Summer of Science 2023
End-Term Report
Artificial Intelligence

Mentee: Deeksha Dhiwakar
Roll number: 22B0988
Mentor: Mahesh Karewad

27 July 2023

Contents

1 Introduction

This report outlines my progress in the Summer of Science project I have undertaken. My learning has been majorly through online courses, the links to which are listed under the References section.

2 Week 1: Python Bootcamp

I spent the first week brushing up the basic concepts of Python as well as learning some advanced concepts. I took guidance from the resources [1] and [2]. The major topics that I covered are as follows.

2.1 Python Objects and Classes

Classes, or object-constructors, form the very essence of Python, it being an Object Oriented Programming Language. Almost all the data structures we encounter in Python, with their methods and properties, are objects. Classes are like 'blueprints' for creating objects.
Following is an example class that I have implemented in one of my programs.

```python
class StackFrontier():
    def __init__(self):
        self.frontier = []

    def add(self, node):
        self.frontier.append(node)

    def contains_state(self, state):
        return any(node.state == state for node in self.frontier)

    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
            return node
```

Figure 1: Python Class

The __init__ class is automatically invoked when we create a new instance of the class. For all other functions, `self` must be given as the first parameter in the declaration, although it can be skipped in the function call.

2.2 File handling

File handling is used quite often in AI related problems, mainly to train our AI agent or to pass input data to it.

2.2.1 Text file handling

Text files can be used in three modes- read, write and append, with the default mode being read. Some important utilities are

- file.read(): Returns the entire contents of the file
- file.readline(): Returns the next line of the file
- file.readlines(): Returns a list of all the lines of the file

2.2.2 CSV file handling

CSV (comma separated value) files are used to store large amounts of tabular data, such as spreadsheets. Commas are used as delimiters for the different fields of each record. CSV files can be handled using a separate csv module in Python.

2.2.3 Binary file handling

Binary files contain data in a format not readable to humans. However, binary files are easier for the computer to process as compared to CSV or text files, hence binary files are often used to store and share information. The `pickle` module of Python is used to handle binary files.

2.3 Python Module - NumPy

NumPy, or Numerical Python, finds significant applications in the domain of Artificial Intelligence as it provides a means for vectorization, which greatly speeds up computation and optimizes algorithms.

2.3.1 NumPy Arrays

NumPy arrays are used to store our datasets and perform computations efficiently. Following is a snippet of code that uses NumPy to implement simple array addition. The code also uses broadcasting, which is another important concept commonly used in neural networks.

```
[1]  import numpy as np
     arr1 = np.array([1, 2, 4, 5])
     element = 1
     arr2 = arr1 + element
     print(arr2)

     [2 3 5 6]
```

Figure 2: Simple array implementation

2.3.2 Commonly used array methods

- array.shape(): Returns the dimensions of the array
- array.reshape(new dimensions): Used to alter the dimensions of an array
- np.sum(array): Used to sum up the elements of an array.

- np.add(arr1, arr2): Elementwise addition of the two arrays
- np.subtract(arr1, arr2): Elementwise difference of the two arrays
- np.multiply(arr1, arr2): Elementwise product of the two arrays
- np.divide(arr1, arr2): Elementwise division of the two arrays
- np.dot(arr1, arr2): Matrix multiplication (in case of 2D arrays) or dot product (in case of column or row vectors)

2.4 Python Module - Matplotlib

From the matplotlib module, we most often use pyplot to visualize our data graphically. The following snippet shows the use of pyplot.

```
from matplotlib import pyplot as plt
y_pts = np.array([3, 1, 5, 2, 4])
plt.plot(y_pts)
plt.show()
```

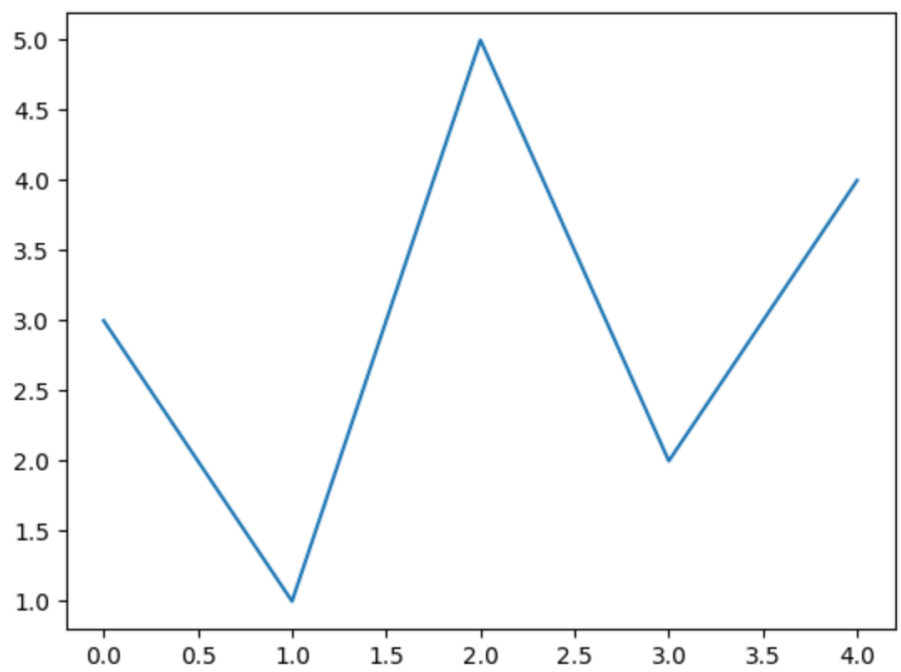

Figure 3: Simple Pyplot implementation

2.5 Python Module - Pandas

Pandas is a Python library used to analyse, explore, manipulate and clean datasets. The following snippet shows the use of pandas.

```
[3] import pandas as pd
    mydataset = {'Values' : [4, 12, 3, 7], 'Occurence' : [30, 10, 20, 40]}
    mytable = pd.DataFrame(mydataset)
    print(mytable)
```

```
   Values  Occurence
0       4         30
1      12         10
2       3         20
3       7         40
```

Figure 4: Simple Pandas implementation

3 Week 2: Supervised Learning via Neural Networks

For this section, my main guide was the Coursera course on Neural Networks and Deep Learning by Andrew Ng [3].

3.1 Logistic Regression

I mainly focused on implementing logistic regression as a binary classifier. Using the training data, we train parameters w and b to implement the fitting $\hat{y} = w^T x + b$. This training is done using gradient descent. The example used in the course was an image classifier that could identify images as 'cat' or 'non-cat'. The input features x are stored as a set of 3 $n \times n$ matrices. These are converted into a column vector of dimensions $(n * n * 3, 1)$ and then fed as input into the classifier. $\hat{y}$ gives the probability that y is 1. In logistic regression, we initialize w to a zero column vector using

```
w = np.zeros(n*n*3, 1)
```

and b to 0. As an activating function, we use the sigmoid function.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The sigmoid function takes only values between 0 and 1, making it ideal to use to determine probability. The graph of the function is shown in the figure.

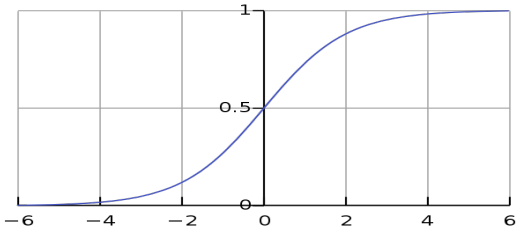

Figure 5: Sigmoid function

Thus $\hat{y}$ is computed as

$$\hat{y} = \sigma(w^T x + b) \tag{1}$$

3.1.1 Cost function

The loss function, or error function, is a function that tells us how well our model is performing on a single training example. It is computed as

$$L(\hat{y}, y) = -[ylog\hat{y} + (1 - y)log(1 - \hat{y})] \tag{2}$$

The cost function is the average of the loss function, computed over the entire training set. The formula for it is

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^i, y^i) \tag{3}$$

Our goal is to find parameters w and b that minimize the cost function. We achieve this using gradient descent.

3.1.2 Gradient Descent

We initialize the parameters with random values. Then we take successive steps in the direction of steepest descent until we achieve the desired level of precision. The algorithm for gradient descent is thus given by

$$repeat[w := w - \alpha \frac{dJ(w,b)}{dw}; b := b - \alpha \frac{dJ(w,b)}{db}] \tag{4}$$

Generally in code, it is common practic to use dw to denote $\frac{dJ}{dw}$ and db to denote $\frac{dJ}{db}$

3.1.3 Computation Graph

It provides a step by step visualization of the output computation process. A left to right pass (forward propagation step) allows us to compute the output whereas a right to left pass (back propagation step) allows us to compute derivatives, which is useful in updating the parameters in gradient descent.

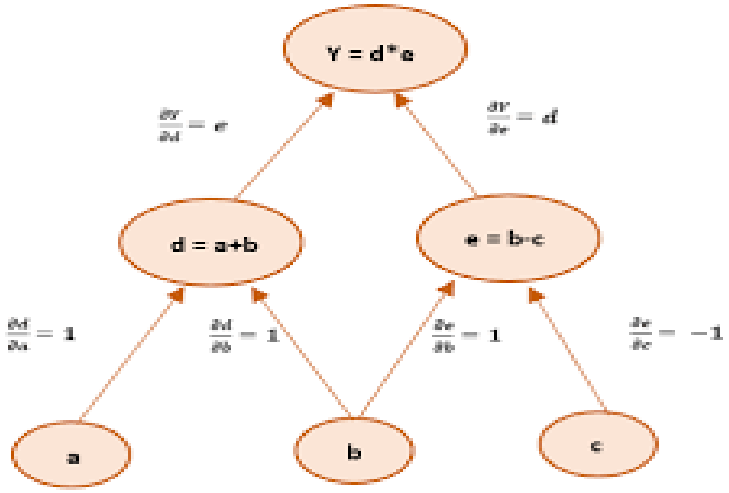

Figure 6: Computation Graph

3.1.4 Logistic Regression Implementation

Let us consider a sample case where we have only two input features, x_1 and x_2. So along with these we need to input parameters w_1, w_2, b.

$$\frac{dL(a,y)}{da} = da = -\frac{y}{a} + \frac{1-y}{1-a} \tag{5}$$

Using this, from the computation graph, dz can be computed as

$$\frac{dL}{dz} = \frac{dL}{da}\frac{da}{dz} \tag{6}$$

$$\frac{da}{dz} = a(1-a) \tag{7}$$

$$dz = a - y \tag{8}$$

$$dw_1 = X_1 dz \tag{9}$$

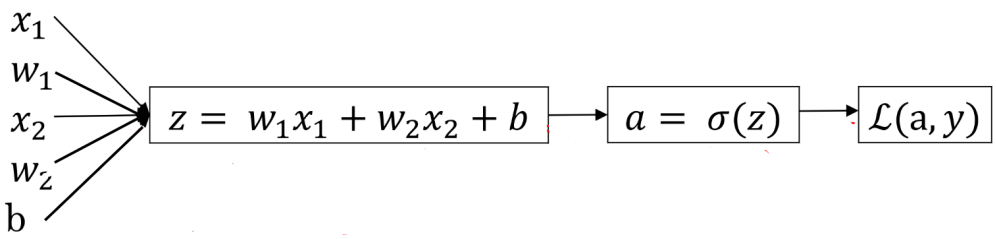

Figure 7: Simple logistic regression model

$$dw_2 = X_2 dz \tag{10}$$

$$db = dz \tag{11}$$

3.1.5 Gradient Descent Implementation over Multiple (m) training examples

The above case computed the derivatives only for a single training example. But in practice, our datasets contain several training examples and hence we must modify our code to compute derivatives independently for each of the m training examples.

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(a^i, y) \tag{12}$$

$$\frac{d}{dw_1} J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{d}{dw_1} L(a^i, y) \tag{13}$$

We initialize $J = 0, dw_1 = 0, dw_2 = 0, db = 0$. The algorithm is as follows.
for i from 1 to m:

$$z^i = w^T x^i + b$$

$$a^i = \sigma(z^i)$$

$$J+ = -[y^i log(a^i) + (1 - y) log(1 - a^i)]$$

$$dz^i = a^i - y^i$$

for j over number of input features (here 2):

$$dw_j+ = x_j^i dz^i$$

$$db+ = dz^i$$

$$J/ = m$$

for j over number of input features (here 2):

$$dw_j/ = m$$

$$db/ = m$$

3.2 Vectorization

Vectorization is a means of optimizing our code by replacing explicit for loops with matrices and vectors in order to speed up computation. The vectorized implementation is several orders of magnitude more efficient than the non-vectorized implementation.
In the vectorized approach, instead of keeping w_1 and w_2 separate, we group them into a single column vector

$$w = [w1, w2]^T$$

and similarly for x_1 and x_2

$$x = [x1, x2]^T$$

Now the code for the forward propagation step is simply
`z=np.dot(w.T, x)+b`
In the previous implementation of logistic regression, we used two explicit for loops. Both loops can be eliminated using vectorization.
Firstly, we eliminate the loop over j by using vector w. The code for these computations hence becomes

$$dw+ = x^i dz^i$$

$$dw/ = m$$

To eliminate the loop over i, we will have to modify the format of our input features. Instead of running a loop over all m training examples, we put all the examples together into a single matrix

$$X = [x^1, x^2 \dots x^m] \tag{14}$$

Hence the forward propagation step reduces to

$$[z^1, z^2 \dots z^m] = w^T X + [b, b \dots b$$

$$Z = np.dot(w.T, X) + b \tag{15}$$

$$A = [a^1, a^2 \dots a^m] = \sigma(Z) \tag{16}$$

For the back propagation step, we define

$$dZ = [dz^1, dz^2 \dots dz^m$$

$$dZ = A - Y \tag{17}$$

So the vectorized version of back propagation is

$$db = \frac{1}{m} np.sum(dZ)$$

$$dW = \frac{1}{m} X dZ^T$$

3.2.1 Final Vectorized Implementation

$$Z = np.dot(w.T, X) + b$$

$$A = \sigma(Z)$$

$$dZ = A - Y$$

$$dw = \frac{1}{m} X dZ^T$$

$$db = \frac{1}{m} np.sum(dZ, axis = 1, keepdims = True)$$

$$w = w - \alpha dw$$

$$b = b - \alpha db$$

If we wish to implement multiple iterations of gradient descent, the only way to do so is to use an explicit for loop. This CANNOT be vectorized.

3.3 Neural Networks

So far, we have studied only logistic regression models with one layer. Now we shift our focus to neural networks, which are nothing but several logistic regression layers put together.

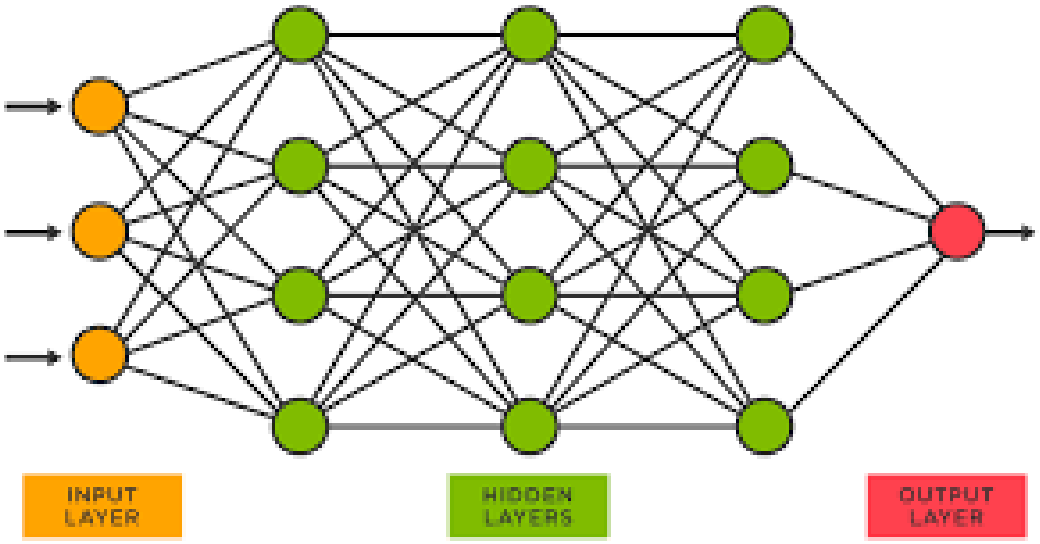

Figure 8: Neural Network

Hidden layer refers to any layer of the neural network that is not the input or output layer. Each hidden layer of the neural network must be fully connected with both the layer before it and the layer after it.
The conventional notation is to use superscript $[i]$ to denote quantities associated with the i^{th} hidden layer. The output of the i^{th} hidden layer is denoted by $a^{[i]}$. $a^{[0]}$ is sometimes used to represent the input set X.
Each node of the neural network is denoted by $a_i^{[l]]}$ where l symbolizes hidden layer number and i symbolizes node number within the hidden layer.

3.3.1 Computing the Output of a Neural Network

The output of a neural network can be found by applying logistic regression multiple times. Now the equations can be tweaked to

$$z_j^{[i]} = w^T x_j^{[i]} + b$$

Let us define

$$W^{[i]} = [w_1^{[i]}, w_m^{[i]} \dots w_m^{[i]}]$$
$$Z^{[i]} = [Z_1^{[i]}, Z_2^{[i]} \dots Z_m^{[i]}]^T$$
$$b^{[i]} = [b_1^{[i]}, b_2^{[i]} \dots b_m^{[i]}]^T$$

So

$$Z^{[i]} = W^{[i]} X + b^{[i]}$$
$$a^{[i]} = \sigma(Z^{[i]})$$

3.4 Activation Functions

In all the previous examples, we used the sigmoid function as our activating function. While this is a popular choice for binary classifiers, it is seldom used elsewhere. This is because if z is large in magnitude, the slope approaches 0 and hence gradient descent proceeds at a very slow rate.

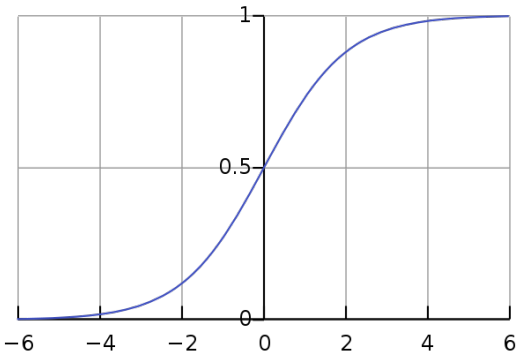

Figure 9: Sigmoid function

An alternate activating function that almost always works better than the sigmoid function is the hyperbolic tan function, defined as

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Its graph is as follows

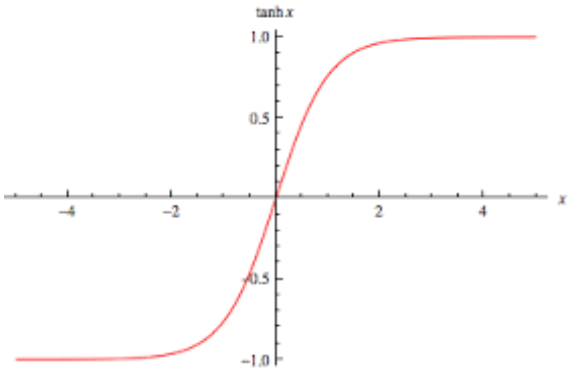

Figure 10: Hyperbolic tan function

This function, although better than the sigmoid function, is also inefficient for large values of $|z|$ The function most often used as an activating function in deep neural networks is the Rectified Linear Unit (ReLU) function. It is the best choice in all layers except the output layer in binary classifiers. ReLU is defined as

$$a(z) = max(0, z)$$

Its graph is as follows

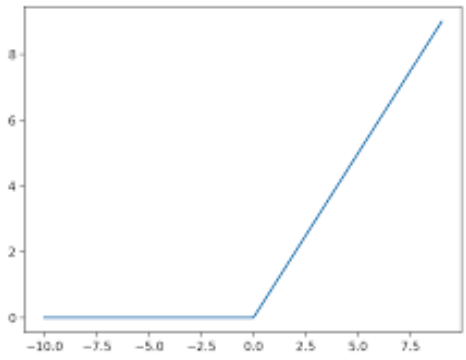

Figure 11: ReLU function

There is another function called the Leaky ReLU function that works even better than ReLU, but is not used much practically. It is defined as

$$a(z) = max(0.01z, z)$$

and its graph is as follows

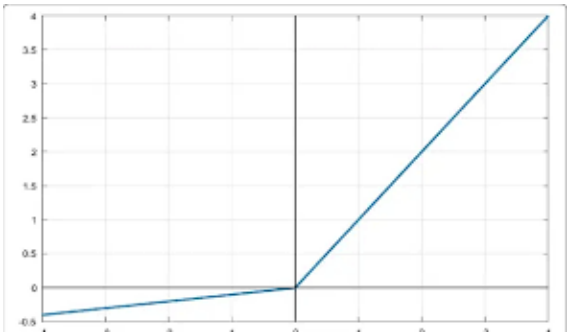

Figure 12: Leaky ReLU function

4 Week 3: Search Algorithms

For this section, I took guidance from Harvard's online CS50x course on AI [4].

4.1 Basic terminology

- Agent- The entity that perceives and acts upon its environment.
- State- A configuration of the agent in its environment
- Actions- Choices that can be made in a state.
- Transition model- A description of what state results from performing any applicable action in any state. It is basically a function that takes the current state and action as input, and outputs the resulting state.
- State space- The set of all states reachable from the initial state by any sequence of actions.
- Goal test- The condition that determines whether a given state is a goal state.
- Path cost- A numerical cost associated with a given path.

4.2 Solving Search problems

We generally use a data structure called a 'node'. It normally keeps track of the following

1. A state
2. The parent node
3. The action to be taken from the parent node to reach the state
4. Path cost from initial state to current state

Approach

We maintain a 'frontier' that contains all the options (nodes) we have left to explore. Initially, it contains only s_i (the initial state from which our agent starts). We repeat the following process:

1. Check if the frontier is empty. If it is, there is no solution (no options left to explore, and goal state has not been achieved). Terminate the loop.
2. Otherwise, remove a node from the frontier.
3. Check if this node satisfies the goal test. If it does, return this as the solution and terminate the loop.
4. Otherwise, 'expand the node', i.e., add neighbour nodes to the frontier and continue the iterative process.

Optimization

The above process has one issue; the agent could always choose to revert to previous states (since the parent node is also a neighbour node) and this could cause an infinite loop. To avoid this, we maintain a set of nodes that have already been explored, and make sure not to include these nodes in the frontier.

4.3 Uninformed Search Algorithms

In this type of search, the agent has no problem specific knowledge. Two examples of this are Depth First Search and Breadth First Search. The difference between them lies in the method of removing a node from the frontier.

4.3.1 Depth first Search (DFS)

In this algorithm, the frontier is implemented as a stack (First in First out). The agent always expands the deepest path and follows it until the end, and moves on to the next path only if the current one does not yield a solution.

- Advantages: In the best possible scenario, DFS finds the optimal path to the goal in its first try, making it the most efficient algorithm. Also in graph traversal problems, it is the simplest to implement.
- Disadvantages: In the worst possible scenario, DFS explores every other path before finding on to the solution, making it the least efficient algorithm.

Following is the code for stack based frontier implementation using a Python list.

```
class StackFrontier():
    def __init__(self):
        self.frontier = []

    def add(self, node):
        self.frontier.append(node)

    def contains_state(self, state):
        return any(node.state == state for node in self.frontier)

    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
            return node
```

Figure 13: Stack based frontier

4.3.2 Breadth first Search (BFS)

In this algorithm, we implement the frontier as a queue (Last in First out). The agent explores all possible paths simultaneously, step by step until one of the path leads to the goal state.

- Advantages: If multiple solutions exist, BFS will always find the optimal one.
- Disadvantages: As it always explores multiple paths, it will always take more time than the fastest possible solution (best case DFS).

Following is the code for queue based frontier implementation using a Python list.

```
7
8   class QueueFrontier():
9       def __init__(self):
10          self.frontier = []
11
12      def add(self, node):
13          self.frontier.append(node)
14
15      def contains_state(self, state):
16          return any(node.state == state for node in self.frontier)
17
18      def empty(self):
19          return len(self.frontier) == 0
20
21      def remove(self):
22          if self.empty():
23              raise Exception("empty frontier")
24          else:
25              node = self.frontier[0]
26              self.frontier = self.frontier[1:]
27              return node
28
```

Figure 14: Queue based frontier

4.4 Informed Search Algorithms

In this type of search, the agent has some knowledge specific to the problem.

4.4.1 Greedy best first search (GBFS)

The agent expands the path that it 'thinks' will lead it to the goal at the lowest cost. It estimates the cost using a heuristic function, which gives some measure of distance of any state from the target. For example, a commonly used heuristic in maze solver agents is the Manhattan distance, which is the sum of absolute difference in coordinates. The efficiency of the GBFS algorithm depends on how good an estimate the heuristic function is. Still, with any heuristic, there is always the possibility that the agent could be led down a path that may be even worse than the one it would have taken in uninformed search. However the overall performance of an informed search algorithm is better than that of an uninformed one.

4.4.2 A* Search

This is an improved version of GBFS. Rather than just minimizing the heuristic $h(n)$, it tries to minimize the sum of the heuristic function and the cost from the starting node to the current node, $h(n) + g(n)$. If at some point on a path, the agent realizes there is a different accessible point with a lower $(h(n) + g(n))$, it will switch over to that path. This prevents it from being misled down an inefficient path by only $h(n)$.
A* search is optimal if the following two conditions are satisfied.

1. $h(n)$ is admissible, i.e., it never overestimates the true cost.
2. $h(n)$ is consistent, i.e., for every node n, for each of its successor nodes n' with step cost from n to n' c, $h(n) \leq h(n') + c$

4.4.3 Adversarial Search

Unlike the algorithms discussed so far, in this type of search, the agent faces an opponent trying to achieve the opposite goal as itself. The most common example of this is in games such as tic-tac-toe or Nim. Now the agent needs to optimize the outcome of the game considering not only its own actions, but those of its opponent as well. This is done using the minimax algorithm.

Minimax Algorithm

Now the outcomes of a game are more abstract and undefined. To represent the notions of our agent winning, losing and drawing with the opponent, we use numbers (eg: 1, -1 and 0 respectively). We represent our agent as MAX and opponent as MIN. The objective of MAX is to maximize the final score, while the objective of MIN is to minimize it. The agents do this by assuming their opponent also plays optimally, exploring all possible actions recursively, and making the move that will ultimately give the maximum (or minimum) final return. An example turn of MAX is as follows.

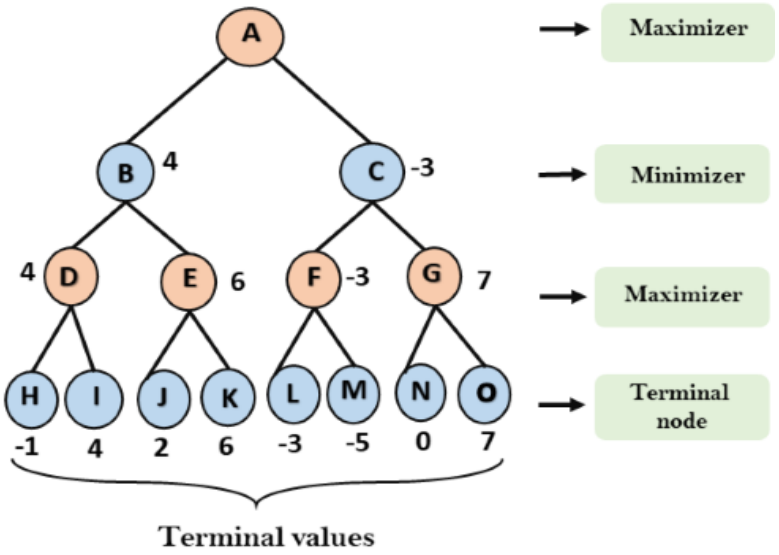

Figure 15: Minimax algorithm

Let us analyse the above example starting from the terminal values. Since the last turn is MAX's, the possible return values for each scenario are 4, 6, -3 and 7. Since MAX is operating under the assumption that MIN is also playing optimally, it deduces that MIN at level 3, via a similar algorithm, will analyse all possible scenarios and end up picking the minimum possible, i.e., 4 in the first case (B) and -3 in the second case (C). Now MAX at level 1 will choose the action that eventually maximizes the outcome, which is 4. SO by performing all these simulations, our agent deduces that itmust take the action that will lead to state B.

Pseudocode for Minimax algorithm

Functions:

- `actions(s)`: Given state s, returns the set of all possible actions that can be taken from s.
- Transition model `result(s, a)`: Returns the state achieved by taking action a from state s.
- `terminal(s)`: Checks if state s is terminal or not.
- `utility(s)`: For terminal state s, returns the score (eg: -1 for MAX losing, 1 for MAX winning, 0 for draw).

Given state s,

MAX picks action a in `actions(s)` that produces the highest value of `min-value(result(s, a))`.
MIN picks action a in `actions(s)` that produces the least value of `max-value(result(s, a))`.

Algorithm:

```
function max-value(state):
    if terminal(state):
        return utility(state)
    v = -INFINITY
    for a in actions(state):
        v = max(v, min-value(result(state, a))
    return v

function min-value(state):
    if terminal(state):
        return utility(state)
    v = INFINITY
    for a in actions(state):
        v = min(v, max-value(result(state, a))
    return v
```

Optimization- $\alpha - \beta$ Pruning

The original Minimax algorithm can be slow in complex games where there are several possibilities to explore. One way of cutting down on the number of possibilities is using $\alpha - \beta$ pruning, as explained using the following example.

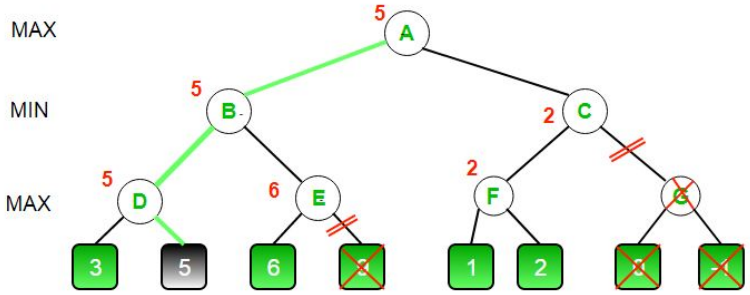

Figure 16: $\alpha - \beta$ pruning

We begin by analysing the terminal values. We traverse the values in order: 3, 5. Since level 3 is the MAX player, it picks 5 for state D. Now we continue traversing the terminal states and encounter 6. Now MAX would pick a value greater than or equal to 6 (call it x) for state E. But at level 2, we have the MIN player. Between 5 and x, MIN will definitely pick 5, and the value of x doesn't matter. So as soon as we encounter 6, we can ignore all other values rooted at E and move on to the next node rooted at F.

Depth-Limited Minimax

In complicated games like chess, where there would be several recursion layers, it is computationally impossible to analyse every possible scenario repeatedly and make an optimal move. In such cases, we limit the depth of recursion and stop simulations there, instead of simulating till a terminal state is reached. To implement this, we will need an additional evaluation function that can tell us the 'value' at intermediate states (since `utility(s)` only gives us the values of terminal states).

5 Week 4: Knowledge

Up until now, we have focused on the implementation of algorithms, not on how the agent processes new information. In this section, I worked on incorporating a knowledge base into the AI agent.

5.1 Basic terms

- Sentence: An assertion about the world in a language used to represent knowledge.
- Propositional symbols: Symbols used to represent statements (P, Q, R etc)
- Logic operators: NOT ($\neg$), OR ($\vee$), AND ($\wedge$), Implication ($\implies$) and Biconditional ($\iff$).
- Model: A description of a 'possible world' by assigning true or false to every propositional symbol.
- Knowledge base: A set of sentences that the AI agent knows to be true.
- Inference: Drawing new conclusions from old ones.
- Entailment: Represented as $\alpha \vdash \beta$ (α entails β); It means that in every model where α is true, β is also true.

5.2 Model Checking Algorithm

The goal of our algorithm is to check if we can conclusively determine a statement α from our knowledge base (KB), i.e., does $KB \vdash \beta$?
We use the following procedure:
Enumerate all models.
If in every model where α is true, β is also true, then $KB \vdash \alpha$.
Otherwise, $KB \vdash \alpha$ does not hold.

However, model checking is inefficient because it loops over all possible values of each symbol and its time complexity is O(2n).
A more efficient method is using Inference Rules.

5.3 Inference Rules

Here we deal only with knowledge, not with 'worlds'. We use inference rules to transform old knowledge into new knowledge. Following are some commonly used inference rules

NAME	PREMISES	INFERENCE
Modus Ponens	$\alpha \implies \beta$, α	β
AND elimination	$\alpha \wedge \beta$	α
Double negation elimination	$\neg(\neg(\alpha))$	α
de Morgan's law - I	$\neg(\alpha \wedge \beta)$	$(\neg\alpha) \vee (\neg\beta)$
de Morgan's law - II	$\neg(\alpha \vee \beta)$	$(\neg\alpha) \wedge (\neg\beta)$
Implication elimination	$\alpha \implies \beta$	$\neg\alpha \vee \beta$
Biconditional elimination	$\alpha \iff \beta$	$(\alpha \implies \beta) \wedge (\beta \implies \alpha)$
Distributive Law - I	$\alpha \wedge (\beta \vee \gamma)$	$(\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$
Distributive Law - II	$\alpha \vee (\beta \wedge \gamma)$	$(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$

Table 1: Inference Rules

Implementation

We implement inference rules by modelling theorem as a search problem, using the following analogies.

Initial State ≡ Knowledge Base
Action ≡ Inference
Transition Model ≡ Function that returns new knowledge base after inference
Goal Test ≡ Check statement we are trying to prove
Path Cost Function ≡ Number of inferences/ Number of steps in proof

5.4 Resolution

This method is more commonly used to implemet inference rules and is based on another powerful inference rule called the 'Resolution Rule'.

$$\text{Premises:} P \vee Q, \neg P; \text{ Inference: Q}$$

$$\text{Premises:} P, \neg P; \text{ Inference: () (Empty Statement)}$$

This can be further generalized to

$$\text{Premises:} P \vee Q, \neg P \vee R; \text{ Inference:} Q \vee R$$

Some terminology:

Clause: A disjunction of literals, for example $P \vee Q \vee R$
Conjunctive Normal Form (CNF): A sentence that is a conjunction of clauses, for example $(A \vee B \vee C) \wedge (D \vee \neg E) \wedge (F \vee G)$

Conversion of any statement to CNF

1. Eliminate biconditionals.
2. Eliminate implications.
3. Move $\neg$ inwards using de Morgan's Laws.
4. Use distribution law to distribute $\vee$ wherever possible.

Inference and Resolution

We can determine if $KB \vdash \alpha$ using the following procedure.

Check if $KB \wedge \neg\alpha$ is a contradiction
If so, $KB \vdash \alpha$
Otherwise, no entailment

To check if $KB \wedge \neg\alpha$ is a contradiction,

1. Convert $KB \wedge \neg\alpha$ into CNF.
2. Keep checking if we can use resolution to produce a new clause.
3. If we ever resolve P and $\neg P$ to produce an empty clause, we reach a contradiction.
4. If we ever reach a state where we cannot produce any more new clauses and haven't produced an empty clause, we do not reach a contradiction.

6 Week 5: Uncertainty

6.1 Probability

Denoted by $P(w)$, it represents the probability 'possible world' w has of being true.

$$0 \leq P(w) \leq 1$$

$$\sum_{w \in \Omega} P(w) = 1$$

Random Variable: Any variable in probability theory that has a domain of values it can take on.

Conditional Probability: $P(a|b) = \frac{P(a \wedge b)}{P(b)}$

Bayes' Theorem $P(b|a) = \frac{P(b)P(a|b)}{P(a)}$

Marginalization: $P(a) = P(a \wedge b) + P(a \wedge \neg b)$ This helps us find unconditional probability distribution from joint probability distribution. More generally,

$$P(X = x_i) = \sum_j P(X = x_i, Y = y_j)$$

Conditioning: $P(a) = P(a|b)P(b) + P(a|\neg b)P(\neg b)$
More generally, $P(a) = \sum_j P(X = x_i, Y = y_j)P(y_j)$

6.2 Bayesian Network

It is a data structure that represents dependencies among random variables in the form of a directed graph. Each node of the graph represents a random variable and an arrow from node X to node Y denotes that X is a parent of Y, i.e., the value of X directly affects the value of Y. Each node has a probability distribution that depends on its parents. Using these, we can determine any kind of joint probability. The nodes without parents have unconditional distribution.

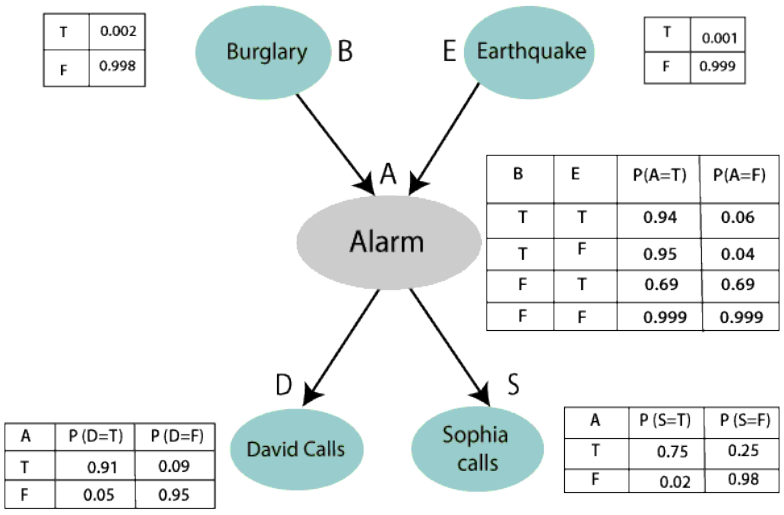

Figure 17: A Bayesian Network

6.3 Uncertainty in Time

In some cases, our variables may change with time. For example, the state of weather changes each day. The naive approach to make predictions in this case would be to store all previous state observations and use them to predict the next state, but this would be computationally inefficient. For example, the weather a year ago will not have a significant impact on tomorrow's whether and hence can be exempted from our calculations. This is formalized by the Markov assumption.

Markov Assumption: The current state depends only on a finite fixed number of previous states.

6.4 Markov Chain

It is a sequence of random variables, each following the Markov assumption. We use a transition model based on joint probabilities to construct the Markov chain.

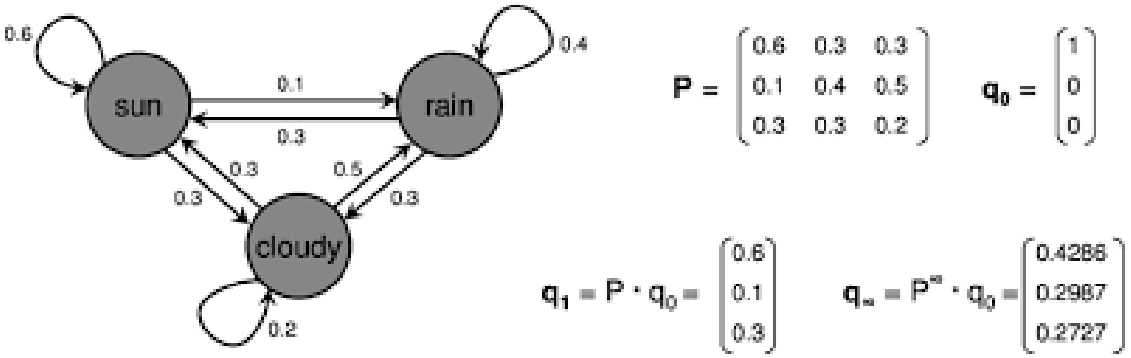

Figure 18: Markov chain

6.5 Hidden Markov Model

In this type of model, the system has 'hidden states' that generate some observed event, and our agent has access only to this event and not to the original hidden state. An example of this is in speech recognition; the words we speak are the hidden state, the sound we generate is the observable event which is given as input to the model. We generally use 'sensor models', that map the state to the observation using probability distributions. A modified version of the Markov assumption is the 'Sensor Markov assumption', which states that the evidence variable depends only on the underlying state.

7 Week 6: Optimization Problems

Optimization problems are essentially problems in which we try to choose the best possible option by minimizing or maximizing some function. Following are some of the methods used to solve optimization problems.

7.1 Local Search

This search algorithm maintains a single node and searches by moving to neighbouring nodes. It formulates search problems as a 'state-space landscape', where each vertical bar represents a state and the height of the bar represents the value of some function of the state. If we wish to maximize this function, we call it the **objective function** and if we wish to minimize it, we call is the **cost function**.

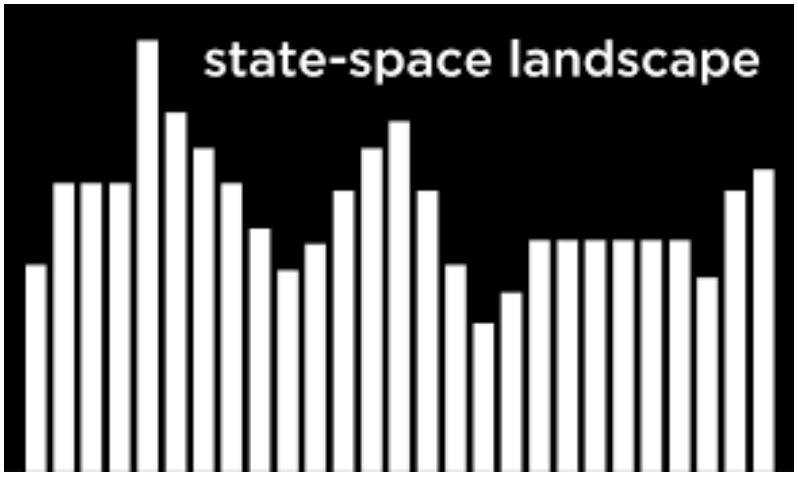

Figure 19: State-Space Landscape

7.1.1 Hill Climbing Algorithm

Let us consider the case where we are trying to maximize the objective function. We start at a random bar, check the bars on either side, and move to the highest one. We keep repeating this check until the current bar is higher than both its neighbours, and we choose this bar as our optimal state.

Pseudocode

```
function hillclimb(problem)
    curr = initial state
    repeat
        neighbour = Highest valued neighbour of curr
        if neighbour not better than curr
            return curr
        curr = neighbour
```

Limitations of Hill climb search

May not reach the globally optimal solution. Depending on where the algorithm starts, it could get stuck at a local maximum and never reach the global one. It may also get stuck in a flat

'plateau' like region comprising of several consecutive bars of the same height.

To overcome these issues, several variants of this algorithm have been introduced.

Hill Climbing Variants

1. **Steepest Ascent:** Our original naive version.
2. **Stochastic:** Chooses randomly from all the highest neighbours.
3. **First-Choice:** Chooses the first higher valued neighbour.
4. **Random Restart:** Hill climbs multiple times from randomly chosen starting points.
5. **Local Beam Search:** Chooses k highest neighbours and keeps track of them all.

7.1.2 Simulated Annealing

The name comes from a technique called 'annealing' where a piece of metal is heated to a high temperature to attain certain properties, and then left to cool to room temperature to 'settle down'. In the search algorithm, at early stages, the agent is in a 'high temperature state' and is more likely to accept unfavourable neighbours. Later on, it reaches a 'low temperature state' and is less likely to accept unfavourable neighbours.

Pseudocode

```
function sim_ann(Problem, time)
    curr = initial state
    for t from 1 to max:
        T = temperature(t)
        neighbour = Random neighbour of curr
        ΔE = By how much is the neighbour better than curr?
        if Δ > 0:
            curr = neighbour
        else:
            With probability e^(ΔE/t)
    return curr
```

7.2 Linear Programming

The goal of the problem is to minimize the cost function $c_1x_1 + c_2x_2 \cdots + c_nx_n$
with constraints of the form $a_1x_1 + a_2x_2 \cdots + a_nx_n \leq b$ or of the form $a_1x_1 + a_2x_2 \cdots + a_nx_n = b$ with bounds on each variable $l_i \leq x_i \leq u_i$.

7.3 Constraint Satisfaction Problem (CSP)

Variables are subject to various constraints (eg: Sudoku) and our goal is to find their values. We have the set of variables $x_1, x_2, \ldots x_n$ with domains for each variable $D_1, D_2 \ldots D_n$ and subject to constraints C.

Hard constraints are constraints that *must* be satisfied in the solution, while soft constraints involve some notion of *preference*.

Unary constraint: Involves only one variable.

Binary constraint: Involves exactly two variables.

Node Consistency: When all the values in a variable's domain satisfy all its unary constraints.

Arc Consistency: When all the values in a variable's domain satisfy all its binary constraints.

To make node X arc consistent with respect to node Y, we remove elements from the domain of X until every choice for X has a possible choice for Y.

Pseusocode to enforce arc consistency between two nodes X and Y

```
function revise(csp, X, Y)
    revised = False
    for x in X.domain:
        if no y in Y.domain satisfies the constraints for (x, y):
            delete x from X.domain
            revised = True
    return revised
```

Pseudocode to enforce arc consistency over the entire CSP

```
function AC-3(CSP)
    queue = All arcs in CSP
    while queue is not empty:
        (x, y) = pop an arc from the queue
        if revise(csp, x, y):
            if size of x.domain == 0:
                return False  No solution
            for z in x.neighbours
                if x == y:
                    continue
                Add (z, x) to queue
    return True
```

However, AC-3 is not always enough to solve problems. We can solve most problems by modelling CSP as a search problem.

CSP as a Search Problem

- Initial state = Empty assignment (No variables have been assigned values)
- Action = Add *variable* = *value* to the assignment
- Transition model = New assignment after adding variable
- Goal test = Check if all variables have been assigned values and all constraints are satisfied
- Path cost function: Irrelevant; All paths are of the same cost

Naive BFS and DFS algorithms are inefficient here. We use an improved search algorithm called Backtracking Search.

Backtracking Search

Variables are assigned values; If we ever get stuck and can't make any more progress, we backtrack to the last assignment and try out other alternatives.

Pseudocode

```
function backtrack(assignment, csp)
    if assignment is complete:
        return assignment
    var = select-some-unassigned-variable(assignment, csp)
    for value in domain-values(var, assignment, csp):
        checking if the value works out
        if value is consistent with assignment:
            add var = value to assignment
            result = backtrack(assignment, csp)
            if result ≠ failure:
                return result
            remove var = value from assignment
    return failure
```

We can minimize the possibility of failure by interleaving backtracking with the AC-3 algorithm. Every time we make a new assignment to a variable x, we can call AC-3, starting with a queue of all arcs (y, x) where y is a neighbour of x.

Further Optimization

1. Instead of randomly selecting a variable in the function `select-some-unassigned-variable`, we could use some heuristics.

   - **Minimum Remaining Values (MRV):** Selects the variable having the smallest domain.
   - **Degree Heuristic:** Selects the variable that has the highest degree (number of neighbours).

2. In the function `domain-values`, going in the default order may not be the most efficient choice. Instead, we could start from the 'most likely values'. These values are determined by another heuristic.

   Least constraining value heuristic: It returns variables ordered by the number of choices that are ruled out for its neighbours. We try out the least constraining values first, because they rule out the fewest options and thereby limit the solution space the least.

8 Week 7: Reinforcement Learning

For this week, I learnt from the YouTube course on reinforcement learning by David Silver [5].

8.1 Basic concepts

The RL agent is trained using the 'delayed feedback mechanism', which means it wants to maximize its expected cumulative reward and will sacrifice immediate reward if it can gain more long term reward.
We denote an observation by O_t, an action by A_t and reward by R_t.

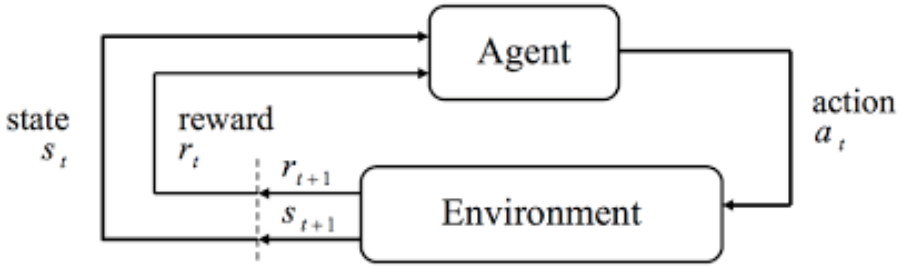

Figure 20: Basic RL model

History: It is a sequence of all the observable variables upto time t.

$$H_t = A_1, O_1, R_1, \dots A_t, O_t, R_t$$

The next action that the agent takes, and the next observation or reward that the environment returns depend on H_t.

However, it is inefficient to use the entire sequence H_t to determine the next event. Instead, we use a 'state' to determine events.

$$S_t = f(H_t)$$

8.2 Markov State/ Information State

A state S_t is Markov if and only if

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, S_2 \dots S_t$$

A Markov state is one that captures all the useful information from the history. The basic idea behind it is that "The future is independent of the past given the present".

8.3 Components of an RL agent

8.3.1 Policy

It is the behaviour function of the agent, which maps state to action. There are two broad categories of policies, Deterministic policy ($a = \pi(s)$ and Stochastic policy ($\pi(a|s) = P(A = a|S = s)$.

8.3.2 Value function

It is a prediction of the total expected future reward. It is a measure of how good each state or action is.

$$V_\pi(s) = E_\pi(R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} \dots | S_t = s)$$

Here γ is the 'discount factor', and represents how much we value future rewards compared to the present reward.

8.3.3 Model

It is the agent's representation of the environment, and helps predict what the environment will do next. There are two models within the RL agent's architecture;

1. **Transition model (P):** It predicts the next state (dynamics).

$$P^a_{ss'} = P(S' = s' | S = s, A = a)$$

2. **Reward model (R):** It predicts the immediate reward.

$$R^a_s = E(R | S = s, A = a)$$

8.4 Categories of RL Agents

1. **Value based:** Contains value function but no policy function. In this case the policy is implicit; to maximize the value function.
2. **Policy based:** Contains a policy function, but no value function.
3. **Actor critic:** Contains both a policy function and a value function.
4. **Model free:** Does not contain a model, but contains a policy function and/ or a value function.
5. **Model based:** Contains a model as well as a policy function and/ or a value function

8.5 Problems within RL

8.5.1 Learning and Planning

These are the two fundamental problems in sequential decision making

1. **Learning:** The environment is initially unknown (eg: A game playing agent starts off without knowing the rules of the game). The agent interacts with the environment and uses the information gained from these interactions to improve its policy.
2. **Planning:** The agent has a model of the environment (eg: A game playing agent knows the rules of the game from the beginning). The agent does internal computation based on the model without interacting with the environment and improves its policy based on this.

8.6 Markov Decision Process (MDP)

It describes the environment for reinforcement learning in the case where the environment is fully observable. Almost all RL problems can be modelled as MDPs.

We can define State Transition Matrix (P) using the state transition probabilities ($P_s s'$), which describes the STP from all states s to all successor states s'.

$$P = \begin{bmatrix} P_{11} & \dots & P_{1n} \\ . & & . \\ . & & . \\ . & & . \\ P_{n1} & \dots & P_{nn} \end{bmatrix}$$

The sum of each row in the matrix is 1.

8.6.1 Markov Process/ Chain

It is a memory-less random sequence of Markov states $s_1, s_2, \dots$.

It is represented as a tuple $\langle S, P \rangle$, where S is a finite set of Markov states and P is the STP matrix

8.6.2 Markov Reward Process (MRP)

It is a Markov chain with values.

It is represented as a tuple $\langle S, P, R, \gamma \rangle$, where γ is the discount factor and ranges from 0 to 1.

8.6.3 Return (G_t)

It is the total discounted reward from timestep t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The goal of reinforcement learning is to maximize G_t.

8.6.4 Reason for using γ

- Our model isn't perfect.
- Mathematically convenient.
- Avoids infinite returns in cyclic processes.
- Animal/ human behaviour prefers immediate rewards over long term ones.

If all sequences terminate, we can use undiscounted MRP ($\gamma = 1$).

8.6.5 Bellman equation for MRPs

The value function ($V(s_t)$) can be divided into the immediate reward (R_{t+1}) and the discounted value of the successor state ($\gamma V(S_{t+1})$). Hence we derive the equation

$$V(s) = E(R_{t+1} + \gamma V(s_{t+1} | s_t = s)$$

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

8.6.6 Markov Decision Process (MDP)

It is an MRP in which the agent can take decisions. It is an environment where all states are Markov.

It is represented as a tuple $\langle S, A, P, R, \gamma \rangle$ where A is a finite set of actions, and P and R now account for agent's actions as well.

In this case, the policy is a distribution over actions given states. It fully defines the agent's behaviour and depends only on the current state, not on the history. It is stationary (time independent).

Our definition of value function changes slightly in MDPs. We now define two value functions.

State-value function ($V_\pi(s)$): It is the expected return starting from state s and then following policy π.

$$V_\pi(s) = E_\pi(G_t|S_t = s)$$

Action-value function ($q_\pi(s, a)$): It is the expected return starting from s, taking action a and then following policy π.

$$q_\pi(s, a) = E_\pi(G_t|S_t = s, A_t = a)$$

8.6.7 Bellman Expectation Equation

Similar to the Bellman equation for MRPs, we can split the value functions as follows.

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

$$q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in s} P_{ss'}^a Y_\pi(s')$$

8.6.8 Optimal Value Function

It is the 'solution' of our MDP. It is defined as the maximum state-value function over all policies.

$$V_*(s) = max_\pi V_\pi(s)$$

8.6.9 Optimal Policy

We define a partial ordering over policies as $\pi \geq \pi_*$ if $V_\pi(s) \geq V_\pi^*(s), \forall s$.
For any MDP. there exists an optimal policy π_*. All π_* achieve the optimal state-value function (V_π^*) and optimal action-value function ($q_*(s, a)$). To find it, we maximize $q_*(s, a)$ over s.

8.6.10 Bellman Optimality Equation

It recursively relates the optimal value functions.

$$v_*(s) = max_a q_*(s, a)$$

$$q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in s} P_{ss'}^a Y_\pi(s')$$

8.7 Planning using Dynamic Programming

For prediction:
Input: MDP $\langle S, A, P, R, \gamma \rangle$, policy π
Output: Value function V_π

For control:
Input: MDP $\langle S, A, P, R, \gamma \rangle$
Output: Optimal value function V_*, optimal policy π_*

8.7.1 Policy Evaluation

It is used to evaluate policy π by finding V_π, using Bellman expectation equation iteratively. It generates the sequence $v_1, v_2, \ldots$ which ultimately converges to V_π.

$$V_{k+1}(s) = \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in s} P_{ss'}^a V_k(s')$$

8.7.2 Policy Iteration

It is used to repeatedly generate better policies, ultimately converging to π_*.
Given a policy π, we evaluate it using policy evaluation and then improve it by acting greedily with respect to V_π.

Although we will eventually converge to the optimal policy, sometimes the process could be too long, in which case we we could introduce some stopping condition, like ϵ convergence, or stopping after a fixed number of iterations.

8.7.3 Value Iteration

It is used to generate the optimal value function V_*. It works based on the principle of optimality.

Principle of Optimality: A policy $\pi(a|s)$ achieves optimal value from state s if and only if for any state s' reachable from s, π achieves optimal value from s'.

Value iteration follows the recursive process

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

8.8 Model Free Prediction

8.8.1 Monte Carlo Learning

The agent learns from complete episodes of experience, without bootstrapping. As the name suggests, there is no model, i.e., the agent has no knowledge of the transitions or rewards of the MDP. This method works only for episodic MDPs, where all episodes terminate.

Goal: To learn V_π from episodes of experience under policy π.
Returns: $G_t = Rt + 1 + \gamma R_{t+2} \cdots + \gamma^{T-1} R_T$

Value function, $V_\pi(s) = E_\pi(G_t|S_t = s)$.
Instead of expected return, MC uses empirical mean return.

First visit Monte Carlo Policy Evaluation

To evaluate state s across all episodes,
In an episode, when s is visited for the first time at t,
Increment counter, $N(s) = N(s) + 1$
Increment total return, $S(s) = S(s) + G_t$
Value = Mean return, $V(s) + \frac{S(s)}{N(s)}$

Every visit Monte Carlo Policy Evaluation

It is the same as first visit MC, but we increment the counters and total return **every** time state s is visited, not just the first time.

8.8.2 Temporal Difference Learning (TD)

The agent learns from incomplete episodes of experience, via bootstrapping. Like MC, there is no model. This method works by updating guesses towards new guesses. An advantage of this approach is that it works in non-terminating environments as well.

8.8.3 Monte Carlo vs Temporal Difference Learning

Both have the same goal- To learn V_π from experience under π.

MC updates V towards the **actual** return, while TD updates it towards the **estimated** return.

MC starts from a state, goes all the way till the final outcome and then comes back and updates the state according to the outcome, while TD learns as it goes without backtracking.

There is a trade-off between bias and variance; MC has high variance and no bias whereas TD has low variance and some bias.

While MC results in good convergence, TD is usually more efficient.

TD works by exploiting the Markov property (by implicitly building and solving for MDP structure) and is usually more efficient in Markov environments. MC doesn't exploit the Markov property and is usually more efficient in non-Markov environments.

8.8.4 TD(λ) Approach

It is regarded as a mix of Monte Carlo and Temporal Difference approaches. It works by checking n steps ahead and updating the guess. The equation for n step TD learning is

$$V(S_t) = V(S_t) + \alpha(G_t^n - V(S_t))$$

The ideal value of n depends on the use case. To find a good n value, we average n step returns over different values of n.

9 Week 8: Natural Language Processing

For this week, I mainly used the Coursera course by Andrew NG on sequence models [6] and other online resources. I worked on building three models, one a simple implementation of a neural network binary classifier, one to get skip-gram word embeddings, and one to perform sentiment analysis using recurrent neural networks.

9.1 Basic Terminology

- **Corpus:** It is the body of text given as input to the model.
- **Tokenization:** It is the process of splitting the corpus into smaller, more manageable chunks.
- **Stemming:** It is the process of removing prefixes and suffixes to get the root word. However, the word it generates may not be meaningful.
- **Lemmatization:** It converts words into their 'basic' form, and always returns a meaningful word. For example, 'better' gets converted to 'good'.
- **Stop words:** These are words that do not contribute meaning to the sentence and can be ignored while processing. Some example stop words are 'the', 'and' and 'in'.
- **n-grams:** They are a method of representing text as sequences of n continuous words.

9.2 Word Embeddings

In all the neural networks covered in Week 2, the model took only numerical inputs. This means that words cannot be directly input to neural networks, and we need some way to represent them in numerical form. Word embeddings are a way of representing words as vectors, where words with similar meanings are grouped close together in the vector space.

9.2.1 One-Hot Encoding

In this method of embedding, we simply represent each word as a vector of size same as the number of words in the vocabulary, where each index corresponds to a word. The word the vector represents is a 1 in the one-hot encoding, and all the others are 0. As expected, this method can be highly inefficient when our vocabulary is large, and we need a way of 'compressing' these vectors into smaller ones.

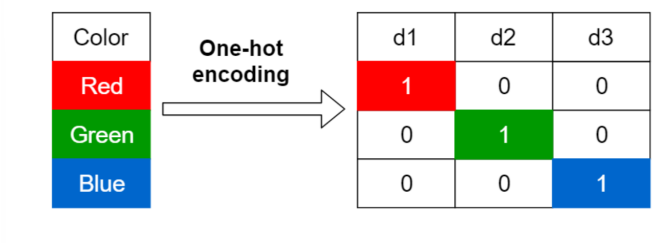

Figure 21: One-Hot Encoding

9.2.2 Skip-Gram Model

This is a more efficient form of word embeddings, derived from one-hot encoded vectors. We begin by preparing n-gram models for all the sentences in the corpus. This representation tells us how a word is used in context with other words in the vocabulary. The neural network used to get these embeddings has two layers- one linear input layer and another layer that predicts the neighbouring words.

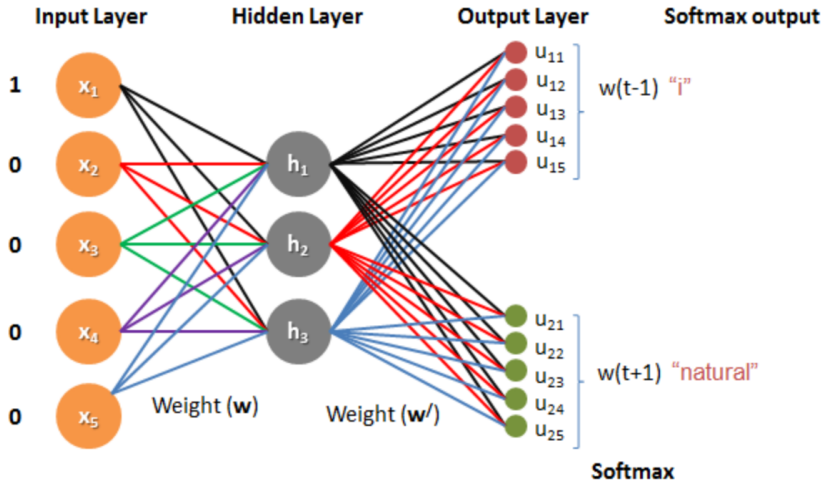

Figure 22: Neural Network for Skip-Gram model

A method to compare the 'closeness' of different words is using their **cosine similarity**. It can be physically thought of as the cosine of the angle between the two word vectors. So a value close to 1 means the words carry similar meaning, and a value close to 0 means the words bear little relation to one another.

9.3 Sentiment Analysis

This is an important application of NLP, where we train a model to predict the sentiment (positive, negative, neutral etc) underlying a body of text. A common use case of this is to analyse the reception of a movie based on viewer reviews.

References

[1] [Online]. Available: https://www.w3schools.com/python/

[2] [Online]. Available: https://github.com/Pierian-Data/Complete-Python-3-Bootcamp

[3] [Online]. Available: https://www.coursera.org/learn/neural-networks-deep-learning?specialization=deep-learning

[4] [Online]. Available: https://pll.harvard.edu/course/cs50s-introduction-artificial-intelligence-python?delta=1

[5] [Online]. Available: https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver

[6] [Online]. Available: https://www.coursera.org/learn/nlp-sequence-models