



SAPIENZA
UNIVERSITÀ DI ROMA

MsC IN ENGINEERING IN COMPUTER SCIENCE
ACADEMIC YEAR 2023/2024

Applying RL to *Flappy Bird* Environment - Q-Table vs DQN

ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

Professor:

Patrizi Fabio

Student:

Franzoso Antonio (2133047)

Contents

1	Introduction	3
2	Theoretical Background	6
2.1	Tabular Q-Learning (Q-Table)	6
2.2	Deep Q-Network (DQN)	7
2.2.1	Double DQN	8
2.2.2	Dueling DQN	9
2.3	General Strategies Applicable to Both	10
2.3.1	Epsilon-greedy Strategy	10
2.3.2	Experience Replay (ER)	11
3	Implementation	12
3.1	<i>Flappy Bird</i> Environment	12
3.2	Code Structure	13
3.3	<code>Agent</code> class	14
3.4	<code>Q_Agent</code> class	15
3.4.1	Q-Table Creation	15
3.4.2	Handling Continuous Observation Spaces	16
3.4.3	Action Selection	17
3.4.4	Updating Q-Table	17
3.5	<code>DQN_Agent</code> class	19
3.5.1	Neural Network Architecture	19
3.5.2	Optimization Step in the DQN Agent	20
4	Results	23
4.1	Experiment Configurations	23
4.2	Hyperparameters Description	23
4.3	Training Results	26
4.3.1	Q-Table	26
4.3.2	DQN	27
4.3.3	Comparison	28
4.4	Test Results	29

5	Conclusions	31
	References	32

1 Introduction

Reinforcement Learning (RL) is one of the three major learning paradigms in Machine Learning (ML), alongside supervised and unsupervised learning. Unlike these approaches, which rely on existing human-collected data and are inherently limited by human intelligence, RL learns directly from interactions with the environment through trial and error and it can achieve performance that may surpass human capabilities.

RL follows a goal-oriented framework where an **agent** interact with an **environment** performing **actions** and collecting a **reward** from it (Figure 1) [1]. These rewards help agents navigate through a complex environment to reach the final goal. Similar to how a toddler learns to walk through repeated attempts, an RL agent can learn how to perform complex tasks by trial and error without human intervention.

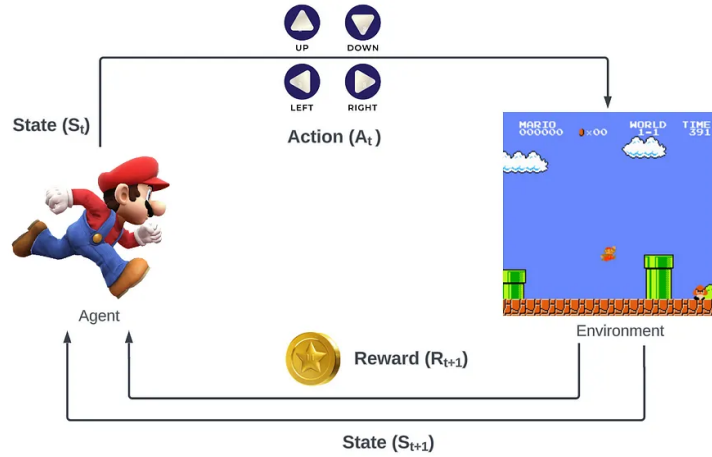


Figure 1: Typical RL problem's loop.

The central principle of RL techniques is to learn a **policy** for selecting actions that maximizes the total cumulative rewards collected by the agent. The underlying assumption is that this will lead to an understanding on how to interact with the environment effectively, always selecting the best action regardless of its current configuration (also called **state**, or **observation**).

This study applies RL to a Gymnasium implementation of the popular *Flappy Bird* game [2], originally released on May 24th, 2013, for Apple and Android. The game quickly became a classic [3], praised for its simple yet addictive gameplay (Figure 2).

Flappy Bird is an arcade-style game in which the player controls a bird that moves persistently to the right. They are tasked with navigating it through pairs of pipes that have equally sized gaps placed at random heights. The bird automatically descends and only ascends when the

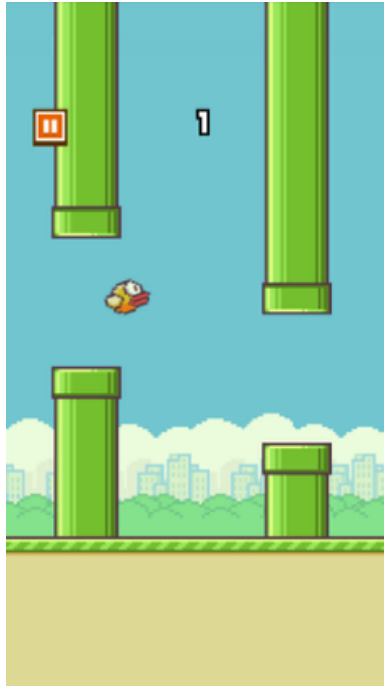


Figure 2: *Flappy Bird*'s screenshot.

player taps on the touchscreen. Colliding with a pipe or the ground ends the gameplay [4].

The game's difficulty has been a subject of debate [5]. While it may initially seem to require minimal skill, mastering it demands a high level of precision and timing. Players must carefully control the bird's altitude to fit through narrow gaps, making small miscalculations fatal. This aspect contributes to its reputation as both frustrating and highly engaging.

This characteristic makes *Flappy Bird* a compelling case study for RL. Analyzing how a virtual agent learns to perform in such a challenging environment provides insight into its decision-making process, learning strategies, and overall behavior during training.

As previously mentioned, the environment used in this study is part of Gymnasium's External Environments collection. Gymnasium, an evolution of OpenAI Gym, is a widely used library that provides ready-to-use RL environments. These serve as standardized benchmarks, allowing researchers to experiment with various RL techniques in a structured setting.

In this study, we apply two different RL techniques to the *Flappy Bird* environment.

- **Tabular Q-Learning (Q-Table)**, a fundamental value-based method, is used as a baseline to understand the feasibility of solving the game with a discrete state-action space.
- **Deep Q-Network (DQN)**, on the other hand, leverages deep learning to approximate the Q-function, enabling the agent to handle the game's continuous and high-dimensional

state space more effectively.

We compare the performance of both approaches, analyzing their learning efficiency, stability, and overall success in mastering the game.

The report is organized as follows. Chapter 2 presents the theoretical foundations of the two RL approaches used in this study, while Chapter 3 provides a detailed description of the environment and explores the source code implementation. Presentation and analysis of the results are contained in Chapter 4, and finally, Chapter 5 summarizes the findings and presents final remarks on the study.

2 Theoretical Background

Chapter 2 presents the theoretical foundations of the two RL approaches used in this study: Tabular Q-Learning and Deep Q-Network (DQN). It also provides a detailed description of the environment, including its state and action spaces, reward structure, and overall mechanics.

2.1 Tabular Q-Learning (Q-Table)

Tabular Q-Learning is a reinforcement learning method that maintains a table, known as the *Q-Table*, to store Q-values for every state-action pair. It is a value-based method, meaning it does not attempt to learn a policy directly but instead estimates the value of state-action pairs.

Q(s, a)				
	Left	Down	Right	Up
State.0	Q-value	Q-value	Q-value	Q-value
State.1	Q-value	Q-value	Q-value	Q-value
State.2	Q-value	Q-value	Q-value	Q-value
⋮				
State.15	Q-value	Q-value	Q-value	Q-value

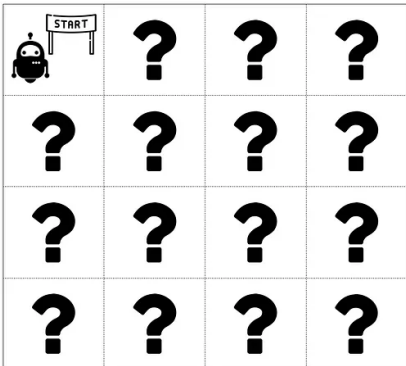


Figure 3: Q-Table example for environment with 16 states and 4 actions.

The Q-table has dimensions $|S| \times |A|$, where $|S|$ is the number of possible states, and $|A|$ is the number of available actions in each state (an example is shown in Figure 3). It is iteratively updated based on agent interactions with the environment using the **Bellman equation**):

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \quad (1)$$

where:

- s and s' are the current and next states,
- a and a' are the current and next actions,
- r is the received reward,

- α is the learning rate,
- γ is the discount factor.

2.2 Deep Q-Network (DQN)

DQN extends Q-learning by approximating the Q-function using a neural network. Instead of storing Q-values in a table, a deep neural network takes a state as input and outputs Q-values for all possible actions (Figure 4).

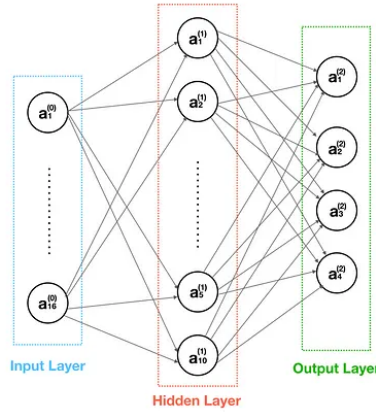


Figure 4: DQN example for previous environment.

The architecture consists of:

- An **input layer** of size $|S|$ (state space dimension),
- One or more **hidden layers**,
- An **output layer** of size $|A|$ (number of possible actions).

The update rule for DQN is based again on the Bellman equation 1, similar to Q-Learning. But this time, in order to apply the update, we must first compute the new Q-value for a specific state-action pair, often referred to as the Q-target. The network parameters are then updated via gradient descent using an optimizer, typically Adam. In this approach, the learning rate α becomes a parameter of the optimizer rather than being applied directly to the update rule.

The DQN update rule, based on the Bellman equation, first computes the Q-target for a specific state-action pair (2). The network parameters are then updated using gradient descent with an optimizer, typically Adam, where the learning rate α is a parameter of the optimizer.

$$Q(s, a; \theta) \leftarrow r + \gamma \max_{a'} Q(s', a'; \theta) \quad (2)$$

The optimizer minimizes the loss function, which in DQN is the Mean Squared Error (MSE) between the predicted Q-values and the target Q-values from the Bellman equation.

$$L(\theta) \leftarrow \mathbb{E}[r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a, \theta)] \quad (3)$$

DQN by default stabilizes training by using Experience Replay (see Section 2.3.2) to store past experiences, so it learns from previously collected experiences rather than relying solely on the latest interactions.

The version described in the DQN original paper (here referred as DQN-original) would require also a second neural network, called the target network (θ^- , used to calculate the target Q-values).

In DQN-original updates are still performed only on the policy network, while target network is kept "frozen" for a few iteration and then periodically synced with θ . This makes the estimations produced by the target network more accurate after the copying has occurred, stabilizing the training.

The updated Bellman equation for the DQN-original so become:

$$Q(s, a; \theta) \leftarrow r + \gamma \max_{a'} Q(s', a'; \theta^-) \quad (4)$$

where:

- $Q(s, a; \theta)$ is the Q-value for the current state, in the policy network θ
- $\max_{a'} Q(s', a'; \theta^-)$ is the maximum Q-value for the next state, in the target network θ^-

In this experiment, though, with "DQN" we will refer to its simplest variant, with just one network (θ), and then we will implement other techniques for comparison.

2.2.1 Double DQN

Double DQN (DDQN) improves upon DQN by reducing the overestimation bias, the fact that agent always choose the non-optimal action in any given state only because it has the maximum Q-value.

The key difference from DQN-original is that the action selection and Q-value computation are decoupled, leading to more stable learning.

Like DQN-original, it uses two separate networks:

- The **policy network** (parameters θ) selects the action,
- The **target network** (parameters θ^-) evaluates the Q-value.

But instead of using the same Bellman equation seen in 4, it changes it like this:

$$\begin{aligned} a^* &= \arg \max_{a'} Q(s', a'; \theta) \\ Q(s, a; \theta) &\leftarrow r + \gamma Q(s', a^*; \theta^-) \end{aligned} \tag{5}$$

- $a^* = \arg \max_{a'} Q(s', a'; \theta)$ - First, the main neural network θ decides which one is the best next action a' among all the available next actions
- $Q(s', a^*; \theta^-)$ - Then the target neural network (θ^-) evaluates this action in the next state, to know its Q-value.

2.2.2 Dueling DQN

Dueling DQN modifies the architecture of DQN by separating the Q-value computation into two different parts:

- **Value function** $V(s)$, which estimates the value of being in state s . This value reflects how good it is to be in a given state, independent of the specific action taken.
- **Advantage function** $A(s, a)$, which measures the relative advantage of taking action a over the average action in state s . This function indicates how much better or worse a particular action is compared to others in the same state.

The final Q-value is computed as:

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|A|} \sum_{a'} A(s, a') \tag{6}$$

By decoupling the value and advantage components, Dueling DQN allows the agent to more efficiently learn state values, especially in environments where certain states are common

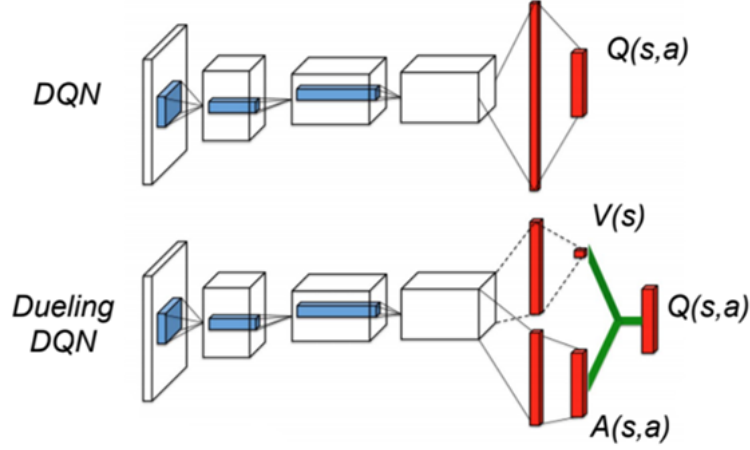


Figure 5: Dueling DQN NN visualization, taken from original paper.

across actions. This structure can improve the learning process by enabling the model to better differentiate between states, even when multiple actions in those states may lead to similar outcomes [6].

2.3 General Strategies Applicable to Both

2.3.1 Epsilon-greedy Strategy

The ε -greedy strategy balances exploration and exploitation. The agent faces a trade-off between choosing actions based on its current knowledge (**exploitation**) and selecting actions that might lead to discovering new knowledge (**exploration**).

The strategy selects an action according to the following rule:

$$a = \begin{cases} \text{random action} & \text{with probability } \varepsilon, \\ \arg \max_a Q(s, a) & \text{with probability } 1 - \varepsilon. \end{cases} \quad (7)$$

Initially, the agent explores more (high $\varepsilon = \varepsilon_{init}$), but it gradually shifts towards exploitation as training progresses. The rate at which exploration decays is controlled by the hyperparameter ε_{decay} , which adjusts how much ε decreases at each step.

$$\varepsilon \leftarrow \min(\varepsilon_{final}, \varepsilon * \varepsilon_{decay}) \quad (8)$$

2.3.2 Experience Replay (ER)

Experience Replay (ER) stabilizes training by storing past experiences in a replay buffer and randomly sampling mini-batches during training. This process helps break the correlation between consecutive experiences, improving sample efficiency and enhancing learning stability.

The replay buffer contains tuples:

$$(s, a, r, s') \tag{9}$$

Applying ER to methods like Q-Table and DQN makes them both **off-policy** methods, as they learn from experiences generated by policies that may differ from the current one. ER enables storing past experiences, allowing the model to update its Q-values using actions taken by a different policy, rather than relying only on the most recent interactions, which would otherwise reflect just the current policy.

3 Implementation

This chapter provides a detailed description of the environment, including its state and action spaces, reward structure, and overall mechanics. Moreover, it covers the implementation of the Python code, explaining the configurations and optimizations applied.

3.1 *Flappy Bird* Environment

The FlappyBird-v0 Gymnasium environment provides numerical information about the game’s state. The environment includes a continuous state observation space and a discrete action space for interacting with the agent.

- **Observation Space:** The observation space consists of 12 continuous float values, normalized between $[-1, +1]$. A full description of them is contained in Table 1.

Index	Description
1	Last pipe’s x -position
2	Last top pipe’s y -position
3	Last bottom pipe’s y -position
4	Next pipe’s x -position
5	Next top pipe’s y -position
6	Next bottom pipe’s y -position
7	Next next pipe’s x -position
8	Next next top pipe’s y -position
9	Next next bottom pipe’s y -position
10	Player’s y -position
11	Player’s y -velocity
12	Player’s rotation

Table 1: Features composing a single observation.

- **Action Space:** The action space consists of two discrete actions:
 - 0: Do nothing

- **1:** Flap
- **Rewards:** Rewards are given based on the player's actions:
 - **+0.1:** For every frame the player stays alive
 - **+1.0:** For successfully passing a pipe
 - **-1.0:** For dying (collision with a pipe)
 - **-0.5:** For touching the top of the screen
- **Episode Termination:** An episode terminates when the player either:
 - Collides with a pipe
 - Touches the top of the screen

3.2 Code Structure

The project source code is organized as follows:

- **models/** - Contains trained models or checkpoints saved during the training process.
- **results/** - Contains plots from training/testing runs.
- **runs/** - Contains output of different training runs in CSV format.
- **videos/** - Includes video recordings of agent performances for visualization purposes.
- **Agent.py** - Defines the agent's behavior, policy updates, and decision-making processes. Contains the `Q_Agent` and `DQN_Agent` classes.
- **ReplayMemory.py** - Implements experience replay, crucial for memory-based reinforcement learning methods.
- **main.py** - The main script that initializes the training/testing pipeline.
- **hyperparameters.yml** - A YAML file storing hyperparameters used across experiments for easier configuration and reproducibility.

3.3 Agent class

Here we'll go through the *Agent.py* code, the one that defines our agent and its learning behaviour.

Code is structured with an abstract parent **Agent** class that defines general attributes and methods used by its children **Q_Agent** and **DQN_Agent** that inherit from it and specify specific behaviour just for Tabular Q-Learning or DQN (Figure 6).

```

19 class Agent:
20     @abc.abstractmethod
21     def get_action(self, obs, is_training=True) -> int: ...
22
23     @abc.abstractmethod
24     def update(self, ...): ...
25
26     @abc.abstractmethod
27     def optimize(self, minibatch): ...
28
29     @abc.abstractmethod
30     def load_model(self, model_name: str): ...
31
32     @abc.abstractmethod
33     def save_model(self, n_episodes: int | None): ...
34
35     @abc.abstractmethod
36     def run(self, n_episodes: int | None, is_training =
37             True, show_plots = True, verbose = True, model_name
38             = None, seed = None): ...
39
40     @abc.abstractmethod
41     def plot_results(self, all_rewards, mean_rewards,
42                     epsilon_values, training = False, temp=False, show
43                     = True, integrated = False): ...

```

(a) Agent

```

523 class Q_Agent(Agent):
524     def get_action(self, obs, is_training=True) -> int: ...
525
526     def update(self, ...): ...
527
528     def optimize(self, mini_batch): ...
529
530     def load_model(self, model_name: str): ...
531
532     def save_model(self, n_episodes: int | None): ...
533
534     def run(self, n_episodes: int | None, is_training =
535             True, show_plots = True, verbose = True, model_name
536             = None, seed = None): ...
537
538     def plot_results(self, all_rewards, mean_rewards,
539                     epsilon_values, training = False, temp = False, show
540                     = True, integrated = False): ...

```

(b) Q_Agent

```

498 class DQN_Agent(Agent):
499     def get_action(self, obs, is_training=True) -> int: ...
500
501     def update(self, ...): ...
502
503     def optimize(self, mini_batch): ...
504
505     def load_model(self, model_name: str): ...
506
507     def save_model(self, n_episodes: int | None): ...
508
509     def run(self, n_episodes: int | None, is_training =
510             True, show_plots = True, verbose = True, model_name
511             = None, seed = None): ...
512
513     def plot_results(self, all_rewards, mean_rewards,
514                     epsilon_values, training = False, temp = False, show
515                     = True, integrated = False): ...

```

(c) DQN_Agent

Figure 6: Code structure in *Agent.py*

Figure 7 illustrates the main loop used for training the agent.

- The process iterates over multiple episodes, where each episode starts with the environment reset.
- The agent interacts with the environment using an ϵ -greedy strategy to select actions, which are then executed.
- The environment returns a new state, a reward, and termination signals.
- The agent updates its model based on the collected experience.
- The episode continues until the termination condition is met, updating the total reward and step count.

This loop ensures that the agent learns by continuously exploring and optimizing its policy over multiple episodes.

Each subclass customizes these methods to fit its specific learning approach. In the following subsections, we will examine how different implementations handle these steps in practice.

```

# Loop over episodes
for self.episode in iterable:
    obs, info = self.env.reset(seed=seed)
    done = False
    rewards = 0
    frames = 0
    self.synced = False

    # Play one episode
    while not done:
        # Choose action based using e-greedy strategy
        action = self.get_action(obs, is_training=is_training)

        # Perform action
        next_obs, reward, terminated, truncated, info = self.env.step(action)

        # Update the agent model
        self.update(obs, action, reward, terminated, next_obs, is_training=is_training)

        # Update environment and collect reward
        done = terminated or truncated
        obs = next_obs
        rewards += reward
        frames += 1

```

Figure 7: Training loop in `Agent` class.

3.4 `Q_Agent` class

The `Q_Agent` class implements a Q-learning agent by maintaining a Q-table that maps state-action pairs to estimated reward values.

3.4.1 Q-Table Creation

A core design choice of this implementation is the use of a `defaultdict` to store Q-values. Instead of initializing a massive table covering all possible states, which would be infeasible for large or continuous state spaces, the agent creates entries only for states it actually visits. This approach is both memory-efficient and scalable.

The default value for a new state is a NumPy array of size equal to the number of possible actions, with all Q-values initialized to zero. This ensures that when a new state is encountered for the first time, a corresponding action-value array is automatically created and initialized accordingly (Figure 8).

```

def _create_numpy_entry(self):
    return np.zeros(self.env.action_space.n)

# Create q_table as defaultdict to save space
self.q_table = defaultdict(_create_numpy_entry)

```

Figure 8: Q-Table creation.

With this method, the Q-table dynamically expands only as needed, preventing unnecessary

memory consumption.

3.4.2 Handling Continuous Observation Spaces

When dealing with environments that have continuous observation spaces (such as environments using `gym.spaces.Box`), the agent must discretize the observations to fit within a tabular Q-learning framework. This is done through the `_handle_box_space` method (Figure 9).

```
def _handle_box_space(self, obs_space: gym.spaces.Box, obs_spaces: list, divisions = 10):
    # Flatten the box
    original_obs_space = gym.spaces.flatten_space(obs_space)

    for i in range(0, original_obs_space.shape[0]):
        # When an obs will be discretized it could have values between 0 and divisions
        # included (divisions+1 total values)
        space = np.linspace(original_obs_space.low[i], original_obs_space.high[i],
                            divisions)
        obs_spaces.append(space)

    # Q-table will have obs_spaces_size x action_spaces
    self.obs_spaces_size = 1
    for i, s in enumerate(self.obs_spaces):
        self.obs_spaces_size *= (len(s)+1)
```

Figure 9: Function for handling continuous observation space.

This method discretizes each dimension of the observation space by dividing its range into a fixed number of bins (controlled by the `divisions` hyperparameter, D). The result is a finite set of discrete states that the agent can handle using a Q-table.

The discretization process is completed by the `_discretize_obs` method, which we will be used during training to map a continuous observation into a discrete state index (Figure 10).

This method finds the appropriate bin for each observation value and converts the multi-dimensional observation into a single discrete index, allowing it to be used in the Q-table.

```
def _discretize_obs(self, obs):
    # Don't discretize if obs is only one integer
    if (isinstance(obs, int)):
        return obs
    # Index where to insert obs
    # Finds where the observation obs[i] falls among predefined bins
    # Sum all the indexes to get a unique index for the whole observation
    obs_idx = sum([np.digitize(obs[i], self.obs_spaces[i]) * (len(self.obs_spaces[i]) ** i)
                  for i in range(len(obs))])
    return obs_idx
```

Figure 10: Function for discretizing continuous observation.

By using this approach, the agent can efficiently handle continuous states without having to store an excessively large Q-table in advance.

3.4.3 Action Selection

The agent follows an ϵ -greedy strategy to decide between exploration and exploitation (Figure 11).

```
def get_action(self, obs, is_training=True) -> int:
    if is_training and np.random.random() < self.epsilon:
        # Exploration -> Choose random action
        return self.env.action_space.sample()
    else:
        # Exploitation -> Follow Q-table
        obs = self._discretize_obs(obs)
        return int(np.argmax(self.q_table[obs]))
```

Figure 11: Function for choosing action at every step.

With probability ϵ , the agent selects a random action (exploration). Otherwise, it chooses the action with the highest Q-value (exploitation).

Since the Q-table is stored as a `defaultdict`, querying a previously unseen state will automatically initialize it with zeros.

3.4.4 Updating Q-Table

After executing an action, the agent updates its Q-table using the observed reward and next state.

- If ER is enabled, store the experience in memory and sample a mini-batch of experiences for batch processing (done by `store_and_sample_memory` superclass method).
- Call the optimization step if the agent is ready to update.

While the first step remains the same regardless the type of agent, the key differences lie in the optimization step, implemented in the `optimize` function and illustrated in Figure 12.

This method processes a batch of stored experiences, updating the Q-values efficiently using NumPy operations. The code is designed to work both with and without ER.

- **When ER is enabled:** The mini-batch contains multiple past experiences, allowing the agent to learn from older interactions and improving stability.
- **When ER is disabled:** The mini-batch contains only the most recent experience, and the update happens immediately after each action. Despite this, the logic remains the same.

```
def optimize(self, mini_batch):
    # Process mini batch all at once to reduce time
    observations, actions, rewards, terminations, next_observations = zip(*mini_batch)

    # Convert tuples to NumPy arrays where possible for efficient batch operations (q_table as defaultdict will not
    # accept numpy array as keys)
    actions = np.array(actions)
    rewards = np.array(rewards)
    terminations = np.array(terminations, dtype=int)

    # Compute max Q-values for next states (set to 0 if terminated)
    future_q_values = (1-terminations) * np.array([max(self.q_table[next_obs]) for next_obs in next_observations])

    # Compute target Q-values
    q_targets = rewards + self.discount_factor * future_q_values

    # Compute temporal differences
    current_q_values = np.array([self.q_table[obs][action] for obs, action in zip(observations, actions)])
    td_errors = q_targets - current_q_values

    # Update Q-table using NumPy arrays (vectorized update)
    updates = self.learning_rate * td_errors
    for obs, action, update in zip(observations, actions, updates):
        self.q_table[obs][action] += update

    # Implement loss as the mean squared error
    total_loss = np.mean((q_targets - current_q_values) ** 2)
    return td_errors, total_loss
```

Figure 12: Function for optimizing model at every step.

The method follows the Bellman equation for Q-Table reported in Equation 1.

- The agent receives a batch of past experiences and converts each component into NumPy arrays where possibly, making computations faster and more efficient.
- Thus, in order to find the best future Q-value, the agent looks at the possible actions in the next state and selects the highest Q-value, which represents the best expected reward from that state. If the episode has ended, this value is set to zero.
- The new target Q-value is determined by combining the immediate reward and the best future Q-value, scaled by the discount factor. This accounts for both short-term and long-term rewards.
- The difference (also called the temporal difference error) is computed between the current Q-value and the target Q-value. This measures how much the agent's expectation differs from reality.
- The Q-value for the (state, action) pair is adjusted by moving it closer to the target Q-value. The update at every step is gradual, controlled by the learning rate.
- In order to track the learning process and later compare it to the DQN implementation, the method computes the MSE loss metric (Equation 3), which represents how much the Q-values are changing.

3.5 DQN_Agent class

The `DQN_Agent` class defines a DQN RL agent using PyTorch. This agent employs a neural network to approximate Q-values instead of relying on a traditional Q-table, which enables it to handle environments with large state spaces.

3.5.1 Neural Network Architecture

The core of the DQN agent is a neural network that approximates Q-values for each action. The architecture is implemented in the nested DQN class (Figure 13a), which inherits from `torch.nn.Module`.

```
class DQN(nn.Module):
    def __init__(self,
                 state_dim,
                 action_dim,
                 hidden_dim = 128,
                 dueling_dqn = False
    ):
        # nn.Module init method
        super().__init__()

        self.dueling_dqn = dueling_dqn

        # Create layers: #states x #hidden x #actions
        self.fc1 = nn.Linear(state_dim, hidden_dim)

        if self.dueling_dqn:
            # Value stream
            self.fc2_val = nn.Linear(hidden_dim, hidden_dim)
            self.val = nn.Linear(hidden_dim, 1)
            # Advantage stream
            self.fc2_adv = nn.Linear(hidden_dim, hidden_dim)
            self.adv = nn.Linear(hidden_dim, action_dim)
        else:
            self.fc2 = nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))

        if self.dueling_dqn:
            v = F.relu(self.fc2_val(x))
            V = self.val(v)
            a = F.relu(self.fc2_adv(x))
            A = self.adv(a)
            q = V + A - torch.mean(A, dim=1, keepdim=True)
        else:
            q = self.fc2(x)

        return q
```

(a) DQN class for define NN architecture.

```
class DQN_Agent(Agent):
    def __init__(self,
                 # Create DQN
                 self.state_dim = self.env.observation_space.shape[0]
                 self.action_dim = self.env.action_space.n
                 self.policy_dqn = self.DQN(
                     self.state_dim,
                     self.action_dim,
                     self.hidden_dim,
                     self.dueling_dqn
                 ).to(self.device)

                 # Create target dqn used while training
                 if self.double_dqn:
                     self.target_dqn = self.DQN(
                         self.state_dim,
                         self.action_dim,
                         self.hidden_dim,
                         self.dueling_dqn
                     ).to(self.device)
                     self.target_dqn.load_state_dict(
                         self.policy_dqn.state_dict()
                     )

                 # NN loss function, MSE = Mean Squared Error
                 self.loss_fn = nn.MSELoss()

                 # Policy network optimizer
                 self.optimizer = torch.optim.Adam(
                     self.policy_dqn.parameters(),
                     lr=self.learning_rate
                 )
```

(b) NN's creation and setup.

Figure 13: DQN_Agent initialization.

The network is structured as follows:

- **Input Layer:** The network takes a state input of size `state_dim` and processes it through a fully connected layer (`self.fc1`).

- **Hidden Layers:** A ReLU activation is applied after the first fully connected layer, where the number of units is defined by the `hidden_dim` hyperparameter, H .
- **Dueling Architecture (if enabled):** Two independent streams are connected to the hidden layer, both using the ReLU activation function
 - **Value Stream:** A fully connected layer (`self.fc2_val`) of H units calculates the state value $V(s)$, which is further processed through `self.val`.
 - **Advantage Stream:** Another fully connected layer (`self.fc2_adv`) with H units computes the advantage function $A(s, a)$, and the output is passed through `self.adv`.
- **Output Layer:** A fully connected layer (`self.fc2`), also using ReLU, with as many units as there are possible actions, which computes the Q-values for the state passed as input.
 - **With Dueling Architecture:** The Q-values for each action are computed by combining the value and advantage streams, as shown in Equation 6.
 - **Without Dueling Architecture:** The Q-values for each action are computed directly from the output of the hidden layer.

In the constructor of the `DQN_Agent` class, shown in Figure 13b, the policy network, which approximates the Q-values, is instantiated by creating an object of the `DQN` class, using the number of observations per experience as input and the number of actions as output.

If Double DQN is enabled, a second network is also created, the target network. This network is instantiated in the same way as the policy network and is initialized with the same weights using the `load_state_dict` method, ensuring that it can be used during training for more stable target generation.

As with the other agent, the loss function used to compute the error between predicted and target Q-values during training is MSE. To update the policy network's weights and minimize the loss, the Adam optimizer is used, with the learning rate α specified.

3.5.2 Optimization Step in the DQN Agent

Again the `optimize` function (Figure 14) has the duty to update the model at every step during training but this time it will be done by updating the weights of the policy network by computing the loss and performing backpropagation. This time also the mini-batch of experiences will be composed of PyTorch tensors for efficient computation.

```

def optimize(self, mini_batch):
    # Process mini batch all at once to reduce time
    observations, actions, rewards, terminations, next_observations = zip(*mini_batch)

    # Stack tensors to create batch tensors, tensor([1,2,3,...])
    observations = torch.stack(observations)
    actions = torch.stack(actions)
    rewards = torch.stack(rewards)
    next_observations = torch.stack(next_observations)
    # Convert true/false in 1.0/0.0
    terminations = torch.tensor(terminations).float().to(self.device)

    # Calculate target q values (expected returns)
    with torch.no_grad():
        if self.double_dqn:
            # Select best action based on next observations in policy dqn, not target dqn
            best_actions = self.policy_dqn(next_observations).argmax(dim=1)
            # Then apply those actions to target dqn
            target_q = rewards + (1-terminations) * self.discount_factor * self.target_dqn(next_observations).gather(dim=1, index=best_actions.unsqueeze(dim=1)).squeeze()
        else:
            # Here simply select the max q-value from policy dqn
            target_q = rewards + (1-terminations) * self.discount_factor * self.policy_dqn(next_observations).max(dim=1)[0]

    # Calculate q values from current policy (use actions done as index in the tensors)
    current_q = self.policy_dqn(observations).gather(dim=1, index=actions.unsqueeze(dim=1)).squeeze()

    # Compute loss for the whole mini batch
    loss = self.loss_fn(current_q, target_q)

    # Optimize the model
    self.optimizer.zero_grad() # Clear gradients
    loss.backward() # Compute gradients (backpropagation)
    self.optimizer.step() # Update network parameters (weight and biases)

    # Copy policy network to target network after a certain number of episodes
    if self.double_dqn and self.episode % self.network_sync_rate == 0 and not self.synced:
        print(f"Syncing policy and target networks at episode {self.episode}.")
        self.target_dqn.load_state_dict(self.policy_dqn.state_dict())
        self.synced = True

    # Compute TD errors and convert to NumPy array
    td_errors = (target_q - current_q).detach().cpu().numpy()
    return td_errors, loss.item()

```

Figure 14: Function for optimizing model at every step.

- As in the Q-Agent, a mini-batch is processed. This time, the mini-batch is composed of Pytorch tensors for efficient computations. All elements of the mini-batch are stacked together into batch tensors.
- The target Q-values are computed using the Bellman equation.
 - If Double DQN is enabled, best action for the next state is selected using the policy network (`self.policy_dqn`). This action is then used to fetch the Q-value from the target network (`self.target_dqn`). This results in calculating the target Q-value using Equation 5.
 - If Double DQN is not enabled, the Q-value is simply computed by selecting the maximum Q-value from the policy network using the traditional Bellman equation for DQN (Equation 2).
- The Q-values predicted by the policy network are gathered for the actions taken in each state in the mini-batch.

```

policy_dqn(observations)

.gather(dim=1, index=actions.unsqueeze(dim=1)).squeeze()

```

- After calculating the MSE loss between the current Q-values and the target Q-values:
 - The gradients are zeroed out using `self.optimizer.zero_grad()`.

- Backpropagation is performed using `loss.backward()`.
 - The weights of the policy network are updated using `self.optimizer.step()`.
- If Double DQN is enabled, the target network is periodically synchronized with the policy network after a set number of episodes (hyperparameter C). This is done to ensure that the target network remains a stable source of Q-value estimates:

```
target_dqn.load_state_dict(policy_dqn.state_dict())
```

- Finally, the temporal difference errors (the difference between the target and current Q-values) are computed and returned as a NumPy array, along with the total loss for that update.

4 Results

This chapter presents a comprehensive analysis of the experimental results, detailing the different tested configurations and their hyperparameters. It further examines and evaluates the training and test performance, comparing the effectiveness of each configuration in achieving optimal gameplay.

4.1 Experiment Configurations

We combined different techniques and optimizations available for the two fundamental approaches, Tabular Q-Learning (Q-Table) and Deep Q-Learning (DQN), to evaluate their performance in learning to play the game using reinforcement learning.

The tested configurations were:

- **Q-Basic:** Traditional implementation of Q-Table, without using any form of memory.
- **Q-ER:** Q-Table implementation enhanced with ER to evaluate whether this technique, commonly used in DQN, can also provide improvements in a tabular setting.
- **DQN-Basic:** Traditional implementation of DQN using a single neural network as the function approximator (and ER).
- **DQN-Double:** Double DQN implementation.
- **DQN-Dueling:** Dueling DQN implementation.
- **DQN-DoubleDueling:** Combination of Double DQN and Dueling DQN.

All experiments were conducted in a local conda environment on a Windows 11 machine equipped with an AMD Ryzen 7 5800U (8 cores), 16GB RAM, and an NVIDIA GeForce MX450 GPU. The availability of CUDA technology in PyTorch likely contributed to speed up training times for the DQN-based models.

By comparing these approaches, we aim to analyze the advantages, disadvantages, and intrinsic characteristics of each configuration.

4.2 Hyperparameters Description

Below is a description of the hyperparameters used in our experiments. The final configuration of the hyperparameters can be found in Table 2.

	Q-Table		DQN			
	Basic	ER	Basic	Double	Dueling	DoubleDueling
α	.0001	.0001	.0001	.0001	.0001	.0001
γ	.95	.95	.95	.95	.95	.95
ε_{init}	1	1	1	1	1	1
ε_{final}	.01	.01	.01	.01	.01	.01
ε_{decay}	.99995	.9995	.9995	.9995	.9995	.9995
N	1 000 000	300 000	30 000	30 000	30 000	30 000
B	-	128	128	128	128	128
m	-	128	128	128	128	128
M	-	131 072	131 072	131 072	131 072	131 072
D	10	10	-	-	-	-
H	-	-	128	128	128	128
C	-	-	-	1000	-	1000

Table 2: Hyperparameters configuration.

- **Learning rate (α):** Controls the step size in updating Q-values or network weights. A lower α leads to more stable but slower learning, while a higher α allows faster learning but risks instability. After testing different values, a smaller learning rate of .0001 was found to yield the best results in this environment.
- **Discount factor (γ):** Determines the importance of future rewards. A higher γ encourages long-term planning, while a lower γ makes the agent more short-sighted.
- **Initial exploration rate (ε_{init}):** The starting value of the exploration parameter ε in ε -greedy. It controls how much the agent initially explores the environment rather than exploiting known information. Kept as default.
- **Final exploration rate (ε_{final}):** The lowest value that ε can reach, ensuring that the agent retains some degree of exploration even in later stages. It was observed that using lower values of it in this environment is beneficial, as in the later stages of training, even

a single random action can lead to an immediate termination due to the high precision required. Therefore, once the agent has learned sufficiently, it is more effective to focus on exploiting the learned policy to improve performance.

- **Exploration decay factor** (ε_{decay}): The rate at which ε decreases over time. Higher decay values lead to faster convergence to exploitation, while lower values allow longer exploration. Applied as an exponential decay, meaning at each episode, ε is multiplied by the same constant decay factor, with the value clamped by ε_{final} to ensure epsilon does not fall below a minimum threshold. In ER-enabled configurations convergence is reached with fewer episodes (multiple experiences processed for step, rather than just one). Hence, we want the model to reach its final training phase (when $\varepsilon = \varepsilon_{final}$) faster. For this reason, it was decided to speed-up this decay by an order of magnitude in those setups compared to the ER-disabled ones.
- **Number of episodes** (N): The total number of episodes in the training process. An episode consists of the agent interacting with the environment for a set number of steps, until episode termination. As before, N is kept smaller in ER-enabled configurations and bigger otherwise.
- **Mini-batch size** (B): *ER only*. The size of the mini-batch sampled from memory during the training. This determines the number of experiences used at every step for model updates. After trying different values, an appropriate value of 64 has been decided.
- **Minimum memory size before updating** (m): *ER only*. If the ER buffer contains fewer than m transitions, no updates are performed. In the experiments, m is always set equal to B to ensure updating as soon as possible.
- **Maximum memory capacity** (M): *ER only*. The maximum size of the ER buffer, which determines how many past transitions can be stored for learning. This value is kept constant across configurations for a fair comparison.
- **State discretization divisions** (D): *Q-Table only*. The number of divisions performed to transform continuous state features into categorical values. A continuous feature (e.g., the x-position of the next pipe) is thus divided into $D + 1$ bins. Larger D values increase state resolution but can make learning harder due to a larger state space. After trying different values, an appropriate value of 10 divisions (11 bins) has been decided.

- **Hidden layer size (H):** *DQN only*. The number of neurons in the hidden layers of the neural network. For the Dueling configuration it is also the number of neurons in the hidden layer of the value stream and the advantage stream. Kept as default.
- **Network sync rate (C):** *DQN only*. When implementing DQN with an additional target network, this controls how often (number of steps) the target network is synced with the policy network.

4.3 Training Results

For each configuration, we plotted the reward (both current and moving average over the last 100 episodes) and loss (current and moving average over the last 100 episodes) across all training episodes. Additionally, we plotted the epsilon values per episode to visualize its decay and assess the balance between exploration and exploitation.

4.3.1 Q-Table

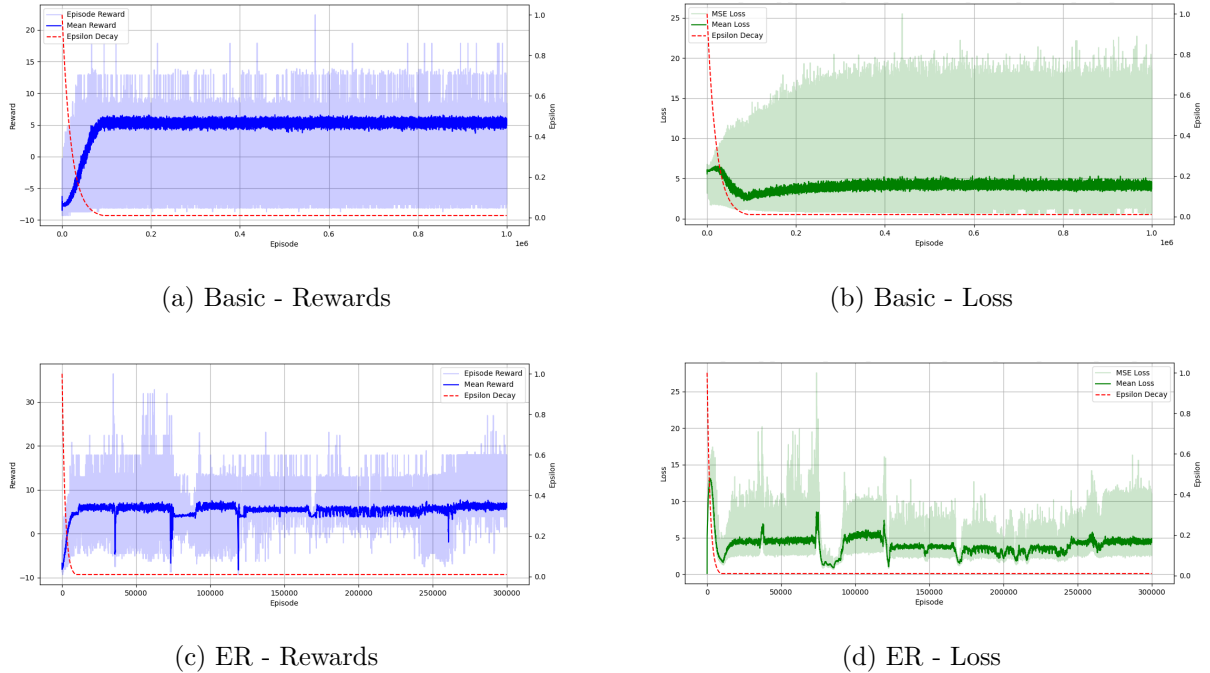


Figure 15: Training results for Q-Table-based configurations.

Figure 15 illustrates the training results for the Q-Table configurations (Basic and ER).

As expected, reward and loss follow the epsilon decay, exhibiting high variability during the exploration phase and becoming more stable as exploitation takes over once epsilon reaches its

final value. Thus, both approaches show slow and steady improvements in the average reward over time, with Q-ER slightly outperforming Q-Basic.

However, a key difference emerges in their stability: in the Q-Basic configuration, the reward fluctuates significantly even in the later stages of training, frequently dropping to low values. This suggests that, even when the agent primarily exploits learned policies (low epsilon), it has not consistently internalized an optimal strategy.

In contrast, the Q-ER configuration exhibits a more stable learning process. The minimum reward progressively increases throughout training, indicating that the agent has learned to avoid particularly poor performances. This difference is also evident in the loss curves: in the Q-Basic configuration, the loss oscillates sharply between low and high values, reflecting unstable updates, whereas in Q-ER, the loss is more stable, supporting the idea that experience replay contributes to a more consistent learning process.

Despite these improvements, both Q-Table approaches remain limited in their ability to reach high rewards, as expected from tabular Q-learning approaches, which struggle with large state spaces.

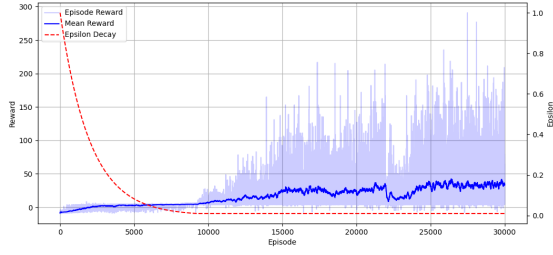
4.3.2 DQN

Figure 16 shows the training results for the DQN-based configurations. The improvements over the Q-Table approaches are immediately evident, with significantly higher rewards achieved.

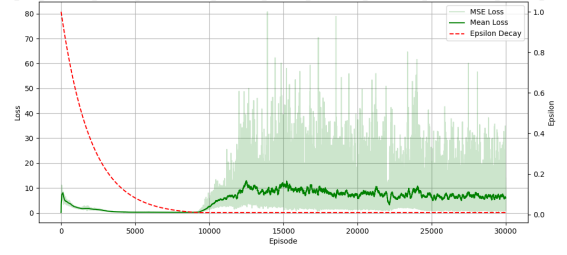
However, unlike what is typically observed in more complex environments, Double-DQN does not provide a clear advantage in this case. The training performance of configurations where this is enabled closely mirrors that of the standard DQN, with similar fluctuations in reward and loss. This suggests that the chosen environment, along with hyperparameters such as a relatively low learning rate, already provides sufficient stability, reducing the need for the overestimation bias correction introduced by Double DQN.

On the other hand, the Dueling DQN architecture leads to a substantial improvement in performance, with both DQN-Dueling and DQN-DoubleDueling significantly outperforming the others. This indicates that separating the advantage and value functions helps the network better distinguish between important and less relevant actions, leading to more effective learning.

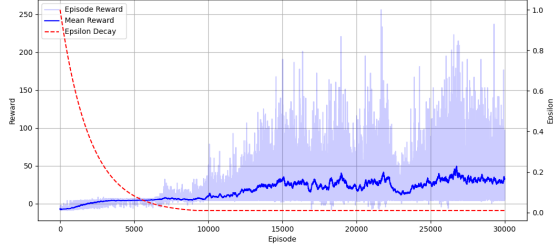
The loss trends across all DQN models reflect the expected pattern: initial high volatility due to extensive exploration, followed by gradual stabilization as epsilon decays and learning consolidates. However, occasional spikes in loss suggest that the models still encounter difficult learning transitions.



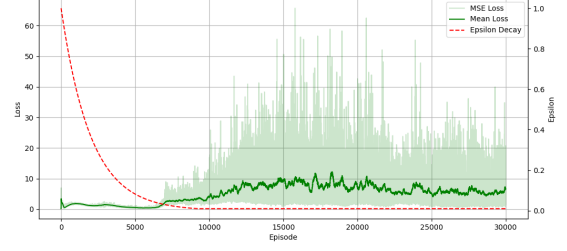
(a) Basic - Rewards



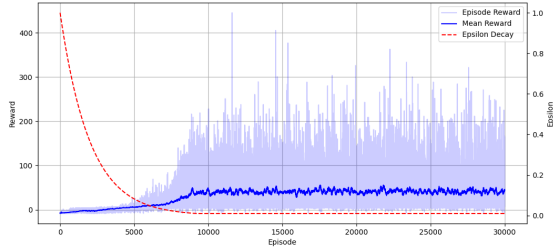
(b) Basic - Loss



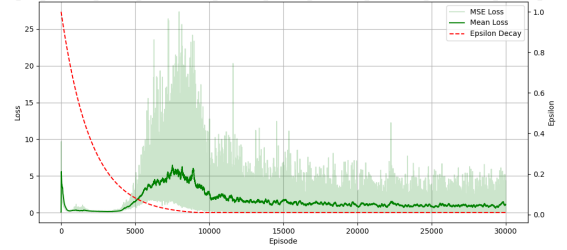
(c) Double - Rewards



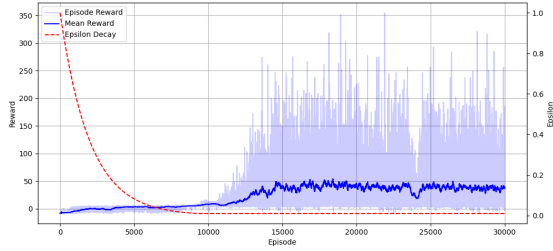
(d) Double - Loss



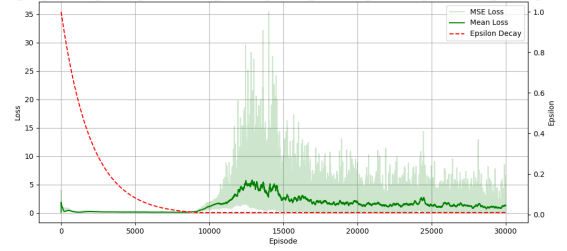
(e) Dueling - Rewards



(f) Dueling - Loss



(g) DoubleDueling - Rewards



(h) DoubleDueling - Loss

Figure 16: Training results for DQN-based configurations.

4.3.3 Comparison

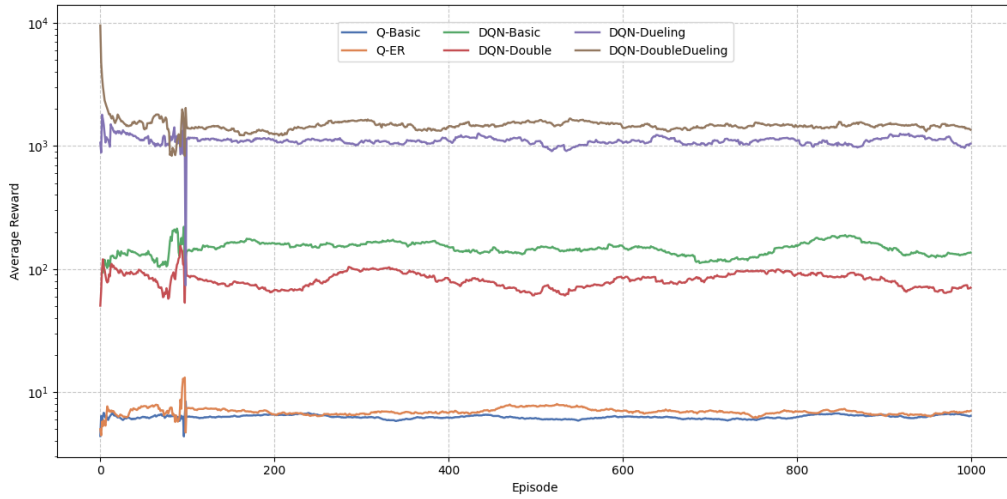
Comparing the two approaches, DQN-based models clearly outperform tabular Q-learning, achieving higher rewards and greater learning stability. The Q-Table methods, despite gradual improvements given by ER, are fundamentally limited by their discrete representation of state-action values and struggle to scale effectively in this environment.

Among the DQN variants, the Dueling architecture proves particularly effective, significantly

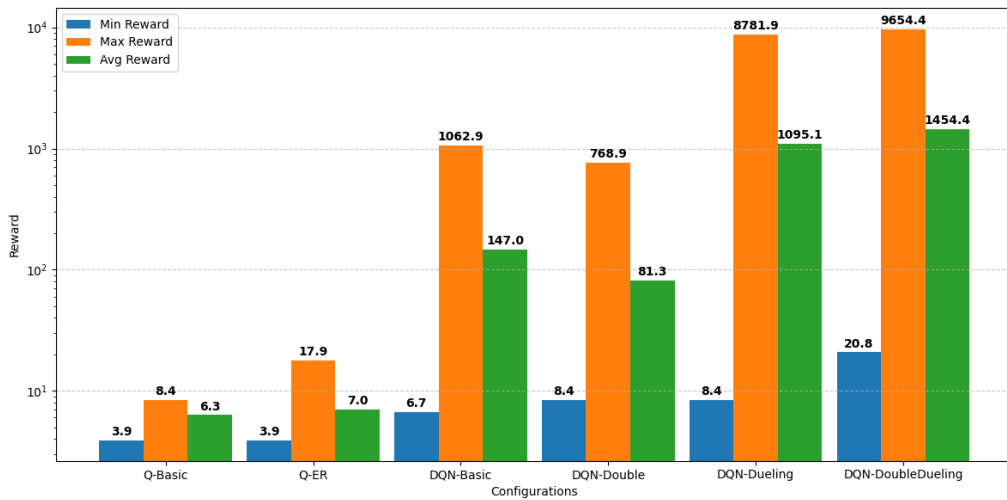
boosting training performance by helping the agent better differentiate valuable states. Conversely, Double DQN does not provide substantial benefits in this setting, likely because the environment’s inherent stability, due to factors like a low learning rate, already help to mitigate overestimation bias.

4.4 Test Results

After training, each configuration was tested over 1000 episodes to evaluate the effectiveness of the learning process. Figure 17a presents the moving average reward (last 100 episodes) for all configurations, while Figure 17b provides a bar plot showing the minimum, maximum, and average rewards achieved during testing.



(a) Mean Rewards per episode (on log scale)



(b) Rewards per configuration (on log scale)

Figure 17: Test results for Q-Table and DQN-based configurations (1000 episodes).

The test results align with the trends observed during training.

Q-Table configurations achieve the lowest rewards, with Q-ER performing slightly better than Q-Basic. However, their maximum rewards remain below 20, highlighting their inherent limitations. In contrast, all DQN-based approaches achieve significantly higher rewards, with DQN-DoubleDueling reaching the highest maximum and average rewards (9654.4 and 1454.4, respectively), almost two orders of magnitude higher than the best Q-Table method (17.9 and 7.0) and one order higher than DQN configurations without the Dueling architecture (1062.9 and 147.0).

The logarithmic scale in Figure 17 effectively illustrates the performance gap between approaches, emphasizing the dramatic improvements enabled by DQN techniques compared to Tabular Q-Learning.

The reward distribution among the DQN variants is the following:

- **DQN-Basic** achieves decent performance but remains unstable.
- **DQN-Double** does not provide a clear improvement over the basic variant, performing slightly worse on average. However, it does achieve a better minimum reward, contributing to more consistent worst-case performance.
- **DQN-Dueling** significantly enhances learning efficiency, achieving rewards almost nine times higher than DQN-Basic.
- **DQN-DoubleDueling** achieves the best overall performance across all metrics (minimum, maximum, and average rewards). The combination of Double and Dueling architectures appears to provide both stability in worst-case scenarios and higher reward optimization, underscoring the effectiveness of this approach.

In summary, these results confirm that integrating Dueling DQN leads to substantial performance gains, while the advantages of Double DQN are effective only for improving worst-case performance. Tabular Q-learning struggles to scale effectively, whereas DQN approaches, especially those implementing Dueling optimization, demonstrate remarkable improvements in learning efficiency and reward optimization.

5 Conclusions

This project successfully applied Reinforcement Learning (RL) approaches to train an agent capable of effectively playing *Flappy Bird*. The focus was on comparing Tabular Q-Learning and Deep Q-Networks (DQN) to assess their performance when implemented from scratch for a real-world game-playing task.

As expected, DQN-based methods outperformed Q-Table approaches due to their ability to approximate the Q-function using a neural network, which allows for handling larger and more complex state spaces. While Q-Table methods struggled with scalability and volatility in results, DQN models demonstrated higher and more stable rewards, highlighting the advantages of function approximation.

Incorporating Experience Replay (ER) into the Q-Table method led to more stable learning, slightly improving reward consistency and reducing volatility. However, the performance of Q-Table-based approaches remained limited due to the state space's size and the inherent restrictions of tabular methods in complex environments.

Regarding DQN optimizations, the addition of Double DQN showed minimal impact, mainly improving the minimum rewards obtained during testing. This suggests that the environment's complexity and selected hyperparameters already provided enough stability, rendering the bias correction typically offered by Double DQN unnecessary in this case.

The most significant improvements were observed with the Dueling DQN configurations, where rewards were consistently higher and more stable compared to other methods. The separation of value and advantage streams in the Dueling DQN architecture proved crucial in enhancing learning efficiency, enabling the agent to focus on which states were most valuable.

Ultimately, DQN with Dueling, especially when combined with Double DQN, emerged as the most effective strategy for this environment, while ER provided useful enhancements for Q-Table methods.

The RL agent now outperforms the average human player and likely exceeds the performance of skilled players, showcasing the success of the project. While further hyperparameter tuning and additional optimizations could improve the results, we are highly satisfied with the current outcomes, as they meet the project's objectives and demonstrate the potential of RL in game-playing tasks.

References

- [1] Aishwarya Hastak. Reinforcement learning. <https://aishwaryahastak.medium.com/reinforcement-learning-bb34d8a369a2>, October 2024. [Online; accessed 13-February-2025].
- [2] Martin Kubovčík. Flappy bird for gymnasium. <https://github.com/markub3327/flappy-bird-gymnasium>, 2024.
- [3] Artem Dogtiev. Flappy bird revenue – how much did flappy bird make? <https://www.businessofapps.com/data/flappy-bird-revenue/>, May 2022. [Online; accessed 12-February-2025].
- [4] Wikipedia contributors. Flappy bird - Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Flappy_Bird&oldid=1275012558, 2025. [Online; accessed 12-February-2025].
- [5] Joseph Bernstein. Why on earth is this borderline crappy, impossibly hard game the most popular download on the app store? https://www.buzzfeednews.com/article/josephbernstein/why-on-earth-is-this-borderline-crappy-impossibly-hard-game?utm_source=dynamic&utm_campaign=bfsharecopy, January 2014. [Online; accessed 13-February-2025].
- [6] Markel Sanz Ausin. Introduction to reinforcement learning. part 4: Double dqn and dueling dqn. <https://markelsanz14.medium.com/introduction-to-reinforcement-learning-part-4-double-dqn-and-dueling-dqn-b349c9a61ea> April 2020. [Online; accessed 19-February-2025].