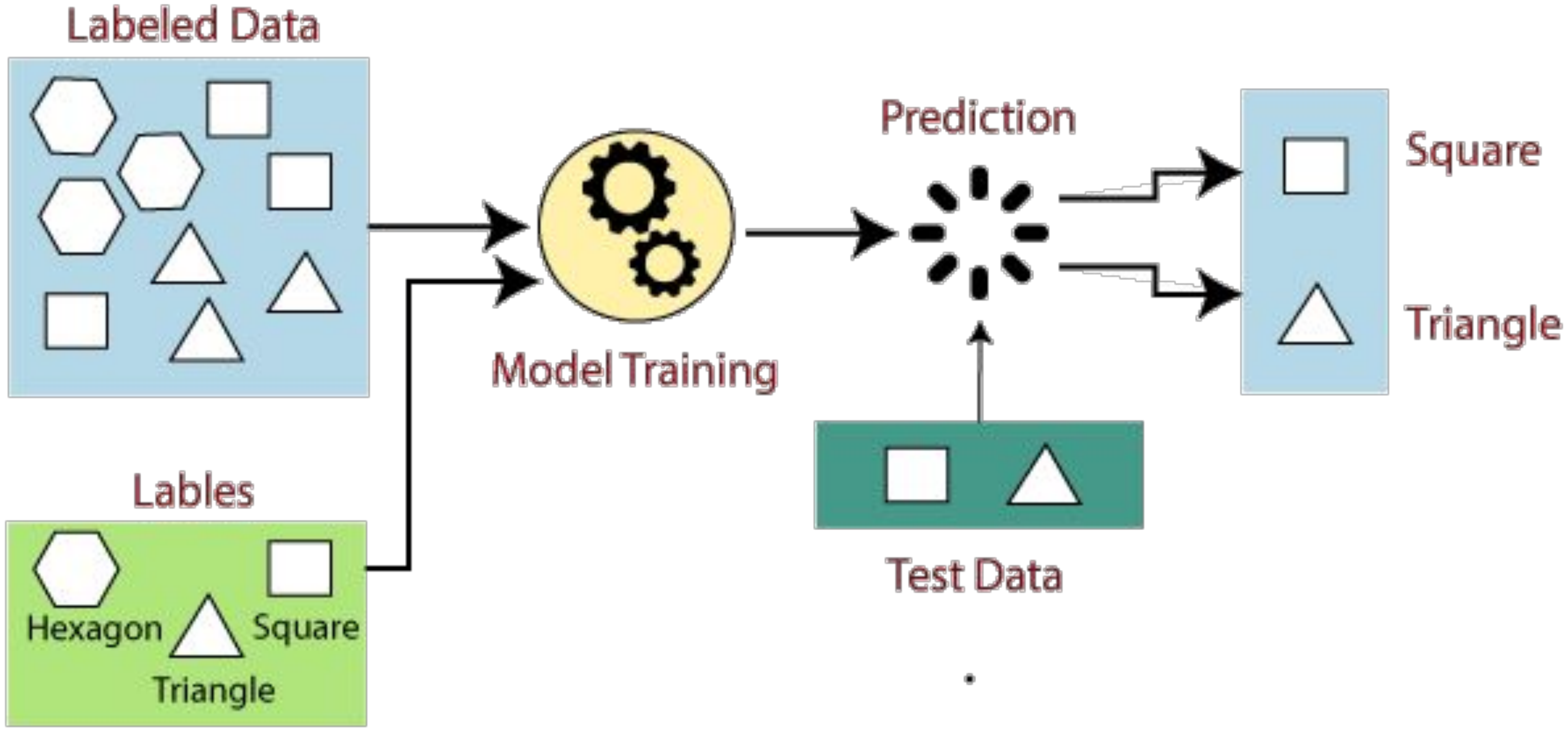
Here the name “supervised” suggests the learning is happening with the help of **ground truth** data. Some very common examples include:

Image classification: Is this image a cat or dog?

Text classification: Is this email spam or not?

Regression based: Given the weather history, how will it be tomorrow?

Supervised Learning problem



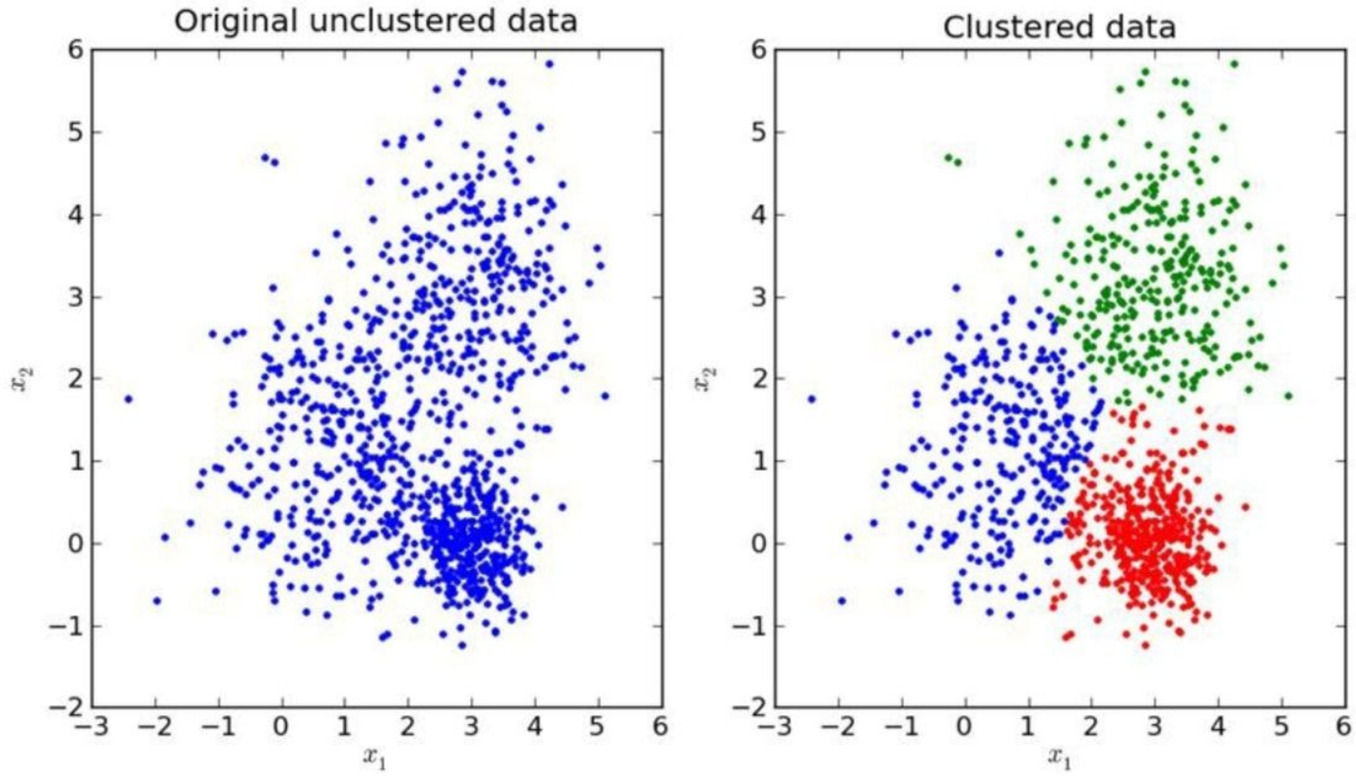
Unsupervised Learning problem

Model is trained on **unlabelled** datasets and it tries to find some **similarities** in dataset.

Some common examples include:

1. **Customer Segmentation:** Understanding different customer groups
2. **GANs:** Generate realistic photographs
3. **Recommender Systems:** Grouping users with similar viewing patterns

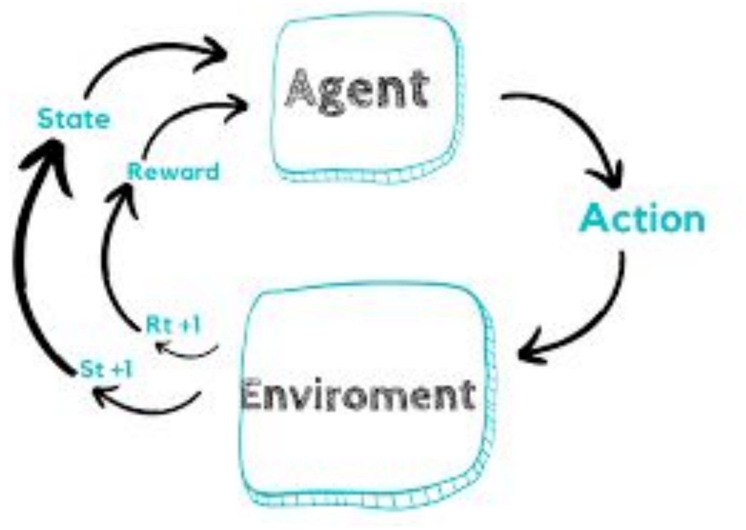
Unsupervised Learning problem



Back to Reinforcement Learning

RL deals with agents performing actions in an environment to maximise a quantity called reward.

The agent gets to know the environment better with time by performing actions and analysing the reward.



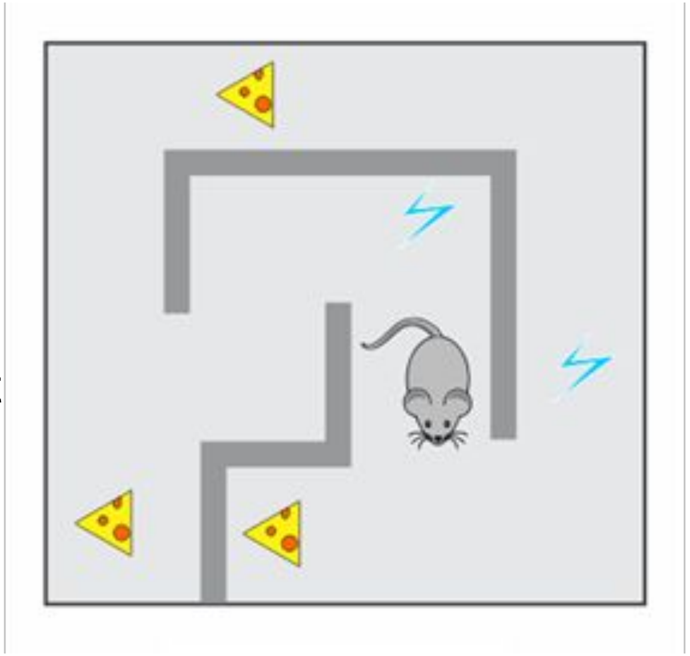
Robot Mouse picture of RL

Let us get a better picture of RL through an example.

Consider an **imaginary maze**, which is the world to a robot mouse. The maze has **food** for the mouse at selected places.

But the problem is, there are places which give an **electric shock** to the mouse(It doesn’t die from it,just a mild one !).

The goal of its life is to find **as much food as possible** while avoiding electric shocks as much as possible.



Here the **environment** will be the maze with food and electricity at some places.

The robot mouse will be our **agent** and it can take the following **actions**

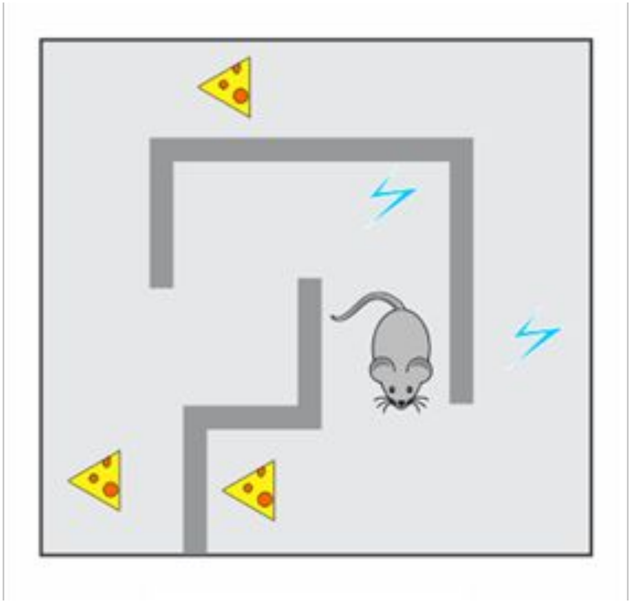
1. turn left
2. turn right
3. move forward

Reward the mouse gets: food(**+ve**) or electric shock(**-ve**)

Goal: **To get as high total reward as possible.**

The main focus here is, no one tells the agent what it sees is good or bad.

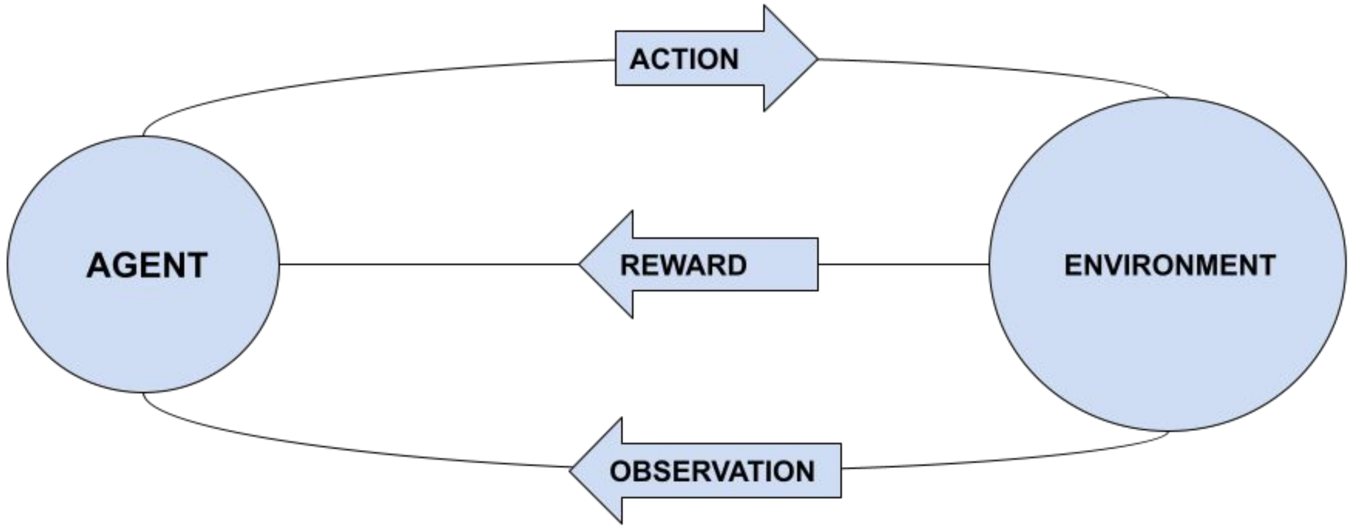
The concept of reward makes sure the **agent learns by itself with time.** It would eventually realise that facing a slight shock to get plenty of food is much better than being still and gaining nothing.



RL formalisms

To formally define what we learnt before,

A RL problem has two major entities - the **agent** and **environment,** and stuff connecting these two - **actions, reward, observations**



RL formalisms

Agent: Anything that interacts with the environment by doing some action based on observations.

Environment: Everything other than the agent comes under this. It’s the platform on which agent reacts.

Reward: It is some scalar value that is given to the agent based on what it does.It can be +ve/-ve, large or small.

Observations: It is the information that the agent can take as input regarding the environment.

State: It hosts the complete information about the environment, not all of which can be accessed or known to the agent. Observations would be a subset of state.

Action: Something that an agent can do in an environment. It could be discrete or continuous. An action may cause a change of state.

The Markov Process

To call a system a Markov Process, it must satisfy the **Markov property**.

Markov property - The state of a system in the future **depends only on the current state.** The past history of states play no role.

Let’s understand this through a simple example.

Consider recording the weather of each day. The sequence of states could be:

[*sunny,sunny,rainy,rainy,sunny,rainy,....*]. (called a **Markov chain**)

Here our **state space** = *{sunny,rainy}.*

So according to MP, if it is sunny today, the **probability of rain tomorrow is same irrespective of no.of sunny/rainy days in the past.**

The Markov Process

We can capture the state transition probabilities in a **transition matrix.**

(The value at index (i,j) represents the probability of transition from state i to state j.)

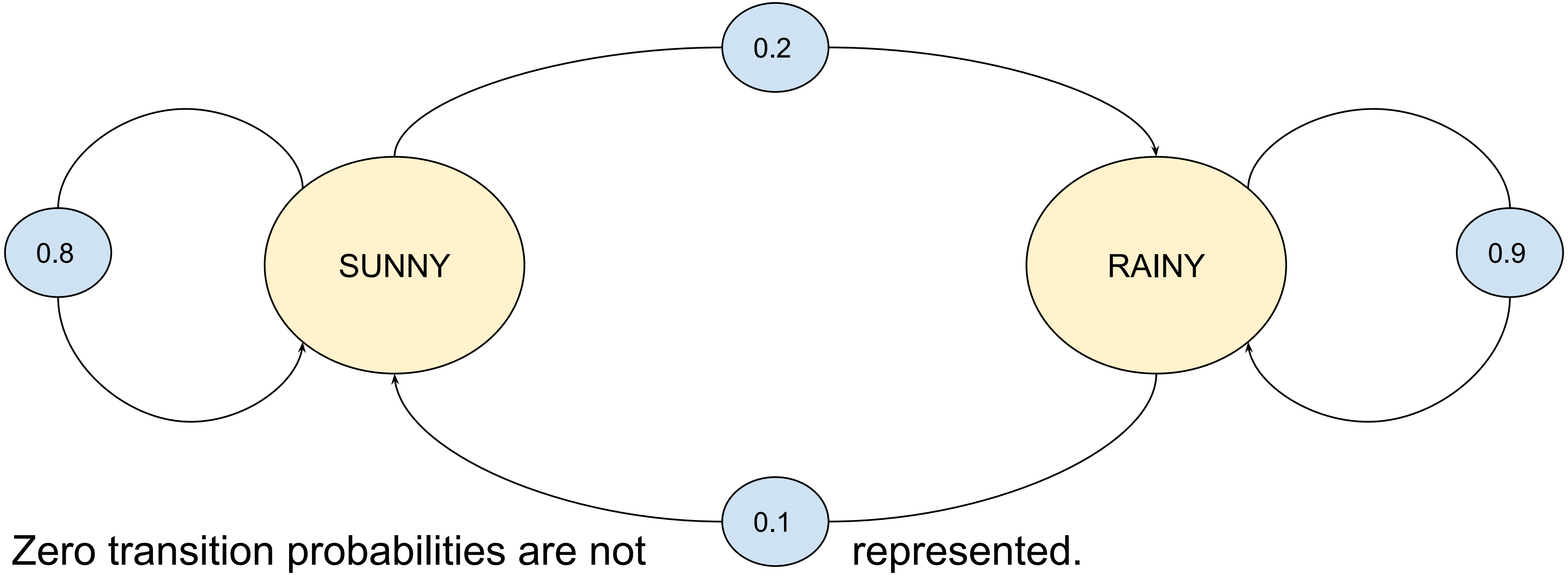
	Sunny	Rainy
Sunny	0.8	0.2
Rainy	0.1	0.9

So a set of states and a transition matrix can fully define a Markov process.

We can also introduce complexity by increasing our state space.

The Markov Process

Another nice way to fully describe a Markov process is through a **state transition graph.**



Zero transition probabilities are not represented.

Return

Return is defined as the total reward obtained for each time step.

$\mathbf{G_t = R_t + R_{t+1} + \ldots + R_T}$

T is the final time step.

But for unending tasks T tends to infinity thus G_t may diverge hence we use discounted return which is defined as

$\mathbf{G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots}$

Where $0 \leq \gamma \leq 1$ is known as the **discount rate.**

γ gives intuition about **how far into the future's rewards are we concerned** .

γ =0 says only the immediate reward matters, γ =1 says immediate and future rewards equally matter.

Value of a State

Value of a state ‘s’ would be the **expectation of the return** for that state. Mathematically,

$$V(s)=E[G|s]$$

It can be interpreted as the **average return** obtained from all the Markov chains possible from a given state.

Let’s get a better picture of what we learnt until now through an example.

An example

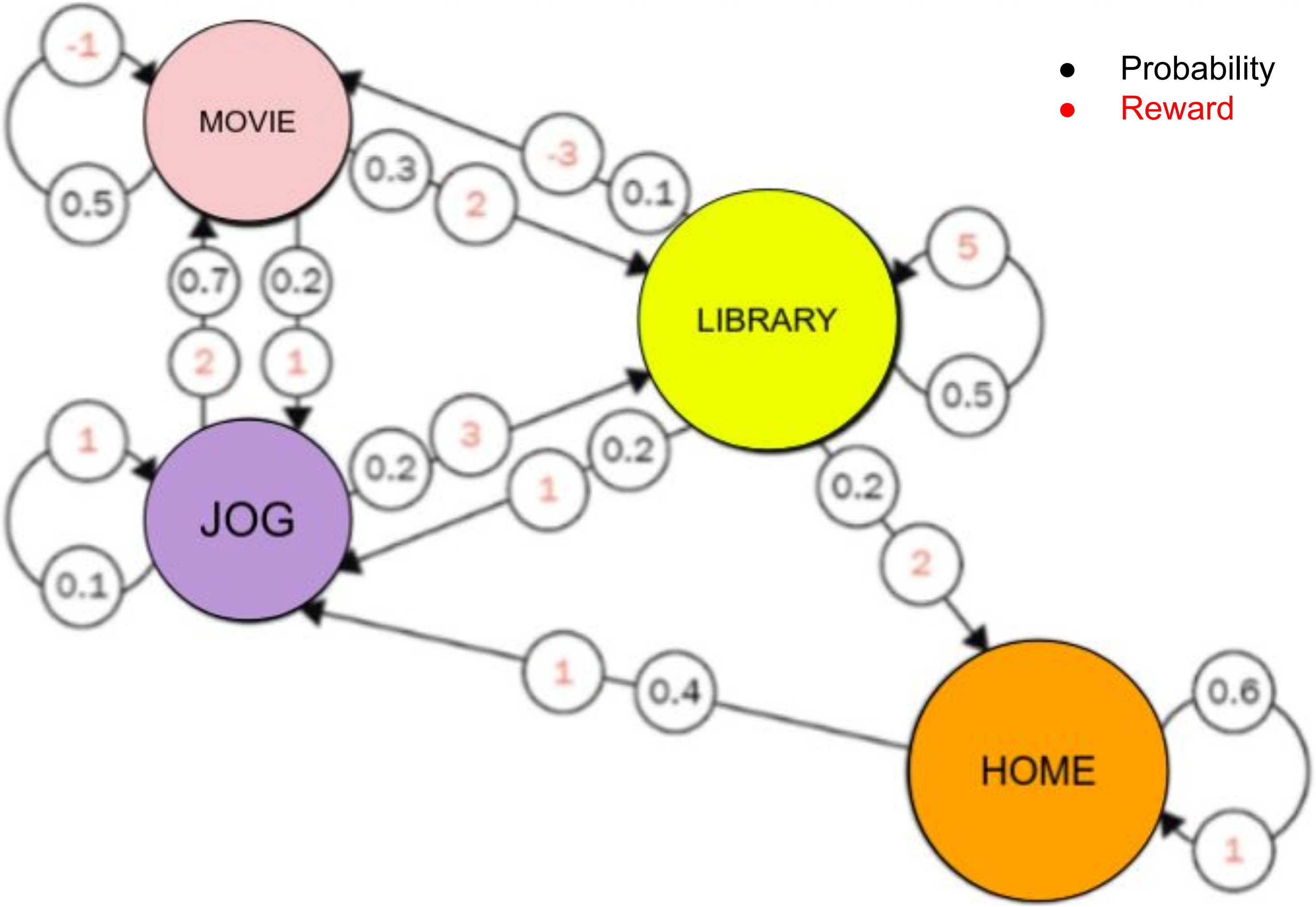
Consider a person whose state space is **{*Home,Movie,Jogging,Library*}.**

One possible Markov chain could be:

Home $\longrightarrow$ Jog $\longrightarrow$ Jog $\longrightarrow$ Movie $\longrightarrow$ Movie $\longrightarrow$ Library $\longrightarrow$ Jog

Let’s check the state transition graph, then find the value of each state.

(Note that Home $\rightleftarrows$ Movie has zero transition probability.)



- ● Probability
- ● Reward

Example contd.

Let’s calculate the value of each state when **γ =0**

V(Movie) = -1*0.5 + 2*0.3 + 1*0.2 = 0.3

V(Jog) = 2*0.7 + 1*0.1 + 3*0.2 = 2.1

V(Home) = 1*0.6 + 1*0.4 = 1.0

V(Library) = 5*0.5 + (-3)*0.1 + 1*0.2 + 2*0.2 = 2.8

Clearly Library is the most valued state. Least being Movie.

What about the case when γ =1 ?

Policy

A policy is what governs the agent's behaviour and the action it takes in each state. It is basically a mapping of states to actions and is defined as:

π: S -> A = {(s_1, a_1), (s_2, a_2), … }

Different policies can give a different value of each state.

Choosing a good and ideal policy is the goal of RL.

Now we will look at how a good policy can be computed, using **Q-Learning**

Q Learning

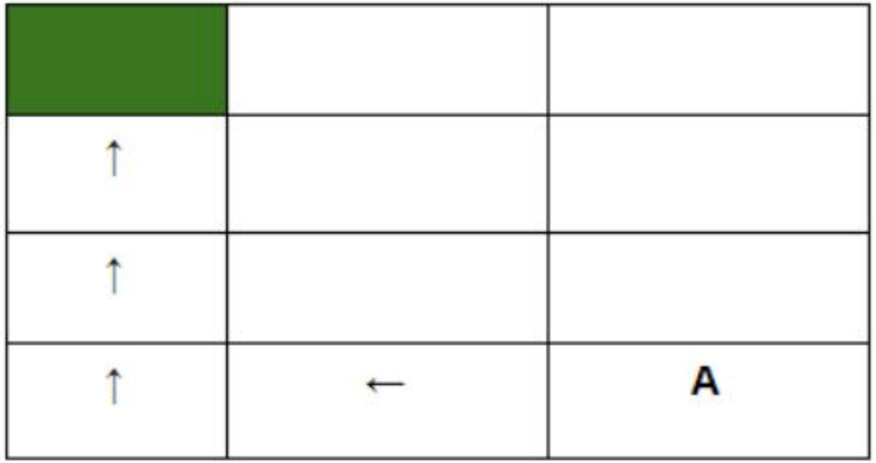
Problem Statement

Throughout this section we will be using the following environment to understand the concepts involving value function, bellman equation and Q learning

Consider this setup where we have an agent that can be randomly dropped in any of the white squares(rooms) shown

The objective is to reach the green room.

For now consider the agent to start from A.



Optimal Value Function

So our job is now to basically associate some value to each state to let the agent know whether it is desirable to be in that state or not

We already saw about the value function but here we use what is called the optimal value function

Optimal Value of a state is the maximum return possible from that state if the best policy is followed. Higher the optimal value of the state more desirable it is to be in that state.

Optimal Value function is a mapping between states and their optimal value

$$\mathbf{V^*(s) = max_{\pi}(R_t + \gamma R_{t+1} + \ldots)}$$

Bellman Equation

π = mapping of states to actions => $\boldsymbol{\pi}$**: S -> A = {(s$_1$, a$_1$), (s$_2$, a$_2$), … }**

By this policy, at time t, let it take action a.
t+1 : a’
t+2 : a’’
… and so on

$\mathbf{V^*(s) = max_{\pi}(R_t+\gamma R_{t+1}+...)}$ [max value achievable over all policies]

$\mathbf{= max_{a,a',a'',...}(R_t+\gamma R_{t+1}+...)}$ [policy == sequence of actions]

$\mathbf{= max_{a,a',a'',...}(R_t+\gamma(R_{t+1}+\gamma R_{t+2}+...))}$ [γ taken out]

$\mathbf{= max_a (R_t+\gamma max_{a',a'',...}(R_{t+1}+\gamma R_{t+2}+...))}$ [max taken in]

$\mathbf{= max_a (R_t + \gamma V^*(s'))}$ [Using expression of V(s’), recursion]

Therefore $V^*(s) = max_a(R(s,a) + \gamma V^*(s'))$ ***[Bellman Equation for Optimal Policy]***

An example

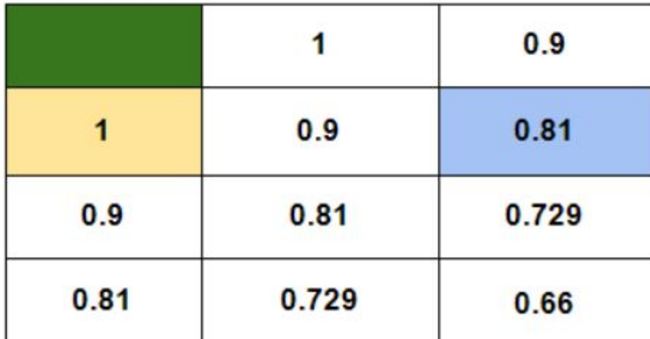
There can be multiple value functions for a given environment. Here we fix the value of the states adjacent to the target state as 1 and calculate the value function. Take γ = 0.9 and compute the value of the state adjacent to the yellow state:

$V^*(s) = R(s,a) + \gamma V^*(s')$

Here we are not giving any rewards for a particular move hence R(s,a) = 0.

Therefore $V^*(s) = \gamma V^*(s') = 0.9*1 = 0.9$.

The values of the rest of the states can be filled this way.



🟩	1	0.9
1	0.9	0.81
0.9	0.81	0.729
0.81	0.729	0.66

Q Learning

Till now what we have done is use Bellman equation to find the value of each state and then use these values to choose an action that leads to a state with a better value.

But this poses a lot of practical difficulty in the sense that in many realistic environments it is difficult to know which action takes you to which state.

Q Learning proposes to overcome this challenge by analysing the ***quality*** of an action instead of the value of the state.

Q Learning

We define quality function Q(s,a) takes in the state and the action as the input and outputs the maximum return possible.

Thus Q(s,a) basically measures the “quality” of an action instead of the value of the state.

Similar to how we derived the Bellman equation for $V^*(s)$, we can write Q(s,a) as,

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a')$$

So now that we quantitatively measure the ‘quality’ of an action, we can create a table with no.rows as the no.states and no.columns as the no.actions and the element in sth row and ath column is Q(s,a).

Once the agent reaches a state s, it can use this table and find the action with highest value for the corresponding state and do that action at that time step.

Temporal Difference (TD)

We can't directly use the quality formula we derived earlier because we assumed a static environment but in reality the environment may be dynamic

Therefore we need to compute Q values for many samples and iteratively improve our estimate.

Temporal Difference is the component using which we iteratively update the Q values.

TD(a,s) = Q_calc - Q(s,a)

Q_calc = R(s,a) + γmax$_{a'}$Q(s',a')

Now we update Q(s,a) as

Q(s,a)* $\leftarrow$ *Q(s,a) + αTD(a,s).

Where $0 \leq \alpha \leq 1$ is known as the **learning rate.**

Exploration vs Exploitation

When we are training our agent, we would want it to explore all possible states and figure out which are the best, and with enough exploration it can build a very accurate Q table

So what is a good exploration strategy? Should we always just choose random actions? The problem with that is you have a very low chance of ever progressing in the game. But we also want to progress in the game, but we want to explore along the way

To do both we use ***epsilon-greedy*** strategy. We fix a base probability to explore which is called epsilon. Suppose epsilon is 0.1, so whenever we have to choose an action, there is a 10% chance of choosing a random action (***exploration***) and an 90% chance of choosing the best action (***exploitation***) as dictated by the Q table.

Q Learning Algorithm

1. Start the training with a Q-table filled with 0s
2. Repeat the following for a fixed number of episodes:

    a. With probability ϵ (epsilon), sample a random action, with 1-ϵ, choose the best action using the Q table

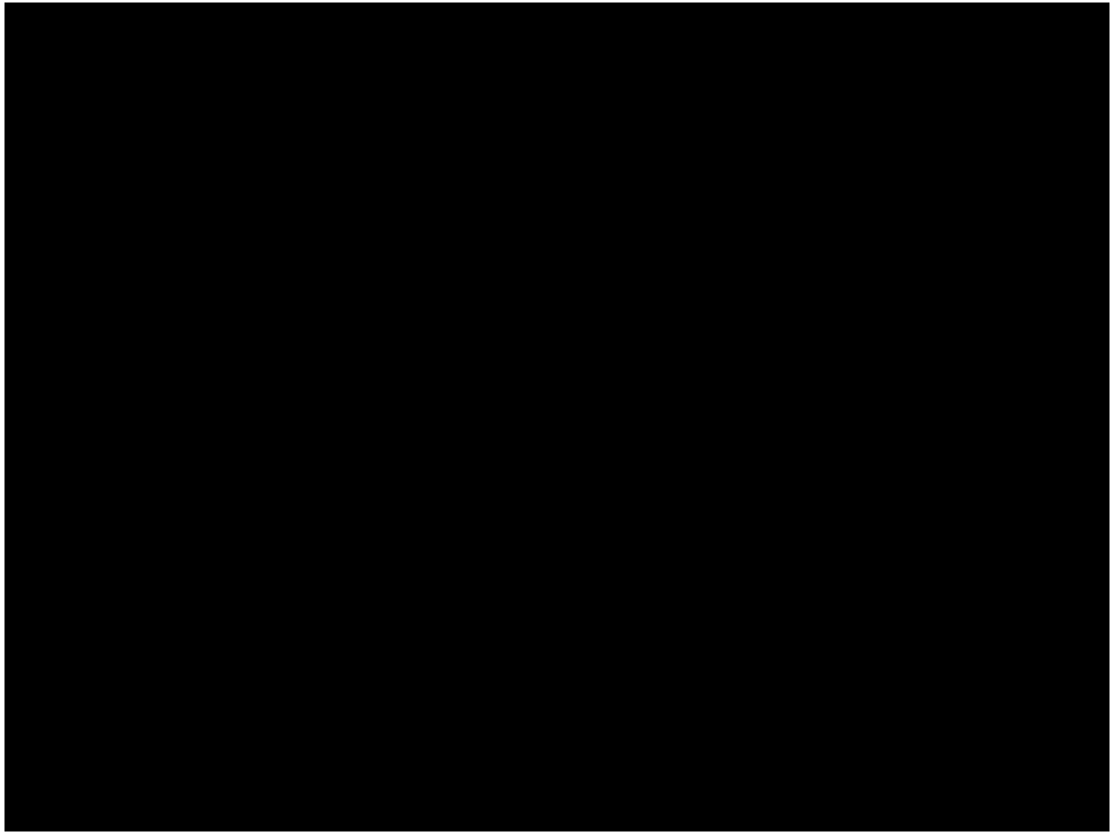
    b. Pass action and state to env, get reward R(s,a) and new state s'

    c. $\boldsymbol{Q\_calc \leftarrow R(s,a) + \gamma \max_{a'} Q(s',a')}$ [Q_calc according to the Bellman Equation]

    d. $\boldsymbol{TD \leftarrow Q\_calc - Q(s,a)}$ [TD between calculated and current Q value]

    e. $\boldsymbol{Q(s,a) \leftarrow Q(s,a) + \alpha TD}$ [Update current Q using TD]

Issues with Q Table

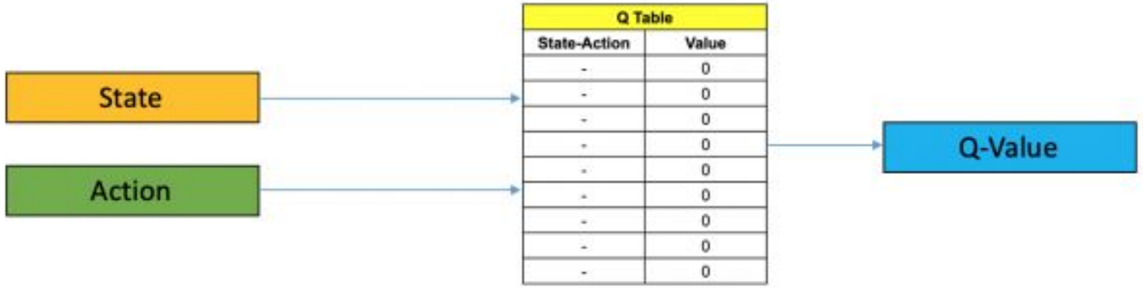


Issues with Q Table

For environment involving continuous state-space or huge number of state keeping track using q table will be infeasible.

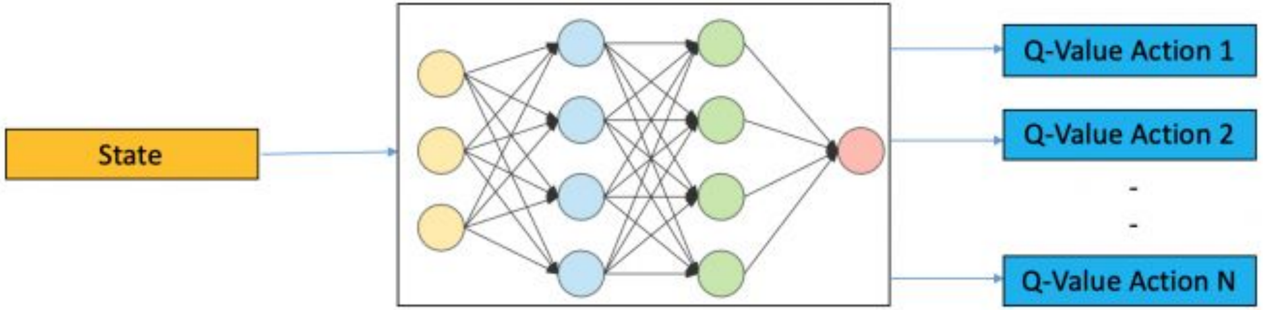
- Memory required to store and update q values(state-space x action-space) will be enormous.
- The amount of time required to explore each state to create the required Q-table would be unrealistic.

Neural Networks to rescue!!!!!!!



Q Table	
State-Action	Value
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

Q Learning



Deep Q Learning

Deep Q Network

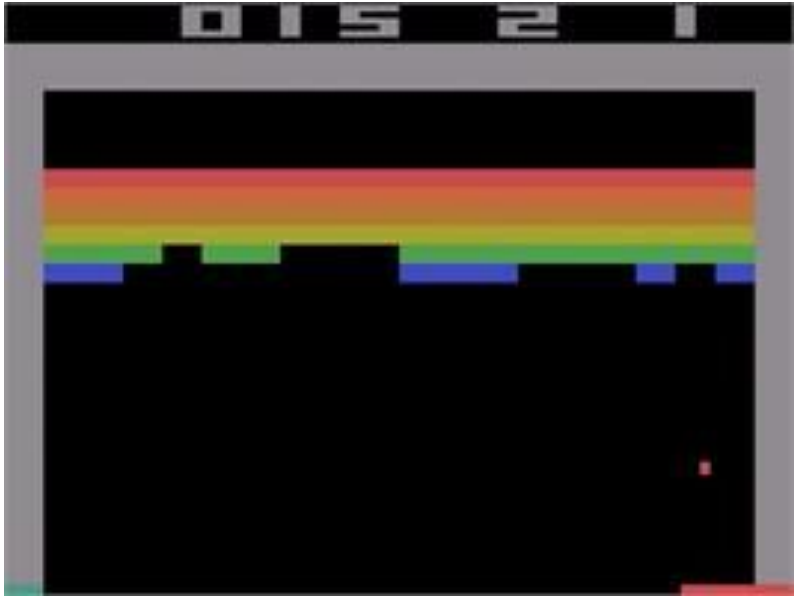
Still we need to solve some problems :

- MDP state for input
- Breaking correlation between input
- Avoid forgetting previous experience
- Chasing a moving target

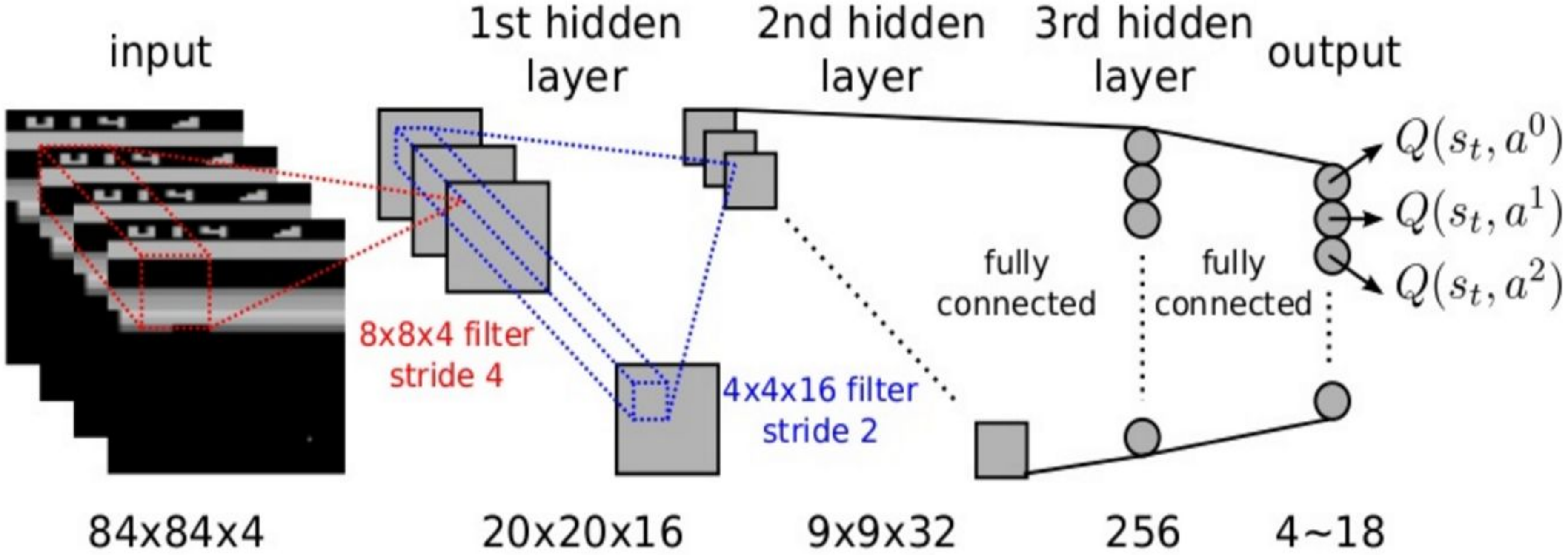
MDP state for input

By looking at only one frame of game we don't get any clarity about the movement of ball (is it moving towards paddle or away from it).

We stack 4 frames together to resolve this problem and treat this as 'state'.



Architecture of DQN

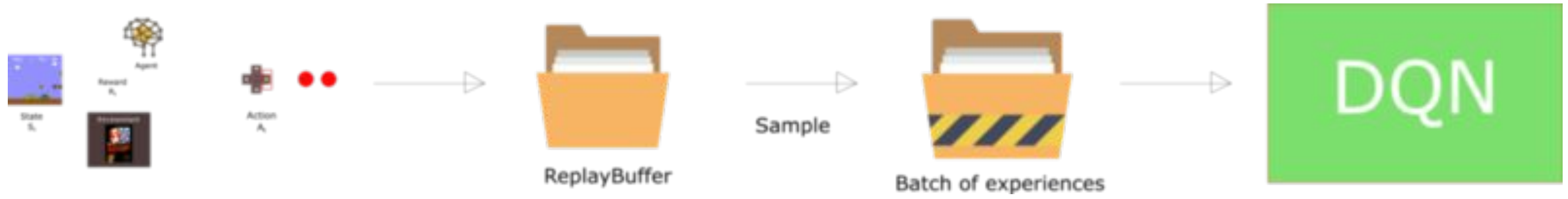


Experience Replay

Experience Replay stores experiences including state transitions, rewards and actions, which are necessary data to perform Q learning, and randomly sample out mini-batches to update neural networks.

Transition Tuple : $e_t = (s_t , a_t , r_t , s_{t+1})$

Set of these tuples , e_t is called **Replay Buffer**.



Problems solved by Experience Replay

- By sampling from the replay buffer at random, we can break this correlation. This prevents action values from oscillating or diverging catastrophically.
- When we give sequential samples from interactions with the environment to our neural network, it tends to forget the previous experiences as it overwrites with new experiences. So, our approach of using experience replay resolves this problem.

Bellman Equation :

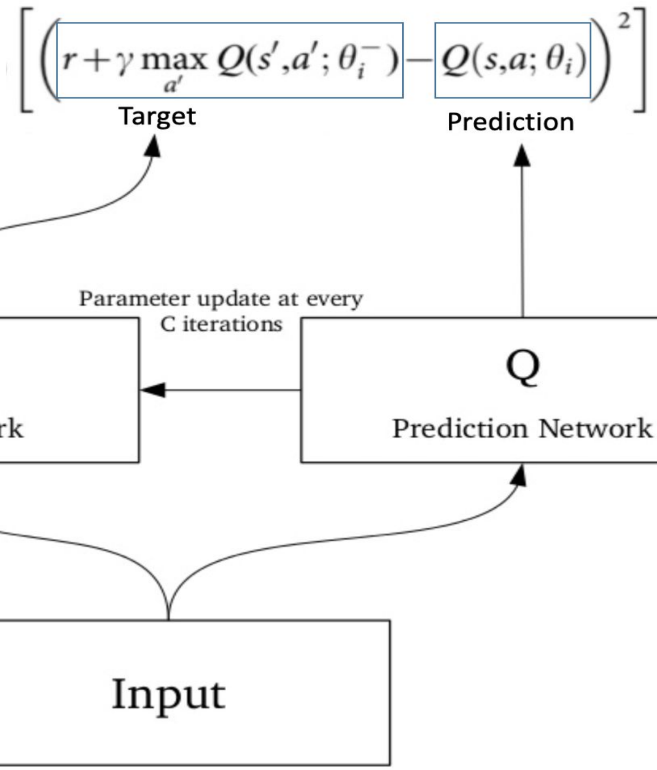
$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s',a)$$

Loss for DQN :

$$\begin{aligned} L(\theta) &= \mathbb{E}_{(s,a,r,s')}\left[\frac{1}{2}(\text{Bellman} - Q(s,a;\theta))^2\right] \\ &= \mathbb{E}_{(s,a,r,s')}\left[\frac{1}{2}\left(R(s,a,s') + \gamma \max_{a'} Q(s',a';\theta) - Q(s,a;\theta)\right)^2\right] \end{aligned}$$

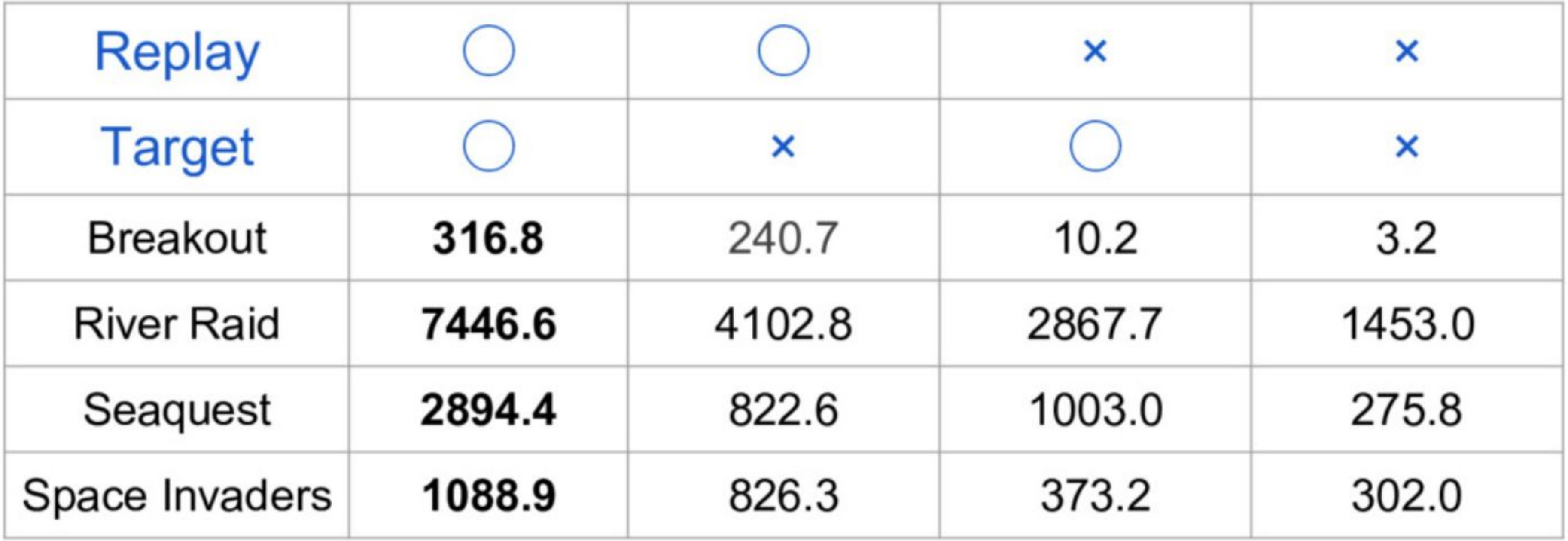
Moving target

Target Network



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

Contribution of Replay and Target Network to stability



Replay	○	○	×	×
Target	○	×	○	×
Breakout	**316.8**	240.7	10.2	3.2
River Raid	**7446.6**	4102.8	2867.7	1453.0
Seaquest	**2894.4**	822.6	1003.0	275.8
Space Invaders	**1088.9**	826.3	373.2	302.0

○ : Present **X** : Not Present

Pseudocode for DQN

```
● Initialize network Q and target network Q'
● Initialize replay memory D
● Initialize Agent to interact with environment
● While not converged do:
    ○ $\epsilon \leftarrow$ new $\epsilon$ using $\epsilon$ decay
    ○ Choose action a using using $\epsilon$-greedy policy
    ○ Agent takes action a, receives reward r and reaches new state
    ○ Store transition ( s, a, r, s', done ) in D
    ○ if enough experience in D then
● Sample random minibatch of N transitions from D.
● For every minibatch do:
    ○ if done then $y_i = r_i$
    ○ else $y_i = r_i + \gamma max\ Q'(s_i', a_i')$
    ○ Calculate loss : $\mathcal{L} = 1/\mathrm{N} \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
        ○ Update Q using gradient descent
    ● Every C step, copy weights from Q to Q'
```

Lets have a look at code for DQN