

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR



**Master in Deep Learning for
Audio and Video Signal Processing**

MASTER THESIS

**DECISION-MAKING EXPLAINABILITY IN
ATTENTION-BASED MODELS FOR
REINFORCEMENT LEARNING**

**Javier Muñoz Haro
Advisor: Luis Lago Fernández**

Junio 2024

DECISION-MAKING EXPLAINABILITY IN ATTENTION-BASED MODELS FOR REINFORCEMENT LEARNING

Javier Muñoz Haro
Advisor: Luis Lago Fernández

Dpto. Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio 2024

Resumen

La explicabilidad en el campo de la Inteligencia Artificial (IA) es una de las tareas más importantes en la actualidad, especialmente en el ámbito del Deep Learning. Las redes neuronales, conocidas por su gran capacidad de aprendizaje, son el estado del arte en casi todas las subdisciplinas de la IA gracias a su habilidad de representación de características en espacios de alta dimensionalidad. Pese a esto, este avance tiene un gran problema: la explicabilidad. Recientemente, con la introducción del mecanismo de self-attention (auto-atención) se ha arrojado un haz de luz, ya que trabajos en el campo de la visión artificial han demostrado su utilidad a la hora de entender a qué da más importancia una red cuando esta genera una salida.

Este trabajo intenta abordar este problema en el paradigma del aprendizaje por refuerzo (reinforcement learning). Uno de los avances más importantes en la última década fue la introducción de redes neuronales profundas para estimar el valor de los estados y acciones, pero a costa de la explicabilidad. Dado que queremos entender cuáles son elementos de un entorno que influyen en las decisiones de un agente, necesitamos una función característica que nos brinde información sobre cómo la red está procesa los datos de entrada. Aprovechando el mecanismo de self-attention, nuestro objetivo es comprobar si existe explicabilidad para el comportamiento de un agente en varios entornos y ver si hay una evidencia general y aplicable a múltiples configuraciones donde las decisiones del modelo puedan explicarse mediante el mecanismo de atención. La explicabilidad es una de las piedras angulares de la IA aplicada, ya que el creciente interés en diferentes sectores en la industria como las finanzas o la automoción puede verse frenado por la falta de explicabilidad en los modelos utilizados.

En este informe, reflejamos las diferentes pruebas y configuraciones de aprendizaje por refuerzo basadas en Deep Q-networks para modelos basados en atención. Hemos entrenado agentes para jugar dos juegos clásicos del entorno Atari 2600. También proponemos varias planificaciones de exploración para el modelo, comparándolos y analizando el por qué de que algunos funcionan mejor que otros. Para explicar el comportamiento del modelo, hemos extraído evidencias visuales para comprender e interpretar las decisiones del modelo, con algunos hallazgos interesantes sobre los mapas de atención y los mapas de activación utilizando algoritmos bien establecidos que se adaptan a este propósito. Argumentamos que hay espacio para entender las decisiones del agente cuando realiza acciones en un entorno, y creemos que la aplicación de este tipo de técnicas a problemas más complejos podría ofrecer una mejor perspectiva sobre por qué los agentes se comportan como lo hacen. Además, hemos desarrollado un pipeline que ha facilitado muchas configuraciones para realizar nuestros experimentos, adaptándose a la naturaleza de prueba y error del paradigma de Machine Learning.

Palabras clave

Redes neuronales, Aprendizaje por refuerzo, explicabilidad, atención.

Abstract

Explainability in the field of Artificial Intelligence (AI) is one of the most important tasks nowadays, particularly in the realm of Deep Learning. Neural networks, known for their advanced learning capabilities, are the state-of-the-art systems in almost every sub-discipline of AI due to their capabilities of processing high-dimensional data. However, this advancement comes at a cost: explainability. Recently, the introduction of the self-attention mechanism has changed this, as recent works have shown that it can provide a wider context for understanding what the network is 'looking' at when generating an output.

Our work tries to tackle this problem in the reinforcement learning problem. One of the most substantial advancements in the last decade was the introduction of deep neural networks to estimate the value of states and actions. Since we want to understand the elements of an environment that influence an agent's decisions, we need a characteristic function that may give us information on how the network is processing the input data. Taking advantage of the self-attention mechanism, our aim is to find explanations for an agent's behavior in several environments and see if there is general evidence across multiple setups where the model's decisions can be explained by the attention mechanism. Explainability is one of the most important fields of applied AI, as the growing interest in different sectors such as industrial, finance, or automotive may be halted by the lack of explainability.

For this report, we have tried several reinforcement learning set-ups based in Deep Q-networks. We have trained agents based on attention models to play two classic Atari 2600 games. We also propose several exploration schedules for the model, comparing them and analysing why some may work better than others. For explaining the model's behaviour, we have extracted visual evidences to explain and interpret the model's decisions, with some interesting findings about the attention maps and the activation maps using well-established algorithms that suit this purpose. We argue that there is room for understanding the agent decisions when it perform actions in an environment, and we believe that the applications of this type of techniques to more complex problems could give a better view on why agents behave as they do. Additionally, we have developed a pipeline that facilitated lots of experimental set-ups, suiting the trial-error nature of the Machine Learning paradigm.

Keywords

Neural networks, reinforcement learning, explainability, attention.

Acknowledgements

I would like to thank my master thesis advisor Luis Lago for his knowledge, his advice and our talks on AI. To my family and friends, for believing in me and give me the courage to always push further. And to the Universidad de Alcalá, especially to the GEINTRA research group for the facilities they have provided me to develop this work.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Objectives	2
1.3	Report structure	3
2	Related work	5
2.1	An introduction to the RL problem	5
2.1.1	Formalizing RL	5
2.1.2	Value-based RL	7
2.1.3	Policy-based RL	9
2.2	Attention-based models	10
2.2.1	What is attention?	10
2.2.2	The Transformer Architecture	11
2.2.3	Vision Transformers	13
2.2.4	SWIN Transformer	14
2.3	Explainable AI	16
2.3.1	Sensitive Analysis	16
2.3.2	Layer-wise Relevance Propagation	17
2.3.3	Class Activation Maps	17
2.4	Reinforcement Learning using Transformers	18
2.5	Explainable RL using Attention	19
3	Design and development	21
3.1	Farama’s gymnasium library	21
3.1.1	Environment dynamics	22
3.1.2	Processing techniques of an environment	23
3.2	Agents development	26
3.2.1	DQN and DDQN training loop	26
3.3	Q-networks integration	32
3.4	Setting up the pipeline	34
3.4.1	Trainer	34
3.4.2	Logger	35
3.4.3	Schedulers	38
3.4.4	Extracting explainability features	40
3.4.5	Additional implementations	43
3.4.6	Overview	44

4 Evaluation	45
4.1 Experimental set-up	45
4.1.1 Computing resources	45
4.1.2 Training set-up	46
4.1.3 Q-networks configuration	47
4.1.4 Environments	48
4.2 Training results	49
4.2.1 Exploration schedules	49
4.2.2 Agents training comparison	50
4.2.3 Final evaluation results	55
4.3 Explainability features	57
4.3.1 Attention maps results	57
4.3.2 Activation maps results	60
5 Conclusions and future work	65
5.1 Conclusions	65
5.2 Future work	65
Bibliography	67
Appendix	73
A Reinforcement Learning	73
A.1 Classic learning methods in RL	73
A.1.1 Monte Carlo estimation:	73
A.1.2 Temporal Difference Learning:	73
A.1.3 On-Policy Control:	75
A.1.4 Off-Policy Control:	76
A.1.5 Actor-Critic Methods	76
B Attention Mechanism	77
B.1 Understanding the attention mechanism	77
B.2 From attention to self-attention	78
C Attention-models code analysis	81
C.1 Attention-based models: Vision Transformer	81
C.1.1 Constructor	81
C.1.2 Patch Embedding	82
C.1.3 ViT encoder block	83
C.1.4 ViT Attention Module	84
C.1.5 The forward method	84
C.2 Attention-based models: SWIN Transformer	85
C.2.1 Constructor	85
C.2.2 BasicLayer	87
C.2.3 SwinTransformerBlock	88
C.2.4 Window Attention	90
C.2.5 The forward method	91

D Schedulers	93
D.1 Gamma for the exponential equation	93
D.2 Gamma for the product of exponential equation	93

List of Figures

2.1	(From [1]) Markov Decision Process with 5 states as the circles, actions are modeled with the red letters over the arrows, that represent the transition between states. When a transition is performed, a reward is returned (letter R), and a new state is achieved. We can see that there might be transitions where we cannot take an action, and we are subject to the transition probabilities (the dynamics) of the MDP.	6
2.2	Aligning inputs using the attention mechanism. First, the inputs $X_1, X_2, X_3, \dots, X_\tau$ are encoded using a bi-directional RNN, obtaining the embeddings \vec{h}, \hat{h} . This embeddings are then pondered by the product of the learnable parameters α_{it} for each input, creating the context vector c_t . Finally, c_t is fed into the decoder, updating the hidden state s_t and producing the output y_t	11
2.3	Multi-head attention. The embeddings from Q, K and V are computed and fed into the attention module. Once computed, the attention matrix is computed for each of the heads, and concatenated to then, be forwarded onto a linear layer, that computes the embedding of the attention-weighted input.	12
2.4	In figure 2.4a (left), from [2], the complete transformer architecture is presented. In figure 2.4b (right) from [3], an attention matrix is presented for a sentence and its translated counter-part. The weights for each word quantify how much <i>attention</i> is the model paying to the words from the original sentence in order to produce the translation to the new sentence.	13
2.5	In figure 2.4a (left) the ViT transformer architecture is presented. In figure 2.5b (right) an attention map is presented. The model that produced the attention maps was trained to track people on a scene, and it is why, the higher attention weights (red) are put in the pixels where people is present.	14
2.6	Swin Transformer architecture and its building blocks. The transformer block, figure 2.6a, has two phases. In the first one (left-most block), it performs a layer normalization, followed by a window-MSHA block that is forwarded onto a MLP with a neurons ratio of 4. In the second, it makes use of the shifted window MSHA, instead of the window-MHSA. In figure 2.6b we can see the whole SWIN Transformer architecture, where each of the "stages" are built upon the transformer blocks, that have as input the patch merged embeddings that are passed at each stage.	15
2.7	Window shifting in the SWIN Transformer architecture (figure 2.7a). From the window partition, shift to the bottom right, so the A, B, C patches are rearranged in the mirrored side of the image. Patch Merging (figure 2.7b), picks blocks of 2 by 2 patches inside a window and concatenates them along the channel dimension.	15
2.8	Explainability of ML models against the prediction accuracy.	16

3.1	In figure 3.1a (left) the goal is to land securely the spaceship (in purple). In figure 3.1b (right) the goal is for the ant to balance itself and start walking.	22
3.2	Shape of the proposed exploration functions for the DQN agent	40
3.3	Example of an attention matrix for one attention head with 4×4 patches. The section under the red rectangle refers to the patches that the class token ponders in order to perform the output. This patches will be our extraction for the attention maps.	41
3.4	Left side, the original frame from the game, in the center, the attention maps for the class token, at the right-side, the Q-values for the observation from the state.	42
3.5	Left-side, the current frame of the game. In the center, the activation maps from the ViT using grad-CAM. At the right-side, the Q-values provided by the neural network.	43
3.6	Implementation overview of all the modules involved in this work	44
4.1	In figure 4.1a (left) we see the Pacman environment. In figure 4.1b (right) we see the Demon Attack environment.	49
4.2	Comparison between different schedules for a DDQN CNN agent for 1.2×10^7 steps.	50
4.3	Comparison between different schedules for a DDQN CNN agent for 2.5×10^7 steps.	51
4.4	Reward vs. Time-steps for the CNN, SWIN and ViT agents.	52
4.5	Reward vs. Time in seconds for the agents in MsPacman environment.	52
4.6	Demon Attack environment results in time-steps.	53
4.7	DemonAttack environment results in seconds.	54
4.8	In game frame where we provide (from left to right) the averaged attention map, the original frame and the Q-values as the output of the network. Additionally we provide the attention heads from the ViT Q-network.	57
4.9	Three snapshots from a situation where the agent encounters a ghost. It acknowledges the ghost and searches for routes to avoid them.	58
4.10	Three snapshots from a situation where the agent tries to find rewards in the environment.	59
4.11	Attention maps for the DemonAttack environment.	60
4.12	Several Grad-CAM activations over different phases of the MsPacman game for the ViT agent.	61
4.13	Several Grad-CAM activations over different phases of the MsPacman game for the SWIN transformer agent.	62
4.14	Grad-CAM activations for the MsPacman game for the SWIN transformer agent, where it is clear that it is not understandable what is the agent looking for.	62
4.15	63
4.16	Activation maps for the SWIN agent in the Demon Attack environment. Some improvements are obtained in terms of interpretability, but nothing conclusive.	64
A.1	TD Returns	74

List of Tables

4.1	Fixed hyper-parameters that are shared across different training procedures.	46
4.2	CNN DDQN network hyper-parameters	47
4.3	ViT Small DDQN network hyper-parameters	47
4.4	ViT Big DDQN network hyper-parameters	48
4.5	SWIN DDQN network hyper-parameters	48
4.6	SWIN DDQN network hyper-parameters	48
4.7	MsPacman environment results for the several trained agents	53
4.8	MsPacman environment results for the several trained agents	55
4.9	Mean Scores for the trained agents	55
4.10	Max Scores for the trained agents	56
4.11	Min Scores for the trained agents	56
4.12	Comparison with SWIN DQN [4] for mean scores for the trained agents . .	56
4.13	Comparison with SWIN DQN [4] for max scores for the trained agents . .	56

Chapter 1

Introduction

1.1 Motivation

Interacting with an environment is one of the most intuitive ways to learn, especially for us humans. Since the beginning of time, species have competed against each other and their environment, with a simple rule, those who adapt survive. A very simple way to describe these dynamics is to say that us humans (agents) perform actions in the world (environment), and the world "rewards" those actions. A simple example of this could be the use of fire. If we use it to heat food, it will kill bacteria and other micro-organisms, which may potentially harm us, but if we put our hands in it, it may create bruises in our skin that could result in an infection and a potential death. Generation after generation, our cognitive abilities as a product of evolution have made possible to pass knowledge about how our environment works. By compiling data and experiences about what gives us the most positive rewards (using fire as a beneficial tool) or the most negative (burning ourselves with fire), we have created a knowledge basis that makes us interact with the world in a beneficial way. This set of dynamics and rules can be formalized into a framework called reinforcement learning. Reinforcement learning (RL) is a field of Artificial Intelligence (AI) that describes an agent that learns to behave, selecting actions that follow some policy in such a way that maximize the reward.

Evolution towards more complex and sophisticated systems for adaptation to an ever-changing environment seems like a pattern in history. At first, very "basic" but understandable kinds of species populated the earth, such as micro-organisms like bacteria or virus, that had "basic" but useful capabilities for extreme conditions resistance such as DNA repair mechanisms or survival strategies such as spores or cyst formations that, although astonishing, were "simple" molecular reactions that made them survive in the most of hostile environments. As generations went on, more complex organisms with lots of diverse functionalities such as eyesight or eco-localization appeared. Similarly, one of the most important breakouts in this field of RL was incorporating complex systems to the agent such as deep neural networks to process the observations of an environment.

At first, problems of RL were solved in "simple" tabular ways using dynamic programming, but as the number of possible states from an environment grew up, these methods fell short to address the problem, since they required of lots of memory and computational power to solve them. Sutton *et al.* coined this problem as the "curse of dimensionality" [5]. To tackle this issue, they propose to introduce function approximators (linear and polynomial), but with the growing interest of neural networks in machine learning, researchers opted to use them instead. Neural networks are of especial use for complex environments

where the number of states its unfeasible to represent using only tabular methods. They are used as systems that command the agent behaviour, like some sort of brain that perceives, evaluates and executes. Although neural networks models give a lot of representation power to the agent, it comes with a cost. Most of the times, they are treated as black boxes, given their complexity, and this causes a problem in interpreting the decisions that an agent may perform. Several improvements have been of use to understand what the network is looking at in order to perform an action, such as using convolutional neural networks (CNNs), but their lack of interpretability, as the network goes deeper, ends up in a set of descriptions that are not understandable for the human eye. Several works [6, 7, 8, 9] have tried to interpret the activations of the receptive fields from the CNN model, but lots of work is yet to be done in this department, since most of the results are biased towards the supervised set-up.

The attention mechanism was introduced for the realm of natural language processing (NLP) as a function that characterizes what the network thinks is more important from the input data, and the relations between the elements of the input sequence. The original target for this type of models was to translate an input sentence or perform next-token-prediction. With this functionality, researchers could now see which parts of the input had higher importance (or weight), giving a richer context on how the model "reasons" about the data that is fed. In the supervised learning domain this is quite useful. For example, in a classification task, such as image classification, the network reveals which are the most descriptive features of a class that make the prediction, or in a regression task, such as energy demand prediction or stock market price estimation, where the attention mechanism tells which parts of the historic data were more relevant to come up with the predicted value.

With this intuition, we propose to use the attention mechanism as a tool for explainability in the decision making of an agent. Our main goal is to use an agent that makes use of attention (along other interpretability and explainability techniques) to evaluate the decision making given the observation of an environment. For example, in an autonomous driving set-up, we could imagine a car going towards a cross-walk when a pedestrian comes across the street. As humans, we focus our attention on the pedestrian, and press the brakes of the car for it to stop. We expect something similar from the agent, as to give us some intuitive visual cues of what it considers important, given an observation of an environment to perform an action.

All the developed code is available in <https://github.com/Javimh18/TLF/tree/main>. Some evidences come from some videos that we have extracted from the agents playing games, those videos are available in <https://drive.google.com/drive/folders/1cABvE9vmXMyWWHmAqBzeffJ0cFBc7GZD?usp=sharing>.

1.2 Objectives

1. First, we will provide a comprehensive review of the basic time difference methods, such as Monte Carlo, SARSA or Q-learning. This is done to show our understanding on the matter, since it falls a little out of scope of the master's program.
2. We will implement the different DQN learning algorithms.
3. For each model we will implement their own attention mechanism, giving a extensive review of the approaches proposed for the implemented models.
4. We will test our implementations in classical environments based on the Arcade Learning Environment (ALE)

5. Implement a pipeline to enable automatic training, evaluation and metrics extraction.
6. Evaluate the results using attention and other explainability techniques in the RL paradigm.

1.3 Report structure

In [chapter 1](#) we give a brief introduction to the main goals of this work. We briefly explain the problem of explainability in reinforcement learning, and how we think attention can help to solve it. Following the first chapter, in [chapter 2](#) we go over the starting point of this work, contemplating several approaches to the reinforcement learning problem and how we can tackle it using attention, addressing previous work as our starting point. A review is done on the main attention models for visual data and how they are incorporated onto the reinforcement learning paradigm. Once we have explained the basic practical and theoretic components, in [chapter 3](#) we go deep into how we have implemented the set-up where we are going to evaluate the explainability of attention-based models. Since the online reinforcement learning set-up does not use a dataset, but rather an interactive environment, we will explain how we connected a model oriented to supervised learning into the reinforcement learning framework. With our design and development explained, [chapter 4](#) provides the procedure involved in obtaining our results and the insights that we have extracted from the attention mechanism as an explainability tool for decision-making. Finally, in [chapter 5](#) we will sum up all the information and give our main thoughts of this work. We will go over the results and give our view on them, and discuss if they are relevant. Also, given the results, we will explore new lines of work that may be interesting to pursue in order to extract better knowledge from the attention mechanism as an intuitive tool for explainability in RL.

Chapter 2

Related work

2.1 An introduction to the RL problem

2.1.1 Formalizing RL

As Sutton and Barto define in [5], reinforcement learning is basically learning what to do, mapping situations to actions in order to maximize a numerical reward signal. Usually, the learner is not told what to do, but instead must discover and perform the actions that get the most rewards. The RL problem usually has two main components:

- **Environment:** A simulated or real world that takes actions as input and produces rewards subsequently as an output along with a new observation for the agent.
- **Agent** (the learner): whose goal is to maximize the reward from the environment

The environment can be formally modeled as a Markov Decision Process (MDP) which is represented in Figure 2.1. An MDP basically is a memory-less random process that is modeled by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where \mathcal{S} is a finite set of states where the agent can land, \mathcal{A} is a finite set of actions the agent can perform, \mathcal{P} is the matrix that models the transition probabilities of changing from the actual state s to the subsequent state s' given an action a . In the case of MDPs, \mathcal{R} is the reward function that models, for a given state s and action a , the numerical reward signal that the environment will produce. It is important to not confuse the set observations from the states \mathcal{O} with the set of states \mathcal{S} . For a MDP, the agent has total awareness of the states, and can obtain every bit of information from it. When this does not happen (i.e. a robot that has visual sensors that point only forward, and not in all possible directions), we say that the agent is in a partially observable Markov decision process (POMDP) and here $\mathcal{O} \neq \mathcal{S}$, meaning that an observation is not the same as the state. At the moment, we are dealing with MDPs, so we assume that $\mathcal{O} = \mathcal{S}$.

As we can see, the process always follows the same pattern. If we define \mathcal{H}_t as the history over the MDP, we could formalize a trajectory as in equation 2.1.

$$\mathcal{H}_t = s_1, a_1, r_2, s_2, a_2, r_3 \dots s_{t-1}, a_{t-1}, r_t, s_t \quad (2.1)$$

But we actually do not need to store in memory whole trajectories of the agent for this problem. A markovian state uses the maxima: "*the future is independent of the past, given the present*", meaning that the current state s_t is a sufficient statistic of what happened in the past and it conditions what may happen next. This is formally defined in equation 2.2, where \mathbb{P} represents the probability and $|$ represents the condition of a probability distribution.

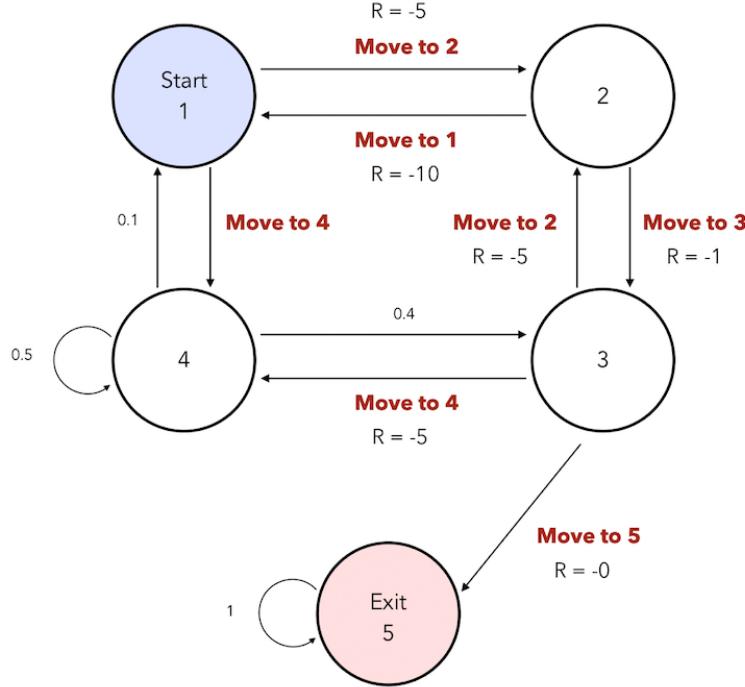


Figure 2.1: (From [1]) Markov Decision Process with 5 states as the circles, actions are modeled with the red letters over the arrows, that represent the transition between states. When a transition is performed, a reward is returned (letter R), and a new state is achieved. We can see that there might be transitions where we cannot take an action, and we are subject to the transition probabilities (the dynamics) of the MDP.

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \dots, S_t] \quad (2.2)$$

The agent has three major components:

- **Policy:** How the agent models the picked actions given a state. It may be deterministic or stochastic, and it is usually referred as π in the notation. For example, if the policy is deterministic then we can define a mapping, such as $\pi(s) = a$, but if the policy is stochastic, then it can be defined as a probability distribution, such as $\pi(a|s) = \mathbb{P}[A = a|S = s]$.
- **Value Function:** A function that models how good a state is. It models a prediction of the expected future reward. Our actions usually will be influenced by how good an outcome will be by selecting an action a , given a state s . The value function weights between the immediate rewards and the future rewards, giving proper estimations to the current states taking into account the potential states the agent may end up. In future sections we will formalize this concept, giving a mathematical and analytical explanation.
- **Model:** Useful information that the agent picks in order to build a model of the environment to understand what it might do next. This modelling consists in estimating two already known elements: the transition probability matrix \mathcal{P} and the reward function \mathcal{R} from the environment. Estimations are modelled by:

1. Transition probability estimation: $\mathcal{P}_{ss'}^a = \mathbb{P}[S' = s' | S = s, A = a]$
2. Reward function estimation: $\mathcal{R}_s^a = \mathbb{E}[\mathcal{R}|S = s, A = a]$

Usually, there are two types of environment modelling: model-based and model-free. In model-free RL, the agent ignores the environment's model, since it does not need to understand how the dynamics work. Instead, it uses observations and reward to build a value function and a policy that maximizes the reward. Model-based approaches try to make the estimations of the environment dynamics, and use them to plan and execute policies that maximize the reward. For this work, we will mainly focus on the model-free approach.

Finally, agent categorization is an important part in the RL framework. There are two principal categories (although with recent advancements, new categories have appeared in the RL scene): value-based agents and policy-based agents. We will delve into this in the following sections, but we will give a brief introduction here. Value-based methods estimate the value of states in the environment and create a function that is used by the policy in order to perform actions. If only states are taken into account to provide this value estimation given a policy π , then the notation used will be $V_\pi(s)$. If the information to provide a value depends on both states and actions, then, we will refer to them either by $Q_\pi(s, a)$ or $q_\pi(s, a)$ depending if we are using the Q-values from Q-learning or other time difference learning methods. On the other hand, policy-based agents only make use of policy optimization to maximize rewards, without using the value function. These two approaches can be mixed up in a symbiotic approach, using the policy and the value function to maximize the returns. This approach is usually called actor-critic methods, where the actor is the policy and the critic is the value function.

2.1.2 Value-based RL

In model-free RL, we do not have any clue of the environment's dynamics. Still, proper choices need to be done in order to maximize rewards. A value function estimates how good a state is, not only for the short-term, but also for the long term. This value function is also conditioned on a given policy π , so we will define the value function as in equation 2.3, where we define the episodes of experience following a policy as $S_1, A_1, R_2, \dots, S_k \sim \pi$ and the returns $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots + \gamma^{T-1} R_{t-1}$. The γ value represents the discount factor, that is a value less than 1 that ponders the future rewards along the time-steps. With this set-up, future rewards will have lower importance in the return value, while immediate rewards will be given more value.

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \quad (2.3)$$

For various purposes, the state information may not be sufficient to estimate the value of a state itself, so the action must be part of the value associated. Formally, it is described as in equation 2.4 and this kind of values are formally known as action values.

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \quad (2.4)$$

Usually, in model-free reinforcement learning Q -values are common approach, since with the state values, the environment dynamics \mathcal{P} should be at least known or estimated. This is why in following sections, the formulation will be more focused on Q -values. Section A.1.3 expands on this matter.

There are several approaches that make use of the value function definition to make estimations about it. In this section we will briefly go over Temporal Difference Learning:

TD(0) and a two special cases: Deep Q-Networks (DQN) [10] and Double Deep Q-Networks [11] (DDQN). Again, if deeper insights are needed, we encourage the reader to take a look at section A.1 where we explain in depth several important aspects of methods like Monte Carlo learning or TD learning. Our main focus will be in the value function approximators, since this is when deep learning made its appearance in the RL framework.

2.1.2.1 Value function approximation

Environments in RL may end up being quite complex. This complexity usually comes from the number of possible states of our environment. For example, a robot walking down a street finds itself constantly in new states, as the observation of the environment changes constantly (i.e people coming by, cars running down the street, etc.). Creating a table that models such a complex space may be infeasible, but we can use and optimize a high parameterized function to approximate the optimal value function, and here is where deep neural networks make their appearance. Formally, the goal of the optimization is described in equations 2.5 and 2.6, where w is a vector that holds the weights (or parameters) from the function approximator.

$$\hat{V}(s, w) \approx V_\pi(s) \quad (2.5)$$

$$\hat{q}(s, a, w) \approx q_\pi(s, a) \quad (2.6)$$

In this section we are going to focus on the action value function $\hat{q}(s, a, w)$. This function approximator could be defined as the combination of the input features ϕ by the weights. For example, in equation 2.7 we see a value function approximation implemented using a linear regression.

$$\hat{q}(s, a, w) = \phi(S, A)^T w = \sum_{j=1}^n \phi_j(S, A) w_j \quad (2.7)$$

The objective of Value Function Approximation is similar to a supervised learning scenario. In equation 2.8 it is defined to minimize the Mean Squared Error (MSE) of the difference between the objective function $q_\pi(s, a)$ and the approximation $\hat{q}(s, a, w)$. This optimization is done via Stochastic Gradient Descent (SGD) (equation 2.9) where we derive the MSE with respect to the set of parameters w , looking for the steepest part of the curve for descending it in order to minimize the error. Since we do not have the objective function $q_\pi(s, a)$, we must adjust our parameters to the estimations we collect as we explore the state space.

$$\mathcal{J}(w) = \mathbb{E}_\pi[(q_\pi(S, A) - \hat{q}(s, a, w))^2] \quad (2.8)$$

$$-\frac{1}{2} \nabla_w \mathcal{J}(w) = (q_\pi(S, A) - \hat{q}(s, a, w)) \nabla_w \hat{q}(s, a, w) \quad (2.9)$$

This approach can be applied to TD learning for example. In the case of TD(0) (equation 2.10), the adjustments for the weights of the function approximation Δ_w would be done using the difference of the information we already know $\hat{q}(s_t, a_t, w)$) and the new information that we have acquired with the new transition $R + \gamma \hat{q}(s_{t+1}, a_{t+1}, w)$.

$$\Delta_w = (R + \gamma \hat{q}(s_{t+1}, a_{t+1}, w) - \hat{q}(s_t, a_t, w)) \nabla_w \hat{q}(s_t, a_t, w) \quad (2.10)$$

Once the adjustments are known, we can update the weights using a learning rule, as we define in equation 2.11, where α is the learning rate and w_t is a vector that represents the current state of the weights in the function approximator.

$$w_{t+1} = w_t + \alpha \cdot \Delta_w \quad (2.11)$$

One of the biggest success in the value function approximation paradigm is Deep Q-Networks (DQN) [10] that uses Q-Networks to estimate the value function using neural network. This algorithm first produces experiences and stores them into the experience buffer $\mathcal{D} = \{(s_i, a_i, r_i, s_{i+1}, d_i)\}_{i=1}^N$, where N is the size of the buffer. Then, with an off-policy Q-learning approach, a loss function (equation 2.12) is proposed. The main goal is to minimize the MSE, and, as it does, it learns a value function that produces the Q -values for each possible action.

$$\mathcal{L}_w = \mathbb{E}_{\mathcal{D} \sim s, a, r, s'} \left[\left(r + \gamma \max_{a'} Q(s', a'; w^-) - Q(s, a; w) \right)^2 \right] \quad (2.12)$$

where:

- $Q(s, a; w)$: represents the online Q-network.
- $Q'(s', a'; w^-)$: represents the target values estimated by the offline Q-network. The weights of this network will be updated periodically copying them from the online Q-network every fixed time-steps.

The main intuition behind this loss function resides in replacing the tabular methods from the Q-learning set-up for Q-network, that estimate the Q-values for a state. The offline network $Q(s', a'; w^-)$ is the target and the online Q-network $Q(s, a; w)$ tries to minimize the error between its predictions and the target. This makes the online network to update its predictions towards what the offline network is estimating. Years later, Double Deep Q-Networks (DDQN) [11] made a few updates upon DQN, achieving ever better results. The goal is to minimize the loss function presented in equation 2.13

$$\mathcal{L}_w = \mathbb{E}_{\mathcal{D} \sim s, a, r, s'} \left[\left(r + \gamma \max_{a'} Q(s', a'; w^-) - Q(s, a; w) \right)^2 \right] \quad (2.13)$$

The motivation behind this update mainly reside in the fact that DQN overestimates the Q-values, which bias the agent into thinking that some states are more valuable than what they really are. This is because in the DQN loss, the action selection is done by maximizing the estimates of the target network, but, since we are trying to evaluate how good the online network is doing, it makes more sense to evaluate the action that the online network selects by using the offline network, which gives us a better idea of how the agent is performing.

2.1.3 Policy-based RL

In the value based methods, the policy usually is deterministic. In section A.1.4, we go deeper in exploring greedy and ϵ -greedy policies as a mapping from state/action value functions to actions, but the main important take, is that deterministic policies are not always enough for some complex environments. We can make use of parameterization in order to make our agent more capable of dealing with harder problems expanding the representation capabilities for decision making. A good policy usually is the one that maximizes the reward in each time-step, so we must always look in the direction that provides greater expected rewards using action selection.

2.1.3.1 Policy Gradient Theorem

The parametrized policy is defined as in equation 2.14. Where θ is the parameter vector of our function approximator (i.e. a neural network).

$$\pi(a|s, \theta) = \mathbb{P}[a|s, \theta] \quad (2.14)$$

These parameters must converge to a solution where our policy maximizes the rewards, taking actions that consequently do so. The policy gradient theorem states that in order to obtain the optimal policy $\pi^*(a|s, \theta)$ that maximizes the rewards, we must compute the gradient defined in equation 2.15

$$\nabla_{\theta} \mathcal{J}(\theta) = \mathbb{E}[\nabla_{\theta} \log(\pi(a|s, \theta)) Q^{\pi_{\theta}}(s, a)] \quad (2.15)$$

Where the term $\nabla_{\theta} \log(\pi_{\theta}(s, a))$ gives us a likelihood estimation of how much does the agent want to be in the same state and action pair. If the reward is big, we want that gradient to be big, and follow that direction, using Stochastic Gradient Ascend (SGA). On the other hand, the term $Q^{\pi_{\theta}}(s, a)$ measures how beneficial was taking the action a . It ponders the direction of the gradient we want to maximize. If we had a situation of great value, then that gradient should be explored more, hence the bigger values, but, if the gradient or the value is low, the agent should not be in that position very often, to avoid loss of potential reward.

One of the first policy gradients method is REINFORCE [12], which is based in the Monte Carlo learning method. Here $Q^{\pi_{\theta}}(s, a) = v_t$, where v_t is the cumulative expected reward for time-step t . The weight updates are computed as defined in equation 2.16, where t is the current time-step. For further reading, more in depth detail about different policy gradients methods such as Actor-Critic (A2C) [13] is discussed in section A.1.5.

$$\Delta \theta_t = \alpha (\nabla_{\theta} \log(\pi(a|s, \theta))(s, a) v_t) \quad (2.16)$$

2.2 Attention-based models

2.2.1 What is attention?

The attention mechanism was first introduced in [14]. There, the authors explain how difficult is to make translations from different languages. The issue in particular was the alignment, meaning that the number of tokens or words in a translated sentence usually is different compared to said sentence in the original language. This presented a challenge for the network since it had to detect which parts of the original sentence produced the resulting translation at token level. For example, the word "crosswalk" in English, translates to Spanish as "paso de peatones" which is not a literal translation. Therefore, the task presented in this particular example would be to teach the network to align a single word in English to the counterpart three words in Spanish. We encourage the reader to go to section B.1 for further reading on the details of the attention mechanism.

The authors tackled this problem by using a recurrent encoder-decoder architecture using recurrent neural networks (RNNs) as presented in figure 2.2, extracted from [15]. The encoder is based on an Bidirectional Long-Short-Term-Memory (Bi-LSTM) network, that takes the token embeddings from the input, and processes them creating the vector h_j , which contains all the information from the input up to time-step j . The decoder is another RNN network that has as input a context vector c_i and maps it to a hidden state vector s_i and an output vector y_i , where i is the time-step of the output. The context vector c_i is obtained by pondering the embeddings produced by the encoder using vector α_i .

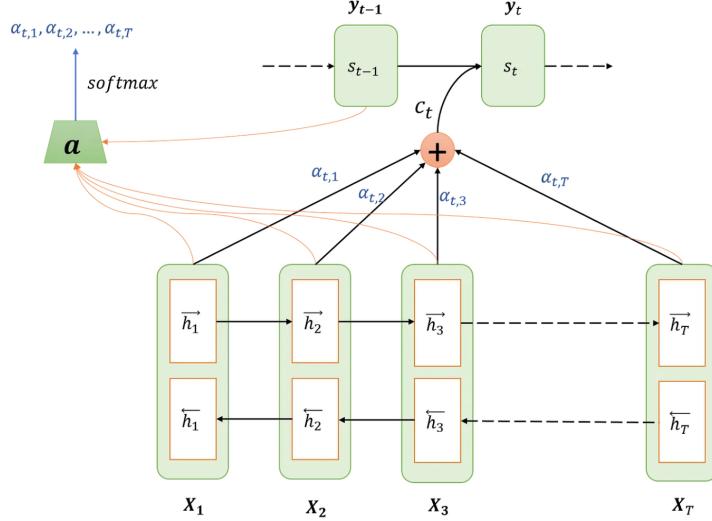


Figure 2.2: Aligning inputs using the attention mechanism. First, the inputs $X_1, X_2, X_3, \dots, X_T$ are encoded using a bi-directional RNN, obtaining the embeddings \vec{h}, \hat{h} . This embeddings are then pondered by the product of the learnable parameters α_{it} for each input, creating the context vector c_t . Finally, c_t is fed into the decoder, updating the hidden state s_t and producing the output y_t .

2.2.2 The Transformer Architecture

The architecture introduced in section 2.2.1 had its limitations, especially in modelling dependencies between sections of the input text that were far apart. This was because the fixed compressed representation (the embedding) of the input of the decoder only holds information from the closest tokens, not being able to model long-range dependencies. To tackle this problem the Transformer architecture [2] introduced self-attention, where the whole sequence is processed at once and the dependencies are computed altogether, instead of processing the inputs one per time-step. Self-attention is defined as in equation 2.17, where Q, K, and V represent the query, key, and value matrices, respectively, d_k represents the dimensionality of the keys and queries, σ represents the soft-max function applied along the rows of the matrix resulting from the dot product of Q and K^T .

$$\text{Attention}(Q, K, V) = \sigma \left(\frac{QK^T}{\sqrt{d_k}} \right) V \quad (2.17)$$

This creates a vector of weights that ponder the importance of each element of the input with respect to the rest of the elements, describing how relevant are for the model. Finally, these weights are multiplied element-wise with the value matrix V to compute the final attention output. We can think about the query (Q) as a vector that holds information about what each token in the input is looking for, meaning, which other tokens is interested in. On the other side, the keys (K) represent the information each token from the input holds. When transforming the input onto the query-value space, vectors (embeddings) that are close will have a higher dot product result, meaning that the "energy" of their semantic relation is bigger. On the other hand, if those vectors are quite far, the dot product will be smaller, hence, the energy will be lower, meaning that those vectors are not quite related. The implications that this work had in the AI research field were huge, and it was in part, because it revealed a new upcoming discovery for explainability in neural networks, thanks to the weights the

attention matrix holds

What we have explained up until now it is known as an attention "head", this means that there is only one participant looking for relevant information in a sequence, but, it is well known that having several model instances cooperating to solve a problem is always better [16]. In order for the attention mechanism to look at different parts of the inputs at the same time, we need multiple instances looking at different parts and communicating between them. This is known as Multi-Head Self Attention (MHSA). When we have multiple attention heads [2], we can extract a rich feature map that ponders the importance of different parts of information from the input. Because multiple heads can focus on different parts of the input, our attention maps will be quite diverse, potentially preventing over-fitting. This approach also takes advantage of distributed computation, because computing the self-attention for each attention head can be done in parallel, making it efficient in time. Figure 2.3 provides a diagram that explains the very same concept discussed in this section.

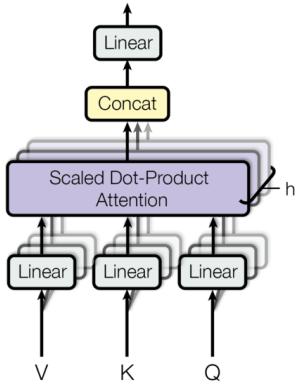


Figure 2.3: Multi-head attention. The embeddings from Q, K and V are computed and fed into the attention module. Once computed, the attention matrix is computed for each of the heads, and concatenated to then, be forwarded onto a linear layer, that computes the embedding of the attention-weighted input.

In figure 2.4a, we can see the complete transformer architecture. First we have the encoder, that uses attention to generate the embeddings of the sequence, where in each time step, the attention heads will focus in different parts of the input. Then, the decoder takes those embeddings and starts producing the outputs in an auto-regressive manner, taking its own outputs as input embeddings to the decoder that are mixed with the embeddings of the encoder, to provide a wider context of the whole sequence.

Naturally, one can see the attention weights are the main piece of information that explains the output. Attention weights provide a numerical description of which parts of the input the algorithm is looking for, thus providing an approximate explanation of the output. In figure 2.4b we have an example of how the bi-RNN model with attention, explained in section 2.2.1, and transformer performs text analysis. The lines in red correspond to the attention performed by the bi-RNN model, where we can see that short-range relations are well modeled, by long range are not. Opposed to that, the lines in green correspond to the transformer, where we can see that the long-range relations between words are more intense, proving that the transformer can model wider contexts in sentences.

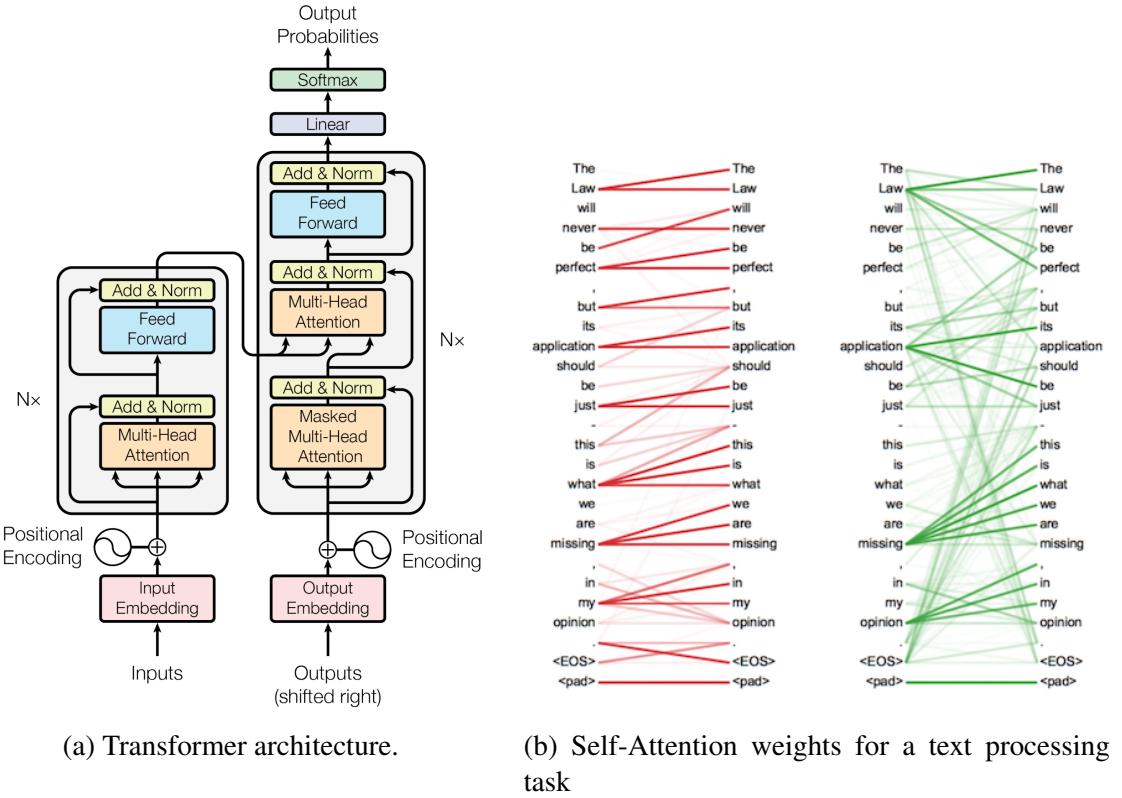


Figure 2.4: In figure 2.4a (left), from [2], the complete transformer architecture is presented. In figure 2.4b (right) from [3], an attention matrix is presented for a sentence and its translated counter-part. The weights for each word quantify how much *attention* is the model paying to the words from the original sentence in order to produce the translation to the new sentence.

2.2.3 Vision Transformers

Self-attention is a powerful mechanism, that can be applied not only to sequences, but also to visual data. The vision transformer (ViT) [17] was one of the first models that leveraged the self-attention mechanism in order to process images. The architecture is presented in figure 2.5a, where the first thing to notice is that there is only an encoder in the architecture. This is because this model was designed for image classification, and the best approach to solve this task is to extract an embedding as a compressed representation of the original image. The input is processed by patching the image, creating different blocks that serve as input. To take context of which patch is in which position of the image, the ViT model has a learnable parameter that serves as an absolute positional embedding. Then, the attention heads will look at each of the patches separately. To do this, the ViT flattens the patches from the images and treats them as a sequence, paying attention to the features from the raw input that are more relevant. By having several attention heads looking at several patches at the same time, the model captures global dependencies between them, obtaining a rich representation of the image. After that, the attention matrix computed from each head is concatenated and forwarded into a multi-layer perceptron (MLP) that produces the embedding that the classifier will use to produce the output.

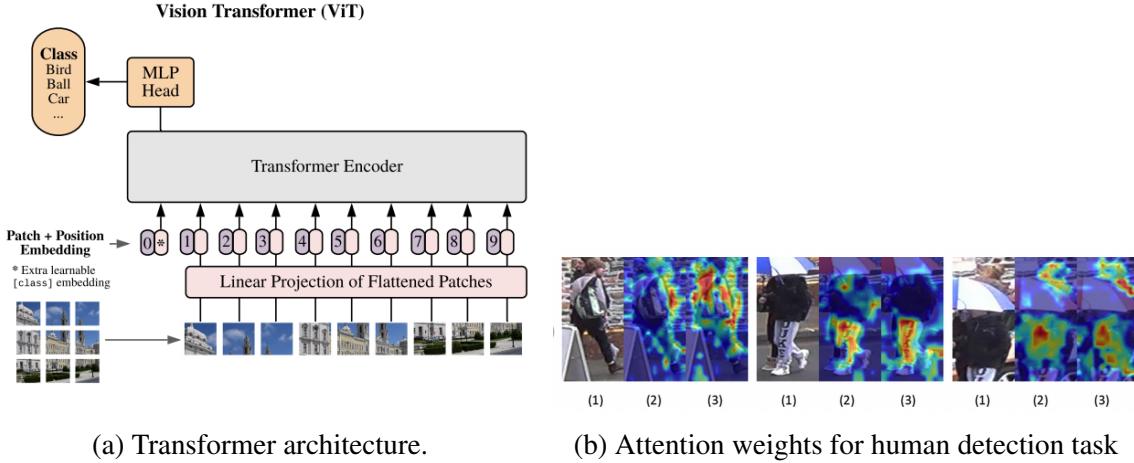


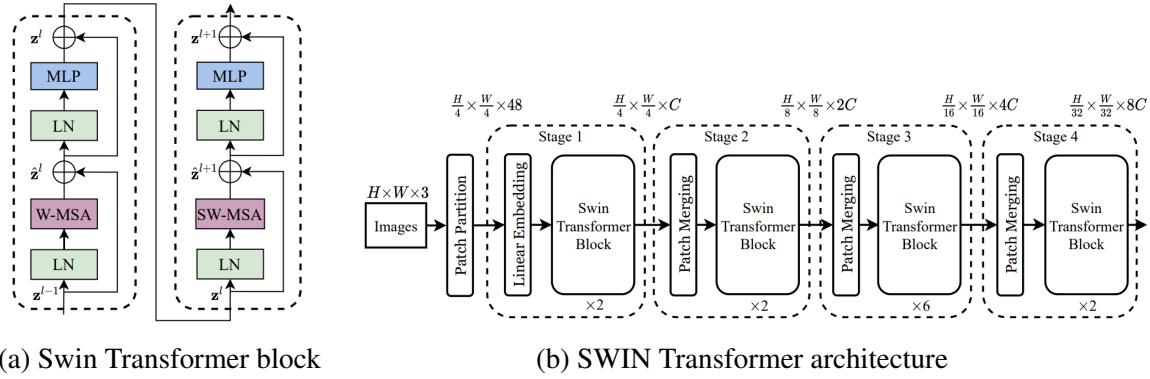
Figure 2.5: In figure 2.4a (left) the ViT transformer architecture is presented. In figure 2.5b (right) an attention map is presented. The model that produced the attention maps was trained to track people on a scene, and it is why, the higher attention weights (red) are put in the pixels where people is present.

2.2.4 SWIN Transformer

One of the main issues that the ViT model holds, is the computational cost of the self-attention module being quadratic. This means that, as images are bigger, the number of patches where the attention is performed will increase, making the input embeddings more expensive to process. To solve this, the SWIN Transformer [18] (figure 2.6) takes an alternative approach, and uses a hierarchical architecture that resembles the CNNs. First, it divides the image in patches, as the ViT model does, but, instead of computing the attention, it adds the concept of window attention to the mix. A window will contain a set of patches, and the attention will be performed between the patches that are inside the window. This reduces the complexity of the attention operation, but at the cost of global feature extraction, since the patches inside of a window only know the existence of themselves, but not of their neighbours under a different window. To solve this, the SWIN Transformer introduces masked window shifting and patch merging.

Masked window shifting, presented in figure 2.7a is a simple but effective procedure to broaden the spatial context of the windowed attention. The intuition is to drift the layout of the patches inside the image, providing a wider spatial context to the attention performed inside the windows. This means that the group of patches that are left out of the image (sections A, B, C) are mirrored to the symmetric part of the image, performing a "cyclic shift". This cyclic shift alters the spatial disposition of the elements in an image, since the parts of the top left are now in the bottom right. To handle this, they provide a mask that ensures that the shifted regions only attend to their corresponding parts. For example, the patches inside region A will only attend to patches that are only inside said, region. This ensures that the spatial information remains unaltered and no "abnormal" representations are learned as a product of this shifting.

On the other side Patch Merging, figure 2.7b, is an operation that uses concatenation along the channel dimension to aggregate the features from the patches inside a window. This provides a hierarchical architecture, where the local features are captured in the first stages of the architecture, since the spatial dimension is bigger, and the global features are captured at the latest stages, where the spatial dimension is diminished, and all the local



(a) Swin Transformer block

(b) SWIN Transformer architecture

Figure 2.6: Swin Transformer architecture and its building blocks. The transformer block, figure 2.6a, has two phases. In the first one (left-most block), it performs a layer normalization, followed by a window-MSHA block that is forwarded onto a MLP with a neurons ratio of 4. In the second, it makes use of the shifted window MSHA, instead of the window-MHSA. In figure 2.6b we can see the whole SWIN Transformer architecture, where each of the "stages" are built upon the transformer blocks, that have as input the patch merged embeddings that are passed at each stage.

features are aggregated along the channel dimension. The patches are selected in a 2 by 2 grid, so the input has a shape of $W \times H \times C$, and after the patch merging, has a shape of $\frac{W}{2} \times \frac{H}{2} \times 4C$. In some type of way, this could be seen as a "*convolutional transformer*", since it quite resembles some of the hierarchical mechanisms that the CNNs have to perform feature extraction from an image.

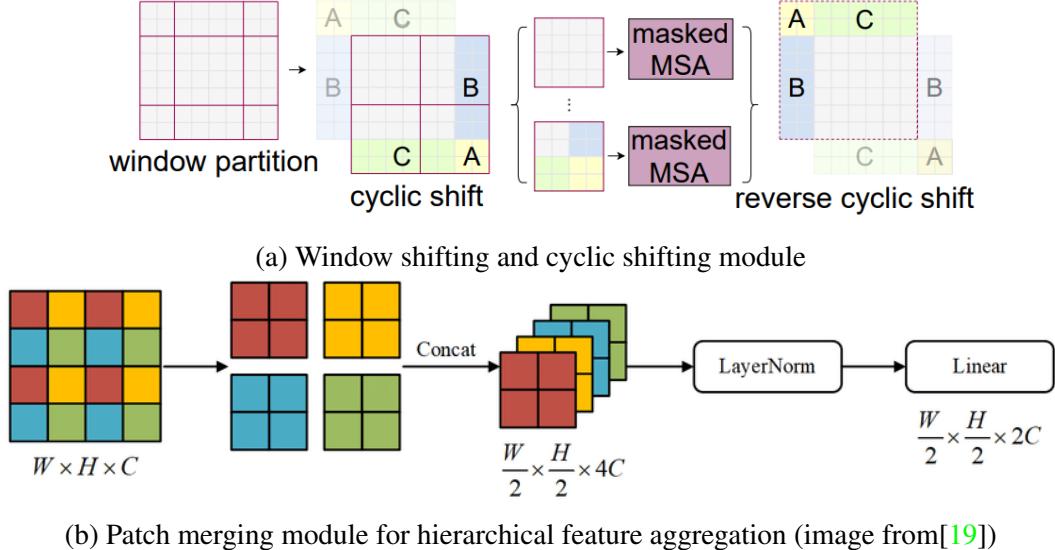


Figure 2.7: Window shifting in the SWIN Transformer architecture (figure 2.7a). From the window partition, shift to the bottom right, so the A, B, C patches are rearranged in the mirrored side of the image. Patch Merging (figure 2.7b), picks blocks of 2 by 2 patches inside a window and concatenates them along the channel dimension.

2.3 Explainable AI

Explainable AI (XAI) refers to techniques and methods that make the behavior and decisions of artificial intelligence systems transparent, understandable, and interpretable to humans. It aims to clarify how AI models reach their conclusions, enhancing trust, accountability, and usability, especially in critical applications like healthcare or finance. The concept of explainable AI has been around for quite some time. In the eighties, in [20], Swartout explains that the need for systems that have "cognitive" abilities to solve complex problems must come with an explanation, and developed a basic rule-system to explain the results. It is clear that, since the AI research began, researchers have always claimed the need for mechanisms and systems to explain decision making. There are some AI models that are quite good at this, like decision trees, since it is quite common to follow down the nodes and leaves (questions and conditions that the tree proposes in order to perform inter-class separability) to understand the decisions performed. The issue comes when the complexity of the model increases. For example, a random forest is a set of decision trees that jointly participate in a poll to determine, the class of a given input. The set of decision trees can be huge for complex datasets, and to trace each and everyone of the decisions performed by all of the trees can be an exhaustive process. Something similar happens with deep neural networks. With the appearance of AlexNet [21] in the ImageNet challenge [22], a shift of paradigm appeared, since the following years, deeper models were presented to the challenge. Results at the time were astonishing, but with a downside, since as the network got deeper, the interpretation of the activation maps in the receptive fields of the layer was almost impossible, which led to a big concern: what is the network looking at in order to make the predictions? One thing is clear, as shown in Figure 2.8 extracted from [23], the accuracy is higher the less explainable is the model.

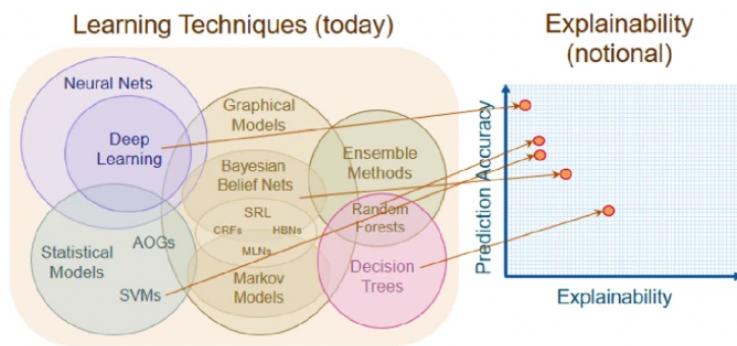


Figure 2.8: Explainability of ML models against the prediction accuracy.

In the past decade, the field of computer vision has been involved a pursuit to make deep neural networks (DNNs) transparent and understandable. In the following sub-sections we will briefly explain the techniques that have been used in this work along with another relevant works from the XAI field.

2.3.1 Sensitive Analysis

In [24, 25], sensitive Analysis (SA) is defined as a method that identifies which parts of the input are more sensitive to the network to perform the predictions. To do so, it assumes that the most relevant input features are the most sensitive for the output, and by "obstructing" pixels or sections of the image, it ponders how relevant are different sections to the predicted

output. The mathematical definition behind this concept is defined in equation 2.18, where they define all the partial derivatives for a function $f(x)$ as the sensitivity s_{ik} of the network to a certain input x_n with respect to all the i-th elements of the input x_i and the output y_k .

$$s_{ik} \Big|_{x_n} = \frac{\partial y_k}{\partial x_i}(\mathbf{x}_n) \quad (2.18)$$

The issue that appears with SA is that it does not explain the function approximation performed by the neural network, but only quantifies the importance of the input. This may not be ideal, as we want to understand what the network gives more importance to with respect to the features of the input, not understand which parts of the network are essential to produce the output.

2.3.2 Layer-wise Relevance Propagation

Layer-wise Relevance Propagation (LRP) [6] is an interpretability method used to explain the predictions of neural networks. The main idea of LRP is to propagate the prediction score $f(x)$ backwards through the network layers, assigning relevance scores R_i to each neuron in the network.

There are several variations of LRP to interpret neural network predictions, but one of the most useful ones is the LRP- $\alpha\beta$, which ponders between the positive and negative contributions to the prediction, given an input. The relevance R_i of each neuron i is traced back layer by layer, indicating the contribution of each neuron to the final prediction. To distinguish between positive and negative contributions, LRP decomposes the term $Z_{ij} = a_i^{(l-1)} \cdot w_{ij}$, where $a_i^{(l-1)}$ is the activation of neuron i and w_{ij} is the weight connecting neuron i to neuron j . The positive and negative parts are defined as $z_{ij}^+ = \max(0, z_{ij})$ and $z_{ij}^- = \min(0, z_{ij})$. The LRP- $\alpha\beta$ then, combines these parts to propagate the relevance from the current layer l to the previous $l - 1$, as defined in equation 2.19

$$R_i^{(l-1)} = \sum_j \left(\alpha \frac{z_{ij}^+}{\sum_{i'} z_{i'j}^+} R_j^{(l)} - \beta \frac{z_{ij}^-}{\sum_{i'} z_{i'j}^-} R_j^{(l)} \right) \quad (2.19)$$

2.3.3 Class Activation Maps

Another widely-known approach for explainability is called class activation maps (CAM). Introduced in [26], the main intuition behind this technique is to use the activation units of the last layers, specially the Global Average Pooling to understand which sections of the input image generate relevant information for the final softmax layer. It is formally defined in equation 2.20, where $M_c(x, y)$ refers to the activation for the x, y pixels in the input space, given the class c , w_k^c refers to the weights that go from the input neuron k (the channel) to the output space c , and $f_k(x, y)$ refers to the unit k in the last convolutional layer, given the input at coordinates x, y of the image.

$$M_c(x, y) = \sum_k w_k^c \cdot f_k(x, y) \quad (2.20)$$

An extension to this work is developed in [7] called Grad-CAM. They use the gradients of the output with respect to the activations to compute the activations of the network under a desired class c . First, The input image \mathbf{I} is passed through the CNN to obtain the feature maps from the last convolutional layer, denoted as A^k , where k is the channel index. The

network also produces a prediction score y^c for each class c . To compute Grad-CAM, we need the gradient of the score for the target class y^c with respect to the feature maps A^k :

$$\frac{\partial y^c}{\partial A^k}$$

This gradient indicates the importance of each pixel in the feature maps A^k for the target class c . Said gradients are globally averaged over the width and height dimensions to obtain the weights α_k :

$$\alpha_k = \frac{1}{Z} \sum_i \sum_j \frac{\partial y^c}{\partial A_{ij}^k}$$

where Z is the number of pixels in the feature map ($Z = H \times W$) which serves as a normalization term, and A_{ij}^k is the value at position (i, j) in the k -th feature map. The operation of the global average pooling seems to perform better empirically, according to [7]. To properly ponder the activations, the weights α_k are used to compute a weighted sum of the feature maps:

$$L_{\text{Grad-CAM}}^c = \text{ReLU} \left(\sum_k \alpha_k A^k \right)$$

Additionally, the ReLU function is applied to the activations in order to obtain only the positive contributions that "help" the gradient to find the most relevant regions from the input.

The results that Grad-CAM provides are coarse, since we are generally focusing in the last layers of the network, where the spatial information is limited. To overcome this, $L_{\text{Grad-CAM}}^c$ is up-sampled to the size of the input image using interpolation methods.

2.4 Reinforcement Learning using Transformers

Recently, there has been a trend into fitting the Transformer model into the RL problem. The motivation behind this resides in their quality in feature representation extraction, and, given the nature of RL and function approximation, it is natural to leverage these models abilities to try to solve the RL problem [27]. In the case of the ViT model in [28] they use a Q-learning set-up in order to test whether the feature representation that provide the ViT is compatible with sample efficiency, which is a great concern in the RL field, since neural networks are known as "data-hungry" models. And, even though results are not state of the art, this work provides an alternative framework for the RL problem. Although the ViT is not widely used in the RL paradigm, we argue that with a proper set-up it can lead to interesting findings in RL.

Swin Transformers have also been used to perform DDQN Learning [4]. Using a similar approach to [11], they use a SWIN Transformer as a Q -value function approximator. As we mentioned in section 2.2.4, the ViT transformer has a quadratic cost in terms of computing the self-attention mechanism, and since DQN-learning is a sample intensive task, researchers opted to use a less computational expensive model to try to solve the problem of RL. Results in the Arcade Learning Environment (ALE) [29] set-up are state of the art in several games, but little is explored about how the attention mechanism influences the decision making of the agent.

Also, we think it is worth mentioning other approaches that have been proposed to solve RL using transformers. In [30], they try to model RL as a big sequence prediction problem in

an offline RL set-up, where the next action is conditioned in the past actions and reward, thus expanding the MDP approach. Right after, in [31] they introduced the decision transformer, a similar model that is focused in predicting the action that maximizes the reward based only in collected experiences from previous, online trained agents, but also in an offline set-up. There has been also several intents of learn-on-the-run transformer-based set ups. For example, in [32], they propose the bootstrapping transformer, a model that generates its own trajectories and learns from them, highly boosting the sequence offline training procedure.

2.5 Explainable RL using Attention

Explainable RL or (XRL) is a field of study that aims to understand and to provide tools that improve a RL agent explainability of their decision making. We as humans, can explain most of the times why we take decisions or perform actions, enabling other humans to understand our behaviour. The same should happen with autonomous systems, so when implemented in industry related environments such as factories, vehicles or bots, we can understand why an agent took a decision or performed an action.

Literature in explainability for attention based architectures in RL is scarce, although some interesting proposals can be found. In [33], they propose an attention-weighted reinforcement learning approach. They use as inspiration [34], where Yuan Chang Leong et al. proposed a series of tasks in order to evaluate which parts do humans pay more attention to solve them. They argue that humans attention can be approached as a hierarchical system where, as we gain knowledge from general features, the attention provides contextual information, coding the essential parts and leaving out the less relevant ones. In order to model this, they propose a value function that takes into account stimuli from multi-dimensional features to then generate a value as a numerical signal associated with them. They weight the values from said features depending on the attention that subjects pay to them. In [33], they model this relations using self-attention, with promising results when correlating features with the attention maps in tasks that involve playing games in the ALE Atari 2600 environment, but this comes with a condition. Their set-up assumes prior knowledge about several positions of elements in the environment, such as the ball coordinates or other players/enemies position in the board, conditioning the learning process, especially in set-ups where observations are visual cues. We argue that using attention-based models that show which parts are they paying attention the most, we could obtain better explanations on the visual cues taken into account to perform a decision.

Chapter 3

Design and development

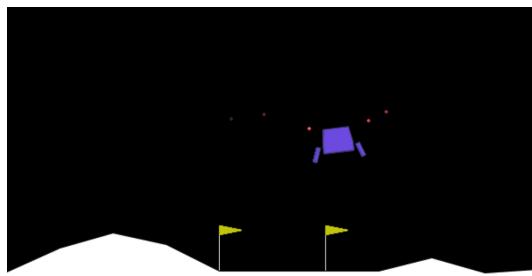
As previously explained, one of the main differences in the RL paradigm is how the agent acquires data. Opposite to the datasets or databases in the supervised learning environment, where we have an input and a ground truth structured in such a way that the model can learn from the data adjusting itself to the ground truth, here, the data is obtained via observations from an environment, that are associated with a reward and a observation of the next state. On the other hand, we have to create a seamless pipeline for a deep neural network model that interacts with this environment and learns from it using two different off-policy algorithms: DQN and DDQN.

In this section we will explain step by step how we implemented this framework, from the environment instantiation, the post-processing of the observations provided by the environment, how we modelled the interactions of the agent over the environment and which tools we used for the memory replay module of the several agents that we developed. Given the short amount for experiments that we had, we have only trained agents for two environments for the Arcade Learning environment: **MsPacman-v5** and **Demon-Attack-v5**. Additionally, we will also explain how we fitted deep learning into this problem, going from less complex models as CNNs to more complex such as transformers.

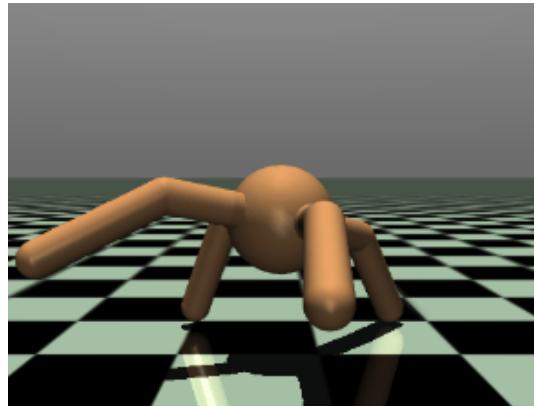
3.1 Farama's gymnasium library

The Farama's gymnasium (Figure 3.1) library was originally developed and released by OpenAI and was a huge leap in the RL realm, since before this, researchers had to develop their own environments or games to test their agents. Using games is a great way to test if an agent is actually learning from an environment, and one of the first frameworks that showed this was the Arcade Learning Environment or ALE [29], where Bellemare *et al.* proposed a platform with different games that were originally developed for the Atari 2600 console. ALE was quite impactful, since lots of relevant works have benefited from this platform to test and launch their agents, providing a kind of benchmark. The ALE environment was included in the Farama's gym, along with other platforms such as Multi-Joint dynamics with Contact (MuJoCo), a physics engine that test an agents ability to perform control over complex robotic entities (Figure 3.1b) or Box2D, a platform that involve toy games in order to perform control on several vehicles, such as space-ships or cars (Figure 3.1a).

This library will be our starting point. A deep understanding in how it works is crucial for an efficient and insightful development of our deep learning-based agents.



(a) Gym landing environment.



(b) Ant MuJoCo environment

Figure 3.1: In figure 3.1a (left) the goal is to land securely the spaceship (in purple). In figure 3.1b (right) the goal is for the ant to balance itself and start walking.

3.1.1 Environment dynamics

The way that Farama's gym models agent-environment interaction is quite similar to what we explained in section 2.1.1. In the listing 3.1 we see a basic example of how an interaction with an environment can be performed. First, we need to instantiate the environment, calling the `gym.make` function, with the argument passed as the name of the environment we are going to interact with. In order to initialize the environment, we must reset it, using the `env.reset()` method. Once we have our environment ready, we can proceed to interact with it performing actions. The number of actions, called action space, varies depending on the environment, but the gymnasium library offers methods and attributes that facilitate them in run-time, such as `env.action_space.n`. In the example provided, the action taken is random, using `env.action_space.sample()`, but we can use more complex methods to select the action our agent is going to perform (i.e. a neural network).

```

1 import gymnasium as gym
2 env = gym.make("LunarLander-v2", render_mode="human")
3 observation, info = env.reset(seed=42)
4 for _ in range(1000):
5     action = env.action_space.sample()
6     observation, reward, terminated, truncated, info = env.step(action)
7
8     if terminated or truncated:
9         observation, info = env.reset()
10
11 env.close()

```

Listing 3.1: Initialization of an environment in Farama's gymnasium

In the RL framework, when an action is selected, the agent performs said action in the environment, and obtains a reward and a next observation. By repeatedly doing so, a series of trajectories are created (as defined in equation 2.1), until the agent reaches a final step. The whole trajectory it is called an episode. In the code, the method `env.step()` is in charge of performing the steps of an episode, simulating in each time-step what the outcome will be, returning the following:

- **Observation:** What the environment lets the agent see about its state. If state = observation then the environment is a MDP, if not, then its a POMDP.

- **Reward:** The numerical signal that represents how good (or bad) the actions the agent is taking are.
- **Terminated:** Boolean flag that represents if the episode has ended.
- **Truncated:** Boolean flag that represents if there has been an unexpected situation that resulted in the current episode ending prematurely.
- **Info:** Additional information from the current state of the environment, such as current lives if the game is a survival, the current score or even and additional final observation if the state is terminal.

The interacting loop could go forever if no additional condition is stated, that is why using the truncated and terminated flags is the common approach to reset the environment and start a new episode. Once the simulation is over, we can close the environment using the `env.close()` method.

3.1.2 Processing techniques of an environment

In most of the environments, observations are usually RGB frames displaying the current situation of the environment. Since our models deal with visual data, it is quite convenient to use some pre-processing techniques that simplify the observation into something that is more "consumable" by the DQN agent. Farama's gymnasium provides an API that allows modifications of the environment, called **wrappers**. These wrappers are conceived so that they can alter the original behaviour of the environment, in terms of the reward, number of frames skipped between steps or the colour gamut of the observation. This operations are analogous to the ϕ operator defined in equation 2.7, since wrappers try to extract the most essential features from the raw input.

In our implementation, we make use of four custom wrappers, along with some pre-defined offered by the gymnasium library. This will ease the observation processing a make lightweight computations by discarding redundant information provided by the environment. In the following section we explain some of the custom wrappers that we have developed, along with some pre-defined ones.

3.1.2.1 Skip N-Frames

In a RL environment, the variation between two consecutive frames usually is very little. If we pass each observation the environment provides, we end up with lots of redundant information which means that lots of computational resources are used in vain. One of the main approaches that is done in RL and other fields such as robotics or autonomous driving, where high frequency sensors are used is to skip a fixed amount of frames per observation. This technique is usually known as "frame skipping". The code developed for this wrapper is shown in listing 3.2. One thing to note is that in terms of reward, we add in the n-th frame the accumulated rewards from the skipped observations, since we are comprising the information of n-frames in just one observation.

```

1 class SkipFrame(gym.Wrapper):
2     def __init__(self, env: gym.Env, skip):
3         """Returns only the 'skip'-th frame."""
4         super().__init__(env)
5         self._skip = skip

```

```

6
7     def step(self, action):
8         """Repeat action, and sum reward"""
9         total_reward = 0.0
10        for _ in range(self._skip):
11            obs, reward, done, trunk, info = self.env.step(action)
12            total_reward += reward
13            if done or trunk:
14                break
15
16    return obs, total_reward, done, trunk, info

```

Listing 3.2: Frame skip wrapper

3.1.2.2 Gray-Scale Observation

Most of the gymnasium's environments are in RGB colour coding. This may be useful for us humans, since we can extract additional information about the state of the game (i.e. detect if the ghosts in the Pac-man game are or colour blue or not, which gives us a hint on whether we can "eat" them and win points). Machine vision does not work like this, and this information sometimes is seen as redundant, since for a machine, the greyscale palette is enough. Also, it has its limitations in terms of computing power and resources. RGB images have three channels, which makes the observations more expensive to process, specially if we are stacking n-frames to them. Also, since we are dealing with experience replay models, storing these observations along with the collected experience can result in more memory consumption, which is not optimal in the training process. To solve this, the RGB observations are transformed to integer grey-scale. In this case, we are going to make use of PyTorch's transforms from the torchvision library that adds several visual transformations such as cropping, rotation, inversions or colour palette transformations. Using these transformations, we can also cast the original data-type of the observation (`numpy.array`) to a `torch.Tensor` datatype, which is convenient for processing said observation using a neural network using PyTorch framework. The code developed to apply this transformation to the environment is displayed in listing 3.3.

```

1 class GrayScaleObservation(gym.ObservationWrapper):
2     def __init__(self, env: gym.Env):
3         super().__init__(env)
4         obs_shape = self.observation_space.shape[:2]
5         # since we are wrapping the observation that an environment
6         # provides,
7         # we must update the observation space to match the new wrapped
8         # env.
9         self.observation_space = Box(low=0, high=255, shape=obs_shape,
10                                     dtype=np.uint8)
11
12     def permute_orientation(self, observation):
13         # permute [H, W, C] array to [C, H, W] tensor for pytorch model
14         observation = np.transpose(observation, (2, 0, 1))
15         observation = torch.tensor(observation.copy(), dtype=torch.float)
16         return observation
17
18     def observation(self, observation):
19         observation = self.permute_orientation(observation)
20         # since we have updated the observation space attribute, we must

```

```

18     # do so with the pixels from the env, casting them from RGB to
19     # grayscale
20     transform = T.Grayscale()
21     observation = transform(observation)
22     return observation

```

Listing 3.3: Grayscale frame wrapper

3.1.2.3 Resize Observation

The size of an image is a crucial element in neural networks for visual data processing, a larger image is preferable, but it also comes at a cost, specially in the training stage. High resolution images require deeper and bigger models to capture all the spatial features, which results in more parameters to adjust during training. The issue comes with the availability of computational resources, since we do not have a cluster with multiple GPUs at our disposal. To adapt to our computational resources, we have added an additional custom wrapper to our environment that resizes the observation from its original shape, 256×240 to a down-scaled shape of 84×84 . The code for doing so is portrayed in listing 3.4, where we use torchvision's transforms for doing so. Additionally, we normalize the pixel values by using the `T.Normalize(0,255)` transform.

```

1 class ResizeObservation(gym.ObservationWrapper):
2     def __init__(self, env: gym.Env, shape):
3         super().__init__(env)
4         # If shape is just a number it creates a tuple with that same
5         # number (i.e. 84 -> (84,84))
6         # if is a list creates a "tuple" object with the given shape
7         if isinstance(shape, int):
8             self.shape = (shape, shape)
9         else:
10            self.shape = tuple(shape)
11
12         obs_shape = self.shape # + self.observation_space.shape[:2]
13         self.observation_space = Box(low=0, high=255, shape=obs_shape,
14                                       dtype=np.uint8)
15
16     def observation(self, observation):
17         # Use torch transforms to resize the observation
18         # to the wanted resize resolution
19         transforms = T.Compose(
20             [T.Resize(self.shape, antialias=True),
21              T.Normalize(0,255)])
22
23         # apply the transformation
24         observation = transforms(observation).squeeze(0)
25         return observation

```

Listing 3.4: Resize frame wrapper

3.1.2.4 Additional Wrappers

Additionally, there are some libraries such as Stable Baselines 3 [35] or TorchRL [36] that provide methods and classes that encapsulate lots of the functionality provided by wrappers which were useful for our experiments:

- **ClipRewardEnv**: Maps the original reward signal from the environment to the range [-1, 1]. This has multiple benefits, such as preventing larger gradients that may affect learning due to outliers in the reward distribution or encourage exploration, since the algorithm will not always go for high-reward actions, since all of them are within a small range. This is further explored in works such as [37] where they remark the importance of designing an optimal reward signal for the agent to boost its behaviour.
- **EpisodicLifeEnv**: For games where the agent has several lives, treats losing a life as the end of the episode. This was originally implemented by Silver *et al.* in [10] to help the agent with value estimation.
- **MaxAndSkipEnv**: Frame skipping with some truncated behaviour (similar to what we have already explained in section 3.1.2.1).
- **NoopResetEnv**: Once the environment is reset, during a certain amount of steps, the agent performs no actions (**no-operations**) over the environment.

While it may be true that RL is not the field in AI that receives the most attention, little by little, lots of frameworks and libraries that make quite easy train a model are being launched. Both TorchRL and Stable Baselines are quite useful libraries, and since they are PyTorch oriented, they helped us quite a lot in the development of this work.

3.2 Agents development

3.2.1 DQN and DDQN training loop

3.2.1.1 Explaining the DQN algorithm

Once we defined all the pre-processing techniques for interacting with the environments, we proceeded to develop the DQN algorithm as Silver *et al.* defined in [10]. In this section we will first explain the algorithm and the we will briefly explain how we program it using pure python and PyTorch.

The DQN pseudocode is defined in Algorithm 1, where first we define a replay memory. This replay memory is basically a data structure that focus on storing experiences that the agent performs in the environment. Formally, we define as experience the tuple as $(\phi_t, a_t, r_t, \phi_{t+1})$, although when developing the code, we made some minor adjustments for better performance and extended functionality, such as store if the experience resulted in a terminal state. Also, we initialize our action-value function approximator Q with random weights, which we will call the online approximator. From there, we will create our target network (the look-ahead), that will be defined as \hat{Q} and copy the weights θ from Q (the on-line/estimation network) to \hat{Q} . This ensures that the online function approximator and the target function approximator start the training process under equal circumstances, avoiding inaccurate estimation for the state-action values.

Once these initialization steps are done, we begin the training loop, where we iterate for a fixed number of episodes N in the training process. Since we want to extract the most relevant features, we assume that there is a function ϕ that will pre-process the environment, extracting the most relevant features from our input s_t creating the feature vector $\phi(s_t)$. For us, this would be the wrappers explained in section 3.1.2. Then, the episode begin and, for each time step t , up until the episode ends (the agent finds itself in a terminal state) the agent starts playing and learning.

Algorithm 1 DQN with Experience Replay

```

1: Initialize replay memory  $D$  to capacity  $N$ 
2: Initialize action-value function  $Q$  with random weights
3: Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$ 
4: for episode = 1,  $M$  do
5:   Initialize sequence  $s_1 = \{x_1\}$  and pre-processed sequence  $\phi_1 = \phi(s_1)$ 
6:   for t = 1,  $T$  do
7:     With probability  $\epsilon$  select a random action  $a_t$ 
8:     otherwise select  $a_t = \arg \max_a Q(\phi(s_t), a; \theta)$ 
9:     Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
10:    Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
11:    Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$ 
12:    Sample random mini-batch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1}) \sim D$ 
13:    Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j + 1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$ 
14:    Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network
parameters  $\theta$ 
15:    Every  $C$  steps reset  $\hat{Q} = Q$ 
16:  end for
17: end for

```

Both DQN and DDQN follow an ϵ -greedy policy as the behavioural policy π , meaning that with a probability ϵ a random action will be selected instead of the one most rewarding one, according to our Q -value estimation. This encourage exploration, and not falling into a sub-optimal set of Q -values as explained in [38], where the exploration vs. exploitation dilemma is discussed. For our implementation, we will have an initial exploration rate ϵ_0 and a final exploration rate ϵ_f and we designed a scheduling that decays the value of ϵ . Doing so provides an exploration stage at first, where the agent is forced to perform lots of random actions, experimenting and evaluating different situations, thus, increasing the variability of the states that it can learn from.

Once the action is performed, the environment will react to it, providing a reward for such action and a new raw observation x_{t+1} . This will create a transition tuple that will be stored in the replay memory buffer D .

As DQN is an off-policy algorithm, we will learn by performing actions that do not follow the behavioural policy π but the target policy μ . By doing so, we will sample a transition tuple from the replay memory D , and applying the greedy policy, μ obtaining the Q -values obtained as the target y_j for the s_{t+1} (the look-ahead) of the sampled transition tuple. For the online network, we will compute the Q -values for the current state s_t and apply action selection using the greedy policy, obtaining the agent estimations for the current state. As explained in section 2.1.2.1, by minimizing this loss as the TD-error, we are teaching the online network to base the value estimations of the current state s_t using the target network estimations s_{t+1} . This could be seen as a "look-ahead" view that glances into the "future" to obtain more information about how good the current state is. Once we got loss value, we adjust the online network weights θ to minimize the loss using the gradient descent algorithm.

Finally, when a fixed number of steps has passed, we will copy the online network parameters θ onto the target network, so that the distribution of the parameters from the target network do not diverge too much from the online network.

3.2.1.2 Main loop implementation for DQN

While the DQN algorithm seems quite straight forward on paper, the implementation tells other story. For doing so, we implemented several modules to compartmentalize the different functionalities. By making the code modular, the process of improving the code, add new functionalities and connect different modules was easier, creating a pipeline that allowed to test several training set-ups in parallel.

The main script is called `train.py` which basically loads all the hyper-parameters that the code needs, such as the discount factor, γ , for the target estimation or the learning rate when adjusting the net hyper-parameters given the loss. After that, it initializes the agent, that can be of two types, DQN or DDQN to then pass all these arguments into the `Trainer` class. This class implements the main functionality explained in algorithm 1, and a code snippet is presented in listing 3.5. We will go deeper into this class in the following sections, but for now we will only explain the main functionality. First, we reset the environment, and using the done and truncated flags provided by the agent, we can check whether the episode finishes or not. Then, the agent performs an action according to the action-state values over the environment, obtaining a transition tuple which is stored in the replay memory. Finally, the agent learns from the pool of experiences stored in the replay buffer.

```

1 while self.curr_step < self.n_steps:
2     # reset environment
3     done, trunc = False, False
4     state = self.env.reset()
5     #measure_array = []
6     while (not done) and (not trunc):
7         # 1. get action for state
8         action = self.agent.perform_action(state, self.curr_step)
9         # 2. run action in environment
10        next_state, reward, done, trunc, info = self.env.step(action)
11        # 3. collect experience in exp. replay buffer for Q-learning
12        self.agent.store_transition(state, action, reward, next_state,
13                                     done, trunc)
14        # 4. Learn from collected experiences
15        q, loss = self.agent.learn(self.curr_step)
16        # 5. Update the current state
17        state = next_state
18        # 6. Update step value
19        self.curr_step += 1
20        logger.log_step(loss, q)
21
22 if 'episode' in info:
23     # episode field is stored in the info dict if episode ended
24     logger.log_episode(ep_length=info['episode'][1], ep_reward=info
25                         ['episode'][r])
26     if not(self.curr_episode % self.log_every) :
27         logger.record(episode=self.curr_episode,
28                       epsilon=self.agent.exploration_rate,
29                       step=self.curr_step)
30         # log the real reward using episode statistics
31         self.curr_episode += 1

```

Listing 3.5: Trainer main loop

3.2.1.3 Digging deeper: the DQN agent

Once we have seen the training loop, we are going to delve into the agent functionality ([dqn_agent.py](#)).

```

1 @torch.no_grad()
2 def perform_action(self, state, t):
3     # decide whether to exploit or explore
4     if np.random.random() < self.exploration_rate:
5         action = np.random.randint(0, self.action_dim)
6     else:
7         # use of __array__(): https://gymnasium.farama.org/main/\_modules/gymnasium/wrappers/frame\_stack/
8         state = first_if_tuple(state).__array__()
9         state = torch.tensor(state, device=self.device).unsqueeze(0)
10        q_values = self.net(state, model='online')
11        action = torch.argmax(q_values, dim=1).item()
12
13    # decrease exploration_rate according to scheduler
14    self.exploration_rate = self.exp_scheduler.step(t)
15    self.exploration_rate = max(self.exploration_rate_min, self.
16                                exploration_rate)
17
18    return action

```

Listing 3.6: Perform action method from the DQN Agent

As we have seen in the listing 3.5, the first thing that our agent does is execute an action. According to algorithm 1, we perform the action according to the target policy. We can see the code in the listing 3.6 the `perform_action` method when we pick the action that maximizes the action values. After that, we update the exploration scheduler. The exploration scheduler updates the exploration rate ϵ for each time-step with the `step`. We have used three different types of functions to do so: exponential, linear and product of exponentials, which we will talk about in section 3.4.3. After the agent selects the action, we store the transition in the replay memory module of the agent using the `store_transition`. In our case, we opted to use `TensorDictReplayBuffer` module from TorchRL, since it has seamless integration with PyTorch, making it quite easy to use and efficient to run, as we can store the replay memory in the GPU, avoiding for bottlenecks in the training process runtime. Inside, this implementation of the replay buffer stores an array of transition tuples as tensors. When the agent samples from the replay memory, it can be done using mini-batches, making the training process more efficient. In the listing 3.7 we define the replay memory buffer, where we can state the size of the memory in terms of transition tuples, the device where the replay memory is stored, the sampler or if the replay memory has a mapping to disk in case we have limited storage resources. Additionally, we can also define the algorithm to follow in the sampling process. In our case, we opted to use prioritized experience replay sampling [39] which basically prioritizes more significant experiences based on higher temporal-difference errors during training, enhancing learning. The motivation behind this work is similar to what happens in unbalanced datasets, as the less frequent classes usually are the hardest to recognize for the classifier. In order to mitigate this, different techniques ensure to weight more importance sampling to the less frequent instances in the dataset.

```

1 # defining the memory (experience replay) of the agent
2 self.memory = TensorDictReplayBuffer(storage=LazyMemmapStorage(
3     max_size=float(agent_config['replay_memory_size']),
4     scratch_dir='./memmap_dir',

```

```

5     device=self.device,
6 ), sampler=PrioritizedSampler(max_capacity=int(float(agent_config['
7     replay_memory_size'])),
8     alpha=1.0,
9     beta=1.0)
)
```

Listing 3.7: Code snippet for the replay memory of the DQN agent

Storing different experiences along the exploration stage tries to ensure that by the time the agent starts to learn by back-propagation, there is enough data variability to capture enough different situations in the environment to learn to act in most of them. In our implementation, the learning process is done by the `learn` method from the agent class. We present the function in listing 3.8, where first, we have a series of if statements that check several conditions. The first if statement checks that a fixed number of steps has passed in the training to synchronize the online and target network parameters (i.e. $\theta_{online} \rightarrow \theta_{target}$). After that, we check if a fixed number of training steps has passed to save the state of the network periodically. In order to fill the replay memory buffer before starting to learn from it, we define a number of steps where the network does nothing, called burning, that ensures that, by the time the network starts to adjust its weights, there is enough variability of experiences to learn from. Also, for computational resources purposes, we made the network learn only every `learn_every` steps.

```

1 def learn(self, step):
2
3     # Once the error is computed, each sync_every the weights of the
4     # target network are updated to the online network
5     if not (step % self.sync_every):
6         self.sync_Q_target()
7
8     # save the model each save_every steps
9     if not (step % self.save_every) and step > 0:
10        self.save(step)
11
12     # Burning lets episodes pass but collects experiences for the memory
13     # buffer
14     if step < self.burning:
15         return None, None
16
17     # Not learning every step, but every "learn_every" steps
18     if step % self.learn_every != 0:
19         return None, None
20
21     state, action, reward, next_state, done, _ = self.recall()
22
23     # once we have our transition tuple, we apply TD learning over our
24     # DQN and compute the loss
25     q_estimate = self.compute_q_estimate(state, action)
26     q_target = self.compute_q_target(reward, done, next_state)
27
28     # update the q_online network using the loss
29     loss = self.loss_fn(q_target, q_estimate) # Compute Huber loss
30     self.optimizer.zero_grad()
31     loss.backward()
32     self.optimizer.step()

33
34     return q_estimate.mean().item(), loss.item()
```

Listing 3.8: `learn` function from the DQN agent

After these checks, we start with the learning process. To do so, we first recall or sample from the replay buffer hat extracts a set of samples with a fixed batch size (i.e. 8, 16, 32, etc)for the model to learn from them. Thanks to tensor operation support provided from PyTorch, this is forwarded through the online and target networks seamlessly. The `compute_q_target` method implements the TD target element in the DQN loss equation 2.12 which is $r + \gamma \max_{a'} Q(s', a'; w^-)$, while the method `compute_q_estimate` implements the TD estimate which is $Q(s, a; w)$ where actions a and a' are selected according to the target greedy policy for the sampled state s from the buffer and its corresponding next state s' .

With the TD values, we compute the loss. We opted for the Huber Loss [40], defined in equation 3.1 where δ is a regularization parameter. This type of loss is useful in these set-ups, thanks to its robustness to outliers in the data compared to the mean squared error loss function, which is of convenience for this problem in cases where loss between Q-values estimates fall far over the hyper-parameter δ .

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \leq \delta, \\ \delta \cdot (|y - f(x)| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \quad (3.1)$$

Finally, once the loss is computed, we perform the back-propagation process in the online network to adjust its weights and we return the mean Q-values for the sampled mini-batch.

3.2.1.4 From DQN to DDQN

In [11], Van Hasselt et al. proposed an updated version of the DQN algorithm. As explained in section 2.1.2.1, the overestimation bias from the DQN comes from the target evaluating and selecting the actions from the target network. To mitigate this, they propose to decouple the action selection for the TD-target, using the online network and evaluating that action with the target network. We present the pseudocode in Algorithm 2. As we can see, the only difference between Algorithm 1 and Algorithm 2 is the action selection. In Algorithm 2, the action to evaluate is selected by the online Q-network in line 13, while in Algorithm 1, the action is selected by the offline Q-network in line 13. With these changes, we had to adapt the code consequently to implement this algorithm.

Since the training loop is basically the same for DQN and DDQN, we made the DDQN agent class inherit from the DQN agent in the `ddqn_agent.py` script, taking advantage of the already developed code. In listing 3.9 we see the implementation of the `compute_q_target` method for the DDQN agent.

```

1 @torch.no_grad
2 def compute_q_target(self, reward, next_state, done):
3     # for the next state, get the actions that have higher q_values
4     online_q_action_value = self.net(next_state, model='online')
5     max_value_action = torch.argmax(online_q_action_value, dim=1)
6     # then, apply those actions onto the target (off-line) model
7     target_q_action_values = self.net(next_state, model='target')
8     q_next_max_value = target_q_action_values[np.arange(0, self.
9         batch_size), max_value_action]
10    return (reward + (1-done.float()) * self.gamma * q_next_max_value)

```

Listing 3.9: DDQN action selection algorithm

Algorithm 2 Double Deep Q-Learning (DDQN)

```

1: Initialize replay memory  $D$  to capacity  $N$ 
2: Initialize primary action-value function  $Q$  with random weights  $\theta$ 
3: Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$ 
4: for episode = 1,  $M$  do
5:   Initialize state  $s_0$ 
6:   for t = 1,  $T$  do
7:     With probability  $\epsilon$ , select a random action  $a_t$ 
8:     Otherwise, select  $a_t = \arg \max_a Q(s_t, a; \theta)$ 
9:     Execute action  $a_t$  and observe reward  $r_t$  and next state  $s_{t+1}$ 
10:    Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $D$ 
11:    Sample random minibatch of transitions  $(s_j, a_j, r_j, s_{j+1})$  from  $D$ 
12:    Set  $a' = \arg \max_a Q(s_{j+1}, a; \theta)$ 
13:    Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j + 1 \\ r_j + \gamma \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$ 
14:    Perform a gradient descent step on  $(y_j - Q(s_j, a_j; \theta))^2$  with respect to the network
       parameters  $\theta$ 
15:    Every  $C$  steps, update target network  $\theta^- = \theta$ 
16:   end for
17: end for

```

3.3 Q-networks integration

Q-networks are the essential part from DQN and DDQN algorithms. In this work we have trained three different agents using three different Q-networks: CNN, ViT and SWIN. The CNN will serve as our baseline, since it is the most simple out of the proposed networks, and we used the same architecture as in [10], which we will use to see which is the performance that our networks should be up to, and hopefully improve.

We first tried to implement the code for the ViT and SWIN transformer from scratch in order to delve into understanding how these models go from intuition to actual implementation, but we face a big drawback. As we performed some tests, we found out that some existent implementations were already more efficient than ours in computing time. The implementation in our work for the SWIN Transformer was taken from [this](#) repository that is the implementation of [18]. On the other hand, the implementation of the Vision Transformer that we used for our work was taken from [here](#), that is the implementation of [17]. If the reader wants a deeper analysis of the code for both the ViT and the SWIN Transformer, they can refer to sections C.1 and C.2 from appendix C.

To integrate the implementation of these models, we had to enable the Q-network to be an instance of either the CNN, ViT or the SWIN Transformer. This is why we created a class called `DQN_network`, that encapsulates the three possibilities in just one class. The most important part of the code is defined in listing 3.10 and the whole code is in the `dqn_models.py` file. We can see that the `is` argument called `config` that is passed to the constructor. The implementation of this functionality is explained in section 3.4.5.1, but for we will just say that this variable is a dictionary that contain several values and configurations for the Q-networks topology and the DQN and DDQN agents.

```

1  class DQN_network(nn.Module):
2      def __init__(self, type:str, n_actions:int, obs_shape:tuple,

```

```

3     config:dict) -> None:
4         super().__init__()
5         nn_config = config[type]
6         if type == 'vit':
7             self.online = ViT(img_size=obs_shape[-1],
8                               patch_size=int(nn_config['patch_size']),
9                               in_chans=obs_shape[0],
10                              embed_dim=int(nn_config['embed_dim']),
11                              num_heads=int(nn_config['n_heads']),
12                              depth=int(nn_config['n_layers']),
13                              num_classes=n_actions)
14
15         self.target = ViT(img_size=obs_shape[-1],
16                           patch_size=int(nn_config['patch_size']),
17                           in_chans=obs_shape[0],
18                           embed_dim=int(nn_config['embed_dim']),
19                           num_heads=int(nn_config['n_heads']),
20                           depth=int(nn_config['n_layers']),
21                           num_classes=n_actions)
22
23     elif type == 'swin':
24         self.online = SwinTransformer(img_size=obs_shape[-1],
25                                       patch_size=nn_config['patch_size'],
26                                       in_chans=obs_shape[0],
27                                       num_classes=n_actions,
28                                       embed_dim=nn_config['embed_dim'],
29                                       depths=nn_config['depths'],
30                                       num_heads=nn_config['num_heads'],
31                                       window_size=nn_config['window_size'])
32         self.target = SwinTransformer(img_size=obs_shape[-1],
33                                       patch_size=nn_config['patch_size'],
34                                       in_chans=obs_shape[0],
35                                       num_classes=n_actions,
36                                       embed_dim=nn_config['embed_dim'],
37                                       depths=nn_config['depths'],
38                                       num_heads=nn_config['num_heads'],
39                                       window_size=nn_config['window_size'])
40
41     elif type == 'cnn':
42         self.online = CNN(n_actions=n_actions)
43         self.target = CNN(n_actions=n_actions)
44     else:
45         print(f"Type of agent for the model is not correctly
46               specified.\n"
47         f"Select between:\n"
48         f"\t - Visual Transformer (vit)\n"
49         f"\t - SWIN Transformer (swin)\n"
50         f"\t - Convolutional Neural Network (cnn)\n"
51         f"Exiting..."
52     )
53     exit()

```

Listing 3.10: Code for the Q-network initialization

Depending on the `type` value from the argument of the constructor, one type of network will be instantiated as the Q-network of the agent. In listing 3.11 we see a little snippet of the `DQNAgent` constructor, where we can see in line 10 the initialization of the network. Recall from section 3.2.1.4 that even though the implementation is on the `DQNAgent`, this function-

ality is inherited by the `DDQNAgent` class. We remark this, as all the training processes that will be displayed in chapter 4 were performed under the DDQN algorithm, given its superior performance against the DQN algorithm.

```

1  class DQNAgent:
2      def __init__(self, obs_shape:tuple, action_dim: int, device:
3          torch.device, save_net_dir: str,
4          exp_schedule: str, prioritized_replay: bool, agent_config:
5              dict, nn_config:dict):
6
7          ##### ADDITIONAL CODE #####
8
9          # defining the DQN
10         self.net = DQN_network(type=self.type,
11             n_actions=self.action_dim,
12             obs_shape=self.obs_shape,
13             config=nn_config).float()
14
15         self.net = self.net.to(self.device)
16
17         # defining the memory (experience replay) of the agent
18         if prioritized_replay:
19             sampler = PrioritizedSampler(max_capacity=int(float(
20                 agent_config['replay_memory_size'])),
21                 alpha=1.0,
```

Listing 3.11: Part of the constructor method for the `DQNAgent` class

A thing worth mentioning is that we were obliged to train these models completely from scratch, since we could not find pre-trained weights or models in the web. This consumed a lot of our time, as will be explained in section 4.

3.4 Setting up the pipeline

Up until this point, all we have explained in this section is the algorithmic part on how we programmed the agent in order to learn, but that is not the only important thing when developing machine learning algorithms. Our implementation comprises several modules to provide the training framework additional information on how the training process is going, which models work best, and save the weights of the neural networks that help the agent to perform better in the environment. We have already explained in previous sections the main parts which work together to train the different agent configurations that we propose. In this section we will delve into the critical aspects from that have taken part to develop them.

3.4.1 Trainer

The trainer class is kind of the orchestrator of our code. Its main functionality is to coordinate the environment with the agent learning procedure while extracting relevant metrics and evidences of the agent's behaviour. Our environment has different wrappers added to them, that alter the behaviour and the dynamics of the environment, which may be an issue if the code is not properly adapted. In section 3.1.1 we explain the different types of wrappers that we used, but we have not explained the implications that they have in the

way the agent behaves and the metric extraction. For example, our environment uses the `EpisodicLifeEnvironment` which basically treats each life of a game as an episode from the agent to learn, which helps the agent to improve its estimates of the rewards. The issue here is that when we extract the real statistics of the episode they come altered. To solve these problems we make use of the wrapper `RecordEpisodeStatistics` which collects the actual real statistics from the environment. This wrapper adds additional information to the environment when the episode has ended. In listing 3.12, that is taken from listing 3.5, we can see that in the info dictionary, that is returned for each `env.step(action)` call, there is a key called `episode` that opens up an embedded dictionary with information that is specific from the episode, such as the accumulated reward or the length of the episode which we extracted and saved for logging purposes. Additional information can be extracted too, such as the last frame that the agent saw.

```

1 if 'episode' in info:
2     # episode field is stored in the info dict if episode ended
3     logger.log_episode(ep_length=info['episode']['l'], ep_reward=info['
4         episode']['r'])
5     if not(self.curr_episode % self.log_every) :
6         logger.record(episode=self.curr_episode,
7             epsilon=self.agent.exploration_rate,
8             step=self.curr_step)
9             # log the real reward using episode statistics
10            self.curr_episode += 1

```

Listing 3.12: Episodic information retrieval from `RecordEpisodeStatistics` wrapper

3.4.2 Logger

To record how well an agent is performing, a main performance indicator needs to be tracked: the reward. The issue is that, sometimes, we may encounter environments where the reward gets progressively sparser, occluding information about how well the agent understands the scene and the actions it may perform. As stated in chapter 3, one of the games we are going to develop our evaluation is the MsPacman game. There, rewards are very common at the beginning, but as the game goes on, they become increasingly difficult to obtain. If we do not take this into account, it may seem like the agent is not learning, which may not be true. That is why, additional indicators such as the episode length, the agent's loss or the mean Q-values can be of use to have a better understanding on whether the agent is learning or not. To comprise all this information, we have developed the logger module (`logger.py`). We must say that for this module, we took inspiration from [41].

The constructor of the logger class (`MetricLogger`), is defined in listing 3.13, where we define the main attributes that hold the relevant information that we want to monitor. First, we indicate to the logger if the agent to log is pre-trained and we are continuing the training or performing a fine-tuning or it is a fresh new training thanks to the `load_pre` flag. If so, we assume that in the `save_dir` path there is already information about a previous logged training, if not the a new log file is created under the indicated directory. Additionally, we define the episodic arrays, that hold the information for each episode and the smoothed episodic arrays, that take the moving average of the metrics extracted from the last 100 episodes. After that, we call the `self.init_episode` to indicate the logger that we are going to start logging a new episode.

```

1 class MetricLogger:

```

```

1 def __init__(self, save_dir:Path, agent_type:str, load_pre=False):
2
3     # paths for saving logs and plots
4     self.save_log = save_dir / "log"
5     if not load_pre:
6         # create if not exists
7         save_dir.mkdir(parents=True)
8         with open(self.save_log, "w") as f:
9             f.write(
10                 f"{'Episode':>8}{'Step':>11}{Epsilon':>10}{MeanReward
11                  :>15}"
12                 f"{'MeanLength':>15}{MeanLoss':>15}{MeanQValue':>15}"
13                 f"{'TimeDelta':>15}{Time':>20}\n"
14             )
15     else:
16         self.load_from_log(save_dir)
17         self.save_dir = save_dir
18         self.agent_type = agent_type
19
20     # History metrics
21     self.ep_rewards = []
22     self.ep_lengths = []
23     self.ep_avg_losses = []
24     self.ep_avg_qs = []
25
26     # Current episode metric
27     self.init_episode()
28
29     # Timing
30     self.record_time = time.time()
31
32     def init_episode(self):
33         self.curr_ep_loss = 0.0
34         self.curr_ep_q = 0.0
35         self.curr_ep_loss_length = 0

```

Listing 3.13: MetricLogger class initialization

As our agent starts to perform actions, metrics will start to pop up. In order to monitor in the step and in the episode domain, we have two functions `log_step` and `log_episode`, defined in listing 3.14, that compute the step information and record it in the episodic arrays defined in the constructor of the class. This information is stored from the beginning of the training up until the very end, but is not yet dumped into the log file. One thing to notice is that, for the `log_step` function, we only accumulate the values if a loss is provided. If we recall the training procedure, the agent does not learn every step, but every fixed number of steps which helps with the computational over-head. When the agent is not learning, the loss is returned as a `NoneType` object, hence, not recording the step metrics in the logger.

```

1 def log_step(self, loss, q):
2     "Mark end of step if loss is provided"
3     if loss:
4         self.curr_ep_loss += loss
5         self.curr_ep_q += q
6         self.curr_ep_loss_length += 1
7
8     def log_episode(self, ep_reward, ep_length):
9         "Mark end of episode"
10        self.ep_rewards.append(ep_reward)

```

```

11     self.ep_lengths.append(ep_length)
12
13     if self.curr_ep_loss_length == 0:
14         ep_avg_loss = 0
15         ep_avg_q = 0
16     else:
17         ep_avg_loss = np.round(self.curr_ep_loss / self.
18             curr_ep_loss_length, 5)
19         ep_avg_q = np.round(self.curr_ep_q / self.curr_ep_loss_length, 5)
20         self.ep_avg_losses.append(ep_avg_loss)
21         self.ep_avg_qs.append(ep_avg_q)
22
23     self.init_episode()

```

Listing 3.14: Step and episode functions for metrics extraction

To dump the information into the log file, we defined the `record` function, showed in listing 3.15. First, for all metrics, we compute the moving average of a window of 100 episodic values. This is because usually, training procedures in reinforcement learning (specially in TD learning) holds lots of variability, which sometimes result in noisy results. After that, we print via command line the statistics, for hands-on visualization. At last, we open the log file, and write the computed metrics, to then read it using a `pandas` data-frame, and plot the values using the `plotly` library, which has great integration with pandas data-frames for data plotting.

```

1 def record(self, episode, epsilon, step):
2     mean_ep_reward = np.round(np.mean(self.ep_rewards[-100:]), 3)
3     mean_ep_length = np.round(np.mean(self.ep_lengths[-100:]), 3)
4     mean_ep_loss = np.round(np.mean(self.ep_avg_losses[-100:]), 3)
5     mean_ep_q = np.round(np.mean(self.ep_avg_qs[-100:]), 3)
6     self.steps.append(step)
7
8     last_record_time = self.record_time
9     self.record_time = time.time()
10    time_since_last_record = np.round(self.record_time - last_record_time,
11        3)
12
13    with open(self.save_log, "a") as f:
14        f.write(
15            f"{{episode:8d}{step:10d}{epsilon:10.3f}}"
16            f"{{mean_ep_reward:15.3f}{mean_ep_length:15.3f}{mean_ep_loss:15.3f"
17            f"    }{{mean_ep_q:15.3f}}"
18            f"{{time_since_last_record:15.3f}}"
19            f"{{datetime.datetime.now().strftime('%Y-%m-%dT%H:%M:%S'):>20}\n"
20        )
21    df = pd.read_csv(self.save_log, header=0, sep='\s+', skipinitialspace
22        =True)
23    for i in df.columns:
24        if i not in NOT_2_PLOT:
25            fig = px.line(df, x = 'Step', y = i, title=f'{i} over time'
26                           for {self.agent_type})
27            fig.write_image(self.save_dir / f"{i}.png")

```

Listing 3.15: Record function, that saves the moving average statistics in the log file and plots the values of the statistics over the time-steps of the learning

The logger is one of the most useful tools that we have developed, since it can manage to monitor the training procedure seamlessly, even when several agents are being trained in

parallel and allow us to better visualize the development of the agent's learning.

3.4.3 Schedulers

In section 3.2.1.3, we talked about how we have proposed three different exploration schemes for the ϵ value: linear decay, exponential decay and product of exponential decay. To formalize this, we proposed the equations 3.2, 3.3 and 3.4 for each type of schedule, ϵ_t is the current exploration rate for a time-step t , ϵ_0 is the initial exploration rate, ϵ_f is the final exploration rate, N is the number of steps of exploration, λ is an hyper-parameter and γ is a parameter that we need to infer, given the upper bound ϵ_0 and the lower bound ϵ_f .

$$\epsilon_t = \frac{\epsilon_f - \epsilon_0}{N} \times t + \epsilon_0 \quad (3.2)$$

$$\epsilon_t = \epsilon_0 \gamma_{exp}^t \quad (3.3)$$

$$\epsilon_t = \epsilon_0 \prod_{i=0}^t \gamma_{prod}^{\lambda i} \quad (3.4)$$

The γ_{exp} value for the exponent decay function is defined in 3.5, where λ is an hyper-parameter, and the rest of the terms involved are the same as in equation 3.3. On the other hand, the γ_{prod} value for the product of exponents function is defined in 3.6, where λ is an hyper-parameter and k is defined in equation 3.7 as the general sum of the N first values, where n is the final value, a_1 is the first value and d is the interval between the numbers of the sum, in this case 1.

$$\gamma_{exp} = \sqrt[\lambda N]{\frac{\epsilon_f}{\epsilon_0}} \quad (3.5)$$

$$\gamma_{prod} = \sqrt[\lambda k]{\frac{\epsilon_f}{\epsilon_0}} \quad (3.6)$$

$$k = \sum_{i=0}^N i = na_1 + d \frac{(n-1)n}{2} \quad (3.7)$$

To clarify that the γ -values used in these equations is completely different from the discount factor hyper-parameter used in the DQN learning equations defined in 2.12 and 2.13. Additionally, the way we came up with them is shown in appendix D. The main goal for defining equation 3.4 is to force the exploration rate to be big in the first stages, and decrease with a lower rate than the exponential decay defined in equation 3.3. Figure 3.2 visually shows what we have already explained.

To properly scheme the different schedulers, we first defined an abstract class, as seen in listing 3.16. This class has three main values, the upper bound `e_0`, the lower bound `e_f` and the number of time-steps for the function to go from the upper bound to the lower one. The `step` method will return the current exploration rate, given the current time-step.

```

1 class Scheduler(ABC):
2     def __init__(self, e_0, e_f, n_steps) -> None:
3         self.e_0 = e_0
4         self.e_f = e_f

```

```

5     self.n_steps = n_steps
6     @abstractmethod
7     def step(self, t):
8         pass

```

Listing 3.16: Abstract class for the scheduler

The implementation of the linear schedule in listing 3.17 is pretty straight-forward. For each step, we pass the number of the time-step value, which would be the t value for 3.2 and 3.3 equations and once we have computed the slope, we use as the y-intercept the upper bound e_0 .

```

1 class LinearScheduler(Scheduler):
2     def __init__(self, e_0, e_f, n_steps) -> None:
3         super().__init__(e_0, e_f, n_steps)
4         self.slope = (e_f - e_0)/n_steps
5
6     def step(self, t):
7         e_t = self.slope*t + self.e_0
8         return max(e_t, self.e_f)

```

Listing 3.17: Linear scheduler class

The exponential decay schedule, displayed in listing 3.18, is quite similar to the linear schedule. Instead of computing the slope, we compute the γ value according to equation 3.5, but the rest is practically the same.

```

1 class ExpDecayScheduler(Scheduler):
2     def __init__(self, e_0, e_f, n_steps) -> None:
3         super().__init__(e_0, e_f, n_steps)
4         self.gamma = (e_f / e_0)**(1/n_steps)
5
6     def step(self, t):
7         e_t = self.e_0 * self.gamma**t
8         return max(e_t, self.e_f)

```

Listing 3.18: Exponential scheduler class

Finally, for the product of exponential decay schedule, which is the implementation of equation 3.4, displayed in listing 3.19, we had to compute the exploration rate in an efficient way to overcome efficiency problems. To do so, instead of looping through the multiplicative product, which would be computationally expensive, we defined an accumulator that held the values for each iteration, so that, when a new time-step comes in, we only have to multiply the accumulator by the exponent of the gamma to the time-step.

```

1 def gauss_sum(a:int, n:int, inc:float):
2     return n/2*(2*a+(n-1)*inc)
3
4 class ProdExpDecayScheduler(Scheduler):
5     def __init__(self, e_0, e_f, n_steps) -> None:
6         super().__init__(e_0, e_f, n_steps)
7
8         k = gauss_sum(a=0, n=n_steps, inc=1.0)
9         self.gamma = (e_f / e_0)**(1/k)
10        self.acc = e_0
11
12    def step(self, t):
13        self.acc *= self.gamma**t
14        return max(self.acc, self.e_f)

```

Listing 3.19: Product of exponentials decay schedule

To have a better understanding on how these equations behave over time, the shape of them is depicted in Figure 3.2, where γ is the value computed according to the bounds of our exploration schedule. We came up with these different schedules to explore if they affect the exploration process when the agent is learning over the first steps. We will discuss more about this in Section 4.

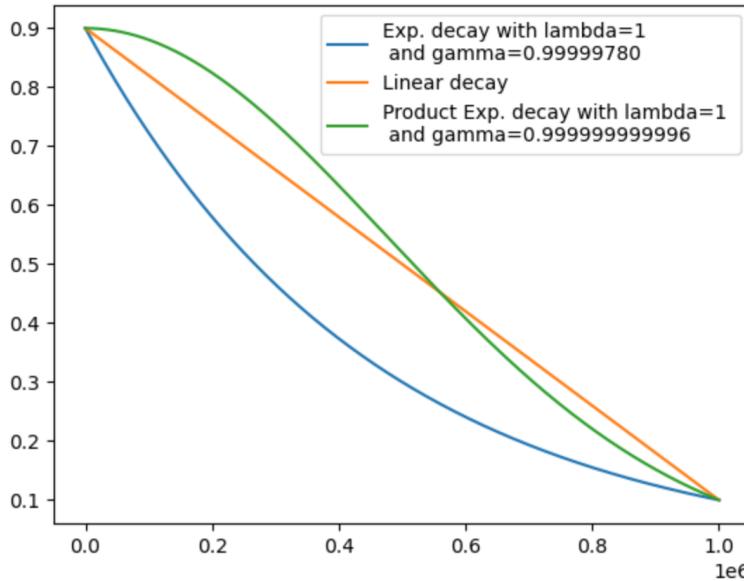


Figure 3.2: Shape of the proposed exploration functions for the DQN agent

3.4.4 Extracting explainability features

In this section we are going to explain how, once we trained our models, we extracted several feature maps to understand what the model is taking into account to perform an action. We must note that for the SWIN Transformer, in section 3.4.4.1, we could not implement a way to extract raw attention maps, nor we found any reference in the literature that would help us with this purpose, so we only extracted the attention maps for the ViT model. For extracting the activation maps using established methods in the XAI field in section 3.4.4.2, the developed method takes into account both ViT and SWIN transformer.

3.4.4.1 Extracting attention maps

As explained in section 2.2.3, the ViT introduces an additional token along the patches, called the class token, which holds the information that class takes into account via self-attention to produce the output. Figure 3.3 illustrates this. For a 4 by 4 patched image, the disposition of the attention matrix for one attention head would give us for the **Class Token** row, all the attention weights with respect to the rest of the patches. The code developed for this section is available in the script [visualize_attention_vit.py](#).

In order to extract the attention maps, the implementation of the ViT used from [42] has the functionality of returning the attention maps in the forward pass of the ViT blocks

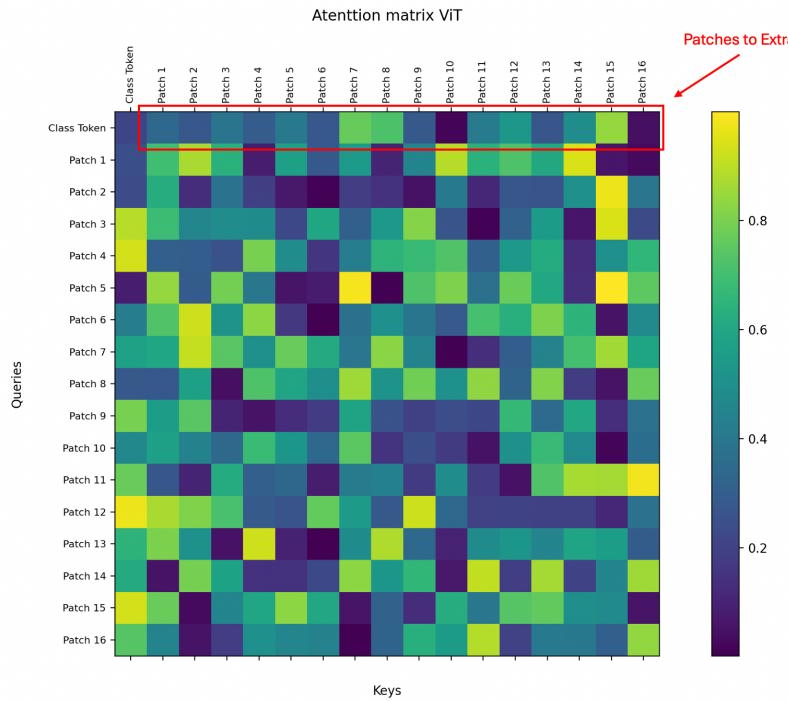


Figure 3.3: Example of an attention matrix for one attention head with 4×4 patches. The section under the red rectangle refers to the patches that the class token ponders in order to perform the output. These patches will be our extraction for the attention maps.

(see listing C.4). The only thing we ought to do, is index the attention matrix correctly to extract the patches of attention with respect to the class token (patches that fall under the red rectangle in Figure 3.3). In listing 3.20, we can see an example of this. First, we obtain the attention weights for each patch by indexing the attention matrix like `attn[0, :, 0, 1:]` in line 3, where the first dimension is de batch, the second is the attention heads, the third would be rows and the fourth the columns. This indexing returns a matrix with dimensions $\mathbb{R}^{nh \times (P-1)}$, where nh refers to the number of heads from the attention mechanism, and P refers to the tokens that go into the attention mechanism. Since we are not interested in the attention weight that the class token has with respect to itself, we exclude it out, hence the `:1` in the indexing for the fourth dimension. To compute the averaged importance of each patch, we perform the mean across the nh dimension, and reshape the P dimension to the original spatial dimensions P_w and P_h for the width and the height respectively (line 4).

Then, by calling the method `interpolate_attn_map` we perform a interpolation to return the attention map to the original size of the input (lines 5 to 8).

```

1 vit = agent.net.online
2 attn = vit.get_last_selfattention(state.float())
3 cls_attn = attn[0, :, 0, 1:]
4 cls_attn = cls_attn.reshape(-1, dimension, dimension).mean(dim=0,
   keepdim=True).unsqueeze(0)
# 3. Interpolate the attention maps to original frame size
5 cls_attn_int = interpolate_attn_map(attribution_map=cls_attn,
6 scale_factor=patch_size,
7 final_shape=original_shape)
8

```

Listing 3.20: Attention map extractor for a single frame

Once we have done this, we can plot the attention map along with the original frame to

observe where does the model pay attention to, given a frame. In Figure 3.4 we provide an example for this. Additionally, in the right side of the figure, we provide the returned Q-values from the network, since they are of vital importance for understanding the network’s decisions. The interpretations that we extract from this will be discussed in chapter 4, along with some additional insights for each attention head.

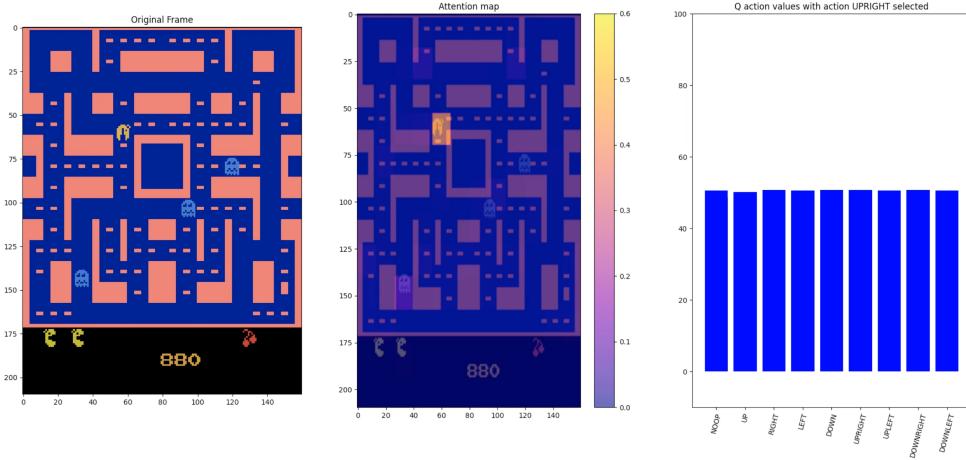


Figure 3.4: Left side, the original frame from the game, in the center, the attention maps for the class token, at the right-side, the Q-values for the observation from the state.

3.4.4.2 Extracting Grad-CAM’s activation maps

We developed something similar to section 3.4.4.1 but using the grad-CAM technique explained in 2.3.3. To do so, we used the [pytorch-cam](#) library [43] that integrates a well varied amount of CAM methods to select. Part of the developed code is displayed in listing 3.21, and the complete code can be found in the [visualize_activations.py](#) script.

```

1  action, q_v = agent.perform_action(state, t=-1, exploit=True)
2  actions.append(action)
3  q_values.append(q_v.squeeze().detach().cpu().numpy())
4  # 2. get the activation maps
5  state = first_if_tuple(state).__array__()
6  state = torch.tensor(np.array([state]), device=device)
7  action_target = ClassifierOutputTarget(action)
8  gray_scale_cam = cam(input_tensor=state, targets=[action_target])
9  gray_scale_cam = torch.tensor(gray_scale_cam[0, :]).unsqueeze(0).
10   unsqueeze(0) # appropriate dims for interpolation
11  activ_map = interpolate_activations(gray_scale_cam,
final_shape=original_shape)

```

Listing 3.21: Activation maps using Grad-CAM

To adapt Grad-CAM from the classification supervised learning set-up to the reinforcement learning set-up, we treated each of the actions the agent selected as the predicted “class”. By doing so, we expected to obtain the biggest activations from the input that the agent considers to perform the output. The [ClassifierOutputTarget](#) method takes the target class (our action in this case), and passes it to the Grad-CAM method (cam), that returns the activations. Then we perform an interpolation to return the resolution to the original frame size, obtaining a map depicted in Figure 3.5. We must say that this method works

for both ViT and SWIN backbones, along with other non-attention based models, such as CNNs.

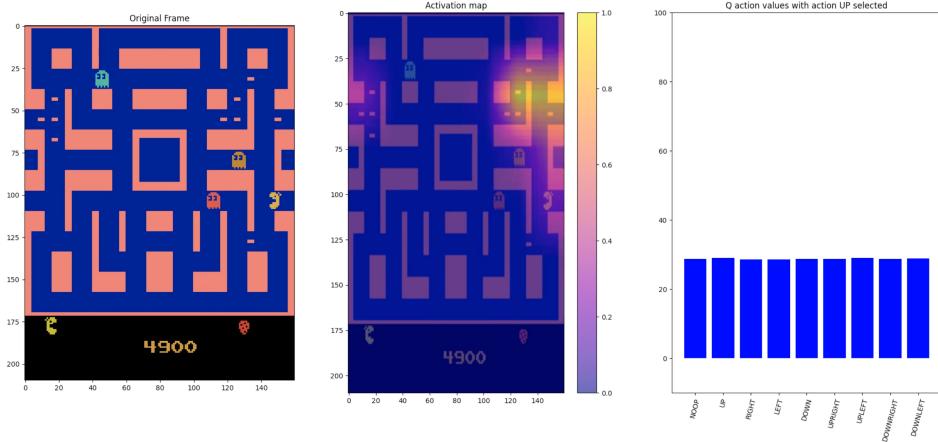


Figure 3.5: Left-side, the current frame of the game. In the center, the activation maps from the ViT using grad-CAM. At the right-side, the Q-values provided by the neural network.

3.4.5 Additional implementations

3.4.5.1 Configuration files

As explained in sections C.1 and C.2, the implementation of these models is highly parametrizable, since we can state lots of parameters that are involved in the topology of the models. Changing these parameters manually in each file is a burden, since we have to check that the changes are done correctly in the several files involved in running the training process. To tackle this issue, we proposed two files: one for the DQN agent hyper-parameters, and another for the Q-networks hyper-parameters called `agents_config.yaml` and `agent_nns.yaml`. For loading this files, we used the `pyyaml` library, that eases a lot the loading and manipulation of yaml files, and used them to properly initialize the corresponding hyper-parameters, such as number of steps for training, the learning rate for the back-propagation and the Q-estimates or the number of layers that our model will have. Thanks to this implementation, we were able to seamlessly change the configuration of the agents, allowing for room to explore different hyper-parameters configurations.

3.4.5.2 Checkpoints

As the training process progresses, the weights of our function approximators will change. There are some cases where the excess of training produces a degradation on the model's performance, ending the training with a sub-optimal model. To change this, we implemented a checkpoint saver in our pipeline, that saves periodically the model's state dictionary, that contains the weights of the model. By doing a snap-shot of the model weights, we have several versions, which will be of use when we run our evaluations of the agents in the environment. Additionally, the checkpoints are of great use if we want to resume a training process. We have also developed adaptations to our code that perform these kinds of operations, and have been of great use when unexpected events (such as lack of computational resources) stopped our training process.

3.4.6 Overview

With almost all the components involved in the training pipeline for our agents, in Figure 3.6 we present the diagram of all the modules developed along with their relationships.

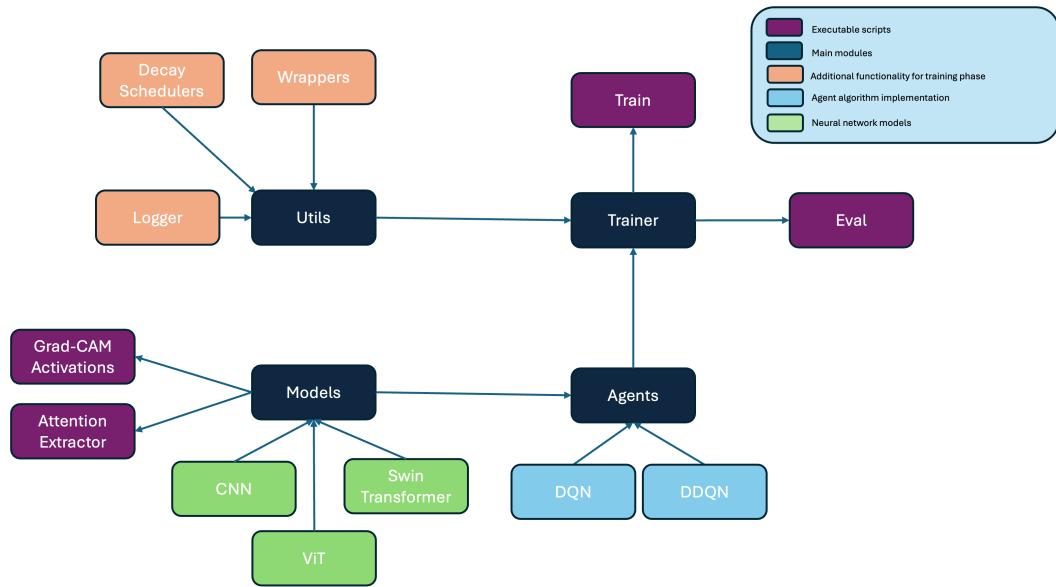


Figure 3.6: Implementation overview of all the modules involved in this work

Chapter 4

Evaluation

In this chapter we will delve into the results from the several trainings we performed with the agents. We will deeply explain the results and discuss them to try to extract insightful knowledge from it. First we will discuss the several models, and computers we have used along this work. We will also compare the models in term of computational complexity, giving a deeper sight on the training times considered in this work. Finally we will go over the explainability results that we have obtained, and discuss if they are insightful in terms of understanding whether the attention-models explain their own behaviour with the attention maps and the explainability techniques already discussed.

4.1 Experimental set-up

4.1.1 Computing resources

To carry out our experiments, we used several computers with several GPUs to meet the time constrains that this work imposed. According to several works [10, 44, 39, 4], the usual number of training steps for DQN learning algorithms is between 50 million and 200 million steps. We could not achieve this for any of our training procedures. Even though the GPUs we used for our training were powerful, they cannot compare to the resources that companies such as Google DeepMind has. For the development of this work, we used 4 different computers:

1. Computer 1 - Personal computer with the following specs:
 - CPU: AMD Ryzen 5600X.
 - GPU: 1 × NVIDIA RTX 3060 (12 GB of GDDR6 VRAM).
 - RAM: 32 GB of DDR4.
2. Computer 2 - Universidad de Alcalá with the following specs:
 - CPU: AMD Ryzen 9 5950X.
 - GPU: 2 × NVIDIA RTX 3080 (10 GB of GDDR6X VRAM).
 - RAM: 32 GB of DDR4.
3. Computer 3 - Artemisa cluster (Universidad de Valencia and Universidad de Alcalá) with the following specs (we only specify the resources we were allowed to use):

- CPU: Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
 - GPU: 1 × NVIDIA V100 (40GB of HBM2e)
 - RAM: 32 GB of DDR4
4. Computer 4 - Anomalia cluster (Universidad Autónoma de Madrid) with the following specs (we only specify the resources we were allowed to use):
- CPU: Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz.
 - GPU: 3 × NVIDIA RTX 3090 (24 GB of GDDR6X VRAM).
 - RAM: 196 GB of DDR4.

The use of this systems made possible to accelerate the training speeds, but we could not get their full potential such as training parallelization. We tried to implement a training procedure similar to [45] where they implement a library to accelerate inference times and sampling from the experience replay buffer across GPUs using MPI [46] in python using PyTorch. We tried to do something similar in our implementation, but could not make it to work.

4.1.2 Training set-up

For the training procedure, we run the models under just one constrain: to end the training if for a portion of the total trained steps until that time, no improvement is done. We were aware that for a lot of training set-ups, the model may face a streak where does not improve for a long period of time, and then, out of the blue, starts to pick up knowledge and exploits it obtaining great scores down the line. We could not afford to do this, given our time constraints, and "limited" resources. Referring to the hyper-parameters used in the training procedures, we fixed the ones that were related to the agent training, such as the γ , the learning rate, the batch size or the initial and final exploration rates (see Table 4.1). We have seen that in several works, such as [11, 44, 4], they implement the same thing for the training procedure.

Table 4.1: Fixed hyper-parameters across different training procedures.

Input	$84 \times 84 \times 4$
Optimizer	Adam
Adam Learning Rate	6.25×10^{-5}
γ	0.99
Initial ϵ	1
Final ϵ	0.01
ϵ -decay steps (frames)	1×10^6
SyncFrames	4×10^4
SkipLearningFrames	4
Replay Buffer Size	1×10^6
Batch Size	32

4.1.3 Q-networks configuration

For each of the Q-networks, the hyper-parameters related to the topology of the networks varies in terms of layers, and the number of weights. Additionally, we also trained a series of models using a CNN similar to the one used in [11] to obtain a “baseline” on the performance and perform different experiments in the exploration schedules, since their computational complexity is inferior to the attention-based models, and given the results, we selected the exploration schedule accordingly.

The hyper-parameters related to the models are defined in tables 4.3, 4.4 and 4.5. In the case of the SWIN DDQN network, we copied the same topology as the one proposed in [4]. In the case of the ViT DDQN network, we tried to balance between the number of weights from the model and the computation complexity of the forward and backward pass, given the quadratic computational complexity from the model. We designed two configurations: ViT-Big and Vit-Small, but because of time constrains, we only could train the ViT-Small configuration.

Additionally, since to the best of our knowledge, we have not seen any work that explores the ViT as the backbone for a value function approximation in DQN-learning, and for the SWIN Transformer, we have stated that there are some works that explore this set-up, but no pre-trained weights where available, which meant we did not have a baseline to start from when training the agents.

Table 4.2: CNN DDQN network hyper-parameters

Layers	3
Filters per layer	32, 64, 64
Number of heads per block	6
Linear dimension after flatten	512
Activation function	ReLU

Table 4.3: ViT Small DDQN network hyper-parameters

Layers	5
Blocks per layer	1
Number of heads per block	6
Embedding dimension	156
Patch size	7
Dropout rate	0.1

Additionally, we state in Table 4.6 the number of weights each backbone that we trained our agent on had. We used the [torchinfo](#) [47] library to obtain these estimations. We state the floating operations per second (FLOPs) for the proposed models using the [ptflops](#) library [48]. The code where we compute the values is available in scripts [size_calculator.py](#) and [flops_calculator.py](#). It is clear that the CNN is the model with less computational complexity, since it has far less FLOPs than the rest of the networks. For the ViT-Small we see that the complexity increases significantly with respect to the number of parameters, as it has roughly the same of parameters as the CNN, but far more FLOPs due to the attention mechanism. The SWIN transformer is by far the most complex model in terms of parameters,

Table 4.4: ViT Big DDQN network hyper-parameters

Layers	5
Blocks per layer	1
Number of heads per block	8
Embedding dimension	208
Patch size	5
Dropout rate	0.1

Table 4.5: SWIN DDQN network hyper-parameters

Layers/Stages	3
Blocks per layer	2,3,2
Heads per layer	3,3,6
Patch Size	3×3
Window Size	7×7
Embedding dimension	96
Dropout rate	0.2

with almost 3 times the number of parameters from the ViT-Small and CNN, and almost 72 times more computational complexity in terms of FLOPs than the CNN. Additionally, the ViT-Big model has only 1.5 more parameters than the CNN, but more FLOPS than the SWIN Transformer (almost 75 times more). With this comparison we try to reflect the great computational cost that comes with the attention-based models in the deep learning paradigm.

Table 4.6: Number of parameters and FLOPs per model per forward pass.(Note: We put the scales in orders of megas and gigas (i.e. 10^6 , 10^9) for a better understanding of the units)

Q-network	Number of Parameters	MACs	FLOPs
CNN	1.69×10^6	9.41×10^6	18.42×10^6
VIT Small	1.52×10^6	218.07×10^6	436.14×10^6
VIT Big	2.68×10^6	677.05×10^6	1.354×10^9
SWIN	5.49×10^6	644.92×10^6	1.289×10^9

4.1.4 Environments

Pac-man is an widely known video-game launched for the Atari 2600 console back in the eighties. In the game, we take part as Pac-Man, a pie-shaped creature that moves around a maze. The main goal of the game is to stay alive by eating the dotted rewards, called **tiles** around the maze while avoiding the coloured ghosts: Blinky, Pinky, Inky, and Clyde. Scattered throughout the maze, there are a set of rectangles that have the power of turning the coloured ghosts to blue for a short period amount of time. When this happens, Pac-Man can eat the ghosts, and win additional points. When we eat all the dots from the maze, we level-up, to more complex mazes when we have to repeat this process. The player is

limited to 8 moves: UP, DOWN, LEFT, RIGHT, UP-LEFT, UP-RIGHT, DOWN-LEFT and DOWN-RIGHT. In Figure 4.1a we show a screenshot of the game.

Demon Attack is also a Atari game where players battle against the enemies in the form of demons. Each wave of demons presents a new challenge, growing more difficult and aggressive as the game progresses. Some of these alien adversaries even split into smaller demons when struck, adding an extra layer of complexity to the gameplay. The game's controls are simple: SHOOT, LEFT, RIGHT. The main goal is to survive while shooting and killing the demons to obtain higher rewards. In Figure 4.1b we show a screenshot of the game.

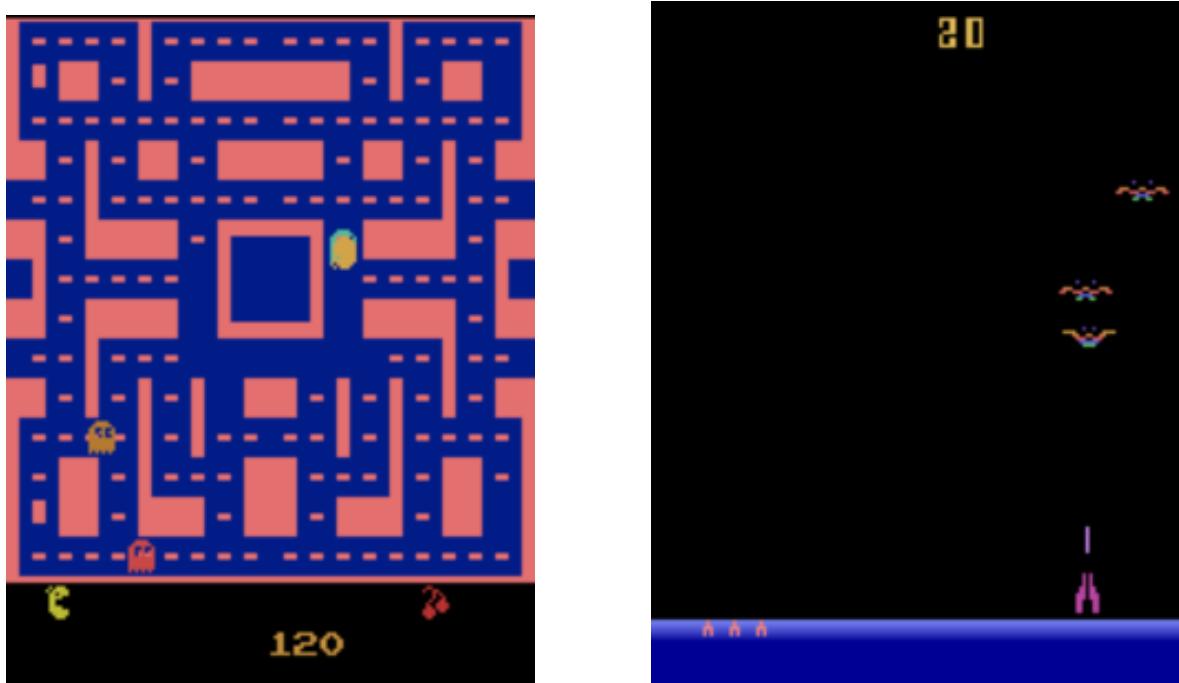


Figure 4.1: In figure 4.1a (left) we see the Pacman environment. In figure 4.1b (right) we see the Demon Attack environment.

4.2 Training results

4.2.1 Exploration schedules

In section 3.4.3 we introduced three different types of scheduling: exponential decay, linear decay and product of exponents decay. To know which schedule we had to use for our final training procedures, we had to test them out. To do so, we used the CNN models as our test model for trying the different schedules. This decision was taken mainly because of time constrains, as we have seen in Table 4.6 that CNNs are the less computational expensive model. We trained all the models in the computer 2 (mentioned in section 4.1) for 1.2×10^7 steps using the DDQN algorithm and checked which one got better results according to the experimental set-up defined in Table 4.1. The environment where we made these tests was **MsPacman-v5**. We are aware that this may not be the best way to test these results, but given the amount of time that we had at our disposal, it was the only plausible way to test

different schedules and take a decision on which one was better for our final results. The comparison of the evolution can be seen in Figure 4.2.

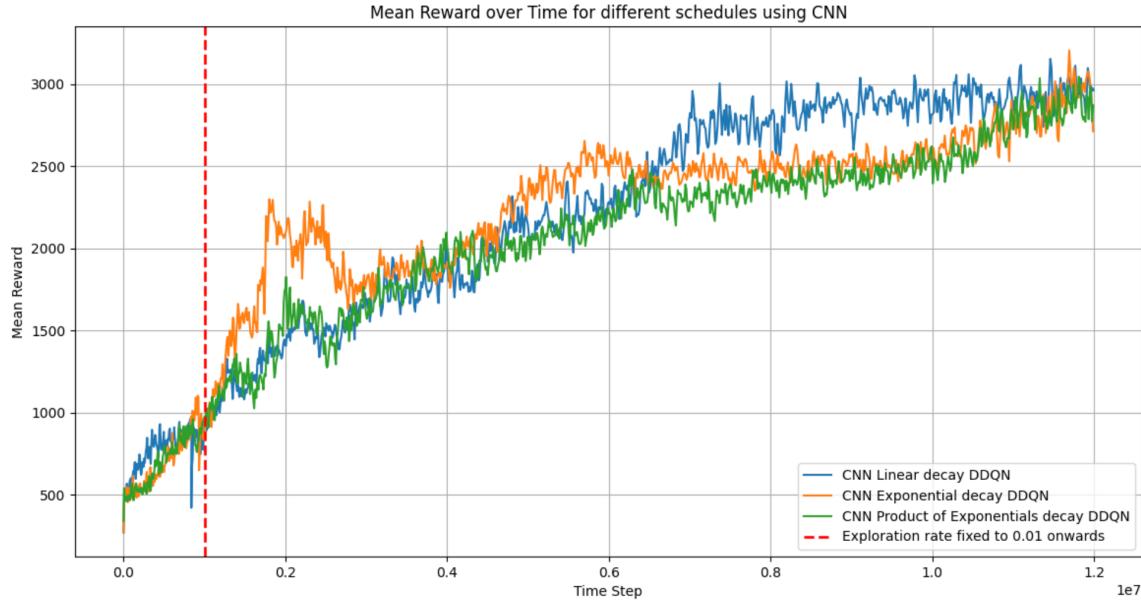


Figure 4.2: Comparison between different schedules for a DDQN CNN agent for 1.2×10^7 steps.

We can see that there is no better performing schedule up until that point. For us, this was it bit strange, since we expected that the linear and power of exponentials schedules would come up top, since they had higher exploration rate for a longer time than the exponential schedule. It was in this point where we decided to opt for the product of exponentials for the rest of the training procedures, since up until the 12 million steps mark, it was the schedule with the greater growth in the last 6 million steps, even if it was marginal. The training time to reach for the 12 million steps mark was around 1 days and 12 hours each approximately. We kept training the agents with these three different schedules and, after almost 3 days and 20 hours of training we found a counter-intuitive surprise in the 25 million steps mark, as can be seen in Figure 4.3, where the exponential decay was actually the best performing schedule from all of them. We say "counter-intuitive" because we held onto the intuition that since the linear schedule and the product of exponentials had more exploration rate for a bigger amount of time, it would help the agent to collect a wider variability of experiences and perform better estimations. If we would have more time to test the proposed schedules, trying different runs and testing which schedule came up on top more consistently, maybe we could have taken a more insightful decision on which schedule select.

4.2.2 Agents training comparison

As stated in section 4.2.1, in the 12 million mark from the comparison training, we selected the product of exponentials as our exploration schedule for **all** the training procedures. We were on a hurry to launch all of our experiments for both MsPacman-v5 and DemonAttack-v5 environments, since we knew the computational cost the proposed Q-networks had. We selected to train our agents in two different environments to compare how the explainability mechanisms behaved in different environments, and if an interpretable set-up was achieved

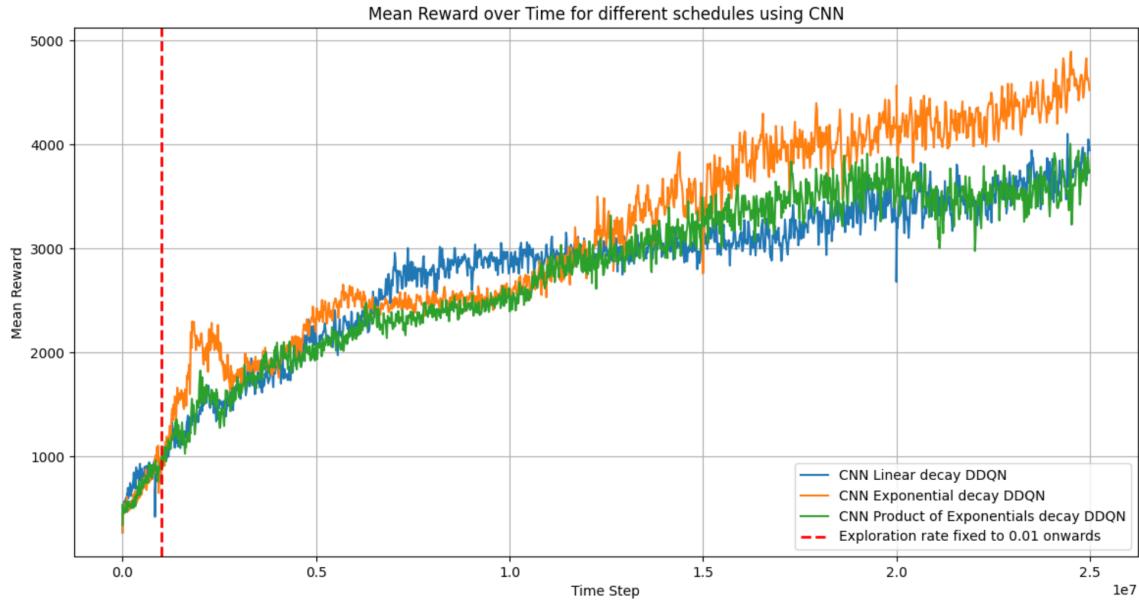


Figure 4.3: Comparison between different schedules for a DDQN CNN agent for 2.5×10^7 steps.

in both of them. In the following sections we will briefly introduce the rules of the games and show the obtained results in both training and evaluation phases.

4.2.2.1 MsPacman training results

For the this environment, we launched the three training procedures and let them train as much as we could. The training time for all of the models went over 6 days, and the training was run in Computer 4. The results are portrayed in Figure 4.4. For the ViT agent, we trained a total of 4.5×10^7 steps with a maximum reward of 5811. For the SWIN agent, we trained a total of 3×10^7 steps with a maximum reward of 5637. For the CNN agent, we trained a total of 6×10^7 steps, with a maximum reward of 5369. The red dashed line represents the point where the exploration schedule finished, remaining the exploration rate in a fixed value of $\epsilon=0.01$.

It is pretty clear that the CNN agent was the one with more time-steps to train, with over 6.5×10^7 , given its lower computational complexity, allowing to play more episodes and explore more in the environment. The ViT agent was the one with more steps to train, with roughly over 4.5×10^7 . Finally, the SWIN agent was the one that got less steps of training, with a bit more of 3×10^7 . It is clear that the CNN agent is the model with less parameters, and the one that needs more time and experiences to actually learn, but it collects experiences faster which compensates, obtaining a competitive reward. The ViT transformer is actually the most efficient in terms of ratio between number of parameters per point of reward. It is also quite evident that even though having a bit less number of parameters compared to the CNN, it actually needs far less number of steps to converge to the same reward, leveraging of the attention mechanism to extract richer features that estimate better situations, probably giving better estimations for the Q-values. The SWIN transformer is clearly the most complex model, which justifies the faster learning in terms of time-steps, as its reward curve ascends faster than the rest in terms of time-steps. It is also the less efficient model in terms of number of parameters per point of reward. But overall, the phenomena

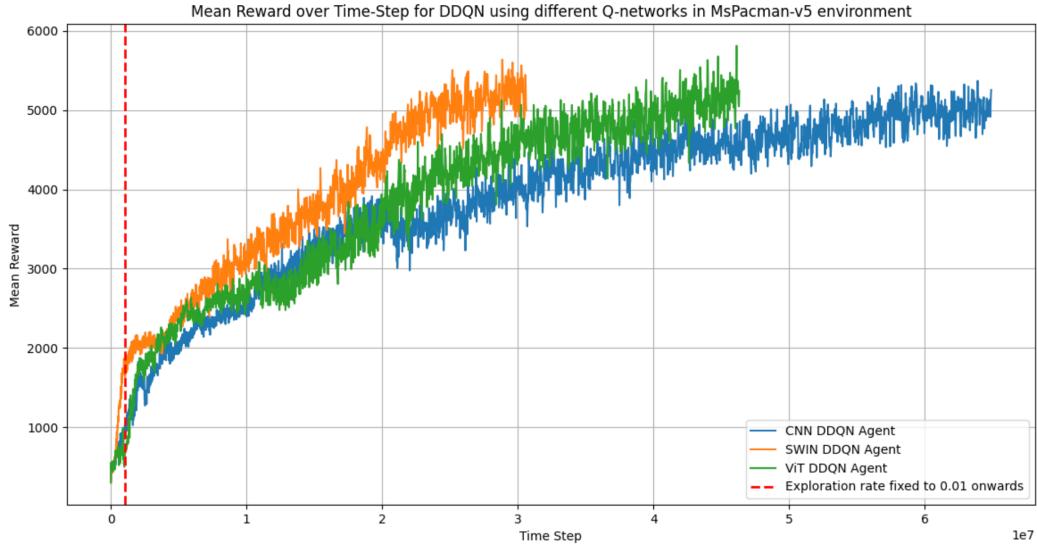


Figure 4.4: Reward vs. Time-steps for the CNN, SWIN and ViT agents.

that surprised us the most is the CNN being able to compete with the attention-based models, even though it is far more simpler.

Additionally, we also display in Figure 4.5 a chart that compares the reward against the time in seconds. There, the red dashed line represents the point where the exploration schedule finished for the CNN model, the magenta dashed line does the same for the SWIN agent and the purple for the ViT agent. The remaining exploration rate for the training will remain in a fixed value of $\epsilon=0.01$. It is quite interesting how the CNN agent actually learns faster in time compared to its the attention-based counter-parts. We associate this to the lower complexity of the model, and having far less parameters, which actually benefits in terms of convergence for the weight's values.

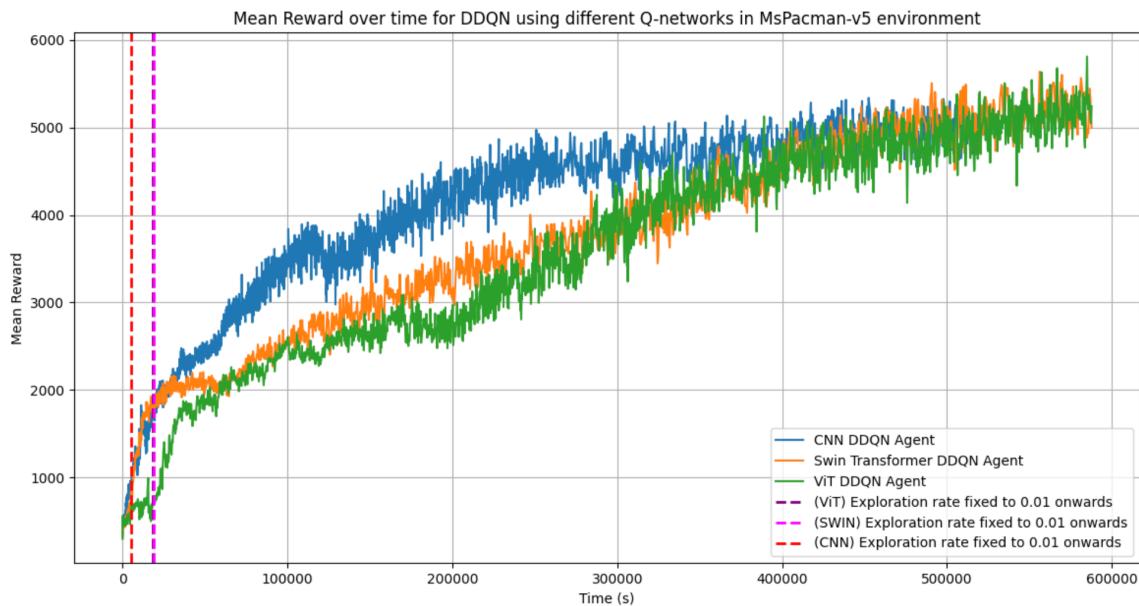


Figure 4.5: Reward vs. Time in seconds for the agents in MsPacman environment.

We can also see, even though is a bit overlapped that the CNN agent started to stuck around the second 2.5×10^5 since the reward did not pick up for the rest of the training. This

maybe because the CNN agent has a harder time improving as the game becomes more difficult, which is probably associated with its smaller amount of parameters. On the other hand, the attention-based models follow a similar pattern, by constantly improving, and achieving better results. It is clear that the attention-based models had potential to achieve higher rewards, and we wonder how they would have been if we could have the time to train them until no improvement was shown. As a final recap, in Table 4.7 we sum up the essential information.

Table 4.7: MsPacman environment results for the several trained agents

Agent type	Total Time (s)	Total Steps	Total Episodes	Max. Reward
CNN	6 days 4 hours	6.4881×10^7	192000	5369
SWIN	6 days 19 hours	3.0978×10^7	93850	5637
ViT	6 days 19 hours	4.6775×10^7	145552	5811

4.2.2.2 DemonAttack training results

For the DemonAttack environment, we followed a similar training approach as with the PacMan environment. We launched the models and trained them as much as we could, but, in the moment we saw that the progress in the reward stalled, we killed the training procedure. The training procedures for the attention-based Q-network agents went over 7 days, while the CNN Q-network agent was only run for over 2 days. We will explain the reason of this later in this section. The final training procedures were run in Computer 2. In Figure 4.6 we show the training results for our agents.

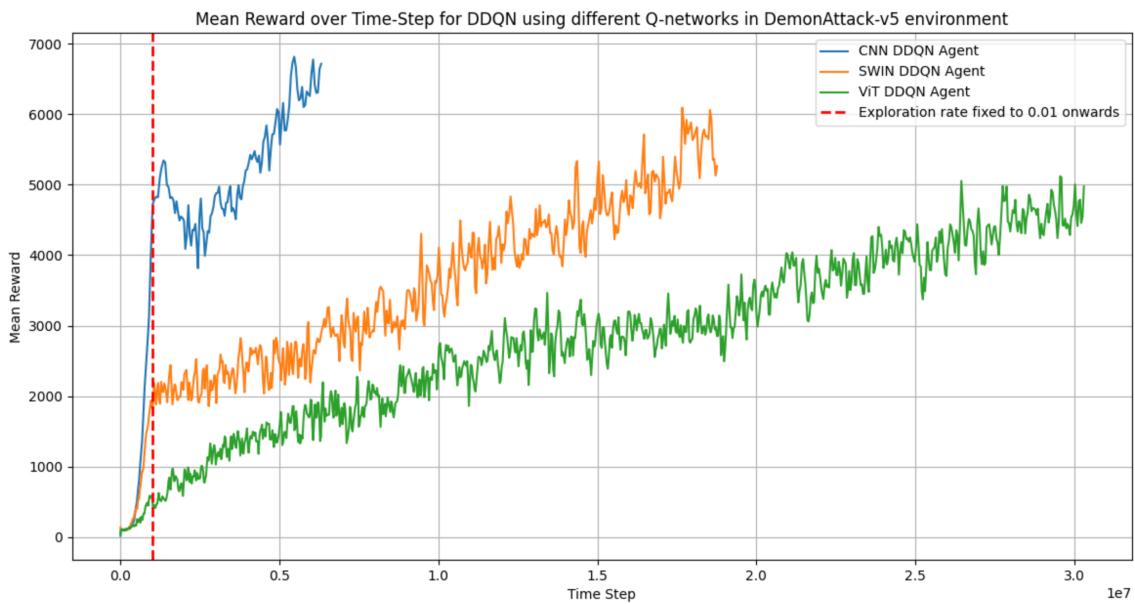


Figure 4.6: Demon Attack environment results in time-steps.

The progress in the training steps remains constant for the attention-based agents, as they linearly improve obtaining more rewards over time. Given the results of the previous section, we opted to only train the attention-based agents, since they over-performed the CNN. As

we launched the training, we face the situation where the learning was too slow. Just to give an idea, in [4] for just 50 million steps, the mean reward was around 80.000, while for our agents, after 30 million steps, it was just 4.000 for the ViT agent. Given these results, we launched the CNN agent to see if it was an issue with the set-up or with the Q-networks of our agents.

It was pretty clear that, as we launched the CNN agent, it started to score far better results than the attention-based agents. We do not have an explanation for this, since we did not expect these results, specially with short amount of time that we train it. It caught our eye how just in the exploration period, the CNN completely outperforms the ViT and the SWIN agent by large amounts of reward. Additionally, the difference between the ViT agent and the SWIN agent in the exploration phase is quite noticeable, which also happened in section 4.2.2.1. But the strangest thing of all, given our intuition of neural networks, is that the SWIN Transformer agent did not outperform the rest of the models, since it has almost three more times parameters than the rest of the models. We would have expected for the agent to eventually catch the CNN agent in terms of performance, and eventually surpassed given the capabilities of this Q-network of extracting meaningful features from visual data.

With respect to Figure 4.7 we can see that this training process took more time than MsPacman's, as reflected in Table 4.8. We saw that this happened because the performance of the GPUs actually differ from Computer 2 (where the agents for the DemonAttack were trained) and Computer 4 (where the MsPacman agent's were trained). We think that this happened because the GPUs from Computer 2, the RTX 3080's are a bit slower on paper than Computer's 3, that are RTX 3090's.

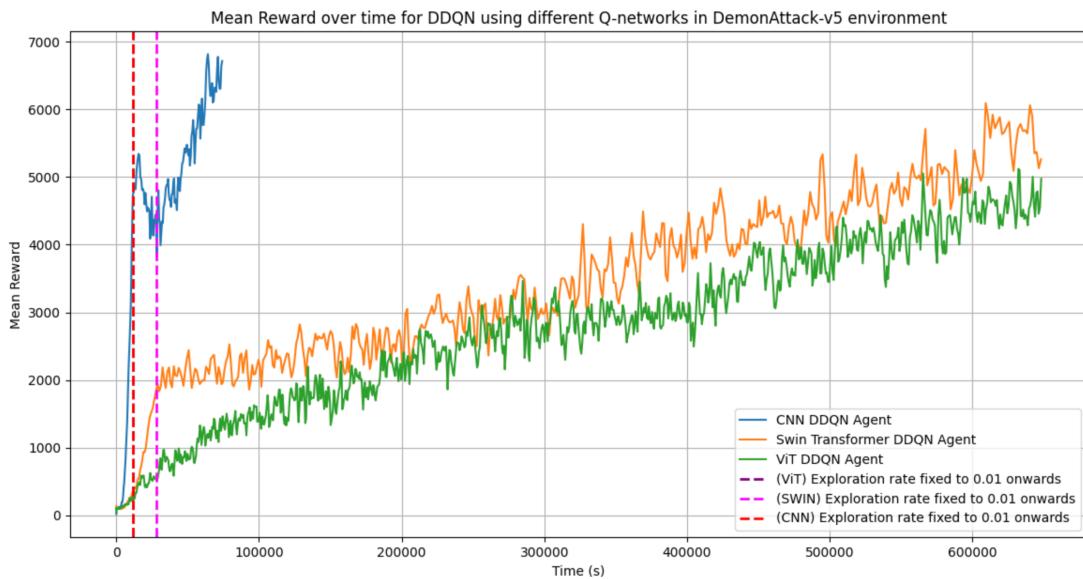


Figure 4.7: DemonAttack environment results in seconds.

Finally, in table 4.8 we summarize the results for the training procedure for the agents. The conditions are the same as in section 4.2.2.1, where the exploration rate is fixed to $\epsilon = 0.01$. This table portrays what we have already discussed, as the CNN agent out-performs the attention-based models, especially the ViT, by a considerable margin.

Table 4.8: DemonAttack environment results for the several trained agents

Agent type	Total Time (s)	Total Steps	Total Episodes	Max. Reward
CNN	2 days 1 hours	6.096×10^6	10400	6727
SWIN	7 days 18 hours	1.936×10^7	28150	6153
ViT	7 days 18 hours	3.127×10^7	51200	5120

4.2.3 Final evaluation results

Once the training procedures were done, we proceeded to evaluate the agents in the several environments where we trained them. For this task, we put the agents under "exploit mode", meaning that the exploration rate of the agents was $\epsilon = 0$. This differs from how we extracted the results previously explained in sections 4.2.2.1 and 4.2.2.2, since the maximal scores from the results in training were obtaining with an exploration rate $\epsilon = 0.01$. In order to obtain meaningful results, we run, for each environment, each agent for 100 episodes, extracting the mean and the standard deviation from all the runs. The code for extracting these results is in the [evaluate.py](#) script.

In Table 4.9 we see the final results scores (rewards) for the agents in evaluation mode. For the DemonAttack environment, the CNN agent was the best model. Scores are higher compared to the training results, which makes sense as the agents are completely exploiting the knowledge that they have acquired during the training phase. The SWIN agent actually surprised us, since it closed the gap to the CNN agent by quite a lot. The ViT agent performed the worst achieving a mean score that was less than the half of the CNN agent score. However, in the MsPacman environment things changed quite a lot. The ViT agent was actually the best performing out of the three. Also, it was the most constant in terms of always delivering the same score, since the standard deviation is the smallest of the three of them.

Table 4.9: Mean Scores for the trained agents

Environment	CNN Agent	SWIN agent	ViT agent
Demon Attack	7865.75 (± 2842.26)	7741.9 (± 2238.2)	3842.9 (± 2208.5)
Pacman	5382.3 (± 1635.37)	5829.12 (± 2067.05)	6044.71 (± 1623.28)

In Table 4.10, the maximum scores for each environment are shown. For the DemonAttack game, the CNN agent was actually the better by quite a margin, consolidating its position as the best performing agent of all them three. In the MsPacman game, something similar occurs for the ViT, as it outperforms by a considerable margin the rest of the agents. Even though the order for the max scores remains the same as the mean scores, in the DemonAttack game, the maximum score is doubled compared to the mean score for the attention-based agent, while it is almost tripled for the CNN agent. On the other hand, this does not happen for the MsPacman game, where the maximum scores are only doubled compared to the mean scores.

We also took into account the minimum scores of our agents, because we wanted not only to know how good our models can be, but also how bad. For the DemonAttack model, the ViT clinched its position as the worst performer in the game of them all, and by a large margin. In the following sections we will explore the explainability features, and hopefully

Table 4.10: Max Scores for the trained agents

Environment	CNN Agent	SWIN agent	ViT agent
DemonAttack	22050	14280	8415
MsPacman	9330	10345	12561

understand what may happen for this model to not perform as good as we expected. The rest of the scores for both the DemonAttack and MsPacman environments were kind of expected.

Table 4.11: Min Scores for the trained agents

Environment	CNN Agent	SWIN agent	ViT agent
DemonAttack	1700	760	140
MsPacman	1520	1990	1940

Additionally, we compared our results with [4], to see how "good" our models were. This comparison made sense for us since, to the best of our knowledge, is the work with the best scores that uses a transformer-based Q-network in the DDQN set-up, called SWIN DQN. They provide results for the whole ALE games, but, since we could only train our agents for two environments, those will be the ones we will comparing ourselves with them. In Table 4.12 we see the mean evaluation results, where in the DemonAttack environment, they outperform our models by almost 12 times. Nevertheless, for the MsPacman game, our ViT agent outperforms their proposed model, achieving a best mean score of over a 100 points.

Table 4.12: Comparison with SWIN DQN [4] for mean scores for the trained agents

Environment	CNN Agent (ours)	SWIN agent (ours)	ViT agent (ours)	SWIN DQN
DemonAttack	7865.7 (\pm 2842.2)	7741.9 (\pm 2238.2)	3842.9 (\pm 2208.5)	89254 (\pm 49893)
MsPacman	5382.3 (\pm 1635.3)	5829.1 (\pm 2067.5)	6044.7 (\pm 1623.2)	5878.0 \pm 2426.4

We also compared our agent's performance in terms of maximal scores, where we got beaten by the SWIN DQN agent in both games. In the DemonAttack game, we got beaten by a score that was 5 times better than ours. In the MsPacman environment, we got beaten by over a 1.400 points, even though our mean scores were better. One explanation that we have for this, is that their agent has a larger standard deviation, which means that the agent is less consistent to recurrently obtain similar scores to the mean, for worse, and for better.

Table 4.13: Comparison with SWIN DQN [4] for max scores for the trained agents

Environment	CNN Agent (ours)	SWIN agent (ours)	ViT agent (ours)	SWIN DQN
DemonAttack	22050	14280	8415	134465
MsPacman	9330	10345	12561	13911

4.3 Explainability features

4.3.1 Attention maps results

For the attention maps, we extracted a whole display of graphs with the aim of providing an insightful analysis on the attention heads for the ViT. Recalling section 3.4.4, we remind the reader we could only extract the attention maps for the ViT Q-network

4.3.1.1 MsPacman attention maps analysis

For the MsPacman environment, the extracted maps are displayed in Figure 4.8, where as the colour-bar suggests, the more yellow a section is, the more attention the ViT is paying to it. All the figures presented in this section are part of a set of videos that we have recorded in order to better analyse the sequences when actions are performed.

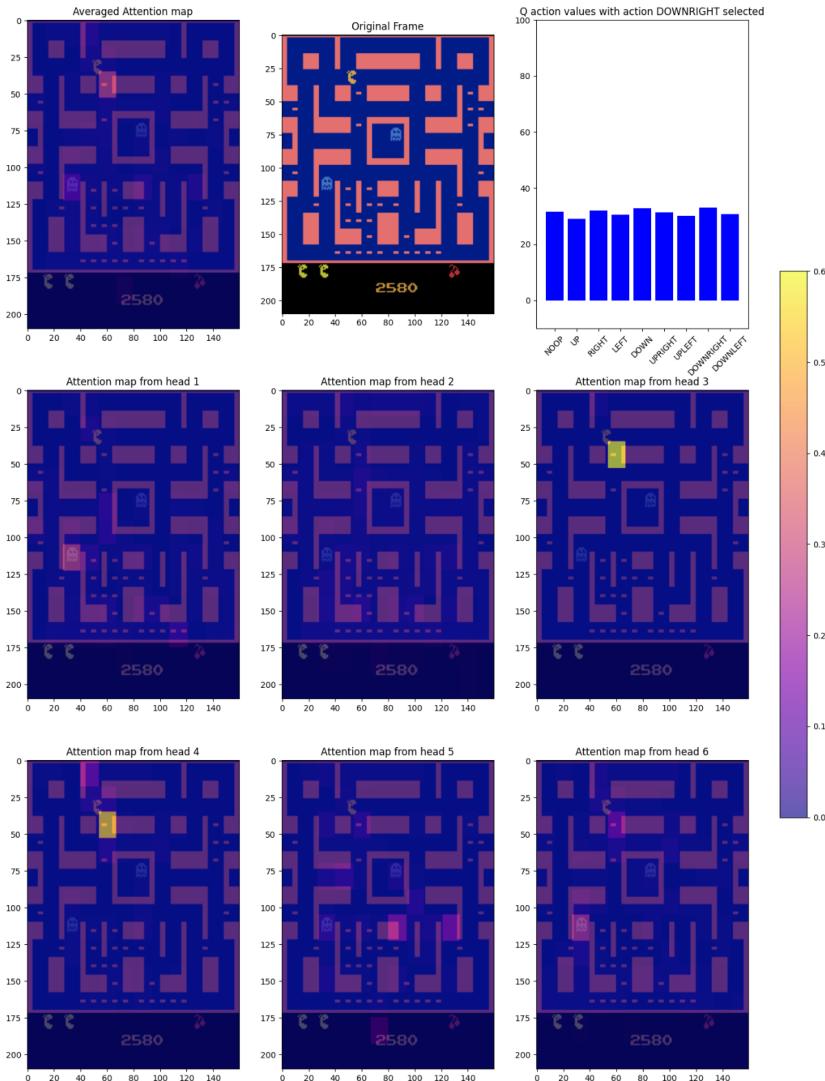


Figure 4.8: In game frame where we provide (from left to right) the averaged attention map, the original frame and the Q-values as the output of the network. Additionally we provide the attention heads from the ViT Q-network.

The videos are available in [this](#) folder in Google Drive. We will try to illustrate as good as we can what we have seen in the videos, but we encourage the readers to see them for themselves, since they provide better context on what we will explain in the following sections. For a better visualization, we used the video player built in Visual Studio Code, since it allows for different reproduction speeds, as does the built-in player in Google Drive.

In general terms, the attention heads pay a lot of attention to where Pacman is in the map, which make a lot of sense, since us humans would actually do the same in order to perform the actions accordingly. But as it can be seen, in Figure 4.8, it additionally pays attention to additional elements in the map, which makes sense also, as we humans can perceive several things at the same time and take into account several factors to perform decision-making.

One of the most critical situations the agent can face in the game is when Pacman encounters a ghost, since it is its principal threat when trying to survive in the game. For example, in Figure 4.9 we have a series of frames where Pacman encounters a ghost. From left to right, in the first frame, we see an activation in the attention head number 4, acknowledging that the ghost is near Pacman. As frames go by, the attention head number 3 seems to be looking for a place in the map that is safe for Pacman to be, as attention head number 4 intensifies the attention paid to the ghost, that is coming towards Pacman. In the final frame, we actually see how the agent has placed Pacman exactly in the spot it searched frames before.

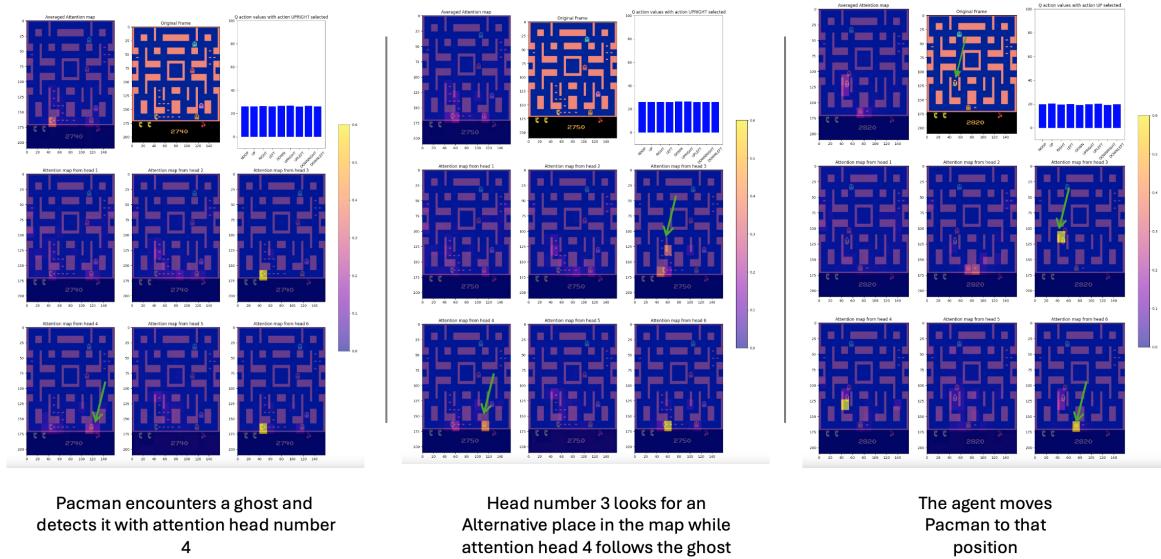


Figure 4.9: Three snapshots from a situation where the agent encounters a ghost. It acknowledges the ghost and searches for routes to avoid them.

Extending this phenomenon, we see that the ViT agent attention indicates where it may want to go in the next consecutive frames. In Figure 4.10 we see an example of this. From left to right, in the first frame we see that Pacman is located by the agent in attention head number 4, but attention head number 3 attends to a region of the map which may be of interest. Additionally, attention head number 2 looks for another region of interest (where the tiles are), since there may be potential reward. In the second frame, we see that the agent is going exactly where the attention head 3 payed attention to in the previous image, and that attention head number 2 still paying attention to the section on the map where the tiles are. In the third image, we see that the agent puts Pacman in the section that the attention head 2 was paying attention from the beginning. As we can see, this could be proof that by looking

at the attention heads of well-trained models we can effectively predict and understand which are the model’s decisions when playing these games and predict which will the next moves or what are the regions of interest for the agent. We remark this phenomenon as it actually happens a frequently in the videos provided in Google Drive.

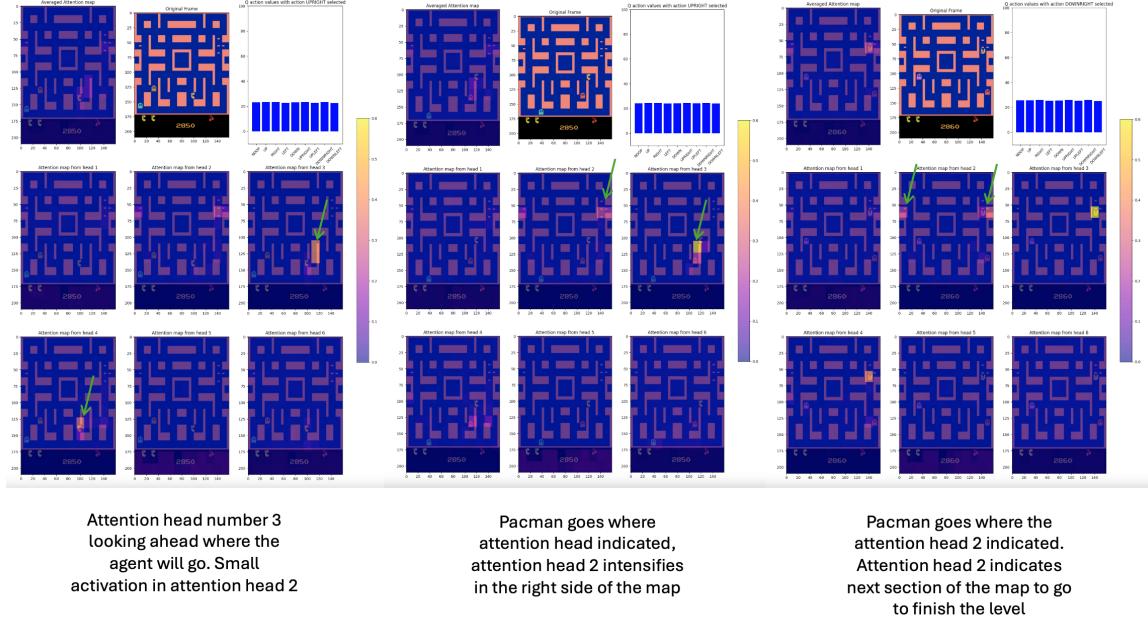


Figure 4.10: Three snapshots from a situation where the agent tries to find rewards in the environment.

4.3.1.2 DemonAttack attention maps analysis

For the Demon Attack environment, we collected a series of episodes where we let the agent play, following the same set-up as the previous section. We were aware of the fact that the attention maps from the agent in this environment would not be as interpretable, since the performance from this model was inferior comparatively. Either way, in Figure 4.11a we see a screenshot of the map, where we see the different attention heads. The activations are clearly not as rich as they are in the MsPacman environment. The attention heads pay attention to the shooter, since its the most important part of the game, but it lacks on amplifying the context. For example, attention heads 1 and 5 do not give any important information, and the activations are 0 for almost the entire part of the game. These heads should be adding information, giving a richer and broader context of the state, so that the Q-network provides accurate estimations on the Q-values for the actions. There is a phenomenon that is even worse, as the attention heads 3 and 6 are paying attention to useless sections of the map, like the bottom part, where there is not relevant information for the gamer to actually obtain greater rewards. In figure 4.11b, we saw that, as the game progresses, the agent starts to pay attention to the scoreboard. We actually think this is a shortcut [49] from the Q-network, since no real reward is obtained by looking directly at the scoreboard, but it might learn in some stage of the game that in that section something interesting usually happens (maybe because the enemies usually appear from the top section of the game). We felt discouraged by the performance of this agent, because it was the best in the MsPacman game, but then we also realized that attention maps explain the poor behaviour of the model, as it is giving

us hints on why it does not perform well. This can be useful when comparing different models based on attention, as we have seen that they will effectively describe what is the model looking for in order to perform decision-making. With respect to the agent's performance, as we are writing this, we are trying to improve the model, with the hopes of confirming what we have stated in previous sections, and remains as future work in this line of research.

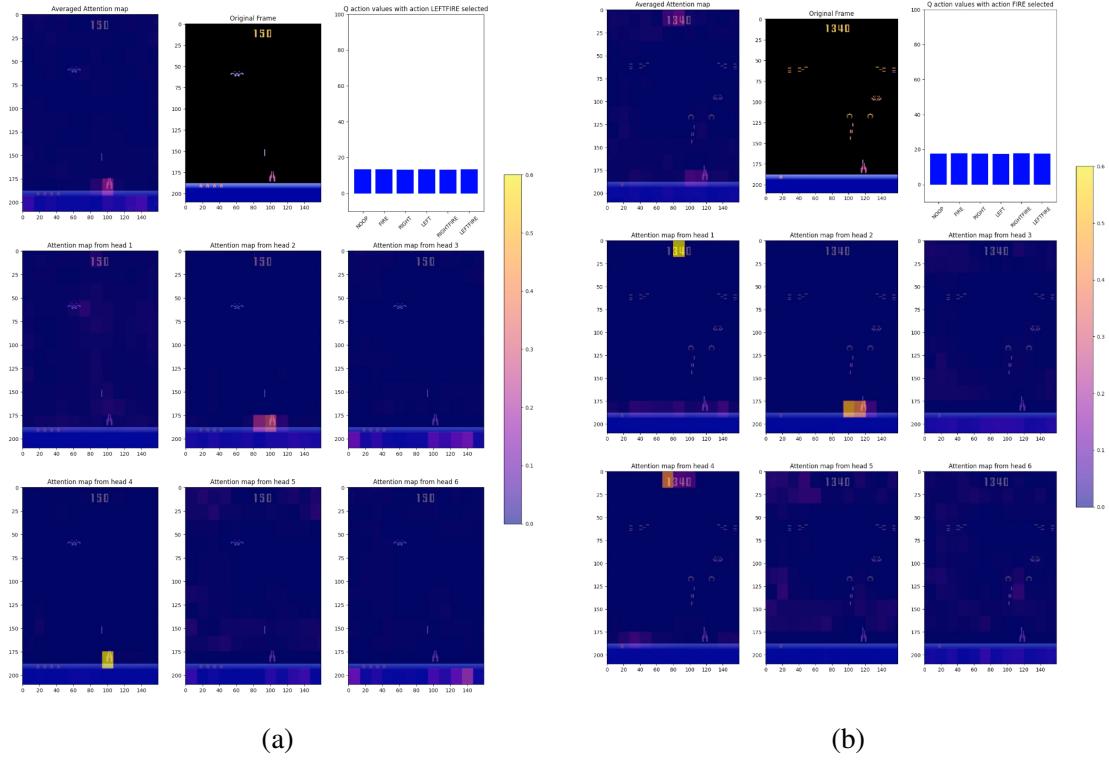


Figure 4.11: Attention maps for the DemonAttack environment.

4.3.2 Activation maps results

Once we analysed the attention maps for the ViT agent, we proceeded to evaluate the activation maps that Grad-CAM provided for both the ViT and the SWIN Transformer agents. We did not evaluate the CNN agent because we wanted to prioritize the attention-based models, as that is the aim of this work. However, we keep this task as future work.

4.3.2.1 MsPacman Activation maps analysis

For the ViT agent, the activations are shown in Figure 4.12. We show a set of three frames from different stages of the game (as it can be seen by checking the score). It is pretty clear that the ViT agent does only care about one thing, the tiles. From top to bottom, we see that in the first stages of the game, the agent has big activations everywhere, since there are a lot of tiles scattered throughout the maze. As the agent starts to move around, eating lots of the tiles, the activation maps show that the sections of the map where there are still tiles have high activation, while sections where there are no tiles have no activation. This matches a lot with the intuition that we were getting about the agent, since it seemed it mostly cared about eating the tiles.

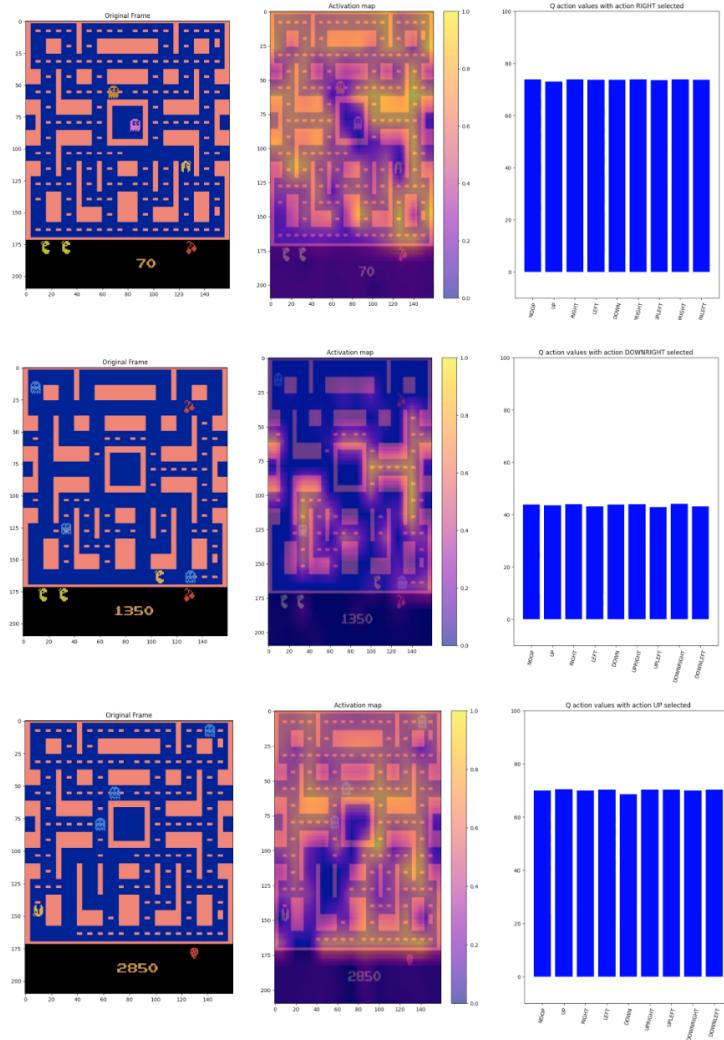


Figure 4.12: Several Grad-CAM activations over different phases of the MsPacman game for the ViT agent.

And it seems like the tiles are the logical option for the agent in order to maximize the reward. Recalling equation 2.3, the value of a state depends on the reward that we obtain not only from the time-step we are in, but also for the potential rewards the agent may land on in future time-steps. Where we are at the beginning of the game, a lot of potential rewards are available, and that is what we think Grad-CAM is showing with these activation maps. As the tiles are eaten, the agent seems to lose those in the sections of the map where there are no tiles. This also matches with the Q-values the agent predicts. When the game starts, lots of potential rewards can be achieved with almost any action, hence in the top section, the Q-values are high for every action. As long as the game progresses, the number of rewards decay, and so does the Q-values for every action. When the ViT agent finds itself again in a position where lots of potential rewards are available (since it has moved-up one level), the Q-values grow again, correlating with the increasing area in the map where the activations are the highest.

For the SWIN transformer, the activation maps were actually far less interpretable. We show in Figure 4.13 a set of frames where the SWIN agent is (hardly) interpretable, since we can see that the higher activations are in sections of interest of the map. The rest of

the activation maps were actually non-sense. For example, in figure 4.14, we see a activation map that does not make sense given the state, and like this, there are lots of them in the evidences that we extracted. We have two hypothesis on why this may not be working. First, there is a compression in the spatial features in the SWIN transformer, in the Patch Merging layer to be concrete. This compression may be altering the features in ways that we cannot comprehend, hence providing this activation maps. The second is that Grad-CAM is supposed to be used in networks that have been trained under classification task, and although there are some similarities between classifying images and select actions for a given state, the truth is that the losses used to train these models differ, since the classification task uses cross entropy loss and DQN uses the MSE.

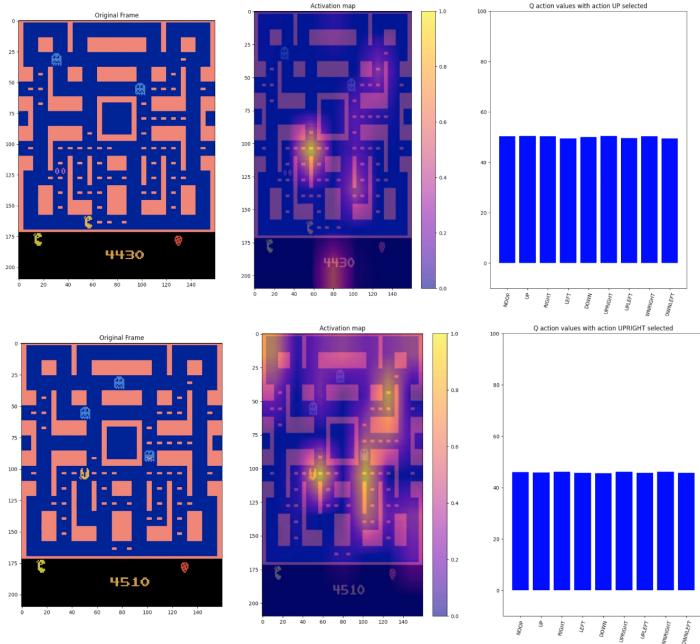


Figure 4.13: Several Grad-CAM activations over different phases of the MsPacman game for the SWIN transformer agent.

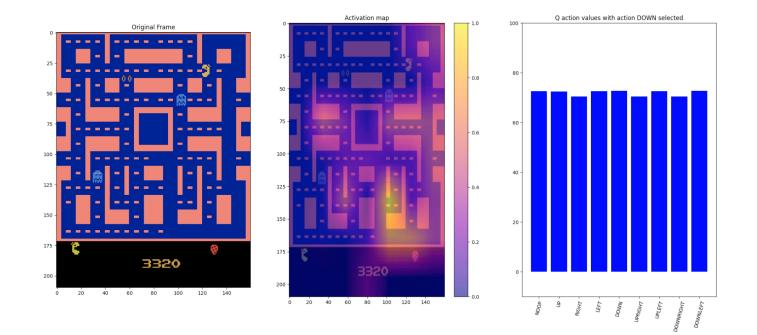


Figure 4.14: Grad-CAM activations for the MsPacman game for the SWIN transformer agent, where it is clear that it is not understandable what is the agent looking for.

4.3.2.2 DemonAttack Activation maps analysis

The activation maps of the ViT agent for the DemonAttack game are available in Figure 4.15. From top to bottom, we see that the frame at the top shows no activations whatsoever, with the exception of the score board, which matches with what the attention maps were showing, although we still believe it does not make sense in terms of the context. For the other two activation maps, we see precisely the opposite, as the activations are all around the frame, even at the bottom of the screen, but there are no activations in the scoreboard. Opposite to what we encountered in the MsPacman environment, here, the activation maps seems to not correlate with what is going on in the game or the Q-values that the Q-network is estimating.

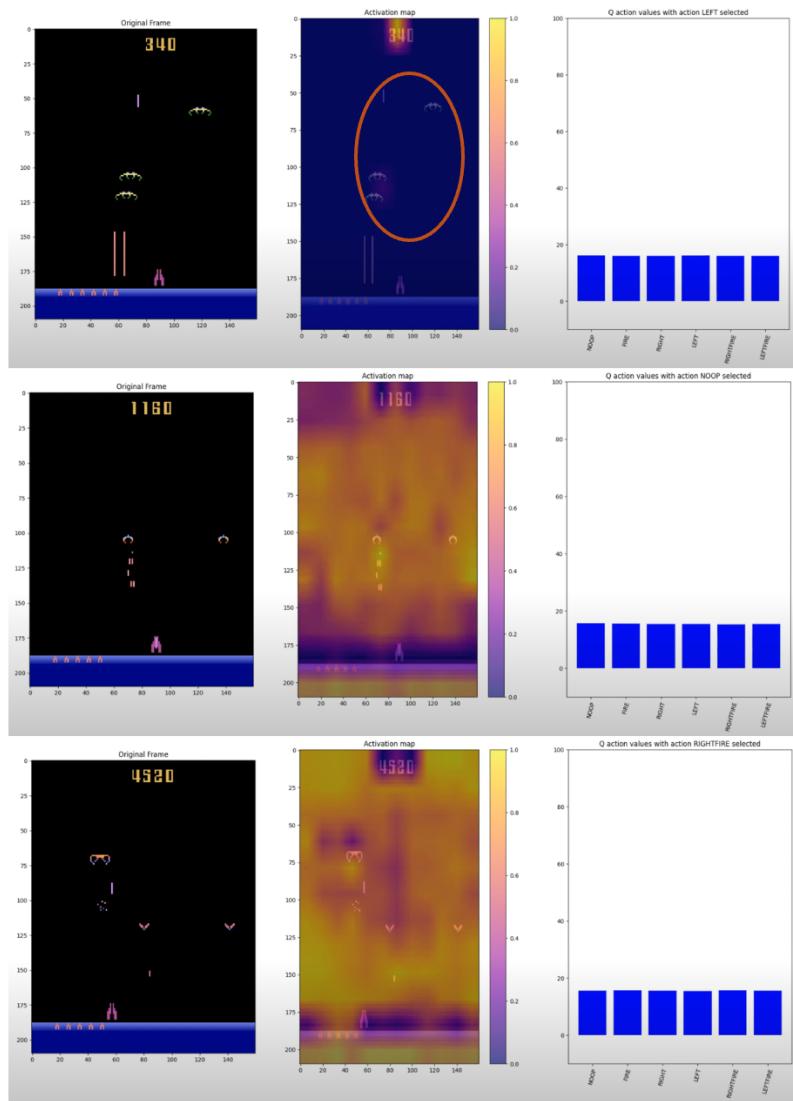


Figure 4.15

The activation maps of the SWIN agent for the DemonAttack game are available in Figure 4.16. From top to bottom, we see that the frame at the top shows a bit of more understandable activation maps, since they are more interpretable as it usually has greater activations for sections where the enemies are. Nevertheless, the middle and the bottom images show that the activation maps are not quite reliable, since they miss lots of enemies in the frames, as we added some orange circles in the sections where the enemies are present, but no acti-

vation seems to appear. This may prove that we are using Grad-CAM in a set-up where we may not be exploiting all its capabilities.

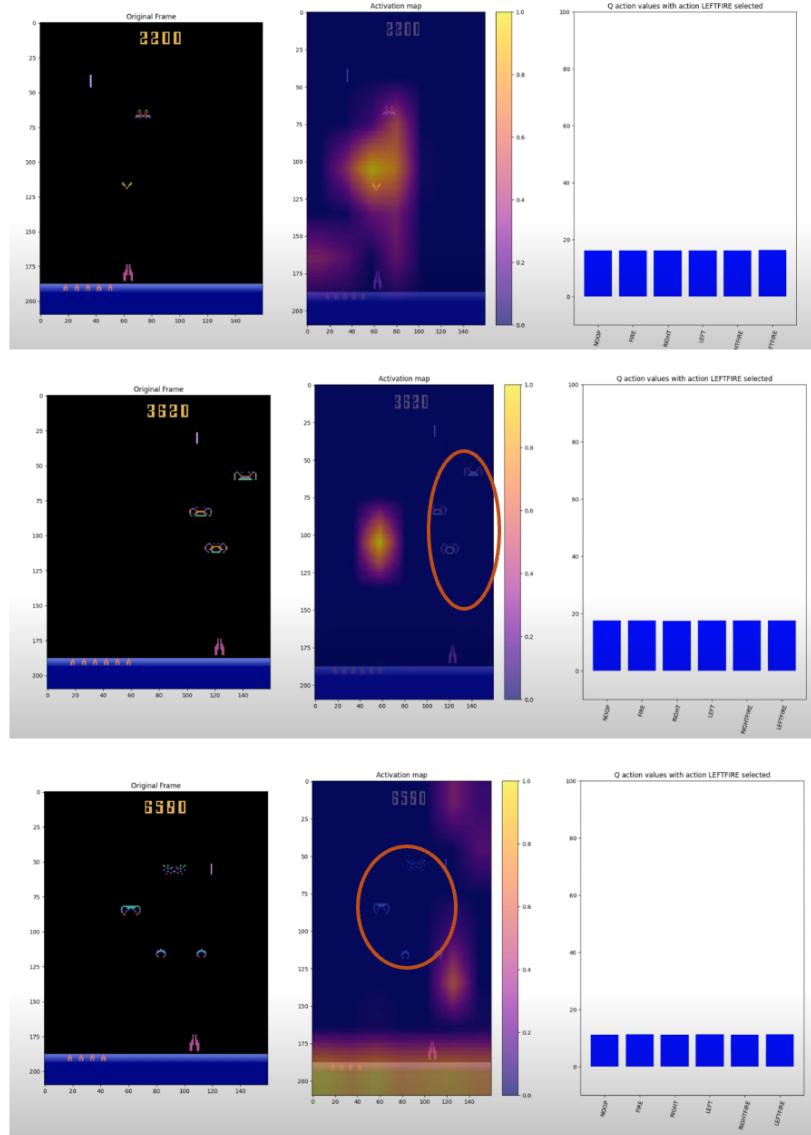


Figure 4.16: Activation maps for the SWIN agent in the Demon Attack environment. Some improvements are obtained in terms of interpretability, but nothing conclusive.

As bottom-line for this section, we can confirm that the Grad-CAM method is not consistent providing explainable or interpretable features so we can understand what is the agent looking for in order to perform in-game decisions.

Chapter 5

Conclusions and future work

5.1 Conclusions

Given the evaluation that we have provided, one thing is clear: Attention can be used as an explainability technique to understand the decisions that an agent may perform. The ViT seems a suitable model to address this issue, as it provides the most explainable features, both when it performs great and when it performs poor. The elephant in the room with this model is its computational complexity, since we have seen that it has the roughly the same number of weights as the CNN agent but orders of magnitude more complexity in the forward pass. The SWIN transformer seems like the most balanced model, since it is not the most complex model, and is efficient computing the attention in a hierarchical fashion, using windows, which reminds to the CNN. The issue with this model is that, since it lacks of a class token, we cannot actually look into which patches or which windows are crucial for the decision on taking some action or another. At last, the CNN model was the most surprising for us, since it held its own in terms of performance, being close to the attention-based models or even surpassing them, as we saw with the DemonAttack environment. This model lacks of explainability, which may be a concern for applications of RL in real world scenarios, but it reminds us that sometimes, new is not always better.

Referring to the training schedules, we have concluded that the best performing schedule is actually the exponential decay. As we have stated in section 4.2.1, this is for us quite counter-intuitive, and further exploration should be done, to clarify whether what happened is actually true or not.

On a personal note, this work has been quite challenging, since we almost had no prior knowledge of reinforcement learning before we developed this master thesis. By working in some out-of-scope field, we have actually learned a lot on why Deep Learning is useful to tackle not only supervised problems, but also another paradigms as the one we have faced in this work. We do not know if RL is the way to go to achieve artificial general intelligence, but it sure does has similarities with humans, and if we actually managed to make it understandable for us, who knows, maybe we could understand what is missing to actually make them behave like us, or even better.

5.2 Future work

Lots of work can potentially be done to continue exploring the area of explainability in the area of reinforcement learning. First, the results were obtained by restraining the total time-

steps to almost a quarter of the usual set-up in the literature, so it would be of great use to train the models up to completion to have better insights on the actual performance of the model. With a well-trained model, we argue that the attention and activation maps would have more relevance, since we would be analysing the behaviour of a "almost" optimal agent.

To speed-up the training procedures, we could implement a pipeline that makes use of distributed GPU computing. In the state of the art's frameworks for Deep Learning, such as TensorFlow or PyTorch, distributed and paralleled training is supported supervised and semi-supervised set-ups, but not for reinforcement learning. We have seen in this work that there have been some advancements as there are developed libraries where this is implemented using Message Passing Interfaces. We believe that developing an efficient parallelized pipeline would improve training times by a lot, making room for more experiments that would give us better insights on the proposed models.

Since we have trained effectively value functions, we could use them for training Actor-Critic methods. As explained, these methods use two models, a policy model, that governs the agent's behaviour and performs actions, and a value function that evaluates how good those actions are. We already have a value function, and we are able to extract explanations from them, so we argue that we could extend its use to Actor-Critic set-ups for better understanding on the developed policy. Additionally, the ViT-Big model was not trained due to time constrains, but we wonder whether it would outperform all the proposed models, giving a even better Q-network to estimate the values from states.

For the explainability and interpretability in attention models, we have shown that attention methods do explain the agent's behaviour to an extent, but sometimes fall short, since attention maps are less expressive than other methods. In the last years, Meta AI has discovered that in a self-supervised set-up using distillation [50] and mean student-teacher training [51], where quite interesting properties emerge in the attention maps. This model is called DINO [42], and we argue that by using transfer learning and adapting the pre-trained weights to our problem, some interesting findings could arise, such as clearer attention maps or faster convergence to an better policy behaviour. Additionally, there is a family of class activation methods (CAM) [52, 53, 8, 9] that we could also try to explore if some relevant findings appear in the activation maps, or use Layer-wise Relevance Propagation and its variants to extract its activation maps and see if some better explanations are provided, since these methods do not need of a up-scaling as CAM methods do, giving pixel-wise relevance activations to the input.

At last, we would have loved to have more time to perform more training procedures. One of the things from this work that are pending is to understand why the exponential scheduling is the better performing one. Maybe we could run several runs or extend the exploration time from 1M to several millions. Additionally, we could try several functions from [54], where they propose several decay schedulers for the learning rate of supervised learning set-ups, that we could use to schedule the decay of the exploration rate.

Bibliography

- [1] M. Fabien, “Markov decision process,” 2022. [xi](#), [6](#)
- [2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2023. [xi](#), [11](#), [12](#), [13](#)
- [3] Z. Min, “Attention link: An efficient attention-based low resource machine translation architecture,” 2023. [xi](#), [13](#)
- [4] L. Meng, M. Goodwin, A. Yazidi, and P. Engelstad, “Deep reinforcement learning with swin transformers,” 2024. [xiii](#), [18](#), [45](#), [46](#), [47](#), [54](#), [56](#)
- [5] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. [1](#), [5](#)
- [6] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, “On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation,” *PloS one*, vol. 10, no. 7, p. e0130140, 2015. [2](#), [17](#)
- [7] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, “Grad-cam: Visual explanations from deep networks via gradient-based localization,” *International Journal of Computer Vision*, vol. 128, p. 336–359, Oct. 2019. [2](#), [17](#), [18](#)
- [8] A. Chattopadhyay, A. Sarkar, P. Howlader, and V. N. Balasubramanian, “Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks,” in *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, IEEE, Mar. 2018. [2](#), [66](#)
- [9] M. B. Muhammad and M. Yeasin, “Eigen-cam: Class activation map using principal components,” in *2020 International Joint Conference on Neural Networks (IJCNN)*, IEEE, July 2020. [2](#), [66](#)
- [10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” 2013. [8](#), [9](#), [26](#), [32](#), [45](#)
- [11] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” 2015. [8](#), [9](#), [18](#), [31](#), [46](#), [47](#)
- [12] M. Lehmann, “The definitive guide to policy gradients in deep reinforcement learning: Theory, algorithms and implementations,” 2024. [10](#)
- [13] Y. Wang, C. Zhang, T. Yu, and M. Ma, “Recursive least squares advantage actor-critic algorithms,” 2022. [10](#)

- [14] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” 2016. [10](#), [77](#)
- [15] S. Yadav, “Understanding attention mechanism,” 2023. [10](#)
- [16] I. D. Mienye and Y. Sun, “A survey of ensemble learning: Concepts, algorithms, applications, and prospects,” *IEEE Access*, vol. 10, pp. 99129–99149, 2022. [12](#)
- [17] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, “An image is worth 16x16 words: Transformers for image recognition at scale,” 2021. [13](#), [32](#)
- [18] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, “Swin transformer: Hierarchical vision transformer using shifted windows,” 2021. [14](#), [32](#), [85](#)
- [19] Y. Dai, Z. Xu, F. Liu, S. Li, S. Liu, L. Shi, and J. Fu, “Parotid gland mri segmentation based on swin-unet and multimodal images,” 06 2022. [15](#)
- [20] W. R. Swartout, “Explaining and justifying expert consulting programs,” in *Computer-assisted medical decision making*, pp. 254–271, Springer, 1985. [16](#)
- [21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Commun. ACM*, vol. 60, p. 84–90, may 2017. [16](#)
- [22] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. [16](#)
- [23] M. Turek, “Darpa - explainable artificial intelligence (xai) program,” 2017. Accessed: 2024-05-13. [16](#)
- [24] F. Xu, H. Uszkoreit, Y. Du, W. Fan, D. Zhao, and J. Zhu, “Explainable ai: A brief survey on history, research areas, approaches and challenges,” in *Natural Language Processing and Chinese Computing* (J. Tang, M.-Y. Kan, D. Zhao, S. Li, and H. Zan, eds.), (Cham), pp. 563–574, Springer International Publishing, 2019. [16](#)
- [25] J. Pizarroso, J. Portela, and A. Muñoz, “Neuralsens: Sensitivity analysis of neural networks,” *Journal of Statistical Software*, vol. 102, no. 7, 2022. [16](#)
- [26] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, “Learning deep features for discriminative localization,” 2015. [17](#)
- [27] W. Li, H. Luo, Z. Lin, C. Zhang, Z. Lu, and D. Ye, “A survey on transformers in reinforcement learning,” 2023. [18](#)
- [28] A. A. Kalantari, M. Amini, S. Chandar, and D. Precup, “Improving sample efficiency of value based models using attention and vision transformers,” 2022. [18](#)
- [29] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The arcade learning environment: An evaluation platform for general agents,” *Journal of Artificial Intelligence Research*, vol. 47, p. 253–279, June 2013. [18](#), [21](#)

- [30] M. Janner, Q. Li, and S. Levine, “Offline reinforcement learning as one big sequence modeling problem,” 2021. [18](#)
- [31] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch, “Decision transformer: Reinforcement learning via sequence modeling,” 2021. [19](#)
- [32] K. Wang, H. Zhao, X. Luo, K. Ren, W. Zhang, and D. Li, “Bootstrapped transformer for offline reinforcement learning,” 2022. [19](#)
- [33] L. Bramlage and A. Cortese, “Generalized attention-weighted reinforcement learning,” *Neural Networks*, vol. 145, pp. 10–21, 2022. [19](#)
- [34] Y. C. Leong, A. Radulescu, R. Daniel, V. DeWoskin, and Y. Niv, “Dynamic interaction between reinforcement learning and attention in multidimensional environments,” *Neuron*, vol. 93, no. 2, pp. 451–463, 2017. [19](#)
- [35] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, “Stable-baselines3: Reliable reinforcement learning implementations,” *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [25](#)
- [36] A. Bou, M. Bettini, S. Dittert, V. Kumar, S. Sodhani, X. Yang, G. D. Fabritiis, and V. Moens, “Torchrl: A data-driven decision-making library for pytorch,” 2023. [25](#)
- [37] J. Eschmann, *Reward Function Design in Reinforcement Learning*, pp. 25–33. Cham: Springer International Publishing, 2021. [26](#)
- [38] S. Sinha, “The exploration–exploitation dilemma: A review in the context of managing growth of new ventures,” *Vikalpa*, vol. 40, no. 3, pp. 313–323, 2015. [27](#)
- [39] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” 2016. [29, 45](#)
- [40] A. Patterson, V. Liao, and M. White, “Robust losses for learning value functions,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 5, pp. 6157–6167, 2023. [31](#)
- [41] Y. Feng, S. Subramanian, H. Wang, and S. Guo, “Train a mario-playing rl agent.” https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html, 2022. Accessed: 2024-02-03. [35](#)
- [42] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin, “Emerging properties in self-supervised vision transformers,” 2021. [40, 66, 81](#)
- [43] J. Gildenblat and contributors, “Pytorch library for cam methods.” <https://github.com/jacobgil/pytorchGradCam>, 2021. [42](#)
- [44] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, “Rainbow: Combining improvements in deep reinforcement learning,” 2017. [45, 46](#)
- [45] A. Stooke and P. Abbeel, “rlpyt: A research code base for deep reinforcement learning in pytorch,” 2019. [46](#)

- [46] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: MIT Press, 1994. [46](#)
- [47] T. Yep, “torchinfo: View model summaries in pytorch.” <https://github.com/TylerYep/torchinfo>, 2022. Accessed: 2024-04-23. [47](#)
- [48] V. Sovrasov, “ptflops: a flops counting tool for neural networks in pytorch framework,” 2018-2024. [47](#)
- [49] R. Geirhos, J.-H. Jacobsen, C. Michaelis, R. Zemel, W. Brendel, M. Bethge, and F. A. Wichmann, “Shortcut learning in deep neural networks,” *Nature Machine Intelligence*, vol. 2, pp. 665–673, Nov 2020. [59](#)
- [50] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” 2015. [66](#)
- [51] A. Tarvainen and H. Valpola, “Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results,” 2018. [66](#)
- [52] P.-T. Jiang, C.-B. Zhang, Q. Hou, M.-M. Cheng, and Y. Wei, “Layercam: Exploring hierarchical class activation maps for localization,” *IEEE Transactions on Image Processing*, vol. 30, pp. 5875–5888, 2021. [66](#)
- [53] R. Fu, Q. Hu, X. Dong, Y. Guo, Y. Gao, and B. Li, “Axiom-based grad-cam: Towards accurate visualization and explanation of cnns,” 2020. [66](#)
- [54] T. Zhang and W. Li, “kdecay: Just adding k-decay items on learning-rate schedule to improve neural networks,” 2022. [66](#)
- [55] T. Gerstner, B. Harrach, D. Roth, and M. Simon, “Multilevel monte carlo learning,” 2021. [73](#)
- [56] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, pp. 279–292, May 1992. [76](#)
- [57] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” 2018. [76](#)
- [58] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017. [76](#)
- [59] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” 2019. [76](#)
- [60] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2015. [83](#)

Appendix

Appendix A

Reinforcement Learning

A.1 Classic learning methods in RL

A.1.1 Monte Carlo estimation:

Monte Carlo estimation [55] uses complete episodes to approximate the value function using the empirical mean. The value function is based on the returns G_t , and since we are trying to approximate $V_\pi(s)$, our returns will be the statistic we will use to adjust the value of a state S_t . In non-stationary problems, where the environment may change over time, the value function will be incrementally updated as in equation A.1, and by the law of large numbers, if the number of times we visited a state S goes close to infinity, then $V(S) = V_\pi(S)$

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)) \quad (\text{A.1})$$

Where α is the step-size hyper-parameter that tells the update function how much error take into account in the when performing the adjustments from the estimates. This approach has unbiased but noisy estimations, as the rewards distribution may not be consistent between episodes, creating prone-to-error estimates that may take a long time to converge to the true value function $V_\pi(s)$.

A.1.2 Temporal Difference Learning:

TD methods learn from episodic and non-episodic experiences, so the concept of bootstrapping is introduced. Bootstrapping refers to the ability of a model to make an approximation of the value function every each n -steps, using the estimation of the returns we have computed as we have collected more and more experience without finishing the current episode. The simplest set-up of TD is TD(0), where we look ahead only one time-step, from S to S_{t+1} , and the update for the value function is defined in equation A.2.

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \quad (\text{A.2})$$

Why is this a good idea? imagine a situation where an agent is driving a car, and another car comes towards the agent, but avoids it in the last moment. In the Monte Carlo learning approach, as it learns from complete episodes, the agent would not have learned anything, since the episode ended without much consequences. In TD(0), since it is learning from each time-step, the agent would have learn from the experience itself, and for example, discover that, when a car is approaching, slowing down to have better maneuverability could be an great approach.

Looking just one step in the future might be a little short-sighted, so we can consider to look several steps ahead, considering:

- For $n = 1 \rightarrow G_t^1 = R_{t+1} + \gamma V(S_{t+1})$
- For $n = 2 \rightarrow G_t^2 = R_{t+1} + \gamma R_{t+2} + \gamma V(S_{t+1})$
- For $n = k \rightarrow G_t^k = R_{t+1} + \gamma R_{t+2} \dots \gamma^{k-1} R_{t+k} + \gamma^k V(S_{t+k})$

What we may not know which n is better, since the search space can be infinite. So the solution that $\text{TD}(\lambda)$ proposes is to have a weighted sum of the returns taking into account different time horizons (equation A.3), and then use it as our estimation of the value as portrayed in update equation A.4. Figure A.1 portrays a visual illustration of how this works.

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^k \lambda^{n-1} G_t^n \quad (\text{A.3})$$

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^\lambda - V(S_t)) \quad (\text{A.4})$$

A few things to comment about the $\text{TD}(\lambda)$ equation is that if $k = \infty$, then we would be in a similar to Monte Carlo learning update, but if $k = 1$, then we would end up in the $\text{TD}(0)$ update equation.

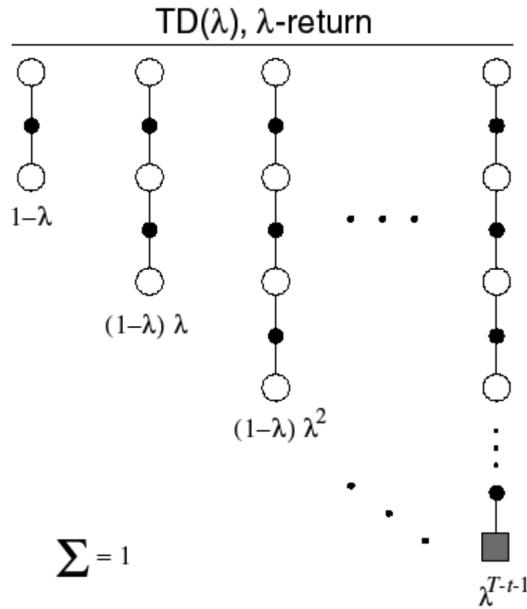


Figure A.1: TD Returns

The policy will use these values to act greedily, as defined in equation A.5. Since what it wants to maximize is the value, given the MDP model and the action space \mathcal{A} , the policy will select the action that gives more value in future states s' . The issue is that, as stated at the beginning of this section, we are in model-free RL and we no longer have the environment dynamics and \mathcal{P} and reward function \mathcal{R} . In the next section we will introduce control, and with it, how the policy problem is tackled.

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s') \quad (\text{A.5})$$

A.1.3 On-Policy Control:

On policy RL means that the updates of the value functions are performed according to following a policy π . Up until this point, everything we have done is evaluate states, and act according to those values. The issue is, that the policy defined in equation A.5 requires a model of the MDP, and since we are in model-free RL, this is not possible. Instead, we introduce action-value functions, $Q(s, a)$ that take into account, the value of an action, given a state (equation A.6).

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \quad (\text{A.6})$$

Now, we re-write the policy in equation A.7, that it still greedy, since we pick the action a that maximizes the action value estimates Q , given a state s .

$$\pi'(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q(s, a) \quad (\text{A.7})$$

One thing to note is that, if our policy always acts greedily, then it will never explore other states, since it will always prioritize rewards over landing on new states that may lead to biggest rewards. This is why the ϵ -greedy policy was introduced. This policy acts greedily with a probability of $p = (1 - \epsilon)$ and randomly with a probability $p = \epsilon$.

Since now we are dealing with action-value functions, we must adapt the TD algorithm. This adaptation is the SARSA algorithm (State-Action-Reward-State-Action) and its update function is shown in equation A.8.

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A)) \quad (\text{A.8})$$

where we define the TD error in equation A.9.

$$\delta = R + \gamma Q(S', A') - Q(S, A) \quad (\text{A.9})$$

If we take the approach of $\text{TD}(\lambda)$, we can use SARSA as a along a n-step action-value estimator, instead of just one look-ahead estimator.

- For $n = 1 \rightarrow q_t^1 = R_{t+1} + \gamma Q(S_{t+1})$
- For $n = 2 \rightarrow q_t^2 = R_{t+1} + \gamma R_{t+2} + \gamma Q(S_{t+1})$
- For $n = k \rightarrow q_t^k = R_{t+1} + \gamma R_{t+2} \dots \gamma^{k-1} R_{t+k} + \gamma^k Q(S_{t+k})$

And using the same weighted average using the λ parameter, we can obtain the $\text{SARSA}(\lambda)$ algorithm, as shown in equation A.10

$$q_t^\lambda = (1 - \lambda) \sum_{n=1}^k \lambda^{n-1} q_t^n \quad (\text{A.10})$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(q_t^\lambda - Q(S_t, A_t)) \quad (\text{A.11})$$

A.1.4 Off-Policy Control:

Dealing with just one policy, π , may limit the scope of learning. To put a easy example, it would be as if humans would only learn following a single philosophy. It is not a bad approach, but as we add new perspectives to our learning process, better outcomes come out of it (i.e. work, university, relationships). Off-policy learning, among other motives, was introduce to make the agent more flexible in the learning process, taking into account different perspectives and explore while learning the optimal policy. In off-policy learning, usually there are two policies: the target policy π that is used to compute the Q values, and the behavioural policy μ , which usually has an exploratory component to it.

One of the most famous algorithms that implements off-policy learning is Q-learning [56]. It is mainly divided in three blocks:

1. The next action A_{t+1} is selected following the behaviour policy $\mu(A|S_t)$
2. The alternate action A' is selected following the target policy $\pi(A|S_t)$
3. Use the update equation (eq. A.12) to update the action-values Q .

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \max_{a'} \gamma Q(S_{t+1}, a') - Q(S_t, A_t)] \quad (\text{A.12})$$

A.1.5 Actor-Critic Methods

Currently, the most common policy gradient methods are based on actor-critic methods. The critic (action-value function) makes evaluations of the actor's actions using function approximation $\hat{Q}_w(s, a)$. The actor tries actions and optimize them in the direction the critic proposes. This is formalized in equation A.13 where the approximation of the policy gradient takes into account the action-value function as part of the update.

$$\Delta\theta_t = \alpha(\nabla_\theta \log \pi_\theta(s, a) \hat{Q}_w(s, a)) \quad (\text{A.13})$$

One thing to note is that in this method we have two approximators, hence, two set of parameters, w and θ . The most recent actor critic methods follow this principle, but add different optimization tools to obtain more stable and robust results in the policy learning process.

- Soft Actor-Critic (SAC) [57]: SAC is another off-policy actor-critic algorithm that incorporates maximum entropy reinforcement learning. It maximizes the expected return while also maximizing entropy, leading to policies that are more exploratory and robust. Right now is the state-of-the-art method for policy optimization.
- Proximal Policy Optimization (PPO) [58]: PPO is an actor-critic method that addresses the limitations of previous policy gradient methods. This is done by limiting the policy update to be close to the previous policy, helping to stabilize training and preventing large policy updates which may lead to performance degradation.
- Deep Deterministic Policy Gradient (DDPG) [59]: DDPG is an off-policy actor-critic algorithm specifically designed for continuous action spaces. This are spaces where the set of actions may be infinite, and a greedy (max) policy do not work. It learns a deterministic policy function using deep neural networks to approximate the Q-function and the policy.

Appendix B

Attention Mechanism

B.1 Understanding the attention mechanism

The attention mechanism appeared as a learnable alignment method for the translation task. This sequence to sequence problem had a weakness in the recursive neural network set-up. Up until that point, the way this kind of problem was addressed was by using a bidirectional RNN (usually a bi-LSTM) in the encoder, that tried to compress the information from the whole sequence in the hidden state vector from the LSTM. This was done forward and backwards to then concatenate the hidden state vectors from both LSTMs into one joint vector. This is defined in equation B.1

$$\mathbf{h}_1, \dots, \mathbf{h}_t = \text{bi-RNN}(x_1, \dots, x_t) \quad (\text{B.1})$$

where \mathbf{h}_i is computed like shown in equation B.2, where the notation $\{\cdot; \cdot\}$ represents the concatenation between two vectors.

$$\begin{aligned} \vec{h}_i &= RNN_{forward}(x_i) \\ \overleftarrow{h}_i &= RNN_{backward}(x_i) \\ \mathbf{h}_i &= \{\vec{h}_i; \overleftarrow{h}_i\} \end{aligned} \quad (\text{B.2})$$

After that, the encoder, usually another RNN, would use the joint vector \mathbf{h}_i to "decode" the sequence into the desired output (i.e. the translated version of the input). As one could sense, compressing the information of big sequences becomes a problem when the hidden vector is of fixed size. What the attention mechanism proposed, was a way to dynamically change the hidden vector as the sequence was processed and new context from the input was acquired.

Bahdanu *et al.* used the same set-up in the encoder in [14], but introduced new modifications at the decoder to make the hidden vector change over the time-steps of the sequence. To do so, they introduced the attention block, where the context vector \mathbf{c}_t is computed for each time step t . This vector represents the relationship between the current output symbol and each term that belongs to the input sequence. Details on how the context vector \mathbf{c}_t is computed are at equation B.3

$$c_t = \sum_{j=1}^T \alpha_{tj} \cdot \mathbf{h}_j \quad (\text{B.3})$$

Where α_{tj} is a scalar that represents the weight of the hidden state vector \mathbf{h}_j over the final context vector in time-step t. As one can interpret from equation B.3, α_t contains the information for all the input tokens processed up to the j-th element, hence, it will change its values for each time-step. Also, to prevent the decoder to "look ahead" of the sequence, the inputs tokens that are after the j-th element in the sequence will be masked, so that the attention block cannot take them into account. The computation of α_{tj} is showed in equation B.4, where s_{t-1} is the hidden state of the decoder at time-step t-1 and f is a learnable function (i.e. a neural network) that outputs the logit (or energy, as they call it in the original work) e_{tj} , that ponders how much correlation exists between vectors h_j and s_{t-1} . This correlation is then normalized by using the softmax function, producing the value α_{tj} .

$$\begin{aligned}\alpha_{tj} &= \frac{\exp(e_{tj})}{\sum_{k=1}^T \exp(e_{tk})} \\ e_{tj} &= f(s_{t-1}, h_j)\end{aligned}\tag{B.4}$$

Finally, after all the alignments are computed, the RNN from the decoder outputs the most probable symbol y_t at the current time-step t (equation B.5).

$$\mathbb{P}(y_t | y_{t-1}, \dots, y_1, x) = RNN_{decoder}(c_t)\tag{B.5}$$

B.2 From attention to self-attention

Results from the attention encoder decoder architecture were a leap of performance with respect to previous work, but they still failed to perform correctly as the number of tokens in the input sequence increased. To tackle this problem, Vaswani *et al.* proposed a different approach. Instead of the neural network processing one token at each time-step and compressing the information in a single hidden state vector iteratively, the sequence is processed as a whole, and each of the input tokens compute how relevant the other tokens are with respect to themselves. This mechanism is called self-attention.

To implement this mechanism, the authors propose to operate with three terms, the query Q, the keys K and the values V. These values are computed by projecting the original embeddings from the input $X \in \mathbb{R}^{n \times d}$, where n is the number of tokens in the input sequence and d is the embedding dimension, to the three matrices such that $Q = XW^Q$, $K = XW^K$ and $V = XW^V$. For clarification, W^Q , W^K and $W^V \in \mathbb{R}^{d \times d_k}$, where d_k represents the dimension of the new space where X is projected to compute Q, K and V.

To follow the same terminology used in section B.1, we are going to dissect the compact form of the attention defined in equation B.6 into the several parts so that resembles the way that the "classic" attention mechanism is computed.

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^\top}{\sqrt{d_k}} \right) V\tag{B.6}$$

In section B.1 we explained how the context vector c_j contained the compressed information of the pondered sum from all the input tokens up to the j-th element of the sequence. On a similar fashion, the self-attention mechanism computes the context vector by projecting the j-th token of the input sequence, x_j , into a new space using W^V and then pondering it with respect to the rest of the i-th elements from the sequence by multiplying them times α_{ij} . As in the "classic" attention, α_{ij} is the normalized correlation between the i-th element and the

j-th, and it is done by applying the softmax function, as shown in equation B.7, but the way e_i is computing differs.

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^n \exp(e_{ik})} \quad (\text{B.7})$$

To compute the e_i , the self-attention formula proposes the scaled-dot product as the compatibility function. This function aims to give a magnitude of how correlated are two vectors from the input, since the dot product is greater the closer two vectors are on an euclidean space. Following this principle, we assume that the tokens that are similar semantically, will be closer in the embedding space, thus producing a greater value. The "scaled" part (i.e. $\frac{1}{\sqrt{d_z}}$) tries to stabilize the softmax function in terms of saturation, by preventing the values of the vectors from getting too big. This function is showed in equation B.8, where \cdot represents the dot-product between the i-th element in the query space and the j-th element in the key space, projected by the matrix-vector multiplication of their respective query and key projection matrices.

$$e_{ij} = \frac{(x_i W^Q) \cdot (x_j W^K)}{\sqrt{d_z}} \quad (\text{B.8})$$

Finally, to compute the attention-pondered final values z_i , we will multiply the weight values α_{ij} by each of the elements of the input X after they are projected into the value space. Finally, we add them up, hence obtaining equation B.9.

$$e_{ij} = \sum_{j=1}^n \alpha_{ij} (x_j W^V) \quad (\text{B.9})$$

As a final reflection, if we pay a closer look to equations B.4 and B.8, we see that the mechanism is practically the same, where h_i are the keys, and s_{t-1} are the queries and the operations between them are practically the same, since h_i and s_{t-1} are the encoded information from the sequence in the embedded space. Something similar occurs with equations B.9 and B.3 where the value matrix is equivalent the h_j vector at the input of the decoder. Here we can see that the relationship between the attention decoder and the self-attention mechanism from the transformer model lie under the same principles, compute which parts of the input are more relevant in terms of context to produce the appropriate output.

Appendix C

Attention-models code analysis

C.1 Attention-based models: Vision Transformer

The aim of this work is to test out if there is any explainability in the decision making that agents perform when the value function is an approximator, which is an ideal set-up for DQN learning. For this section we will discuss the implementation of the attention-based models (i.e. vision transformers) that were used to carry out our experiments. First, we will talk about the Vision Transformer, discussing about the relevant aspects involved from the intuition to the actual implementation, that we took from [42].

In several implementations for the vision transformer, we see that one of the most important things is for the model to be flexible to different configurations, where the number of blocks, embedding dimension or the number of heads in the multi-head attention block changes. This leads to easier ways to try out different configurations for the training process. For this section, first we are going to go over the constructor of the model and discuss its several parts. After that we will delve in the implementation of the most relevant.

C.1.1 Constructor

The constructor of the ViT is portrayed in listing C.1. We can see that the class is flexible to parameterizations, since we can specify the image size, the patch size, the channel of the input or the embedding dimension. In general terms, the constructor initializes the patch embedding module. This component is in charge of taking the input image and extract the patches from the spatial coordinates while enlarging the channel dimension from channels to the embedding dimension. After that, it declares two essential parameters for the ViT: the class token and the positional embedding, which both of them are self learnable parameters that the network adjust during training. The next main component is a sequence of ViT blocks, which are implementations of the transformer encoder which are stacked one on top of each other. Finally, we have the classification head, which maps from the embedding dimension of the class token vector to the output dimension of the network, producing the corresponding Q value of an action as an output. We would also like to notice that for this implementation to work with the DQN and DDQN algorithm, we made some minor adaptations to the code, such as adding some additional dense layers to map from the feature space to the action space.

```
1 def __init__(self, img_size=224, patch_size=16, in_chans=3,  
             num_classes=0, embed_dim=768, depth=12,
```

```

2     num_heads=12, mlp_ratio=4., qkv_bias=False, qk_scale=None, drop_rate
3         =0., attn_drop_rate=0.,
4     drop_path_rate=0., norm_layer=nn.LayerNorm, **kwargs):
5     super().__init__()
6     self.num_features = self.embed_dim = embed_dim
7
8     self.patch_embed = PatchEmbed(
9         img_size=img_size, patch_size=patch_size, in_chans=in_chans,
10        embed_dim=embed_dim)
11    num_patches = self.patch_embed.num_patches
12
13    self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
14    self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + 1,
15        embed_dim))
16    self.pos_drop = nn.Dropout(p=drop_rate)
17
18    dpr = [x.item() for x in torch.linspace(0, drop_path_rate, depth)] #  
      stochastic depth decay rule
19    self.blocks = nn.ModuleList([
20        Block(
21            dim=embed_dim, num_heads=num_heads, mlp_ratio=mlp_ratio, qkv_bias=
22                qkv_bias, qk_scale=qk_scale,
23            drop=drop_rate, attn_drop=attn_drop_rate, drop_path=dpr[i],
24                norm_layer=norm_layer)
25        for i in range(depth)])
26    self.norm = norm_layer(embed_dim)
27
28    # Classifier head
29    self.head = nn.Linear(embed_dim, num_classes) if num_classes > 0 else
30        nn.Identity()
31
32    trunc_normal_(self.pos_embed, std=.02)
33    trunc_normal_(self.cls_token, std=.02)
34    self.apply(self._init_weights)

```

Listing C.1: ViT model initialization

C.1.2 Patch Embedding

The patch embedding module is of great importance, since it is what transforms the visual data into something that is "consumable" for the ViT. The code of the patch embedding is in listing C.2. To perform the patch projections, they use a trick leveraging the 2D convolutional operator, where they specify the filter size and the stride as the size of the patch. This ensures that the projections are non-overlapping and reduced in the spatial dimension. Additionally, since we want the channels to be projected from the `in_channels` dimension to the embedding dimension, the number of filters specified in the 2D convolution operator is the same as the embedding dimension. With this trick, the implementation is more efficient and readable, providing the embedded patches from the original image.

```

1  class PatchEmbed(nn.Module):
2      """
3          Image to Patch Embedding
4      """
5
6      def __init__(self, img_size=224, patch_size=16, in_chans=3, embed_dim
7          =768):
8          super().__init__()

```

```

7     num_patches = (img_size // patch_size) * (img_size // patch_size)
8     self.img_size = img_size
9     self.patch_size = patch_size
10    self.num_patches = num_patches
11
12    self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size,
13                          stride=patch_size)
14
15    def forward(self, x):
16        B, C, H, W = x.shape
17        x = self.proj(x).flatten(2).transpose(1, 2)
18        return x

```

Listing C.2: Patch Embedding module

C.1.3 ViT encoder block

Once we have our embedded image, we can proceed to process it using the ViT encoder blocks. The code of a single block is depicted in listing C.3. The input of a single block is either the embedded input image of the previous block output, to which the self attention layer will be applied, or the original image embedded. We will delve into the implementation of the attention layer after, but for now, the only thing we ought to know is that the attention layer returns the input embedding pondered by the importance of each patch. After that, it applies an projection to a layer with four times the embedding dimension to then apply a dropout layer. Since these models are very deep, in the forward pass we see that a residual connection [60] is implemented in lines 17 and 18, that eases the gradient flow in the back-propagation stage.

```

1  class Block(nn.Module):
2      def __init__(self, dim, num_heads, mlp_ratio=4., qkv_bias=False,
3                   qk_scale=None, drop=0., attn_drop=0.,
4                   drop_path=0., act_layer=nn.GELU, norm_layer=nn.LayerNorm):
5          super().__init__()
6          self.norm1 = norm_layer(dim)
7          self.attn = Attention(
8              dim, num_heads=num_heads, qkv_bias=qkv_bias, qk_scale=qk_scale,
9              attn_drop=attn_drop, proj_drop=drop)
10         self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.
11             Identity()
12         self.norm2 = norm_layer(dim)
13         mlp_hidden_dim = int(dim * mlp_ratio)
14         self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim,
15                       act_layer=act_layer, drop=drop)
16
17     def forward(self, x, return_attention=False):
18         y, attn = self.attn(self.norm1(x))
19         if return_attention:
20             return attn
21         x = x + self.drop_path(y)
22         x = x + self.drop_path(self.mlp(self.norm2(x)))
23
24     return x

```

Listing C.3: ViT blocks implementation

C.1.4 ViT Attention Module

The attention blocks are the core functionality of this model. The code from the implementation is in listing C.4. In the constructor of the module, we can see that to compute the dimension of each head, it divides the embedding dimension by the number of heads we are going to apply. After that, it defines the weight matrices of the query, key and value from the self-attention module. To do so, it uses another trick to reduce the number of the layer's weights, by multiplying by three the embedding dimension, and then rearranging the tensor, so the highest order dimension is the one which contains the query, key and value values of the attention layer. After that it performs the self-attention operation, according to the equation 2.17, obtaining the pondered embeddings. Finally, it performs some additional linear projections to obtain the final embedding. One thing that is of great use from this ViT block is that, it also returns the attentions maps from the attention blocks.

```

1  class Attention(nn.Module):
2      def __init__(self, dim, num_heads=8, qkv_bias=False, qk_scale=None,
3          attn_drop=0., proj_drop=0.):
4          super().__init__()
5          self.num_heads = num_heads
6          head_dim = dim // num_heads
7          self.scale = qk_scale or head_dim ** -0.5
8
8          self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
9          self.attn_drop = nn.Dropout(attn_drop)
10         self.proj = nn.Linear(dim, dim)
11         self.proj_drop = nn.Dropout(proj_drop)
12
13     def forward(self, x):
14         B, N, C = x.shape
15         qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, C // self.
16             num_heads).permute(2, 0, 3, 1, 4)
17         q, k, v = qkv[0], qkv[1], qkv[2]
18
18         attn = (q @ k.transpose(-2, -1)) * self.scale
19         attn = attn.softmax(dim=-1)
20         attn = self.attn_drop(attn)
21
22         x = (attn @ v).transpose(1, 2).reshape(B, N, C)
23         x = self.proj(x)
24         x = self.proj_drop(x)
25         return x, attn

```

Listing C.4: Attention module for the ViT model

C.1.5 The forward method

With these main modules from the ViT explained, we can now address the `forward` method from the ViT class, which is portrayed in listing C.5. First, the `forward` method calls `prepare_tokens`, which is a function that encapsulates the patch embedding functionality plus the initialization of the positional embedding and the concatenation of the token class to the patch embedding tensor, as shown in figure 2.5b. After that, the positional embedding is added to the embedded tensor, giving additional context on how the patches are arranged, to then be passed to the ViT blocks. The ViT blocks are held into a `ModuleList` type of object from PyTorch's library. This enables creating lists where each element is a `nn.Module`

which can be tracked down by PyTorch’s graph engine to perform forward and backward passes. The code iterates over the list of blocks, to then obtain a normalized embedding using a `NormLayer` module. Finally, to project from the embedding class to the action space, we perform a linear projection and obtain the class token indexing by `x[:, 0]`.

```

1  def prepare_tokens(self, x):
2      B, nc, w, h = x.shape
3      x = self.patch_embed(x)    # patch linear embedding
4
5      # add the [CLS] token to the embed patch tokens
6      cls_tokens = self.cls_token.expand(B, -1, -1)
7      x = torch.cat((cls_tokens, x), dim=1)
8
9      # add positional encoding to each token
10     x = x + self.interpolate_pos_encoding(x, w, h)
11
12     return self.pos_drop(x)
13
14 def forward(self, x):
15     x = self.prepare_tokens(x)
16     for blk in self.blocks:
17         x = blk(x)
18     x = self.norm(x)
19     x = self.head(x)
20
21     return x[:, 0]

```

Listing C.5: ViT forward method

C.2 Attention-based models: SWIN Transformer

In this section we will do something similar to section C.1, where first we will talk about the intuition the changes with respect to the vision transformer, and then we will go over the implementation of those changes, using the code from [18], since this model is actually more complex in the development aspect than the ViT. Another thing to note is that we will not go over the whole code used and implemented for these models, instead, we will give a comprehensive review of the implementation and delve only into the most critical aspects of them.

The SWIN transformer emerged as a version of the ViT that takes into account hierarchical features, resembling the convolutional neural networks approach to solve vision problems. It is composed from different stages, that hold different blocks. Additionally, the model introduces spatial dimension reduction, which is a big change in terms of how the model interacts and manipulates the data. Additionally, it introduces the concepts of windows in order to reduce the computational overhead of the self-attention operation. In listing C.6 we show a reduced version of the original implementation.

C.2.1 Constructor

One of the main changes that the SWIN Transformer introduces is the positional embedding, which is relative between patches, instead of absolute as the ViT. The main reason behind this is the fact that the patches are now within windows, and the attention is computed within those windows, so each patch should have some kind of notion on where the other patches

inside its window are. The code for the constructor is define in listing C.6. First, the declaration of the SWIN layers follows exactly the same logic as the ViT, but with a minimal difference. According to figure 2.6b from section 2.2.4 we want for each SWIN stage to have different SWIN transformer layers, and, at the end, a patch merging transformation to reduce the spatial dimension. To do so, the BasicLayer module contains a flexible set-up where parameters such as number of heads, the depth for each layer or the window size may vary depending on the stage where is declared. The implementation is also flexible to a variable number of BasicLayer layers, since it uses the `ModuleList` object.

```

1  class SwinTransformer(nn.Module):
2
3      def __init__(self, img_size=224, patch_size=4, in_chans=3,
4          num_classes=1000, embed_dim=96, depths=[2, 2, 6, 2], num_heads=[3,
5              6, 12, 24], window_size=7, mlp_ratio=4., qkv_bias=True, qk_scale=
6              None, drop_rate=0., attn_drop_rate=0., drop_path_rate=0.1,
7              norm_layer=nn.LayerNorm, ape=False, patch_norm=True,
8              use_checkpoint=False, **kwargs):
9          super().__init__()
10
11      # ... Already explained variables in ViT transformer + Patch
12      # Embedding
13
14      # absolute position embedding
15      if self.ape:
16          self.absolute_pos_embed = nn.Parameter(torch.zeros(1, num_patches,
17              embed_dim))
18          trunc_normal_(self.absolute_pos_embed, std=.02)
19
20      # ... Already explained variables in ViT transformer
21
22      # build layers
23      self.layers = nn.ModuleList()
24      for i_layer in range(self.num_layers):
25          layer = BasicLayer(dim=int(embed_dim * 2 ** i_layer),
26              input_resolution=(patches_resolution[0] // (2 ** i_layer),
27                  patches_resolution[1] // (2 ** i_layer)),
28              depth=depths[i_layer],
29              num_heads=num_heads[i_layer],
30              window_size=window_size,
31              mlp_ratio=self.mlp_ratio,
32              qkv_bias=qkv_bias, qk_scale=qk_scale,
33              drop=drop_rate, attn_drop=attn_drop_rate,
34              drop_path=dpr[sum(depths[:i_layer]):sum(depths[:i_layer + 1])],
35              norm_layer=norm_layer,
36              downsample=PatchMerging if (i_layer < self.num_layers - 1) else None,
37              use_checkpoint=use_checkpoint)
38          self.layers.append(layer)
39
40      # ... Already explained variables in ViT transformer
41
42      self.apply(self._init_weights)

```

Listing C.6: SWIN Transformer constructor

C.2.2 BasicLayer

The `BasicLayer` module, defined in listing C.7, has inside of it the functionality that goes inside of every stage of the SWIN transformer. To do so, the class attribute `self.blocks` is declared as a `ModuleList` where the `SwinTransformerBlock` instances are declared. Since at the end of some stages, the patch merging module is applied, the `down_sample` boolean flag indicates whether that stage performs down-sampling via patch merging or not. The forward method is pretty straight-forward, as the code first iterates through the `SwinTransformerBlock` instances and then down-samples the spatial resolution of the returned embeddings if needed.

```

1  class BasicLayer(nn.Module):
2
3      def __init__(self, dim, input_resolution, depth, num_heads,
4          window_size,
5          mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0.,
6          drop_path=0., norm_layer=nn.LayerNorm, downsample=None,
7          use_checkpoint=False):
8
9          super().__init__()
10         self.dim = dim
11         self.input_resolution = input_resolution
12         self.depth = depth
13         self.use_checkpoint = use_checkpoint
14
15         # build blocks
16         self.blocks = nn.ModuleList([
17             SwinTransformerBlock(dim=dim, input_resolution=input_resolution,
18                 num_heads=num_heads, window_size=window_size,
19                 shift_size=0 if (i % 2 == 0) else window_size // 2,
20                 mlp_ratio=mlp_ratio,
21                 qkv_bias=qkv_bias, qk_scale=qk_scale,
22                 drop=drop, attn_drop=attn_drop,
23                 drop_path=drop_path[i] if isinstance(drop_path, list) else drop_path,
24                 norm_layer=norm_layer)
25             for i in range(depth)])
26
27         # patch merging layer
28         if downsample is not None:
29             self.downsample = downsample(input_resolution, dim=dim, norm_layer=
30                 norm_layer)
31         else:
32             self.downsample = None
33
34         def forward(self, x):
35             for blk in self.blocks:
36                 if self.use_checkpoint:
37                     x = checkpoint.checkpoint(blk, x)
38                 else:
39                     x = blk(x)
40                 if self.downsample is not None:
41                     x = self.downsample(x)
42
43             return x

```

Listing C.7: BasicLayer module that encapsulates the SWIN block functionality. It is the definition of a stage of the SWIN transformer, as defined in 2.6b

C.2.3 SwinTransformerBlock

In the `SWINTransformerBlock` we have the core functionality of the SWIN transformer. The first part of the constructor is defined in listing C.8. We can see that the main part is the `WindowAttention` class, which we will explain in the following section.

```

1  def __init__(self, dim, input_resolution, num_heads, window_size=7,
2   shift_size=0,
3   mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0.,
4   drop_path=0.,
5   act_layer=nn.GELU, norm_layer=nn.LayerNorm):
6   super().__init__()
7   self.dim = dim
8   self.input_resolution = input_resolution
9   self.num_heads = num_heads
10  self.window_size = window_size
11  self.shift_size = shift_size
12  self.mlp_ratio = mlp_ratio
13  if min(self.input_resolution) <= self.window_size:
14  # if window size is larger than input resolution, we don't partition
15  # windows
16  self.shift_size = 0
17  self.window_size = min(self.input_resolution)
18  assert 0 <= self.shift_size < self.window_size, "shift_size must in
19  0-window_size"
20
21  self.norm1 = norm_layer(dim)
22  self.attn = WindowAttention(
23   dim, window_size=to_2tuple(self.window_size), num_heads=num_heads,
24   qkv_bias=qkv_bias, qk_scale=qk_scale, attn_drop=attn_drop, proj_drop=
25   drop)

```

Listing C.8: First part of the SWIN block constructor

After that, in the second part of the constructor, displayed in listing C.9, handles the initialization of the window shifting for the `WindowAttention` model, which will be of crucial importance in order to properly arrange the context of the shifted parts of the image. To do so, it creates masks, which activate several parts of the image depending on the context, following the same scheme as explained in section 2.2.4. After that, it applies the window partitioning using the created mask. The window partition basically uses a projection of the patches that fall into the window size, which will be of use when computing the window attention.

```

1  self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.
2   Identity()
3  self.norm2 = norm_layer(dim)
4  mlp_hidden_dim = int(dim * mlp_ratio)
5  self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim,
6   act_layer=act_layer, drop=drop)
7
8  if self.shift_size > 0:
9  # calculate attention mask for SW-MSA
10 H, W = self.input_resolution
11 img_mask = torch.zeros((1, H, W, 1)) # 1 H W 1
12 h_slices = (slice(0, -self.window_size),
13 slice(-self.window_size, -self.shift_size),
14 slice(-self.shift_size, None))
15 w_slices = (slice(0, -self.window_size),
16

```

```

14     slice(-self.window_size, -self.shift_size),
15     slice(-self.shift_size, None))
16     cnt = 0
17     for h in h_slices:
18       for w in w_slices:
19         img_mask[:, h, w, :] = cnt
20         cnt += 1
21
22     mask_windows = window_partition(img_mask, self.window_size) # nW,
23     window_size, window_size, 1
24     mask_windows = mask_windows.view(-1, self.window_size * self.
25     window_size)
26     attn_mask = mask_windows.unsqueeze(1) - mask_windows.unsqueeze(2)
27     attn_mask = attn_mask.masked_fill(attn_mask != 0, float(-100.0)).
28     masked_fill(attn_mask == 0, float(0.0))

```

Listing C.9: Second part of the SWIN block model

Once the main components of the class are initialized, the `forward` (listing C.10) reshapes the input tensor from dimensions B, L, C, where B is the batch dimension, L is the length dimension that refers to the flattened number of patches from the input (usually as a product of the spatial dimensions: height and width) and C is the channels dimension. It reshapes the input from the flattened form to its spatial counter-part in line 7, to apply the window shifting using the `torch.roll` operator. Once the image is shifted, the `window_partition` method projects a window that holds inside a specified number of patches. For each of those windows, the attention module computes the self-attention between the patches that fall into their corresponding windows. With the attention weights applied for each window, the `window_reverse` method merges the window projections, returning the granularity of the input to patches. Finally, the `torch.roll` operator performs the shifting operation but in reverse, returning the input to its original form, but with each patch pondered by the attention weights computed for its corresponding window.

```

1 def forward(self, x):
2     H, W = self.input_resolution
3     B, L, C = x.shape
4     assert L == H * W, "input feature has wrong size"
5
6     shortcut = x
7     x = self.norm1(x)
8     x = x.view(B, H, W, C)
9
10    # cyclic shift
11    if self.shift_size > 0:
12        shifted_x = torch.roll(x, shifts=(-self.shift_size, -self.shift_size),
13                               dims=(1, 2))
14    else:
15        shifted_x = x
16
17    # partition windows
18    x_windows = window_partition(shifted_x, self.window_size) # nW*B,
19     window_size, window_size, C
20    x_windows = x_windows.view(-1, self.window_size * self.window_size, C
21     ) # nW*B, window_size*window_size, C
22
23    # W-MSA/SW-MSA
24    attn_windows = self.attn(x_windows, mask=self.attn_mask) # nW*B,
25     window_size*window_size, C

```

```
22
23 # merge windows
24 attn_windows = attn_windows.view(-1, self.window_size, self.
25     window_size, C)
26 shifted_x = window_reverse(attn_windows, self.window_size, H, W) # B
27     H' W' C
28
29 # reverse cyclic shift
30 if self.shift_size > 0:
31     x = torch.roll(shifted_x, shifts=(self.shift_size, self.shift_size),
32                     dims=(1, 2))
33 else:
34     x = shifted_x
35 x = x.view(B, H * W, C)
36
37 # FFN
38 x = shortcut + self.drop_path(x)
39 x = x + self.drop_path(self.mlp(self.norm2(x)))
40
41 return x
```

Listing C.10: Forward method for the SWIN Block module

C.2.4 Window Attention

The Window Attention module from the SWIN Transformer is probably what resembles the most from the ViT. The two main differences in the SWIN transformer attention are the relative positional embedding, which was first proposed for this architecture, and the masked attention which preserves the spatial context in shifted scenarios. The code is displayed in C.11. From lines 11 to 25 the relative positional index is initialized. The process to obtain it is quite convoluted, but the main intuition relies on understanding that, for each window, it is important for the patches to not only know where they are in the picture, but also know where the neighbouring patches are. By giving the relative position, the model provides a richer framework to understand not only the position of a single patch, but its position with respect to the rest of the patches inside the projected window. After this, the model uses the masks to understand which part of the window patches does it need to compute the attention. The rest of the module is pretty similar to the standard self-attention code explained in the ViT.

```
1 def __init__(self, dim, window_size, num_heads, qkv_bias=True,
2     qk_scale=None, attn_drop=0., proj_drop=0.):
3
4     super().__init__()
5     self.dim = dim
6     self.window_size = window_size # Wh, Ww
7     self.num_heads = num_heads
8     head_dim = dim // num_heads
9     self.scale = qk_scale or head_dim ** -0.5
10
11    # define a parameter table of relative position bias
12    self.relative_position_bias_table = nn.Parameter(
13        torch.zeros((2 * window_size[0] - 1) * (2 * window_size[1] - 1),
14                    num_heads)) # 2*Wh-1 * 2*Ww-1, nH
```

```

14     # get pair-wise relative position index for each token inside the
15     # window
16     coords_h = torch.arange(self.window_size[0])
17     coords_w = torch.arange(self.window_size[1])
18     coords = torch.stack(torch.meshgrid([coords_h, coords_w])) # 2, Wh,
19     # Ww
20     coords_flatten = torch.flatten(coords, 1) # 2, Wh*Ww
21     relative_coords = coords_flatten[:, :, None] - coords_flatten[:, None
22     , :] # 2, Wh*Ww, Wh*Ww
23     relative_coords = relative_coords.permute(1, 2, 0).contiguous() # Wh
24     # *Ww, Wh*Ww, 2
25     relative_coords[:, :, 0] += self.window_size[0] - 1 # shift to start
26     # from 0
27     relative_coords[:, :, 1] += self.window_size[1] - 1
28     relative_coords[:, :, 0] *= 2 * self.window_size[1] - 1
29     relative_position_index = relative_coords.sum(-1) # Wh*Ww, Wh*Ww
30     self.register_buffer("relative_position_index",
31     relative_position_index)
32
33     self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
34     self.attn_drop = nn.Dropout(attn_drop)
35     self.proj = nn.Linear(dim, dim)
36     self.proj_drop = nn.Dropout(proj_drop)
37
38     trunc_normal_(self.relative_position_bias_table, std=.02)
39     self.softmax = nn.Softmax(dim=-1)

```

Listing C.11: Window attention module constructor

C.2.5 The forward method

With all the main modules explained, we use the `forward` method as an overview of the model. The code is in listing C.12. The `forward_features` method handles the feature extraction from the input image to the compressed attention-weight representation that the SWIN transformer blocks output. These features go then to the output head, that maps the embedding dimension to the output dimension, in our case the actions available in the environment.

```

1 def forward_features(self, x):
2     x = self.patch_embed(x)
3     if self.ape:
4         x = x + self.absolute_pos_embed
5     x = self.pos_drop(x)
6
7     for layer in self.layers:
8         x = layer(x)
9
10    x = self.norm(x) # B L C
11    x = self.avgpool(x.transpose(1, 2)) # B C 1
12    x = torch.flatten(x, 1)
13    return x
14
15 def forward(self, x, head=None):
16
17     x = self.forward_features(x)
18     x = self.head(x)

```

19 `return x`

Listing C.12: Forward method for the SWIN Transformer

As we can see, the SWIN transformer is a step-up in comparison to the ViT, with more complex and efficient ways to compute and extract relevant features using attention and spatial reduction.

Appendix D

Schedulers

D.1 Gamma for the exponential equation

As explained in section 3.2.1.3, the equation we must solve for gamma is the following:

$$\epsilon_t = \epsilon_0 \gamma^{\lambda t} \quad (\text{D.1})$$

But, since we want the equation to be bounded between ϵ_0 and ϵ_f in the domain $[0, N]$, so we must substitute ϵ_t by ϵ_f . Additionally, since we want the value of $\epsilon_t = \epsilon_f$ when $t = N$, then we substitute it accordingly in the equation.

$$\epsilon_f = \epsilon_0 \gamma^{\lambda N} \quad (\text{Original equation}) \quad (\text{D.2})$$

$$\frac{\epsilon_f}{\epsilon_0} = \gamma^{\lambda N} \quad (\text{Divide by } \epsilon_0 \text{ in both sides}) \quad (\text{D.3})$$

$$\sqrt[\lambda N]{\frac{\epsilon_f}{\epsilon_0}} = \gamma \quad (\text{Exponentiation in both sides by } \frac{1}{\lambda N}) \quad (\text{D.4})$$

Hence, obtaining γ for the exponent function.

D.2 Gamma for the product of exponential equation

Introducing again the equation in 3.2.1.3, now we must solve for gamma:

$$\epsilon_f = \epsilon_0 \prod_{i=0}^N \gamma^{\lambda i} \quad (\text{D.5})$$

Lets begin with the procedure:

$$\epsilon_f = \epsilon_0 \prod_{i=0}^N \gamma^{\lambda i} \quad (\text{Original equation}) \quad (\text{D.6})$$

$$\frac{\epsilon_f}{\epsilon_0} = \prod_{i=0}^N \gamma^{\lambda i} \quad (\text{Divide by } \epsilon_0 \text{ in both sides}) \quad (\text{D.7})$$

$$\log_\gamma \left(\frac{\epsilon_f}{\epsilon_0} \right) = \log_\gamma \left(\prod_{i=0}^N \gamma^{\lambda i} \right) \quad (\text{Apply logarithms to both sides}) \quad (\text{D.8})$$

$$\log_\gamma \left(\frac{\epsilon_f}{\epsilon_0} \right) = \sum_{i=0}^N \log_\gamma \gamma^{\lambda i} \quad (\text{By the product property of logarithms}) \quad (\text{D.9})$$

$$\log_\gamma \left(\frac{\epsilon_f}{\epsilon_0} \right) = \sum_{i=0}^N \lambda i \log_\gamma \gamma \quad (\text{By the exp. property of logarithms}) \quad (\text{D.10})$$

$$\log_\gamma \left(\frac{\epsilon_f}{\epsilon_0} \right) = \sum_{i=0}^N \lambda i \quad (\text{Simplifying logarithms}) \quad (\text{D.11})$$

$$\log_\gamma \left(\frac{\epsilon_f}{\epsilon_0} \right) = \lambda \sum_{i=0}^N i \quad (\text{Taking the constant out of the sum}) \quad (\text{D.12})$$

Let $\sum_{i=0}^N i = k$, then we can compute it using the general summation formula:

$$k = \sum_{i=0}^N i = na_1 + d \frac{(n-1)n}{2} \quad (\text{D.13})$$

Where n is the final value, a_1 is the first value and d is the interval between the numbers of the sum. Then, by substituting D.13 in D.12, we get the following:

$$\log_\gamma \left(\frac{\epsilon_f}{\epsilon_0} \right) = \lambda k \quad (\text{D.14})$$

And by the definition of a logarithm $\log_b(x) = y \Leftrightarrow b^y = x$, we can express equation D.14 as:

$$\left(\frac{\epsilon_f}{\epsilon_0} \right) = \gamma^{\lambda k} \quad (\text{Applying the definition of a logarithm}) \quad (\text{D.15})$$

$$\sqrt[\lambda k]{\frac{\epsilon_f}{\epsilon_0}} = \gamma \quad (\text{Exponentiation of both sides by } \frac{1}{\lambda k}) \quad (\text{D.16})$$

Hence, obtaining γ for the product of exponents function.