# A brief look into RL and the tools used in DL

Gautham B,Dheemanth R Joshi and Ch Sai Pranay

PES1UG20EC044,PES1UG20EC059 and PES1UG20EC048

# Contents

# Chapter 1

# Reinforcement Learning

Reinforcement learning(RL) is an area of machine learning concerned with how intelligent *agents* ought to take *actions* in an *environment* to maximize a notion of cumulative *reward*. RL is one of the three basic machine learning(ML) paradigms, alongside *supervised* and *unsupervised* learning. RL differs from supervised learning in not needing labelled input/output pairs to be present and in not needing to be corrected for sub-optimal actions. Instead the focus is on finding a balance between *exploration*(of uncharted paths) and *exploitation*(of learnt knowledge). This is done through the use of a single scalar feedback signal called the *reward*.

## 1.1 Intro to RL

## 1.2 Components of a RL based system

Any RL setup can be reduced down to three interoperating components.

1. Agent
2. Environment
3. Reward

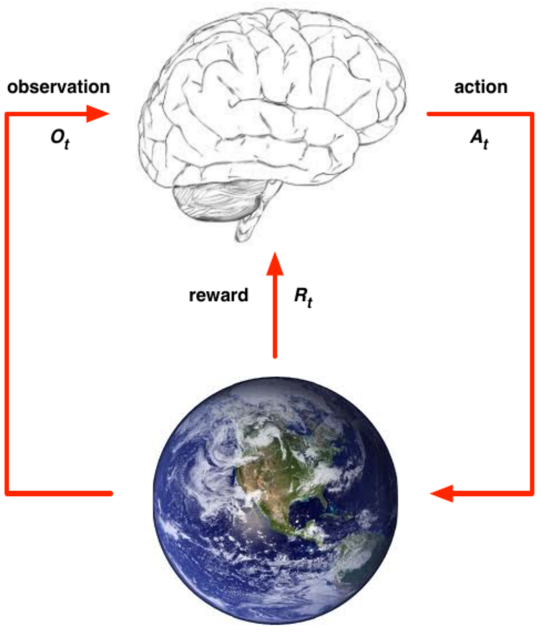


Figure 1.1: Agent-environment interaction

### 1.2.1 Agent

The agent is the component of the RL setup which performs the process of sequential descision making. Its goal is to maximize the total future reward.

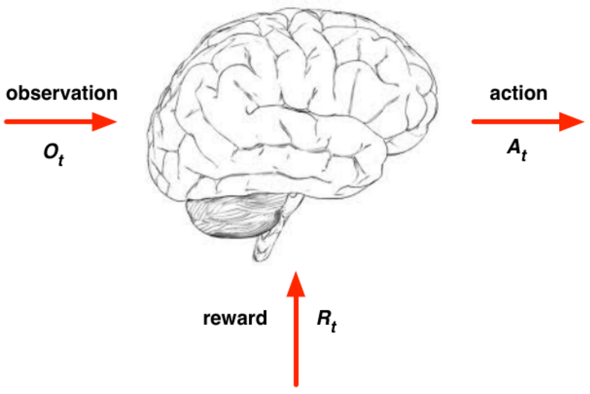


Figure 1.2: Agent

### 1.2.2 Environment

The environment is the 'playground' through which the agent enacts its actions. The enviroment also provides the feeback to the agent via a reward signal as shown in 1.1. The agent environment interation can be summarized as follows:

- At each step t, the agent:
  1. Executes action $A_t$.
  2. Receives observation $O_t$.
  3. Recieves scalar reward $R_t$.

- The environment:
    1. Recieves action $A_t$.
    2. Emits observation $O_{t+1}$.
    3. Emits scalar reward $R_{t+1}$.

### 1.2.3 Reward

A reward $R_t$ is a scalar feedback signal. It indicates how well an agent in doing at a step $t$. RL is based on the *reward hypothesis*:

**All goals can be described by the maximization of cumulative reward.**

So we would like our rewards to encode the following characteristics:

- Actions may have long term consequences.
- Rewards may be delayed.
- It may be better to sacrifice immediate reward to gain more long-term rewards.

## 1.3 History and State

The *history* is a sequence of observations, actions and rewards i.e all observable variables up to time $t$.

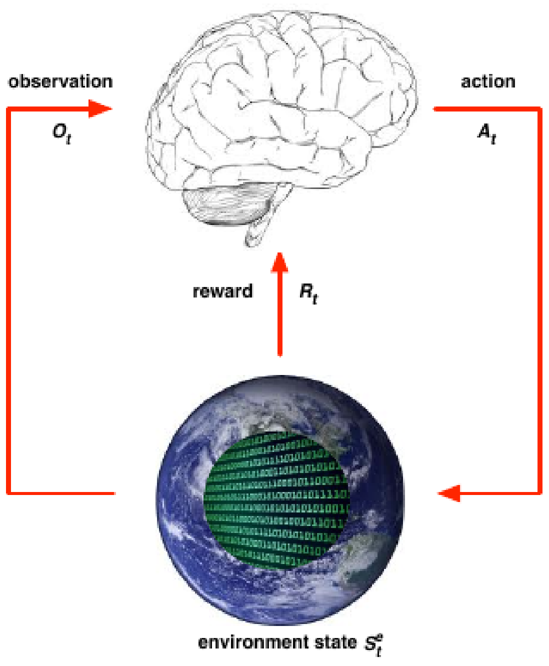
$$H_t = O_1, R_1, A_1 \dots, A_{t-1}, O_t, R_t$$

The history determines what will happen next int he given process, more specifically it is the *state* which determines what heppens next. State is a function of history i.e:

$$S_t = f(H_t)$$

### 1.3.1 Environment State

The environment state ${S_t}^e$ is the environments priavte representation of the events to come i.e it is whatever data the environment uses to pick the next observation/reward. This state is not usually visible to the agent and even if it is, environment state may contain irrelevant information.



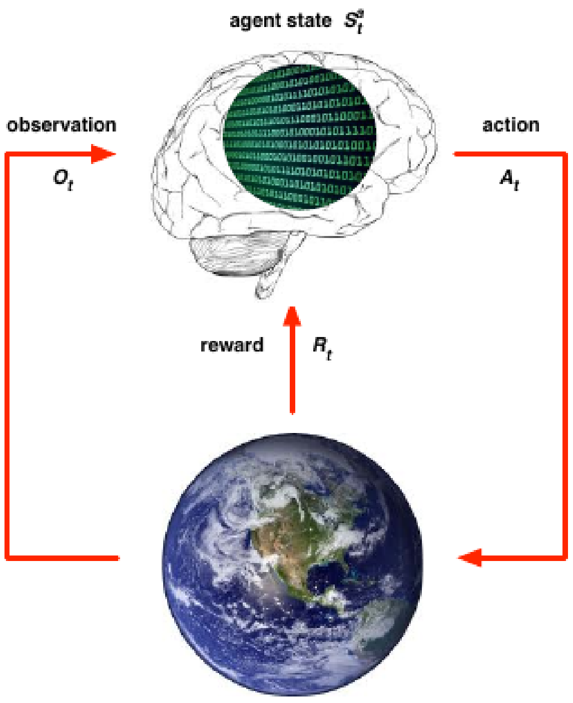
In the case of fully observable environments the agent is able to durectly observe ${S_t}^e$ i.e:

$$O_t = {S_t}^a \ = {S_t}^e$$

Such environments are formally described by *Markov Decision process*(MDP) see 1.5.

### 1.3.2 Agent State

The agent state ${S_t}^a$ is the agent's internal representation of the environment i.e it is whatever data the agent uses to pick the next observation/reward.



## 1.4 Components of an RL agent

An RL agent may include one or more of the following:

- Policy
- Value function
- Model

### 1.4.1 Policy

A *policy* is a map from state to action i.e it mathematical descriptor of an agent's behaviour. A policy can be deterministic or stochastic. Policies are denoted with $\pi$.

### 1.4.2 Value function

A *value* function is a measure of the prediction of future rewards. It is used to evaluate the quality of a state and hence can be used to select between actions.

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots \mid S_t = s]$$

### 1.4.3 Model

A *model* is a representation of the environment created by the agent.

### 1.4.4 Types of RL agents

Based on the presence or absence of the above components, an RL agent agent can be classified into the following fields:

- Value based
  - ~~Policy~~
  - Value function
- Policy based
  - Policy
  - ~~Value function~~
- Actor Critic
  - Policy
  - Value function

## 1.5 Markov Decision Process

*Markov Descision Processes*(MDP) are a formal definition of fully observable environments in the setting of RL. To build an understanding of MDPs we must first being by learning the building blocks of MDPs which are MP and MRPs.

### 1.5.1 Markov Process

A Markov process is a memoryless random process, i.e a sequence of random states $S_1, S_2, \ldots$ with the *Markov property.*

**Markov Property:**

A state $S_t$ is Markov iff

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, \ldots, S_t]$$

"The future is independent of the past given the present."

The state captures all relevant information from the history and once the state is know the history can be thrown away.

**State Transition Matrix:**

For a Markov state $s$ and sucessor state $s\prime$, the state transition matrix is defined as:

$$P_{ss\prime} = \mathbb{P}[S_{t+1} = s\prime \mid S_t = s]$$

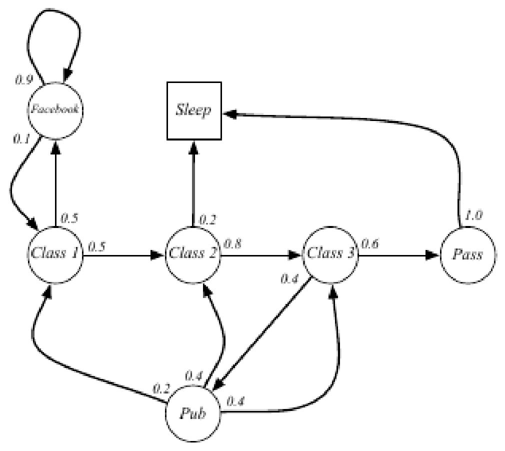
So this matrix defines the transition probabilities from all states $s$ to all successor states $s\prime$.

$$\mathcal{P} = \mathit{from} \overset{\mathit{to}}{\begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix}}$$

Figure 1.3: State transition matrix $P$

Given the above definition of Markov property it is clear that a Markov process can be cleanly represented by the state space $S$ and the probability transition matrix $P$.

**A Markov process is a tuple $\langle S, P \rangle$ .**



$$\mathcal{P} = \begin{array}{c} \\ C1 \\ C2 \\ C3 \\ Pass \\ Pub \\ FB \\ Sleep \end{array} \begin{array}{ccccccc} C1 & C2 & C3 & Pass & Pub & FB & Sleep \\ & 0.5 & & & & 0.5 & \\ & & 0.8 & & & & 0.2 \\ & & & 0.6 & 0.4 & & \\ & & & & & & 1.0 \\ 0.2 & 0.4 & 0.4 & & & & \\ 0.1 & & & & & 0.9 & \\ & & & & & & 1 \end{array}$$

Figure 1.4: An example Markov chain

### 1.5.2 Markov Reward Process

The Markov chain has dealt only with a Stochastic environment. Let us now bring it closer to the problem of RL by including one of the components : **rewards**.

A MRP is a Markov chain with *values*.

**A Markov Reward Process is a tuple** $\langle S, P, R, \gamma \rangle$.

Where $R$ is the reward function and $\gamma \in [0, 1]$ is called the discount factor.

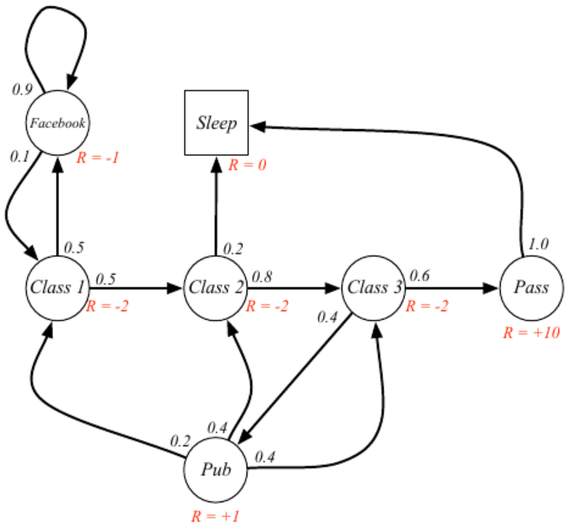


Figure 1.5: An example Markov Reward Process

Note the changes between 1.4 where now each arc contains a reward upon exiting the state.

**Return:**

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

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

. The discount factor $\gamma$ is a measure of the "value" of receinving reward $R$ after $k + 1$ time steps from $t$.

$\gamma$ close to 0 leads to "myopic" evaluation(value immedeate rewards, do not care about future rewards.)

$\gamma$ close to 1 leads to "far-sighted" evaluation(weight rewards equally and consider the scope of all rewards from this time step onwards).

**Value Function:**

The value function $v(s)$ gives the long-term value of a state $s$. The state value function of an MRP is the expected return starting from state $s$.

$$v(s) = \mathbb{E}[G_t \mid S_t = s]$$

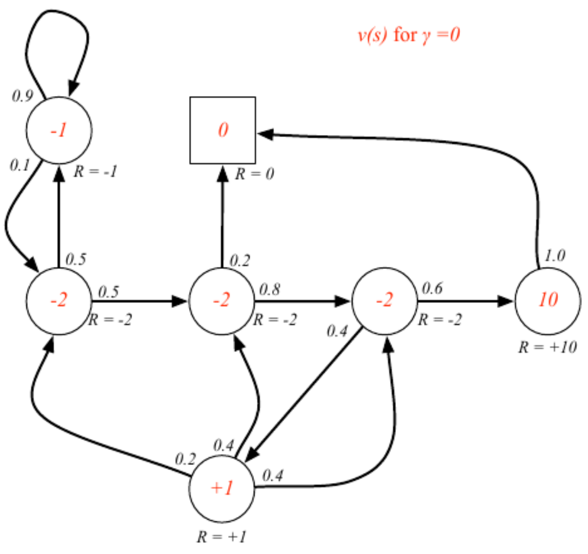


Figure 1.6: MRP with values

**Bellman Equation for MRPs**

The value function can be decomposed into two parts.

- immediate reward $R_{t+1}$.
- discounted value of successor state $\gamma v(S_{t+1})$.

$$v(s) = R_s + \gamma \sum_{s\prime \in S} P_{ss\prime} v(s\prime)$$

$$v(s) = \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s\right]$$

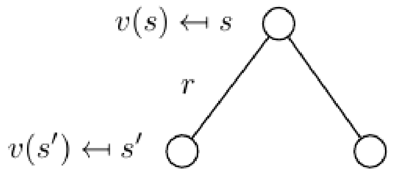


Figure 1.7: Computation graph demonstrating the Bellman equation

The Bellman equation can be expressed more concisely using matrices:

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{11} & \dots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

Figure 1.8: Bellman equation in matrix form

$$v = R + \gamma P v$$

Which gives us a simple linear equation that can be solved as:

$$v = (I - \gamma P)^{-1} R$$

This may sound great, however the computational complexity of solving this is $O(n^3)$ given $n$ states. So the direct solution obtained this way is only feasible for small MRPs. The use of iterative methods is required for large MRPs

### 1.5.3 Markov Decision Process

The MRP came closer to the discription of an RL setting, however it is missing one key element that will enable it to describe an RL setting i.e agency or *actions*. An MDP is an MRP with decisions/actions.

**A Markov Decision Process is a tuple $\langle S, A, P, R, \gamma \rangle$.**

Where $A$ is a finite set of actions. $P$ is the state transition matrix. Now every action must have its own state transition matrix.

$$P_{ss\prime}{}^{a} = \mathbb{P}[S_{t+1} = s\prime \mid S_t = s, A_t = a]$$

$R$ is the reward function which may also depend on the action taken from a state $s$.

$$R_s{}^{a} = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

**Policies:**

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

$$\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$$

This is a stochastic policy.

A policy fully defines the behaviour of an agent.

MDP policies depend only on current state(they follow the Markov property) i.e the policies are stationary(time-independent).

#### Reduction of MDP to an MP or MRP

Given an MDP $M = \langle S, A, P, R, \gamma \rangle$ and a policy $\pi$, we can reduce it down to:

- an MP $\langle S, P^{\pi} \rangle$
- an MRP $\langle S, P^{\pi}, R^{\pi}, \gamma \rangle$

where $P^{\pi}$ and $R^{\pi}$ are the averaged transition probability and rewards over the policy $\pi$.

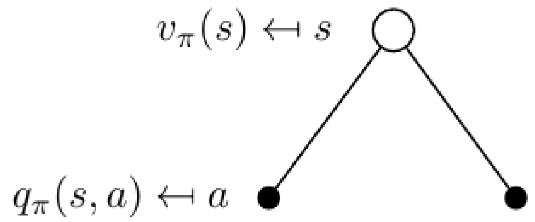
**Value functions:**

- The state value fucntion $v_{\pi}(s)$ of an MDP is the expected return starting from state $s$ whilst following the policy $\pi$.
- The action value function $q_{\pi}(s, a)$ is the expected return starting from state $s$, taking an action $a$, and then follwing policy $\pi$.

**Bellman Equations:**

Once again, the Bellman equation can be used to decompose both the state and action value functions into the immediate reward and the value of the state you will end up in.

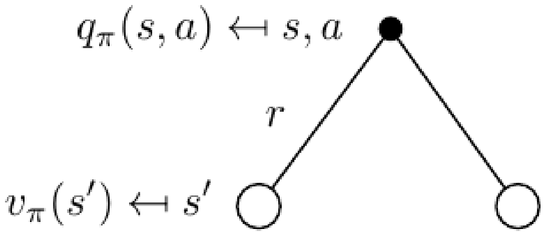
**Bellman equation for** $v_\pi$


$v_\pi(s) \leftarrow\!\shortmid\, s$
$q_\pi(s,a) \leftarrow\!\shortmid\, a$

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a)$$

Figure 1.9: The state value written as the look-ahead of action value

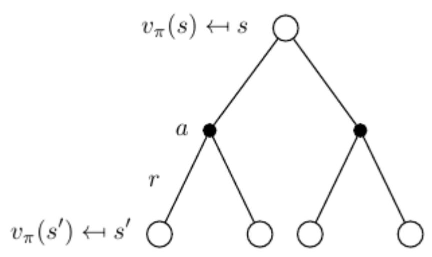
**Bellman equation for** $q_\pi$


$q_\pi(s,a) \leftarrow\!\shortmid\, s,a$
$r$
$v_\pi(s') \leftarrow\!\shortmid\, s'$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$

Figure 1.10: The action value written as the look-ahead of state value

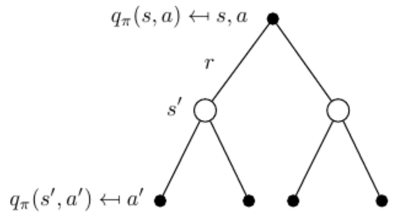
By chaining 1.10 into 1.9 we obtain a way to obtain state value as a look-ahead of itself. **Bellman equation for** $v_\pi$



$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left( \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

Figure 1.11: The state value written as the look-ahead of itself

By chaining 1.9 into 1.10 we obtain a way to obtain action value as a look-ahead of itself. **Bellman equation for $q_\pi$**



$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a')$$

Figure 1.12: The action value written as the look-ahead of itself

An example of state values determined by the above process is shown below.

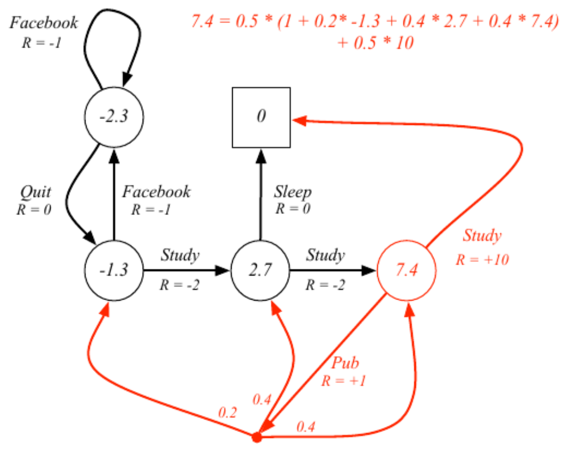


Figure 1.13: State value and its verification using the Bellman equations. Note that here $\pi(a \mid s) = 0.5$

The above options enable us to solve for the value functions in two ways:

- Use 1.5.3 to convert the MDP to an MRP and solve for $v$ using the Bellman equations in matrix form.

- Use dynamic programming to do a look ahead of the value functions.

the second solution is preferred because of the $O(n^3)$ time complexity of the first solution.

**Optimal Value Function:**

The optimal state value function $v_\star(s)$ is the maximum value function over all policies.

$$v_\star(s) = \max_\pi v_\pi(s)$$

The optimal action value function $q_\star(s,a)$ is the maximum action-value function over all policies.

$$q_\star(s,a) = \max_\pi a_\pi(s,a)$$

The optimal value function specifies the best possible performance in the MDP. The MDP is solved when the optimal action val function is known because now you chose the actions which with the highest value of $q_\star$.

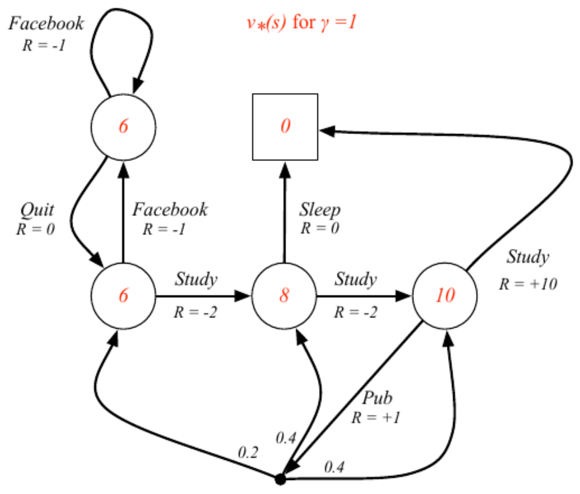


Figure 1.14: Optimal state value for the MDP

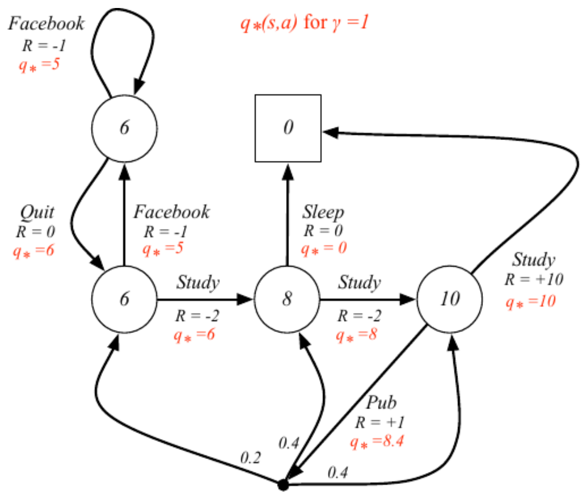


Figure 1.15: Optimal action value for the MDP

# Chapter 2

# Useful tools when working with Deep Learning

This section is a demonstration of the various tools that can be used to train deep learning models. We will demonstrate this by implementing a simple linear regressor from scratch(base modules and numpy) and using inbuilt tools from deep learning libraries.

## 2.1 Python

### 2.1.1 No modules

In this section we will try to implement the training of a linear regressor $y = 2x + 3$ without inbuilt functions. For this purpose we require three things.

1. Model
2. Calculation of gradients
3. Updation of parameters

**Model**:
We will define the model wherein we store the value of weight and bias in the form of two variables $w$ and $b$.

**Calculation of gradients**:
For this task we must define a loss function $L(y, \hat{y})$. Let us consider the *Mean squared loss*:

$$L(y, \hat{y}) = \frac{(y - \hat{y})^2}{2}$$

The gradients for $w$ and $b$ will hence become(from the chain rule):

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial w} = (y - \hat{y})x$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial b} = (y - \hat{y})$$

**Updation of parameters**:

Can be done as:

$$w = w - \eta\frac{\partial L}{\partial w}$$

$$b = b - \eta\frac{\partial L}{\partial b}$$

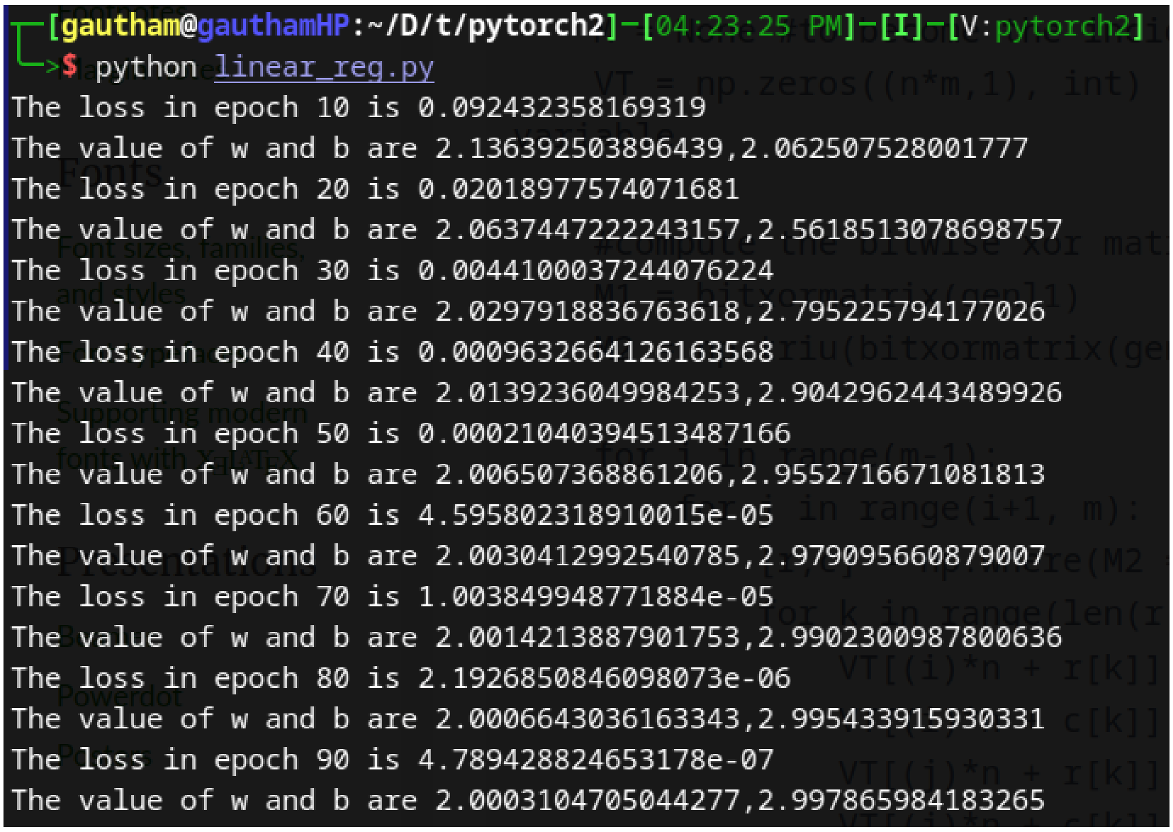
**Implementation**:

Code:

```
import numpy as np

def y(x):
    return 2*x + 3

def loss(y,y_hat):
    return 0.5*(y-y_hat)**2
w = 0 #trainable params
b = 0
X = np.array([1,2,3,4,5,6]) #Dataset
Y = y(X) #Targets
eta = 0.08
for epoch in range(1,100):
    l=0
    for x,y in zip(X,Y):
        out = w*x + b
        l+= loss(out,y)
        w = w - eta*(out-y)*x
        b = b - eta*(out-y)
    if epoch%10==0:
        print(f'The loss in epoch {epoch} is {l/6}')
        print(f'The value of w and b are {w},{b}')
```



```
┌─[gautham@gauthamHP:~/D/t/pytorch2]─[04:23:25 PM]─[I]─[V:pytorch2]
└─>$ python linear_reg.py
The loss in epoch 10 is 0.092432358169319
The value of w and b are 2.136392503896439,2.062507528001777
The loss in epoch 20 is 0.02018977574071681
The value of w and b are 2.0637447222243157,2.5618513078698757
The loss in epoch 30 is 0.0044100037244076224
The value of w and b are 2.0297918836763618,2.795225794177026
The loss in epoch 40 is 0.0009632664126163568
The value of w and b are 2.0139236049984253,2.9042962443489926
The loss in epoch 50 is 0.00021040394513487166
The value of w and b are 2.006507368861206,2.9552716671081813
The loss in epoch 60 is 4.595802318910015e-05
The value of w and b are 2.0030412992540785,2.979095660879007
The loss in epoch 70 is 1.003849948771884e-05
The value of w and b are 2.0014213887901753,2.9902300987800636
The loss in epoch 80 is 2.1926850846098073e-06
The value of w and b are 2.0006643036163343,2.995433915930331
The loss in epoch 90 is 4.789428824653178e-07
The value of w and b are 2.0003104705044277,2.997865984183265
```

### 2.1.2 pytorch

The use of modules such as pytorch can make the implementation of Deep learning models very simple as pytorch provides the neccessary tools to build your model while also providing the functions required to train them. We will now implement the same linear regeressor but using the following features:

**Model**:

pytorch provides the base linear regressor as an inbuilt *nn.Linear* class.

**Gradient caculations**:

pytorch is able to build up the computational graph for model and is hence able to backpropagate the gradients using any of the inbuilt loss functions that are provided.

**Updation of parameters**:

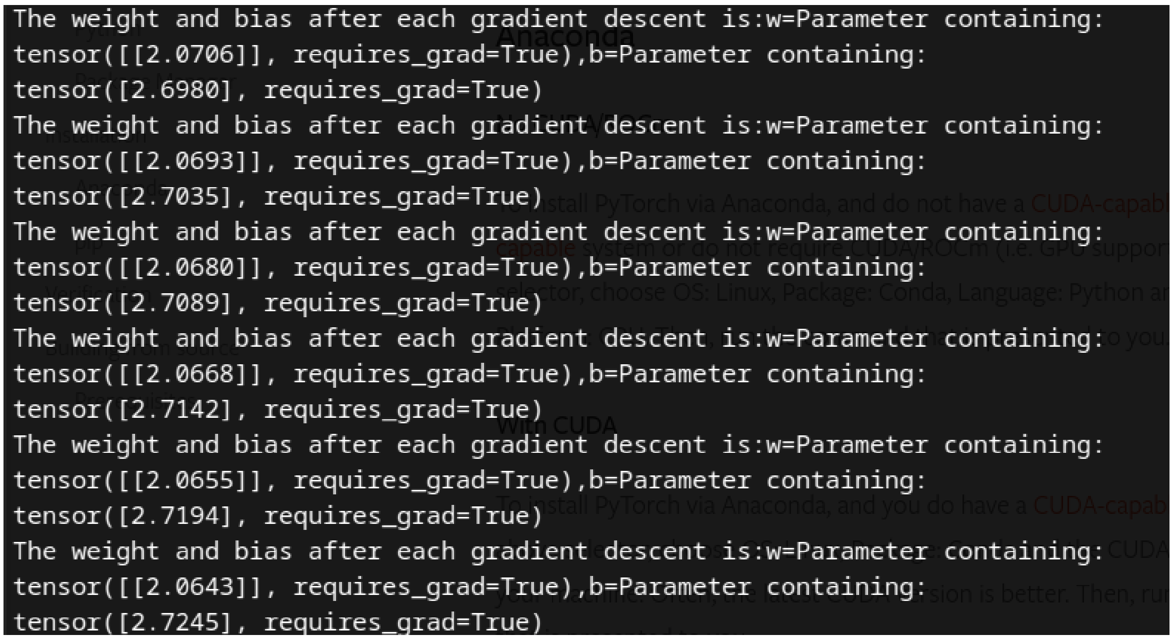
pytorch also has built in optimizers in the *torch.optims* class that can train your model.

Code:

```
import torch
import torch.nn as nn

X=torch.tensor([[1],[2],[3],[4],[5],[6]],dtype=torch.float32)
Y=2*X + 3
model=nn.Linear(1,1,bias=True)
loss=nn.MSELoss()
optimizer = torch.optim.SGD(params=model.parameters(),lr=0.05)

for epoch in range(1,100):
    w,b=model.parameters()
    print(f'The weight and bias after each gradient descent is:w={w},b={b
    y=model(X)
    l=loss(Y,y)
    l.backward()
    optimizer.step()
    optimizer.zero_grad()
    if epoch%10==0:
        print(f'The value of w and b are {model.parameters}')
```



```
The weight and bias after each gradient descent is:w=Parameter containing:
tensor([[2.0706]], requires_grad=True),b=Parameter containing:
tensor([2.6980], requires_grad=True)
The weight and bias after each gradient descent is:w=Parameter containing:
tensor([[2.0693]], requires_grad=True),b=Parameter containing:
tensor([2.7035], requires_grad=True)
The weight and bias after each gradient descent is:w=Parameter containing:
tensor([[2.0680]], requires_grad=True),b=Parameter containing:
tensor([2.7089], requires_grad=True)
The weight and bias after each gradient descent is:w=Parameter containing:
tensor([[2.0668]], requires_grad=True),b=Parameter containing:
tensor([2.7142], requires_grad=True)
The weight and bias after each gradient descent is:w=Parameter containing:
tensor([[2.0655]], requires_grad=True),b=Parameter containing:
tensor([2.7194], requires_grad=True)
The weight and bias after each gradient descent is:w=Parameter containing:
tensor([[2.0643]], requires_grad=True),b=Parameter containing:
tensor([2.7245], requires_grad=True)
```