Space Invaders: A journey through the path of Atari's games with Reinforcement Learning

Catia Teixeira, *Student, FEUP*

***Abstract*—This paper explores the intersection of classic video game adaptation, particularly focusing on Space Invaders, within the context of reinforcement learning (RL). This article provides an overview of the game rules, an overview of the related work and techniques for this task as well as a detailed description of the methodology and implementation details of two different Reinforcement Learning (RL) agents to play Space Invaders. The author aims to evaluate the models performance in the task of learning to play the game by RL. The implemented deep learning model are Dueling Deep Recurrent Q-Networks (DDRQN) and Dueling Deep Transformer Network (DDTN) with network improvements such as Prioritized Experience Replay (PER), Multi-Step Learning and Dueling Architecture itself. The work also includes conclusions from initial trials with several parameters variation using a simpler Deep Recurrent Q-Network (DRQN). The results showed that both agents were able to learn over time and achieve an acceptable average score. DDRQN achieved slightly better performance, but overall both agents evolution are very similar. Improvements to the network and memory depth of the agent were crucial in achieving these results.**

***Index Terms*—Space Invaders, Atari Games, Reinforcement Learning, DQL, DQN, DoubleQN, DDQN, DRQN, DDRQN, DTQN, ER, PER, LSTM, CNN, Transformers, Intelligent Systems, Artificial Intelligence, Computer Vision.**

I. Introduction

SPACE INVADERS is a single-player arcade video game that was created by the Japanese engineer and game designer Nishikado Tomohiro and was released in 1978, distributed by Taito Corp. In 1980, the American game manufacturer Atari, Inc., adapted a version of the game for the Atari 2600 console, popularizing the so-called "technological marvel" at the time of its release [1]. Its rules are quite simple: the player controls a spaceship, aiming to defeat waves of descending alien attackers while avoiding their projectiles and taking cover behind destructible barriers.

In Space Invaders, the players control a spaceship positioned at the bottom of the screen that can be moved horizontally. The aliens move side to side and gradually downwards, aiming to reach the bottom. The goal is to eliminate them before they reach there, or before the player gets hit 3 times by the enemies' bullets. The player needs to dodge alien attacks and use barriers for cover. The barriers are destructible, which means they can get destroyed by both aliens' and player's bullets [2].

The adaptation of classic video games like Space Invaders into the realm of Reinforcement Learning (RL) has sparked considerable interest across various research domains. This includes leveraging RL techniques for mastering gaming environments and tackling intricate real-world challenges such as complex control tasks in robotics.

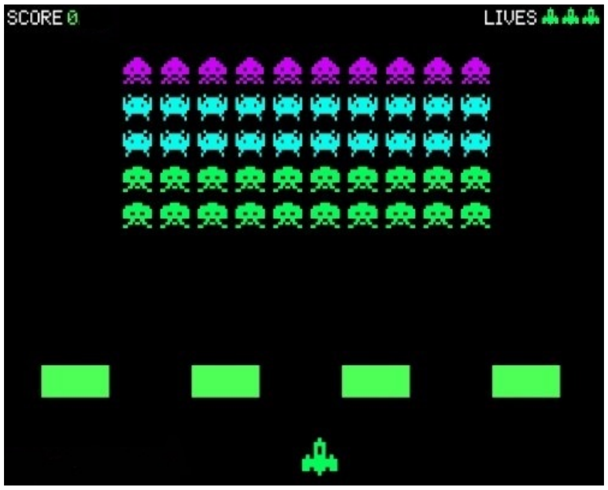

Fig. 1: Space Invaders screen at game start

RL can be formalized by using ideas from dynamical systems' theory, specifically, as the optimal control of incompletely-known Markov decision processes. A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment. Markov decision processes (MDP) are intended to include just these three aspects—sensation, action, and goal—in their simplest possible forms without trivializing any of them. Any method that is well suited to solving such problems can be considered to be a RL method [3]. RL algorithms therefore, typically require interacting with environments and collecting unlabeled state transition data [4].

A. Background

As referred previously, RL agent has the goal of learning the best way to accomplish a task through repeated interactions with its environment. In order to accomplish this, the agent must evaluate the long-term value of the actions that it takes. At time t the Agent takes an action A_t that affects the environment, causing it to transition from state S_t to state S_{t+1}. As a result of the action A_t, and the subsequent transition from state S_t to state S_{t+1}, an immediate reward is generated.

The Agent uses the state information (observation) and immediate reward to generate the next action, continuing the cycle. The goal of the Agent is to learn a mapping from states to actions called a policy $\pi(A_t = a|S_t = S)$ that maximizes a long-term sum of future rewards called a value function $v_\pi(s)$ [5]. If the observation only contains partial state information, the environment is partially observable. Nevertheless, if the observation contains the complete state information of the environment, the environment is fully observable.

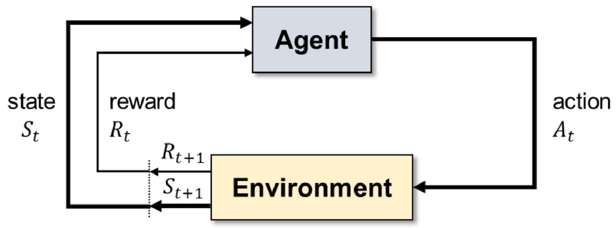

Fig. 2: Basic setting for Reinforcement Learning - source:[5]

In reinforcement learning, a trajectory or episode τ is a sequence of states, actions, and rewards which records how the agent interacts with the environment. The transition from a state to the next state can be either a deterministic transition process or a stochastic transition process. For the stochastic transition process, the next state S_{t+1} is described as a probabilistic distribution

Given a list of immediate reward r for each time step in a single trajectory τ, a return is the cumulative reward of a trajectory [6]

The return function, or simply the return $R(\tau)$, is a long-term measure of rewards. There are three possible expressions for the return. In case of stochastic reinforcement, we have the *the finite-horizon model*, *the discounted return (infinite-horizon model)*, and *the average-reward model*. We only use the discounted return, which is also used in the Dynamic Programming approach. A Markov process is a discrete stochastic process with Markov property, which simplifies the simulation of the world in continuous space.

The two main types of RL methods are actor-critic methods and value-based methods. Both of which rely on estimating the value function: the expected sum of discounted rewards under the current policy. The agent then uses the value function directly to take actions in value-based methods, or uses it to improve a policy in actor-critic methods. The most commonly used method for training value functions is temporal-difference (TD) learning, which is rooted in the Bellman equation. The Bellman equation (Eq.1) provides a recursive formulation for the value function in terms of the value at the next time-step. Specifically, instead of using the sum of discounted rewards sampled from a particular state in the environment, TD learning uses the immediate reward and the discounted value at the next time-step as a training target [7].

$$v_\pi(s) = r(s, \pi(s)) + \gamma \sum_{s'} p(s'|s, \pi(s)) v_\pi(s') \tag{1}$$

Optimal performance is choosing the best action at anytime by acting greedily with respect to the optimal state-value function $v_{\pi*}$:

$$v_{\pi*}(s) = \max_a \left\{ r(s, a) + \gamma \sum_{s'} p(s'|s, a) v_{\pi*}(s') \right\} \tag{2}$$

We can interpret γ in several ways. It can be seen as an interest rate, a probability of living another step, or as a mathematical trick to bound the infinite sum [8]. The value of γ permits to modulate the period the agent takes the reinforcements into account. If γ = 0, the agent is "myopic" and considers the immediate reward only; the more γ is near of 1, the more the agent looks ahead.

Besides the state-value function, another special type of value-function is the action-value function q_π , defined by the expected future reward after taking an action a given a state s and following policy π forever after [9]:

$$q_\pi(s, a) = r + \gamma \sum_{s'} p(s'|s, a) \sum_{a'} \pi(a'|s') q_\pi(s', a') \tag{3}$$

Similarly to Eq. 2, choosing the best action for action-value function $q_{\pi*}$ gives us:

$$q_{\pi*}(s, a) = \max_a \left\{ r(s, a) + \gamma \sum_{s'} p(s'|s, a) q_{\pi*}(s', a') \right\} \tag{4}$$

However, in model-free methods, in order to know the optimal value function, the agent should execute trial actions, i.e. actions that are not optimal with respect to the current value function; this is called exploration [10].

During exploration, the agent is increasing existing knowledge by taking actions and interacting with the environment. During exploitation, the agent is applying the knowledge already gathered so far. Exploration-exploitation trade off is the balance between both cases.

ϵ-greedy is a policy used to balance exploration and exploitation in many reinforcement learning setting. Here, with probability ϵ the agent takes a random action, and with probability 1-ϵ the agent takes the best action known to it [3]. It is from a class of ϵ-soft policies. It is also important to note that exploration is more important when the agent does not have enough information about the environment it is interacting with. Once an agent does have enough information it needs to interact optimally with the environment, allowing it to exploit its knowledge. A common way to obtain this is by multiplying ϵ by a real value less than 1 every episode. It is also known as exponential decay [11]. Another approach could be to decay ϵ linearly over time. It has also been used in reinforcement learning tasks [12].

B. Problem Characterization

In this study, the aim is to train an agent to play the classic arcade game Space Invaders by employing a diverse array of reinforcement learning techniques. The objective was to enable the agent to autonomously learn and adapt its strategies through iterations, leveraging advancements such as Deep Q-Networks, Prioritized Experience Replay, Dueling Architecture, Multi-step Learning.

C. Related work

Atari games have been a widely accepted benchmark for deep reinforcement learning for several years. In the task of surpassing human performance in these games, several advancements have been made in Reinforcement Learning and different approaches and concepts have been re-applied to this specific purpose. The following concepts are worth noting:

1) Deep Q-Learning (DQL): Deep reinforcement learning (deep RL) methods we use deep neural networks to approximate any of the following components of reinforcement learning: value function, $\hat{v}(s;\theta)$ or $\hat{q}(s,a;\theta)$, policy $\pi(a|s;\theta)$:, and model (state transition function and reward function). Here, the parameters θ are the weights in deep neural networks [13].

2) Deep Q-Network (DQN): The term 'Deep' in the 'Deep Q Networks' (DQN) refers to the use of 'Deep' 'Convolutional Neural Networks' (CNN) in the DQNs [14]. This concept was first introduced by Google DeepMind for solving Atari games, also introducing Experience Replay for DQL and the new Target-Q-Network [15].

3) Double Deep Q-Network: When combined with DQN, improves performance by removing overestimation bias [16]. Double Q-learning stores two Q functions: QA and QB. Each Q function is updated with a value from the other Q function for the next state. Since each is updated on the same problem, but with a different set of experience samples, this can be considered an unbiased estimate for the value of this action. It is important that both Q functions learn from separate sets of experiences, but to select an action to perform one can use both value functions [17].

4) Dueling Deep Q-Network (DDQN): Dueling network represents two separate estimators: one for the state value function and one for the state-dependent action advantage function. The two streams are combined via Eq. 5 to produce an estimate of the state-action value function Q. The advantage of the dueling architecture lies partly in its ability to learn the state-value function efficiently [18].

$$\begin{aligned} Q(s,a;&\theta,\alpha,\beta) = \\ &V(s;\theta,\beta) + A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a_0\in A} A(s,a_0;\theta,\alpha) \end{aligned} \tag{5}$$

Where: θ denotes the parameters of the convolutional layers, α and β are the parameters of the two streams of fully-connected layers.

5) Deep Recurrent Q-Network (DRQN): Because an input of DQN is the game frames of the last four steps, DQN had difficulty on mastering such games that need to remember events earlier than four steps in the past. In DRQN, the first fully-connected layer just after convolutional layers is replaced with an LSTM to incorporate past information. By using LSTM, DRQN can choose actions considering all of the past information. [19].

6) Deep Transformer Q-Network (DTQN): Recurrent neural networks in reinforcement learning are often fragile and difficult to train, and sometimes fail completely as a result. DTQN architecture utilizes transformers and self-attention to encode an agent's history. DTQN also utilizes learned positional encodings, empowering the model to learn domain-specific encodings which match the temporal dependencies of the environment [20].

7) Experience Replay (ER): By experience replay, the learning agent remembers its past experiences and repeatedly presents the experiences to its learning algorithm as if the agent experienced again and again what it had experienced before. The result of doing this is that the process of credit/blame propagation is sped up, and therefore the networks usually converge more quickly. An experience is a quadruple, (x, a, y, r), meaning that the execution of an action a in a state x results in a new state y and reinforcement r. A lesson is a temporal sequence of experiences starting from an initial state to a final state, where the goal may or may not be achieved [21].

8) Prioritized Experience Replay (PER): Experience replay lets online reinforcement learning agents remember and reuse experiences from the past. In PER, important transitions are replayed more frequently, and therefore learned more efficiently. Temporal distance (TD) signifies the number of time steps or actions an agent takes to transition from one state to another within an environment.

Prioritized replay relies on assigning importance to transitions based on their TD error magnitude, reflecting their unexpectedness [22].

To define the priority:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{6}$$

To define the importance sample weights:

$$w_i = \left(\frac{1}{N}\cdot\frac{1}{P(i)}\right)^\beta \tag{7}$$

9) Multi-Step Learning: Conventional prediction-learning methods assign credit by means of the difference between predicted and actual outcomes, while Multi-Step methods assign credit by means of the difference between temporally successive predictions. Temporal-difference methods require less memory and less peak computation than conventional methods, and they produce more accurate predictions [23]. In other words, we are calculating the cumulative reward discussed previously over n-steps. The Truncated N-Step Return from a given state is given by Eq. 8:

$$R_\tau^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1} \tag{8}$$

10) Rainbow: Many powerful extensions of DQN have been developed making it capable of achieving high performance and, hence, making it coequal to other RL algorithms. Combining several extensions of DQN can be particularly fruitful when explicitly choosing complementary extensions. Rainbow combines Double DQN, Prioritized experience replay, Dueling Architecture, Multi-step learning, Distributional RL and Noisy Nets. Rainbow agent is far better than any of the extensions on their own, as well as all combinations of extensions that have been examined before [24].

II. Methodology

As this project focuses on a single agent trying to learn from its experiences with the environment, this project is formulated as a RL problem. Although the original version of Atari's Space Invaders is deemed deterministic since it had no feature for generating random numbers, this is not the case on the implemented game source code. In this way, this specific Space Invaders' environment is stochastic, accessible, episodic, dynamic, and discrete.

Hence, it is a reasonable problem to solve by reinforcement learning. In this section, we will discuss the details of the formulation and implementation of the project.

A. Formulation of the problem

The source code for the Space Invaders game was obtained from *Lee Robinson*'s GitHub repository[1]. Alterations were applied to the code to adapt it to the developed AI players.

There are countless variations of Space Invaders, since the game was adapted several times since its creation. In this version of the game, it begins with a total of 50 aliens distributed in a matrix of 5 rows of 10 aliens each. The beginning of the game with the original alien positions, barriers and player spaceship is represented in Fig. 1.

Each alien destroyed corresponds to a different score depending on what row it's positioned. For each green alien corresponding to the two front rows, the player gets 10 points for shooting them. If an alien in the next two rows gets eliminated, corresponding to the blue aliens, the player scores 20 points. For the purple aliens in the last row, the player receives 30 points for each. There is also an additional enemy called the Mystery Ship. It appears from time to time and can help achieve additional points. For each Mystery Ship destroyed, the player receives either 50, 100, 150 or 300 points. It is worth mentioning that after the player reaches 1000 score points during the game, the ship begins to shoot two bullets at once.

B. Implementation details

To implement the Space Invaders RL Agent, Python programming language was used in Visual Studio Code integrated development environment. The code is divided in two main modules:

- **game_v2.py**: main game engine adapted from source code
- **segment_tree.py**: segment tree for prioritized replay buffer from *Curt Park* repository[2].
- **main_1.ipynb**: contains the parameters definitions, model classes, agent class, and train functions for improved model 1.
- **main_2.ipynb**: contains the parameters definitions, model classes, agent class, and train functions for improved model 2.
- **main_3.ipynb**: contains the parameters definitions, agent class, for random agent.

Several Python libraries are used such as *pygame*, *opencv2* and *pytorch*. On an initial stage, experiments were performed using cuda and cuDNN libraries on a NVIDIA GeForce GTX 1050 GPU. At a later stage, the GPU used was NVIDIA GeForce RTX 3080 Ti.

[1]https://github.com/leerob/space-invaders
[2]https://github.com/Curt-Park/rainbow-is-all-you-need/blob/master/segment_tree.py

C. Plan

The development of this work was very exploratory. On a first phase, an initial model was tested and several variations of different parameters were tested. This was extremely important to determine the impact of different parameters on the agent's performance. At a later stage, the knowledge and insights gained from initial experiments were used to determine what was considered a more suitable set of parameters and general direction of the work. These conclusions were then applied in a second phase on two different model architectures. A random model was used for performance comparison. The procedure is summarized in Figure 3.

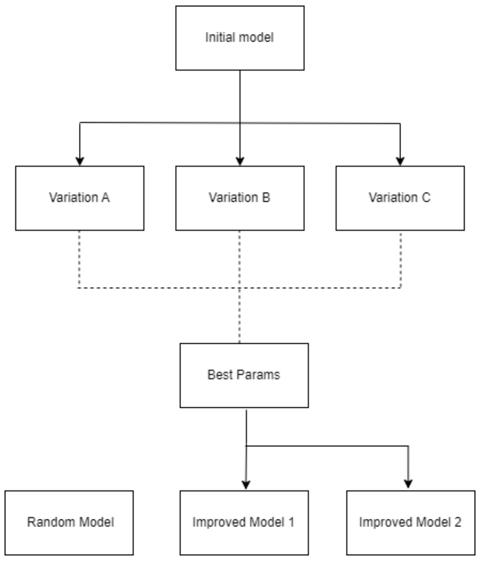

Fig. 3: Summary of experiences plan

The full description of the first phase trials methods and results can be found in Appendix A. For the sake of simplicity, and in order to justify the direction taken in the improved models, the main empirical knowledge gathered in the first phase can be summarized as:

1) **Frame Skip and State Depth**: Skipping frames that are input to the model can be beneficial, since it is more computational efficient, speeds up training, can improve exploration, and somewhat reduces correlation, enabling the agent to capture more diverse experiences. However, it was also observed that an excessive number of skipped frames has a negative impact. Hence, a good trade-off that balances speed training and performance is essential. This negative impact of excessive skipped frames was observed in *Variation A*. Using stacked frames as state representation serve several crucial purposes such as temporal context, motion perception, enhanced feature representation, addressing partial observability, temporal abstractions. The initial model considered stacked frames (see *Variation A*). However, it was concluded that the use of LSTM layer was enough to capture these mentioned purposes and the use of stacked frames was abandoned for the second phase.
2) **Learning Rate and ϵ decay**: Learning rate controls the magnitude of updates to the Q-values and policy

estimates after each iteration. A higher learning rate leads to faster learning, but can also make the learning process unstable or cause oscillations. Conversely, a lower learning rate might make learning slower which can cause the agent not to reach a satisfactory performance within an acceptable number of training rounds. This last fact was observed in *Variation B*. Regarding the exploration and exploitation trade-off it was observed that in the initial trials, the exponential ϵ decay resulted in low values for ϵ almost near the end of the 5,000 games. In comparison to other authors ([11], [24]) it was concluded that in this first approach, the model could be exploiting less than recommended. Because of this, the second phase approach, the direction was to use linear ϵ decay on the initial 20% of the 5,000 games (and to fix ϵ to 0.1 afterward) in order to give enough opportunity for the agent to exploit the acquired knowledge.

3) **Reward Functions**: Reward functions articulate the objectives and goals the agent must achieve within the environment, so they are essential in agent's learning. Too complex reward functions can lead to overfitting to the training environment and hindering generalization to new scenarios. On the other side, too simple rewards might not sufficiently guide the agent towards optimal behavior or adequately capture the complexity of the task. This was tested in *Variation C*, and the main insight was that for this specific problem, simpler reward functions yielded better results.
4) **Memory Depth**: Memory depth, especially in the context of temporal dependencies and sequence learning, holds significant importance in RL tasks. It refers to the capability of RL agents to retain and utilize information from past observations over extended periods. A memory depth that is too small can lead to limited temporal understanding, inability to learn sequential strategies, difficulty in capturing delayed rewards, limited context awareness, reduced generalization and overestimation of values. The effect of memory depth was tested by performing some minor trials that increased the initial depth considered, resulting in much better performance by the agent. The motivation for these trials came from the work in [15]. A particularly interesting behavior seen with limited number of experiences in memory depth was the agent's score over time. With a small memory depth, the agent score increased substantially in the first games played where the ϵ was high. As the ϵ decreased and stored memories were replaced by new ones (sliding window) the agents score decreased abruptly to values even lower then the first scores achieved. This was probably due to important memories from the mainly exploratory phase that lead to higher scores being lost as time passes. This implies memory depth is one of the most important factors, if not the most important, in the ability of the agent to retain relevant information over extended periods and apply them pertinently in later stages of the training. This is determinant in achieving a good performance in a complex task as playing Space Invaders.

D. Final Approach

1) Observation space: The observation space consists of a sequence of encoded features extracted from raw frames using a CNN. Each frame passes through the CNN to produce a set of high-level features that represent the visual information captured in that frame. These feature vectors then form a series of observations that serve as input.

2) Action space: The formulated agent has 4 actions. In this sense, the spaceship being a two-dimensional shooter, the possible actions are as follows:

- Move left
- Move right
- Shoot bullet
- Do nothing

3) Experiences: Experiences are a tuple containing:

- Initial State (Single frame)
- Action
- Reward
- Final State (Single Frame)
- Game Over Flag

The Game Over Flag is important for calculating the updated Q values when the game is over. This affects the γ parameter. When Game Over Flag is set, the value of γ is zero, meaning the agent should act as "myopic".

4) Frame Skipping: As previously mentioned, it is important to find a good trade-off between training speed and performance. According to the first phase insights, and the work developed by the author [25] (which suggests that with a frame skip of 4 some bullets might "disappear"), a value of 3 for frame skipping was chosen for the final approach.

5) Reward functions: As mentioned earlier, the intuition was that the simpler reward functions resulted in the best performance in this specific problem. Therefore, the rewards considered for the final approach in agents learning was simply the game score. No penalties were applied for losing a life or losing the game, and no special reward for winning the game.

6) Memory Depth: According to the insights already described from the first trials, the memory depth considered was 100,000 experiences. This is the maximum capacity possible with the available resources.

E. Implemented algorithms

Two different model architectures were applied to solve this problem, plus a random agent for the sake of comparison. The common parameters used in the two models were the following.

- **Number of episodes**: Each model was trained for 5,000 games (episodes).
- **Learning Rate**: A value of 1×10^{-3} was considered for improved model 1, taking into account the first trial's intuition that lower learning rates result in a training that is more time-consuming for achieving an acceptable performance. Also, this value was applied by the authors of [24] (that apply the model improvements considered in this work). For improved model 2 the Learning Rate was 3×10^{-4} based on the work of [20].

- ϵ **decay**: A linear epsilon decay was considered, given the work in [24]. The ϵ linearly decays from 1.0 to 0.1 during the first 1,000 games, staying constant at 0.1 after that.
- **Optimizer**: The ADAM optimizer was used with SMOOTHL1LOSS (mean absolute error with clipping, to prevent gradient explosion) for improved model 1 and Mean Squared Error (MSE) for improved model 2.
- **Discount factor** (γ): The value considered was 0.99 in the majority of the steps, except for the game over step where a value of 0 is considered. The high discount factor aims to prioritize and emphasize long-term rewards over immediate rewards in an agent's decision-making process.
- **Batch size**: A batch size of 32 was considered for sampling the Agent's memory during replay.
- **Replay frequency and target update**: The replay is done each step as soon as the Agent Memory gathers 1 batch size of experiences (32). The target models are updated each 100 replays.

*1) **Convolutional Neural Network (CNN)**:* As the state are represented by the raw RGB frames, we need to preprocess them in order to extract the relevant features using a CNN.

The initial frame size 600x800 pixels and are rescaled with a factor of 8 resulting in an input image of size 75x100 pixels.

The CNN architecture starts with an initial convolutional layer using 3 input channels (RGB images), followed by a subsequent layer with 64 channels and an (8x8) kernel size.

Sequentially, a second convolutional layer with 128 channels and similar parameters further refines features, succeeded by a third layer utilizing 256 channels and a reduced (4x4) kernel size, enhancing feature abstraction with smaller receptive fields. Subsequently, a final convolutional layer with 512 channels and the same kernel size as the previous layer (4x4) continues the pattern, refining intricate details.

Each convolutional layer is accompanied by rectified linear unit (ReLU) activations, introducing non-linearity and enhancing the network's capability to learn complex representations.

Stride of 2 and no padding are employed in all layers.

After the last convolution, a flatten operation reshapes the output into a one-dimensional tensor, followed by a fully connected layer with 3072 input neurons and 256 output neurons.

*2) **Dueling Deep Recurrent Q-Network (DDRQN)**:* The architecture for improved model 1 was based mainly in the work of the authors [15], [18], [19] and [26].

After the frames are processed in the CNN, the outputs are passed to a LSTM layer with 256 hidden cells and 5 hidden layers. The output is fed into two fully connected layers: the dueling architecture.

The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action. Both networks output Q-values for each action. The two streams are combined via Eq. 5 to produce an estimate of the state-action value function Q.

The logic of having an online and target network were also maintained. In this sense, the online network is a dueling network that predicts the Q values during the emulation and during training we use another set of dueling networks (the target network) to calculate the target Q-values like in the traditional DQN method.

*3) **Dueling Deep Transformer Q-Network (DDTQN)**:* The architecture for the improved model 2 was based mainly in the work of the authors [15], [18] and [20].

After the frames are processed in the CNN, the output is fed into a fully connected layer which is connected to an encoder layer with 2 heads (multiheadattention).

The output of the encoder is fed to a transformer encoder layer, which is a stack of 5 encoded layers. The final output goes to two fully connected layers: the dueling architecture. Each of these layers has an output size of 256 which is followed by ReLU activation and a final output layer. These two outputs predict the advantage and the value.

During training the traditional DQN method of online and target networks were also maintained.

*4) **Network Improvements**:* These improvements applied took into account the work by the authors [24]. In fact, dueling architecture itself can also be considered as an improvement as well to the DQN.

In addition to what was already described, both models were implemented with the following techniques:

a) Gradient Clipping: In order to prevent exploding gradients, gradient clipping was applied. The norm is computed over all gradients together and are modified in-place. Max norm of 10 was used in DDRQN and for DDTQN a max norm of 1 was considered.

b) Prioritized Experience Replay: Based on both [22] and [24]. Eq. 6 and Eq. 7 are used to define the priority and importance, respectively. α is 0.2 and β is 0.6.

c) Multi-Step Learning: Multi-step learning enhances the efficiency of learning by incorporating future rewards into the current action, enabling quicker convergence and improved sample efficiency. Based on both [23] and [24], Eq. 8 is applied for this purpose with an n-step of 3 ($n = 3$).

*5) **Random Agent**:* An Agent that takes random actions for 5,000 games was used in order to serve as comparison to the previous models applied.

III. Results and Discussion

The main results are presented in Table I, where we can find the average and maximum score, average and maximum reward and number of training epochs for both DDRQN and DDTQN agents in comparison with the Random agent.

TABLE I: Results of final models

	Random	DDRQN	DDTQN
Avg. score	95.91	393.53	359.64
Max score	730	1670	1670
Avg. batch reward	N.A.	15.07	12.39
Max batch reward	N.A.	90.81	88.08
Epochs	N.A.	973,420	973,420
Runtime	$\approx$ 240 min	$\approx$ 1440 min	$\approx$ 1270 min

We can see the maximum score was equal between DDRQN agent and DDTQN agent. However, DDRQN presents a slightly higher average score. The maximum batch reward was achieved by DDRQN agent as well as the highest average batch reward. Both models performed the same number of epochs and the training time of DDRQN was slighlty higher than DDTQN.

In the following Fig. 4 we can see the evolution of the score during training for the 3 agents.

The figure presents two stages in the learning: the linear decay ϵ from 0 to 1,000 games and the fixed 0.1 ϵ after 1,000 games.

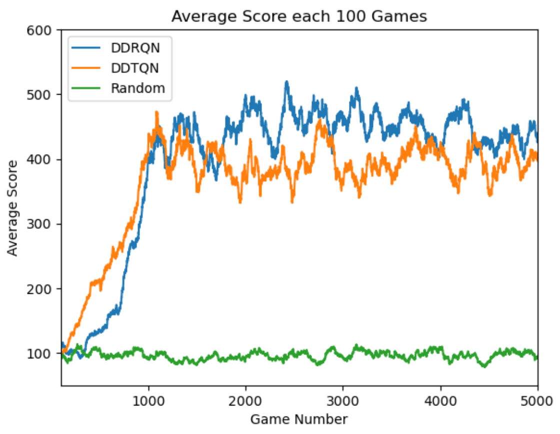

Fig. 4: Average score per 100 games for each agent

Both models appear to present roughly the same performance, both achieving better results than the random agent (500% improvement).

DDTQN agent learns to achieve higher scores much more rapidly and steeply than DDRQN agent, (approximately linear growth). DDQN presents a learning curve that more resembles exponential growth, eventually getting closer to the scores of DDTQN at later moments. These two different behaviors could be due to the different Learning Rates of the models.

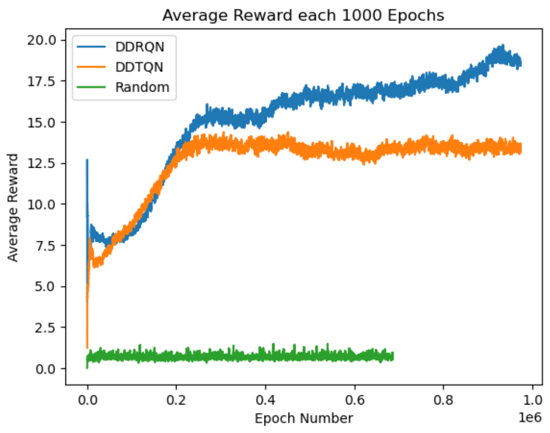

Fig. 5: Average reward per 1,000 epochs for each agent

When the ϵ reaches its minimum, DDTQN agent achieves a superior score than DDRQN agent at that moment. However, DDRQN manages to maintain higher scores until the end of the 5,000 games. After exploration is minimum, the score for both models stays approximately at the same average scores (and even with slight increase). This implies PER helps to maintain the relevant information from the past.

Fig 5 presents the evolution of the rewards throughout the training process.

We can see DDTQN starts with lower rewards than DDRQN but soon achieves higher rewards than DDRQN roughly until 200,000 epochs. Around 300,000 epochs, ϵ stabilizes. However, the rewards of DDRQN continue to increase at a lower rate then in the previous exploratory phase while DDTQN doesn't improve significantly its rewards. Probably DDTQN Learning Rate was a bit too low and could benefit if it was the same as DDQRN.

IV. Conclusion

In this study, two different AI agents were trained using various reinforcement learning techniques to play the arcade game Space Invaders. Several experiments were performed in order to evaluate the impact of different parameters in the

performance of the agents. Two different final architectures for the model were implemented, achieving the main objective of learning throughout training, both achieving far better results than a random agent. Several model improvements were applied to each of the models such as dueling architecture, prioritized experience replay and multi-step learning. These improvements were very beneficial in comparison with the results of the first trials. Memory depth was also a very important aspect that helped improve the overall performance of the final models. The best performance was achieved by DDRQN agent, although DDTQN achieved close results.

V. Threats to Validity and Future Work

The decision to use a custom game environment instead of OpenAI Gym ALE gave more flexibility to customize parameters such as reward functions and draw insights from these experiments on one side. On the other side, it also made comparison with other published work more difficult, since most of the papers available use this OpenAI environment for their experiments.

For future work, it would be interesting to train the DDTQN with the same Learning Rate as the DDRQN. Due to time constrains, the decision was made to use the same Learning Rate as the authors of [20].

The use of an even higher value for state depth would propose a good comparison in order to better understand the effect of this condition in the agent's performance.

At the same time, repeating the experiments with ϵ decaying for a longer period of time could be interesting to see if both agents achieve higher scores.

Also, it would be interesting to experiment with Visual Transformers and compare performance with models using CNN to extract features from frames.

VI. Acknowledgements

The author would like to thank *Lee Robinson* and for sharing their codes for the Space Invaders game on GitHub as well as to all other authors referenced that have also published their work and shared their code online.

Appendix A
First phase trials

A. Methods

The first trials used the same CNN as described previously. The initial model used had a standard DRQN (256 hidden cells, 5 hidden layers) with no improvements. The initial memory depth was 500 episodes. ϵ decay method was exponential decay, fixing the minimum ϵ to 0.15. The remaining parameters are described in Table II.

TABLE II: Description of Baseline Parameters

Item	Value
state depth	10
frame skip	4
learning rate	1×10^{-5}
ϵ decay	0.99970
rewards	V0

V0 Rewards considered the following conditions:

- For each frame alive: + 0.1
- For each enemy killed: + 10 $\times$ alien points
- Win game: + 1000
- Each move down by aliens: - 0.1 $\times$ alive aliens
- Each lost life: - 300
- Each game over: - 1000

1) Variation A (State depth and frame skip): Baseline conditions were compared with a variation of these two parameters: State depth of 7 and frame skip of 8.

2) Variation B (Learning Rate and ϵ decay): Baseline conditions were compared with a variation of these two parameters: Learning rate of 1×10^{-6} and ϵ decay of 0.99987.

3) Variation C (Rewards): Baseline conditions were compared with two different conditions for rewards: V1 uses a simpler reward system without the penalty for the move down by aliens. V2 uses a more complex reward system when compared to V0, with an additional reward of + 0.1 for dodging a bullet.

B. Results

The results for the first trial are summarizes in Table III. Note reward results are not comparable with the final approach since the reward systems changed greatly.

TABLE III: Results of first trials

	Baseline	Var. A	Var. B	Var. C-V1	Var. C-V2
Avg. score	225.06	171.80	159.86	237.92	236.03
Max score	1410	1230	1230	1630	1580
Avg. reward	-36.36	-23.51	-39.02	-35.63	-34.08
Max reward	165.79	8.86	14.03	59.16	76.89
Epochs	12,898	6,125	11,755	13,145	12,590

In fig 6 we can visualize the average score between Baseline and the several variations.

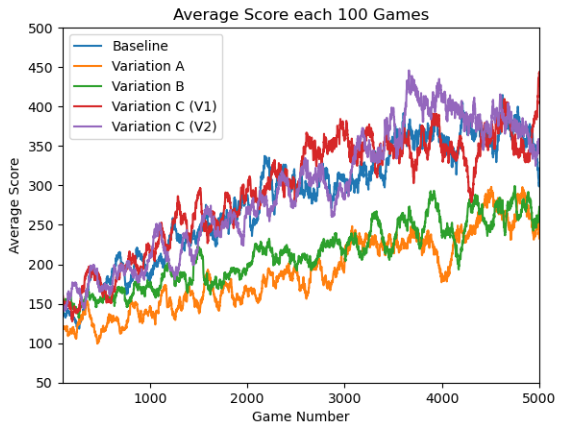

Fig. 6: Average score per 100 games for each agent

References

[1] E. of Encyclopedia Britannica, "Space invaders: elctronic game," 2023. https://www.britannica.com/topic/Space-Invaders [Accessed: 09.12.2023].

[2] AtariAge, "Space invaders manual - atari - atari 2600," 2023. https://atariage.com/manual_thumbs.php?SystemID=2600SoftwareID=1306itemTypeID=MANUAL [Accessed: 09.12.2023].

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[4] Y. Lan, X. Xu, Q. Fang, and J. Hao, "Sample efficient deep reinforcement learning with online state abstraction and causal transformer model prediction," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2023.

[5] J. Shin, T. A. Badgwell, K.-H. Liu, and J. H. Lee, "Reinforcement learning – overview of recent progress and implications for process control," *Computers Chemical Engineering*, vol. 127, pp. 282–294, 2019.

[6] Z. Ding, Y. Huang, H. Yuan, and H. Dong, "Introduction to reinforcement learning," *Deep reinforcement learning: fundamentals, research and applications*, pp. 47–123, 2020.

[7] J. Romoff, *Decomposing the Bellman Equation in Reinforcement Learning*. McGill University (Canada), 2021.

[8] A. W. M. L. P. Kaelbling, M. L. Littman, "Deep reinforcement learning: An overview," *Journal of Artificial Intelligence*, vol. 4, 1996.

[9] F. S. Neves, G. A. Andrade, M. F. Reis, A. P. Aguiar, and A. M. Pinto, "Decoding reinforcement learning for newcomers," *IEEE Access*, vol. 11, pp. 52778–52789, 2023.

[10] S. Ishii, W. Yoshida, and J. Yoshimoto, "Control of exploitation–exploration meta-parameter in reinforcement learning," *Neural Networks*, vol. 15, no. 4, pp. 665–687, 2002.

[11] A. Maroti, "Rbed: Reward based epsilon decay," *arXiv preprint arXiv:1910.13701*, 2019.

[12] G. Kumar, G. Foster, C. Cherry, and M. Krikun, "Reinforcement learning based curriculum optimization for neural machine translation," *arXiv preprint arXiv:1903.00041*, 2019.

[13] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[14] M. Sewak, *Deep Q Network (DQN), Double DQN, and Dueling DQN: A Step Towards General Artificial Intelligence*, pp. 95–108. 06 2019.

[15] D. S. e. a. V. Mnih, K. Kavukcuoglu, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, p. 529–533, 02 2015.

[16] A. M. Hafiz, "A survey of deep q-networks used for reinforcement learning: State of the art," in *Intelligent Communication Technologies and Virtual Mobile Networks* (G. Rajakumar, K.-L. Du, C. Vuppalapati, and G. N. Beligiannis, eds.), (Singapore), pp. 393–402, Springer Nature Singapore, 2023.

[17] H. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems* (J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, eds.), vol. 23, Curran Associates, Inc., 2010.

[18] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning*

(M. F. Balcan and K. Q. Weinberger, eds.), vol. 48 of *Proceedings of Machine Learning Research*, (New York, New York, USA), pp. 1995–2003, PMLR, 20–22 Jun 2016.

[19] H. Oh and T. Kaneko, "Deep recurrent q-network with truncated history," in *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pp. 34–39, 2018.

[20] R. P. K. Esslinger and C. Amato, "Deep transformer q-networks for partially observable reinforcement learning," *arXiv*, pp. 1–15, 2022.

[21] L. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, vol. 8, p. 293–321, 1992.

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

[23] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 8, p. 9–44, 1988.

[24] J. Jäger, F. Helfenstein, and F. Scharf, *Bring Color to Deep Q-Networks: Limitations and Improvements of DQN Leading to Rainbow DQN*, pp. 135–149. Cham: Springer International Publishing, 2021.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[26] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," in *2015 aaai fall symposium series*, 2015.