Università di Pisa
Department of Computer Science
Master's Degree in Computer Science

Introducing Unsupervised Skills in Continual Reinforcement Learning Agents

Supervisors:
Prof. Davide Bacciu
Prof. Vincenzo Lomonaco

Author:
Elia Piccoli

Academic Year 2021/2022

Contents

Abstract

Reinforcement Learning in recent years has reached astonishing results exploiting huge and complex deep architectures. However, this has come at the cost of unsustainable computational efforts. A common characteristic of all state of art approaches, common in the majority of Machine Learning algorithms, is that the agent's network learns to solve the task "from scratch", that is from a randomized initialization, without reusing previously learned skills or doing it only to a very limited extent. In order to challenge the problem of transfer and reuse, we propose a new approach called Skilled Deep Q-Learning, which leverages pre-trained unsupervised skills as agents' prior knowledge. In the first part of the work, we discuss the implementation of this approach comparing its performance using the Atari suite and investigate how the agent uses these skills. In the second part, we focus on Continual Reinforcement Learning scenarios, trying to extend the proposed approach in a setting where the Reinforcement Learning agent learns more than one game simultaneously. Finally, we present various research paths that can be explored to further develop, understand and improve the proposed approach.

Chapter 1

Introduction

The recent Deep Learning revolution has shown the possibility and capability of Artificial Intelligence (AI) data-driven algorithms to achieve extraordinary performances on different tasks. Reinforcement Learning (RL) [54], which is one of main the Machine Learning [10] paradigms, formulates the learning process as a sequence of interactions with the environment. The agent, which interacts with the world, learns to solve a particular task based on these experiences, by creating a policy - a mapping from states to actions which determines the agent behaviour - to maximize a scalar reward. This approach is similar to the way in which humans learn throughout their life, but at the same time humans are able to quickly learn new tasks and adapt to new scenarios while leveraging previous knowledge and skills. Continual Learning (CL) [59] develops data-driven algorithms that seek to create models that are able to incrementally learn from a sequence of tasks. Continual Reinforcement Learning (CRL) combines the non-stationarity assumption of CL to RL settings, creating agents that learns multiple tasks in a sequential fashion. Another approach related to CL is Meta Learning, "learning to learn", it refers to machine learning algorithms that learn from the output of other machine learning algorithms, in particular, the model is trained over a variety of tasks from a stationary distribution in order to ease the optimization of parameters when faced a new unseen task from the same distribution.

In recent years, Reinforcement Learning has seen a huge growth with the release of benchmarks such as the Arcade Learning Environment (`Atari` suite) [19], but also astonishing results obtained in various games achieving super-human performances and even beating professional players, for instance, [36] presents the agent, whose model is reported in Figure 1.1, which was able to defeat Go world champion Lee Sedol. However, this has come at the cost of unfeasible computational efforts, which makes impractical a large-scale adoption of Deep RL based solutions. This is primarily the results of the high complexity of the problems solved by RL agents paired by the fact that such agents typically learn to solve the problem "from scratch" without reusing previously learned skills. This is fundamentally against any principle of compositionality which subsumes scalable computer science solutions.

Approaches such as Continual Learning or Meta Learning have addressed this problem trying to propose possible solutions in order to re-use knowledge. In this work we propose a new approach inspired by how our brain works: we - as humans - have much prior knowledge

and abilities that we implicitly use and re-use while approaching a new problem, basically through a compositional approach. From this ideas we first try to give a formal definition to "prior knowledge", introducing a set of abilities, called *Skills*, that are provided to the agent, moreover, we define a new algorithm where the agent learns to exploit this know-how while mastering a new task.

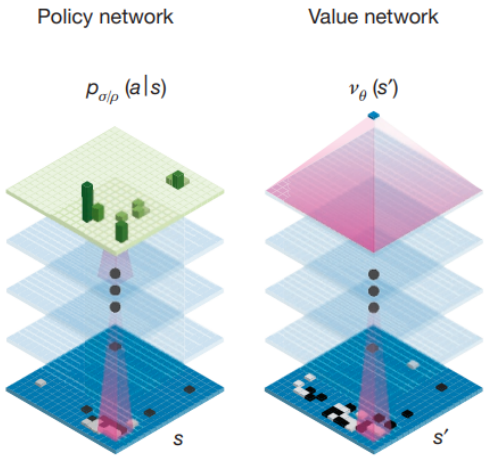


Figure 1.1: Neural network architectures used in AlphaGo [36] who defeated world champion Lee Sedol. The Policy network given a representation of the board predicts a probability distribution $p_{\sigma/\rho}(a|s)$ over legal moves, while Value network $v_\theta(s')$ computes a scalar value which represents the expected outcome.

As far as concern the structure of the thesis, we first provide a theoretical background presenting the topics related to this work - Chapter 2 -, which are Reinforcement Learning, Continual Learning and Continual Reinforcement Learning, at the same time analyzing the current state-of-the-art and libraries involved in the project's implementation. In Chapter 3, we introduce our new approach, Skilled DQN (SDQN), giving a formal definition to skills, analyzing how they are leveraged by the agent while learning a new task. Towards the closing of the chapter, we compare SDQN performances with a classical RL algorithm Deep Q-Learning, which is the first deep learning model trained with a variant of Q-learning [4] able to learn control policies [23]. Thereafter, Chapter 4, we extend the proposed algorithm to CRL scenarios, in particular, Progressive Neural Networks [35] training an agent to play more than one game simultaneously while transferring knowledge and exploiting skills. In the end, Chapter 6, we portray possible research paths that can arise from this work.

All the source code of the project is publicly available on GitHub at:
`https://github.com/EliaPiccoli/Master-Thesis`.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement Learning (RL) [54] is, together with Supervised and Unsupervised Learning, one of the major paradigms in Machine Learning (ML) [10]. This approach focuses on creating intelligent agents that, via interaction with the *environment*, learn how to choose a sequence of *actions* such that the agents can achieve their goal, which is to *maximize the rewards* obtained during the interaction with the world.
In recent years, with the surge of Deep Learning (DL) [29], RL jointly with DL - Deep Reinforcement Learning (DRL) [23] - has produced substantial advancements in various research fields and applications such as games, beating all Atari games at super-human performance with a single algorithm [64] or beating pro players in games like GO, StarCraft II and DOTA [36][63][57], but also in robotics manipulating a Rubik's cube [55] and lastly in chemistry with major discoveries in protein folding prediction [75].
When comparing RL with other ML approaches there are some differences that can be highlighted. For instance, while Supervised and Unsupervised Learning use a stationary dataset coming from a data distribution $P(X)$, RL agents instead influence directly the data - *observations* - with its actions, adding non-stationarity to the learning process. Also, differently from other methods, RL does not require strict supervision, it exploits the *reward* signal it receives from the environment to learn how to interact with it. Given its definition, this approach is well suited for problems requiring *Sequential Decision Making*, where a *decision maker* - the agent - chooses a *sequence* of actions. The previously mentioned problems are just a small part of this broad category, that also includes problems such as trading [74] and map-less navigation [70].

2.2 Reinforcement Learning Definition

Section 2.1 covered a brief and general introduction to RL, however it already portrayed its key elements, which are: *agent*, *actions*, *environment*, *reward* and *observations*.

Sutton and Barto in [54] provided a formal definition of RL using *Markov Decision Processes* (MDP), which consists of a tuple of five elements $\langle S, A, R, P, \gamma \rangle$:

- `S` set of *states*, each possible state represents a particular configuration of the environment, often the configuration will be related to a specific time using s_t.
- `A` set of *actions*, also here a_t will denote an action at time t. Not all actions may be available at a certain state s, to represent the set of valid action $a(s)$ will be used.
- `R` is the *reward function*, $r(s, a, s')$ represents the reward obtained by taking action a in state s reaching the new state s', r_t symbolizes the reward at time t.
- `P` is the *transition function*, it defines all the dynamics of the environment, how the agent moves from one to state to another. $p(s'|s, a)$ is the probability to transition from state s to s' under action a. It is worth mentioning that the next state s' is conditionally independent with respect to all the past states and actions, satisfying the *Markov property*.
- γ is the *discount factor*, $\gamma \in [0, 1]$, it determines the present weight of future rewards.

Previously the objective of RL was described as *maximize the reward*, giving it a more formal definition the RL agent's goal is to maximize the cumulative reward, formally defined as *expected return*, where the return G_t is defined as $\sum_{i=t+1}^{\infty} r_i$.
The γ parameter, previously introduced, is exploited to define the *expected discounted reward*, through *discounting* the agent's goal is to choose the actions that will lead to maximize the sum of discounted reward it will receive in the future, formally:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

The core element that describes how the RL agent behaves is its *policy*, which is a mapping between the possible states of the environment to actions: it can be deterministic $\pi(s)$ or stochastic $\pi(a|s)$. Figure 2.1 depicts all the elements just mentioned and the typical agent-environment interaction: the agent chooses action $a \in A$, leading to a reward r and new state of the environment s', following the reward function R and transition function P. The reward r and new state s', or a new observation o based on s', are provided to the agent to pursuit learning the policy π that maximizes the expected discounted reward G.
Another key concept in RL is that of *value functions*, while the reward expresses what is good in the current state of the environment, these functions specify the *long-term* desirability of the current state given all the states that will likely follow and their rewards.
The *state-value function*, $v_\pi(s)$, estimates how good is to be in state s acting under policy π, and it is defined as the expected return given the state s following policy π:

$$v_\pi(s) = \mathop{\mathbb{E}}_{\pi}[G_t | s_t = s], \text{where } \mathbb{E} \text{ represent Expectation.}$$

On the other end, the *action-value function*, $q_\pi(s, a)$, assigns a value to choosing action a in state s and act thereafter with policy π:

$$q_\pi(s, a) = \mathop{\mathbb{E}}_{\pi}[G_t | s_t = s, a_t = a]$$

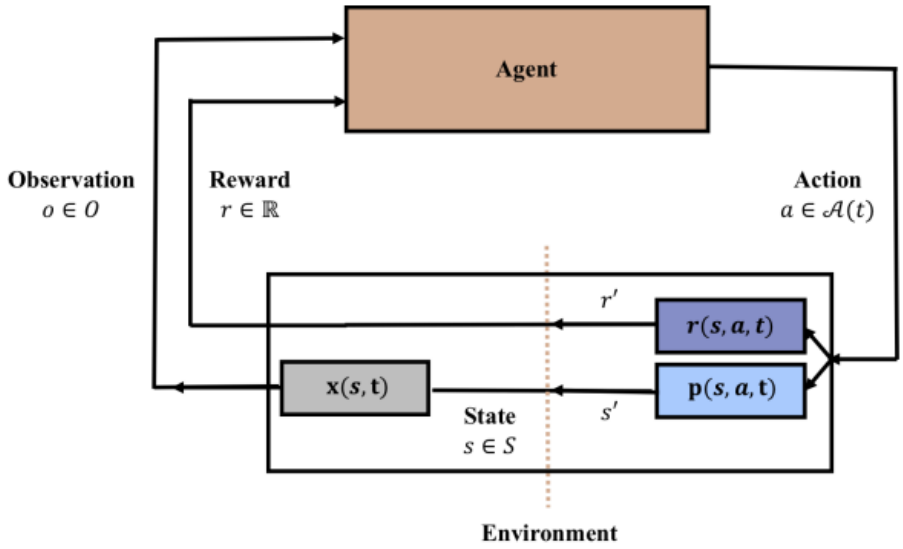


Figure 2.1: Illustration of agent-environment interaction in RL [67].

2.3 Reinforcement Learning Approaches

This section will briefly introduce and analyze the main families of RL algorithms, and then will provide a more in-depth analysis of Q-Learning, one of the most famous algorithms, which is a core part for the implementation of this work.

Reinforcement Learning algorithms generally can be divided into two main categories: *model-free* and *model-based*. The former focuses on creating a policy π that the agent can exploit to interact with the environment step by step without building any knowledge about the "world". The latter, as the name suggests, aims at creating a *model* of the environment, which means it has access to the *environment dynamics* that are exploited to learn the transition function $p(s', r|s, a)$ in order to use such model to *plan ahead*. A further classification among model-free approaches is between *policy-based* and *value-based*: on one hand, policy-based RL explicitly creates a parameterized function representing the policy π which at each step evaluated and optimized, on the other hand, value-based RL aims to learn the action-value function in order to generate the optimal policy, hence the policy is implicit and can be derived directly from the value function by taking the action with highest value. Finally, a major distinction classify RL algorithms in two groups: *on-policy* methods, attempt to evaluate and improve the policy while using it for control, on the other hand, in *off-policy* methods these two ideas are separated, in particular, it distinguishes between the *behaviour*

policy, the one used to generate the behaviour, and the *target policy*, which is evaluated and improved. Figure 2.2 exhibits a taxonomy of the most common RL algorithms divided with respect to the classification just analyzed.

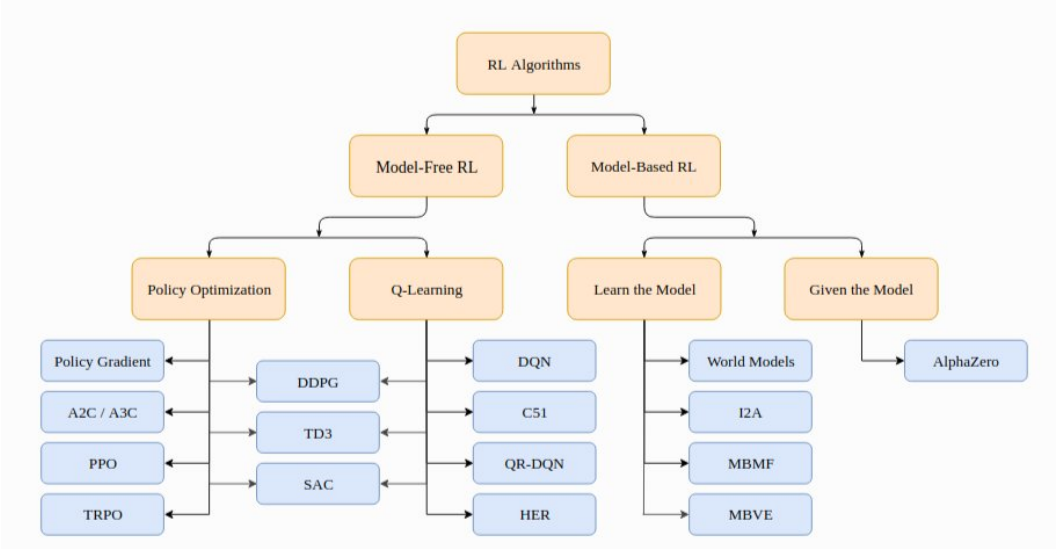


Figure 2.2: Taxonomy of algorithms in modern Reinforcement Learning [46].

2.3.1 Policy Optimization

Policy Gradients methods [13] learn a parameterized policy π_θ as a function of states, formally:

$$\pi_\theta(a|s) = \pi(a|s,\theta) = P(a_t = a|s_t = s, \theta)$$

where θ are the parameters of a differentiable function representing the policy π, e.g. neural network.
The agents' goal is to learn a policy π_θ which maximizes the cumulative discounted reward, return, from the starting state s_0, in particular, the function which is optimized to learn π_θ is denoted as $J(\theta)$:

$$J(\theta) = \mathbb{E}\left[G_t|s_t = s_0\right]$$

These algorithms adjust the parameters θ following the gradient ∇J exploiting *gradient ascent* - $J(\theta)$ must be maximized - according to the following update rule:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$$

The main problem in policy gradient methods is to obtain a good estimator of $\nabla J(\theta)$, as a matter of facts, *Policy Gradient Theorem* [13] lays the theoretical foundation for these approaches showing that:

$$\nabla J(\theta) \propto \mathop{\mathbb{E}}_{\pi}\left[G_t \frac{\nabla \pi(a|s,\theta)}{\pi(a|s,\theta)}\right] \tag{2.1}$$

The REINFORCE algorithm [5] is the forefather for this family of approaches and it directly applies Eq. 2.1 in the update step.

$$\Delta\theta \propto G_t \frac{\nabla\pi(a|s,\theta)}{\pi(a|s,\theta)} \tag{2.2}$$

However, this approach has some limitations, in particular, it is extremely data-hungry and relies upon the return G_t, hence requiring complete rollouts in order to perform each update step.
Actor Critic (AC) methods [13][17] improve the performances of REINFORCE algorithm by exploiting a learned approximation of the state-value function, hence replacing the return G_t with the immediate reward and an estimation of the expected return thereafter, exploiting bootstrapping [16]. This leads to a new formulation for the update step [13]:

$$\theta_{t+1} = \theta_t + \alpha(r_{t+1} + \gamma v_w(s_{t+1}) - v_w(s_t)) \frac{\nabla\pi(a|s,\theta)}{\pi(a|s,\theta)} \tag{2.3}$$

where v is learnable function approximating the state-value function with parameters w, while the term between parentheses represents the TD error, that is the difference between the agent's current estimate and target value. In this scenario the state-value function is referred as *critic*, while the policy as *actor*. Policy gradients methods fall within the family of on-policy algorithms as experience is collected interacting with the environment using policy π_θ. Differently from other approaches, i.e. Q-Learning, AC methods can lead to both deterministic and stochastic policies [27].

2.3.2 Q-Learning

Most algorithms for learning in MDPs rely on learning a value function. however, if the state space of the process is fairly large, the exact representation of a value function becomes unfeasible, requiring a huge amount of data and computation, also known as *curse of dimensionality* [40]. In order to scale-up these algorithms to real-world problems, approximate methods are exploited in order to approximate the value function, in such a way that its representation becomes manageable. The value function can be approximated by a parametric representation with parameters w, that can be learned so approximate values are "close enough" to true values:

$$\hat{v}_\pi^w(s) \approx v_\pi(s)$$

In order to effectively learn and update the parameters w the error between the approximation and the real function is computed as:

$$J(w) = \mathbb{E}[v_\pi(s) - \hat{v}_\pi^w(s)]^2$$

and via stochastic gradient descent the update rule for w can be defined as follows:

$$\Delta w = \alpha(v_\pi(s) - \hat{v}_\pi^w(s)) \nabla v_\pi^w(s)$$

Unfortunately these algorithms do not have access to the real value function $v_p i$, so, in order to learn it there are different approaches that can be exploited: Monte-Carlo and TD Learning. The former uses complete episodes to compute the return G_t of a given state which is

then used as target value. The latter, instead, exploits the current reward and the discounted current value estimate of subsequent states, which combined with the value estimate of the current state lead to the so called TD-Error [54].

$$\delta_t = r_{t+1} + \gamma v_t(s_{t+1}) - v_t(s_t)$$

One of the most important breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning.
Q-Learning [4] learns a policy π while interacting with the environment, iteratively approximates the *optimal* action-value function $q_*(s, a)$ in order to create a deterministic and greedy policy. The agent for each state will play the optimal - the one with highest value - move such that it maximizes the expected return:

$$\pi(s) = \arg\max_x q_*(s, a) \tag{2.4}$$

One fundamental property of q_* is that it must satisfy the *Bellman optimality equation* [54], which states that the value of the optimal q function in state s, choosing action a and following the optimal policy thereafter is equal to the expected return while taking the best possible action in state s, formally:

$$q_*(s, a) = \mathbb{E}[r_{t+1} + \gamma \max_{a'} q_*(s_{t+1}, a') | s_t = s, a_t = a] \tag{2.5}$$

The Bellman equation is also used to define the update of the q function, the idea is to push $q(s, a)$ as close as possible to the right-hand term of 2.5 such that it will eventually converge to the optimal q_*. The update rule can be defined as follows:

$$q'(s, a) = (1 - \alpha)q(s, a) + \alpha[r_{t+1} + \gamma \max_{a'} q(s', a')] \tag{2.6}$$

The actual convergence towards the optimal action-value function is guaranteed by the *Policy Improvement Theorem*, whose proof is reported in [54], and states:

$$\forall s \in S, q_\pi(s, \pi'(s)) \geq v_\pi(s) \implies v_{\pi'}(s) \geq v_\pi(s)$$

Another important and critical aspect of Q-Learning is the *exploration-exploitation dilemma*. It refers to the fundamental trade-off that must be implemented during training, which heavily impacts the data generation process. The choice is between *exploring* the environment - i.e. using a random policy - in order to gather unseen data (states), or *exploiting* the current policy, hence maximizing the expected return given the current knowledge, therefore visiting already known states. Q-Learning is an *off-policy* method since it aims at learning the optimal policy π_*, formulated as 2.4, which means it differentiate between the *target policy*, the one which is learnt, and the *behaviour policy* which is the one used to explore the environment.

In 2013, DeepMind introduced the so called *Deep Q-Learning* (DQN) algorithm, which uses a neural network as function approximator of q. Using this approach, as shown in [23], it is possible to reach super-human performances on many games of the Atari testing suite by means of a general algorithm and with little to no hyperparameters tuning. A key addition to DQN

is *replay memory*, it stores state transition tuples (s, a, s', r) allowing to create mini-batches composed by randomly sampled data removing correlations between consecutive samples.
The DQN version described up to now, is the naive version that was introduced in DeepMind's work, in recent years many significant publications have lead to huge improvements to the algorithm: the addition of a *target network* $\hat{q}$ [30] to improve learning stability, combined also with *soft update* [76] in order to smoothly update it. Another great upgrade of DQN is *Double DQN* [37], introducing a second neural network it is possible to decouple *selection* and *evaluation*, the first model is used to choose the actions meaning it is still estimating the values of the greedy policy, but the actual evaluation of such actions is done by the second network in order to reduce the overoptimistic value estimates [6].
Section 3.3 will analyze more in depth the implementation of the DQN algorithm with additional features and also report the pseudocode, since it is a core element of this work.

2.3.3 Learn the Model

This section and the following one challenge the model-based RL world, which is related to approaches that exploit a model of the environment - known or learned - and learn to approximate a global value or policy function [71].

World Models [49] introduces a new approach based on generative neural networks, that can be trained in an unsupervised fashion to learn a compressed spatial-temporal representation of the environment. Thereafter, the agents receives as input the extracted features and learns a policy to solve a particular task. The structure of the model can be divided into three different components, as shown in Figure 2.3: a visual component that creates a low-dimensional representations - codes -, a memory component that predicts future states based on historical information and a decision-making component to select actions. Analyzing each component more in the details:

- *Vision Model (V)*: the objective of this model is to learn a compressed and abstract representation of the observed input data, a Variational Autoencoder [21] is trained to encode the input frames into a latent representation z.
- *Memory RNN (M)*: this component is exploited to predict the future z vectors that vision models is expected to produce, in this case a RNN [2] is trained to output a probability distribution of the next latent vector based on past information, in particular, it is defined as $P(z_{t+1}|a_t, z_t, h_t)$, where a_t represents the action taken at time t, while h_t the hidden state of the model. Furthermore, the distribution is approximated using a mixture of Gaussian distribution exploiting the approach of Mixture Density Network [7] applied to RNN (MDN-RNN) [20].
- *Controller (C)*: is the responsible for choosing which actions to take in order to maximize the expected cumulative reward during an episode. In this case the complexity of the model is limited to the other components so a simple linear model is exploited in order to predict actions.

The model is trained by first sampling a fixed amount of random rollouts of the environment, then, the vision model is trained to encode the input frames into a meaningful low

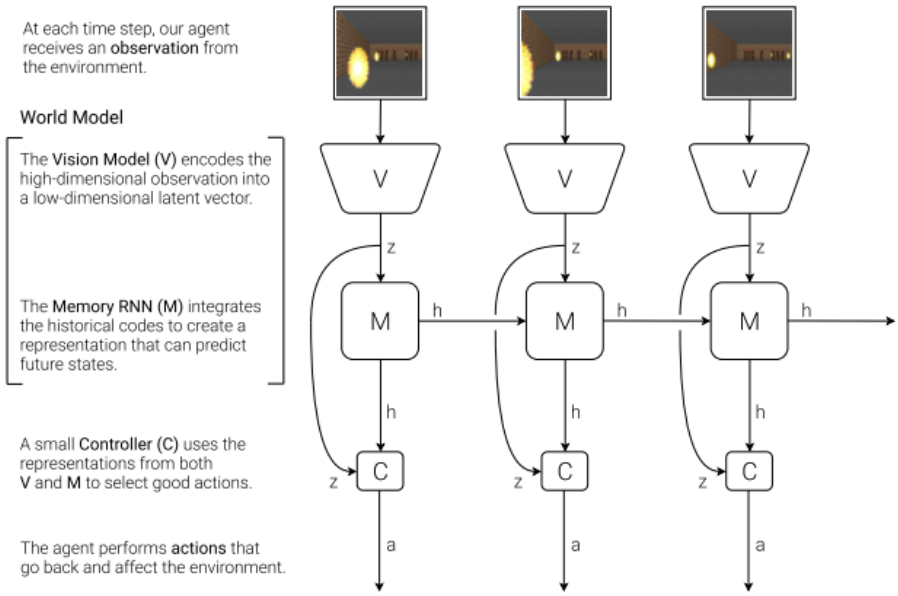


Figure 2.3: Representation of the three components: Vision (V), Memory (M) and Control (C) [49].

dimensional latent representation z by minimizing the difference between the original image and the decoder reconstruction given the latent vector. Thereafter, the trained (V) is used to create the latent vectors z_t that together with a_t from rollouts are exploited to model $P(z_{t+1}|a_t, z_t, h_t)$ as a mixture of Gaussians, finally, the controller is trained via CMA-ES [34] to maximize the expected cumulative reward.

2.3.4 Given the Model

In this scenario, which still is part of model-based RL, the agent has access to a model of the environment which is leveraged to ahead planning in order to build a policy able to assess the value of each possible state.

AlphaGo Zero [44] has been able to achieve super human performances in the game of Go exploiting convolutional neural networks and learning exclusively via RL from games of self-play. AlphaZero [43] is a more generic version of AlphaGo Zero, it removes handcrafted features or rules by just using deep neural network and reinforcement learning. In particular, the model takes as input the current board position and outputs (1) a probability distribution the actions $p_a = P(a|s)$ and (2) a value estimating the expected outcome z - win/loss - from state s, $v_t \approx \mathbb{E}[z|s]$. These outputs are learned by the agent via self-play and leveraged to guide its search, in particular, AlphaZero uses Monte-Carlo tree search (MCTS) algorithm where each search is composed by a series of simulated games via self-play. Each simulation chooses actions in different ways, i.e. low visit count, high probability or value, returning a vector π representing a probability distribution over moves. The model, as already mentioned, is trained by self-play RL, the agents play the game by selecting moves using MCTS,

$a_t \sim \pi_t$, at the end of the match the agent can access the game outcome z, which is then used to update the network parameters. In particular, the agent is trained to minimize the error between the predicted outcome v_t and the true result z, but also maximize the similarity between policy vector p_t and seach probabilities π_t, hence the error function is define as follows:

$$L_\theta = (z - v_t)^2 - \pi \log p_t + c||\theta||^2$$

where θ are the parameters of the model and c is the regularization hyperparameter.

AlphaZero has been tested and compared to state-of-the-art algorithms for complex and famous games, as reported in Figure 2.4 which shows the results for the three different games:

- Chess: AlphaZero is able to beat Stockfish, the strongest hand-crafted engines for chess, 155 times while losing only six games.
- Shogi: AlphaZero achieves over 90% winrate against Elmo, a computer shogi evaluation function that exploits alpha–beta search engine.
- GO: defeated AlphaGo Zero [44] winning more than 60% of games.

An interesting aspect which goes beyond numbers, is related to how the agent effectively plays the game. In chess, AlphaZero, while learning via self-play, is able to learn the basic strategies which also humans apply - i.e. pawn structure -, but at the same time was able to develop its own ideas and strategies.

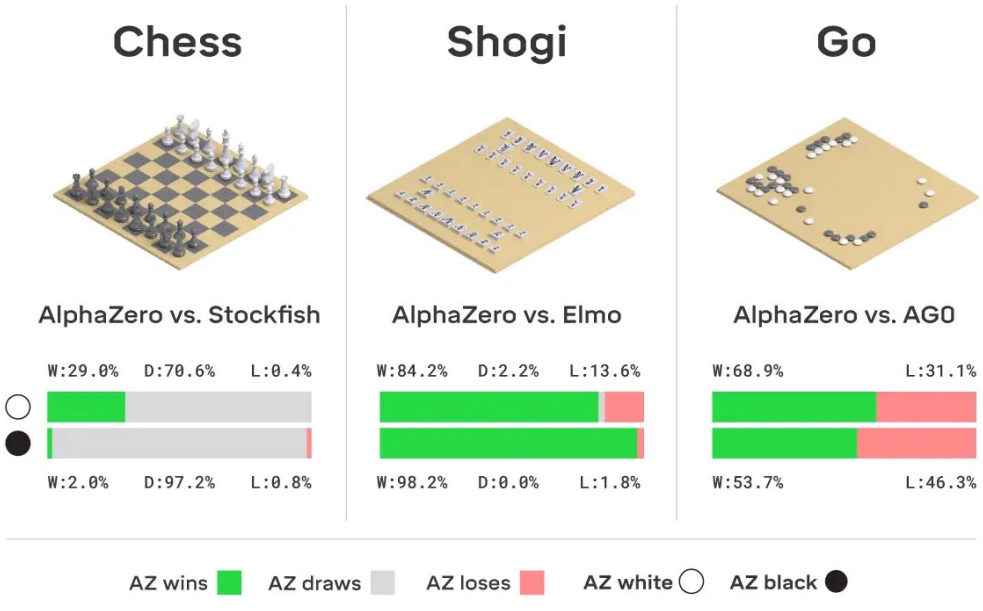


Figure 2.4: Results reported by AlphaZero against the best algorithm for each game [53].

2.4 Continual Learning

While Machine Learning approaches usually learn from data sampled from a stationary data distribution, Continual Learning (CL) deals with the problem of learning from a non-stationary stream of data. In particular, the first scenario - non-continual setting - allows to access all data immediately, without restricting the access to such data. On the other hand, the continual data stream is usually empirically split in various parts - *tasks* - which are temporally bounded and contains the data from which the model must learn in order to achieve a goal specified by the task. It is worth mentioning that tasks can be disjoint or related to each other, in terms of learning objectives, and depending on the setting.

Continual Learning brings Machine Learning toward a real-world setting. First and foremost the learning process becomes more similar to that of humans, continually acquiring, fine-tuning, and transferring knowledge and skills throughout different tasks, but also it faces constraints and limitations related with the real world. As each task is temporally limited and discarded, the ability to forget and maintain knowledge is a critical and fundamental aspect for continual learning. A key aspect is the sequential nature of the learning process, only a small portion of data from various tasks is available at once, the challenge is to learn without *catastrophic forgetting* [12][3]: the knowledge and performance on a previously learnt task should not decrease over time as new knowledge is added.

2.5 Continual Learning Definition

While introducing Continual Learning it is easy to see that one of the fundamental aspects of this paradigm is the interaction with a sequential stream of *tasks* from which the continual learning algorithm learns. So, first and foremost, it is important to give a formal definition to the concept of task. To do that *Definition 2* of [68] comes in handy, which states:

A task is a learning experience characterized by a unique task label t and its target function $g_t^*(x) \equiv h^*(x, t = \hat{t})$, i.e., the objective of its learning.

Considering a task space T, a task - identified by its task label t - is sampled form such space $t \sim T$ during the learning process. Once the task is sampled, usually is divided into smaller parts called *experiences*, which contain the data from which the model must learn from in order to solve the task. As previously mentioned, the interactions with a particular experience might be limited - i.e. real-life scenarios - bounding the data access, the ability to retain and forget knowledge which may or may not be useful in the future is a critical aspect and is one of the main topics that continual learning focuses on.
Changes in the data distribution over time are commonly referred to as *concept drift* [33]. Different types of concept drift can be distinguished: changes in the input distribution $P(x)$, *virtual concept drift*, or changes in the underlying task $P(y|x)$, *real concept drift*. Further, concept drifts can be gradual or abrupt, in the second scenario is referred as *concept shift*.

In CL is possible to differentiate various approaches [68], called *scenarios*, based on how the sequence of N tasks changes over time. Three different scenarios can be defined:

1. *Single-Incremental-Task (SIT)*: $t_1 = t_2 = ... = t_N$, the task t is always the same, but the data distribution changes over time. For example, creating a model which learns to classify the MNIST [18] digits, but it experiences all 0s digits, then all 1s and so on, and at the end it must be able to classify all digits. The task is always the same, but the concept drift might lead to forgetting.

2. *Multi-Task (MT)*: $\forall i, j \,.\, i \neq j \Rightarrow t_i \neq t_j$, in this case there are multiple tasks, and the algorithm experiences one task after the other exactly once. A possible example is to learn to classify dogs from cats, then dogs from cows, cats from tigers, without forgetting.

3. *Multi-Incremental-Task (MIT)*: $\exists i, j, k \,.\, t_i = t_j \wedge t_i \neq t_k$, in this scenario the same task is present several times in the sequence of tasks but it is not the only one.

The above scenarios introduce and define how the stream of tasks is sampled from T, in order to create the learning data for the continual algorithm. A peculiar aspect of a learning problem is related to the ability to adapt to new concepts to be learnt. In particular, each task is coupled with its *task label*, though ideally such information should be avoided. The way in which the task label is used during learning, to mark concept drifts, further introduces a new classification for CL scenarios, based on *task label assumptions*:

- *No task label*: changes in the distribution are not signaled.

- *Sparse task label*: changes in the distribution are sparsely signaled.
- *Task label oracle*: every change in the data distribution is signaled.

Figure 2.5: Illustration of different scenarios based on task label and concept drifts [68].

2.6 Continual Learning Approaches

Models that operate and interact with the real world are exposed to continuous streams of data and must learn and remember multiple tasks from dynamic data distribution. A key aspect for these models is the ability to continually learn over new data while retaining previously learned features. The main issues is related to *catastrophic forgetting* [12], this phenomenon leads to performance decrease over time of previously learned tasks while adding new knowledge. Current deep learning models excel at a number of tasks based on large set of training samples, however, these approaches assume that all data is available during training, therefore, to adapt to changes in the data distribution the model must be retrained from the beginning on the new data. Therefore, when trained on sequential tasks, the performances of these neural network models significantly decrease as new tasks are learned. To overcome catastrophic forgetting the model should be able to acquire new knowledge but at the same time refine existing knowledge in order to retain the key features to solve previous tasks.
To address this problem a wide variety of approaches have been proposed and studied over the years. In particular, they can be divided into four main non-exclusive categories, Figure 2.7, which are extensively analyzed in [59][68].

- *Regularization approaches*: the regularization approaches in CL consist in modifying the update of weights when learning in order to avoid overfitting of the new task, hence forgetting previous skills.
  One possible approach that belongs to this category is Learning without Forgetting (LwF) [41], where the network - in this case CNN - previously trained on a particular task is enforced to be similar the the network which is learning the new task by using

knowledge distillation. In particular, the model has shared parameters θ_s across all tasks and task-specific parameters θ_o, the goal is to add task-specific parameters θ_n for a new task and learn parameters that work well on old and new tasks. The model parameters are updated using the following formula:

$$\theta_s^*, \theta_o^*, \theta_n^* \leftarrow \text{argmin}_{\hat{\theta}_s, \hat{\theta}_o, \hat{\theta}_n} \left(\lambda_o \mathcal{L}_{old} \left(Y_o, \hat{Y}_o \right) + \mathcal{L}_{new} \left(Y_n, \hat{Y}_n \right) + \mathcal{R} \left(\hat{\theta}_s, \hat{\theta}_o, \hat{\theta}_n \right) \right)$$

where, $\mathcal{L}_{old}(Y_o, \hat{Y}_o)$ and $\mathcal{L}_{new}(Y_n, \hat{Y}_n)$ minimize the difference between predicted values Y and ground-truth values $\hat{Y}$ for the old and new task respectively, λ_o balances the importance of the old task, higher value favors old task performance over the new task's, lastly, $\mathcal{R}$ is a regularization term to prevent overfitting.
Another different approach is Elastic Weight Consolidation (EWC) [39], it computes a penalty term which is added to the loss function in order to prevent updates of task-critical weights. Given two tasks, a task A previously learned and a new task B, when learning the new task the objective is to find a configuration of the optimal parameters θ_B^* such that is close to the optimal configuration of the old task θ_A^*. While learning task B, EWC maintains the performance in task A by constraining the parameters to stay in a region of low error centered around θ_A^*, as shown is Figure 2.6. The relevance of parameters θ with respect to a set of $\mathcal{D}$ is modelled as the posterior distribution $p(\theta|\mathcal{D})$. Condidering $\mathcal{D}$ as the union of the data for task A and B, the log value of posterior is:

$$\log p(\theta|\mathcal{D}) = \log p(\mathcal{D}_B|\theta) + \log p(|\mathcal{D}_A) - \log p(\mathcal{D}_B)$$

where the posterior probability $\log p(|\mathcal{D}_A)$ encodes all the information of the previously learned task A. The true posterior probability is intractable, is approximated as a Gaussian distribution, with mean given by the parameters θ_A^* and a diagonal precision given by the diagonal of the Fisher information matrix F [24]. Given this approximation, EWC's loss function can be formulated as follows:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \sum_i \frac{\lambda}{2} F_i (\theta_i - \theta_{A,i}^*)^2$$

where $\mathcal{L}_B(\theta)$ is the loss function for task B, while λ is the parameter that weights the importance of old task with respect to new one. This formulation can be easily extended to a third task C by adding an additional penalty term.

- *Dynamic Architectures approaches*: these approaches can be either explicit - add or clone parameters - or implicit - freeze weights - dynamic modification of the model exploited during learning. As far as concern the first category of algorithms, one of the first approaches in this paradigm is Progressive Neural Network [35], which keeps all the knowledge of previously learned task by dynamically expanding the model adding a new sub-network for each new task. Given a model trained over N tasks, when a new task T_{N+1} is given, the model allocates a new neural network which exploits lateral connections to perform knowledge transfer form previously learned tasks. This architecture avoids catastrophic forgetting since the parameters for the already learned task are "frozen" and only parameters θ_{N+1} are effectively optimized. A more in-depth analysis of this approach is reported in Sec. 4.1.

On the other hand, implicit architecture modification adapt the model for CL without changing its architecture, in particular, the adaptation is achieved by inhibiting some learning units or modifying how the model computes its output. An approach that belongs to this category of algorithm is PackNet [51], which iteratively performing pruning and re-training is able to sequentially learn multiple tasks leveraging a single neural network with a minimal drop in performances. In particular, the idea is to exploit network pruning to free parameters that can be used to learn new tasks without adding information to the network. This approach first trains the model to learn a particular Task I, after it prunes a certain fraction of the weights, obviously this leads to a drop in performance which is overcome with a small training, in this way the model is still able to solve Task I but using a sparser representation. Once training for a new Task II the pruned weights are effectively used for learning, in such manner the model combines the weights for Task I with new weights for Task II. The same approach is applied, a pruning phase followed by a re-train phase which lead the model to be able to solve both Task I and II, and ready to learn more tasks. A representation of the training algorithm is reported in Figure 2.8.

- *Rehearsal approaches*: refer to methods that save samples as memory of the past to reduce catastrophic forgetting. These samples are used to maintain knowledge about the past in the model and are chosen in order to be representative of past tasks. An example of this kind of algorithms is iCaRL [42], which is used to train classifiers from a sequential stream of data in class-incremental form, which means that the model will learn to classify class 1 from data X^1, then data X^2 is presented to learn how to classify also class 2. The model in order to incrementally learn various classes relies on a set of *exemplar data* dynamically selected from the data stream. For each class observed the model store an exemplar set, which are then leveraged to train the model to classify multiple classes. Another class of algorithms called *Generative Replay*, or pseudo-rehearsal [8], instead of storing and re-using past samples train generative models on the data distribution, so that they can later sample past data from previous tasks while learning new tasks ensuring that past knowledge is not forgotten.

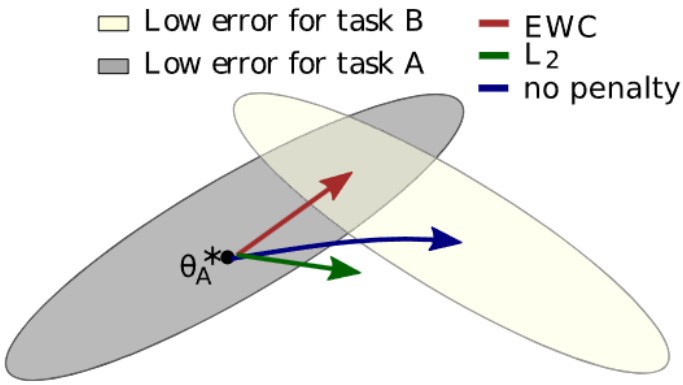


Figure 2.6: EWC optimizes parameters for new task B keeping as close as possible to the optimal value of parameters of old task A [39].

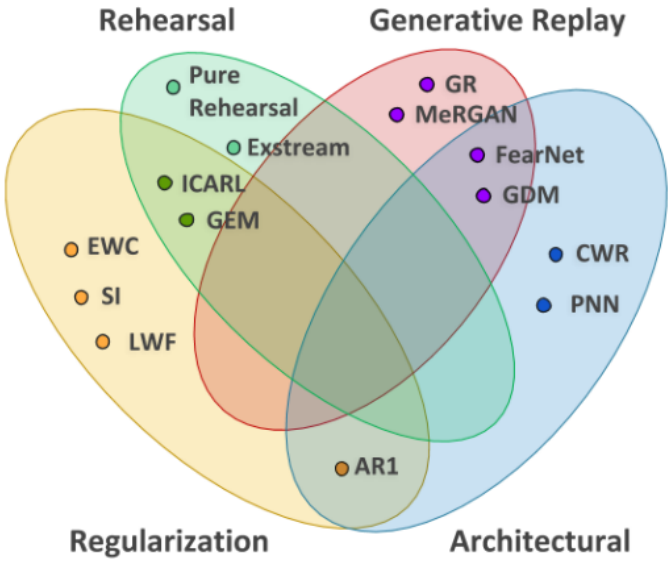


Figure 2.7: Diagram of some of the most popular CL approaches [68].

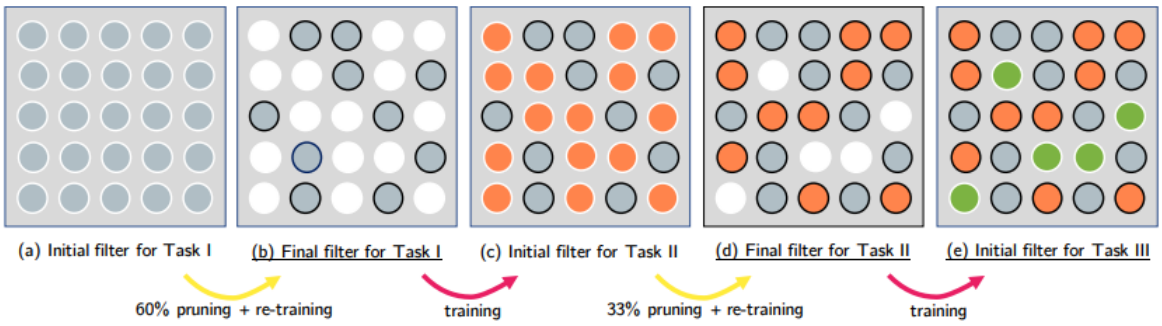


Figure 2.8: Training process of PackNet [51]: (a) train the model on Task I then applies pruning and re-train to obrain a sparser representation (b). Thereafter the same process is applied while learning Task II (c)-(d) and eventually Task III (e).

2.7 Continual Reinforcement Learning

Continual Reinforcement Learning (CRL) is an up-and-coming field of research and it is acquiring more and more interests as shown by the recent publications [67][69][39][62].
This approach stems from the similarity between RL and CL scenarios. As already presented in previous sections, RL algorithms deal with non-stationary data due to the direct influence of the policy in the data generation process but also with partial observability of the environment. Moreover, as the task or its properties change with time, also the MDP describing the problem can change over time, similarly to how tasks may change in a CL scenarios. Therefore, all the CL considerations from the previous section still hold, and it is safe to say

that RL could often be framed as a CL problem [68].

An important generalization of MDPs is given by *Semi-Markov Decision Processes* (SMDPs), a special formalization appropriate for continuous-time discrete-event systems, which allows the agent to execute actions at different time scales and taking variable amounts of steps. In order to define such formalization a new concept must be introduced: skills or *options* [14]. Similarly to the definition of MDPs in section 2.2, SMDPs are defined by a tuple $< S, \Sigma, R, P, \gamma >$, where Σ is the set of *skills*, with each skill defined by a tuple (I, π, β). $I \subseteq S$ is the initiation set, that is the set of states where the skill can be initiated, $\pi : S \times A \to [0, 1]$ which is the policy followed during the execution of the skill, and lastly $\beta : S \to [0, 1]$ the termination condition. A skill is available in state s if and only if $s \in I$, once the option is taken, then actions are selected according to π until it terminates according to β.

The Semi-MDP formalization, together with other works like [11], provide the foundation for *Hierarchical Reinforcement Learning* (HRL). HRL creates a hierarchy of sub-modules, each of them will solve various sub-tasks that will be exploited by the upper levels to learn the agent's policy. This approach reduces the computational complexity but also allows the agent to learn low level abilities. The latter property, in particular, provides the possibility to embrace a CL strategy exploiting these skills to build a ground knowledge that can be reused to learn new tasks, as a matter of fact, HRL challenges continual learning in works such as [45]. The Hierarchical Deep Reinforcement Learning Network controller learns to solve complicated tasks in Minecraft by learning reusable RL skills in the form of pre-trained Deep Skill Networks (DSNs), which can be implemented into two different way: a DSN array (Figure 2.10, module A) or a multi-skill distillation network (Figure 2.10, module B). The agent's policy π at each steps chooses whether to execute a primitive action or re-use pre-learned skills. In the first case the agent performs a single step, in the second one, instead, it follows the skill policy μ until it terminates. A skill policy is a mapping from states to a probability distribution over skills $\mu : S \to \Delta_\Sigma$, its action-value function $q : S \times \Sigma \to R$ represent the value of taking skill σ in state s and act under skill policy μ. The idea is to achieve knowledge transfer among skills but also re-use learned policies to perform new tasks, leveraging the policy π operating at different time scales.

Another notable work to CRL, that tackles the problem from a different perspective, is presented with the introduction of *Progressive Neural Networks* (PNN) [35]. PNN aim at solving complex sequences of tasks, while leveraging transfer and avoiding catastrophic forgetting. In particular, the model is immune to catastrophic forgetting by design, in fact PNN allocate a new neural network - *column* - for each task being solved. The model starts with a single column, a neural network with parameters Θ^1, and learns to solve a particular task. When approaching a second task, the parameters of the first model Θ^1 are frozen and a new column with parameters Θ^2 is introduced, in this case though each layer of the second neural network receives input from its previous layers but also from the previous column via lateral connections, in this way transfer is enabled via lateral connections to features of previously learned columns, while learning task t_i all columns from 1:i-1 are frozen. Given that PNN are a key element of this work, section 4.1 will analyze more in depth their definition and implementation.

Finally, a work which not only introduces a different algorithm for CRL problems but also defines a possible benchmark for these category of algorithms is "Continual Reinforcement Learning in 3D Non-stationary Environments" [69]. In the first part of the work a novel 3D maze environment is introduced, called *CRLMaze*, where the task is to learn to navigate in a complex environment while collecting "*column bricks*" and avoiding "*flaming lanterns*". In order to assess the capabilities of CRL algorithms the environment is non-stationary with various levels of difficulty, in particular, lights, texture and objects can vary over time. The CRL agent will challenge one of the three possible scenarios, *rows*, reported in Figure 2.9, and sequentially face the different tasks $M_1 \rightarrow M_2 \rightarrow M_3$ following the various changes. Later in the work, a new strategy is introduced *CRLUnsup*, where idea is to update memory only when a substantial difference between the expected reward and the actual one is detected. In this way distribution shifts are detected based on the agent's performance in the task. The proposed approach exploits the regularization method EWC [39], to avoid undesired updates to key parameters, Section 2.6, while using a simple moving average and threshold ν on rewards to detect task changes.

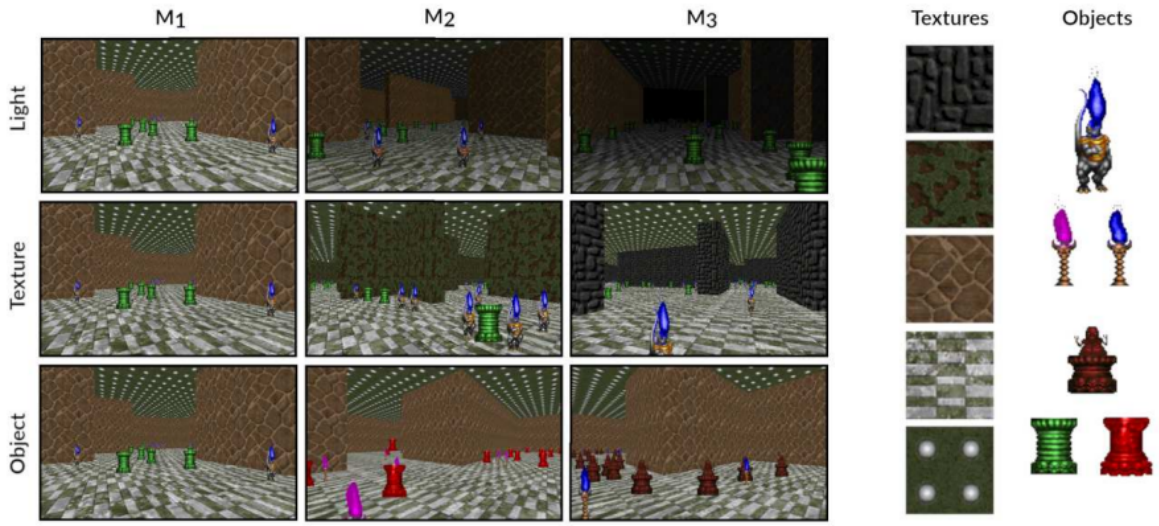


Figure 2.9: 3D CRLMaze [69]. On the left, the possible scenarios with the evolution of the environment, on the right, possible textures and objects used in the environment.

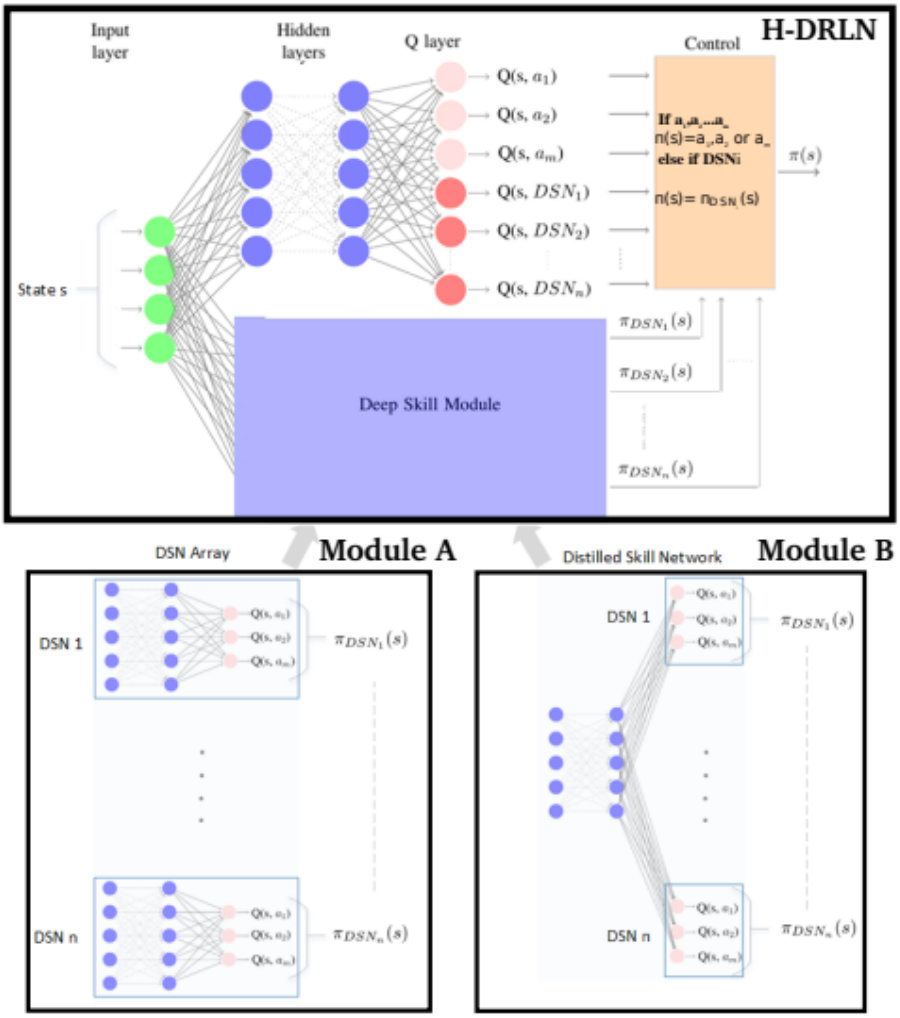


Figure 2.10: Implementation of Hierarchical DRL Network, the model can either choose a primitive action or access the *Deep Skill Module*, which has two possible implementations and contains all the skills trained on sub-tasks and can be exploited to act for a fixed number of steps [45].

2.8 Related Works

The previous sections have introduced the theoretical foundations of RL, which have always been a landmark, but also its recent advancements with the advent of Deep Learning, as a matter of fact, lead the DRL research field to constantly evolve trying to break the frontiers and improve the capability of the agents, for instance the evolution from *AlphaGo* to *MuZero* [36][43][73].

Deep Reinforcement Learning has been successful in learning sophisticated behaviours and solve more and more complex tasks, however this has come at the cost of unsustainable computational efforts, requiring a massive volume of training data - sample-inefficient - which

makes impractical a large-scale adoption of DRL based solutions. This is related to the high complexity of the problems solved by RL agents paired by the fact that such agents typically learn to solve the problem "from scratch", whereas human learners can attain reasonable performances on any of a wide range of tasks with comparatively little experience, benefiting from their prior knowledge about the world [47].

A step towards overcoming the inefficiency of RL methods is via *Meta-Learning*, "learning to learn", a system is trained to learn how generalize a stationary task distribution in order to be able to tackle new experiences drawn form the same task distribution. Meta-Learning usually involves two learning systems: one lower-level system that learns relatively quickly, and which is primarily responsible for adapting to each new task; and a slower higher-level system that works across tasks to tune and improve the lower-level system.
Two main works have lead to the introduction of this approach in the RL field, leading to Meta Reinforcement Learning (Meta-RL), both sharing similar ideas. In particular, the first work is RL^2 [32], as the name suggests RL is applied at two levels: a "slow" RL algorithm learning the *policy representation* using Gated Recurrent Units (GRU) [25] which receive as input the classical RL tuple $(s, a, r, dones)$ and it computes the distribution over actions, but at the same time it retains its state across episodes in a given MDP; while the second part is *policy optimization* where the actual policy is optimized. The second work, that tries to bring the ideas of supervised meta-learning in a RL scenario, is *Learning to RL* [38]. The architecture is similar to the previous work, in fact it is composed by a recurrent component, LSTM [9], trained on fixed-length episodes, each involving a task randomly sampled from a task distribution - the LSTM hidden state is initialized at the beginning of each episode. The policy learnt by the agent is history-dependent - given the recurrent architecture - and when tested with new MDPs is able to adapt and optimize rewards for that particular task.

Another important aspect, that has been mentioned at the beginning of the section, is related to the re-use of knowledge and skills. Reinforcement learning agents usually learn from scratch, without leveraging prior experience, as a matter of fact, agents need to collect a large amount of experience while learning the target task, which is computationally and time expensive. A possible approach to solve this task is by means of an offline dataset of pre-recorded agent experience in the form of state-action trajectories, and use this data to learn various skills that can be later exploited by the agent while learning to solve the task. A recent work which makes use of this idea is *Skill-Prior RL* (SPiRL) [72]. Given a large, unstructured and *task-agnostic* dataset with trajectories collected in different ways - i.e. trained agents, random policies - it learns various skills a_i, which are a sequence of actions with fixed horizon. In order to learn low-dimensional skills a latent skill embedding space Z is introduced, and to achieve a rich skill embedding space an encoder $q(z|a_i)$ - decoder $p(a_i|z)$ architecture is implemented using neural networks, at the same time the model learn a prior over skills $p_a(z|.)$. The conditioning of the prior can be adjust and should be informative about the set of skills that are meaningful for the current task, in this work the prior is conditioned by the current state $p_a(z|s)$. To learn the skill prior the Kullback-Leibler divergence [1] between the predicted prior and the inferred skill posterior, $D_{KL}[q(z|a_i)||p_a(z|s_t)]$, is included in the optimization process of the RL algorithm. At the end of the training process SPiRL learns two models: $\pi(a|z, s)$ that decodes latent representation into sequence of actions and

the prior over latent skill variables $p_a(z|s)$ which can be leveraged to explore in the skill space.

Although SPiRL is more efficient than learning from scratch, it still requires a significative number of interaction with the environment to learn a new task. To overcome this problem *Skill based Meta-RL* [77] tries to address this problem extending the work of [72] in a Meta-RL scenario. This approach can be divided into three different phases as reported in Figure 2.11: it leverages the skill extraction approach proposed in SPiRL, in particular it trains (1) a skill encoder $q(z|s_{0:k}, a_{0:k-1})$ which embeds a k-step trajectory into the latent space, and (2) a low-level policy $\pi(a|s, z)$ which is trained to reproduce the action sequence given the skill embedding. The second phase is *Skill-based Meta-Training* which aims at learning a policy that can quickly learn to leverage the extracted skills to solve new tasks. To learn such policy first a task-encoder $q(e, c)$ is trained to produce task embedding e, in particular, the task-encoder is formalized considering N independent factors [61]:

$$q_\phi\left(\mathbf{z} \mid \mathbf{c}_{1:N}\right) \propto \Pi_{n=1}^{N} \Psi_\phi\left(\mathbf{z} \mid \mathbf{c}_n\right) \text{ where } \Psi_\phi\left(\mathbf{z} \mid \mathbf{c}_n\right) = \mathcal{N}\left(f_\phi^\mu\left(\mathbf{c}_n\right), f_\phi^\sigma\left(\mathbf{c}_n\right)\right)$$

to keep the model tractable Gaussian factors Ψ_ϕ are used, in particular, f_ϕ is a neural network parameterized by ϕ and is trained to predicts the mean as well as the variance of the Gaussians as a function of c_n. The task-encoder is used together with learned skills to train a task-embedding-conditioned policy over the skills $\pi(z|s, e)$ where the meta-learning task is to learn how to combine these skills instead of learning from scratch and based on primitive actions. Lastly, *Target Task Learning*, when a target task is given the meta-trained policy is exploited first to explore different skill options to learn about the task, and then to narrow down to the optimal set of skills that solve the task.

The methods just analyzed, but also from previous sections, tried to improve RL algorithm introducing the idea of skills. In [45] skills are policies that are able to solve a particular sub-task and are used to solve a higher-level task, in [72][77], instead, skills are a particular sequence of actions which the agent learns to encode and use in its policy.
We - as humans - have much prior knowledge and abilities that we implicitly use and re-use while approaching a new problem. This previous know-how defines the set of skills that are present in our brain, for example when approaching a game we can easily detect our character, the score etc. and we implicitly use this knowledge while playing. This work tries to give a different perspective and definition to skills in the RL world, in fact instead of having an agent that learns from frames of the game to build a winning policy, it leverages various skills, which are task-agnostic pre-trained models on unsupervised tasks, that pre-process the frames enriching its information and giving the agent the ability to exploit these skills to solve the task. Section 3.1 will analyze in depth and formally this idea.

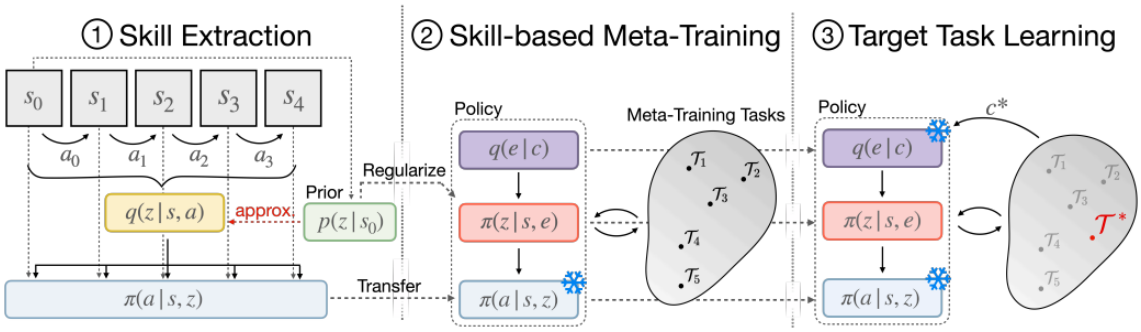


Figure 2.11: Skill-based meta-RL method proposed in [77]. *Skill Extraction* learns reusable skills via a skill extractor (yellow) and low-level policy (blue), it also trains a prior distribution over skill embeddings (green). *Skill-based Meta-Training* meta-learns high-level policy (red) and task encoder (violet) exploiting the low-level policy. *Target Task Learning* leverages the meta-trained hierarchical policy to quickly learn a new target task T^*.

2.9 Code & Libraries

This section provides an overview of the software libraries involved in the development of the project explaining their purpose and structure as well as the project implementation.

First things first, all the experiments of this work use environments provided by the `Atari 2600` [19] suit. Since, the programming language for the development of the project is `Python`, in order to easily access these environments and setup the experiments `OpenAI Gym` [31] library is used. With just few lines of code it is possible to create an instance of the game and let the agent play with it - the game and simulation are computed on the CPU - moreover, it allows to render the environment thus helping during the validation process to appreciate the learned agent's policy visually. Sections 3.5 and 4.2 will provide more detailed description about the actual environment used for the experiments, both training and validation.

As far as concerns the actual implementation of the project, all the source code is publicly available on GitHub at `https://github.com/EliaPiccoli/Master-Thesis`, the code is divided into different folders with respect to the various models used in this work, and also additional material can be found. In order to create a workspace without any conflict and inclined to various experiments or scenarios, all the models and their training procedure have been written from scratch using `Pytorch` [60]. Furthermore, in order to keep track of the various metrics during learning, collect data and save models the `wandb` [65] platform has been employed. Using a simple API and few lines of code it is possible to set-up a real-time interface where it is possible to monitor different metrics of the experiments and store information, this was crucial during RL experiments given the huge amount of time required for the experiments.

Chapter 3

Skilled DQN

Chapter 2 analyzed in depth the main concepts of Reinforcement Learning, highlighting the evolution of its algorithms and the current frontiers of research. At the same time, it ventures in the world of Continual Learning first defining this new paradigm and then analyzing the recent intersection between RL and CL.

Relying on the background provided by the previous chapter, the next one will present the main contribution of this work, introducing a new algorithm for Reinforcement Learning which leverages the use of previous knowledge, *skills*, in order to learn and solve a task. As has already been analyzed in Section 2.8, the concept of skill has been used in different RL algorithms with different connotations, this work provides its distinctive definition of this concept in attempt to create RL agents with built-in knowledge that can be used compositionally to easily learn without starting from scratch, similarly to how humans approach and learn new tasks.

This leads to the introduction of our algorithm called *Skilled DQN* (SDQN). As the name suggests it builds on DQN, taking advantage of some improvements of the naive algorithm, exploited to learn the optimal policy for a particular task, but the model will be coupled with a set of pre-trained skills always available to the agent.

The next sections of the chapter will breakdown all the components of the algorithm, in particular: Sections 3.1, 3.2 formally introduce the concept of skills, the models used in this work to provide the agent the additional knowledge and their implementation. Section 3.3 provides the actual implementation of the RL agent and its training procedure using DQN with particular attention on how skills are leveraged 3.4, and lastly, Section 3.5 analyzes the results obtained by SDQN with respect to classical DQN.

3.1 Unsupervised Skills

In Reinforcement Learning, talking about skills is a very general term, that is used in different scenarios with different meanings. In fact, [54] refers at skill as a sequence of consecutive actions that can be exploited by the agent, a similar connotation is used in [72], while [45] refers at skills as sub-policies that are able to solve a particular task and the agent might use to complete a certain routine.

In this work the concept of skill is reconsidered trying to give it a more meaningful idea. Starting from an example: a brand new game has been released and we try to play, our brain has previous knowledge about how games works - how to move, the typology of the game, etc. - but more importantly looking at a frame it is easy to recognize our character, possible enemies or rewards, locate the score or how many lives are left and with a relative small amount of tries we are able to understand the mechanics of the game and eventually get a high score. However, trying to apply the same example to a RL agent the scenario is completely different. The agent usually starts from zero knowledge receiving as input a frame of the game and without any information about the game, during its training process after a significant amount of steps, exploiting the reward signal, the agent eventually learns a policy to play the game and maximize its performance. By doing that the neural network, used to approximate the q function, has learned how to process the game's frames received as input to obtain at the end a distribution over the set of actions.
Despite the astonishing results obtained by state of the art models in RL, these algorithms are becoming more and more a feature engineering challenge building complex architecture trying to always improve the performances of the agent [63]. However, all these models learn to solve one problem "from scratch" without reusing previously learned skills. This is fundamentally against any principle of compositionality which subsumes scalable computer science solutions. The idea underlying this work is to propose an approach that exploits previously learned models, hence re-use knowledge, to solve a task.

At this point, we give a formal definition of skills and explain how this knowledge is created and exploited. Starting from the formulation of the RL problem, the tuple representing the MDP, Section 2.2, is extended as follows: $\langle S, A, R, P, \Sigma, \gamma \rangle$, where Σ is the set of pre-trained skills provided to the model. Each skill, $\sigma : S \to \mathbb{R}^{n \times m \times \cdots}$, is a models that takes as input the current state of the game and process it to produce as output a meaningful representation of some aspect of the current state, which may be a vector, images or filters, next section will cover the skills used in this work. Another important aspect highlighted in the title of the section is the *unsupervised* fashion of this skills. In fact, each one is trained in a unsupervised way using a *task-agnostic* dataset, meaning that data is collected from the environment using a random agent in order to be as general as possible and avoid any possible bias given by a policy underlying the data collection process. The RL agent during the learning process will use the set of skills Σ to pre-process the input frame, compositionally combine all the various outputs of the skills and then exploiting the Q-network produce, as output, a distribution over the set of action A, hence learning an effective policy for the task.

3.2 Skills implementation

Before introducing the skills that have been developed for this work, it is worth giving an insight of the experimental environment in order to better understand the underlying reasons about the skills choice. The environment in which the RL agent using Skilled-DQN will be trained, as already mentioned, is one of the Atari games, in particular `Pong`. Given this scenario the chosen models share some characteristics: use as input one or more images, which can easily be frames of the game, the learning data is a set of images that can be collected by interacting with the environment using a random policy and lastly are related to represent the state of the game or identify key elements of the game - for instance the players paddles or the ball.

The first model taken into consideration it concerns with *Object Keypoints* [58], it introduces a new architecture to discover temporally, spatially and geometrically aligned keypoints, each one representing and tracking the coordinate of and object among various different frames. The model, called *Transporter*, it trains over a set of data composed by video frames pairs (x_s, x_t) which can be easily collected by exploring the environment with a random agent. During the training process the model learns to transform a source video frame x_s into a target frame x_t by transposing image features, in particular, it computes spatial feature maps $\Phi(x)$ using a *CNN* and keypoints coordinates $\Psi(x)$ using *KeyNet* [50], which are brought to the same spatial dimension of feature maps using Gaussian heatmaps $H_{\Psi(x)}$. After computing the features, the transport phase, formally defined as

$$\hat{\Phi}\left(\boldsymbol{x}_s, \boldsymbol{x}_t\right)\left(1-\mathcal{H}_{\Psi\left(\boldsymbol{x}_s\right)}\right) \cdot\left(1-\mathcal{H}_{\Psi\left(\boldsymbol{x}_t\right)}\right) \cdot \Phi\left(x_s\right)+\mathcal{H}_{\Psi\left(\boldsymbol{x}_t\right)} \cdot \Phi\left(x_t\right)$$

combines feature from the source and target image in order to compute a the transported feature map which is fed to the *RefineNet* whose objective is to reconstruct the target image x_t, the architecture of the model is reported in Figure 3.1. During inference the CNN and KeyNet are used to compute the feature maps of the current frame, this will be exploited by the RL agent while applying this skill to the current state of the game.
Listing 3.1 reports the model architecture, with the different layers and parameters.

```
Transporter
(encoder): Encoder
(encoder): Sequential
  (0): Conv2d(3, 32, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
  (1): BatchNorm2d(32, eps=1e-05, momentum=0.1)
  (2): ReLU()
  (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): BatchNorm2d(32, eps=1e-05, momentum=0.1)
  (5): ReLU()
  (6): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (7): BatchNorm2d(64, eps=1e-05, momentum=0.1)
  (8): ReLU()
  (9): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (10): BatchNorm2d(64, eps=1e-05, momentum=0.1)
```

```
    (11): ReLU()
    (12): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (13): BatchNorm2d(128, eps=1e-05, momentum=0.1)
    (14): ReLU()
    (15): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): BatchNorm2d(128, eps=1e-05, momentum=0.1)
    (17): ReLU()
  (key_net): KeyNet
  (keynet): Sequential
    (0): Conv2d(3, 32, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1)
    (2): ReLU()
    (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1)
    (5): ReLU()
    (6): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1)
    (8): ReLU()
    (9): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1)
    (11): ReLU()
    (12): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (13): BatchNorm2d(128, eps=1e-05, momentum=0.1)
    (14): ReLU()
    (15): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): BatchNorm2d(128, eps=1e-05, momentum=0.1)
    (17): ReLU()
  (reg): Conv2d(128, 4, kernel_size=(1, 1), stride=(1, 1))
  (refine_net): RefineNet
  (refine_net): Sequential
    (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1)
    (2): ReLU()
    (3): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1)
    (5): ReLU()
    (6): UpsamplingBilinear2d(scale_factor=2.0, mode=bilinear)
    (7): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(64, eps=1e-05, momentum=0.1)
    (9): ReLU()
    (10): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(32, eps=1e-05, momentum=0.1)
    (12): ReLU()
    (13): UpsamplingBilinear2d(scale_factor=2.0, mode=bilinear)
    (14): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
(15): BatchNorm2d(32, eps=1e-05, momentum=0.1)
(16): ReLU()
(17): Conv2d(32, 3, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
(18): BatchNorm2d(3, eps=1e-05, momentum=0.1)
(19): ReLU()
```

Listing 3.1: Video Object Segmentation Model

The second skill used by the RL exploits a model trained for the task of *Video Object Segmentation* [48], which learns a structured representation of the input capturing information about the moving objects in the video frame. Similarly at the previous model, also in this case the data required to train the skill is a collection of video frames of the game. As far as concerns the architecture and training procedure, the model given two consecutive frames x_0, x_1 predicts K *object segmentation masks* $M \in [0,1]^{W \times H}$. It compresses the input images to a 512-dimensional embedding, which contains information about the moving objects in the input frames. From the embedding two different components branch out: the first one consists in a fully-connected layer that computes object translation t_k and camera translation c, while the other one up-samples the embedding via bi-linear interpolation and predicts the K *object masks*. To correctly train the model, since there is not ground truth, it learns to interpolate between the two input frames x_0, x_1, in more detail: using the object masks and the translations it computes the optical flow F, defined as $F_{ij} = \sum_k (M_{ij}^k \times t^k) + c$, and use it to wrap the frame x_1 into an estimate of x_0, $\hat{x_0}$, which can be done differentiably using the methods proposed in [28], thereafter, structural dissimilarity (DSSIM) [15] is used as distance metric between the two images. The RL agent will exploit this skill to compute a reconstruction of the current frame coupled with the respective objects masks.
Listing 3.2 reports the model architecture, with the different layers and parameters.

```
VideoObjectSegmentationModel
(cnn): Sequential
    (0): Conv2d(2, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
    (6): Flatten()
(relu): ReLU()
(sigmoid): Sigmoid()
(fc_conv): Linear(in_features=3136, out_features=512, bias=True)
(obj_trans): Linear(in_features=512, out_features=40, bias=True)
(cam_trans): Linear(in_features=512, out_features=2, bias=True)
(fc_m1): Linear(in_features=512, out_features=10584, bias=True)
(conv_m1): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv_m2): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv_m3): Conv2d(24, 20, kernel_size=(1, 1), stride=(1, 1))
(upsample1): UpsamplingBilinear2d(size=(42, 42), mode=bilinear)
```

```
(upsample2): UpsamplingBilinear2d(size=(84, 84), mode=bilinear)
```

Listing 3.2: Video Object Segmentation Model

Last but not least, the last model differs from the previous ones since it does not provide any visual feature, instead it computes a representation of its state, hence solving the task of *State Representation* [56]. The model given a set of observations X, that are collected using a random agent, aims at learning an abstract representation that grasps the latent factors of the environment, focusing on high-level semantics (*e.g.* players, enemies, score) and ignore low-level details such as the background. The approach relies on maximizing an estimate of the mutual information over consecutive observations, this is achieved sampling data and dividing in *positive* samples, couple of consecutive samples (x_t, x_{t+1}), or *negative* samples that correspond to pairs of non-consecutive observations (x_t, x_{t^*}). During the training process the model computes global features over the current frame and local features for positive/negative samples, using bi-linear models it computes a scoring function for both samples combining global and local features, these functions are then exploited by a discriminator to correctly classify the positive samples. The RL agent will use as skill the global features computed by the encoder to access the state representation of the current frame. Figure 3.3 shows the training process visually and for a more formal description we refer to [56][52].
Listing 3.3 reports the model architecture, with the different layers and parameters.

```
NatureCNN
(main): Sequential
    (0): Conv2d(1, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2))
    (5): ReLU()
    (6): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1))
    (7): ReLU()
    (8): Flatten()
    (9): Linear(in_features=3456, out_features=256, bias=True)
```

Listing 3.3: State Representation Model

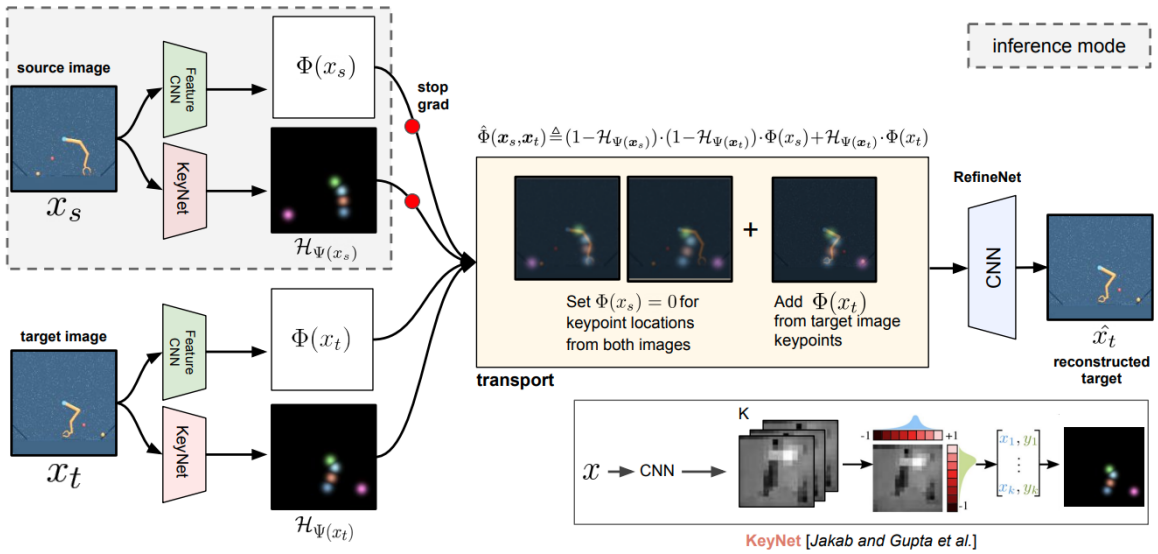


Figure 3.1: Complete architecture of *Transporter* [58]. The model discovers object keypoints by learning to transform a source frame x_s into target frame x_t by transporting images features. Spatial feature maps $\Phi(x)$ and keypoints coordinates $\Psi(x)$ are predicted using *CNN* and *KeyNet* [50]. After the transport phase the final *RefineNet* predicts the reconstructed target frame $\hat{x}_t$.

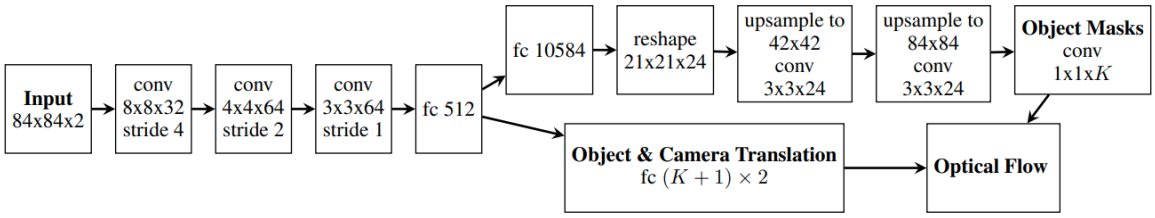


Figure 3.2: Unsupervised object segmentation model architecture [48]. The upper branch computes the object masks while the lower one predicts object and camera translations.

3.3 DQN Implementation

When implementing DQN there are two main aspects that must be taken into consideration: how the agents acts in the environment, so which actions performs, and how to update the model in a way that will effectively learn to solve the task. The former, also described in Section 2.3.2, is addressed by taking advantage of the exploration and exploitation trade-off, to be precise, an ϵ-greedy strategy is implemented, the variable ϵ balances the ratio between the two strategy and its value will be decreased after each episode, the less its value the more the agent exploits its policy. The latter, instead, to be dealt with requires more components, in particular the *replay memory* and *target network*.

The memory replay, or experience replay, as the agent interacts with the environment it

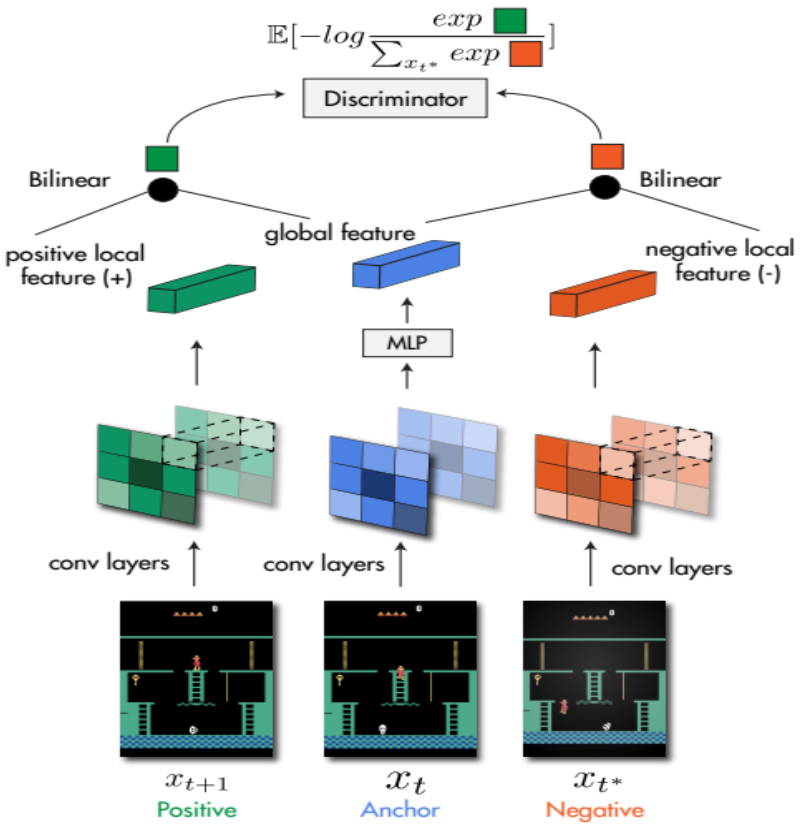


Figure 3.3: A schematic overview of SpatioTemporal DeepInfoMax (ST-DIM) [56]. Simplified version of the global-local contrastive task.

stores state transition tuples $(s, a, s', r, done)$ using a doubly ended queue and it allows to easily add and gather data that will be used to update the network, moreover data can be sampled randomly in order to remove correlation between samples - sampling state transition tuples of different episodes.
The target network $\hat{Q}$[30] improves the stability of the algorithm, reducing oscillations or divergence of the policy, and is used to compute the Q-Learning targets y which are employed to compute the loss. The target network is initialized with the same weight of the Q network, during the training process it will be updated at a given rate, furthermore, instead of completely overwrite the weights of the target network a soft target update approach is applied, meaning that the update is adjusted by a parameter τ according to the following rule:

$$\hat{Q}_\theta = \tau Q_\theta + (1 - \tau)\hat{Q}_\theta$$

Smaller but more frequent updates of target network should lead to a more stable learning. On the other hand, the Q network update is performed by sampling a N state transition

tuples from the replay memory and computing the loss, MSE, with respect to expected q-values using the targets calculated using the target network.
Algorithm 1 describes the structure of the DQN algorithm using the OpenAI Gym interface to interact with the environment.

Algorithm 1: Deep Q-Learning (DQN)

Input : *DQN Q, target DQN* $\hat{Q}$*, replay memory of size N, env, episodes, episodes_steps, eps, eps_min, eps_decay, update_every_C_steps,* γ, τ

begin
  for *e* `in` range(episodes) **do**
    state = env.reset()
    done = False
    ep_steps = 0
    while `not` *done* `and` *ep_steps* < episodes_step **do**
      `# ϵ-greedy exploration`
      if *random.uniform(0,1)* < eps **then**
        action = env.action_space.sample()
      else
        action = $\arg\max_a Q_\theta(s, a)$
      end
      s', reward, done, _ = env.step(*action*)
      `# store transition in replay memory`
      memory.push(*state, action, reward, s', done*)
      ep_steps += 1
      state = *s'*
      if *ep_steps* % update_every_C_steps **then**
        `# fit model`
        `# sample batch from replay memory`
        states, actions, rewards, states_, dones = memory.sample(batch_size)
        y = $rewards + \gamma max_{a'} \hat{Q}_{\theta'}(states\_, a')$
        loss = (y - $Q_\theta(states, actions)$)^2
        optimizer.step()
        `# soft-update target network` $\hat{Q}$
        $\hat{Q}_\theta = \tau Q_\theta + (1 - \tau)\hat{Q}_\theta$
      end
    end
    `# decrease` ϵ
    eps = max(eps_min, eps*eps_decay)
  end
end

3.4 Skills Integration

A critical and fundamental aspect of the proposed method, Skilled-DQN, is related to how skills are introduced and exploited by the agent during the learning process. In particular, these skills are neural network models that, in a unsupervised fashion, learn to solve a particular task. The problems solved by these models can be very different, for example the tasks can range from state representation to object segmentation as previously described, this variety implies that the models' output differs semantically but also in the output space, hence finding an effective way to combine these information together is a key requirement.

Section 3.2 has introduced the models that have been chosen and implemented in this work. The first two models described given an image compute an output still in the image space, which represent the encoding of the image and keypoints or segmentation masks, while the last model creates a linear representation of the current state of the game. Given the two different topology of data, the way in which combine this information has been one of the focuses of this work. The following considerations are based on the scenario featured in this project, however, it can be easily adapted and abstracted to any possible scenario where the skill models solve orthogonal problem with heterogeneous outputs.

In this project we address different approaches in order to combine different skills and provide them to the RL agent. The first idea, which is the one implemented and applied in the experiments, is to bring the outputs of the skills to the same space, in this case the image-space, and combine them compositionally in order to create a single data structure that contains all the information. In particular, the output of the first two skills - object keypoints and object segmentation -, $o_i \in \mathbb{R}^{C_i \times n_i \times m_i}$, exploiting learnable adapters are transformed to the same image-space, $o_i \in \mathbb{R}^{C_i \times k \times j}$, on the other hand the state representation model's output, $o \in \mathbb{R}^n$, is reshaped to match the dimension featured by the other two skills. The skills, combined in a single data structure, are then provided as input to the RL agent's network upon which it will learn the policy interacting with the environment.
Another idea, similar to the one just described, is to combine all the skills encoding their information in a linear space $\mathbb{R}^n$. However, this approach may result worse to be applied since encoding all data into a linear space would result in a huge sized representation and at the same time it will disentangle the multidimensional data from their original meaning and characterization. A possible solution to this problem would be to add some layers after the skills, before combining them, which during training will learn how to efficiently transform data to a different dimension without loosing information.
Finally, the last idea takes a different approach and based on the dimensionality of the skills it exploits the information at different levels of the agent's network. To be more precise, the skills are divided with respect to their dimension - i.e. linear, images - and are combined to the model's network at different levels. The output of multi-dimensional skills are combined and provided to the agent's network as input, while the linear skills are later introduced once the network has processed input data and has reached a linear representation of it inside its network. In this way the skill's feature structure is kept while utilizing it, however, it is important to find and build the correct structure to introduce this knowledge in the model without overwhelming its internal representation of the data.

3.5 Experiments

This section introduces the experimental setup that was used to train the RL agent using the proposed approach, Skilled DQN, providing also a comparison of performances with DQN.

For this experiments the RL agent learns to play a single game, in particular we used one of the games from the Atari suite exploiting the interface available from `OpenAI Gym`. As far as concerns the architecture, the agent's network is implemented using the structure of progressive networks instantiating one single column, which is equivalent to defining a neural network (formal definition 4.1).

In this scenario two different agents are created one exploiting the naive version of `DQN`, Alg. 1, the other one leveraging the proposed approach `Skilled DQN`, whose architecture is reported in Figure 3.7. The two agents, as reported in Table 3.1, share the same architecture, the only and major difference is the input size, in fact DQN agent receives one single frame of the game, while, Skilled DQN agent leverages the set of pre-trained skills Σ to pre-process the game frame and provide it as input. In order to do a fair comparison between the two algorithms and effectively study possible differences in performance the same hyperparameters were used, a complete description is reported in Table 3.2. The environment used is the game `Pong`, in this scenario a reward of $+1$ is awarded each time the RL agent scores a point, and -1 when it concedes a goal to the hard-coded agent (CPU), Fig. 3.4. The agents are trained over 10^5 steps, which is a very small number of episodes - the limit is imposed by the time and computation resources required -, nevertheless, both algorithms are able to reach solid results.

Parameter	Value
activation function	*ReLU*
optimizer	Adam [26]
	DQN
input	$3 \times 84 \times 84$
NN architecture	[*Conv2D*(3,32,3,1,1) *Conv2D*(32,32,3,1,1) *Linear*(256) *Linear*(6)]
#parameters	58,453,136
	Skilled DQN
input	$156 \times 16 \times 16$
NN architecture	[*Conv2D*(154,32,3,1,1) *Conv2D*(32,32,3,1,1) *Linear*(256) *Linear*(6)]
skills	*Object Keypoints, Video Object Segmentation, State Representation*
#parameters	2,791,024

Table 3.1: Models parameters and architecture.

Hyperparameter	Value	Description
env	PongNoFrameskip-v4	Atari environment
action space	6	Number of possible actions in the environment
learning rate	5×10^{-4}	Initial learning rate
batch size	64	Number of state transition tuples sampled from the memory during the update
eps init	1	Initial value for the ϵ-greedy exploration
eps decay	0.995	Decay factor for ϵ at each step
eps min	0.02	Minimum value for ϵ
gamma	0.97	Discount factor
tau	0.05	Parameter for the soft-update of target network
max episodes	10^5	Maximum number of episodes used for training
max step episode	10^4	Maximum number of steps for each episode
update rate	100	Update the network every C episodes
memory size	5×10^4	Size of memory replay

Table 3.2: Hyperparameters for both experiments *DQN* and *Skilled DQN*.

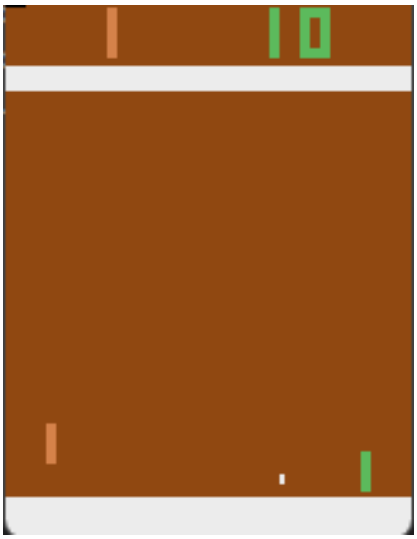
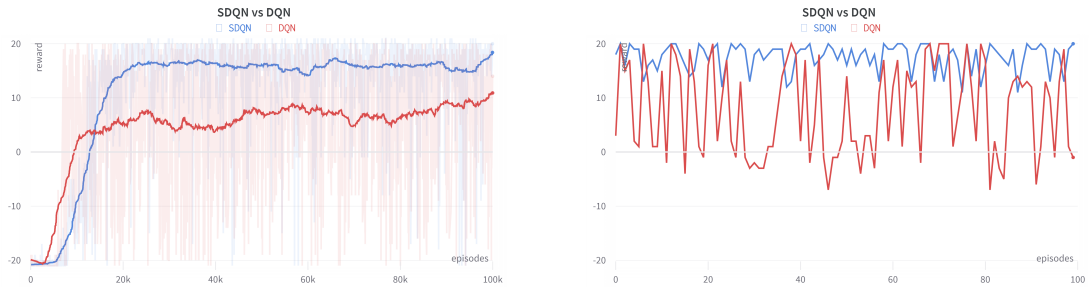


Figure 3.4: Pong game frame, CPU vs RL agent.

Figure 3.5a reports the results for the experiments, our approach is able to reach better performances with higher cumulative reward, average reward of ~ 17, in far fewer steps than the DQN agent. As previously mentioned, the reward is strictly correlated with the scores of RL agent and CPU, as a matter of fact, Figure 3.6 shows in detail the points of the two players, while both agents almost all the time are able to win, the main difference is the amount of points allowed to the CPU, the less the higher the reward. A peculiar and important aspect that differentiates the two approached is related to the model sizes, in fact as reported in Table 3.1, the SDQN agent has way less learnable parameters, almost $30\times$



(a) (b)

Figure 3.5: Comparison between Skilled DQN vs DQN RL agents. SDQN is reported in blue while DQN in red. The plots show training (a) and validation (b) results of the two algorithms.

less, nonetheless reusing pre-trained knowledge provided by the unsupervised skills is able to achieve amazing performances and outperform the DQN agent, in a simple scenario as Pong. Figure 3.5b reports the testing of the two agent evaluated over 100 episodes, SDQN agent achieves 100% win rate with a small variance in the total reward among all episodes, on the other hand the DQN agent is more unstable reporting high scores but also some losses.

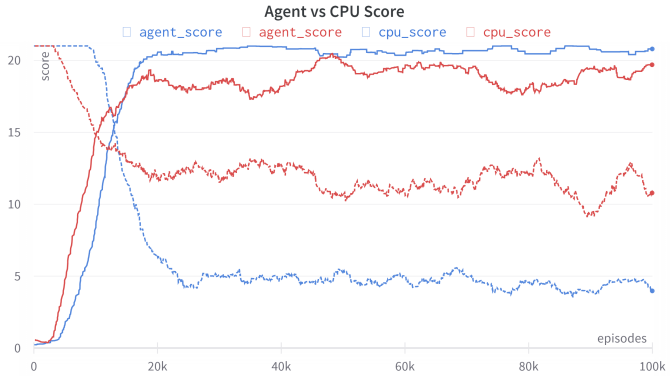


Figure 3.6: Comparison between the Agent and CPU score for both algorithms.

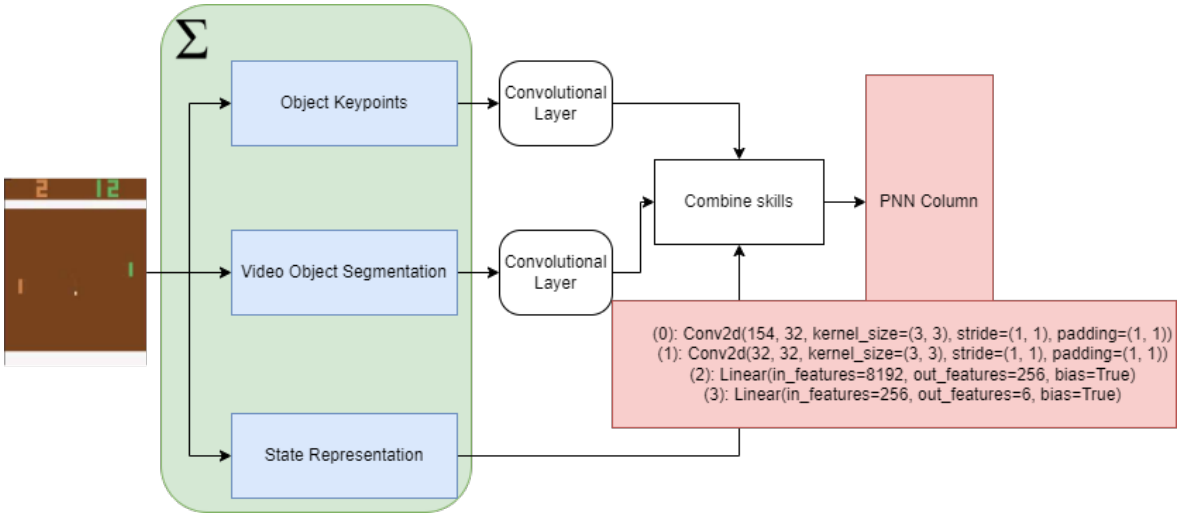


Figure 3.7: Skilled DQN model architecture. As introduced in Section 3.1 and 3.2 the set of pre-trained skills Σ is composed by three models. Section 3.4 presented how skills are leveraged to pre-process the input frames and learn a policy. The actual model, PNN Column, is relative small with two convolutional layers and two linear layers.

Chapter 4

Learning New Games

This chapter covers the second part of the work trying to extend the results reported in Chapter 3 embracing and challenging Continual Learning scenarios. Up to now the analysis and experiments dealt with the agent having to learn to play at a single game while leveraging prior knowledge, on the other hand, the following sections will address the ability to learn and play more than on game simultaneously.

Continual Reinforcement Learning, 2.7, is acquiring more and more interests in the research community and this segment of the work tries to address one of the main problems which is multi-task learning meaning the ability of the agent to learn from various different MDPs. This scenario can be more challenging than single task learning as different environments may require conflicting policies in order to achieve success. The following sections analyze if Skilled-DQN can be a valid alternative to current RL algorithms and its impact in agents' performances, since it introduces an additional layer of reusable knowledge through skills that should adapt to dynamics environments.

In particular, Section 4.1 will present more in details Progressive Neural Networks [35], which is the architecture implemented for the experiments, and how skills are exploited in this model, while, Section 4.2 will present and compare results obtained PNN, while learning two games at the same time, with other approaches that learn one single task.

4.1 Progressive Neural Networks

Neural networks in order to reach and acquire human-level intelligence must learn to solve complex tasks, leveraging previous knowledge and avoiding catastrophic forgetting. One of the main methods to satisfy these requirements is *fine tuning*: a model is trained on a initial domain and via backpropagation the last layers are adapted to the target domain. However, this approach has various drawbacks which make it unsuitable for transferring knowledge across different tasks. For instance, while fine tuning allows to improve performances in the target task this is a destructive process that modifies the previously learned features. Another aspect is that this approach exploits previous knowledge only at initialization complicating the process of defining the correct model initialization so the tuning process is effective, but also leading to a limited ability to maintain and propagate knowledge across different tasks.

On the other hand, Continual Learning aims at creating agents that not only learn a series of tasks but also have the ability to transfer knowledge from previous tasks. Moreover, CL particularly focuses on the problem of catastrophic forgetting, in fact to address these problem there is a variety of strategies that can be divided into non-exclusive categories [68] - already presented in Sec. 2.6 but reported as reminder -: *Rehearsal* based which refers to methods that save samples as memory of the past to reduce catastrophic forgetting, *Regularization* based - e.g. Elastic Weight Consolidation (EWC) [39] - introducing a penalty term in order to prevent the update of task-essential weights, and lastly *Architectures* approaches which modify the architecture of the most so it learns new features without interfering with old ones.

With regard to the last category a famous approach, and key for this second part of the work, are *Progressive Neural Networks* (PNN) [35], which introduce a model architecture with explicit structure for transfer knowledge across sequences of tasks. Differently from fine tuning, where prior knowledge is provided only at initialization, progressive networks maintain a set of pre-trained models throughout all training process and learn lateral connections to extract useful features for new tasks. In particular, the addition of new network alongside the pre-trained models allow the models to achieve a higher degree of compositionality since they can re-use old knowledge but also learn new features and combine them together.
Following this general introduction we now provide the formal definition highlighting the various features just mentioned. First and foremost, PNN avoid catastrophic forgetting by instantiating a new neural network - *column* - for each task, knowledge transfer between networks is achieved by adding lateral connections to previous columns. The model is initialized with a single column, which is a neural network having L layers, parameters Θ^1 and hidden activations $h_i^1 \in \mathbb{R}^{n_i}$, where n_i is the number of units at layer $i < L$, and is trained to solve a particular task. When approaching a second task, the parameters of the first model Θ^1 are frozen and a new column with parameters Θ^2 is introduced, but in this case each layer h_i^2 receives input from both h_{i-1}^2 and h_{i-1}^1 via lateral connections. Introducing further new tasks the process is the same, hence freezing all the previous columns and exploit lateral connections from them in order to transfer knowledge, the following formula generalizes the case of K tasks:

$$h_i^k = f(W_i^k h_{i-1}^k + \sum_{j<k} U_i^{(k:j)} h_{i-1}^j) \tag{4.1}$$

Where $W_i^k \in \mathbb{R}^{n_i \times n_{i-1}}$ is the weight matrix for layer i of column k, $U_i^{(k:j)} \in \mathbb{R}^{n_i \times n_j}$ is the matrix representing lateral connections from layer $i-1$ of column j to layer i of column k, f is the activation function (element-wise non-linearity). Figure 4.1 shows the architecture for $K = 3$.

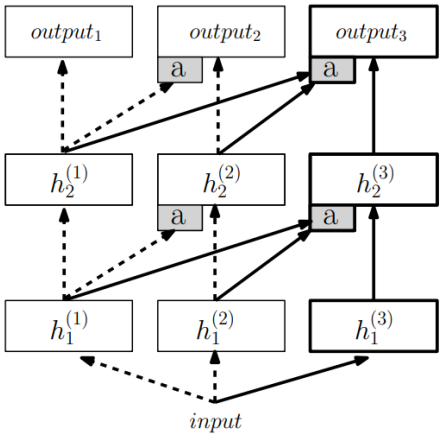


Figure 4.1: Representation of a three column progressive network [35]. The first two columns are trained on task t_1, t_2 while a third column is added for the final task having access to previously learned features. The gray box represent the adapter layers (see below).

As the number of columns K grows is important to limit the growth of the number of parameters, especially induced by lateral connections. Therefore *adapters* are introduced, with the objective of improving initial conditioning and perform dimensionality reduction, in particular: a vector of previous features $h_{i-1}^{(<k)}$ is weighted by a learned scalar, whose role is to adjust different scales of various columns. The weighted vector is fed to a single layer MLP which replaces the linear lateral connections, the hidden layer of non-linear adapters is projected to a n_i dimensional subspace, in order fix the number of parameters of lateral connections. Equation 4.1 is updated as follows:

$$h_i^k = f(W_i^k h_{i-1}^k + U_i^{(k:j)} f(V_i^{(k:j)} \alpha_{i-1}^{(<k)} h_{i-1}^{(<k)})) \tag{4.2}$$

Where $V_i^{(k:j)} \in \mathbb{R}^{n_{i-1} \times n_{i-1}^{(<k)}}$ is the projection matrix and α is the learned scalar value. While the adapters previously described apply to dense layers, dimensionality reduction for convolutional layers is performed via 1×1 convolutions [22].

In this work PNN are applied to RL scenarios, each column is trained to solve a different MDP, in particular, column k defines and learns a policy $\pi^k(a|s)$ for the k^{th} task leveraging previous columns $\pi^{(1:k-1)}$. Moreover, in the experiments each column is coupled with its set of pre-trained unsupervised skills Σ which are exploited as described in Section 3.4.

4.2 Experiments

This section outlines the experiments related to Progressive Neural Networks, comparing its performances with simple RL agents that learn via Skilled DQN or DQN.

In this experiment the agents challenge a more complex game from the Atari suite `Pacman`, Figure 4.2, in fact it has a large state space, e.g. possible position of pac-man and ghosts inside the board, it requires a good amount of exploration in order see the whole environment. The reward function assigns the agent a positive reward for eating dots, fruits and ghosts - when allowed - no negative reward is assigned, still the agents has only three life that loses when eaten by one of the four ghosts, once all three life are lost the episode's end is reached. In this scenario we compare three different approaches, first we train two models, exploiting DQN and Skilled DQN, to learn only the game of Pacman, then we create a two column progressive network, where the first column is the SDQN model from Section 3.5 and the second one is a new network which effectively learns to play Pacman. Both the SDQN and the PNN agents are provided with the same set of prior skills, but in the second case the model, exploiting lateral connections, leverages also knowledge from the network trained for Pong.

Table 4.1 reports the architectures of the three different models, while Table 4.2 the hyperparameters used in the experiments. The agents are trained for 10^5 episodes, which given the complexity of the task are a very small number of steps - most of them are used for exploration - but due to time and computational constraints we limited the experiments within this extent, nevertheless we obtained interesting results.

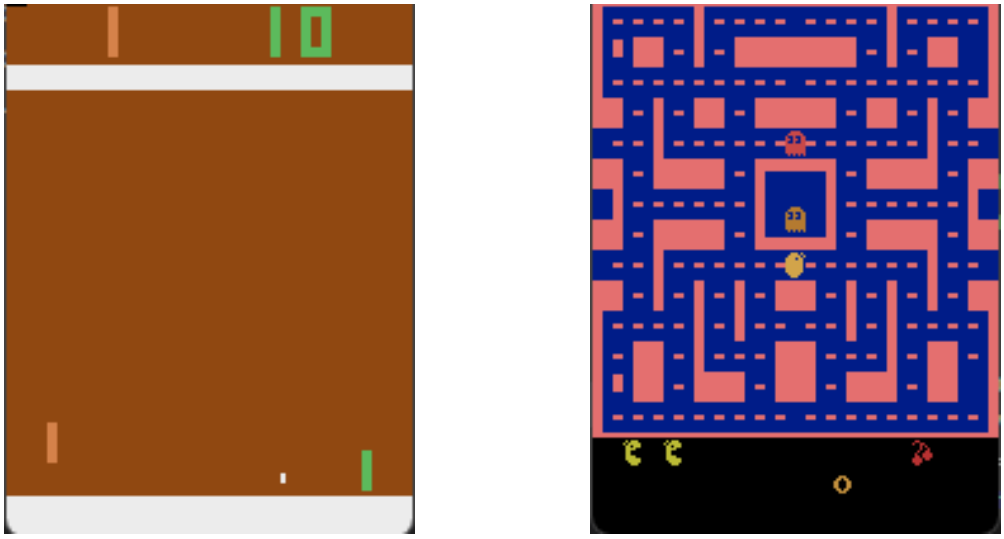
Parameter	Value
activation function	*ReLU*
optimizer	Adam [26]
	DQN
input	$3 \times 84 \times 84$
NN architecture	[*Conv2D*(3,32,3,1,1) *Conv2D*(32,32,3,1,1) *Linear*(256) *Linear*(6)]
	Skilled DQN
input	$156 \times 16 \times 16$
NN architecture	[*Conv2D*(154,32,3,1,1) *Conv2D*(32,32,3,1,1) *Linear*(256) *Linear*(6)]
skills	*Object Keypoints, Video Object Segmentation, State Representation*
	PNN
input	$156 \times 16 \times 16$
NN architecture	[*Conv2D*(154,32,3,1,1) *Conv2D*(32,32,3,1,1) *Linear*(256) *Linear*(6)]
skills	*Object Keypoints, Video Object Segmentation, State Representation*

Table 4.1: Models parameters and architecture.

Hyperparameter	Value	Description
env	MsPacmanNoFrameskip-v4	Atari environment
action space	9	Number of possible actions in the environment
learning rate	1×10^{-4}	Initial learning rate
batch size	64	Number of state transition tuples sampled from the memory during the update
eps init	1	Initial value for the ϵ-greedy exploration
eps decay	0.99995	Decay factor for ϵ at each step
eps min	0.1	Minimum value for ϵ
gamma	0.97	Discount factor
tau	0.05	Parameter for the soft-update of target network
max episodes	10^5	Maximum number of episodes used for training
max step episode	10^4	Maximum number of steps for each episode
update rate	100	Update the network every C episodes
memory size	5×10^4	Size of memory replay

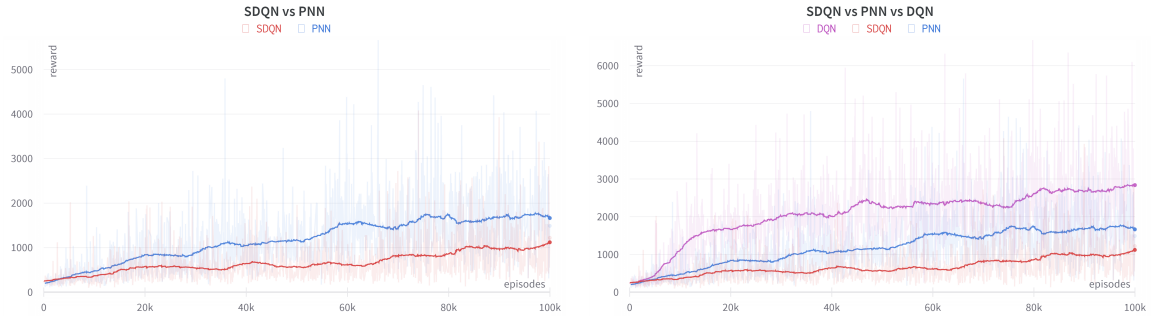
Table 4.2: Hyperparameters for all experiments *PNN, Skilled DQN and DQN*.

Figure 4.3a shows the comparison between the SDQN and PNN agent, both agents slowly learn to play the but there is a clear difference between the two approaches. Despite using the set of pre-trained skills the SDQN agent is able to reach an average score around 1000, on the other hand, PNN using the same set of abilities but adding, via lateral connections, the possibility to re-use features from the Pong column is able to outperform the other approach achieving an average score of 2000. Given that the two models differs in the amount of prior knowledge, the gap in performances can be attributed to the fact that PNN agent is able to leverage Pong's features to improve and ease his learning of the new environment.
Figure 4.3b includes in the comparison also the DQN agent, the latter reaches a higher average score with respect to the previously mentioned agents. At first this could come as surprise, but there are various factors that must be taken into consideration: the DQN agent reaches the range of performances also reported in [30], a possible explanation is that the other two agents are under-performing given the reduced amount of parameters and training, moreover, as previously mentioned the number of steps exploited for the training process is very limited. In particular, related to this second aspect we tried to address this problem allowing PNN and DQN agent to train for approximately an additional 50k steps, still a small number of episodes but enough to appreciate a strong improvement in performances. Table 4.3 reports the average score obtained during validation of the model obtained with extended episodes, the gap between the score is significantly decreased with respect to Figure 4.3b where the DQN is improved by a small margin while, on the other hand, PNN agent has increased by ~ 700 points. With just a small amount of steps PNN agent is able to reduce the gap, with the correct amount of episodes eventually it can reach the same performances as DQN agent if not better.



(a) (b)

Figure 4.2: Atari games involved in the experiments: Pong (a) and Ms.Pacman (b). *PNN* agent learns to play both games at the same time, while *SDQN/DQN* agents only learn to play (b).



(a) (b)

Figure 4.3: Comparison between the average reward achieved by the various algorithms. On the left (a), *SDQN* (red) vs *PNN* (blue), on the right is also reported the data of *DQN* agent (purple).

Model	Reward	Steps
DQN	3126.8	981.08
PNN	2761.4	705.24

Table 4.3: Average reward and steps over 100 episodes between DQN and PNN agent extending training episodes.

Chapter 5

Ablation Study

5.1 Skills Relevance

An important aspect in the Machine Learning community is related to models' explainability, in fact often neural networks are seen as black boxes able to solve a particular task with high accuracy. As [66] states, the majority of current RL research is being conducted in an engineering context, rather than as part of an effort to model brain function, more and more complex architecture are exploited to achieve better results.

In this work we tried to make a step towards a more explainable and human-like model: the agents is able to exploit prior knowledge, both skills and other games policy, in order to learn to solve a new task. In particular, Chapter 3 introduced an analyzed in depth our approach called Skilled DQN, Section 3.1 formally introduced the concept of unsupervised skills and 3.5 reported the results achieved by this algorithm. This section aims at better understand how the unsupervised skills are used and, in particular, how they impact the performances of the agent. In this analysis the agents interact with the environment for 100 episodes but the various skills are zeroed, hence provide no information to the agent.
Table 5.1 reports the average reward of the various experiments.

Model	Reward
SDQN	19.4
SDQN no state representation	18.6
SDQN no object masks	18.4
SDQN no state representation and object masks	18
SDQN no object keypoints	-19

Table 5.1: Average reward over 100 episodes by inhibiting the various skills.

The results shows that the various skills have a different impact in the performances of the agent, which could be in some way predicted based on the dimension of skills' feature encoding.

In particular, the skills have the following dimensions:

- State Representation: $1 \times 16 \times 16$
- Object Masks: $21 \times 16 \times 16$
- Object Keypoints: $132 \times 16 \times 16$

Object Keypoints is the predominant skill, as a matter of fact, by inhibiting it the agent is lost and is no longer able to play. Nevertheless, the analysis is interesting since it shows that the agent exploiting all skills achieves the highest score meaning that all skills, some more some other less, impact agent's behaviour and all of them provide some knowledge which is exploited to refine the policy. In other scenarios, where the agents is given a big set of skills, this analysis would help to identify the most important abilities, study and understand how their information is exploited by the agent and also possibly perform pruning of useless knowledge to introduce new one. Other experiments can also try to add noise to the game frames in order to study how the skills react and if the agent is still able to play the game.

Chapter 6

Conclusion & Future work

This thesis presented a novel approach in the field of Reinforcement Learning, but also reaching the emerging Continual Reinforcement Learning. The new algorithm, called `Skilled DQN`, extends the naive version of Deep Q-Networks by providing the agent a structured prior knowledge that can be exploited to easily learn new tasks.

The key contributions of this work are:

- A formal definition of prior knowledge, that the agent can access during training, via Unsupervised Skills, a skill is a pre-trained task-agnostic model which is able to solve a simple task. Given a set of skills Σ the agent exploits this knowledge to pre-process and/or enrich the input, or the features encoded in its network, moreover, various methodologies to compositionally combine and exploit the prior knowledge given by skills are presented and analyzed.
- In order to build an environment prone to various different experiments all the skills and progressive networks are implemented from scratch - both architecture and training algorithm - exploiting Pytorch framework.
- The implementation of a novel approach called Skilled DQN (SDQN), which trains a RL agent to play games from the Atari suite while leveraging a set of Unsupervised Skills. Further, a possible methodology to understand the relevance of skills is presented, together with a in-depth comparison with respect to the classical DQN algorithm.
- The integration of the proposed approach in Continual Reinforcement Learning scenarios, in particular, implementing and using Progressive Neural Networks in order to create agents that can learn more than one game simultaneously. This architecture, that allows knowledge transfer via lateral connections, is extended to receive and exploit prior knowledge composed by Unsupervised Skills. This approach is compared with other algorithms analyzing the differences in performances but most importantly how previous knowledge is leveraged.

Future Work

This thesis is just the beginning of a broader and more complex project that can be developed under different aspects. This projects has various levels of complexity and a variety of problems that should be addressed and analyzed exhaustively in order to determine the most effective approaches.
In particular, the most important features that we are looking to take on are:

- Extend the formalization of Unsupervised Skills, attempting to create "general purpose" skills, that are skills that can solve a particular task on more than one environment at the same time. Also increase the number of abilities and analyze how the agent leverages this prior knowledge, if it can be dynamically extended and adjusted, i.e. add a new skills during or after training.

- Investigate possible approaches to compositionally combine skills, understand the most effective way to combine the prior knowledge especially having abilities with different structure and semantics.

- Deepen the experiments of Skilled DQN on a bigger variety of environments, but also attempt to add the "Skilled" component to other state-of-the-art algorithms.

- Extend Progressive Neural Networks to access a variable set of Unsupervised Skills general for all the columns. This is strictly correlated with the first bullet point, that is the possibility to create skills that can solve unsupervised tasks on different environments at the same time.

Bibliography

[1] Imre Csiszár. "I-divergence geometry of probability distributions and minimization problems". In: *The annals of probability* (1975), pp. 146–158.

[2] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[3] Michael McCloskey and Neal J Cohen. "Catastrophic interference in connectionist networks: The sequential learning problem". In: *Psychology of learning and motivation*. Vol. 24. Elsevier, 1989, pp. 109–165.

[4] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3 (1992), pp. 279–292.

[5] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3 (1992), pp. 229–256.

[6] Sebastian Thrun and Anton Schwartz. "Issues in using function approximation for reinforcement learning". In: *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*. Vol. 6. 1993, pp. 1–9.

[7] Christopher M Bishop. "Mixture density networks". In: (1994).

[8] Anthony Robins. "Catastrophic forgetting, rehearsal and pseudorehearsal". In: *Connection Science* 7.2 (1995), pp. 123–146.

[9] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[10] Tom M Mitchell and Tom M Mitchell. *Machine learning*. Vol. 1. 9. McGraw-hill New York, 1997.

[11] Ronald Parr and Stuart Russell. "Reinforcement learning with hierarchies of machines". In: *Advances in neural information processing systems* 10 (1997).

[12] Robert M French. "Catastrophic forgetting in connectionist networks". In: *Trends in cognitive sciences* 3.4 (1999), pp. 128–135.

[13] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems* 12 (1999).

[14] Richard S Sutton, Doina Precup, and Satinder Singh. "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial intelligence* 112.1-2 (1999), pp. 181–211.

[15] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. "Image quality assessment: from error visibility to structural similarity". In: *IEEE transactions on image processing* 13.4 (2004), pp. 600–612.

[16] Shalabh Bhatnagar, Mohammad Ghavamzadeh, Mark Lee, and Richard S Sutton. "Incremental natural actor-critic algorithms". In: *Advances in neural information processing systems* 20 (2007).

[17] Jan Peters and Stefan Schaal. "Natural actor-critic". In: *Neurocomputing* 71.7-9 (2008), pp. 1180–1190.

[18] Li Deng. "The mnist database of handwritten digit images for machine learning research [best of the web]". In: *IEEE signal processing magazine* 29.6 (2012), pp. 141–142.

[19] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. "The arcade learning environment: An evaluation platform for general agents". In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.

[20] Alex Graves. "Generating sequences with recurrent neural networks". In: *arXiv preprint arXiv:1308.0850* (2013).

[21] Diederik P Kingma and Max Welling. "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[22] Min Lin, Qiang Chen, and Shuicheng Yan. "Network in network". In: *arXiv preprint arXiv:1312.4400* (2013).

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Daan Antonoglou Ioannis anWierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[24] Razvan Pascanu and Yoshua Bengio. "Revisiting natural gradient for deep networks". In: *arXiv preprint arXiv:1301.3584* (2013).

[25] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Holger Bougares Fethi anSchwenk, and Yoshua Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[26] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[27] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, and Martin Wierstra Daan anRiedmiller. "Deterministic policy gradient algorithms". In: *International conference on machine learning*. PMLR. 2014, pp. 387–395.

[28] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. "Spatial transformer networks". In: *Advances in neural information processing systems* 28 (2015).

[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Marc G Veness Joel anBellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, and Georg anothers Ostrovski. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[31] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, and Jie Tanand Wojciech Zaremba. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[32] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. "RL^2: Fast reinforcement learning via slow reinforcement learning". In: *arXiv preprint arXiv:1611.02779* (2016).

[33] Alexander Gepperth and Barbara Hammer. "Incremental learning algorithms and applications". In: *European symposium on artificial neural networks (ESANN)*. 2016.

[34] Nikolaus Hansen. "The CMA evolution strategy: A tutorial". In: *arXiv preprint arXiv:1604.00772* (2016).

[35] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, Koray Kirkpatrick Jameand Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. "Progressive neural networks". In: *arXiv preprint arXiv:1606.04671* (2016).

[36] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, George Sifre Laurent anVan Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, PanneershelvamVeda, Marc Lanctot, et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[37] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[38] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Munos-Remi, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. "Learning to reinforcement learn". In: *arXiv preprint arXiv:1611.05763* (2016).

[39] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Andrei A Desjardins Guillaume anRusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. "Overcoming catastrophic forgetting in neural networks". In: *Proceedings of the national academy of sciences* 114.13 (2017), pp. 3521–3526.

[40] Michail G. Lagoudakis. "Value Function Approximation". en. In: *Encyclopedia of Machine Learning and Data Mining*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2017, pp. 1311–1323. ISBN: 9781489976871.

[41] Zhizhong Li and Derek Hoiem. "Learning without forgetting". In: *IEEE transactions on pattern analysis and machine intelligence* 40.12 (2017), pp. 2935–2947.

[42] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. "icarl: Incremental classifier and representation learning". In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2017, pp. 2001–2010.

[43] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Arthur Lai Matthew anGuez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm". In: *arXiv preprint arXiv:1712.01815* (2017).

[44] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Thomas Guez Arthuand Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. "Mastering the game of go without human knowledge". In: *nature* 550.7676 (2017), pp. 354–359.

[45] Chen Tessler, Shahar Givony, Tom Zahavy, Daniel Mankowitz, and Shie Mannor. "A deep hierarchical approach to lifelong learning in minecraft". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 1. 2017.

[46] Joshua Achiam. "Spinning Up in Deep Reinforcement Learning". In: (2018).

[47] Rachit Dubey, Pulkit Agrawal, Deepak Pathak, Thomas L Griffiths, and Alexei A Efros. "Investigating human priors for playing video games". In: *arXiv preprint arXiv:1802.10217* (2018).

[48] Vikash Goel, Jameson Weng, and Pascal Poupart. "Unsupervised video object segmentation for deep reinforcement learning". In: *Advances in neural information processing systems* 31 (2018).

[49] David Ha and Jürgen Schmidhuber. "World models". In: *arXiv preprint arXiv:1803.10122* (2018).

[50] Tomas Jakab, Ankush Gupta, Hakan Bilen, and Andrea Vedaldi. "Unsupervised learning of object landmarks through conditional image generation". In: *Advances in neural information processing systems* 31 (2018).

[51] Arun Mallya and Svetlana Lazebnik. "Packnet: Adding multiple tasks to a single network by iterative pruning". In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2018, pp. 7765–7773.

[52] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. "Representation learning with contrastive predictive coding". In: *arXiv preprint arXiv:1807.03748* (2018).

[53] David Silver, Thomas Hubert, Julian Schrittwieser, and Demis Hassabis. *AlphaZero: Shedding new light on chess, shogi, and Go*. en. 2018.

[54] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[55] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. "Solving rubik's cube with a robot hand". In: *arXiv preprint arXiv:1910.07113* (2019).

[56] Ankesh Anand, Evan Racah, Sherjil Ozair, Yoshua Bengio, and R Devon Côté Marc-Alexandre anHjelm. "Unsupervised state representation learning in atari". In: *Advances in neural information processing systems* 32 (2019).

[57] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Christy Debiak Przemysłaand Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. "Dota 2 with large scale deep reinforcement learning". In: *arXiv preprint arXiv:1912.06680* (2019).

[58] Tejas D Kulkarni, Ankush Gupta, Catalin Ionescu, Sebastian Borgeaud, Andrew Reynolds Malcolm anZisserman, and Volodymyr Mnih. "Unsupervised learning of object keypoints for perception and control". In: *Advances in neural information processing systems* 32 (2019).

[59] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and WermterStefan. "Continual lifelong learning with neural networks: A review". In: *Neural Networks* 113 (2019), pp. 54–71.

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Trevor Chanan Gregorand Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[61] Kate Rakelly, Aurick Zhou, Chelsea Finn, Sergey Levine, and Deirdre Quillen. "Efficient off-policy meta-reinforcement learning via probabilistic context variables". In: *International conference on machine learning*. PMLR. 2019, pp. 5331–5340.

[62] René Traoré, Hugo Caselles-Dupré, Timothée Lesort, Natalia Sun Te anDıaz-Rodrıguez, and David Filliat. "Continual reinforcement learning deployed in real-life using policy distillation and sim2real transfer". In: *arXiv preprint arXiv:1906.04452* (2019).

[63] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Junyoung Dudzik Andreand Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* 575.7782 (2019), pp. 350–354.

[64] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, and Charles Blundell. *Agent57: Outperforming the Atari Human Benchmark*. 2020.

[65] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: `https://www.wandb.com/`.

[66] Matthew Botvinick, Jane X Wang, Will Dabney, Kevin J Miller, and Zeb Kurth-Nelson. "Deep reinforcement learning and its neuroscientific implications". In: *Neuron* 107.4 (2020), pp. 603–616.

[67] Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. "Towards continual reinforcement learning: A review and perspectives". In: *arXiv preprint arXiv:2012.13490* (2020).

[68] Timothée Lesort, Vincenzo Lomonaco, Andrei Stoian, Davide Maltoni, and Natalia Filliat David anDıaz-Rodrıguez. "Continual learning for robotics: Definition, framework, learning strategies, opportunities and challenges". In: *Information fusion* 58 (2020), pp. 52–68.

[69] Vincenzo Lomonaco, Karan Desai, Eugenio Culurciello, and Davide Maltoni. "Continual reinforcement learning in 3d non-stationary environments". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 2020, pp. 248–249.

[70] Enrico Marchesini and Alessandro Farinelli. "Discrete Deep Reinforcement Learning for Mapless Navigation". In: (2020), pp. 10688–10694.

[71] Thomas M Moerland, Joost Broekens, and Catholijn M Jonker. "Model-based reinforcement learning: A survey". In: *arXiv preprint arXiv:2006.16712* (2020).

[72] Karl Pertsch, Youngwoon Lee, and Joseph J Lim. "Accelerating reinforcement learning with learned skill priors". In: *arXiv preprint arXiv:2010.11944* (2020).

[73] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Simon Sifre Laurenand Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. "Mastering atari, go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609.

[74] Zihao Zhang, Stefan Zohren, and Stephen Roberts. "Deep reinforcement learning for trading". In: *The Journal of Financial Data Science* 2.2 (2020), pp. 25–40.

[75] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, FigurnovMichael, Olaf Ronneberger, Kathryn Tunyasuvunakool, Augustin Bates Russ anŽıdek, Anna Potapenko, et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (2021), pp. 583–589.

[76] Taisuke Kobayashi and Wendyam Eric Lionel Ilboudo. "T-soft update of target network for deep reinforcement learning". In: *Neural Networks* 136 (2021), pp. 63–71.

[77] Taewook Nam, Shao-Hua Sun, Karl Pertsch, Sung Ju Hwang, and Joseph J Lim. "Skill-based Meta-Reinforcement Learning". In: *arXiv preprint arXiv:2204.11828* (2022).