UNIVERSITY OF PISA

Department of Computer Science

Master's Degree in Computer Science
Curriculum: Artificial Intelligence

Adaptively Combining Skill Embeddings for Reinforcement Learning Agents

Supervisors:

Prof. Davide Bacciu

Dott. Elia Piccoli

Candidate:

Giacomo Carfì

ACADEMIC YEAR 2023/2024

Abstract

Reinforcement Learning (RL) aims to learn agent behavioral policies by maximizing the cumulative reward obtained by interacting with the environment. Typical RL approaches learn an end-to-end mapping from observations to action spaces which define the agent's behavior. On the other hand, Foundational Models learn rich representations of the world which can be used by agents to accelerate the learning process. In this thesis, we study how to combine these representations to create an enhanced state representation. Specifically, we propose a technique called Weight Sharing Attention (WSA) which combines embeddings of different Foundational Models, and we empirically assess its performance against alternative combination approaches. We tested WSA on different Atari games, and we analyzed the issue of out-of-distribution data and how to mitigate it. We showed that, without fine-tuning of hyperparameters, WSA obtains comparable performance with state-of-the-art methods achieving faster state representation learning. This method is effective and could allow life-long learning agents to adapt to different scenarios over time.

Contents

Chapter 1

Introduction

Reinforcement Learning (Sutton and Barto, 1998) is a foundational paradigm of Machine Learning (ML). It consists of an agent learning its optimal behavior in an environment that can change over time. Its objective is to maximize the cumulative reward obtained by an agent resulting from a sequence of actions executed in its world. It formulates the learning process as a sequence of interactions with the environment and uses a trial-and-error approach to process data. Agents learn to solve a specific task based on their experiences with the world, they learn from the feedback of each action, i.e., the reward, and discover the best way to achieve their goal.

Over the past decades, RL algorithms have increasingly improved, achieving amazing results in many tasks. For example, RL can be applied in many real-world scenarios, including applications such as recommendation systems (Zhao et al., 2019), where RL agents can customize suggestions to individual users based on their interactions, or financial predictions (Zhang et al., 2019), where RL algorithms can optimize long-term returns by considering transaction costs and adapting to market shifts. Finally, it has been used in industry automation (Levine et al., 2018) to teach robots to act more efficiently than humans or for self-driving cars (Kiran et al., 2020). Some recent achievements in RL that are related to the topics that will be developed in this thesis are the works of Mnih et al. (2013, 2015), which started to explore the use of deep neural networks in games, specifically using the Arcade Learning Environment (ALE) (Bellemare et al., 2013). As a result of these works, although based on problems that may seem as basic as those in video games, new development perspectives have been created

and an increasing interest has turned toward Deep Reinforcement Learning (DRL) (Dong et al., 2020). In fact, DRL has produced substantial advancements in various research fields and applications, such as games, winning all Atari games at super-human performance with a single algorithm (Badia et al., 2020) or defeating pro players in games like GO, StarCraft II and DOTA (Silver et al., 2016; Vinyals et al., 2019; Berner et al., 2019), but also in robotics (OpenAI et al., 2019; Bousmalis et al., 2023) and lastly toward general agents for 3D environments including different data modalities (Team et al., 2024).

In a typical RL setting, agents receive as input the state of the environment without any additional information on the elements that characterize it. Learning consists of creating a map from input to action space i.e., a policy. A DRL agent trains a model from scratch to solve one specific task and, therefore, requires a lot of interaction with the environment to perform well, much more than the number of actions a human would need. End-to-end solutions for the learning process implicitly hide a significant effort related to understanding how to process the input representations in the best possible way. While one of the promises of deep learning algorithms is to automatically construct well-tuned representations, the same might not emerge from the training of deep RL agents. This is because agents learn how to solve the task while indirectly learning how to process and extract useful information from the raw input data. While this approach has been the predominant one in literature, it adds a layer of complexity to RL algorithms eventually requiring significant computational resources, and forcing the RL agent to solve a much more complex task than just policy acquisition.

Several works analyze the differences between the learning process of humans and RL agents, highlighting how prior knowledge can influence the learning curve (Tenenbaum, 2018; Dubey et al., 2018). Learning for humans, however, consists of acquiring new skills and incrementally improving those already known, leveraging prior knowledge, and adapting strategies based on experience. In new environments with few interactions, humans can make use of their prior abilities to accomplish basic tasks like distinguishing new elements or reading guidelines. Humans can understand the effects of their actions and this allows them to focus mostly on learning a good policy, rather than also learning how to interpret the environment. RL agents, in the same way as humans, could define the learning pattern by leveraging prior abilities, where with abilities we mean that the agent has access to several methods it already knows and it is already capable of

computing multiple tasks like image segmentation, predictions of how the environment will evolve, object tracking, and so on.

The idea behind this thesis is, therefore, to provide RL agents with abilities, i.e., recognizing objects, predicting future frames, etc. This thesis aims to study a way to encode skills in such a way as to incorporate prior knowledge into an RL agent. In this way, representation learning is done by leveraging prior knowledge, and during the training process, agents can focus more on learning the actual policy rather than learning how to extract meaningful information from the state.

In the development of this thesis, we will then try to answer two questions:

- How can we represent skills for an RL agent?
- Once skills are defined, how can we encode them and combine information coming from different skills or choose which one to best achieve the agent policy objectives?

We decided to represent skills using pre-trained models, capable of extracting a rich representation of the environment's state from different points of view. In short, compared to much bigger models known in the literature, we will use relatively small models and test them in simple environments. This is in such a way as to keep the agent's inference of deciding an action simple and not time-consuming. However, all the work can be scaled without additional effort in complex environments and using larger and more general models. Incorporating all the information in a meaningful way is therefore necessary. Among the other ways of combining information, we propose Weight Sharing Attention (WSA) as a combination module to incorporate different encodings from several FMs. We executed experiments across three different Atari games that are *Pong*, *Ms. Pacman*, and *Breakout*, and we studied the behavior of seven different combination modules. WSA achieves comparable performance with end-to-end solutions, while also adding scalability and explainability to RL agents. We show that without any fine-tuning of hyperparameters, our methodology can achieve competitive results with established RL end-to-end solutions. Moreover, we study the problem of out-of-distribution data, which presents itself as one of the main challenges for these approaches.

The thesis is organized as follows. Ch. 2 will introduce and explain the background of this work. Ch. 3 surveys related works and current state-of-the-art. Ch. 4 discuss our approach. Ch. 5 provides the experimental

analysis we conducted to test our method along with the results for each one, and finally, Ch. 6 provides the conclusions of the work and discusses possible future extensions.

Chapter 2

Background

In this chapter, we provide a brief overview of the main concepts that are necessary to understand the work presented in this thesis. First of all, we start by introducing the concept of Machine Learning and Deep Learning in 2.1. Then, in 2.2 we will talk in depth about one specific kind of learning paradigm, i.e., Reinforcement Learning, and in 2.3 we will talk about Deep Reinforcement Learning which is instead the main focus of this work.

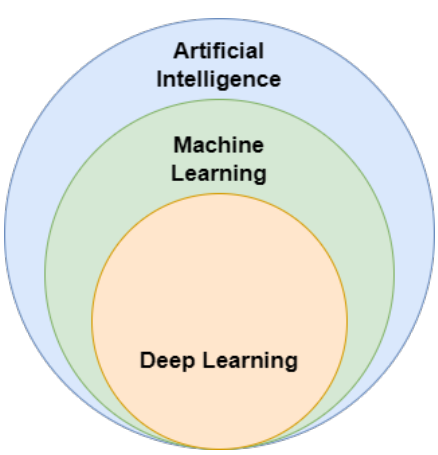
2.1 Machine Learning

Machine Learning, Fig. 2.1a, is the branch of Artificial Intelligence that focuses on developing models and algorithms that let computers learn from data and improve from previous experience without being explicitly programmed for every task. In simple words, ML teaches the systems to think and understand like humans by learning from data. ML finds application in many fields, including natural language processing (Devlin et al., 2018), computer vision (He et al., 2016), speech recognition (Hinton et al., 2012), email filtering (Carreras and Màrquez, 2001), medicine (Esteva et al., 2017), and many more

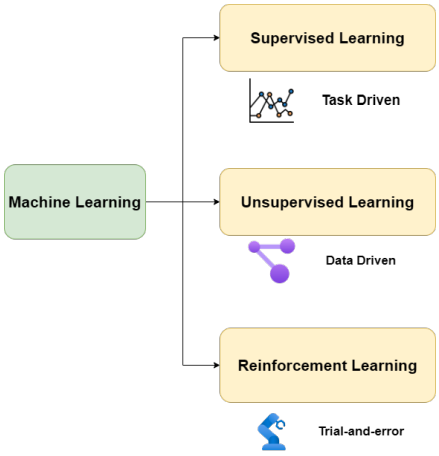
There are several types of ML family of algorithms (Fig.2.1b), each with its characteristics and applications. Some of the main types are Supervised Learning (Kotsiantis, 2007), Unsupervised Learning (Hastie et al., 2009), Self-Supervised Learning (Balestriero et al., 2023), and finally Reinforcement Learning (Sutton and Barto, 1998) A subset of ML is Deep Learning (DL) (LeCun et al., 2015), which focuses on training neural networks with

many layers.

We will talk about the different kinds of learning algorithms and DL in the following subsection, while since RL is the focus of this work we will dedicate a separate section.



(a) Artificial Intelligence hierarchy.



(b) Families of ML algorithms.

Supervised Learning

Supervised Learning is a branch of ML where the model learns from annotated examples. In this context, models are equipped with *labeled* data, meaning that each training example is paired with an output label. The goal is to learn a mapping function from inputs to outputs that can be used to predict the target for new, unseen inputs. The process of training a supervised learning model consists of two main phases: training and testing. During the training phase, the algorithm searches for patterns in the data that can be used to make predictions. After the training phase, the model can receive as input new unseen data and determine which label it belongs to, this is called the testing phase.

For example, consider to have a dataset of pairs consisting of flower images and their labels representing the flower species, we refer to this as training data or training set. The aim is to train a model on the training set that learns to predict the species of a flower given its image, based on the patterns learned from the training data like color, shape, and size. When the model outputs the predicted species of a flower image, it can be compared with the true label to evaluate the model's performance. The learning process

is iterative, and the model is updated based on the errors made during the training phase. If the predicted species is the same as the true label, the model has made a correct prediction, and it does not need to be updated, otherwise, it has made an incorrect prediction, and the model needs to update its parameters in order to obtain the correct prediction. Once the model is trained such that it recognizes most of the species of the flowers in the training set, it can be used to predict the species of a new flower image that was not present in the training set.

Two main categories of supervised learning are:

- Classification - The goal is to predict a discrete label. For example, classifying emails as spam or not spam, or recognizing handwritten digits. Classification algorithms learn how to map the input features to one of the predefined classes.
- Regression - The goal is to predict a continuous value. For example, predicting the price of a house given its features, or the temperature for a given day. Regression algorithms learn to map the input features to a continuous numerical value.

Over the years, many algorithms have been developed to solve supervised learning tasks, including Linear Regression (James et al., 2013), Support Vector Machines (Cortes and Vapnik, 1995), Decision Trees (Breiman et al., 1984), Naive Bayes (McCallum et al., 1998), and Neural Networks (LeCun et al., 1998).

Supervised Learning models can have high accuracy when they are trained on a huge amount of quality labeled data. Also, they can be used as pre-trained models, which saves time and resources when developing new models from scratch. Moreover, many supervised learning models like decision trees, are interpretable, meaning that it is possible to understand how the model makes predictions. This is very helpful in contexts like medicine, where it is important to understand the reasons behind the model's predictions. They have some limitations though, in fact, sometimes they need a huge amount of data to perform well, meaning that in the context where there is a lack of data, they may not be the best choice, meanwhile, in the context where there is a huge amount of data, they can be time-consuming to process all the data. Also, they may suffer from *overfitting* problem, which means that the model learns the training data too well, and it is not able to generalize on unseen data.

Unsupervised Learning

Unsupervised Learning paradigm is the opposite of Supervised Learning. It is a technique in which an algorithm discovers patterns and relationships using unlabeled data, i.e., it does not require labeled data as target outputs. The primary goal of Unsupervised learning is to discover hidden patterns, similarities, or clusters within the data, which can then be used for various purposes, such as data exploration, visualization, dimensionality reduction, and more.

For example, given a dataset of customer purchase history, the goal is to group customers based on their purchasing behavior. The algorithm will group customers with similar purchasing behavior into clusters, and this information can be used to target specific groups of people with personalized marketing campaigns.

The main categories of unsupervised learning are:

- Clustering - This is the process of grouping data points into clusters based on their similarities.
- Association Rule learning - It is a method used for finding relationships between variables in a large database. A typical example of the Association Rule is the Market Basket Analysis which determines the set of items that occur together like people who buy a specific item also tend to purchase another item.

Many algorithms have been developed through the years for clustering applications like K-means (MacQueen et al., 1967), Hierarchical Clustering (Johnson, 1967), or DBSCAN (Ester et al., 1996). Also, there are algorithms that use unsupervised learning for dimensionality reduction like PCA (Jolliffe et al., 1986). Finally, for association rule learning, there are algorithms like Apriori (Agrawal and Srikant, 1994).

Unsupervised Learning is really helpful in discovering hidden patterns and relationships when labels for data are not available. But it has some limitations, in fact, it is difficult to evaluate the performance of the model since there are no labels to compare the output with.

Self-Supervised Machine Learning

Self-supervised Learning paradigm falls in between supervised and unsupervised learning. As the name suggests, it is a technique where the model

learns from the data itself without human supervision. The training process is divided into two phases: pre-training and fine-tuning. In the pre-training phase, the model learns to solve a pretext task, which consists of generating labels from the data itself. This means that the model learns to extract useful feature representations from the data. In the fine-tuning phase, the extracted features are used to solve the downstream task, for example, classification or regression in a supervised learning setting.

For example, in the context of computer vision, given a dataset of images that have one part obscured or missing, the model can be trained to predict the missing part. This is a pretext task. Then, the same model used for the pretext task can be fine-tuned on a downstream task like image classification or object detection.

In the literature, there are many works that use self-supervised learning for various tasks, including image classification (Chen et al., 2020), object detection (He et al., 2020), and natural language processing (Devlin et al., 2018; Radford et al., 2018).

Self-supervised learning is really helpful when labeled data is missing, or difficult to obtain. It can be used to pre-train models on large amounts of unlabeled data and then fine-tune them on smaller labeled datasets. This is really helpful in the context of transfer learning, where the model learns from one task and then applies the knowledge to another task. However it has some limitations, in fact, it requires a lot of computational resources and time to train the model on large amounts of data, and it may not perform well on tasks that are very different from the pretext task.

Deep Learning

In recent years, deep learning has gained a lot of attention and has been successful in many fields. It can be trained in supervised, unsupervised, and self-supervised learning. It focuses on training neural networks with many layers, where each layer is responsible for extracting features from the input data at different levels of abstraction.

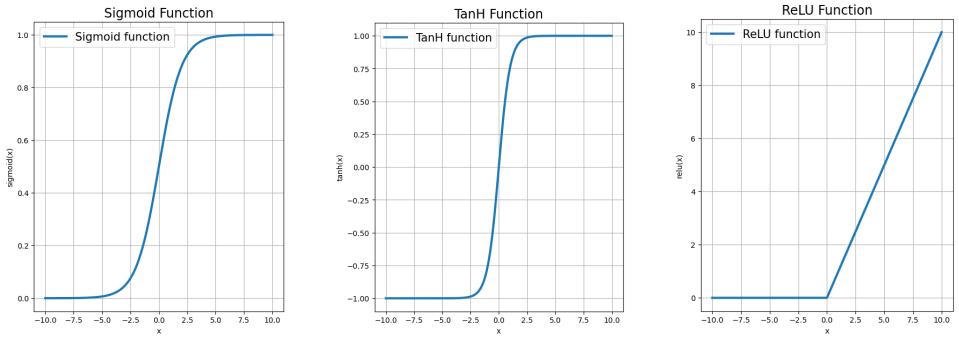
In particular, a Neural Network (NN) (Rumelhart et al., 1986) is an Artificial Intelligence technique that draws inspiration from the functioning of the human brain. Every NN consists of layers of interconnected nodes, which are called neurons. A neuron is a computational unit that takes multiple inputs and, to produce an output, a weighted summation of the inputs for some weights is computed. Also, neurons have an activation function f

that decides whether a neuron should be activated or not. Activation functions can be linear or non-linear functions, and they are used to introduce non-linearity in the model, which allows the model to learn more complex patterns in the data rather than linear ones. Examples of non-linear activation functions are the Sigmoid (Fig. 2.2a), the Rectified Linear Unit (ReLU) (Fig. 2.2c), and the Hyperbolic Tangent (TanH) (Fig. 2.2b).

In specific, the computation of a neuron can be expressed as:

$$\dot{y} = f(\sum_{i=1}^{n} w_i x_i + b) \tag{2.1}$$

Where $\dot{y}$ is the output of the neuron, f is the activation function, w_i are the weights that represent the learnable parameters, x_i are the inputs, and b is the bias term. Fig. 2.3 shows a picture of a single neuron.



(a) Sigmoid (b) TanH (c) ReLU

Figure 2.2: We show the plot of different activation functions. With TanH we refer to Hyperbolic Tangent, while with ReLU we refer to Rectified Linear Unit.

A NN can be seen as a composition of multiple neurons, where the output of one neuron is the input of the next one. **Multilayer Perceptron** (MLP) (Rumelhart et al., 1986) are defined as a sequential model that is composed of multiple layers of neurons. In specific, one *input layer* which is responsible for receiving the input data, one or more *hidden layers* that are responsible for processing the data in a way that encodes the information in a latent space, and finally an *output layer* which is responsible for producing the output information. More precisely, *feed-forward neural networks*, are

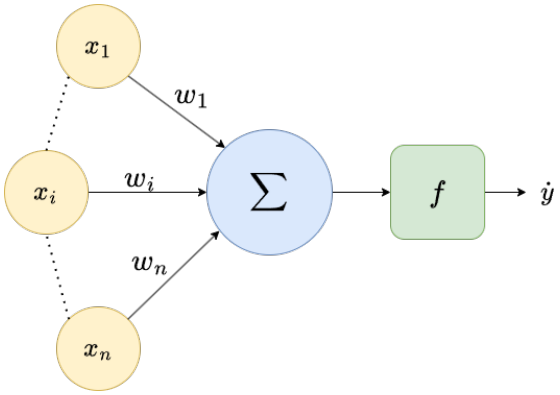


Figure 2.3: We show the image of a single neuron of a neural network. Considering having n input examples, the output $\dot{y}$ is computed by computing the weighted summation of the input first and then passing through an activation function f.

those in which information flows from the input layer to the output layer without any feedback connections, and *fully-connected neural networks* are those in which every neuron in one layer is connected to every neuron in the next layer. The mathematical representation of a feed-forward neural network is expressed in Eq. 2.2 while Fig. 2.4 shows a picture of a feed-forward neural network.

$$\hat{y} = f_n(f_{n-1}(\dots f_1(\sum_{i=1}^{n} w_i x_i + b_1) \cdots + b_{n-1}) + b_n) \tag{2.2}$$

The process of learning in NNs consists of changing the weights and biases of the model over time and relies on the **backpropagation** algorithm (Rumelhart et al., 1986) that comprises two main steps: forward pass and backward pass.

Assuming to have a dataset consisting of pairs (x, y) where x is the example y is the ground-truth target, in the forward pass, the input data is passed through the network, and the predicted output $\hat{y}$ is computed. Then the error committed by the model is calculated using a loss function which is a distance metric and measures the difference between the predicted output $\hat{y}$ and the ground-truth y.

The backward pass then consists of propagating the error back through the

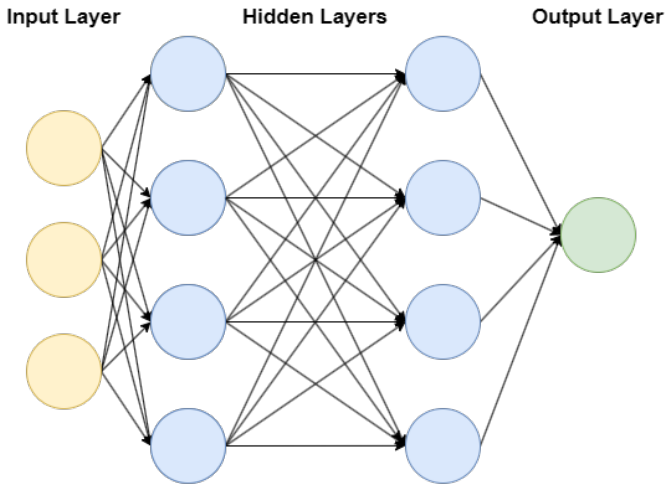


Figure 2.4: We depict an Artificial Neural Network scheme composed of an input layer in yellow, one or more hidden layers in blue, and one output layer in green. Each neuron is linked to all the neurons in the next layer through weighted connections.

network, from the output layer to the input one. It is responsible for updating the weights and biases proportionally to the error computed in order to minimize the error between the predicted output and the ground-truth. The learnable parameters are updated using the *gradient descent algorithm*, which is an optimization method that iteratively updates the parameters in the direction that minimizes the loss function. In neural networks, backpropagation is implemented using the chain rule to compute the gradients of the loss function with respect to the weights. Decomposing the computation of the gradient of the loss into simpler computations makes the computation of the gradients more efficient and allows the training of deep neural networks. In Fig. 2.5 it is possible to see an image that represents the gradient descent method, while in Algorithm 1 we provide the pseudocode of the gradient descent algorithm.

In literature, multi-layered neural networks are referred to as Deep Neural Networks (DNNs), and the process of training them is called Deep Learning.

Algorithm 1 Gradient Descent Algorithm

Initialize the weights w randomly
while not converged **do**
  Compute the predicted output $\hat{y}$
  Compute the loss L
  Compute the gradient of the loss w.r.t the weights $\nabla_w L$
  Update the weights $w = w - \alpha \nabla_w L$
end while

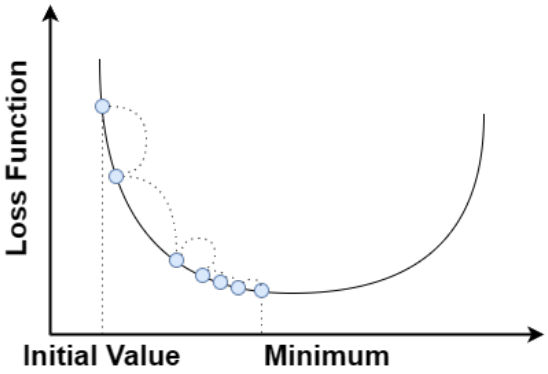


Figure 2.5: We depict the plot of the *loss function*. Considering to start from the initial value, each step of the gradient descent method leads to a point closer and closer to the minimum point.

2.2 Reinforcement Learning

Reinforcement Learning is a paradigm of ML works in a way that mimics the trial-and-error learning process that humans use to achieve their goals. The principal elements that form RL are the agent, the environment, and the reward signal.

The environment represents the world where the agent lives. The agent instead is the learner that interacts with it, and for each action that the agent makes, it obtains a reward that usually is a scalar number. RL focuses on training agents to make sequences of decisions in the environment to maximize the cumulative reward. In particular, in RL, there is no direct supervision. The only feedback that the agent receives is the reward signal that indicates how well it is doing and should be informative enough to let the agent learn the optimal behavior. Fig. 2.6 shows a picture of the RL

paradigm.

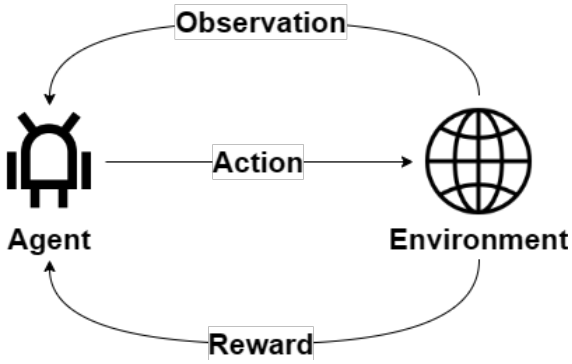


Figure 2.6: A schematic of an RL setting depicted. Agent explores the environment doing actions, for each action it receives a new observation and a reward.

Environments can be *fully observable* when the agent has access to the complete state of it, or *partially observable* when the agent has only partial information of the full environment. Specifically, in the presence of fully observable environments, the RL problem is formalized as a Markov Decision Process (MDP), which is a tuple consisting of (S, A, P, R, γ), where:

- S is the set of states that the agent can be in.
- A is the set of actions that the agent can take.
- P is the transition probability function, which defines the probability of transitioning from one state to another given an action.
- R is the reward function, which defines the reward that the agent receives when transitioning from one state to another.
- γ is the discount factor, which determines the importance of future rewards. If it is close to 0, the agent will consider only immediate rewards, while if it is close to 1, the agent will consider future rewards.

Both actions and states can be discrete when they are uniquely represented by a discrete value, or continue when they are represented by a continuous range of values.

It is possible to define the return of an agent in an MDP as the sum of

discounted rewards from a specific timestep t. In particular:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.3}$$

The principal components of an RL agent are three and in order are the policy π, the value function v, and the model of the environment m.

The policy π defines the behavior of the agent, it is a mapping from states to actions. Policy can be deterministic if it maps states to a single action, or stochastic if it maps states to a distribution over all possible actions. The policy can be represented as a table, a function, or a neural network, and in the context of MDPs, the policy is defined in Eq. 2.4, where P is the probability of taking action a considering to be in state s.

$$\pi(a|s) = P(a|s) \tag{2.4}$$

The value function v estimates the expected return that the agent can achieve starting from a given state and following a given policy π. It is used to evaluate how good a state is, and it is defined as:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s] = \mathbb{E}_\pi[G_t | S_t = s] \tag{2.5}$$

This is also called the **state-value function**. It exists an equivalent function that estimates the expected return that the agent can achieve starting from a given state, taking a specific action, and then following a given policy π. This is called the **action-value function**, and it is defined in Eq. 2.6. Value functions too can be represented as tables, functions, or neural networks.

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \end{aligned} \tag{2.6}$$

The model m is used to predict what the environment will do next. If an agent knows the model, it means that it knows what will be the next state given the current state and action (Eq. 2.7), and also, it can predict the next reward given the current state and action (Eq. 2.7).

$$P^a_{s,s'} = P(S_{t+1} = s' | S_t = s, A_t = a) \tag{2.7}$$

$$R^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] \tag{2.8}$$

In RL, there are two fundamental tasks, *prediction* and *control*. Prediction involves estimating the value function of an unknown MDP, while control involves finding the optimal policy.

The concept of learning in RL is based on the **Bellman Equations**, which form the basis for many RL algorithms. They are divided into two main categories: the *Bellman Expectation Equations* and the *Bellman Optimality Equations*. To estimate the goodness of a state, i.e., the value functions, it is possible to use the Bellman Expectation Equations for the state-value function and the action-value function, which provides a recursive relationship between the value to be in a state at a specific time and the expected return starting from the next state.

Bellman Expectation for the state-value function is defined in Eq. 2.9 where the expected return is decomposed into the immediate reward and the discounted value of the next state, and then the marginalization over all the possible actions is computed to obtain the expected value of the state in terms of the action-value function.

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] = \sum_{a \in A} \pi(a|s) q_\pi(s, a) \tag{2.9}$$

Similarly, the Bellman Expectation for the action-value function is defined in Eq. 2.10 where the expected return is decomposed into the immediate reward and the discounted value of the next state-action pair, and then the marginalization over the next states is performed to obtain the expected value of the action in terms of the state-value function.

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \\ &= R^a_s + \gamma \sum_{s' \in S} P^a_{s,s'} v_\pi(s') \end{aligned} \tag{2.10}$$

Now it is important to note that the Bellman Equation for the state-value function defined in Eq. 2.9 can be rewritten substituting the action-value function in it, obtaining:

$$v_\pi(s) = \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{s,s'}^a v_\pi(s')) \tag{2.11}$$

Similarly, the Bellman Equation for the action-value function defined in Eq. 2.10 can be rewritten substituting the state-value function in it, obtaining:

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{s,s'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s',a') \tag{2.12}$$

A policy π is defined to be better than or equal to a policy π' if an agent, following the policy π is expected to obtain a return that is greater than or equal to the one obtained by an agent following the policy π'. There is always at least one policy that is better than or equal to all other policies. This is an *optimal policy*. There may be more than one optimal policy, but all the optimal policies are denoted by π^*. For finding the optimal policy π^*, it is possible to use the **Bellman Optimality Equations** both for the state-value function and the action-value function. Supposing that the optimal action-value function $q_*(s,a)$ is given, i.e., a function that estimates correctly the expected return, it is possible to obtain the optimal policy simply by choosing always the action that maximizes the action-value function in a given state:

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_a q_*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

So the Bellman Optimality Equation to find the optimal state-value function is defined as follows:

$$v_*(s) = \max_a q_*(s,a) = \max_a R_{t+1}^a + \gamma \sum_{s'} P_{s,s'}^a v_*(s') \tag{2.14}$$

The Bellman Optimality Equation to find the optimal action-value function is as follows:

$$q_*(s,a) = R_s^a + \gamma \sum_{s'} P_{s,s'}^a v_*(s') = R_s^a + \gamma \sum_{s'} P_{s,s'}^a \max_{a'} q_*(s',a') \qquad (2.15)$$

Solving the Bellman Equations means finding the optimal policy.

In the world of RL, there exist many algorithms that can be distinguished into model-based algorithms, where the agent knows the environment, and model-free algorithms, where the agent does not need a model of the environment. Within model-free RL, some algorithms focus on optimizing the value function (value-based RL) (Sutton, 1988), other tries to optimize the policy (policy-based RL) (Sutton et al., 1999), and some of them both (Actor-Critic) (Konda and Tsitsiklis, 1999). In the following sections, we will explore briefly model-based and model-free RL algorithms, we will talk about how to approximate the value functions and the policy using neural networks. We will also talk about some of the most famous RL algorithms, including *Q-learning* (Watkins and Dayan, 1992), *Deep Q-Learning* (Mnih et al., 2013), and *Proximal Policy Optimization* (Schulman et al., 2017).

2.2.1 Model Based Reinforcement Learning

Model-based RL is more like planning, agents build a model of the environment and then use it to plan their actions. Model-based RL algorithms are based on the Bellman Equations and include algorithms like **Policy Iteration** (Howard, 1960) and **Value Iteration** (Bellman, 1966). These algorithms store information about the transition probabilities and the reward function in tabular form. While more recent works like **PlaNet** (Hafner et al., 2019) and **Dreamer** (Hafner et al., 2020) use neural networks to learn a model of the environment and then use it to plan their actions.

Policy Iteration is an algorithm that computes the optimal policy by iteratively applying the *Bellman Expectation Equation*. It alternates between policy evaluation, which computes the value function for a given policy using the Bellman Expectation Equation, and policy improvement, which computes the optimal policy given the value function acting greedily i.e., choosing the action that maximizes the action-value function in a given state. It is guaranteed to converge to the optimal policy, but it is computationally expensive.

Value Iteration is an algorithm that computes the optimal value function by

iteratively applying the *Bellman Optimality Equation*. There is no explicit policy here, its goal is to find an optimal policy π so, at each iteration, it computes the value function for all states, trying to improve the estimate of it. It is guaranteed to converge to the optimal value function.

2.2.2 Model Free Reinforcement Learning

In the model-free approach, a model of the environment is not provided. Agents learn the optimal policy without explicitly learning nor knowing the dynamics of the environment so there is no knowledge of the transition probabilities and the reward function. Agents learn by interacting with the environment and observing the rewards.

Model-free control RL algorithms can be divided into two main categories: **on-policy** and **off-policy** algorithms. On-policy means the agent learns the policy π while following the current policy, so it learns from experience sampled from π. Off-policy means the agent learns the policy π while following a different policy μ. This is very helpful in the context of *learning from imitation* or reusing experiences.

In the context of Off-policy learning, a well-known algorithm is **Q-learning** (Watkins and Dayan, 1992). Q-learning learns a policy π while interacting with the environment and updating each time the estimation of the optimal action-value function $q_*(s, a)$. In order to do this, it applies iteratively the Bellman Optimality Equation. Since a model of the environment is not known, it is not possible to compute the expected value of the next state, so Q-learning uses the concept of *Temporal Difference* (TD) learning (Sutton, 1988), which is a bootstrapping method, i.e., updates the current estimate of the value function using the next estimate of the value function. In particular *TD learning* computes the *TD error*, which is defined in Eq. 2.16 where $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ is called *TD target* and $Q(S_t, A_t)$ is the current estimate of the action-value function. In Q-learning, *TD target* and *TD error* are also called *Q target, and Q error*.

$$\delta = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \tag{2.16}$$

Q-learning agents learn the optimal policy π by acting greedily, i.e., choosing the action that maximizes the action-value function in a given state. So, the *Q-target*, in this case, is defined as the immediate reward plus the maximum action-value function of the next state. In Eq. 2.17 we show the Q-learning

update rule, while in Fig. 2.7 we show a picture of the Q-learning algorithm. It is important to notice that Q-learning values are stored in a table.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)] \quad (2.17)$$

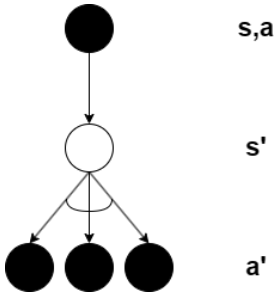


Figure 2.7: We show what is called the *Backup Diagram* of Q-Learning. In state s, action a is executed leading to state s'. Here all the possible actions are explored and only the action a' that returns the maximum reward is taken.

The next action instead is chosen following a *behavior policy* μ which is the ϵ-greedy policy. This means that with probability ϵ the agent chooses a random action, and with probability 1-ϵ the agent acts greedily and chooses the action that maximizes the action-value function. This is done in training in order to explore the environment using the random policy and avoid getting stuck in local optima. We report the pseudocode of the Q-learning algorithm in Algorithm 2. The Q-learning algorithm is guaranteed to converge to the optimal policy, but it may take a long time to converge.

2.3 Deep Reinforcement Learning

So far it has been shown how to solve the RL problem using tabular methods, i.e., the value function scores and the policy are represented as tables. These methods are intractable when the state space is large or continuous. In this section, we will talk about how value functions and policies can be predicted using neural networks as universal function approximators. The combination of DL techniques and RL is called Deep Reinforcement Learning (DRL) (Mnih et al., 2015).

Algorithm 2 Q-Learning Algorithm

Initialize $Q(s, a)$, $\forall s \in S, a \in A$ arbitrarily and $Q(\text{terminal state}, \cdot) = 0$
for each episode **do**
  Initialize S
  for each step of the episode **do**
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_{a'} Q(S', a') - Q(S, A)]$
    $S \leftarrow S'$
  end for
  Until S is terminal
end for

DRL has been successful in many tasks, including playing video games, controlling robots, and optimizing complex systems. Both prediction and control problems can be addressed using DRL. We will make distinctions between value function approximation and policy function approximation talking about policy gradient methods. We will provide an overview of two of the most famous DRL algorithms, i.e., **Deep Q-Learning** (DQL) and **Proximal Policy Optimization** (PPO) (Schulman et al., 2017).

2.3.1 Value Function Approximation

In the context of DRL, the value function can be approximated using different types of approaches, including linear function approximation, decision tree and in particular neural networks.

The goal is to find a parameter vector $\mathbf{w}$ that minimizes the MSE between the predicted value function $\hat{v}(s, \mathbf{w})$ and the true value function $v_\pi(s)$. This can be done using the *Gradient Descent* algorithm and can be accomplished in a batch or online way. The convergence though is not guaranteed for all algorithms and depends on the choice of the value function approximator (linear or non-linear).

Regarding batch methods, a well-known algorithm is the one called Deep Q-Learning. It is based on the Q-learning algorithm, but it uses neural networks to approximate the action-value function. Also, it uses the concept of ϵ-greedy policy as in Q-learning. Finally, *experience replay* is used in order to save the episodes of the agent while interacting with the environment. In particular, it consists of storing the agent's experiences in a replay buffer,

which is called D. The buffer will contain the agent's transitions, which are tuples consisting of (s, a, r, s'), where s is the initial state, a is the action taken, r is the reward received, and s' is the next state. To update the action-value function, the agent randomly samples a batch of transitions from the replay buffer, this transitions represent the ground-truth, and are used to compute the TD error. This is done in order to stabilize the learning process and avoid the correlation between the samples.

The neural networks involved in the DQL algorithm are two, the online network and the target network, and they are called Deep Q-networks (DQN). The online network is used to predict the current estimation of the action-value function, while the target network is used to compute the Q target.

The loss function is defined in Eq. 2.18 and it is the Mean Squared Error (MSE) between the predicted action-value function and the target action-value function. In the formula, D is the replay buffer, $\mathbf{w}$ are the weights of the online neural network, and in specific $\mathbf{w}^-$ are the weights of the target network, which are updated less frequently than the weights of the online network. The two different sets of parameters $\mathbf{w}$ and $\mathbf{w}^-$ are needed to improve the stability and the convergence of the training process.

$$L(\mathbf{w}) = \mathbb{E}_{(s,a,r,s')\sim D}[((r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-)) - Q(s, a, \mathbf{w}))^2] \tag{2.18}$$

The pseudocode of the DQL algorithm is reported in Algorithm 3.

2.3.2 Policy Gradients

As we have seen, the value function can be approximated using neural networks, but also the policy can be approximated using neural networks. The goal is, given a policy $\pi(a|s, \mathbf{w})$, to find the parameter vector $\mathbf{w}$ that maximizes the expected return. First of all, in Eq. 2.19 it is possible to define the objective function to maximize, where π_w is the policy parameterized by $\mathbf{w}$, and S_0 is the initial state.

$$J(\mathbf{w}) = \mathbb{E}_{\pi_w}[\sum_{t=0}^{\infty} \gamma^t R_t | S_0 = s] \tag{2.19}$$

According to the **Policy Gradient Theorem** (Sutton et al., 1999), the gradient of the objective function can be expressed as the expected value

Algorithm 3 Deep Q-Learning Algorithm

Initialize replay buffer D
Initialize online network Q with random weights $\mathbf{w}$
Initialize target network Q^- with weights $\mathbf{w}^- = \mathbf{w}$
for each episode **do**
  Initialize S
  for each step of the episode **do**
    Choose A from S using ϵ-greedy policy
    Take action A, observe R, S'
    Store transition (S, A, R, S') in D
    Sample random minibatch of transitions (s, a, r, s') from D
    Compute target $y = r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-)$
    Compute loss $L = (y - Q(s, a, \mathbf{w}))^2$
    Update weights $\mathbf{w}$ by minimizing the loss
    Every C step, update target network weights $\mathbf{w}^- = \mathbf{w}$
    $S \leftarrow S'$
  end for
  Until S is terminal
end for

of the gradient of the log-probability of the action multiplied by the return. This allows to write the policy gradient as in Eq. 2.20. In the formula, $\nabla_{\mathbf{w}} \log \pi(a|s, \mathbf{w})$ is the score function.

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \mathbb{E}_{\pi_w}[\nabla_{\mathbf{w}} \log \pi(a|s, \mathbf{w}) Q^{\pi}(s, a, \mathbf{w})] \tag{2.20}$$

To compute the score function, Softmax Policy (Williams, 1992) or Gaussian Policy (Lillicrap et al., 2016) are two common choices.

Policy Gradient methods are effective, but they can be unstable. In fact, they need a high amount of samples to converge, and they can suffer from high variance, meaning that, for a parameterized policy $\pi(a|s, \mathbf{w})$ which returns a distribution over actions, small changes in the policy parameters can lead to large changes in the policy distribution and so in choosing actions. Changes need to be controlled, so the gradient of the policy needs to be regulated. In this context, **Natural Policy Gradients** (Kakade, 2001) tries to solve this issue. It attempts to find the direction in the parameter space that maximizes the expected return, but it performs the updates of the direction in a way that the changes in the policy are small.

One algorithm that uses Natural Policy Gradients is Proximal Policy Optimization (PPO) (Schulman et al., 2017), which works by clipping the policy gradient, so it limits the change in the policy parameters. The PPO algorithm is based on the idea of alternating between sampling data from the environment and updating the policy. Its architecture is composed of two neural networks:

- **Policy Network** - It is a network that takes the state of the environment as input and outputs the probability distribution over actions. It represents the behavior of the agent.
- **Value Network** - It is a network that takes the state of the environment as input and outputs the value function, so an estimate of the expected return that the agent can achieve starting from a given state. Including the value function in the PPO algorithm helps to reduce the variance of the policy gradient.

The objective function of PPO changes with respect to one of the policy gradients defined in Eq. 2.19 in a way that includes a penalty term that clips the gradient. It is defined in Eq. 2.21, where $r_t(\mathbf{w})$ is the probability ratio between the new policy and the old policy, we want it to be close to 1 as in this way the changes between the policy are small. A_t is the advantage function, which is a measure of how good an action is compared to the average action. Finally, ϵ is a hyperparameter that controls the clipping of the policy gradient.

$$\begin{aligned} L(\mathbf{w}) &= \mathbb{E}_t[\min(r_t(\mathbf{w})A_t,\ \text{clip}(r_t(\mathbf{w}), 1-\epsilon, 1+\epsilon)A_t)] \\ r_t(\mathbf{w}) &= \frac{\pi(a_t|s_t, \mathbf{w})}{\pi(a_t|s_t, \mathbf{w}_{\text{old}})} \end{aligned} \tag{2.21}$$

The pseudocode of the PPO algorithm is reported in Algorithm 4.

Algorithm 4 Proximal Policy Optimization Algorithm

Initialize policy network $\pi(a|s, \mathbf{w})$ and value network $V(s, \mathbf{w})$
for each iteration **do**
    for each epoch **do**
        Collect a batch of data by running the policy in the environment
        Compute the advantage function A_t
        Compute the probability ratio $r_t(\mathbf{w})$
        Compute the clipped objective function $L(\mathbf{w})$
        Compute the value function loss $L_v(\mathbf{w})$
        Update the policy network by minimizing $L(\mathbf{w})$
        Update the value network by minimizing $L_v(\mathbf{w})$
    end for
end for

Chapter 3

Related Works

In this chapter, we provide an overview of the most relevant works in the field of RL which are related to the topics of this thesis, and we will also provide information about Foundational Models (FMs). We start by explaining in Sec. 3.1 what is a FM, then we move to show FMs that have been recently exploited to improve the performance of RL agents in Sec. 3.2. Finally, in Sec. 3.3 we discuss the most recent works that combine multiple models in the context of RL.

3.1 Foundational Models

A FM is a large DL model, generally self-supervised, that is trained on massive datasets to learn hidden patterns and extract a general representation from the data (Devlin et al., 2018; Radford et al., 2018). Over the last few years, rather than training a model from scratch, researchers all over the world have been using these pre-trained models to extract features from the input data. They then adjust the model to a wide range of tasks and domains using fine-tuning. FMs are beneficial in many applications, especially when the amount of labeled data is limited. Their strength lies in the fact that they are very adaptable to a wide range of disparate tasks with a high degree of accuracy. However, developing a FM from scratch is computationally expensive and requires a large amount of data to be trained and money to be spent on computational resources. For example, a FM can be trained on a large corpus of text data to learn the language structure and then be fine-tuned on a smaller dataset to perform a specific task like sentiment analysis, text classification, etc. The same can be done with images, where

a FM can be trained on a large dataset of images to learn their structure, and then be fine-tuned on a smaller dataset to perform a specific task like object detection.

Recent works have revolutionized the field of natural language processing (NLP) by introducing Large Language Models (LLMs) like GPT-3 (Brown et al., 2020), BERT (Devlin et al., 2018), and Minerva (Orlando et al., 2024). Also, in the field of computer vision, models like Stable Diffusion (Rombach et al., 2022) or DALL-E (Ramesh et al., 2021) have changed the way images are processed and generated. Regarding the field of timeseries forecasting instead models like TimeGPT (Liao et al., 2024) have been recently proposed.

3.2 Foundational Models in Reinforcement Learning agents

The idea of improving RL agents' feature representation by providing information already processed by other models is not new. Most of the works apply a single model that pre-processes the input data and then learns the actual policy.

In specific, some work focused on feature representation that can be detached from the learning process by having a module that is able to extract relevant information. For example, the work of Shah and Kumar (2021) uses a pre-trained ResNet (He et al., 2016) for representation learning. A ResNet is a specific type of Convolutional Neural Network (CNN) whose architecture was designed to address the vanishing gradient problem in very deep networks. In fact, it makes use of *residual blocks* consisting of multiple convolutional layers, followed by a shortcut connection that skips one or more layers. In their work, they pre-train the encoder of the ResNet on a wide variety of images with different classes. Then, they use the encoder to extract features from the input data in the RL setting and fuse this information with features coming from robotic sensors. In their case, the combination of different information is done by a simple concatenation of the features. As in our work, they freeze the encoder during the learning process of the agent. A very similar work is the one of Yuan et al. (2022) that uses also a pre-trained ResNet to extract features, but differently from the previous work, they do not combine features extracted with other ones. Montalvo et al. (2023) proposed semantic segmentation too as a pre-processing step for RL agents to enhanching their performance. A dataset consisting of

Super Mario Bros standard frames and segmented frames was employed to pre-training a model. Then they use this model to preprocess frames during the training of a RL agent.

Several approaches exploit FMs to learn from collected data a general representation by applying different self-supervised methodologies, which can also be fine-tuned during the learning of the policy. For example, the work of Anand et al. (2019) proposed a method to learn state representations on Atari games. In fact, they proposed a model that, given observations of an agent that plays Atari games, exploits the spatio-temporal information of the frames to learn a linear representation of the state. This linear representation has to match the RAM state of the game, which is a representation of the game state that is used by the game itself. Other works instead, proposed self-supervised models that perform computer vision tasks. Like Kulkarni et al. (2019), Fig. 3.1, that presented a model that can discover keypoints i.e., coordinates of specific interesting points in the image. Their model uses a KeyNet (Jakab et al., 2018) and a CNN to extract the keypoints of objects and keep track of them over time.

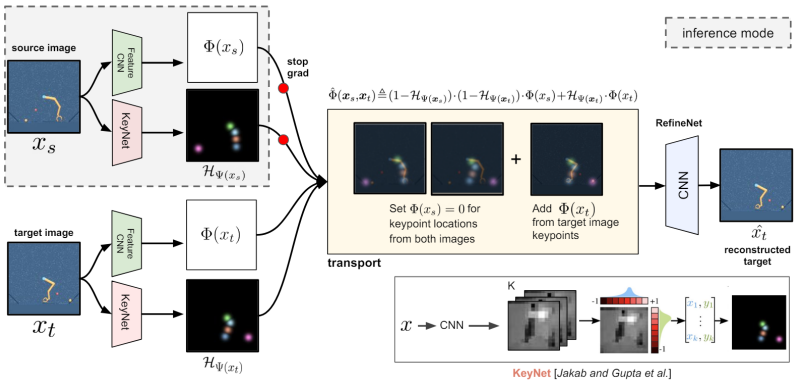


Figure 3.1: A schematic of the model presented in Kulkarni et al. (2019) is illustrated. The image is taken from the related paper. A KeyNet and a CNN are used together to extract and compute object keypoints.

Goel et al. (2018), Fig. 3.2, proposed a method for video object segmentation. In this case, a CNN was used to extract K different object masks from the frames of the game. Then they use the encoder of the video segmentation model to extract features from the frames and concatenate them with the features extracted from a static object network. Finally, the concatenated features are used to train an RL agent.

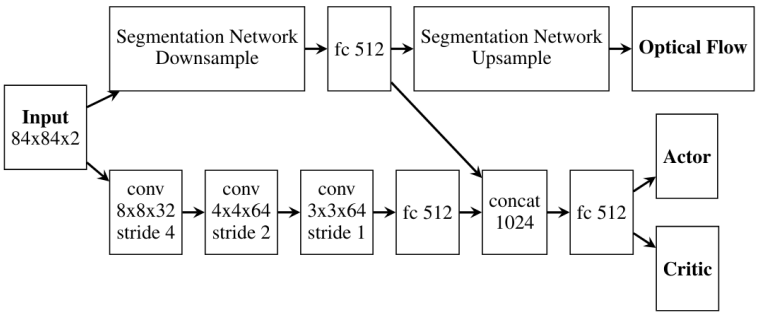


Figure 3.2: A schematic of the model presented in Goel et al. (2018) is depicted. The image is taken from the related paper. The Segmentation Network extracts moving object information from the current state. Also, the same state is given to a CNN to extract features. Both extracted features are concatenated together to form the enriched state representation for an RL agent.

Wu et al. (2021) proposed a self-supervised attention module to generate attention masks on images. The goal is to pre-train a model that is able to detect salient regions in the frame. They then combine the attention masks with the features extracted from the frame using a CNN by multiplying them. Finally, the enhanced state is used as input for the RL agent. As benchmarks, the works above, use the Atari games and show that the agent is effective in solving the games. Some works enhance the state representation by using human-labeled data for eye-tracking (Zhang et al., 2020) to improve the performance of the agent. The work of Thammineni et al. (2023) is an example. They use a model to predict a human eye-gaze map from the game frames and utilize this data to augment the agent's observations.

Regarding instead works that do not focus on video game benchmarks, Xiao et al. (2022) proposed a self-supervised model that receives as input a masked image, and the goal is to reconstruct the original image. Using a pre-trained encoder and freezing its weight, they are able to learn complex motor control from pixels. Others pre-train models on frame prediction tasks using a dataset consisting of YouTube video clips, human motions, driving, and more (Finn et al., 2016). The learned model is then used for decision-making in vision-based robotic control tasks.

In order to be effective, these representations should be as general as possible without having any bias with respect to the task the agent should solve.

Some works try to overcome this problem by presenting methodologies to build representations that are reward-free - meaning they are unbiased with respect to the task. For example, Stooke et al. (2021) proposed a method that uses CNN to associate pairs of observations separated by a short time interval. Then the encoder of the model is used in online RL. Another example is the work of Lan et al. (2023) which adds auxiliary objectives to help shape the latent representation.

Recently, LLMs have been used to improve the performance of RL agents too. Without going too much into detail, in Wang et al. (2023), skills are formalized as pieces of code and automatically generated. LLMs are also used in Yu et al. (2023) to produce a reward function, or in Lifshitz et al. (2023) where the state representation is enhanced by combining it with embeddings from a LLM. Finally, in Zitkovich et al. (2023) a vision-language model is used to improve agent efficiency.

3.3 Combining multiple models

How to effectively combine multiple latent features in a single representation is still an open problem. There is almost non-existent literature about works that try to apply more than one model at the same time to enhance the representation.

Many works focus on combining different skills learned by the agent by a simple concatenation of the features. Considering to have x as input, and k embeddings coming from pre-trained models or learned-from-scratch features, $E_1(x), E_2(x), \ldots, E_k(x)$. The final environment state representation $\mathcal{R}$ is computed as in Eq. 3.1, where with $\oplus$ we indicate the concatenation operator.

$$\mathcal{R} = E_1(x) \oplus E_2(x) \oplus \cdots \oplus E_k(x) \tag{3.1}$$

For example, Sahni et al. (2017) defined a skill as a policy that is able to solve a specific task. Each skill is trained separately using a reward function only relevant to the one. Then skill embeddings are generated, and the combination of the skills is done by a simple concatenation.

Other works use a *mixture of experts* (MoE) approach (Shazeer et al., 2017) to combine different embeddings. This method consists of having several *experts* subnetworks inside a much larger NN. The input is divided into multi-

ple parts. Each part is routed to a specific expert through the use of a *gating network*, which decides to activate only a (or multiple) expert(s). In such a way the experts will specialize in solving a specific part of the input. The output is computed considering a weighted summation of the experts' response, where the weights are determined by the gating network. Specifically, Eq. 3.2 provides an example of how the state representation $\mathcal{R}$ is computed using a MoE approach. Supposing to have k experts, $E_1, E_2, \ldots, E_k$, each expert E_i takes as input x and produces the output $E_i(x)$. The gating network takes the same input x and produces a set of weights $g_1(x), g_2(x), \ldots, g_k(x)$, where $g_i(x)$ represents the weights assigned to the expert E_i. The output $\mathcal{R}$ is then a weighted sum of the experts' outputs.

$$\mathcal{R} = \sum_{i=1}^{k} g_i(x) E_i(x) \tag{3.2}$$

Obando-Ceron et al. (2024) proposed a MoE approach for an RL setting. They indeed showed how scaling up the number of parameters of a RL agent can improve its performance. While the focus of their work is not one of combining different models, it still can be taken as a reference to explore a different way to combine embeddings. In fact, given the frame of Atari games as input, they use a CNN encoder to extract feature maps. The feature maps are then tokenized in different ways and each token is given to a different expert. The result of the experts is then combined using SoftMoE (Puigcerver et al., 2023) to produce the final representation.

In the context of multi-task learning Sodhani et al. (2021) proposed a method where a mixture of encoders extracts different features from the input data and produces different embeddings. Then, an attention mechanism is used to combine the embeddings using as context the metadata of the tasks processed by an LLM $\mathcal{Z}_{context}$. The attention module outputs a linear representation, which is called $\mathcal{Z}_{enc}$, that can be computed using the Eq. 3.2. In this case, g_i is the attention weight associated with a specific encoder E_i. To compute the final state representation $\mathcal{R}$, $\mathcal{Z}_{enc}$ is concatenated with $\mathcal{Z}_{context}$. This final representation is used as input for the RL agent. Fig. 3.3 provides a scheme of the architecture of their model.

All of these works combine multiple embeddings in different ways, but they do not consider the case where the embeddings come from different pretrained models. FMs, instead, are able to extract information from different domains, like images, text, video, etc. Exploiting their information could

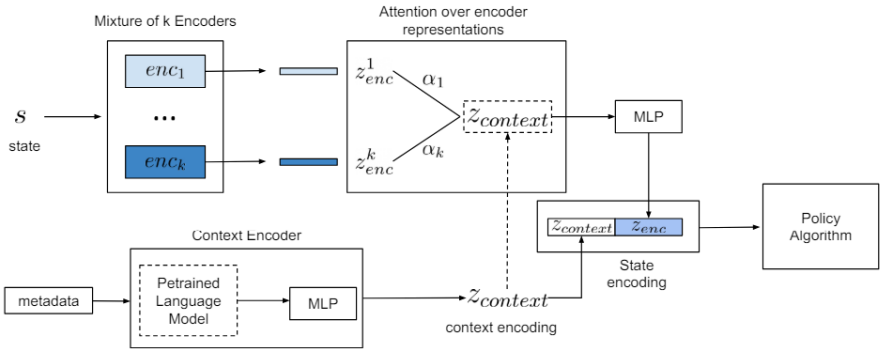


Figure 3.3: A schematic of the model presented in Sodhani et al. (2021) is illustrated. The image is taken from the related paper. k different encoders take the state as input and produce k encodings. Given the metadata of a task, an LLM is used to produce the context of an attention mechanism that will weight the importance of each one of the k encoders. The final state representation is constructed by concatenating the output of the attention module with the LLM embedding.

lead to a more general and effective representation of the state space.

To the best of our knowledge, Team et al. (2024) is the first work that provides the agent with multiple instances of FMs. In their work, they trained the agent first collecting a huge dataset of video of games and other research environments. Also, they collected a broad range of text instructions, clustering the tasks into different categories. Using this data, they pre-trained a video-prediction model, an image-text alignment model, and a text encoder model. Then, the agent receives as input visual information from a simulated 3D environment and language instructions, and preprocess the information using the pre-trained models. To combine the embeddings they use a trained-from-scratch transformer-based model (Dai et al., 2019) which includes attention mechanisms on its architecture. The output of the agent is keyboard and mouse actions. Figure 3.4 depicts their model's architecture.

They showcase that combining pre-trained models and trained-from-scratch components leads to better-performing agents. Unfortunately, due to the large scale of their work, there is no particular attention towards how FMs' representations are combined. Also, using such a big model as a transformer could be computationally expensive and not scalable. In fact, in cases where

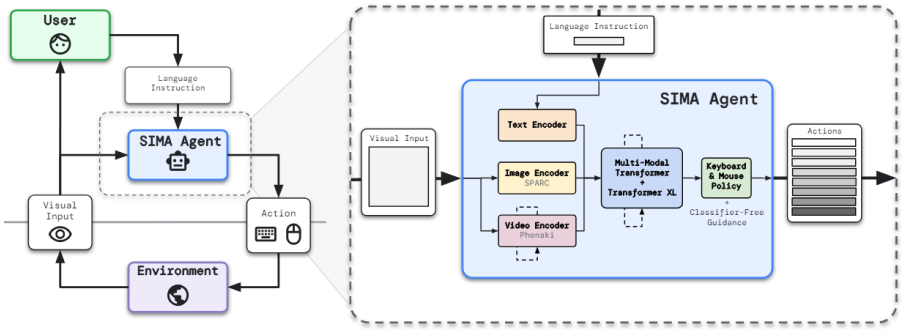


Figure 3.4: A schematic of the model presented in Team et al. (2024) is depicted. The image is from the referenced paper. A multimodal Transformer is used to gather information from text, image, or video encodings.

the agent has to learn or use new abilities, the whole model has to be retrained.

In recent years, more emphasis has been dedicated to agents capable of continuously acquiring knowledge across numerous tasks, adapting to any situation, and in constant training. This is a paradigm shift from traditional RL approaches and we will refer to it as Lifelong Learning Agents (LLA) (Parisi et al., 2019). In the context of LLA, an effective and scalable solution to combine FMs encoding is needed in such a way that skills can be added to the agent without the need to retrain the whole model. Recent developments in attention models could prove very useful in this context, as they are able to focus on different parts of the input data and combine them in a meaningful way. Some works already proposed attention mechanisms to extract and enhance specific features of the state representation built by the RL agent. For example, both Bramlage and Cortese (2022) and Blakeman and Mareschal (2022) proposed different attention-like methods to focus on specific features of the state representation. These works do not focus on combining multiple representations extracted from different models, but they restrict the agent to concentrate only on certain features.

To this purpose, in this work, we propose Weight Sharing Attention (WSA) as an attention-like combination mechanism to merge different pre-trained encodings. Agents learn how to perform a task composing different enriched elements instead of having to learn the state space from scratch. Also, they are able to adapt to many situations just by weighting the importance of the different abilities.

Chapter 4

Method

As already highlighted in Ch. 1 and Ch. 3 a typical RL setting consists of an agent that receives as input the state of the environment and has to learn a policy to solve a specific task. Most of the training time is spent trying to create a useful representation of the state, which is then used to learn the policy. The focus of this work is to simplify as much as possible the representation learning of current RL algorithms leveraging prior knowledge and letting the agent focus on learning the actual policy to solve the task. Many previous works already, mentioned in Sec. 3.2, share the idea of leveraging FMs to enhance the state representation facilitating the transfer of world knowledge and simplifying the RL training process. Almost none of them have tried to combine multiple models at the same time. Our methodology is designed to use multiple models at the same time, instead of just one, to enhance the efficiency and effectiveness of RL agents by leveraging a set of pre-trained models tailored to the current task.

Our work poses two main questions, how to represent skills for an RL agent and how to combine them. To answer the first question, FMs present themselves as good candidates, as they provide a general representation of the environment. These representations can be leveraged in RL to speed up the learning process and improve performance on complex tasks. Using FMs' knowledge, RL agents can achieve higher efficiency and effectiveness, even in environments with sparse rewards or limited data. First, our work proposes a set of basic abilities. RL agents are equipped with these FMs, which serve as the prior knowledge needed to generate diverse representations of the environment. While agents interact with the environment, the current

observation is collected and processed through each pre-trained model. This process produces multiple perspectives or views of the world, each distinctively represented according to the particular model used.

As for the latter question, instead, this thesis aims to study how different FM embeddings can be combined in several ways to compute a rich and informative representation of the environment. We focus on how to effectively combine their latent representations to improve agents' performance. The challenge lies in integrating these possible divergent views into a single and enriched representation. In order to achieve this, we employ a specialized combination module, that synthesizes the information from the various models into a single latent representation. The intuition is that the enriched representation provides a comprehensive understanding of the current state of the environment, derived from the collective insights of the pre-trained models. By using FMs, our approach significantly alleviates the RL agent's need to learn the environmental representation from scratch, allowing it to focus more on refining the action-mapping process. This not only speeds up the learning process but also enhances overall performance by providing a well-structured and informative representation.

The architecture of our agents is divided into two main components, and it is shown in Fig. 4.1. The first part takes care of processing the current observation into multiple latent representations and then combines them into a single enriched representation. We refer to this part as the *Feature Extractor* module. The second part is the *Policy Learning* network, where the unified representation is fed into a small fully-connected neural network whose responsibility is to map the latent encoding to actions.

In the following sections, we will describe in detail our methodology, and how we map information from observations to actions. In particular, Sec. 4.1, will describe how we designed the Feature Extractor module focusing on various combination techniques to merge the representations extracted by the FMs. While in Sec. 4.2, we will provide some information about the Policy Learning network.

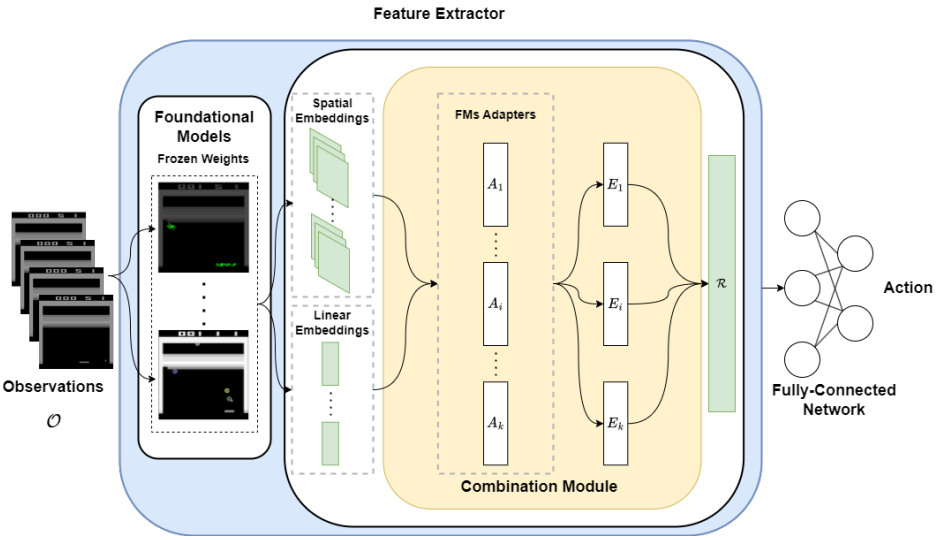


Figure 4.1: We show a general representation of the agent's architecture, depicting information flow from observations to actions. The last 4 frames of the game are stacked together to form the observations. The Feature Extractor module is responsible for processing the current observation, extracting meaningful representations, and combining them. It is equipped with a set of pre-trained models and a combination module. In general, combination modules are composed of a set of adapters $\mathcal{A}$ which learn a representation of the FM output and eventually resize it to a fixed size. They produce a set of embeddings E which are then combined in different ways to produce the final representation $\mathcal{R}$. The Policy Learning network is the second part of the agent's architecture used to map the current state $\mathcal{R}$ to an action.

4.1 Feature Extractor

The Feature Extractor module is the first part of our agent, it is responsible for processing the current observation and extracting a meaningful representation of the environment in a compact latent space. It is composed of two main parts, the first part is a set of pre-trained models, while the second part of the module is the combination module, which is the core of the Feature Extractor.

4.1.1 Foundational Models

Agents receive as input the last four frames of the game, which are stacked together to form the observations. Starting from the observations, the first step of our methodology is to gather information from the environment at different levels of abstraction. We equipped our agents with a set of k pre-trained models $\Psi = \{\psi_1, \psi_2, \ldots, \psi_k\}$, capable of represent the current observation $\mathcal{O}$ under different perspectives. Meaning that the models are pre-trained on different tasks. During the training of RL agents, their weights are frozen in order to preserve the knowledge acquired during the pre-training phase and speed up the learning process. In particular, we will have a set of k embeddings E^{FM} in output from the FMs, $E_i^{FM} = \psi_i(\mathcal{O})$. The models operate both in the spatial field and in the linear field. Embeddings in the spatial field consist of a set of feature maps, and have a shape of the form $E_i^{FM} \in \mathbb{R}^{j \times m \times n}$ where j is the number of feature maps, m, n are the width and height dimensions. Linear encodings instead are a one-dimensional vector of shape $E_i^{FM} \in \mathbb{R}^q$ where q is the length of the vector. This is the initial step of our methodology. We will talk more in detail about the FMs selected in Sec. 5.1.

4.1.2 Combination Modules

The next step is to combine the representations extracted by the FMs. Combining different FM encodings is a challenging task, as each model provides a different view of the environment. The goal is to merge these perspectives into a single enriched representation that captures the most relevant information. To achieve this, we developed several combination modules, each with its own strengths and weaknesses. All solutions act as interchangeable components, and we tested and compared their performance. All combination modules have in common a first *preprocessing step*, consisting of a set of adapters $\mathcal{A}^s$, called *Spatial Adapters*, which are used for embeddings E_i^{FM} that operate in the spatial field. This set of adapters is implemented as two convolutional layers followed by a ReLU activation function. The first convolutional layer is implemented with a kernel of dimension 1×1, this is done to decrease the computational cost and reduce the number of feature maps. The second layer applies convolution and resizes the latent representation of the feature maps to a fixed size, ensuring compatibility among different models. While the FMs that operate in the linear field are left as they are. Figure 4.2 shows the scheme of this phase. Each spatial adapter will output a set of feature maps of dimension `16` $\times$ `16`, respectively

for width and height.

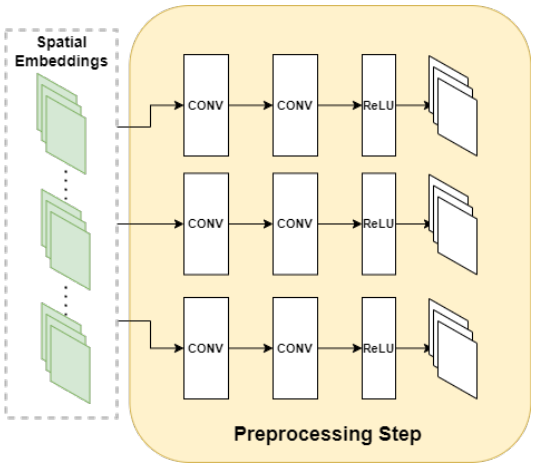


Figure 4.2: We show the subnetwork for the *preprocessing step*. It is composed of two convolutional layers followed by a ReLU layer. The first layer applies a 1×1 convolution, this is done to reduce the number of feature maps and save computational cost. The second layer applies convolution and reduces the dimension of the feature maps. Each spatial adapter will produce a set of kernels of dimension 16×16.

Eventually, a combination module can define another set of adapters called *Linear Adapter* $\mathcal{A}^l$ used to fix the dimension of various representations. These will produce a set of linear embeddings E^l. Each combination module then outputs a **linear representation** of the state that is provided as input to the *Fully-Connected Network*.

First of all, we will present one by one all the combination techniques that we have developed to merge the representations extracted by the FMs, with a particular emphasis on the **Weight Sharing Attention** module, which is the component that we propose as a solution to the problem of combining different FMs in an effective way.

Weight Sharing Attention

With a view to lifelong learning agents, the aim is therefore to equip the agent with prior skills that can be changed or improved over time to adapt to new unseen tasks. To this end, a model capable of re-adapting to new abilities with little or no effort is needed. To facilitate the integration of different pre-trained models, we propose WSA. The WSA module leverages the

concept of **weight sharing**, as seen for the first time in CNNs (Fukushima, 1980), and **attention** (Vaswani et al., 2017) principles to efficiently merge outputs from diverse FMs. In specific, WSA dynamically adjusts the weights assigned to each pre-trained model based on the current context, emphasizing the most relevant models for the current state.

This module, depicted in Fig. 4.3, receives as input the representation extracted by the FMs, on which it applies the preprocessing step described before. Then, spatial embeddings are flattened to obtain a one-dimensional vector. WSA module is also equipped with *Linear Adapters* $\mathcal{A}^l$, where each adapter $\mathcal{A}_i^l$ is a small neural network consisting of a single linear layer followed by a non-linearity layer, in this case, a ReLU activation function. These adapters are used to learn a representation of FMs output and also to resize the different linear latent representations of the state to a fixed size. This guarantees consistency between various models. As a result of this operation, we obtain an embedding for each FM representation, called E_i^l. All the embeddings have the same dimension, and it is possible to combine them in a meaningful way.

Among the provided FMs, one is used as **State Encoder** $\mathcal{E}$ to compute a representation of the current state, which serves as *context* $\mathcal{C}$ for the attention mechanism. The State Encoder produces a spatial encoding of the current observation which is flattened to obtain a linear embedding. A linear adapter $\mathcal{A}_0^l$, of the same shape as the other linear adapters, is used to resize the latent representation to a fixed size, and this will be the context $\mathcal{C}$.

To compute the attention weights, we use a shared MLP f_θ that takes as input the context $\mathcal{C}$ concatenated with the single embedding E_i^l of each FM. This MLP outputs the model-specific weight w_i to determine the contribution of each model in the final representation. Weights are stacked together and passed through a softmax function to ensure that they sum to one. It is important to note that this MLP is shared across all models, ensuring uniform weight computation. By dynamically adjusting weights based on the current context, WSA emphasizes the most relevant models for any given situation, resulting in a more accurate and adaptable representation. The final representation $\mathcal{R}$ is computed using the weights for all pre-trained models and computing a weighted sum of their embeddings. The output obtained is a combination of the various perspectives, provided by the pre-trained models, into a single enriched latent representation. This final linear representation is used for action mapping, enhancing the performance and

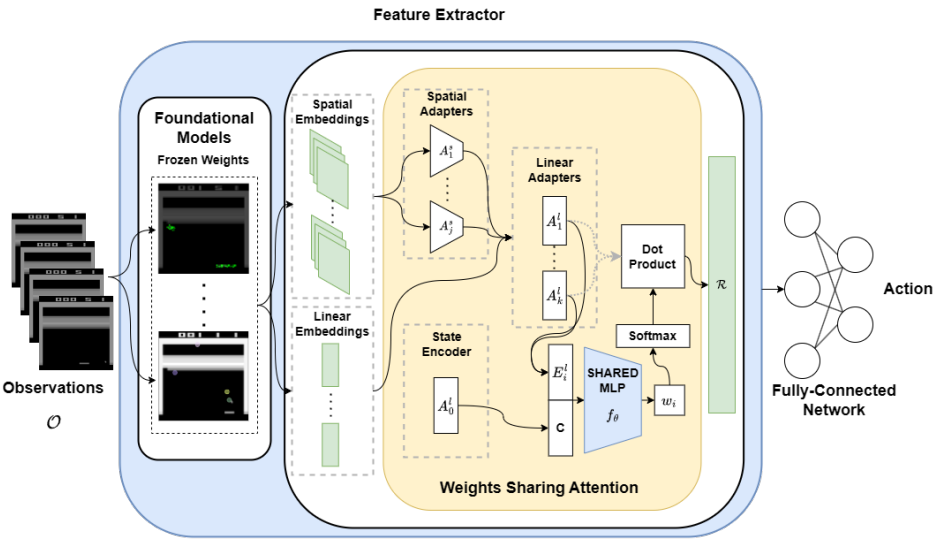


Figure 4.3: We show in specific all the information courses from game frames to agent actions specifying the architecture of the WSA module. The last four frames are stacked together and passed as input to the FMs and their spatial adapters $\mathcal{A}_1^s, \dots \mathcal{A}_j^s$ first, and then to the linear adapters $\mathcal{A}_1^l, \dots \mathcal{A}_k^l$ to obtain an embedding of the current state E_i^l. A State Encoder $\mathcal{E}$ and its adapter $\mathcal{A}_0^l$ compute an encoding of the state, that will be used as context $\mathcal{C}$. A shared MLP layer takes as input the context $\mathcal{C}$ and the representation of the i^{th} FM, E_i^l, and produces the weight w_i for that specific view of the state. The final enriched representation $\mathcal{R}$ is obtained from the weighted summation between each weight w_i and its respective embedding E_i^l. $\mathcal{R}$ is given as input to agents' *Fully-Connected Network*.

learning efficiency of RL agents by integrating the strengths of multiple FMs. The weights of the adapters are optimized during training. We report the pseudocode for the actual implementation in Algorithm 5, where with $\mathcal{A}$ we include both spatial and linear adapters as a unique block.

Two additional features are guaranteed by the design approach. WSA adds **explainability** to the model, where with explainability we mean the ability to understand the decision-making process of the agent. In fact, by examining the weights assigned to different pre-trained models, we can gain insights into which models are most influential in a given context. Additionally, WSA is **scalable** and **adaptable** to any number of pre-trained models. Its shared component can handle any number of models, we can easily add

Algorithm 5 Weight Sharing Attention

`In blue we highlight the components that are updated during training.`

Require: Current observation $\mathcal{O}$, List of FMs Ψ, State Encoder $\mathcal{E}$

$\mathcal{C} = \mathcal{A}_0(\mathcal{E}(\mathcal{O}))$ ▷ Compute current context representation
for FM ψ in Ψ **do**
    $x = \psi_i(\mathcal{O})$ ▷ Forward pass using current state
    $E_i = \mathcal{A}_i(x)$ ▷ Use the adapters to compute the resized embedding
    $w_i = f_\theta(\mathcal{C}, E_i)$ ▷ Compute the weight for current model using shared component
end for
$\mathcal{R} = \sum_{i=0}^{|\Psi|} w_i * E_i$ ▷ Final representation: weights W, embeddings $E \rightarrow W \cdot E$

or remove FMs from the set without the need to retrain the whole agent, making it a flexible solution for dynamically evolving agents.

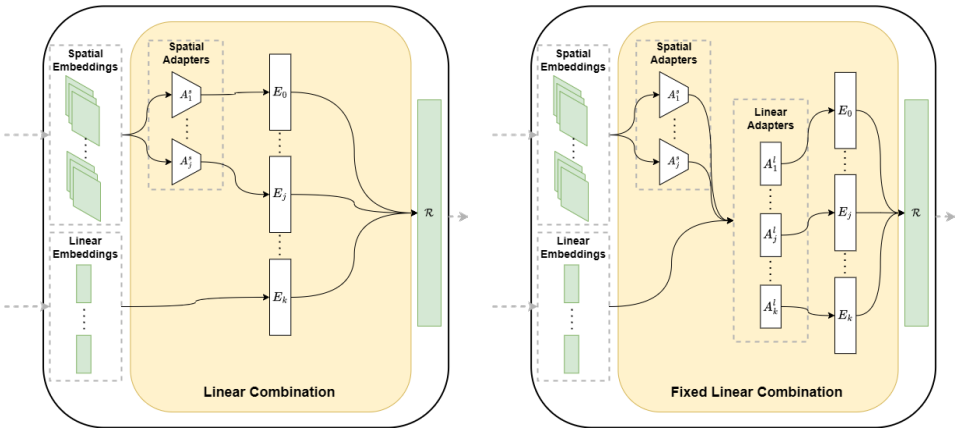
Linear Combination Modules

Linear combination modules are the simplest way to merge the representations extracted by the FMs. Considering the set of k pre-trained models Ψ, each of them can output an embedding in the spatial field or a one-dimensional vector. We apply the spatial adapters to the embeddings that operate in the spatial field, and we leave the linear embeddings as they are. Then, we flatten the spatial embeddings to obtain a linear encoding. The embeddings of each model are then concatenated to create a large feature vector. Specifically, given two linear embeddings $E_i^l \in \mathbb{R}^n$, $E_j^l \in \mathbb{R}^m$ the concatenation of the two representations $E_i^l \oplus E_j^l$ will produce an encoding of size $E_c^l \in \mathbb{R}^{n+m}$. This constitutes the final representation of the environment $\mathcal{R}$. We refer to this module as *LIN* for short.

Building such a large vector can be helpful since the information from different models is not compressed, and the final representation is rich and informative. One problem though with this combination module is that the FMs may have very different dimensionality. An embedding might be huge compared to the others, leading to a situation where one model becomes more important than the others just because its dimension is larger. So, we developed another combination module, which we refer to as *FIX*, where

first we obtain a linear representation as in the previous module for each model. Then, we map the representation of each model into an embedding of fixed dimensions using k linear adapters $\mathcal{A}^l$ of the same size. Each adapter is a neural network constituted of a linear layer followed by a ReLU activation function. The new embeddings E^l are then concatenated as before to compute the representation $\mathcal{R}$ and used as input for the rest of the network. In specific, given two linear embeddings in output from the linear adapters $E^l_i \in \mathbb{R}^n$, $E^l_j \in \mathbb{R}^n$ the concatenation of the two representations $E^l_i \oplus E^l_j$ will produce an encoding of size $E^l_c \in \mathbb{R}^{2n}$. Note that in this case, the dimension n of the two encodings is the same. In Fig. 4.4 we show the architecture of the two linear combination modules. In general, Eq. 4.1 represents how final state space representation $\mathcal{R}$ is computed both for LIN and FIX, where $k = |\Psi|$.

$$\mathcal{R} = E^l_1 \oplus E^l_2 \oplus, \ldots, \oplus E^l_k \tag{4.1}$$



(a) `LIN` (b) `FIX`

Figure 4.4: We show the architecture of the Linear Combination Module (LIN) and the Fixed Linear Combination Module (FIX). In LIN, the embeddings of each model are first flattened to one-dimensional vectors and then concatenated to create a large feature vector. In FIX, the embeddings are also mapped into encodings of fixed dimensions using k different linear layers with the same size and then concatenated

Convolutional Combination Modules

In this combination module, we exploit the fact that FMs operate in the spatial field. In fact, flatting the feature maps and concatenating them as in the linear combination modules may not be the best choice, as the spatial information is lost. To overcome this issue, after applying the usual preprocessing step consisting of the spatial adapters, we reshaped the one-dimensional embeddings into spatial ones, to match the dimensions of the other representations. At this point, each embedding is constituted by a set of feature maps of dimension 16×16, so it is possible to **concatenated along the channel dimension**. This will output an encoding of shape $j \times 16 \times 16$, where j is the sum of the number of channels of each spatial embedding. This encoding is then passed to one or more convolutional layers $\mathcal{A}^c$, one after another, to extract features. Each convolutional layer is followed by a non-linearity function, in this case, a ReLU. These convolutional layers do not restrict the dimension of the feature maps, but they restrict the total number of channels outputting a tensor of shape $32 \times 16 \times 16$ where 32 is the number of channels. At this stage, the output of the convolutional layer is flattened to constitute the final representation $\mathcal{R}$, and the embedding is passed to the rest of the network. We refer to this combination module as *CNN*. Eq. 4.2 provides information about how the final state representation is built. E_i^s are spatial embeddings, while *conv* is the convolutional operator, whose output is flattened.

$$\mathcal{R} = conv(E_1^s \oplus E_2^s \oplus, \ldots, \oplus E_k^s) \tag{4.2}$$

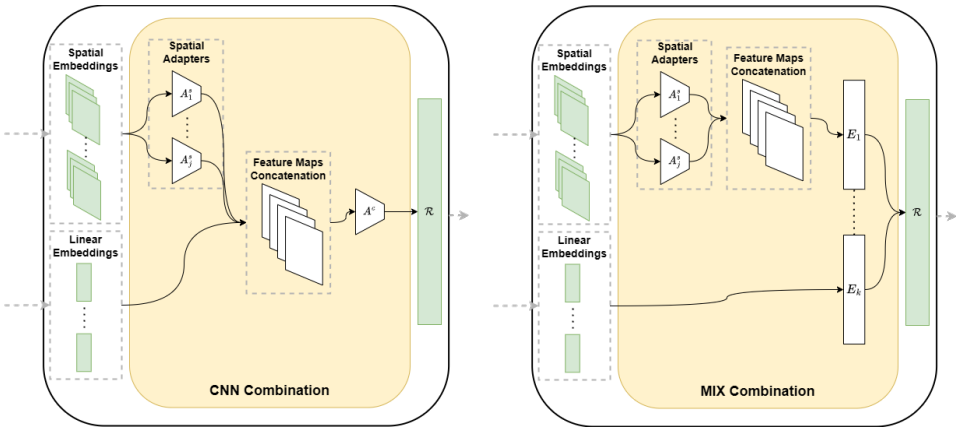
This module is particularly useful when the FMs are designed to capture spatial information, as in the case of images, but there may be cases where the FMs are not designed to capture spatial information. Assuming that the set of FMs is composed of models that operate in the spatial field and models that operate in the linear one, we can exploit these two types of models to create a more complex combination module.

As before, first, we preprocess the output of the FMs using the spatial adapters. An initial stage of concatenation is done between FMs embedding that operates in the spatial field along the channel dimension, obtaining as before an encoding of shape $j \times 16 \times 16$. We then pass this embedding through a single convolutional layer to extract spatial features, followed by a ReLU activation function. Again, this convolutional layer only restricts the total number of channels outputting a tensor of shape $32 \times 16 \times 16$

where the first dimension represents the number of channels. Finally, we flatten the output of the convolutional layer and a second concatenation is done considering the embeddings of the linear FMs to create the final representation $\mathcal{R}$. We call this combination module *MIX* as it is a mixture of the previous two modules. Eq. 4.3 provides information about how MIX produced the final state representation. E_i^s are spatial embeddings, E_j^l are linear embeddings, while *conv* is the convolutional operator whose output is flattened. We note that $p + q = |\Psi|$.

$$\mathcal{R} = E_1^l \oplus, \ldots, \oplus E_p^l \oplus conv(E_1^s \oplus, \ldots, \oplus E_q^s) \tag{4.3}$$

Figure 4.5 shows the architecture of the two convolutional combination modules.



(a) `CNN` (b) `MIX`

Figure 4.5: We show the architecture of the Convolutional Combination Module (CNN) and the Mixed Combination Module (MIX). In CNN, we first reshape all embedding such that can be concatenated along the channel dimension. Then we use a CNN to extract features and produce the representation $\mathcal{R}$. In MIX, first, we concatenate the spatial embeddings together, then we flatten their representation to a linear one and concatenate all the linear embeddings.

Reservoir Combination Module

Inspired by the work of Gallicchio et al. (2017), we decided to explore reservoir dynamics properties to map the input into a lower-dimension latent space using a non-linear transformation. To develop this combination module, we created a *reservoir* with fixed weights using a predefined number of neurons. As input, for the reservoir, we used the representation $\mathcal{R}$ built by the LIN combination module. The output of the reservoir is a vector of the size of the number of neurons on it. This is the new computed representation $\mathcal{R}$, which is used as input to the policy learning network, which can be viewed as the *readout*. We refer to this combination module as *RES*. Figure 4.6a shows the architecture of the reservoir combination module.

In specific, suppose we have a linear embedding $E^l \in \mathbb{R}^{1\times n}$ which is the result of the concatenation as in *LIN*. We construct two weight matrices $\mathbf{W}_{\text{in}} \in \mathbb{R}^{n\times m}$ and $\mathbf{W}_{\text{res}} \in \mathbb{R}^{m\times m}$ with fixed weights. The length of the vector is n, while m is the number of neurons in the reservoir. The input projection into a latent space is computed by a matrix product, $IN = E^l \times \mathbf{W}_{\text{in}}$. We note that $IN \in \mathbb{R}^{1\times m}$. This is fed as input to the reservoir, which will compute the output by performing the matrix multiplication $H = IN \times \mathbf{W}_{\mathbf{res}}$. The internal state of the reservoir is computed as in Eq. 4.4, where with *tanh* we refer to the Hyperbolic Tangent activation function and it is used to add non-linearity. The internal state of the reservoir will be the final representation $\mathcal{R}$ used as input to the *Fully-Connected* network.

$$\mathcal{R} = tanh(IN + H) \tag{4.4}$$

DotProduct Attention Combination Module

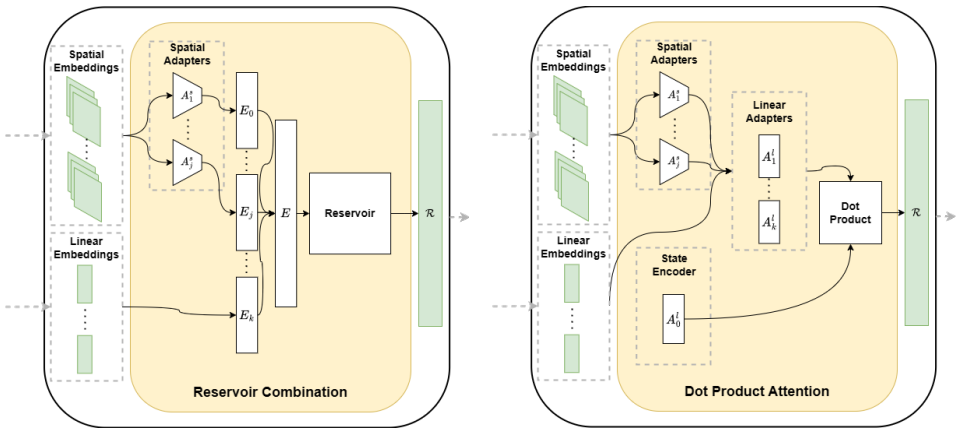
We also explored the possibility of combining representations using different types of attention-like mechanisms conditioned on the configuration of the environment. In this case, we used the *scaled dot product attention* (Vaswani et al., 2017). As for the WSA module, the pre-trained model representations are reduced to a vector of fixed size using the spatial and linear adapters. In specific, $E_i^l \in \mathbb{R}^n$ where n is the dimension of the encoding. These are then stacked together to form a tensor of shape $(k \times n)$, where k is the number of available FMs. This constitutes the **key** (K) part and the **value** (V) part of the attention module. For the **query** (Q) part, we used the same procedure as for the WSA module to produce the context $\mathcal{C}$. We used the State Encoder to encode the current state of the game in a latent space, and we used an

adapter to obtain a vector of the same dimensions. This attention module first computes the dot product of the query with the keys, divided by the square root of the dimension of the query, then it uses a softmax function to obtain the weights. Finally, the output of the module is computed as the dot product of the weights with the values. We call this combination module *DPA* and in Fig. 4.6 we show the architecture of the module. In Eq. 4.5 we report the general attention formula as known in the literature, which will produce the weights w_i for each pre-trained model.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4.5}$$

As for WSA, the final representation is computed by a weighted summation of the attention weights for the linear embeddings, Eq. 4.6.

$$\mathcal{R} = \sum_{i=0}^{|\Psi|} w_i * E_i^l \tag{4.6}$$



(a) `RES` (b) `DPA`

Figure 4.6: We show the architecture of the Reservoir Combination Module (RES) and the DotProduct Attention Combination Module (DPA). In RES, we use reservoir dynamics to encode information coming from different FMs. In DPA, the FMs are mapped to a fixed dimension using k identical linear layers and the State Encoder ε serves as context $\mathcal{C}$, similar to WSA.

4.2 Policy Learning

This is the last part of the agent's architecture, where the enriched representation obtained from the Feature Extractor module is used to map the current state to an action. It receives as input the augmented representation $\mathcal{R}$ as a one-dimensional vector. $\mathcal{R}$ is assumed to contain all the relevant information, simplifying the learning process of the agent. The Policy Learning network is a small fully-connected neural network that takes as input $\mathcal{R}$ and outputs either the probability distribution over the possible actions π or the value function, $v(s)$ or $q(s, a)$. The network generally is composed of just one layer with a small number of neurons. This simplicity helps in fast computation and reduces the risk of overfitting. Eventually, the architecture can be scaled to capture more complex dynamics. In fact, the size of this network can be increased by adding more layers and incorporating various activation functions such as ReLU to introduce non-linearity and capture complex patterns within the state representation. Each additional layer allows the network to learn higher-level features, potentially leading to improved decision-making capabilities and overall agent performance.

As already mentioned in Ch. 2, to optimize the policy or the value function, different loss functions can be defined depending on the task and the learning algorithm. For example, in the case of policy optimization, the loss function can be the negative log-likelihood of the action taken by the agent, Eq. 4.7.

$$\mathcal{L}(\mathbf{w}) = -\mathbb{E}_{\pi_\mathbf{w}}[A(s, a) \log \pi_\mathbf{w}(a|s)] \tag{4.7}$$

Where $\pi_\mathbf{w}$ is the policy parameterized by $\mathbf{w}$, $A(s, a)$ is an *Advantage Function* which is a measure of how much better an action is compared to other actions in a given state, and $\mathbb{E}_{\pi_\mathbf{w}}$ is the expectation of the distribution over the action given states, under the policy $\pi_\mathbf{w}$.

In the case of value function optimization, the loss function can be the MSE between the predicted value and the target value, Eq. 4.8. Where $V_\mathbf{w}(s)$ is the predicted value function, and $V_{\text{target}}(s)$ is the target value function.

$$\mathcal{L}(\mathbf{w}) = \mathbb{E}_{\pi_\mathbf{w}}[(V_\mathbf{w}(s) - V_{\text{target}}(s))^2] \tag{4.8}$$

Weights are updated using gradient descent to minimize the loss function, with gradients computed via backpropagation.

Chapter 5

Experiments & Results

In this chapter, we are going to provide information about how we implemented the various modules of our work, the experiments that we conducted to test our agents, and the results obtained. We start by describing the experimental setup in Sec. 5.1, talking about the libraries and tools used, the data collected, the training process, and the environments used to evaluate our agents. We will present the experiments we performed one by one and for each one, we will show the results obtained. We start from the initial experiments in Sec. 5.2, where we tested different combination modules to find the best-performing ones. Then, in Sec. 5.3, we will consider only the best three performing modules for each game, and we will compare them with PPO. In Sec. 5.4, we will focus on the *Breakout* game, which showed a different behavior compared to the other games. We will analyze the reasons behind this behavior and propose some possible solutions to improve the performance of the agent in this game. In Sec. 5.5, we will provide insight into the explainability of the WSA. In Sec. 5.6, to prove that our approach is not dependent on the RL algorithm used, we will test the effectiveness of WSA using **DQL**. Finally, in Sec. 5.7 we will discuss some final considerations of our proposed method.

5.1 Experimental Setup

In order to interact and simulate Atari environments (Bellemare et al., 2013) we used broadly the API provided by Gymnasium (Towers et al., 2023), which has extensive support for Atari games and has been widely used in the RL community. The training performance was tracked using *Weights*

and Biases (Biewald, 2020), which is a machine learning platform that provides tools for experiment tracking, model optimization, and more. The platform allows to monitor the agents and store the results in a cloud-based database.

Our work uses `Stable-Baselines3` (SB3) (Raffin et al., 2021) as a framework for agent architecture and reliable implementations of RL algorithms in PyTorch. SB3 implements the algorithms with a modular architecture, which allows to easily modify the agent's structure. This is particularly useful for us, as we need to combine different representations in the agent's architecture. In fact, SB3 provides a simple interface to decompose the agent's network into a *Feature Extractor* and a *Fully-Connected Network*, and this is the schema that we reported in our architecture as shown in Figure 4.1. We used the *Feature Extractor* to implement the different combination modules providing as output the composed latent representation. The *Fully-Connected Network* is kept as simple as possible, this is because most of the information should be provided by the FMs' representations. It receives as input the FMs' combined linear encoding and subsequently maps it to actions (or values).

For our experiments, we mainly used PPO algorithm (Schulman et al., 2017) as it is one of the most widely used algorithms in the RL community. We maintained the hyperparameters provided by Raffin (2020) for PPO. To prove that a better state representation leads to better-performing agents, we did not conduct any hyperparameter search this time, but instead, we compared our agents with already fine-tuned ones. Also, this avoids adding complexity to the experiments and growing the time. Instead, we only modify the model architecture. Table 5.1 shows the hyperparameters used for the PPO agent.

To set the environment in order to be used by the agent, we used the *Atari Preprocessing* wrapper provided by SB3. It preprocesses the frames by grayscaling them, resizing them to 84x84 pixels, and then stacking the last four frames to provide the agent with temporal information. We also run multiple instances of the environment in parallel to speed up the training process, in particular, we used `8` parallel environments. So, the agent receives as input a tensor of shape $8 \times 4 \times 84 \times 84$, where the first dimension represents the number of parallel environments, the second dimension represents the number of frames stacked together and the last two dimensions represent the size of the frame. The experiments were conducted on a machine equipped with 4 Intel Xeon Gold 6140M CPUs for

Hyperparameter	Value
N. Envs.	8
N. Stacks	4
N. Steps	128
N. Epochs	4
Batch Size	256
N. Timesteps	10.000.000
Learning Rate	2.5e-4
Clip Range	0.1
VF. Coef.	0.5
Ent. Coef.	0.01
Normalize	True

Table 5.1: We provide the hyperparameters used for our PPO agents both for end-to-end PPO and for the agents with FMs.

a total of 144 threads, 4 Nvidia Tesla V100 GPUs with 16GB of memory, and 1.2TB of RAM. The experiments were conducted on a single GPU, eventually running multiple experiments in parallel. The project code can be accessed at the GitHub repository in Carfì (2024) `https://github.com/EliaPiccoli/skilled_rl`.

5.1.1 Environments

In this section, we provide a brief overview of the environments used to evaluate our agent. We have chosen a set of Atari games that are widely used in the literature to evaluate their performance. We focused on games that have a discrete action space and discrete observations. In particular, we have selected games that require different skills to be solved, in order to test the effectiveness of our agent in a variety of tasks. We will list the games used in the experiments, and we will provide a brief description of each one. In Fig. 5.1 we provide an example of game frames for each one.

Pong

The game of Pong is a table tennis simulation. This game is played by two players, each controlling a paddle that can move up and down along the left and right edges of the screen. The paddles are used to hit a ball back and forth. The right paddle is controlled by the RL agent, while the left paddle

is controlled by the opponent. Points for a player are accumulated when the opponent is unable to successfully return the ball, resulting in the ball passing beyond their paddle. The objective of the game is to score as many points as possible preventing the opponent from doing the same. The game ends when one of the players reaches the score of `21`. From the viewpoint of the RL agent, the game is considered solved when it is able to consistently outperform the opponent.

Breakout

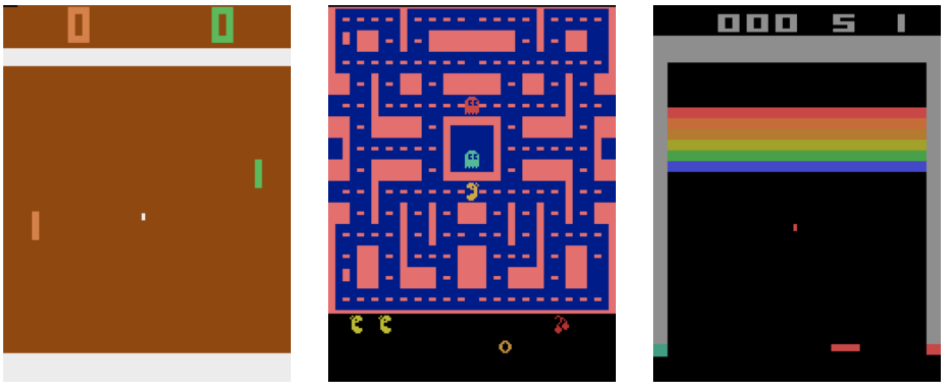
Breakout is a game where the dynamic is similar to Pong, but the objective is different. In this case, the environment is constituted by a paddle that can move horizontally at the bottom of the screen, which hits a ball that bounces on the walls. At the top of the screen, there is a wall of bricks, and the agent has to break them using the ball that bounces on the paddle. The more bricks the agent breaks, the higher the score it gets. When the agent breaks all the bricks the environment is reset and the agent can start a new game. When the agent fails to intercept the ball, it loses one life, and when it loses all lives, it loses the game. Again, the goal of the game is to score as many points as possible.

Ms. Pacman

This game is a maze game where the agent controls a character that moves through the environment and collects various objects. The objects are represented by dots, fruits, or power pellets that are scattered throughout the maze. The agent navigates the maze searching for dots, the more dots the agent collects, the higher the score it gets. Various fruits and other bonus items appear in the maze at certain times. Eating these items grants additional points. The agent also has to avoid the ghosts that move through the maze, as they can kill the agent. The only way to kill the ghosts is to eat power pellets that appear in the maze from time to time and give the agent the ability to eat the ghosts. Eating ghosts also grants additional points. The game ends when the agent loses all its lives, or when it collects all the dots in the maze.

5.1.2 FMs Data and Training

For the main experiments of our approach, we use three different models which provide us with four different pre-trained networks: *Video Object Segmentation* (VOS) (Goel et al., 2018), *State Representation* (SR) (Anand



(a) `Pong` (b) `Ms. Pacman` (c) `Breakout`

Figure 5.1: A snapshot of the environments.

et al., 2019), and finally *Object Keypoints* (Kulkarni et al., 2019) which provides us the object encoder (OKE) and the object keypoint network (OKK). These models constitute the set of basic skills that the agent is equipped with. We believe that these models provide a good starting point for the agent to learn the environment, and they are informative enough to provide a good representation of the state. In fact, SR gives a general representation of the state in a compact form, VOS tracks moving objects in the frame which can be useful for tracking the ball in Pong or Breakout or the ghosts in Ms. Pacman, and finally, Object Keypoints provides the agent with the position of the objects in the frame.

Additionally, for attention-based combination modules, we train a deep autoencoder - inspired by Nature CNN (Mnih et al., 2015) - to encode the current state and leverage its representation to compute the context. The architecture of Nature CNN has been slightly modified as it appears in Tab. 5.2 to match the dimensions of other FMs' representations.

To train the different FMs and the Autoencoder we first create a specific dataset for each game. For each environment, we collected `1M` frames via agents interacting with the environment. The agents act randomly in this case, and the frames are collected without any reward. This is done to ensure that the data is not biased towards a specific policy. The dataset contains grayscale game frames with a size of 84x84 pixels. During the training process of the models, game frames are randomly sampled to avoid any correlation between elements due to the collection phase since the frames

Layer	In. Channels	Out. Channels	Kernel size	Stride
1st CNN Layer	1	32	8	4
2nd CNN Layer	32	64	3	1
3rd CNN Layer	64	64	3	1

Table 5.2: We provide the encoder architecture of the Deep Autoencoder. It takes as input only the last frame in grayscale. The stride on the second convolutional layer was decreased from `2` to `1` with respect to Nature CNN. Also, the kernel size is decreased from `4` to `3`. Each convolutional layer is followed by a ReLU activation function. The decoder part is specular.

are collected sequentially. We implemented and trained the models for all the games using the default architecture and hyperparameters provided by the authors.

Tables 5.3, 5.4, 5.5 report the hyperparameters used to train all models, the architecture presented in the original works is kept as the backbone of the models. Next, we provide a brief description of each model and the shape of their latent representation. For spatial representation, the first dimension denotes the number of feature maps while the second and third dimensions denote the width and the height of each one.

- `Video Object Segmentation` (Goel et al., 2018). This model takes as input the last two frames of the game and computes K feature maps that highlight the moving objects.
  Embedding shape: $[20 \times 84 \times 84]$.
- `State Representation` (Anand et al., 2019). Given the current state, it computes a linear representation exploiting the spatial-temporal nature of visual observations.
  Embedding shape: [512].
- `Object Keypoints` (Kulkarni et al., 2019). The model is composed of two heads that are used during inference: a convolutional network to compute spatial feature maps and a KeyNet (Jakab et al., 2018) to predict the keypoints co-ordinates.
  Embedding shape: CNN $[128 \times 21 \times 21]$, KeyNet $[4 \times 21 \times 21]$.
- `Deep Autoencoder`. It is used to encode the current state. It represents the context in attention-based combination modules like WSA

and DPA. Its architecture is inspired by NatureCNN (Mnih et al., 2015).
Embedding shape: $[64 \times 16 \times 16]$.

Hyperparameters	**Value**
Num. Tr. Envs	10
Episodes	1000
MAX ITER	1e6
Batch Size	64
Image Channels	1
K	4
Learning Rate	1e-3
Learning Rate Decay	0.95
Learning Rate Deacy Len	1e5

Table 5.3: Object keypoints hyperparameters (Kulkarni et al., 2019).

Hyperparameters	**Value**
Num. Frames	2
Steps	500.000
Batch Size	64
Learning Rate	1e-4
Max Grad. Norm	5.0
Learning Rate Deacy Len	1e5
Optimizer	Adam
K	20

Table 5.4: Video object segmentation hyperparameters (Goel et al., 2018).

Hyperparameters	**Value**
Image Width	160
Image Height	210
Grayscaling	Yes
Action Repetitions	4
Max-pool over last N action repeat frames	2
Frame Stacking	4
Batch size	64
Learning Rate (Training)	5e-4
Learning Rate (Probing)	3e-4
Entropy Threshold	0.6
Encoder training steps	80000
Probe training steps	30000
Probe test steps	10000
Epochs	100
Feature Size	512
Pretraining-steps	100000
Num Parallel Envs.	8

Table 5.5: State representation hyperparameters (Anand et al., 2019).

While training the agent, pre-trained models' weights are frozen and no longer updated. It is important to note that we build a model for each game, and we do not use a single model for all the games. These FMs are small models that can be easily changed to more complex models without affecting the structure of our work if needed.

5.2 Preliminary Experiments

As detailed in Section 5.1.1, our benchmark selection for this study includes three prominent Atari games, namely *Pong, Breakout, and Ms. Pacman,* The first choice of the games was dictated by the availability of the RAM annotations provided in Anand et al. (2019).

To maintain experimental clarity and efficiency, we decided to limit our testing scope to these specific games. This approach allows us to avoid unnecessary complexity and prolonged experimental durations. We believe that this choice of games provides a fair compromise between simple games like *Pong* where the agents just need to bounce the ball back, and much more complex games with many moving parts like *Ms. Pacman* where the agent needs more strategy to avoid ghosts or eventually eat them in order to complete the game.

For each environment, we selected the *NoFrameskip-v4* version of the game which is the most common version used in the literature. In this version, the agent can take an action in every frame.

After training each FM, we ran a first round of experiments using the three games. Our approach involved implementing the *Feature Extractor* using all the combination modules outlined in Section 4.1 one at a time, varying embedding sizes and configurations to thoroughly assess their performance. The *Fully-Connected Network* is set to a single layer of `256` units both for policy network and value function network.

As detailed in Table 5.6, our experiments explored diverse configurations for each module, ensuring a comprehensive evaluation of their effectiveness for our RL framework.

Feature Extractor	Embedding Size
LIN	-
FIX	256, 512, 1024
CNN	1, 2, 3
MIX	-
RES	512, 1024, 2048
DPA	256, 512, 1024
WSA	256, 512, 1024

Table 5.6: We show all the extractors' configurations tested in the initial phase of experiments. For FIX, DPA, and WSA the values indicate the fixed dimensions of embeddings and context, in output from the linear adapters. For RES it indicates the size of the reservoir and for CNN the number of convolutional layers used to extract features from the feature maps.

This first phase of experiments aims to find the `3` best-performing combination modules that will be used for our empirical analysis along with our proposed method. Agents are trained for `10M` steps using the parameters provided by `rl-zoo`, and throughout the learning process, agents are repeatedly evaluated for each `40.000` step for `100` episodes. During the training phase of these first experiments, we applied **early stopping** for agents that showed no improvement over `5` consecutive evaluations. This strategy allowed us to promptly identify the best-performing configurations and to save computational resources as these experiments are computationally expensive. In Appendix A.1 we report all the learning curves in this experimental phase.

Here instead we show our first results providing the best-performing modules for each game, the chosen embedding size is detailed between parentheses:

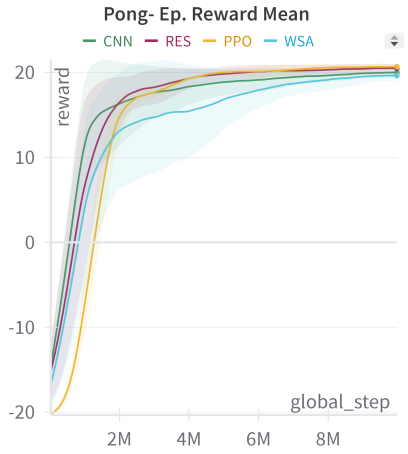
- `Pong`: WSA (1024), RES (1024), CNN (2)
- `Ms. Pacman`: WSA (256), RES (1024), CNN (2)
- `Breakout`: WSA (256), FIX (512), CNN (3)

5.3 Top 3 Performer

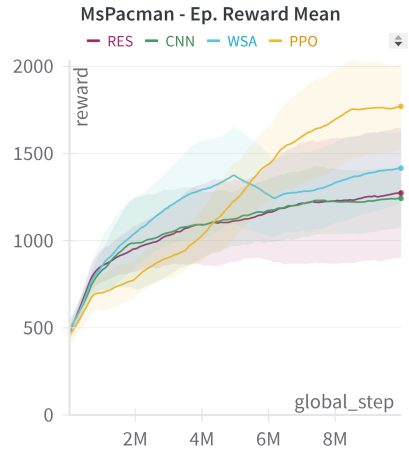
In the subsequent phase, the research progressed with a refined focus. We used the same methodology and setup as the initial experiments for our empirical evaluation, but we restricted the tests only to the best-performing modules for each game. This time we avoid using early stopping, and we let the agents train for the full `10M` steps. For each game, for each combination module i.e., for each agent, we executed multiple instances of the experiment using different seeds for a total of `4` runs per agent per game. The training results are averaged over the different seeds, and we also considered a Running Average of the cumulative reward to smooth the learning curves. We also included an agent trained with the standard PPO algorithm as a reference, so without pre-trained models and trained from scratch, for a total of `4` agents per game. Figure 5.2 shows the average learning curves of agents during training where the shaded area depicts the standard deviation.

Analyzing these results, with a focus on *Pong* Fig. 5.2a and *Ms. Pacman* Fig. 5.2b, we can see that our approach, WSA, and other combination modules, yield a high reward in the first stages of the training process. This suggests that FMs, as we hypothesized in the premises of our work, deliver valuable insights and a broad understanding of various elements in the environment without requiring additional training, as they offer an insightful and comprehensive base of knowledge straight out of the box. In the case of *Pong*, the performance of the agents is almost identical during training. The agents reach the maximum reward of `21` in the first stages of training. We notice that WSA in this case is a bit slower than the other agents to reach the maximum reward on average, but it is still able to reach it. On the other hand, in *Ms. Pacman*, the agents show a different behavior, exhibiting a more marked difference in performance during training between WSA and other combination modules. In the second phase of training, eventually, end-to-end PPO catches up and reaches a higher final reward score. This suggests that the FMs' representations are more effective in the early stages of training, providing the agent with many insights about the environment and allowing it to learn faster. The fact that end-to-end PPO reaches a higher final score may suggest that agents trained with FMs need to be refined to perform at their best. Again, it is important to note that we did not perform any hyperparameter search, and the agents' performance could be further improved by tuning the hyperparameters.

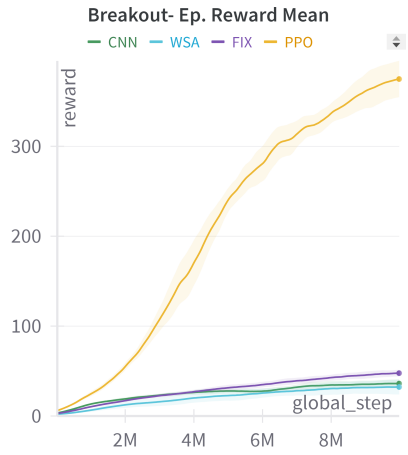
To evaluate our agents, we selected for each model the run that performed best during training, and we tested it across `5` random seeds for `50` episodes.



(a) `Pong`



(b) `Ms. Pacman`



(c) `Breakout`

Figure 5.2: Cumulative reward of the best-performing combination modules during training on different games. Each subfigure shows the mean score, with shaded areas indicating the standard deviations across multiple runs of the same agent.

The seeds specifically are `47695`, `32558`, `94088`, `71782` and `66638`. Table 5.7 reports the averaged results during the evaluation of the best agent.

Environment	Agent	Reward
Pong	**CNN**	**21 ± 0.00**
	RES	20.85 ± 0.29
	WSA	**21 ± 0.00**
	PPO	**21 ± 0.00**
Ms. Pacman	CNN	1801.30 ± 20.95
	RES	1369.27 ± 565.23
	WSA	**2530.20 ± 23.09**
	PPO	2258.40 ± 1.42
Breakout	CNN	65.98 ± 1.62
	FIX	87.17 ± 6.87
	WSA	99.58 ± 6.66
	PPO	**413.51 ± 1.10**

Table 5.7: Performance during evaluation averaged across 5 different seeds. In bold, the best-performing agent for each game.

Looking at the evaluation results, WSA matches the maximum reward (21) on *Pong* and achieves a higher score (2530) on *Ms. Pacman* than end-to-end PPO (2258). From these results, we can make two important considerations. The first one is the difference between training and evaluation scores. In particular, in *Ms. Pacman* is more marked, WSA provides a solid generalization for the task, scoring around 1250 during training compared to 2530 during evaluation. The second one concerns the difference compared to the PPO final score during learning. With respect to end-to-end solutions, our approach has a limited number of components that are updated during training, in particular, the *Fully-Connected Network* is only one layer. This could lead to underfitting, and the performance of the agent could be further improved by increasing the complexity of the model.

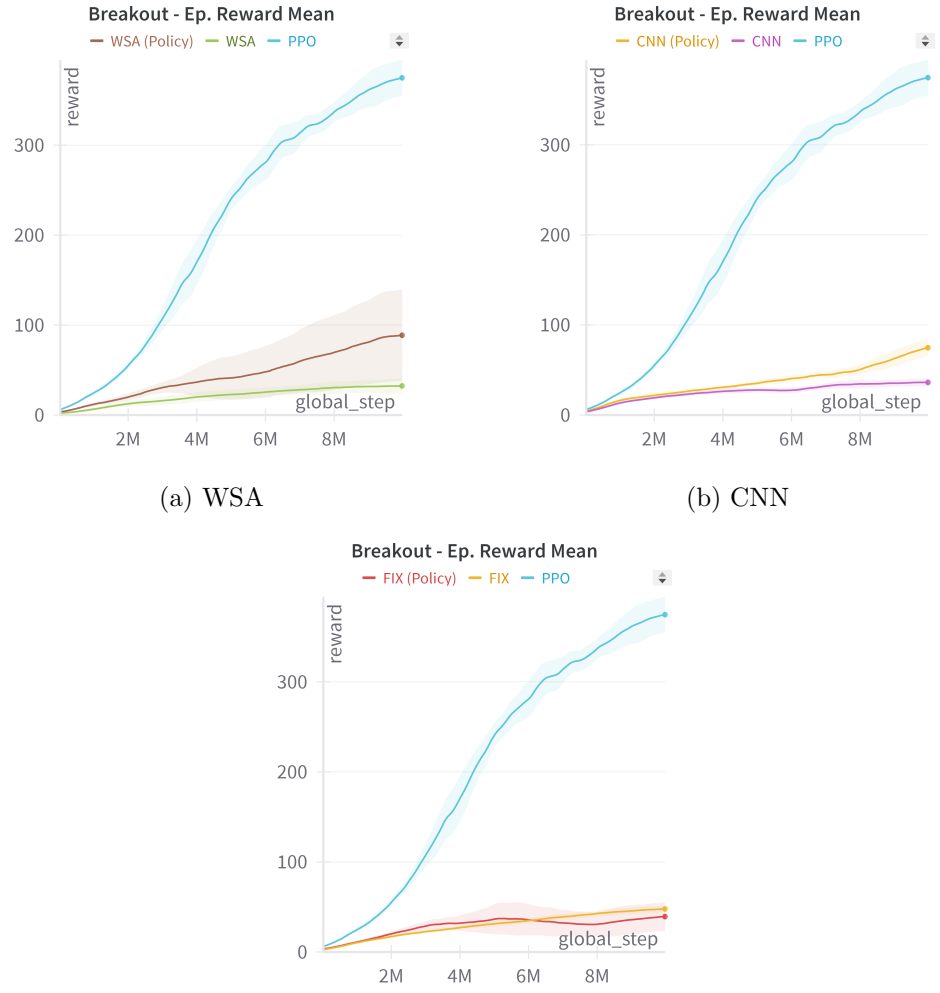
We notice that the results of *Breakout* are not as good as the other games. These results will be discussed in the next section, where we will analyze the reasons behind this behavior and propose some possible solutions to improve the performance of the agent in this game.

5.4 Breakout: Out of Distribution Data

While on *Pong* and *Ms. Pacman* the agents performed well, reaching the maximum reward during training and achieving high scores during evaluation, the results on *Breakout* were not as good. WSA and the other combination modules did not work right out of the box, and the agents struggled to learn the environment. Specifically, both in Figure 5.2c and in Table 5.7, the performance of WSA is extremely low compared to an end-to-end PPO (`99` vs `413`). So, in this Section, we will analyze the possible reasons for the behavior of the agents in this environment.

As we analyzed in Section 5.3, agents could suffer from underfitting problems due to the limited complexity of the model. So, a first attempt to overcome the problem was to increase the number of parameters of the model, adding more expressive power to the network that learns the policy. We believe that the *Fully-Connected Network* may be too simple to capture the complexity of the environment, and the agents could benefit from a bigger network. We increased the size of the policy learning network to three linear layers of size `1024`, `512`, and `256` respectively for the first, second, and third layer, and used ReLU as activation function to avoid vanishing gradients and allow the network to learn more complex patterns. Figures in 5.3 show the learning curve of the top three agents during training considering the augmented *Fully-Connected Network*.

With respect to the default scenario, Fig 5.2c, there is an improvement in performance for all the agents, and in particular, WSA almost doubled its score. This indicates that a more complex policy network can help the agent to learn the environment better, and can capture more information from the FMs' representations. The increased complexity allows the network to make better use of the detailed features extracted by the FMs, leading to more effective decision-making. However, despite these improvements, the agents' performance is still far from the end-to-end PPO. We can conclude that the problem is not only due to the complexity of the model.



(a) WSA

(b) CNN

(c) FIX

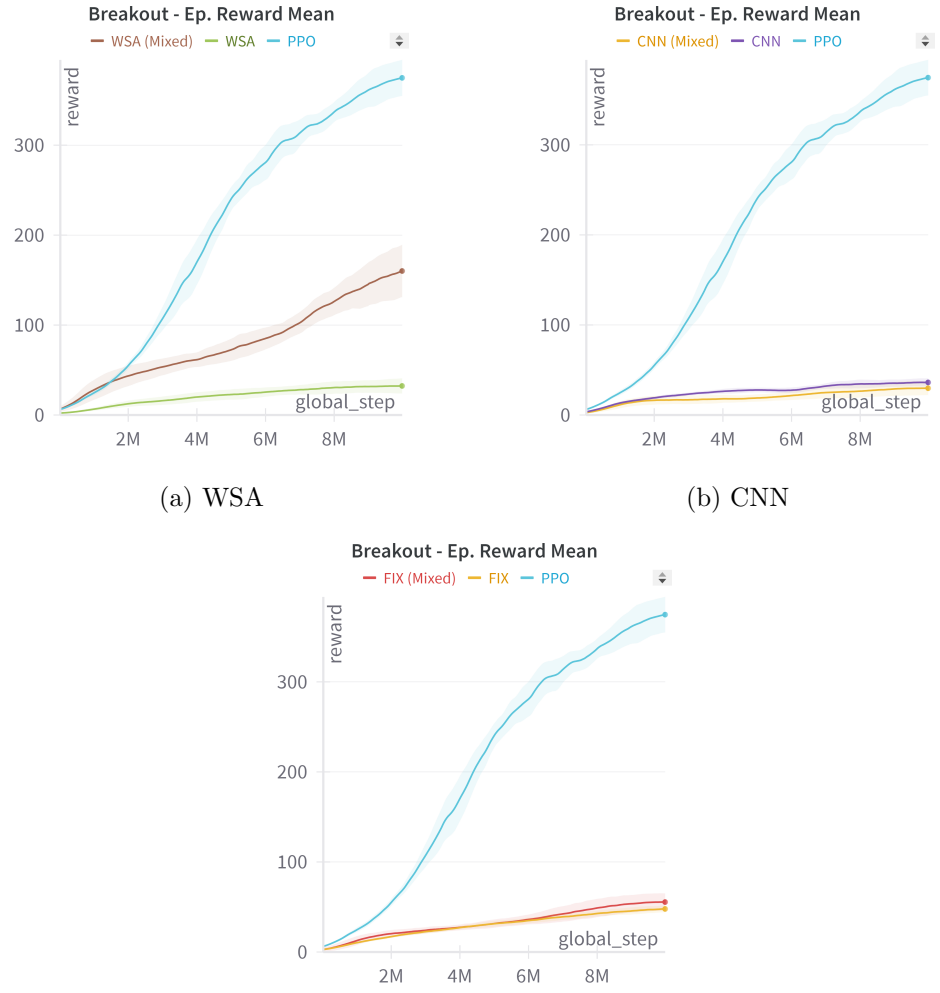
Figure 5.3: Performance comparison of different agents in *Breakout*. Each subfigure plots the averaged cumulative reward of the basic version, the version considering the augmented *Fully-Connected Network* and an end-to-end PPO. Shaded areas depict the standard deviation.

A second attempt to improve the performance of our agents was to better analyze the characteristics of the environments. Considering *Pong* and *Ms. Pacman*, the screen remains relatively stable as the game progresses. In *Pong* the only moving objects are the ball and the paddles, while in *Ms.*

Pacman the ghosts move in the maze but the only disappearing objects are the dots. In *Breakout* instead, the environment dynamically evolves over time as more blocks are removed with score progression. Leading to scenarios where the agent has to deal with fewer blocks and more open space. While in the first stage of the game, the agent can earn points just by intercepting and bouncing the ball back, in the late stages the agent has to be more precise and hit the ball in the right direction to break the remaining blocks. This poses a **distributional shift** problem between training and test data. Our initial training data, collected from random agents, lacks late-game scenarios with few blocks remaining. This could lead to situations where the FMs are not able to generalize well to unseen data and produce representations that are not informative enough for the agent. For example, FMs like *OKK* could output the wrong position of the ball or the blocks, and this could lead to a wrong decision by the agent.

To address the problem and illustrate the consequences of a limited training dataset, we took several steps. Initially, we trained an end-to-end PPO agent. We used it to collect new data, leveraging its capacity to master the game and reach late-game scenarios. We collected a dataset consisting of both random data and expert agents data, to cover early and final game scenarios. Then, we retrain all the FMs and RL agents using a single layer *Fully-Connected Network* of `256` units as before. Significant improvements are observed, as shown in Figure 5.4.

Notably, WSA demonstrates a substantial increase in the agent's final score, approaching the performance of PPO. This suggests that the agent's performance is highly dependent on the quality of the training data, and the agent can perform at most as well as the FMs do. The better the FMs are, the better the agent will perform. Unlikely, as we can see in Figures 5.4b and 5.4c, the other combination modules do not show the same improvement in training, and the agents' performance is still lower than WSA. This may be due to the fact that WSA can weight the contribution of each pre-trained model, such that can ignore the less informative ones. Even though the FMs are trained with mixed data, there may be some scenarios where they are not able to provide the right information, and the agent could benefit from ignoring them. For example, FMs that track moving objects can struggle to identify correctly the ball in a game. Considering Atari games and in particular *Breakout*, the ball is typically very small and might be considered as an unimportant part of the frame by a model, even though it is the most critical element for understanding the context.



(a) WSA

(b) CNN

(c) FIX

Figure 5.4: Performance comparison of different agents in *Breakout*. Each subfigure plots the averaged cumulative reward of the basic version, the version considering the *Mixed* data. and an end-to-end PPO. Shaded areas depict the standard deviation.

The other combination modules perform a simple concatenation or extract patterns from the whole input. They do not have the capability of selecting part of the feature, and they are forced to use all the pre-trained models, even the less informative ones. This lack of selective focus can significantly

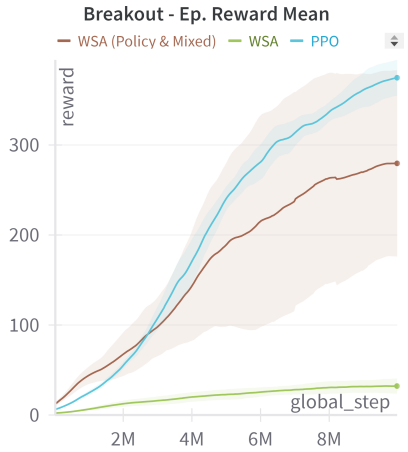
inhibit the agent's performance.

To complete the set of experiments, we also tested the combination of the two previous configurations. We increased the complexity of the *Fully-Connected Network* and we used mixed datasets to further evaluate the impact on agent performance. Figures in 5.5, show the results of these experiments.

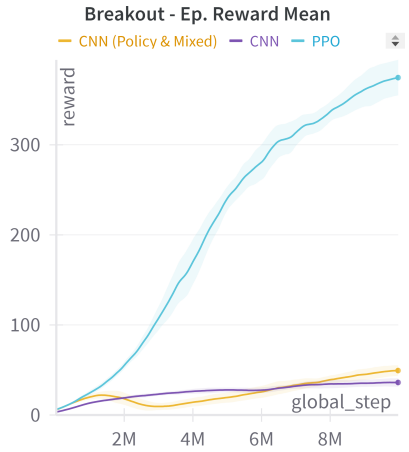
We notice how the combination of the two strategies produces the best-performing configuration of WSA with a slightly lower but competitive score to PPO. While the other agents did not achieve the desired results.

Evaluation results of all the attempts are reported in Table 5.8 where we refer to *Policy* for the experiments regarding the first attempt, *Mixed* for the second, and *Policy & Mixed* for the experiments regarding the third attempt, as it is a mixture of both. Regarding the first attempt, WSA scored `156` compared to `413` of PPO. The second attempt, as shown during the analysis of the learning curves, is the one that contributed to the greatest leap in performance, and the score went from `156` to `345`. Also, FIX shows a significant improvement in evaluation score, going from `87` to `199`. Finally, with the third attempt, WSA achieved the best score and obtained comparable results with PPO, `387` vs `413`.

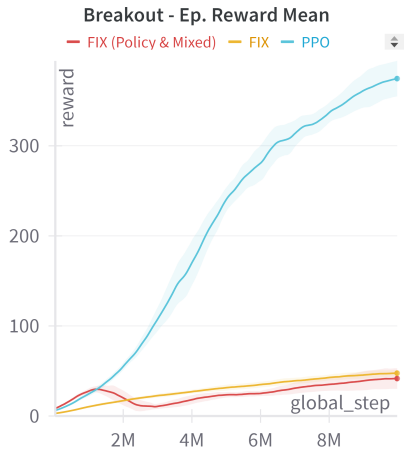
As a final note, it is important to underline that we conducted these experiments considering only the top-performing modules for each game highlighted in Section 5.3 and we did not test again all the combination modules with different configurations.



(a) WSA



(b) CNN



(c) FIX

Figure 5.5: Performance comparison of different agents in *Breakout*. Each subfigure plots the averaged cumulative reward of the basic version, the version considering the augmented *Fully-Connected Network* and the use of mixed data, and an end-to-end PPO. Shaded areas depict the standard deviation.

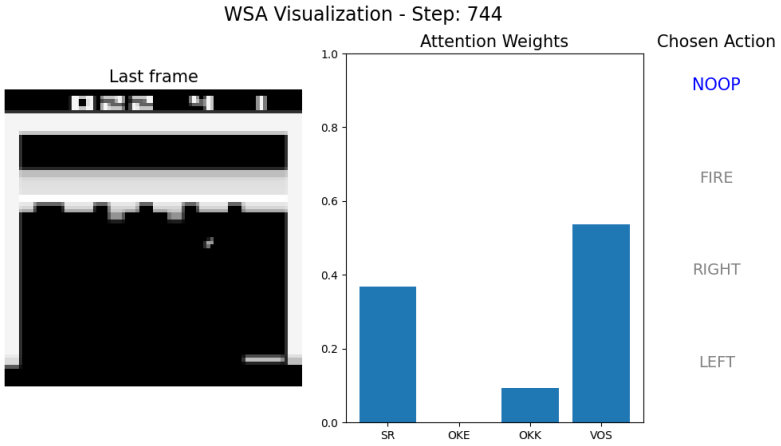
Environment	**Agent**	**Reward**
`Breakout`	CNN	65.98 ± 1.62
	FIX	87.17 ± 6.87
	WSA	99.58 ± 6.66
`Breakout - Policy`	CNN	118.71 ± 4.30
	FIX	106.46 ± 10.84
	WSA	156.17 ± 3.59
`Breakout - Mixed`	CNN	62.21 ± 1.98
	FIX	199.08 ± 7.46
	WSA	345.52 ± 6.47
`Breakout - Policy & Mixed`	CNN	68.51 ± 1.85
	FIX	71.06 ± 5.04
	WSA	387.15 ± 0.43
	PPO	**413.51 ± 1.10**

Table 5.8: We show the cumulative reward obtained by the agents using different strategies. With *Policy*, we denote an agent that was trained increasing the dimension of the *Fully-Connected* network for policy learning. With *Mixed* we refer to an agent that was trained using FMs pre-trained with random data and expert data. With *Policy & Mixed* we refer to an agent that was trained with both the strategies. We see how WSA reaches comparable results with an end-to-end PPO by using both strategies.

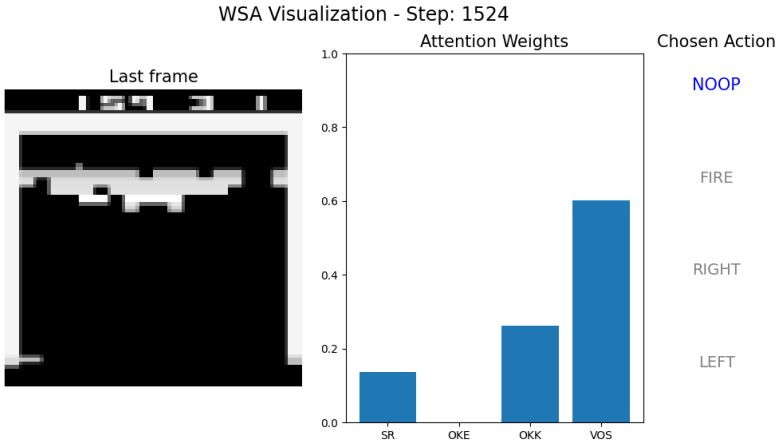
5.5 Weight Sharing Attention Explainability

In this Section, we provide an analysis of the explainability of the WSA module. Thanks to the nature of the attention mechanism, we can analyze the weights assigned to each pre-trained model in different scenarios. This can provide insights into how the agent uses the FMs to make decisions and how the agent adapts its prior knowledge to the environment.

Figure 5.6 shows the attention weights assigned by the WSA module to each pre-trained model during the evaluation phase of the agent on *Breakout*. The frames depict also the current frame of the game and the chosen action by the agent.



(a) The first scenario depicts the early stages of the game. The agent assigns more weight to SR and VOS.



(b) The second scenario shows the late stages of the game. SR becomes less relevant and the agent assigns more weight to OKK and VOS.

Figure 5.6: The attention weights assigned by the WSA module to each pre-trained model during the evaluation phase of the agent on *Breakout* are illustrated. It is possible to appreciate different weight combinations in two separate game situations.

We analyzed the agent playing the whole game and decided to consider two different behaviors in two scenarios. In the first one, the agent is in the

early stages of the game with many blocks, at this stage, it only needs to bounce the ball as he hits the blocks very easily. The second scenario that we considered is one where the agent is in the late stages of the game and needs to be more precise to hit the ball in the right direction to break the bricks. Considering the two frames, we can see how OKK and VOS are most important in the end-game scenarios rather than early-stage scenarios. OKK and VOS in fact both track moving objects like the ball or the paddle. So, their information appears very helpful at the end of the game as the agent needs this information to make the right decision.

Regarding the other two games, we analyzed the attention weights assigned by the WSA module and discovered that weights collapsed on a single model. In specific, in *Pong* the agent always assigns the same weight to the OKK, while in *Ms. Pacman* attention weights collapsed on the VOS model. This is a well-known problem in attention mechanisms, and it is called *attention collapse*. Fig. 5.7 highlights the attention weights collapse problem on *Pong*.

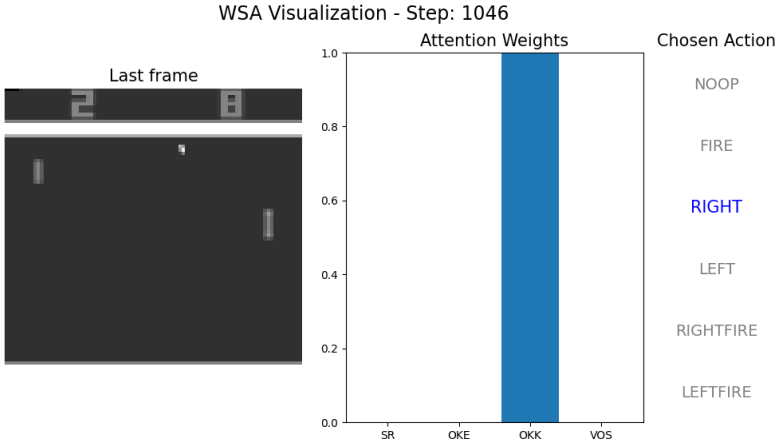


Figure 5.7: Attention weight collapse problem on *Pong*. Weights collapsed to OKK model.

Fig. 5.8 highlights the attention weights collapse problem on *Ms. Pacman*.

To solve this issue different strategies have been tried that we will list below.

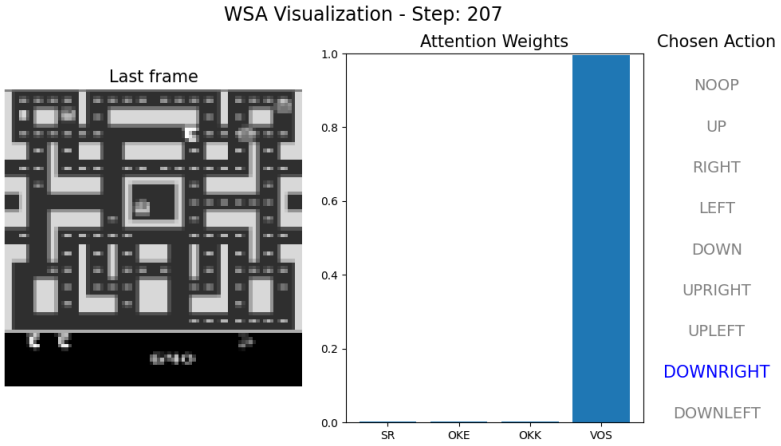


Figure 5.8: Attention weight collapse problem on *Ms. Pacman*. Weights collapsed to VOS model.

Dropout

Dropout is a regularization technique used in DL to prevent overfitting. It consists of randomly setting some weights to zero during training, with a probability p. During the evaluation phase instead, dropout is not applied. We add a dropout layer after the linear adapters for the skills, after the linear adapter for the state, or after the softmax layer for the weights. We tested different values of p, in specific, `0.1`, `0.2` and `0.3`. We observed that in *Pong* using p=0.1, the weights still collapsed on the same model but differently from before, when the state does not expose the ball, the agent assigns more weight to the VOS model. Using p=0.2 and p=0.3, lead to a uniform distribution of the weights at each step. Regarding *Ms. Pacman* the agent could not learn using p=0.2 and p=0.3, while using p=0.1 the weights are uniformly distributed at each step.

Batch Normalization

Batch normalization is a technique used to normalize the input of a layer so that the mean is zero and the variance is one. We add a batch normalization layer after the linear adapters for the skills, or after the linear adapter for the state. In this case, both for *Pong* and *Ms. Pacman* the agent was not able to learn properly.

Changing Activation Function

Considering the structure of our agents, we use ReLU as an activation function for the adapters. This is a common choice in DL, but regarding attention mechanisms, ReLU could delete some information as it sets to zero all the negative values. We then tried to remove the ReLU activation function and use a linear activation function both in the linear and spatial adapters. In both games, we discover that agents were not able to learn properly or the weights were uniformly distributed at each step. We also made another experiment where we used a Sigmoid activation function instead of ReLU, but the results were the same.

Entropy Penalty Term

Finally, we consider adding a penalty term to the loss function of PPO based on the entropy of the attention weights. Low entropy means that the weights are concentrated on a single model, while high entropy means that the weights are uniformly distributed. In specific, we computed the entropy of the weights at each time step as in Equation 5.1, where w_i is the weight assigned to the i-th model, N is the number of models. ϵ is a small value to avoid the logarithm of zero. Since we run multiple environments in parallel, we computed the mean entropy across all the environments. We then added the entropy term to the loss function of PPO with a coefficient of `0.01`.

$$H = -\sum_{i=1}^{N} w_i \log(w_i + \epsilon) \tag{5.1}$$

As for the results, in *Pong* weights still collapse on the same model, while in *Ms. Pacman* the weights are uniformly distributed at each step and agents were not able to learn.

Considerations

The fact that regularization techniques did not solve the problem may suggest that the FMs are already too informative for these two games. Each pre-trained model learns the environment well, and the agent does not need to use all of them to make decisions. It does not matter which model the agent uses, the agent will perform well anyway. It is important to notice

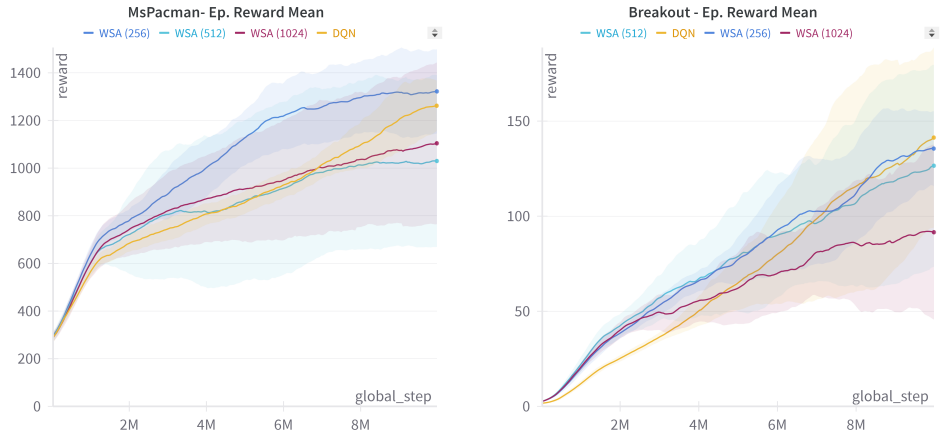
that each strategy is applied independently of the others, and a combination of them could lead to better results.

5.6 Deep Q-Learning Experiments

One last experiment is the one to prove that our approach is not limited to the PPO algorithm, but is independent of the RL algorithm used. To do so, we tested the effectiveness of the WSA module in combination with a Deep Q-learning agent. As WSA is the main focus of our work, we decided to test only this module with different embedding sizes. We did not test the other combination modules, this is to save computational resources and time since we already proved that combination modules are effective in RL, so we assume that they work also with DQL. Using the same methodology and structure of experiments of the main work, we study the performance of the module. We also restricted to test only the games of *Breakout* and *Ms. Pacman* as *Pong* is very simple, and regarding *Breakout* we FMs trained on the mixed dataset.

Figure 5.9 shows that WSA works very well in comparison to the standard DQL agent, and the one with an embedding size of `256` results in the best performance. In *Ms. Pacman* the average cumulative reward is higher than DQL for all the training, meaning that the agent learns faster and better. While in *Breakout* it is faster to learn on the first part of the training, and in the end DQL results a little bit better. This is a behavior similar to the one observed with PPO agents in Section 5.4.

We selected for evaluation the agent with `256` as embedding size since it is the best performing during training. This time to evaluate the agents, instead of selecting the best model between the `4` runs during training, we tested all the models across `5` random seeds for `50` episodes. Table 5.9 reports the averaged results during evaluation. We notice how WSA results in a higher score in both games and in particular in *Breakout* the agent scores `213` compared to `166` of DQL. Please note that PPO agents reached a higher score in *Breakout* and these results are not directly comparable as the two algorithms have different learning dynamics and different ways to interact with the environment. Lastly, we can conclude that WSA works well regardless of the RL algorithm used, and it can be a valuable tool to improve the performance of agents in different scenarios.



(a) `Ms. Pacman` (b) `Breakout`

Figure 5.9: Cumulative reward during training of different agents, comparing WSA and DQL. Each subfigure shows the mean score, with shaded areas indicating the standard deviations, across multiple agents. The number in parenthesis indicates embedding size.

Environment	**Agent**	**Reward**
`Ms. Pacman`	**WSA**	**2047.27 ± 231.18**
	DQL	1701.00 ± 490.41
`Breakout`	**WSA**	**213.14 ± 39.37**
	DQL	166.65 ± 20.19

Table 5.9: Performance during evaluation averaged across 5 different seeds, using WSA with the embedding size of 256.

5.7 Final considerations

In this Section, we want to provide some final considerations about our work. We have shown that our approach is effective in improving the performance of agents in different scenarios, and it can be applied to different RL algorithms. Also, we have shown how the quality of the training data is crucial for the performance of the agent and how WSA can provide explainability to the decision-making process of the agent. But there are still some

considerations that need to be addressed.

First of all, the computational cost of our approach. Using pre-trained models can be computationally expensive, and it can be a bottleneck in the training process. In fact, compared to an end-to-end solution, our approach is able to process more or less half of the frames per second [1]. This is due to the fact that, even if the pre-trained model weights are frozen during training, they need to process the frames before the agent can make a decision. As the number of FMs increases, the computational cost increases and the agent needs to wait more time to make a decision. In our implementation, the frames are processed sequentially, and the agent needs to wait for all the FMs to provide their representation before making a decision. This can be a problem in real-time applications, where the agent needs to make decisions in a short amount of time. A possible solution to this problem could be to process the frames in parallel, and to make the agent wait only for the slowest FM to provide its representation.

Another consideration is in the number of trainable parameters of the whole model. Different combination modules have different numbers of parameters and may result in different computational costs. Table 5.10 shows the number of parameters of the different combination modules compared to an end-to-end PPO. We would like to remember that to calculate the number of the trainable parameters we considered only the models with *Fully-Connected Network* of `256` units.

Feature Extractor	**Parameters**
LIN	`8.7M`
FIX	`4.9M–19.4M`
CNN	`4.2M`
MIX	`4.5M`
RES	`0.3M–1.1M`
DPA	`8.7M–34.7M`
WSA	`8.7M–34.7M`
PPO	`1.6M`

Table 5.10: We provide the number of trainable parameters of the whole model, from the smallest configuration to the biggest one.

[1] The training time is measured considering a shared server and can be influenced by other processes running on the machine.

We notice how for some agents the number of parameters is higher than an end-to-end PPO. The aim of our work is to provide a more general and effective solution to RL problems, and we did not consider the number of parameters as a constraint.

Chapter 6

Conclusions

In this thesis, we analyzed the problem of RL algorithms given from the state representation learning during training. In particular, typical RL agents learn an end-to-end mapping from observation to actions, spending much time trying to obtain a useful state representation from which to choose an action to accomplish. We proposed a method to equip RL agents with prior knowledge, i.e., skills, represented by unsupervised or self-supervised pre-trained models capable of extracting different features from the current observation. This simplifies the representation learning of the environment's state, allowing the agent to learn faster. We proposed, among the other techniques, WSA, as a combination method to mix various skills. We tested our method on three *Atari* games, namely *Pong, Breakout* and *Ms. Pacman*. We showed how combining different encodings is effective and useful for the agent in order to learn faster in two of three environments, *Pong, and Ms. Pacman*. We also analyzed the out-of-distribution problem in *Breakout*, where agents do not perform well when in the presence of never-seen states due to the failure of skills or not completing prior knowledge. About this, we proposed different strategies to solve the problem for example to pre-train FMs with data collected by a random and an expert agent that plays the game. This means that the quality of the training data used for skills is crucial for the performance of the agent. We showed also that our approach is independent of the RL algorithm. In fact, we conducted experiments using two different procedures, i.e., PPO and DQL. Considering PPO, the performance is comparable, while our agent performs better with respect to DQL. Finally, we also analyzed the explainability of WSA considering different scenarios of the environment.

While the results of this thesis are being finalized to be submitted to a conference, we highlight some future works and promising directions to extend the work and address some limitations.

- *Hyperparameters Search* - While for end-to-end PPO we used the fine-tuned hyperparameters, our methods miss a hyperparameters search. This could improve the performance of the agents.
- *More Experiments* - Conducting such experiments is time-consuming, and performing more extensive tests would have been outside the purpose of this thesis. More tests are needed to obtain more reliable results on average.
- *Other Benchmarks* - To test the effectiveness of WSA, it is important to test it on other environments such as other games or completely different tasks like autonomous driving or robot control.
- *Different Skills* - Studying the behavior of the proposed agents by varying the number of skills, searching for the best ones for different use-cases, perhaps scaling to very big FMs (Winterbottom et al., 2024; Zhao et al., 2023; Dosovitskiy et al., 2021; Singh et al., 2022).
- *WSA Explainability* - We showed that WSA is helpful in providing insights about the most relevant models at the current timestep. A more in-depth study is needed considering scaling with new skills and addressing new tasks.

Bibliography

R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994. URL http://www.vldb.org/conf/1994/P487.PDF.

A. Anand, E. Racah, S. Ozair, Y. Bengio, M. Côté, and R. D. Hjelm. Unsupervised state representation learning in atari. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8766–8779, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/6fb52e71b837628ac16539c1ff911667-Abstract.html.

A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, et al. Agent57: Outperforming the atari human benchmark. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 507–517. PMLR, 2020. URL http://proceedings.mlr.press/v119/badia20a.html.

R. Balestriero, M. Ibrahim, V. Sobal, A. Morcos, S. Shekhar, T. Goldstein, F. Bordes, A. Bardes, G. Mialon, Y. Tian, A. Schwarzschild, A. G. Wilson, J. Geiping, Q. Garrido, P. Fernandez, A. Bar, H. Pirsiavash, Y. LeCun, and M. Goldblum. A cookbook of self-supervised learning. *CoRR*, abs/2304.12210, 2023. doi: 10.48550/ARXIV.2304.12210. URL https://doi.org/10.48550/arXiv.2304.12210.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell.*

Res., 47:253–279, 2013. doi: 10.1613/jair.3912. URL https://doi.org/10.1613/jair.3912.

R. Bellman. Dynamic programming. *science*, 153(3731):34–37, 1966.

C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, et al. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019. URL http://arxiv.org/abs/1912.06680.

L. Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

S. Blakeman and D. Mareschal. Selective particle attention: Rapidly and flexibly selecting features for deep reinforcement learning. *Neural Networks*, 150:408–421, 2022. doi: 10.1016/j.neunet.2022.03.015. URL https://doi.org/10.1016/j.neunet.2022.03.015.

K. Bousmalis, G. Vezzani, D. Rao, C. Devin, A. X. Lee, et al. Robocat: A self-improving foundation agent for robotic manipulation. *CoRR*, abs/2306.11706, 2023. doi: 10.48550/arXiv.2306.11706. URL https://doi.org/10.48550/arXiv.2306.11706.

L. Bramlage and A. Cortese. Generalized attention-weighted reinforcement learning. *Neural Networks*, 145:10–21, 2022. doi: 10.1016/j.neunet.2021.09.023. URL https://doi.org/10.1016/j.neunet.2021.09.023.

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984. ISBN 0-534-98053-8.

T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *Advances in neural information processing systems*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

G. Carfì. Skilled rl. https://github.com/EliaPiccoli/skilled_rl, 2024.

X. Carreras and L. Màrquez. Boosting trees for anti-spam email filtering. *CoRR*, cs.CL/0109015, 2001. URL https://arxiv.org/abs/cs/0109015.

T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton. A simple framework for contrastive learning of visual representations. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 1597–1607. PMLR, 2020. URL http://proceedings.mlr.press/v119/chen20j.html.

C. Cortes and V. Vapnik. Support-vector networks. *Mach. Learn.*, 20 (3):273–297, 1995. doi: 10.1007/BF00994018. URL https://doi.org/10.1007/BF00994018.

Z. Dai, Z. Yang, Y. Yang, J. G. Carbonell, Q. V. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, pages 2978–2988, 2019. doi: 10.18653/V1/P19-1285. URL https://doi.org/10.18653/v1/p19-1285.

J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

H. Dong, Z. Ding, S. Zhang, H. Yuan, H. Zhang, J. Zhang, Y. Huang, T. Yu, H. Zhang, and R. Huang. *Deep Reinforcement Learning: Fundamentals, Research, and Applications*. Springer Nature, 2020. http://www.deepreinforcementlearningbook.org.

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. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.

R. Dubey, P. Agrawal, D. Pathak, T. Griffiths, and A. A. Efros. Investigating human priors for playing video games. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1348–1356. PMLR, 2018. URL http://proceedings.mlr.press/v80/dubey18a.html.

M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In E. Simoudis, J. Han, and U. M. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-*

96), Portland, Oregon, USA, pages 226–231. AAAI Press, 1996. URL http://www.aaai.org/Library/KDD/1996/kdd96-037.php.

A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nat.*, 542(7639):115–118, 2017. doi: 10.1038/NATURE21056. URL https://doi.org/10.1038/nature21056.

C. Finn, I. J. Goodfellow, and S. Levine. Unsupervised learning for physical interaction through video prediction. *Advances in neural information processing systems*, pages 64–72, 2016. URL https://proceedings.neurips.cc/paper/2016/hash/d9d4f495e875a2e075a1a4a6e1b9770f-Abstract.html.

K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

C. Gallicchio, A. Micheli, and L. Pedrelli. Deep reservoir computing: A critical experimental analysis. *Neurocomputing*, 268:87–99, 2017. doi: 10.1016/J.NEUCOM.2016.12.089. URL https://doi.org/10.1016/j.neucom.2016.12.089.

V. Goel, J. Weng, and P. Poupart. Unsupervised video object segmentation for deep reinforcement learning. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 5688–5699, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/96f2b50b5d3613adf9c27049b2a888c7-Abstract.html.

D. Hafner, T. P. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson. Learning latent dynamics for planning from pixels. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 2555–2565. PMLR, 2019. URL http://proceedings.mlr.press/v97/hafner19a.html.

D. Hafner, T. P. Lillicrap, J. Ba, and M. Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2020. URL https://openreview.net/forum?id=S1lOTC4tDS.

T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer

Series in Statistics. Springer, 2009. ISBN 9780387848570. doi: 10.1007/978-0-387-84858-7. URL https://doi.org/10.1007/978-0-387-84858-7.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90.

K. He, H. Fan, Y. Wu, S. Xie, and R. B. Girshick. Momentum contrast for unsupervised visual representation learning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 9726–9735. Computer Vision Foundation / IEEE, 2020. doi: 10.1109/CVPR42600.2020.00975. URL https://doi.org/10.1109/CVPR42600.2020.00975.

G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Process. Mag.*, 29(6):82–97, 2012. doi: 10.1109/MSP.2012.2205597. URL https://doi.org/10.1109/MSP.2012.2205597.

R. A. Howard. Dynamic programming and markov processes. 1960.

T. Jakab, A. Gupta, H. Bilen, and A. Vedaldi. Unsupervised learning of object landmarks through conditional image generation. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 4020–4031, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/1f36c15d6a3d18d52e8d493bc8187cb9-Abstract.html.

G. James, D. Witten, T. Hastie, R. Tibshirani, et al. *An introduction to statistical learning*, volume 112. Springer, 2013.

S. C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967.

I. T. Jolliffe et al. Principal component analysis and factor analysis. *Principal component analysis*, 372:115–128, 1986.

S. M. Kakade. A natural policy gradient. *Advances in neural information processing systems*, pages 1531–1538, 2001.

URL https://proceedings.neurips.cc/paper/2001/hash/4b86abe48d358ecf194c56c69108433e-Abstract.html.

B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. K. Yogamani, and P. Pérez. Deep reinforcement learning for autonomous driving: A survey. *CoRR*, abs/2002.00444, 2020. URL https://arxiv.org/abs/2002.00444.

V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, pages 1008–1014, 1999. URL http://papers.nips.cc/paper/1786-actor-critic-algorithms.

S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica (Slovenia)*, 31(3):249–268, 2007. URL http://www.informatica.si/index.php/informatica/article/view/148.

T. D. Kulkarni, A. Gupta, C. Ionescu, S. Borgeaud, M. Reynolds, et al. Unsupervised learning of object keypoints for perception and control. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10723–10733, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/dae3312c4c6c7000a37ecfb7b0aeb0e4-Abstract.html.

C. L. Lan, S. Tu, M. Rowland, A. Harutyunyan, R. Agarwal, et al. Bootstrapped representations in reinforcement learning. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18686–18713. PMLR, 2023. URL https://proceedings.mlr.press/v202/le-lan23a.html.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791. URL https://doi.org/10.1109/5.726791.

Y. LeCun, Y. Bengio, and G. E. Hinton. Deep learning. *Nat.*, 521(7553): 436–444, 2015. doi: 10.1038/NATURE14539. URL https://doi.org/10.1038/nature14539.

S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *Int. J. Robotics Res.*, 37(4-5):421–436, 2018. doi: 10.1177/0278364917710318. URL https://doi.org/10.1177/0278364917710318.

W. Liao, F. Porté-Agel, J. Fang, C. Rehtanz, S. Wang, D. Yang, and Z. Yang. Timegpt in load forecasting: A large time series model perspective. *CoRR*, abs/2404.04885, 2024. doi: 10.48550/ARXIV.2404.04885. URL https://doi.org/10.48550/arXiv.2404.04885.

S. Lifshitz, K. Paster, H. Chan, J. Ba, and S. A. McIlraith. STEVE-1: A generative model for text-to-behavior in minecraft. *Advances in Neural Information Processing Systems*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/dd03f856fc7f2efeec8b1c796284561d-Abstract-Conference.html.

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. *arXiv preprint arXiv:1509.02971*, 2016. URL http://arxiv.org/abs/1509.02971.

J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

A. McCallum, K. Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Madison, WI, 1998.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nat.*, 518(7540):529–533, 2015. doi: 10.1038/NATURE14236. URL https://doi.org/10.1038/nature14236.

J. Montalvo, Á. García-Martín, and J. Bescós. Exploiting semantic segmentation to boost reinforcement learning in video game environments. *Multim. Tools Appl.*, 82(7):10961–10979, 2023. doi: 10.1007/S11042-022-13695-1. URL https://doi.org/10.1007/s11042-022-13695-1.

J. S. Obando-Ceron, G. Sokar, T. Willi, C. Lyle, J. Farebrother, J. N. Foerster, G. K. Dziugaite, D. Precup, and P. S. Castro. Mixtures of experts unlock parameter scaling for deep RL. *CoRR*, abs/2402.08609, 2024.

doi: 10.48550/ARXIV.2402.08609. URL https://doi.org/10.48550/arXiv.2402.08609.

OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, et al. Solving rubik's cube with a robot hand. *CoRR*, abs/1910.07113, 2019. URL http://arxiv.org/abs/1910.07113.

R. Orlando, L. Moroni, P.-L. H. Cabot, S. Conia, E. Barba, and R. Navigli. Minerva large language model. https://nlp.uniroma1.it/minerva/blog, 2024. Accessed: 2024-06-18.

G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71, 2019. doi: 10.1016/J.NEUNET.2019.01.012. URL https://doi.org/10.1016/j.neunet.2019.01.012.

J. Puigcerver, C. Riquelme, B. Mustafa, and N. Houlsby. From sparse to soft mixtures of experts. *CoRR*, abs/2308.00951, 2023. doi: 10.48550/ARXIV.2308.00951. URL https://doi.org/10.48550/arXiv.2308.00951.

A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. Improving language understanding by generative pre-training. 2018.

A. Raffin. Rl baselines3 zoo. https://github.com/DLR-RM/rl-baselines3-zoo, 2020.

A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. Zero-shot text-to-image generation. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8821–8831. PMLR, 2021. URL http://proceedings.mlr.press/v139/ramesh21a.html.

R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pages 10674–10685. IEEE,

2022. doi: 10.1109/CVPR52688.2022.01042. URL https://doi.org/10.1109/CVPR52688.2022.01042.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. URL https://www.nature.com/articles/323533a0.

H. Sahni, S. Kumar, F. Tejani, and C. L. I. Jr. Learning to compose skills. *CoRR*, abs/1711.11289, 2017. URL http://arxiv.org/abs/1711.11289.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

R. M. Shah and V. Kumar. RRL: resnet as representation for reinforcement learning. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 9465–9476. PMLR, 2021. URL http://proceedings.mlr.press/v139/shah21a.html.

N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017. URL https://openreview.net/forum?id=B1ckMDqlg.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, et al. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529 (7587):484–489, 2016. doi: 10.1038/nature16961. URL https://doi.org/10.1038/nature16961.

A. Singh, R. Hu, V. Goswami, G. Couairon, W. Galuba, M. Rohrbach, and D. Kiela. FLAVA: A foundational language and vision alignment model. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pages 15617–15629. IEEE, 2022. doi: 10.1109/CVPR52688.2022.01519. URL https://doi.org/10.1109/CVPR52688.2022.01519.

S. Sodhani, A. Zhang, and J. Pineau. Multi-task reinforcement learning with context-based representations. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 9767–9779. PMLR, 2021. URL http://proceedings.mlr.press/v139/sodhani21a.html.

A. Stooke, K. Lee, P. Abbeel, and M. Laskin. Decoupling representation learning from reinforcement learning. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 9870–9879. PMLR, 2021. URL http://proceedings.mlr.press/v139/stooke21a.html.

R. S. Sutton. Learning to predict by the methods of temporal differences. *Mach. Learn.*, 3:9–44, 1988. doi: 10.1007/BF00115009. URL https://doi.org/10.1007/BF00115009.

R. S. Sutton and A. G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998. ISBN 978-0-262-19398-6. URL https://www.worldcat.org/oclc/37293240.

R. S. Sutton, D. A. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, pages 1057–1063, 1999. URL http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.

S. Team, M. A. Raad, A. Ahuja, C. Barros, F. Besse, et al. Scaling instructable agents across many simulated worlds. *CoRR*, abs/2404.10179, 2024. doi: 10.48550/ARXIV.2404.10179. URL https://doi.org/10.48550/arXiv.2404.10179.

J. Tenenbaum. Building machines that learn and think like people. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, page 5. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018. URL http://dl.acm.org/citation.cfm?id=3237389.

C. Thammineni, H. Manjunatha, and E. T. Esfahani. Selective eye-gaze augmentation to enhance imitation learning in atari games. *Neural Comput. Appl.*, 35(32):23401–23410, 2023. doi: 10.1007/S00521-021-06367-Y. URL https://doi.org/10.1007/s00521-021-06367-y.

M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, et al. Gymnasium, Mar. 2023. URL https://zenodo.org/record/8127025.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need.

Advances in neural information processing systems, pages 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, et al. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019. doi: 10.1038/s41586-019-1724-z. URL https://doi.org/10.1038/s41586-019-1724-z.

G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models. *CoRR*, abs/2305.16291, 2023. doi: 10.48550/ARXIV.2305.16291. URL https://doi.org/10.48550/arXiv.2305.16291.

C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256, 1992. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

T. Winterbottom, G. T. Hudson, D. Kluvanec, D. L. Slack, J. Sterling, J. Shentu, C. Xiao, Z. Zhou, and N. A. Moubayed. The power of next-frame prediction for learning physical laws. *CoRR*, abs/2405.17450, 2024. doi: 10.48550/ARXIV.2405.17450. URL https://doi.org/10.48550/arXiv.2405.17450.

H. Wu, K. Khetarpal, and D. Precup. Self-supervised attention-aware reinforcement learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 10311–10319. AAAI Press, 2021. doi: 10.1609/AAAI.V35I12.17235. URL https://doi.org/10.1609/aaai.v35i12.17235.

T. Xiao, I. Radosavovic, T. Darrell, and J. Malik. Masked visual pre-training for motor control. *CoRR*, abs/2203.06173, 2022. doi: 10.48550/arXiv.2203.06173. URL https://doi.org/10.48550/arXiv.2203.06173.

W. Yu, N. Gileadi, C. Fu, S. Kirmani, K. Lee, M. G. Arenas, H. L. Chiang, T. Erez, L. Hasenclever, J. Humplik, B. Ichter, T. Xiao, P. Xu, A. Zeng, T. Zhang, N. Heess, D. Sadigh, J. Tan,

Y. Tassa, and F. Xia. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 229:374–404, 2023. URL https://proceedings.mlr.press/v229/yu23a.html.

Z. Yuan, Z. Xue, B. Yuan, X. Wang, Y. Wu, et al. Pre-trained image encoder for generalizable visual reinforcement learning. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/548a482d4496ce109cddfbeae5defa7d-Abstract-Conference.html.

R. Zhang, C. Walshe, Z. Liu, L. Guan, K. S. Muller, J. A. Whritner, L. Zhang, M. M. Hayhoe, and D. H. Ballard. Atari-head: Atari human eye-tracking and demonstration dataset. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 6811–6820. AAAI Press, 2020. doi: 10.1609/AAAI.V34I04.6161. URL https://doi.org/10.1609/aaai.v34i04.6161.

Z. Zhang, S. Zohren, and S. J. Roberts. Deep reinforcement learning for trading. *CoRR*, abs/1911.10107, 2019. URL http://arxiv.org/abs/1911.10107.

X. Zhao, C. Gu, H. Zhang, X. Liu, X. Yang, and J. Tang. Deep reinforcement learning for online advertising in recommender systems. *CoRR*, abs/1909.03602, 2019. URL http://arxiv.org/abs/1909.03602.

X. Zhao, W. Ding, Y. An, Y. Du, T. Yu, M. Li, M. Tang, and J. Wang. Fast segment anything. *CoRR*, abs/2306.12156, 2023. doi: 10.48550/ARXIV.2306.12156. URL https://doi.org/10.48550/arXiv.2306.12156.

B. Zitkovich, T. Yu, S. Xu, P. Xu, T. Xiao, F. Xia, J. Wu, P. Wohlhart, S. Welker, A. Wahid, Q. Vuong, V. Vanhoucke, H. T. Tran, R. Soricut, A. Singh, J. Singh, P. Sermanet, P. R. Sanketi, G. Salazar, M. S. Ryoo, K. Reymann, K. Rao, K. Pertsch, I. Mordatch, H. Michalewski, Y. Lu, S. Levine, L. Lee, T. E. Lee, I. Leal, Y. Kuang, D. Kalashnikov, R. Julian, N. J. Joshi, A. Irpan, B. Ichter, J. Hsu, A. Herzog, K. Hausman, K. Gopalakrishnan, C. Fu, P. Florence, C. Finn, K. A. Dubey, D. Driess, T. Ding, K. M. Choromanski, X. Chen, Y. Chebo-

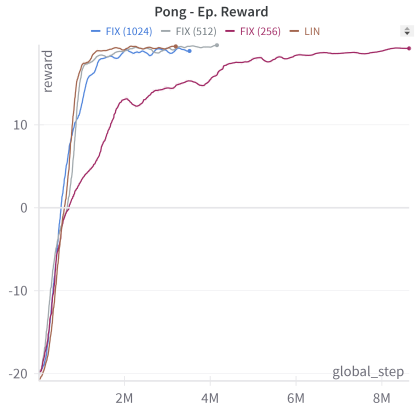
tar, J. Carbajal, N. Brown, A. Brohan, M. G. Arenas, and K. Han. RT-2: vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 229:2165–2183, 2023. URL https://proceedings.mlr.press/v229/zitkovich23a.html.

Appendix

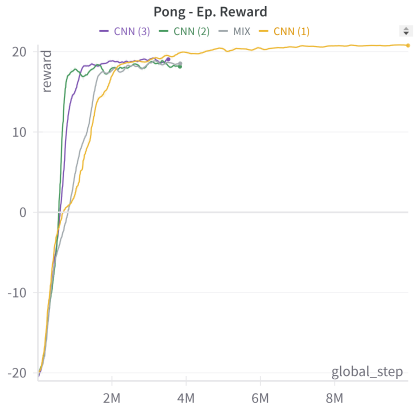
A.1 Combination Modules Analysis

In this section, we provide additional insights about the results of the initial analysis. The experimental setup, outlined in Sections 5.1-5.2, structures a robust framework for evaluating the various combination modules.

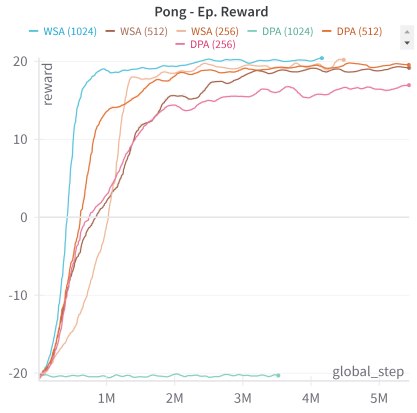
As shown in Table 5.6, we explore a variety of configurations for each module, ensuring a comprehensive assessment of their performance. Figures 1, 2, and 3 illustrate the training performance of each agent, offering a visual representation of their learning trajectories. Early stopping was applied for agents who showed no improvement over multiple evaluations.



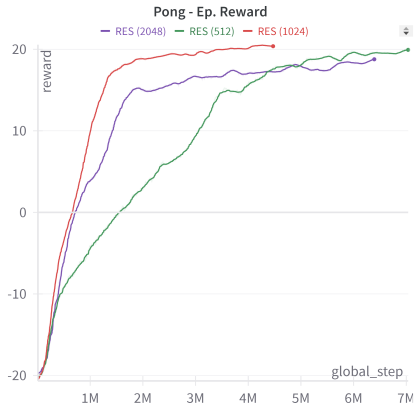
(a) `Linear` and `Fixed Linear`



(b) `Convolutional` and `Mixed`

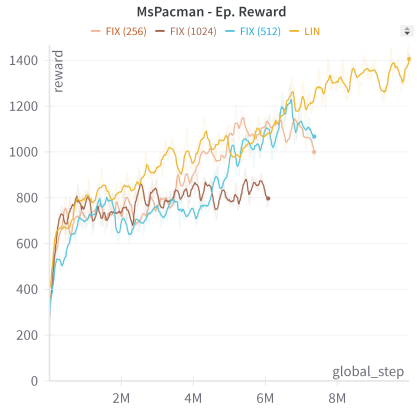


(c) `Weight Sharing Attention` and `Dot Product Attention`

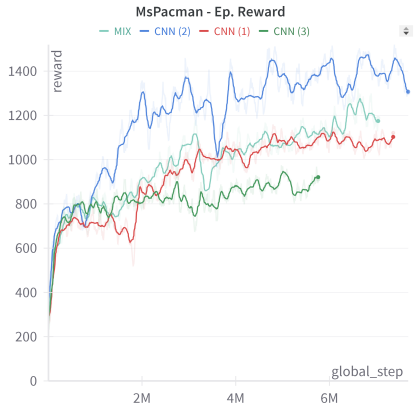


(d) `Reservoir`

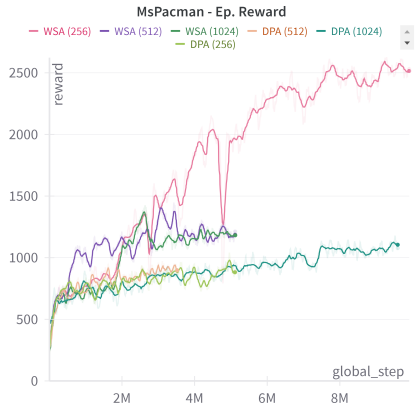
Figure 1: Initial analysis between different combination modules configurations on `Pong`.



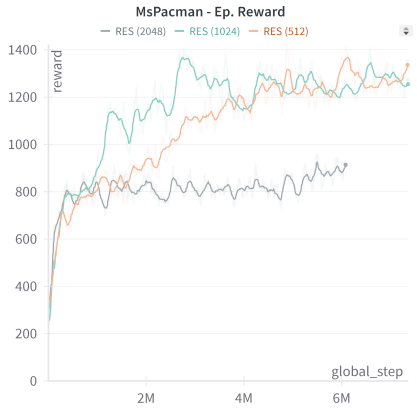
(a) `Linear` and `Fixed Linear`



(b) `Convolutional` and `Mixed`

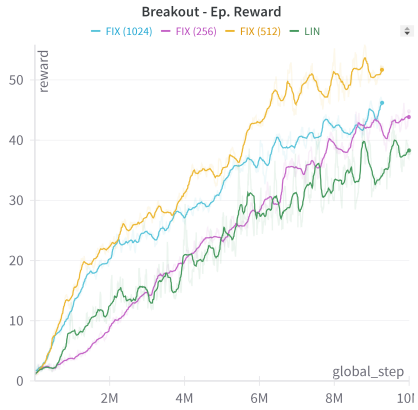


(c) `Weight Sharing Attention` and `Dot Product Attention`

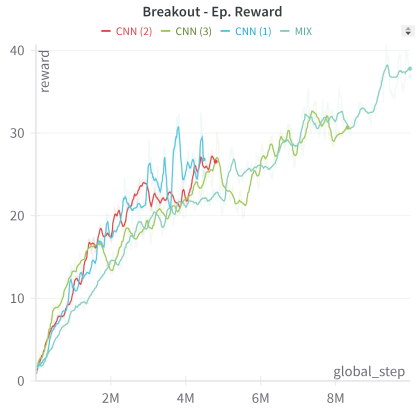


(d) `Reservoir`

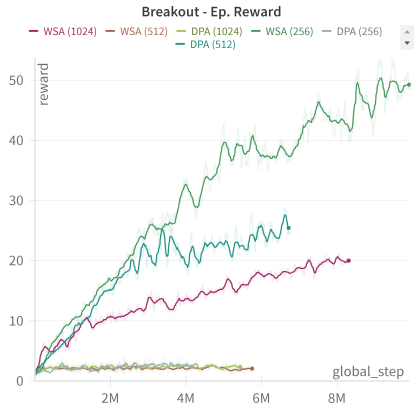
Figure 2: Initial analysis between different combination modules configurations on `Ms. Pacman`.



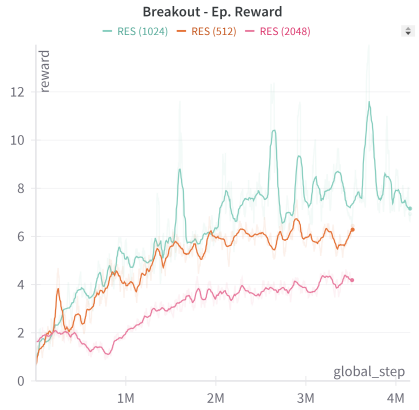
(a) `Linear` and `Fixed Linear`



(b) `Convolutional` and `Mixed`



(c) `Weight Sharing Attention` and `Dot Product Attention`



(d) `Reservoir`

Figure 3: Initial analysis between different combination modules configurations on `Breakout`.