Playing ATARI with Reinforcement Learning

Comprehensive Analysis and Implementation of a Pioneering Research Paper in Deep Reinforcement Learning

Hephaestus Applied Artificial Intelligence Association

Authors:

Member	Role
Leonardo Tonelli	Head
Federico Scaffidi Muta	Member
Velina Todorova	Member
Federica Kulka	Member

Milan, February 15, 2025

Contents

1 | Introduction

1.1 | Motivations

This study focuses on the papers "Human-level control through deep reinforcement learning" [8] and its predecessor, "Playing Atari with Deep Reinforcement Learning"[7], both[1] authored by a team of scientists at **Google DeepMind**, which have had a significant impact on the fields of Computer Science, Robotics, and many others. The goal of this project is to highlight the **importance** of these papers, examining their **impact** while providing a fully **functional implementation** of the techniques discussed in the research (link to GitHub). The papers introduced a new, efficient Reinforcement Learning architecture, trained and tested specifically on a variety of Atari games using an emulator. This same architecture could be applied across all the games from Atari, achieving astonishing performance, surpassing expert human-level performance in most games . The papers present algorithmic solutions to challenges arising from applying **deep learning** to Reinforcement Learning problems, paving the way for future applications and techniques to address these types of challenges. The reading is recommended for those with a basic understanding of neural networks, probability theory, and algorithmic thinking. We highly recommend complementing the reading of Chapter 3 **with the code** available on GitHub (link to GitHub). Before diving in the paper, we will first introduce the Reinforcement Learning framework and the standard classification of algorithms used to address such problems.

1.2 | Reinforcement Learning Framework and Terminology

Reinforcement Learning (RL) is an essential branch of machine learning that intersects control theory and artificial intelligence. It is primarily a framework devised to teach an agent how to interact with a complex environment through experience, utilizing trial and error. This approach draws inspiration from biological systems, where animals, including humans, learn behaviors through interactions, feedback, and rewards.
Reinforcement learning revolves around the concept of an **agent** interacting with an **environment** to achieve a specific goal. The agent learns from the outcomes of its actions, continuously refining its strategy to maximize cumulative **rewards** over time. In this context, the learning process is driven by positive reinforcement (rewards for desirable actions) and negative reinforcement (penalties for undesired behaviors). This mirrors real-life training, such as conditioning a pet with treats.

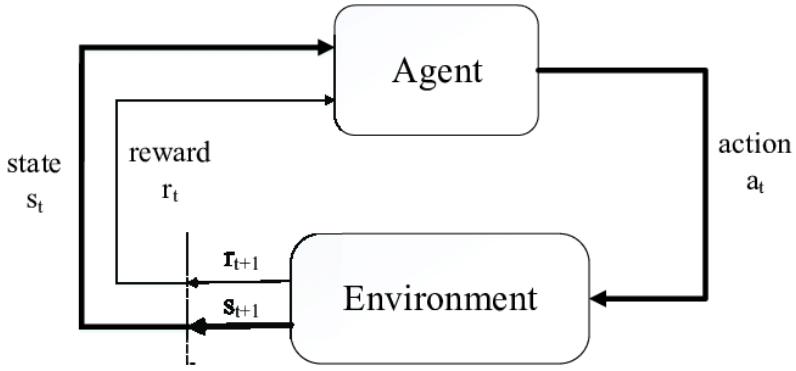


Figure 1.1: The Reinforcement Learning Framework. Source: [13]

The RL framework consists of several key components that define the agent's interaction with its environment. The agent is the decision-maker, which observes the environment and takes actions based on those observations. The state represents the current situation or context perceived by the agent, while the action refers to the choices available to the agent in that state. Each action yields a reward, which serves as feedback on the action's desirability. The goal is to **maximize the total reward** over time.
The agent's behavior is guided by a **policy** (π), which is essentially a strategy mapping states to actions. The **value function (V)** assesses the long-term reward potential of being in a given state while following a particular policy. Complementarily, the **Q-function (Q)** evaluates the quality of taking a specific action in a given state, extending the value function by considering both states and actions. The central challenge in RL is for the agent to learn an optimal policy that maximizes its expected cumulative reward.

[1] Throughout the report, we will refer to "the paper" to indicate one of the two related studies

Unlike supervised learning, which relies on explicit labels, RL often involves delayed and sparse feedback, making it more challenging to optimize the agent's behavior effectively.
To demonstrate the RL framework, consider a scenario where a mouse must navigate a maze. Here, the mouse acts as the agent, and the maze represents the environment. The mouse can only perceive its immediate surroundings, which form its current state. Based on this perception, it decides which action to take next, such as turning left, right, or moving forward. Rewards are sparse in this scenario: the mouse only receives a positive reward, such as finding a piece of cheese, upon successfully reaching the end of the maze.
This example highlights a critical issue known as the **credit assignment problem**. Since the reward is only given at the end, it becomes difficult for the agent to discern which specific actions contributed to reaching the goal. This issue is pervasive in many RL problems where rewards are delayed and sparse, complicating the optimization process.
In modeling the RL framework, we often assume a critical property known as the **Markov property**, which states that the future state of the system depends only on the current state and action, independent of past events. This assumption allows us to formalize the RL problem using a **Markov Decision Process (MDP)**, a mathematical framework for sequential decision-making in uncertain environments. An MDP is defined by a set of states, available actions, a reward function, and transition probabilities that determine the likelihood of moving from one state to another given a chosen action. By adhering to the Markov property, MDPs provide a structured way to model dynamic systems where decision-making is based solely on the present state.
In a Markov Decision Process (MDP), the **discount factor (γ)** plays a fundamental role in balancing immediate and future rewards. This factor, ranging between 0 and 1, determines how future rewards are valued relative to immediate ones. The concept of discounting future rewards is deeply rooted in behavioral economics and psychology, where it is observed that individuals often prefer immediate rewards over delayed ones, a phenomenon known as *time discounting*. Research indicates that present rewards are weighted more heavily than future ones, and as the delay to receiving a reward increases, its perceived value decreases ([4]). This tendency is captured in MDPs through the discount factor, which reduces the weight of future rewards, aligning the model with observed human behavior in decision-making scenarios.
In reinforcement learning, the value of a state s is defined as the expected cumulative reward an agent receives when starting from that state and following a given policy. This is formally expressed as:

$$V(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t\right] \tag{1.1}$$

where $V(s)$ represents the value function of state s, γ is the discount factor, and R_t is the reward received at time step t. The discount factor γ ensures that rewards received sooner contribute more to the value of a state than those received further in the future, aligning with observed human preferences for immediate rewards ([4]).
One of the most significant challenges in RL is finding the right balance between **exploration** and **exploitation**. Exploration involves trying new actions to discover their effects, which can potentially yield higher rewards in the long term. In contrast, exploitation focuses on leveraging known actions that have previously produced favorable outcomes. Striking the right balance is essential for optimizing long-term rewards; excessive exploitation can lead to suboptimal strategies, while too much exploration may waste resources.

1.3 | Algorithms classification

Reinforcement Learning algorithms can be categorized based on various criteria, each highlighting different aspects of the learning process. One common classification distinguishes between **model-free** and **model-based** methods. Model-free algorithms, such as Q-learning and Policy Gradient methods, learn optimal policies directly from interactions with the environment without constructing an explicit model of the environment's dynamics. In contrast, model-based algorithms develop an internal model of the environment's transition dynamics and rewards, enabling planning and decision-making by simulating future states. This distinction is crucial, because model-based methods often achieve higher sample efficiency by leveraging their internal models, while model-free methods may require more interactions to learn effective policies.
Within the class of model-free methods, a further distinction can be made between **Monte Carlo methods** and **Temporal Difference (TD) learning**. Monte Carlo methods, such as first-visit and every-visit Monte Carlo, estimate value functions by averaging returns from complete episodes, which

makes them well-suited for episodic tasks where learning can occur after an episode has fully unfolded. In contrast, TD learning methods update value estimates incrementally using bootstrapping, estimating values based on other estimates rather than waiting for complete returns. This allows TD methods to learn from incomplete episodes, making them more applicable to continuous and online learning environments.
Another classification approach focuses on the **value-based**, **policy-based**, and **actor-critic** paradigms. Value-based methods, such as Deep Q-Networks (DQN), aim to learn the optimal value function, which estimates the expected return of states or state-action pairs, and derive policies by selecting actions that maximize this value. Policy-based methods, on the other hand, directly parameterize and optimize the policy without explicitly learning a value function, which can be an advantage in high-dimensional or continuous action spaces. Actor-critic methods combine these approaches by maintaining both a value function (critic) to evaluate actions and a policy (actor) to select actions, facilitating more stable and efficient learning.
Furthermore, RL algorithms can be classified based on their data utilization strategies into **on-policy** and **off-policy** methods. On-policy algorithms, such as SARSA, learn the value of the policy currently being executed, necessitating exploration during learning. Off-policy algorithms, such as Q-learning, evaluate and improve a target policy different from the behavior policy used to generate data, allowing the use of past experiences and data generated by other policies to enhance learning efficiency.

2 | Theory

2.1 | Q-learning

Q-learning is a model-free reinforcement learning algorithm that aims to find the optimal action-selection policy by estimating the optimal action-value function $Q^*(s, a)$. The optimal Q-value function is defined as the maximum expected cumulative reward that an agent can achieve by following the best policy π, where future rewards are discounted by γ to account for their decreasing importance over time.

$$Q^*(s, a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \mid s_t = s, a_t = a, \pi\right] \tag{2.1}$$

Every time the agent takes an action, it receives a reward r and transitions to a new state s'. The goal is to update the Q-value for the state-action pair (s, a) based on this new experience. The update is made by comparing the current Q-value estimate with a new estimate, which combines the immediate reward r the agent just received, along with the expected future reward. The expected future reward is determined by the maximum Q-value of the next state s', reflecting the best possible action the agent can take from that state.
The standard Q-learning procedure follows these steps:

1. **Initialize** the Q-table arbitrarily (or with zeros).
2. **Observe the state** s.
3. **Select an action** a, typically using an exploration-exploitation strategy like ϵ-greedy.
4. **Execute the action** and receive the reward r and the next state s'.
5. **Update the Q-value** using the update rule.
6. **Repeat** the process until convergence or a stopping condition is met.

2.2 | Deep Q-Network

The paper presents **Deep Q-Networks (DQN)**, a reinforcement learning approach that integrates **deep neural networks** with **Q-learning** by using a deep neural network as a function approximator to estimate the Q-values. This integration allows DQN to overcome the limitations of traditional Q-learning in high-dimensional environments by generalizing across similar states rather than relying on a tabular representation, enabling artificial agents to learn control policies directly from high-dimensional sensory inputs. The **DQN agent** in the paper surpassed previous reinforcement learning algorithms and achieved a level comparable to that of a professional human games tester across a set of **49 Atari 2600 games**, using the same algorithm, network architecture and hyperparameters. The aim of reinforcement learning is to learn a very complex mapping from states to actions that maximizes cumulative reward over time. DQN is particularly useful in this case because video games have high-dimensional state spaces (raw pixel data) and sparse rewards, making traditional RL techniques inefficient. By leveraging deep learning, DQN extracts meaningful features from raw inputs, allowing it to learn effective policies without the need for handcrafted features. This ability to process raw visual input and determine optimal actions makes DQN applicable to various real-world problems. Earlier methods required using handcrafted features or low-dimensional state spaces but the DQN operates **end-to-end**, receiving only raw pixel data and game scores as input. This made it one of the first AI agents capable of learning diverse tasks from experience alone.

2.3 | Problems

DQN addresses the following key challenges in reinforcement learning (RL):

1. **High-dimensional input spaces** – Traditional RL models struggle with processing raw sensory data and requires feature engineering. Their applicaton is limited to domains which features can be handcrafted or to domains with low-dimensional fully observed state spaces. DQNs bridge the jump between high-dimensional sensory inputs and actions - it can successfully learn policies directly from the inputs using end-to-end RL. The solution is leveraging convolutional neural networks (CNNs). By using CNNs for automatic feature extraction and allowing end-to-end learning, DQN effectively

solves the problem of high-dimensional input spaces and opens up RL to more complex environments. This innovation makes DQN more applicable to a wide range of real-world tasks, from video games to robotics.

2. **Generalization** – The basic approach of RL is often impractical because the action-value function is estimated separately for each state-action pair in **tabular Q-learning**. In tabular Q-learning, the algorithm stores a separate Q-value for each possible state-action pair, making it infeasible for environments with large or continuous state spaces. For example, in an Atari game, where the state space consists of raw pixel inputs, storing individual Q-values for each state would be impossible due to the vast number of possible states. Instead, generalization in DQN is achieved by using a function approximator to estimate the action-value function,

$$Q(s, a; \theta) \text{ such that } Q(s, a; \theta) \approx Q(s, a)$$

   Instead of maintaining a separate Q-value for each state-action pair, the neural network learns a shared representation across different states, enabling it to generalize to unseen situations. This allows DQN to scale efficiently to complex environments where exact Q-value tables would be infeasible.

3. **Training instability** – RL is unstable or even sometimes diverges when a NN is used to represent the action-value function (Q). This happens because of the correlations in the sequence of observations, the fact that small updates to Q may significanlty change the policy (therefore the data distribution and the action values vs. target values correlations).
   DQN solves these instabilities by integrating two key ideas - experience replay and iterative update that adjusts the action-values (Q). Experience replay is a biologically inspired mechanism[3] that randomizes over the data which helps remove correlations in the observation sequence and smooth over changes in the data distribution. By using the second approach the action-values are adjusted towards target values that are only periodically updated which reduces correlations with the target.

   Other stable methods (like Neural fitted Q-iteration) exist but these methods are inefficient when used with large neural networks because they repeat the training of networks over hundreds of iterations.

2.4 | Mathematical Formulation of DQN

The agent's objective is to **maximize cumulative future rewards**. Formally, it learns an **optimal action-value function** $Q^*(s, a)$:

$$Q^*(s, a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \mid s_t = s, a_t = a, \pi\right] \tag{2.2}$$

where:

- s **(State) - Vector**
  Represents the current **state** of the environment at a given time step. In many practical applications (e.g., Atari games), the state is represented as a feature vector or an image.
  Data type: Vector (continuous or discrete, depending on the environment).

- a **(Action) - Discrete Value**
  The **action** chosen by the agent from a set of possible actions. In DQN, the action space is typically finite (e.g., move left, right, up, down).
  Data type: Integer (discrete).

- r_t **(Reward) - Number**
  The **reward** received by the agent at time step t after executing an action. This scalar value indicates whether an action was beneficial or not.
  Data type: Float or Integer (scalar).

- π **(Policy) - Function**
  A function that maps a given state s to a probability distribution over actions. The optimal policy maximizes expected cumulative rewards.
  Data type: Function.

- γ **(Discount Factor) - Constant**
  A scalar value between 0 and 1 that determines how much future rewards are valued compared to immediate rewards.
  Data type: Float (constant, typically 0.9 or similar).

- $Q^*(s, a)$ **(Optimal Action-Value Function) - Function**
  Represents the **expected cumulative reward** the agent can obtain from state s by taking action a and following the optimal policy thereafter.
  Data type: Function (returns a scalar value).

- $\mathbb{E}[\cdot]$ **(Expectation Operator) - Mathematical Operator**
  Denotes the **expected value**, considering the stochastic nature of state transitions and rewards.
  Data type: Operator.

- $\sum_{t=0}^{\infty} \gamma^t r_t$ **(Discounted Reward Sum) - Number**
  Represents the cumulative discounted reward over time. The discount factor γ reduces the impact of rewards obtained in the distant future.
  Data type: Float (scalar).

DQN approximates $Q^*(s, a)$ using a **deep convolutional neural network (CNN)** parameterized by weights θ:

$$Q(s, a; \theta) \approx Q^*(s, a) \tag{2.3}$$

2.5 | Key Innovations in DQN

The paper introduces three main techniques to stabilize training and improve learning efficiency:

1. **Experience Replay:** Instead of learning from **consecutive observations**, which are highly correlated, DQN stores past experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ in a **replay memory** D and samples **mini-batches** during training:

   $$D = \{e_1, e_2, ..., e_N\} \tag{2.4}$$

   This removes **temporal correlations** and improves stability.

2. **Target Network Stabilization:** A **separate target Q-network** with weights θ^- is updated **less frequently** than the main network θ, reducing oscillations. The Q-learning loss is:

   $$L(\theta_i) = \mathbb{E}_{(s,a,r,s')} \left[(y_i - Q(s, a; \theta_i))^2 \right] \tag{2.5}$$

   where the target value is:

   $$y_i = r + \gamma \max_{a'} Q(s', a'; \theta^-) \tag{2.6}$$

   The reason for using this target value is that, in standard Q-learning, the Q-values are updated using the current network's estimates, which introduces instability because the target values change at every iteration. This leads to divergence due to high correlations between predictions and targets. By maintaining a **separate target network** that updates less frequently, DQN ensures that the target values remain more stable, making learning more reliable.

   The target network is updated every C steps, as part of the previously mentioned **iterative update** strategy. This iterative adjustment helps reduce correlations between the estimated Q-values and their targets, contributing to stable convergence.

3. **Reward Clipping:** To ensure **consistency across games**, rewards are clipped between -1 and +1:

   $$r_{\text{clipped}} = \max(-1, \min(r, 1)) \tag{2.7}$$

   This prevents extreme rewards from dominating training, making DQN **more robust**.

3 | Implementation

3.1 | General Details

Our implementation of the Atari Reinforcement Learning experiment was developed in Python, using key libraries such as PyTorch for deep learning, OpenAI Gymnasium for environment simulation, and ALE-Py for interfacing with Atari 2600 games. The core architecture follows the Deep Q-Network (DQN) approach, with a convolutional neural network (CNN) to approximate the Q-value function. Preprocessing was crucial to enhance learning efficiency, including frame resizing to 84×84 pixels, frame skipping (set to 4), grayscale conversion, and action-repeat mechanisms using the AtariPreprocessing wrapper. The goal of our implementation was to replicate the DQN training pipeline, optimize hyperparameters, and analyze how different architectural choices impact learning efficiency and performance in Atari games, specifically Demon Attack.
The implementation follows the Deep Q-Network (DQN) framework to train an agent in a reinforcement learning (RL) environment. The algorithm utilizes a deep neural network with convolutional layers for image-based input, followed by fully connected layers to output Q-values for all possible actions. The goal is to maximize cumulative rewards over time by learning a policy that maps states to actions. Through a process of experience replay and Q-network updates using the epsilon-greedy policy, the agent gradually learns to predict optimal actions. The main components are:

- **model.py**: Defines a convolutional neural network (CNN) to approximate the Q-value function.
- **agent.py**: Implements the agent, including action selection, experience replay, and Q-value updates.
- **train.py**: Runs the training loop, interacting with the environment and updating the agent.
- **training.py**: Initializes the hyperparameters, the environment and the model for training.
- **agent-Q.pth**: Stores the model weights after training.
- **eval.py**: Tests the trained model with the chosen game (in our case, Demon Attack).

3.2 | Algorithm

Algorithm 1 Deep Q-Learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode $= 1$ **to** M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1$ **to** T **do**
    With probability ϵ, select a random action a_t
    Otherwise, select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t, image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ w.r.t. θ
    if $t \mod C = 0$ **then**
      Reset $\hat{Q} = Q$ (i.e., set $\theta^- = \theta$)
    end if
  end for
end for

1. The Deep Q-Learning with Experience Replay algorithm follows a structured process to train a reinforcement learning agent using a Deep Q-Network (DQN).

2. It begins by initializing a replay memory D to store past experiences and defining two neural networks: the primary Q-network Q with weights θ and a target Q-network $\hat{Q}$ with weights θ^- , which are periodically updated to improve training stability.
3. For each episode, the agent starts from an initial state, and for each time step, it selects an action ϵ-greedily—either choosing a random action with probability ϵ for exploration or selecting the action with the highest Q-value otherwise. The action is executed in the environment, and the resulting transition (state, action, reward, next state) is stored in the replay memory.
4. The agent periodically samples random minibatches from this memory to break temporal correlations and stabilizes training. The Bellman equation is used to compute target Q-values y_j , considering whether the episode terminated or not.
5. The model is then updated using gradient descent to minimize the mean squared error (MSE) loss between the predicted Q-values and the targets. Additionally, every C steps, the target network $\hat{Q}$ is synchronized with the main network Q to stabilize learning.

3.3 | Model and Agent Architecture

The **Q-network** in DQN is a convolutional neural network (CNN) that maps input states (preprocessed frames) to Q-values for each action. The architecture consists of:

- **Convolutional Layers**: Extract spatial features (such as edges and textures) from the input state.
- **Fully Connected Layers**: Process extracted features to output Q-values.
- **Activation Functions**: ReLU (Rectified Linear Unit) is used to introduce non-linearity and improve convergence.
- **Loss Function**: Huber loss is used instead of mean squared error (MSE) to handle large temporal-difference errors robustly.
- **Optimizer**: RMSprop is used for efficient weight updates.

A typical architecture for processing visual input (e.g., Atari games) is:

Layer	Type	Parameters	Activation
Input	RGB Frames	84x84x4	-
Conv1	Conv2D	32 filters, 8x8 kernel, stride 4	ReLU
Conv2	Conv2D	64 filters, 4x4 kernel, stride 2	ReLU
Conv3	Conv2D	64 filters, 3x3 kernel, stride 1	ReLU
FC1	Fully Connected	512 units	ReLU
Output	Fully Connected	**num_actions**	Linear

Table 3.1: Architecture of the Deep Q-Network (DQN)

The forward pass can then be described as follows.
We begin with an automatically-preprocessed input image, which is passed through a sequence of the three convolutional layers. Each convolutional layer applies filters to detect spatial features (such as edges and textures), followed by ReLU activation to introduce non-linearity. After the convolutional layers, the extracted feature maps are flattened into a one-dimensional tensor. Then the representation is processed by a fully connected layer that learns high-level representations of the input state. Lastly, the final output layer produces Q-values for all possible actions, whose number is num-actions. The agent then either uses these Q-values to select the action with the highest expected reward, or chooses a random action.
The **agent** in DQN is responsible for interacting with the environment, selecting actions, and updating its Q-network. It follows an **epsilon-greedy strategy**, where it explores by selecting random actions with probability ε and exploits learned knowledge otherwise. Over time, ε decays to favor exploitation over exploration. The decay is exponential, allowing for faster initial exploration and a more gradual transition to exploitation, preventing premature convergence to suboptimal policies. This method provides a more adaptive exploration-exploitation tradeoff.
The Agent class implements key methods that enable the DQN to interact with the environment, store experiences, and update its Q-values.

- The **get_action** method follows an ϵ-greedy policy, selecting a random action with probability ϵ for exploration and choosing the action with the highest Q-value otherwise.
- The **store_memory** method manages the experience replay buffer, ensuring efficient training by storing transitions while applying reward clipping to stabilize learning.
- To update the Q-network, the **update_Q** method samples a minibatch from the replay buffer, computes target Q-values using the Bellman equation, and performs gradient descent to minimize the loss. Additionally, the update-Q-at() method synchronizes the target network with the main Q-network every few steps, which helps stabilize training.
- The **decay_epsilon** method progressively reduces exploration over time following an exponential decay schedule.

These methods collectively ensure efficient learning and prevent instability in training by leveraging experience replay, target networks, and adaptive exploration strategies.

3.4 | Training Strategy

- **Batch Updates**: The Q-network is updated using minibatches of transitions, rather than single experiences, to improve generalization.
- **Target Network Updates**: The target network is updated periodically (not every step), reducing training instability.
- **Discount Factor (γ)**: Determines how much future rewards influence current Q-values. Our choice is $\gamma = 0.99$.
- **Epsilon Decay**: ε is annealed exponentially from 1.0 to 0.1 over training to shift from exploration to exploitation.

By combining these techniques, the agent efficiently learns optimal policies in complex environments.

4 | Experimental Results and Considerations

4.1 | Training Details

Our implementation aims to replicate the training conducted in the paper for the specific game "Demon Attack" adhering to the **hyperparameters** outlined in Table 4.1 and the **algorithm** previously defined in this report (Algorithm 1). The primary limitation of our replication, as expected, stems from **constrained computational resources**, which restricted us to training on a modest 1000 episodes[2] (corresponding to approximately 3 hours of training). While this number may initially seem large, it is quite limited compared to the approximately 150000 episodes[3] used in the original work. This constraint significantly distanced our results from those obtained in the paper. Nevertheless, it did not prevent us from gaining insights from training progress data, which indicated a **healthy learning** process of the DQN.

Table 4.1: Hyperparameters for Deep Q-Learning with Experience Replay

Hyperparameter	Value	Description
minibatch size	64*	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	1,000,000	SGD updates are sampled from this number of most recent frames.
target network update frequency	10,000	The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter C from Algorithm 1).
discount factor	0.99	Discount factor γ used in the Q-learning update.
action repeat	4	Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.
update frequency	4	The number of actions selected by the agent between successive SGD updates.
learning rate	0.00025	The learning rate used by RMSProp.
squared gradient momentum	0.99*	Squared gradient (denominator) momentum used by RMSProp.
min squared gradient	1e-08*	Constant added to the squared gradient in the denominator of the RMSProp update.
initial exploration	1	Initial value of ε in ε-greedy exploration.
final exploration	0.1	Final value of ε in ε-greedy exploration.
replay start size	50,000	A uniform random policy is run for this number of frames before learning starts.
no-op max	30	Maximum number of "do nothing" actions performed by the agent at the start of an episode.

*Parameters indicated with the asterisks are the one that differ from the original paper, based on our limited fine tuning.

4.2 | Evaluation

Although our training sample is significantly smaller than that of the study, the plots (Figure 4.1) show an upward trend in the average Q-value and a noisy improvement in total rewards per episode from episode 600 up to 1000. These observations suggest that the training is progressing in the **right direction**, despite our inability to test on the full sample size used in the paper.

[2] An episode is defined as a full run of the game until termination

[3] The exact number of episodes is not specified in the paper; however, they report using 60 million frames, which corresponds to 150000 episodes assuming a reasonable 400 frames per episode

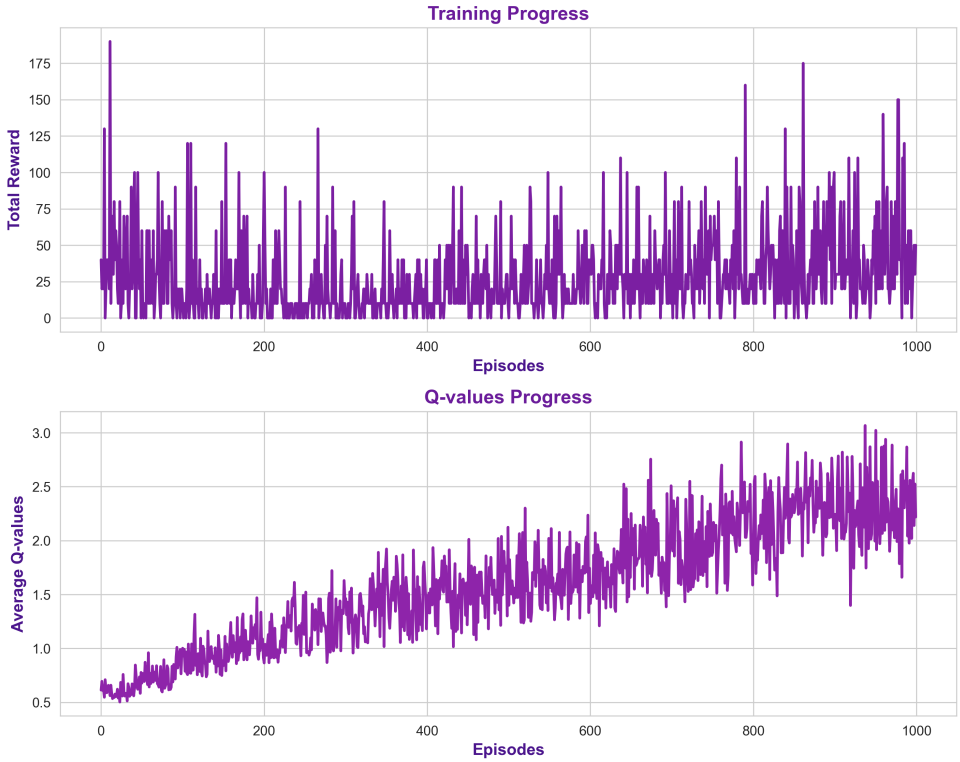
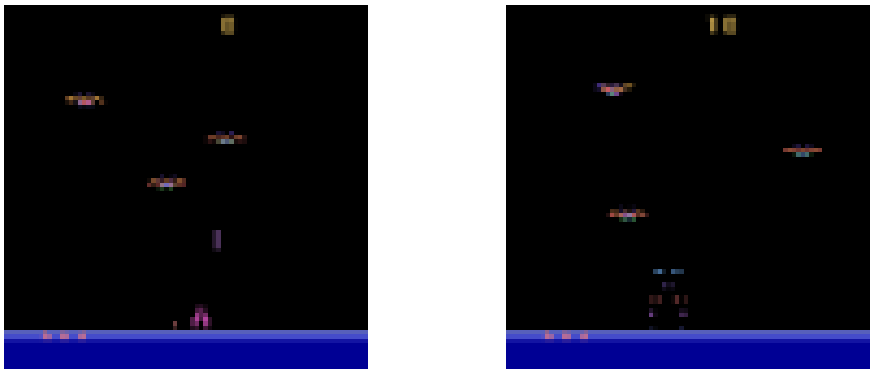


Figure 4.1: Training progress

After completing the training, we ran 10 evaluation games, fixing the exploration rate ϵ at 0.01 to maintain a good balance between exploration and exploitation.
The obtained rewards are not directly comparable to those in the paper. However, observing the simulations, we notice that the agent's movements have become more natural and rational compared to a random walk, further indicating the **correctness** of our implementation.
During the model evaluation, we captured frames with the highest and lowest Q-values (Figure 4.2) to assess whether the model had learned meaningful information. Below, we present these two frames, however already preprocessed to the compressed resolution. Despite the low quality of the images, we can still analyze the dynamics of the situation and describe them.



(a) Frame for highest Q-value

(b) Frame for the lowest Q-value

Figure 4.2: Collected frames in the evaluation runs

We observe that in the frame with the highest Q-value, the agent has just fired its beam, and the beam is about to hit a "demon spaceship" indicating that the agent **correctly anticipates** an expected spike in the game's value. On the other hand, the frame with the lowest Q-value corresponds to the exact moment when the agent is hit by an enemy, marking the end of the episode and a correct prediction of the low value for this state.

4.3 | Considerations and Paper's results

We acknowledge the limitations of our training, including the constrained computational resources, and recognize that our version is **not the most efficient**. However, we are content with this, as the focus of the article, as stated in the introduction, was on studying the paper and the DQN, while gaining hands-on experience, rather than achieving the fastest or most optimal implementation. Nevertheless, our implementation shows signs of healthy training and positive progress in the agent's learning. This is **sufficient for the purposes of our article**. Using the same architecture, the researchers at DeepMind achieved remarkable results. The DQN agent performed at a level comparable to that of a professional human game tester across a set of 49 games, reaching more than 75% of the human score in over half of the games.
Specifically, for Demon Attack, the model outperformed the human expert level, having an average evaluation score almost 3 times higher than the human. The model in the paper even reached a point where it had **learned strategies** that only highly experienced players would consider. In the following figure (Figure 4.3), you can observe the level of understanding attained by the well-trained model in the paper.

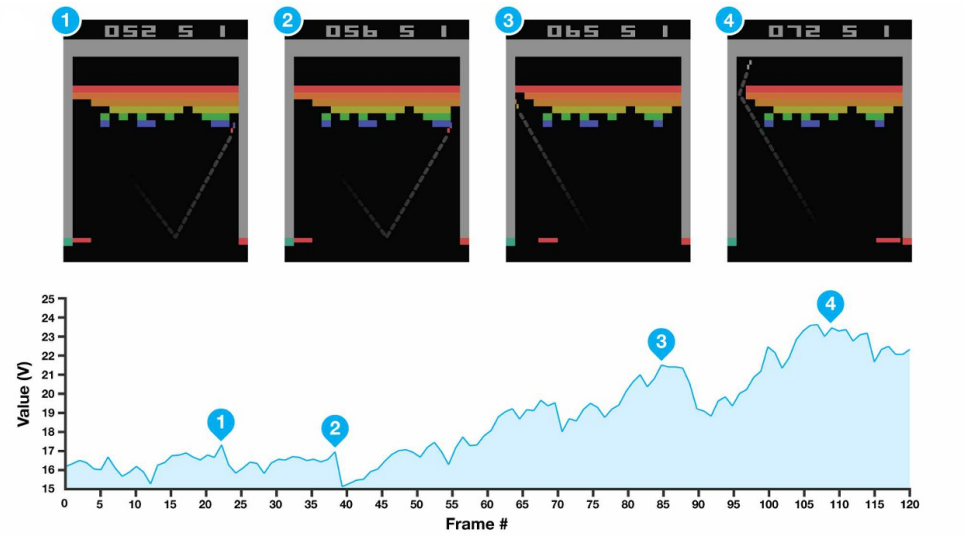


Figure 4.3: From the paper, a visualization of the learned value function on the game Breakout

5 | Conclusion

5.1 | Key Advancements

DQN's primary innovation lies in its integration of deep learning with Q-learning, which allows the approximation of action-value functions for complex, high-dimensional state spaces. To enhance training stability, the authors implemented two key techniques: experience replay and fixed Q-target.
The experience replay mechanism involves storing the agent's experiences - comprising the current state, action taken, reward received, and subsequent state - in a replay memory. During training, random samples from this memory are then used to update the network. This randomization proves to be useful in breaking the temporal correlations between consecutive experiences, leading to more stable and efficient learning. To mitigate instability arising from the continually shifting Q-values during training, DQN employs a separate target network. This network, a copy of the original Q-network, is updated at regular intervals and provides stable target values for the Q-learning updates, reducing oscillations and divergence during training. These are the fixed Q-targets, which the policy network needs to now approximate.

5.2 | Suggested Directions

The paper suggests **prioritized sweeping**, a reinforcement learning technique primarily used for model-based algorithms, where updates are prioritized according to a measure of urgency. It aims to focus computational resources on the most significant updates. It works as follows:

- A priority queue is maintained for all state-action pairs whose values would change substantially if updated.
- When a state-action pair is updated, the effect on its predecessor state-action pairs is computed.
- If the effect exceeds a certain threshold, the predecessor pairs are added to the queue, prioritized by the size of the change (i.e., temporal-difference error).

This technique makes learning more efficient by focusing updates on the most important state-action pairs,specifically those with large **temporal-difference (TD) errors**. This way, the agent doesn't waste effort on transitions that contribute little to learning.
Although the original paper did not implement prioritized sweeping, later research has built on the idea. One notable extension is prioritized experience replay (PER), which applies this concept to model-free methods like DQN. While in **standard experience replay**, all transitions are sampled equally, PER gives higher sampling priority to experiences with larger TD errors - the difference between predicted and actual rewards - which signal greater potential for learning.
For example, imagine an agent learning to play the Atari game "Breakout." The agent records various experiences such as successful ball hits, missed balls, and brick destructions, and some events are more important than others. A missed ball that leads to a lost life typically has a high TD error because of the significant negative reward. PER ensures that such critical experiences are replayed more often, helping the agent learn from its mistakes and adjust its strategy.
Research has shown that PER can improve **learning efficiency and performance**. In their seminal paper, [10] demonstrated that integrating PER into the Deep Q-Network (DQN) framework allowed the agent to outperform the standard DQN with uniform sampling on 41 out of 49 Atari 2600 games. Later, [9] explored some limitations of PER—such as outdated priorities and insufficient coverage of the sample space—and proposed methods to overcome these issues.
These findings underline the importance of experience prioritization in reinforcement learning and highlight its continued evolution to tackle challenges in training intelligent agents.

5.3 | DQN current state

Since the introduction of DQN, several enhancements have been proposed to address its limitations and improve upon its performance and stability:

- **Double DQN**: [12] introduced this approach to mitigate overestimation bias in Q-learning. This is achieved by separating the action selection and evaluation processes, leading to more precise value estimates and better overall performance. By employing different networks for selecting and evaluating actions, Double DQN helps prevent overestimation of action values, making learning more stable.

- **Dueling DQN**: [14] proposed this architecture to distinguish between state value estimation and advantage functions. This separation enables the agent to identify valuable states without having to assess the impact of each action individually, resulting in more efficient learning. By structuring the model this way, it improves generalization across states, resulting in faster learning and better decision-making.
- **Rainbow DQN**: [5] introduced Rainbow DQN, an algorithm that integrates multiple improvements to the standard DQN, including Prioritized Experience Replay (PER), Double DQN, Dueling Networks, Multi-step Learning, Noisy Nets, and Distributional RL. By combining these techniques, Rainbow DQN offers greater stability and efficiency, achieving significantly better performance than the original DQN across various benchmarks.

These advances collectively improve the stability, efficiency, and performance of deep reinforcement learning agents in complex environments, making them more reliable in practical applications.

5.4 | State-of-the-art Reinforcement Learning

Beyond DQN and its variants, the field of reinforcement learning has seen the development of several state-of-the-art algorithms:

- **Actor-Critic Methods**: Algorithms like Proximal Policy Optimization (PPO) and Advantage Actor-Critic (A2C/A3C) combine value-based and policy-based approaches, enabling efficient learning in both discrete and continuous action spaces. PPO, in particular, uses a clipped surrogate objective to improve stability in training ([11]).
- **Model-Based Reinforcement Learning**: Methods such as Model-Based Value Expansion (MBVE) and Model-Based Policy Optimization (MBPO) incorporate learned models of the environment to plan and make decisions, improving sample efficiency. MBPO refines the learning process by using short-horizon model rollouts to generate additional training data ([6]).
- **Distributional RL**: Algorithms like C51 and Quantile Regression DQN (QR-DQN) model the distribution of returns rather than just the expected return, providing a more comprehensive understanding of the value function and leading to improved performance. C51 introduces a categorical distribution representation of Q-values ([1]), while QR-DQN extends this idea by approximating the full return distribution using quantiles ([2]).

These algorithms represent the forefront of reinforcement learning research, each addressing specific challenges and contributing to the advancement of intelligent agents capable of learning from complex environments.

6 | References

[1] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[2] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[3] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay, 2020.

[4] Shane Frederick, George Loewenstein, and Ted O'Donoghue. Time discounting and time preference: A critical review. *Journal of Economic Literature*, 40(2):351–401, 2002.

[5] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[6] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. In *33rd Conference on Neural Information Processing Systems*, 2019.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Kirkeby Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[9] Yangchen Pan, Jincheng Mei, Amir massoud Farahmand, Martha White, Hengshuai Yao, Mohsen Rohani, and Jun Luo. Understanding and mitigating the limitations of prioritized experience replay. *arXiv preprint arXiv:2007.09569*, 2022.

[10] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[11] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[12] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[13] Hongbign Wang, Xin Chen, Qin Wu, Qi Yu, Xingguo Hu, Zibin Zheng, and Athman Bouguettaya. Integrating reinforcement learning with multi-agent techniques for adaptive service composition. *ACM Transactions on Autonomous and Adaptive Systems*, 12:1–42, 05 2017.

[14] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, 2016.