Reinforcement Learning in Video Games

Valentí Torrents Vila

May 26, 2024

Resum– Resum del projecte, màxim 10 línies. ....

Paraules clau– Paraules clau del treball, màxim 2 línies . ...

Abstract– Versió en anglès del resum . ...

Keywords– Versió en anglès de les paraules clau. ....

1 INTRODUCTION

1.1 Preliminary Information

REINFORCEMENT learning, or RL, is a type of machine learning that revolves around how an intelligent agent should behave in an environment in order to achieve an specific goal. Unlike other machine learning paradigms, like supervised learning or unsupervised learning, RL's goal is to generate an intelligent agent that learns on its own while interacting with a dynamic environment. Through trial and error, the agent must chose which of the possible actions yield the most reward (both immediate and long-term), until achieving the necessary "knowledge" to go through the environment without any problems and solving it.

1.2 Objectives and expected results

The goal of this project is to dive into the world of Reinforcement Learning while using the Gym library effectively. Going from the fundamentals of RL, the project aims to understand and implement different RL algorithms. Starting with a quick introduction to this world, and going through

- E-mail: valentovi55@gmail.com
- Specialisation: Computation
- Work tutored by: Jordi Casas Roma
- Year 2023/2024

the basics with different tabular methods, our final objective is to go deep into the most complex methods applicable to video games. The following objectives mark the trajectory of this project:

1. Develop a foundational understanding of Reinforcement Learning and proper use of the Gym library.
2. Study and implement tabular methods such as Dynaim Programming, Monte Carlo, or Q-learning to create an agent capable of solving simple environments.
3. Study and comprehend some non-tabular methods, like policy-based methods and Deep Q Networks (DQN).
4. Apply the acquired knowledge to solve more complex and sophisticated environments, enhancing the agents' capabilities.
5. Generate a series of agents, each more complex, capable of achieving desired results in various games, while documenting findings and utilizing visualization tools to ensure a thorough exploration of Reinforcement Learning in video games.

1.3 Metodology

I have chosen to work using the Agile methodology for my project, working in iterative sprints, each lasting two weeks, which will be shown in the Gantt Diagram in figure 8. Using this methodology allows continual reassessment and adjustment if necessary since reinforcement learning is a dynamic field and it will need some adaptability. The tasks shown in the Gantt Diagram will be accomplished in sprints of up to two weeks, depending on the workload. Through the regular review sessions with my tutor, we will check the gradual progression of the project, and provide checkpoints for evaluation, adaptation, and resolution of new obstacles as they appear.

1.4 Planning

The project planning is visualized through a Gantt Diagram in figure 8. This will be a dynamic tool to help the development of the project. At first it shows the main objectives and milestones necessary and, over time, using the Agile methodology, the Gantt diagram will show more, depending on the obstacles faced and the necessary changes that appear in the process.

2 State of the art

Having talked about the basics of this project, we can now get into a more detailed explanation of what is the State of the art like, and how our first implementations of reinforcement learning methods has been like.

2.1 Introduction to RL

As we have just explained in the preliminary information, RL stands at an in-between point of artificial intelligence and decision-making. Thanks to that, it offers a powerful structure for creating and teaching agents to navigate through sequential decisions in dynamic environments. Simply put, RL revolves around the interaction between an agent and a specific environment. Through observations, this agent can perceive its environment and, based on its current state, selects one action or another. Furthermore, and depending on the specifics of the environment, the agent receives feedback in the form of rewards. Depending on the actions taken by the agent in each situation, these rewards can vary, indicating the desirability of said actions. Through these rewards, the agent is guided towards learning optimal strategies, which let it maximize these rewards over time. To further understand how Reinforcement Learning works, we have to talk about its components:

- The environment is, basically, the problem space with which the agent interacts. All the states, possible actions, rewards and rules that the agent has to abide come from the environment. Its dynamics dictate how the agent's actions modify future states and rewards, and even though we usually see dynamic environments in RL, there can also be static ones.
- The agent is the principal entity in RL, and the one that takes the decisions, earns the rewards, and goes through the environment. By navigating through the environment, and recieving different inputs from it, the agent learns to take better decisions. Its goal is to follow a behaviour (policy) which maximizes the rewards over time, solving then the task at hand.
- Actions are the choices available to the agent at each step within the environment. Depending on which action the agent takes, its surroundings will be influenced one way or another. Actions can be discrete(finite options) or continuous.
- The states repressent the current position or configuration of the environment. Depending on the moment and the position of the agent the state changes, and with it all the relevant information used by the agent to make decisions.
- Rewards, as we said before, are provided by the environment to the agent, and are signals that evaluate its actions. They can present inmediate or delayed feedback, and are the resposibles for the changes in the agent's behaviour over time, since its objective is to maximize these rewards in order to solve the environment.
- Policies define the agent's behavior by relating all states to actions. They mark the strategy followed by the agent to achieve its objectives. These policies can be deterministic, where each state has a specific action, or stochastic, where all actions have a probability of happening depending on the state and policy parameters. The objective of all Reinforcement Learning algorithms is to achieve the optimal policy v_π that maximizes long-term rewards, and leads the agent to solving the environment.

By interacting iteratively with its environment, and learning from its experience, RL algorithms generates agents that learn autonomously how to behave, solving different tasks, from video games to controlling robots. By learning through trial and error, and without supervision, RL ends up being perfectly adequate to solving scenarios where the agent must adapt to dynamic and complex environments.

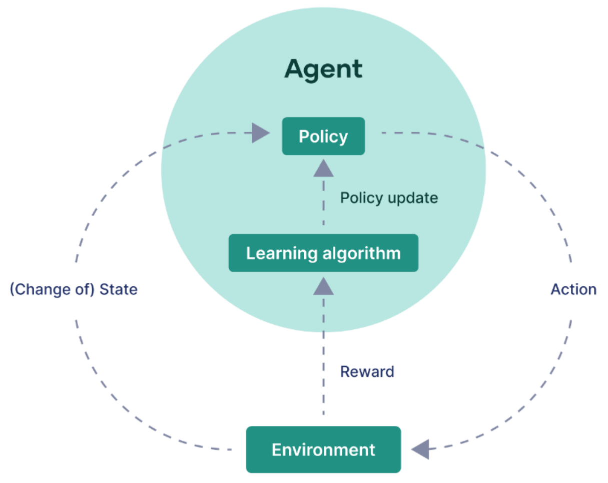


Fig. 1: RL General Framework

2.2 OpenAI Gymnasium

OpenAI Gym serves as a versatile platform for the development and evaluation of reinforcement learning algorithms. Offering a diverse array of environments spanning from classic control problems to complex simulated scenarios, Gym provides a standardized interface for interacting with different tasks. Each environment in Gym is encapsulated as a Markov decision process (MDP), allowing agents to interact with states, take actions, receive rewards, and transition between states based on probabilistic dynamics. With its user-friendly API and extensive documentation, OpenAI Gym empowers researchers and practitioners to prototype, benchmark, and iterate on various reinforcement learning techniques with ease. Moreover, Gym's compatibility with popular RL libraries and frameworks fosters collaboration and accelerates the advancement of RL research and applications.
`https://gymnasium.farama.org/`

2.3 RL Methods

In the current state of reinforcement learning (RL), researchers employ tabular and non-tabular methods. Tabular methods, like dynamic programming and Monte Carlo, excel in small-scale tasks with discrete state and action spaces. They offer theoretical guarantees but struggle with larger, continuous spaces. Non-tabular methods, including deep reinforcement learning (DRL), leverage function approximation, particularly neural networks, to handle complex, high-dimensional environments. DRL algorithms like deep Q-networks (DQN) and policy gradients have demonstrated success in diverse applications, from robotics to gaming. Both approaches complement each other, with tabular methods providing theoretical foundations and non-tabular methods offering scalability and adaptability to real-world problems.

3 Tabular Methods

To get to understand the basics of RL we first have to talk about the tabular methods. These are the simplest ones in RL, since they store a specific action for each state in a table-like structure. The tabular methods we will talk about are Dynamic Programming, Monte Carlo methods, and Q-learning. To do so, we will work with finite Markov Decission Processes(MDP), trying to estimate value functions, which are functions of states that tell use how positive is for the agent to be in a specific state.

3.1 Markov Decission Processes

A Markov Decission Process(MDP) is a mathematical model used in RL to make sequential decisions in a stochastic environment. Basically, an MDP describes a system(or environment) where an agent makes decisions which results are subject to uncertainty. Just like the parts of a RL environment, a MDP contains states, actions and rewards. However, a MDP also contains a Transition Model, which describes the transition probabilities from one state to another after taking a certain action. In order to solve a finite MDP, we can use different tabular methods, which we are going to talk about now.

3.2 Dynamic Programming

Dynamic Programming(DP) is used to find the optimal policies to follow in order to solve a finite MDP. In a finite MDP, states, actions, and rewards are finite, and their dynamics are represented by transition probabilities. In order to achieve a good policy in RL in general, we use value functions. With DP we can compute these value functions by satisfying the Bellman optimality equations.
The Bellman Optimality Equations express the principle of optimality, that an optimal policy can be decomposed into smaller subproblems, each of which is solved optimally. In the field of MDPs, the Bellman Optimality Equations are:

$$v_*(s) = \max_a \sum_{s'.r} p(s', r \,|\, s, a) \Big[r + \gamma v_*(s') \Big],$$

Fig. 2: Equation for State-Value Function (V)

$$q_*(s, a) = \sum_{s',r} p(s', r \,|\, s, a) \Big[r + \gamma \max_{a'} q_*(s', a') \Big]$$

Fig. 3: Equation for Action-Value Function (Q)

In our project, we employed policy iteration to find this optimal value function. First, we initialized a policy, and then evaluated and improved it iteratively until it converged. To keep evaluating the policy we had, we estimated the value function with the current policy using the Bellman Equations. Afterwards, we selected the actions that maximized the expected returns in order to update the policy. After repeating this 2 steps several times and observing no further improvements, we had the optimized value function and policy that let us guide the decision-making through the MDP.

3.3 Monte Carlo Methods

Even though Monte Carlo and DP are both tabular methods, Monte Carlo methods do not require a model of the environment to operate. Instead, they rely on trial and error, sampling trajectories by interacting with the environment.
In our Monte Carlo policy evaluation process, we began by executing episodes according to the current policy, collecting state-action-reward trajectories. After each episode, we calculated returns for each visited state, providing estimates of their values. These returns were then averaged over multiple episodes to update the value function incrementally. Optionally, we improved the policy based on the estimated values, possibly selecting actions greedily to maximize returns. This iterative refinement continued over multiple episodes until convergence, where the estimated value function approached the optimal one as the number of episodes increased.

3.4 Q-learning

Now that we have talked about both Monte Carlo and DP methods, we can get to understand temporal-difference(TD) learning, one of the most important methods of RL. TD is a mix of DP and Monte Carlo ideas, since it can learn directly from experience like Monte Carlo, and also update estimates from past estimates without having to wait for the final outcome, like DP.
From all the different TD methods, we just have worked with Off-Policy Q-learning, because it presents a different logic from the rest of the tabular methods. In Q-learning, the agent maintains an estimate of the value of each state-action pair represented by the action-value function Q(s, a). This function quantifies the expected cumulative reward that the agent will receive by taking action a in state s, and then following the optimal policy. With Q-learning, we aim to iteratively improve this estimate until it converges to the optimal action-value function. However, in off-policy Q-learning, we find that the agent learns from experiences generated by following a different policy than the one being learned.
Off-policy Q-learning allows agents to learn from past experiences generated under different policies, leading to faster learning. It effectively balances exploration (trying new actions) and exploitation (leveraging known information) in complex environments. This approach is versatile, enabling both policy evaluation and improvement, and benefits from experience replay to stabilize learning. Additionally, it's suitable for batch learning settings, where agents learn from fixed datasets, making it practical in scenarios where online exploration is challenging or costly.

3.5 Tabular Methods testing and results

To further understand the algorithms behind all three tabular methods we have talked about, we applied them to the Frozen Lake environment from the Gymnasium library.

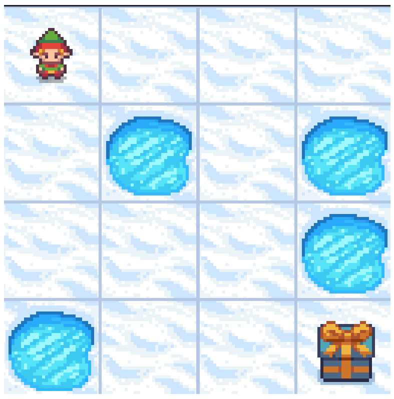


Fig. 4: Gymnasium Frozen Lake 4x4

With DP, we followed this pseudocode to achieve the optimal policy:

```
Algorithm 1: Policy Evaluation
Data: $\pi$, the policy to be evaluated,
A threshold $\theta > 0$ accuracy of estimation,
$V(s)$ initialized arbitrarily.
while $\Delta > \theta$ do
    $\Delta \leftarrow 0$
    foreach $s \in S$ do
        $v \leftarrow V(s)$
        $V(s) \leftarrow$
          $\sum_a \pi(a|s) \sum_s', rp(s', r|s, a)[r + \gamma V(s')]$
        $\Delta \leftarrow max(\Delta, |v - V(s)|$
```

The policy generated was
$\pi = [0, 3, 3, 3, 0, 0, 0, 0, 3, 1, 0, 0, 0, 2, 1, 0]$, with which the agent got to the end 82% of the time. If we used the 8x8 Frozen Lake environment, this percentage went to a 100%, since there was a possible route with no chances of failing.

AQUI PARLEM DE MONTECARLO

For Q-learning, the pseudocode was as follows:

```
Algorithm 2: Q-learning Off-policy
Data: Step size $\alpha \in (0, 1]$,
$\epsilon > 0$
$Q(s, a)$ for all $s \in S^+, a \in A(s)$ initialized
arbitrarily.
foreach episode do
    Initialize $S$
    foreach step of episode do
        Choose $A$ from $S$ using policy derived from
          Q($\epsilon$-greedy)
        Take action $A$, observe $R, S'$
        $Q(S, A) \leftarrow$
          $Q(S, A) + \alpha[R + \gamma max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$
    until $S$ is terminal
```

The policy generated was the same as in DP, making them equally efficient at solving the 4x4 Frozen Lake environment.

4 Approximate Solution Methods

Now that we have understood the basics of RL, we can dig into a more complex field. Unlike when we were working with tabular methods, most of the state spaces we will be working with RL will be extremely complex and large. That is why we cannot aim to find an optimal policy v_π, but instead to find an approximate solution. To do so, the main tool we will be using is Generalization. Through generalizing the problems we face, we treat them like similar problems we have already faced, thus achieving proper results without the need of dedicating enormous quantities of time and data. In this second section of the project, we will be talking about different Approximate Solution Methods and how to implement them in different environments.

4.1 On-Policy Prediction

We begin this chapter focusing on the utility of function approximation to estimate the state-value function from on-policy data. This means that, through using a known non-optimal policy π, we can estimate the state-value function v_π. The main difference we will work with from now on, is that instead of representing v_π as a table, we will represent it as a parametrized functional form with a weight vector $w \in R^d$. Now, instead of using any of the previous formulas to calculate the value of a state s, we will approximate it with the weight vector as $\hat{v}(s, w) \approx v^\pi(s)$. One of the cases in which we can use $\hat{v}$ may be in a neural network, where $\hat{v}$ is the function computed, using w as the vector of connection weights in through all the layers of the network. Also, we can use w to define the split points of a decission tree which computes the function $\hat{v}$.
One thing we have to keep in mind for future applications of these methods, is that the dimensionality of w may be way less than the number of states. What this means is that unlike in tabular methods, where we changed only the value of the state we updated, every time we update a state, the values of other states will change too, due to generalization between them. This generalization provides a more powerful learning potential, but also a more difficult management and understanding.
This chapter sets the foundation for understanding on-policy approximations in reinforcement learning, highlighting their theoretical underpinnings and practical implications in both fully and partially observable environments.

4.2 Value-function Approximation

Up until now, when we updated a specific state, we only shifted a tiny bit the overall behaviour of our agent towards the desirable policy, by modifying only the estimated value of the state we updated. Each of these updates is represented by $s \mapsto u$, where s is the state updated and u is the update target where s is shifting towards. Now, with generalization, the updates at s also change the estimated values of many other states. Machine learning methods that do this are called $supervisedlearningmethods$, and when the outputs are numbers, we call them $functionapproximation$. To generate an estimated value function using these methods, we take the $s \mapsto u$ of each update as training examples. By doing this, we enable these methods to be treated like any other learning method, and can now apply some of their algorithms. However, due to the necessity of having interactions and live updates, we need methods able to handle non stationary target functions.

4.3 PREDICTION OBJECTIVE

Through all this chapter, we have talked about the weight vector w but, how can we choose a proper w?
In tabular methods, the learned value function could end up being the true value function. However, here we cannot. Changing one state affects others, so we will never reach the correct value for all of them at the same time. That is why we need to give more importance to some of the states, specifying a distribution $\mu(s) \geq 0, \sum_s \mu(s) = 1$, which represents how much we care about the error in each state. The error here is calculated with the difference between the approximate value $\hat{v}(s, w)$ and the true value $v_\pi(s)$. Knowing all of this, we can now create an objective function, $\overline{VE}(w)$, which we will want to minimize in order to obtain the best w.

$$\overline{\mathrm{VE}}(\mathbf{w}) \doteq \sum_{s\in\mathcal{S}} \mu(s)\Big[v_\pi(s) - \hat{v}(s,\mathbf{w})\Big]^2$$

Fig. 5: Mean Squared Value Error

At first, this formula seems too complicated to apply, since we are summing through an exponentially big space (μ), and we are working with the true value v_π which we don't know. However, there are some algorithms with which we can approximately optimize it.

4.4 Stochastic Gradient Descent

In order to approximately optimize what we just talked about, we can use Stochastic Gradient Descent, also known as SGD. To work with SGD, we need to assume a few things:
Since we will be updating w as we bounce through the state space according to the on-policy distribution, all states must be visited in proportion to μ, which is what we have chosen from the on-policy distribution. Furthermore, our value function $\hat{v}(s, w)$ has to be differentiable with respect to w, necessary to calculate the gradient. And, as we said before, there needs to be a surrogate for $v_\pi(S_t)$, which is U_t.
With all this, SGD provides us with an update rule to apply to w for each visit to a state.

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t)$$

Fig. 6: SGD update rule

Where α is the step-size parameter, and $\nabla\hat{v}(S_t, w)$ is the gradient. Following this rule, SGD methods adjust w by a small ammount in the direction that would reduce most the error in that specific visit to that state.
Even though it may not seem obvious at first, it is necessary to carefully select which is the value of α. If the step on a

certain direction was too big, we could end up with a value function with zero error in a certain state, but an abysmal error in others. Taking small steps, and adjusting w little by little, will let SGD methods converge to a local optimum.
We now focus on the target U_t. If the expected value of U_t in a state S_t equals the true value $v_\pi(s)$, we call U_t unbiased, and w will undeniably converge into a local optimum of $\overline{VE}$.

4.4.1 Examples of Gradient Descend methods

Monte Carlo algorithm requires us to generate the states through interaction with the environment, following a policy π. If U_t is created by following π, the target of Monte Carlo ($U_t \to G_t$) has to be an unbiased approximation of $v_\pi(S_t)$. That is why, if we apply a gradient descent on Monte Carlo, it is going to find a locally optimal solution. However, having to wait until the end of each episode can make the learning slower. Therefore, we can contemplate other alternatives, like bootstrapping.
When bootstrapping, rather than waiting until the end of an episode to make updates based on the true return, we update at each time step using the current estimates of future returns. This means that bootstrapping targets, such as n-step returns or DP target, all are biased. Up until now, the target U_t has been independent of w_t. However, with bootstrapping it isn't the case, which means that the update rule is not a true gradient step anymore but a Semi-gradient, which makes guaranteing convergence impossible. In spite of its complications, we can still work with semi-gradient methods, such as semi-gradient TD(0), which uses $U_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, w)$ as its target. Using this, the update rule, looks like this:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$$

Fig. 7: Semi-gradient TD

And as we saw in the previous chapter, where TD methods like Q-learning were faster than Monte Carlo, here we can say the same. (POTSER POSAR REFERENCIA A QUAN FACI LES PROVES).

4.5 Linear Methods

In reinforcement learning, linear methods for function approximation are widely used due to their simplicity and efficiency. Since we have talked about how Gradient Monte-Carlo and SGTD work, we will now see how linear methods combined with these methods work. In linear function approximation, the value function, which estimates the expected return of being in a state or taking an action, is represented as a linear combination of features. Suppose we have a state s represented by a feature vector $\phi(s)$; the value function $V(s)$ is then approximated as $\hat{V}(s, \mathbf{w}) = \mathbf{w}^T\phi(s)$, where $\mathbf{w}$ is a vector of weights. Similarly, for a state-action value function $Q(s, a)$, the approximation takes the form $\hat{Q}(s, a, \mathbf{w}) = \mathbf{w}^T\phi(s, a)$, with $\phi(s, a)$ being the feature vector for the state-action pair.

The parameters $\mathbf{w}$ are updated using gradient-based methods, ensuring the function approximations improve over time. In Monte Carlo methods, the weights are updated based on complete episodes. After an episode ends, the return G_t for each state s_t in the episode is used to adjust the weights: $\mathbf{w} \leftarrow \mathbf{w} + \alpha(G_t - \hat{V}(s_t, \mathbf{w}))\phi(s_t)$, where α is the learning rate. This approach, while straightforward, requires waiting until the end of an episode to make updates, which can be slow.

In contrast, gradient TD methods such as TD(0) update the weights incrementally after each step, making them more efficient for online learning. The TD error δ is calculated as $\delta = R_{t+1} + \gamma\hat{V}(S_{t+1}, \mathbf{w}) - \hat{V}(S_t, \mathbf{w})$, where R_{t+1} is the reward received after transitioning to state S_{t+1}, and γ is the discount factor. The weights are then updated as $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\phi(S_t)$.

Linear methods are favored for their ease of implementation and their ability to handle large state spaces effectively, provided the feature vectors $\phi(s)$ or $\phi(s, a)$ are appropriately chosen to capture the relevant aspects of the environment. These methods offer a good balance of performance and computational efficiency, making them a practical choice for many reinforcement learning problems, especially when using SGD-based approaches like Monte Carlo and gradient TD.

4.6 Approximate solution methods testing and results

5 DEEP REINFORCEMENT LEARNING

....
....
....
....
... .
....
....
....
....
....

5.1 Introduction to DRL

....
....
....
....
... .
....
....
....
....
....

5.2 DQN

....

5.3 DQN Testing and results

....

....

6 Conclusions

....

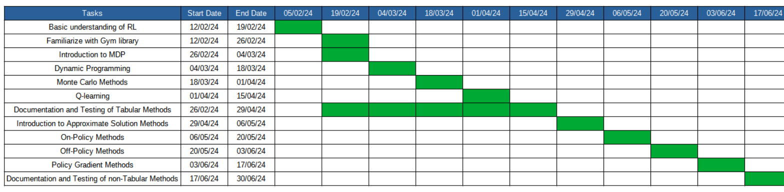
Special Thanks

... .

References

https://gymnasium.farama.org/

Appendix



Tasks	Start Date	End Date	05/02/24	19/02/24	04/03/24	18/03/24	01/04/24	15/04/24	29/04/24	06/05/24	20/05/24	03/06/24	17/06/24
Basic understanding of RL	12/02/24	19/02/24	🟩										
Familiarize with Gym library	12/02/24	26/02/24		🟩									
Introduction to MDP	26/02/24	04/03/24		🟩									
Dynamic Programming	04/03/24	18/03/24			🟩								
Monte Carlo Methods	18/03/24	01/04/24				🟩							
Q-learning	01/04/24	15/04/24					🟩						
Documentation and Testing of Tabular Methods	26/02/24	29/04/24		🟩	🟩	🟩	🟩	🟩					
Introduction to Approximate Solution Methods	29/04/24	06/05/24							🟩				
On-Policy Methods	06/05/24	20/05/24								🟩			
Off-Policy Methods	20/05/24	03/06/24									🟩		
Policy Gradient Methods	03/06/24	17/06/24										🟩	
Documentation and Testing of non-Tabular Methods	17/06/24	30/06/24											🟩

Fig. 8: Gantt Diagram