During this project, we focused on **Reinforcement Learning (RL)**, using a reference book that covered key concepts such as **agents, environments, rewards, policies, and value-evaluated functions**. We learned how agents make decisions and how to encourage them to make the right ones. This included designing environments, training agents to navigate them, and understanding how they learn from their surroundings.

As part of decision-making processes, we studied **Markov Decision Processes (MDPs), policy evaluation, policy improvement, the E-Greedy algorithm, and different methods for selecting optimal actions**. These concepts helped in refining learning techniques and improving decision-making accuracy.

Throughout the project, we implemented our learning by solving well-known problems such as **Unbundled Problems, Frozen Lake Problems, Slippery Wall Problems, Jack's Car Rental Problem, and the Gambler's Puzzle**. Finally, we applied **Reinforcement Networks (RN) to solve a 15-puzzle problem**. Each week, we tackled a different problem, applying a new method to train agents in understanding their environment and finding solutions.

Week 0

Week 0 was dedicated to learning the **basics of Python**, along with essential libraries such as **NumPy, Pandas, and Matplotlib**. This turned out to be the most difficult week for me. For some reason, every time I tried installing **VSCode** and running **Jupyter Notebook files**, my computer kept crashing. Setting up the development environment was unexpectedly challenging and took considerable effort to resolve.

Week 1

During **Week 1**, we were tasked with designing an **agent and an environment** where the agent was presented with **n different options**, each associated with a **numerical reward**. These rewards followed a **bell curve distribution** and had a **mean reward value**. The agent's goal was to determine the **optimal choice** over multiple trials.

There were three main approaches to solving this problem:

1. **Pure Exploration** – The agent would try every option exactly **n** times, choosing each option at least once. After collecting reward values for all options, it would determine the **optimal choice** and stick with it indefinitely.

2. **Exploration with a Fixed Probability** – In this approach, the agent would explore **only a certain percentage of the time**. Whenever the agent chose to explore, it would randomly select an option and observe the reward. This method allowed the agent to gradually **develop a better understanding of the reward distribution** and make an informed decision about the best option.

3. **Optimistic Initial Values** – Here, the reward values for all **n options** were set **higher than their actual mean reward values**. This encouraged the agent to **explore more options early on** before settling on the optimal choice. By overestimating the rewards initially, the agent was motivated to test all available choices multiple times before favoring the best one.

Each of these methods provided different insights into how **agents balance exploration and exploitation** when making decisions.

None of these methods would work effectively in a **changing environment**, where the **mean reward value keeps shifting**. The third method, **Optimistic Initial Values**, would fail entirely because once the agent initially explores and settles on a choice, it would not adapt to changes in the environment.

Even the **exploration-based method** would not perform efficiently in a dynamic setting. This is because it **assigns too much importance to earlier rewards**, which may no longer be relevant as the environment evolves. In a changing environment, the **most recent results matter more**, so relying heavily on past rewards would prevent the agent from making optimal decisions in the long run

We also learned how to **navigate a changing environment**, though I don't recall which specific week it was. In this, we studied a formula that helps adapt decision-making in **dynamic environments**.

The formula is:

$$Q_{n+1} \quad = \quad (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha (1-\alpha)^{n-i} R_i .$$

This formula ensures that **recent decisions have a greater influence** on the associated reward compared to older ones. In a **changing environment**, where the **mean reward keeps fluctuating**, this approach helps keep the reward estimates more **aligned with the current state** of the environment rather than being overly influenced by outdated rewards.

In **Week 2**

we studied a well-known **decision-making process** called **Markov Decision Processes (MDPs)**. In **Reinforcement Learning (RL)**, we generally assume that all environments operate under **MDPs**, where future states depend only on the **current state** and **action taken**, not on past history.

To apply this concept, we worked with two different environments:

1. **Bandit Walk Environment** – This was a simple **grid-based environment** with **three steps**. The agent **always started from the middle step** and could move **left or right**. A **reward of 1** was given only if the agent reached the **rightmost cell**, while all other moves resulted in a reward of **0**. However, the agent had no prior knowledge of these rules and had to discover them through learning.
2. **Slippery Walk Environment** – This introduced **uncertainty in movement**. Even if the agent chose to move **left**, there was a probability that it might still end up moving **right**, depending on the **slipperiness factor**. Just like in the Bandit Walk environment, the agent was unaware of this and had to **learn the behavior of the environment** through trial and error.
3. **Frozen Lake Environment** – This was a more **complex grid-based environment** with a **larger state space**. We implemented this environment using **Python**, allowing us to simulate decision-making in a more challenging scenario where the agent had to **navigate across a frozen surface** while avoiding pitfalls and reaching a goal.

Each of these environments helped in understanding how **agents learn to make decisions under uncertainty**, a fundamental concept in **Markov Decision Processes**.

Week 3

In **Week 3**, we studied the chapter on **Dynamic Programming** and solved two problems: **Jack's Car Rental Problem** and **Gambler's Problem**. The key concepts we focused on were **policy evaluation, policy improvement, policy iteration, and value iteration**.

1. **Policy Evaluation** – This involved computing the **state-value function** VπV^\piVπ for an arbitrary policy π\piπ. This step helps in assessing how good a given policy is.

Formula:
$$V_{k+1}(s) = \max_a \sum_{s',r} P(s', r|s, a) \left[r + \gamma V_k(s')\right].$$

2. **Policy Improvement** – The purpose of computing the **value function** for a policy is to use it to find a **better policy**. This is done using the **Policy Improvement Theorem**, which ensures that updating policies iteratively leads to better decision-making.

Formula:
$$\pi'(s) = \arg\max_a \sum_{s',r} P(s', r|s, a) \left[r + \gamma V(s')\right].$$

3. **Policy Iteration** – This process involves continuously improving policies in a **chain-like manner**. Each iteration results in a **monotonically improving sequence** of policies and value functions. Since a **finite Markov Decision Process (MDP)** has only a finite number of deterministic policies, this iteration **must eventually converge** to an **optimal policy** and its corresponding **optimal value function** in a finite number of steps.

4. **Value Iteration** – One drawback of **policy iteration** is that each iteration involves **policy evaluation**, which can be computationally expensive and time-consuming. To address this, **value iteration** modifies the process by combining **policy improvement** with a **truncated version of policy evaluation**. This allows convergence to the optimal solution while reducing computational cost.

Formula:
$$V_{k+1}(s) = \max_a \sum_{s',r} P(s', r|s, a) \left[r + \gamma V_k(s')\right].$$

Week 4

In **Week 4**, we applied everything we had learned so far to create an **agent that could solve the 15-puzzle problem**.

The **15-puzzle** consists of a **4×4 board** with **15 numbered tiles** and **one empty space**. The goal is to arrange the tiles in **increasing numerical order** by sliding them into the empty space. Initially, the tiles are **scrambled**, making it a challenging problem to solve optimally.

At first, I attempted to find an **optimal strategy** to solve the puzzle directly. However, this turned out to be a **bad idea**—my code **wouldn't run** properly, and even after multiple modifications, it still failed to solve the puzzle. Even when it did run, the solution was either **incorrect or inefficient**.

Eventually, I abandoned this approach and went back to the method suggested in the **reference article**, which was to **solve the puzzle row by row**. This approach made the problem significantly easier and, interestingly, **mirrored the way humans solve it manually**.

The row-by-row method worked as follows:

- First, I **focused only on the first row** and arranged the numbers **1 to 4** in their correct positions.
- Next, I moved to the **second row**, isolating **only the numbers 5, 6, 7, and 8** while blocking out the rest of the board. The goal was to position them correctly while keeping the first row intact.
- This process continued for the remaining rows until the entire board was solved.

This structured approach **simplified the problem significantly**, and I was finally able to get the agent to solve the puzzle. Completing this task felt **really satisfying**, as I could see firsthand how **breaking down a complex problem into smaller, manageable steps** made a huge difference in achieving a working solution.

Conclusion

This project taught me **a lot**—and when I say **a lot**, I truly mean it. I learned about **GitHub, VS Code, Python, NumPy, Matplotlib, and Pandas**, among many other things. It was a deep dive into **machine learning**, and I was both **surprised but not shocked** to realize that **machine learning closely mimics how humans learn**. The only difference is that **machines have built-in calculators**, while humans don't.

Beyond coding, this project exposed me to a **much bigger picture**. I discovered the vast number of **articles, YouTube channels** that exist for learning these concepts**, and open-source projects on GitHub**. Before this, I had no idea how much **open-source collaboration** drives the field.

I also realized that **VS Code is actually quite good**—once it's properly set up, it's much smoother than **Xubuntu** for coding. But one of the biggest lessons I learned was **the importance of writing the environment itself**. Before starting this project, I assumed my focus would be on **making the agent work**—tuning its decision-making, optimizing learning, and so on. But I soon realized that **designing the environment is just as crucial** because the agent's learning depends entirely on how well the environment is defined.

Most importantly, I gained a solid understanding of **how machines learn**, which was the entire goal of this project. Along the way, I also learned **how environments function** and why they play a critical role in **reinforcement learning**.