IAJ-Learning
Grupo 02

Nome: João Costa **Número de aluno: ist199985**
Nome: João Azevedo **Número de aluno: ist196877**

1 QLearning

Q-Learning is a popular algorithm in reinforcement learning that relies on iterative updates to learn the best actions an agent should take in various states to maximize cumulative rewards.

The key to Q-Learning is its use of a *Q-table*, a lookup table where each entry, $Q(s, a)$, represents the value or "quality" of taking action a in state s. Initially, all values in the Q-table are often set to zero or some baseline value, indicating that the agent has no prior knowledge of rewards in the environment.

As the agent explores the environment, it updates the Q-values in the table using the *Q-learning update rule* in equation 1.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big) \tag{1}$$

where:

- s is the current state,
- a is the action taken,
- r is the immediate reward received after taking action a,
- s' is the next state,
- α is the learning rate, which controls how much new information overrides previous Q-values,
- γ is the discount factor, which balances immediate and future rewards.

The Q-learning algorithm also uses an exploration-exploitation strategy to balance trying new actions (exploration) with leveraging the Q-values it has learned (exploitation). Initially, the agent chooses actions randomly with a 70% probability, allowing it to explore and gather information about the environment. Over time, this randomness decreases, shifting the agent toward using the Q-values to make decisions based on learned knowledge. This is often implemented through an ϵ-greedy policy, where the probability of taking a random action, ϵ, starts at 0.7 and gradually decreases as learning progresses.

Through these repeated updates and the balance between exploration and exploitation, the Q-values converge toward optimal values, effectively encoding the optimal policy. While this table-based approach works well in smaller, discrete environments, it can become impractical in larger or continuous spaces due to the exponential growth of possible state-action pairs.

1.1 Table

For our Q-Learning implementation, we experimented with various state representations and ultimately chose a table that captures three key states: `HpState`, `ManaState`, and `LevelState`. Since Q-Learning requires discretized state values to learn properly, each state was divided into a few distinct levels based on gameplay conditions.

For the `HpState`, we defined four levels based on the combined values of health (HP) and shield:

- **Level 4:** HP + Shield is above 12 — Sir Uthgard can eliminate all enemies.
- **Level 3:** HP + Shield is between 7 and 12 — Sir Uthgard can defeat orcs and skeletons.
- **Level 2:** HP + Shield is between 3 and 6 — Sir Uthgard wins against skeletons.
- **Level 1:** HP + Shield is below 2 — Sir Uthgard is vulnerable to all enemies and risks death.

For the `LevelState`, we simply distinguish between the first level and all subsequent levels. After the first level, the HP limit is at 20, allowing Sir Uthgard to defeat any enemy with full health.

`ManaState` has only two levels to keep the model simple:

- **High Mana:** Mana is greater than or equal to 5, allowing healing actions like `ShieldOfFaith` (5 mana) and `LayOnHands` (7 mana).
- **Low Mana:** Mana is less than 5, where only the damage action `DivineSmite` (2 mana) is feasible.

In terms of actions, we track each distinct action since various routes (like those involving different enemies or potions) can alter the optimal path. This is especially relevant with the `orc patrol`, where path choices may expose Sir Uthgard to different enemies.

Given the states and actions defined, we can calculate the maximum number of entries in the Q-table. With:

- 4 levels for `HpState`,
- 2 levels for `ManaState`,
- 2 levels for `LevelState`,

we have $4 \times 2 \times 2 = 16$ possible state combinations.

With a maximum of 22 actions (calculated as $1+2+2+9+1+5+1+1+5 = 27$), the total entries in the Q-table are:

$$16 \times 27 = 432$$

Thus, the maximum number of entries in our Q-table is 432, covering all possible state-action combinations in this setup however we only put that combination on the table when we encounter it to minimize space.

At the beginning of training, Sir Uthgard encounters only a few unique state-action pairs per episode—typically around 3 to 5—as he is often defeated quickly. As his strategies improve, the number of unique state-action pairs encountered per episode increases, averaging around 10 to 15 towards the later stages.

Our training is expected to be upwards of three thousand episodes to ensure adequate training and refinement of Q-values. This range is typical for moderate-complexity tasks, allowing for sufficient exploration and convergence across varied encounters.

In the following section (1.2), we illustrate how reducing the action space by removing *Sword Attack*, the most frequently occurring action, simplifies training. With this adjustment, the Q-table shrinks to 342 entries, resulting in a faster and more efficient training process.

1.2 Training and Results

For the training phase of our algorithm, we adjusted the reward function to better guide the agent's learning process. Instead of assigning a simple penalty of -100 when the agent dies and a reward of +100 for winning, we introduced additional incentives and penalties to shape the agent's behavior. Specifically, a negative reward was calculated based on the number of actions taken up to that point, scaled by 0.1. This approach penalizes inefficient strategies that prolong survival without achieving goals. Additionally, we prioritized the collection of gold by assigning a positive reward proportional to the collected amount, defined as *Money * 0.2*.

We conducted multiple experiments to evaluate these changes, all experiments had a maximum of 4000 iterations and a learning rate of 0.05. Initially, we trained the agent in a controlled environment with enemies in a "sleep" state and with only 3 available actions, *PickUpChest*, *GetHealthPotion* and *Levelup*. Figures 1 and 2 show the accumulation of gold and the duration of survival over iterations, respectively, under these conditions.

Figure 1: Accumulated gold over iterations with a low time limit and enemies asleep.

Figure 2: Time survived over iterations with a low time limit and enemies asleep.

Upon analyzing Figures 1 and 2, we observed that the agent consistently survived until the set time limit of 150 iterations, suggesting that this duration might be insufficient for the agent to explore all possible actions effectively. Consequently, we increased the time limit to 250 iterations.

Figures 3 and 4 illustrate the impact of this change, with notable performance improvements. The agent's survival duration increased significantly, and it achieved a 20% win rate

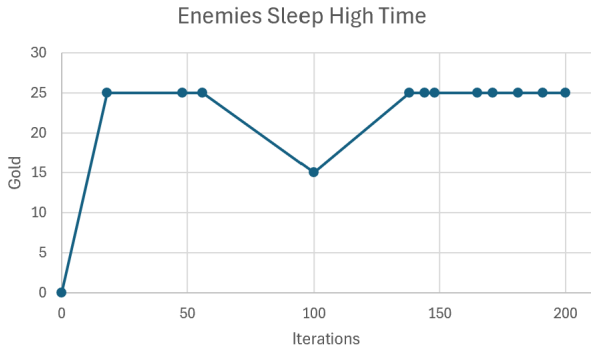

Figure 3: Accumulated gold over iterations with a high time limit and enemies asleep.

Figure 4: Time survived over iterations with a high time limit and enemies asleep.

within just 200 iterations, indicating more efficient learning. We maintained this extended time limit for subsequent experiments.

Next, we tested the agent with all available actions enabled. The results of these experiments are shown in Figures 5, 6, and 7.

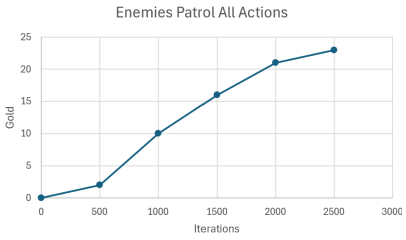

Figure 5: Accumulated gold over iterations with all actions available.

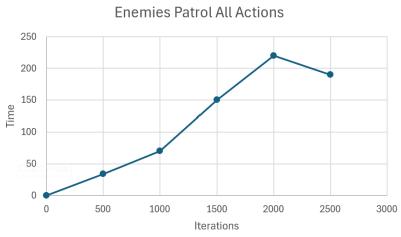

Figure 6: Time survived over iterations with all actions available.

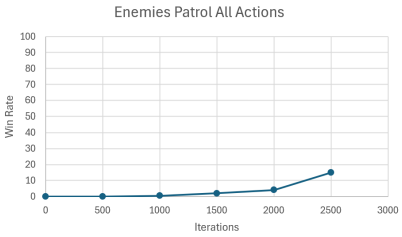

Figure 7: Win rate over iterations with all actions available.

After analyzing these plots, we observed that while the model shows signs of learning, it is progressing very slowly. Even after four thousand iterations, the win rate remained at just 15%, indicating that the learning process might not be as efficient as desired.

For our final experiment, we decided to disable the *Sword Attack* action to investigate whether this would speed up learning and improve performance.

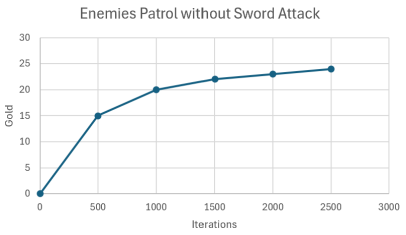

Figure 8: Accumulated gold over iterations without Sword Attack.

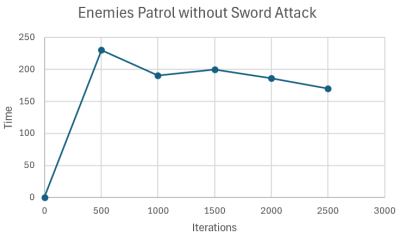

Figure 9: Time survived over iterations without Sword Attack.

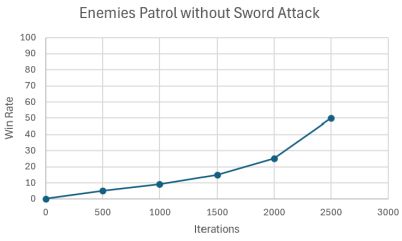

Figure 10: Win rate over iterations without Sword Attack.

The results shown by figures 8, 9 and 10 indicate a significant improvement: the model learned much faster and achieved a considerably higher win rate compared to the previous

setup. This suggests that removing the *Sword Attack* action and reducing the table streamlined the agent's decision-making process, leading to more efficient and effective learning.

2 DeepQLearning

2.1 Deep Q-Learning Model

Deep Q-Learning (DQL) is an approach in reinforcement learning that combines traditional Q-learning, a value-based method, with deep neural networks to handle high-dimensional state spaces. In scenarios with large state spaces, maintaining a Q-value table becomes infeasible. Instead, DQL replaces this Q-table with a deep neural network, known as the *Q-network*, to approximate Q-values.

2.2 Q-Network Architecture

The Q-network takes in the current state, s, as input and outputs Q-values for each possible action, a. This network is trained to minimize the difference between the predicted Q-values and the target Q-values using a loss function, which is optimized through backpropagation.

2.2.1 State Representation

The state space for this implementation consists of five continuous attributes `LevelState`, `XPState`, `HpState`, `ManaState`, and `GoldState`. Initially, only three of these attributes were used (from the prior Q-learning model), but the neural network allowed for additional states without significant computational overhead, enabling improved reward distribution.

2.2.2 Experience Replay

Instead of updating the Q-network after every experience (s, a, r, s') – where s is the state, a the action, r the reward, and s' the next state – experiences are stored in a *replay buffer*. During training, random samples are drawn from this buffer to reduce the correlation between consecutive experiences, enhancing training stability.

2.2.3 Exploration-Exploitation Strategy

An ϵ-greedy policy is used to balance exploration and exploitation. In this policy, the agent selects a random action with probability ϵ (exploration) or chooses the action with the highest predicted Q-value with probability $1 - \epsilon$ (exploitation).

2.3 Neural Network Structure

The Q-network is a feedforward neural network using the ReLU activation function, chosen for its simplicity and efficient gradient propagation. The network architecture includes:

- 5 input nodes, corresponding to the state attributes,
- Two hidden layers with 20 nodes each, which were selected to balance model complexity and computational efficiency,
- Output layer with x nodes, where x represents the number of possible actions.

2.3.1 Training Methodology

As the agent collects experiences, the neural network is trained using gradient descent. For each experience, the network minimizes the difference between the predicted Q-values and target Q-values using backpropagation to update the network weights.

In neural networks, backpropagation calculates the gradient of the loss function with respect to each weight in the network. For a weight $w^{(l)}$ at layer l, the weight update in gradient descent is defined in equation 2.

$$w^{(l)} = w^{(l)} - \eta \frac{\partial L}{\partial w^{(l)}} \tag{2}$$

where:

- $w^{(l)}$: weight at layer l,
- η: learning rate,
- L: loss function, and
- $\frac{\partial L}{\partial w^{(l)}}$: gradient of the loss function with respect to $w^{(l)}$.

2.3.2 Gradient Calculation Using the Chain Rule

Using the chain rule, the gradient $\frac{\partial L}{\partial w^{(l)}}$ is computed in equation 3.

$$\frac{\partial L}{\partial w^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial w^{(l)}} \tag{3}$$

where:

- $a^{(l)}$: activation of layer l,
- $z^{(l)}$: weighted input to the activation function at layer l,
- $\frac{\partial L}{\partial a^{(l)}}$: gradient of the loss with respect to the activation,
- $\frac{\partial a^{(l)}}{\partial z^{(l)}}$: derivative of the activation function,
- $\frac{\partial z^{(l)}}{\partial w^{(l)}}$: derivative of the weighted input with respect to the weight $w^{(l)}$.

2.4 Training and Results

2.4.1 Reward Structure

To optimize the agent's learning process, the reward function was refined to provide both immediate feedback and strategic incentives throughout gameplay. Rather than exclusively rewarding the agent with $+100$ for winning and -100 for losing.

Specifically, the agent receives a reward of $+20$ for progressing to a new level, which encourages advancement in skill. Gains in health and mana award $+1$ and $+0.5$ respectively, promoting effective resource management. Similarly, experience points and gold increments yield $+1$ and $+0.5$ per unit, fostering consistent progression and incentivizing the collection of

resources. Additionally, each action incurs a minor penalty of -0.1 to discourage unnecessary steps, motivating the agent to prioritize efficient strategies and avoid prolonged, unproductive gameplay.

We also conducted multiple experiments to evaluate these changes, all experiments had a maximum of 4000 iterations and a learning rate of 0.05. Initially, we trained the agent in a controlled environment with enemies in a "sleep" state with all actions available. Figures 11, 12 and 13 show the accumulation of gold, the survival duration, and the win rate over iterations, respectively, under these conditions.

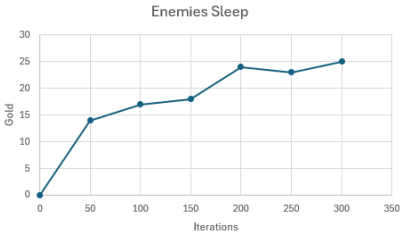

Figure 11: Accumulated gold over iterations with enemies asleep.

Figure 12: Time survived over iterations with enemies asleep.

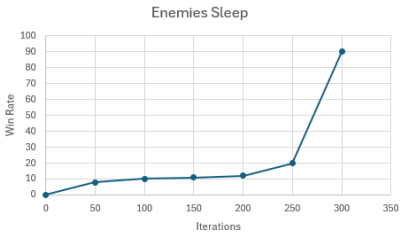

Figure 13: Win rate over iterations with enemies asleep.

As we can see DQL learns much faster and manages to achieve almost 100% win rate in 300 iterations.

We also conducted one last experiment with Orc Patrol.

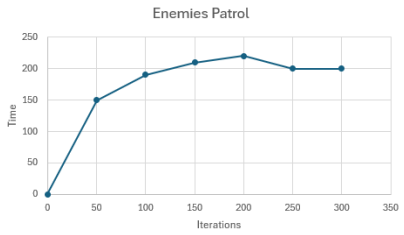

Figure 14: Accumulated gold over iterations with enemies with the patrol.

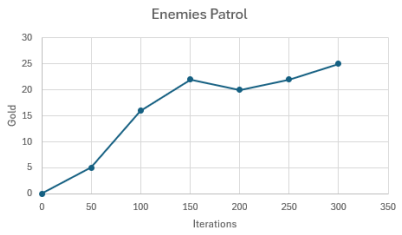

Figure 15: Time survived over iterations with enemies with the patrol.

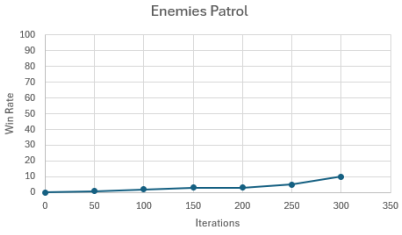

Figure 16: Win rate over iterations with enemies with the patrol.

By looking at the figures 14, 15 and 16 we can say that even with the patrol activated the algorithm found the victory much quicker.

That concludes that the DQL agent outperformed the Q-Learning agent, especially in complex scenarios with many states and actions. With the continuous state space, the neural network enabled more granular decision-making than the Q-table. This flexibility reduced computational limitations and demonstrated the advantages of DQL for large, high-dimensional environments.

3 Conclusion

In this project, we successfully implemented both Q-Learning and Deep Q-Learning algorithms for a game-based reinforcement learning environment. Through iterative training,

reward shaping, and adjustments to the action space, we demonstrated the evolution of an agent from a novice with no knowledge of the environment to a proficient player capable of strategic decision-making.

After more than 300 hours of training, we observed key differences between the algorithms, particularly in how they represent the state of the game. Traditional Q-Learning is effective in smaller, discrete environments where states can be straightforwardly represented, although it requires discretization in complex settings. In contrast, DQL can process continuous state spaces directly, leveraging neural networks to interpret high-dimensional inputs and enabling a more scalable and efficient approach for complex environments. DQL effectively learns faster and achieves better results but is significantly more complex to implement and tune compared to Q-Learning.

Another essential factor in achieving optimal results was the computational power of the hardware. With a more powerful computer, we observed faster training times and more consistent model performance, which allowed for more effective model fine-tuning.

For future work we would like to have even more time to explore the algorithms and their results and could have integrated advanced techniques such as Double DQL to enhance stability further and reduce overestimation biases, potentially improving training efficiency and model robustness.