POLITECNICO DI MILANO
Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

FAST REINFORCEMENT LEARNING USING DEEP STATE FEATURES

AI & R Lab
Laboratorio di Intelligenza Artificiale
e Robotica del Politecnico di Milano

Supervisor: Prof. Marcello Restelli
Co-supervisors: Dott. Carlo D'Eramo, Dott. Matteo Pirotta

Master's Thesis by:
Daniele Grattarola (Student ID 853101)

Academic Year 2016-2017

To. . .

Contents

List of Figures

List of Tables

List of Algorithms

Summary

Il sommario deve contenere 3 o 4 frasi tratte dall'introduzione di cui la prima inquadra l'area dove si svolge il lavoro (eventualmente la seconda inquadra la sottoarea più specifica del lavoro), la seconda o la terza frase dovrebbe iniziare con le parole "Lo scopo della tesi è ..." e infine la terza o quarta frase riassume brevemente l'attivitàï¿½ svolta, i risultati ottenuti ed eventuali valutazioni di questi.

NB: se il relatore effettivo è interno al Politecnico di Milano nel frontesizo si scrive Relatore, se vi è la collaborazione di un altro studioso lo si riporta come Correlatore come sopra. Nel caso il relatore effettivo sia esterno si scrive Relatore esterno e poi bisogna inserire anche il Relatore interno. Nel caso il relatore sia un ricercatore allora il suo Nome COGNOME dovràï¿½ essere preceduto da Ing. oppure Dott., a seconda dei casi.

Acknowledgments

I thank...

Chapter 1

Introduction

Chapter 2

Background

In this chapter we outline the theoretical framework which will be used in the following chapters. The approach proposed in this thesis draws equally from the fields of *deep learning* and *reinforcement learning*, in a hybrid setting usually called *deep reinforcement learning*.

In the following sections we give high level descriptions of these two fields, in order to introduce a theoretical background, a common notation, and a general view of some of the most important techniques in each area.

2.1 Deep Learning

Deep Learning (DL) is a branch of machine learning which aims to learn abstract representations of the input space by means of complex function approximators. Deep-learning methods are based on multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level [23].

Deep learning is at the heart of modern machine learning research, where deep models have revolutionized many fields like computer vision [21, 37], machine translation [43] and speech synthesis [39]. Generally, the most impressive results of deep learning have been achieved through the versatility of *neural networks*, which are universal function approximators well suited for hierarchical composition.

In this section we give a brief overview of the basic concepts behind *deep neural networks* and introduce some important ideas that will be used in later chapters of this thesis.

2.1.1 Artificial Neural Networks

Feed-forward Artificial Neural Networks (ANNs) [4] are universal function approximators inspired by the connected structure of neurons and synapses in biological brains.

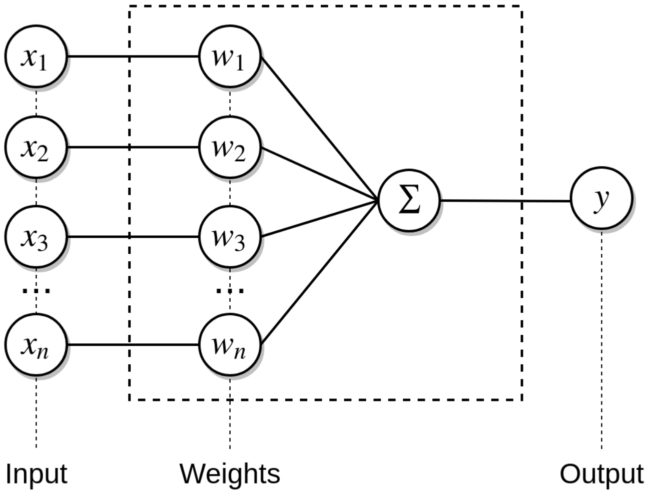


Figure 2.1: Graphical representation of the perceptron model.

ANNs are based on a fairly simple computational model called *perceptron* (Figure 2.1), which is a transformation of an n-space into a scalar value

$$z = \sum_{i=1}^{n} (w_i \cdot x_i) + b \tag{2.1}$$

where $x = (x_1, ..., x_n)$ is the n-dimensional input to the model, $w = (w_1, ..., w_n)$ is a set of weights associated to each component of the input and b is a bias term (in some notations the bias is embedded in the transformation by setting $x_0 = 1$ and $w_0 = b$).

In ANNs, the simple model of the perceptron is used to create a layered structure, in which each *hidden* layer is composed by a given number of perceptrons (called *neurons*) which (see Figure 2.2):

1. take as input the output of the previous layer;
2. are followed by a nonlinearity σ called the *activation function*;
3. output their value as a component of some m-dimensional space which is the input space of the following layer.

In simpler terms, each hidden layer computes an affine transformation of its input space:

$$z^{(i)} = W^{(i)} \cdot \sigma(z^{(i-1)}) + B^{(i)} \tag{2.2}$$

where $W^{(i)}$ is the composition of the weights associated to each neuron in the layer and B is the equivalent composition of the biases.

The processing of the input space performed by the succession of layers which compose an ANN is equivalent to the composition of multiple non-linear transformations, which results in the production of an output vector on the co-domain of the target function.

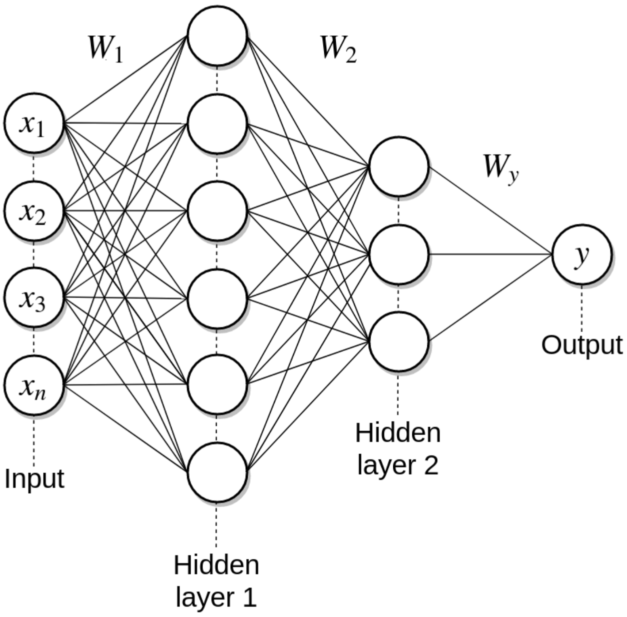


Figure 2.2: A fully-connected neural network with two hidden layers.

2.1.2 Backpropagation

Training ANNs is a parametric learning problem, where a *loss* function is minimized starting from a collection of *training samples* collected by the real process which is being approximated. In parametric learning the goal is to find the optimal parameters of a mathematical model, such that the expected error made by the model on the training samples is minimized. In ANNs, the parameters which are optimized are the weight matrices $W^{(i)}$ and biases $B^{(i)}$ associated to each hidden layer of the network.

In the simple perceptron model, which basically computes a linear transformation of the input, the optimal parameters are learned from the training set according to the following *update rule*:

$$w_i^{new} = w_i^{old} - \eta(\hat{y} - y)x_i, \forall i = (1, ..., n) \tag{2.3}$$

where $\hat{y}$ is the output of the perceptron, y is the real target from the training set, x_i is the i-th component of the input, and η is a scaling factor called the *learning rate* which regulates how much the weights are allowed to change in a single update. Successive applications of the update rule for the perceptron guarantee convergence to an optimum if and only if the approximated function is linear (in the case of regression) or the problem is linearly separable (in the case of classification) as shown in Figure 2.3.

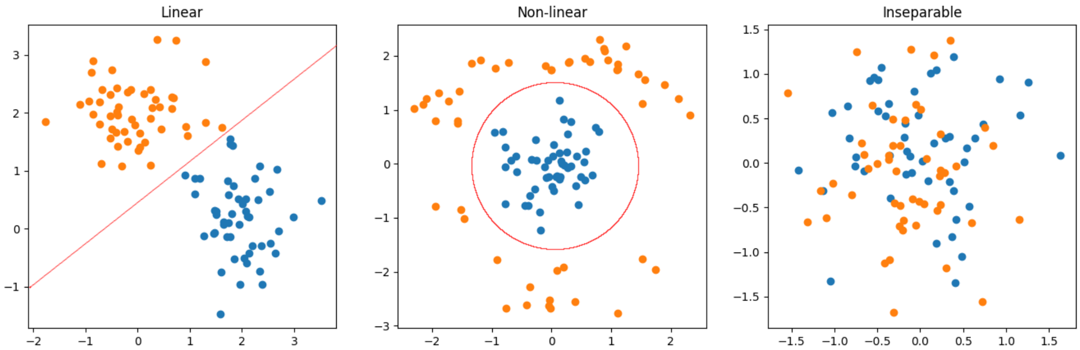


Figure 2.3: Different classification problems on a two-dimensional space, respectively linearly, non-linearly and non separable. The perceptron would be able to converge and correctly classify the points only in the first setting.

The simple update rule of the perceptron, however, cannot be used to train an ANN with multiple layers because the true outputs of the hidden layers are not known a priori. To solve this issue, it is sufficient to notice that the function computed by each layer of a network is nonlinear, but differentiable with respect to the layer's input (i.e. it is linear in the weights). This simple fact allows to compute the partial derivative of the loss function for each weight matrix in the network to, in a sense, impute the error committed on a training sample proportionally across neurons. The error is therefore propagated backwards (hence the name *backpropagation*) to update all weights in a similar fashion to the perceptron update. The gradient of the loss function is then used to change the value of the weights, with a technique called *gradient descent* which consists in the following update rule:

$$W_i^{new} = W_i^{old} - \eta \frac{\partial L(y, \hat{y})}{\partial W_i^{old}} \tag{2.4}$$

where L is any differentiable function of the target and predicted values that quantifies the error made by the model on the training samples. The term *gradient descent* is due to the fact that the weights are updated in the opposite direction of the loss gradient, moving towards a set of parameters for which the loss is lower.

Notice that traditional gradient descent optimizes the loss function over all the training set at once, performing a single update of the parameters. This approach, however, can be computationally expensive when the training set is big; a more common approach

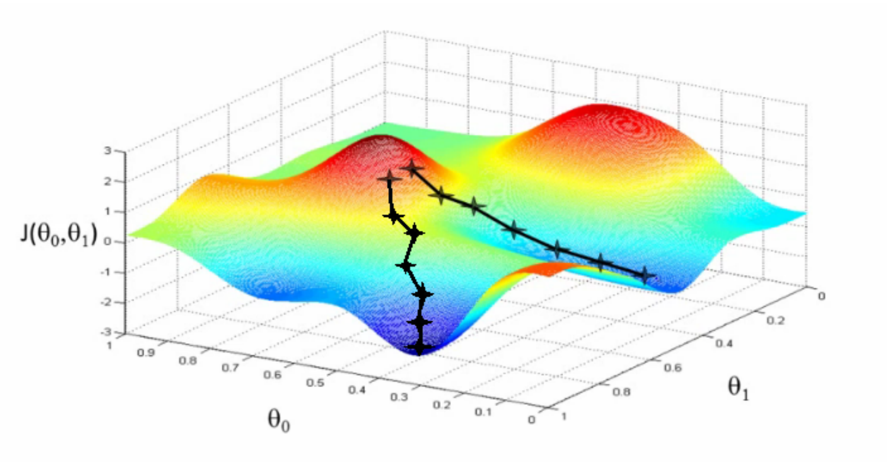


Figure 2.4: Visualization of SGD on a space of two parameters.

is to use *stochastic gradient descent* (SGD) [4], which instead performs sequential parameters updates using small subsets of the training samples (called *batches*). As the number of samples in a batch decreases, the variance of the updates increases, because the error committed by the model on a single sample can have more impact on the gradient step. This can cause the optimization algorithm to *miss* a good local optima due to excessively big steps, but at the same time could help leaving a poor local minima in which the optimization is stuck. The same applies to the learning rate, which is the other important factor in controlling the size of the gradient step: if the learning rate is too big, SGD can *overshoot* local minima and fail to converge, but at the same time it may take longer to find the optimum if the learning rate is too small (Figure 2.5).

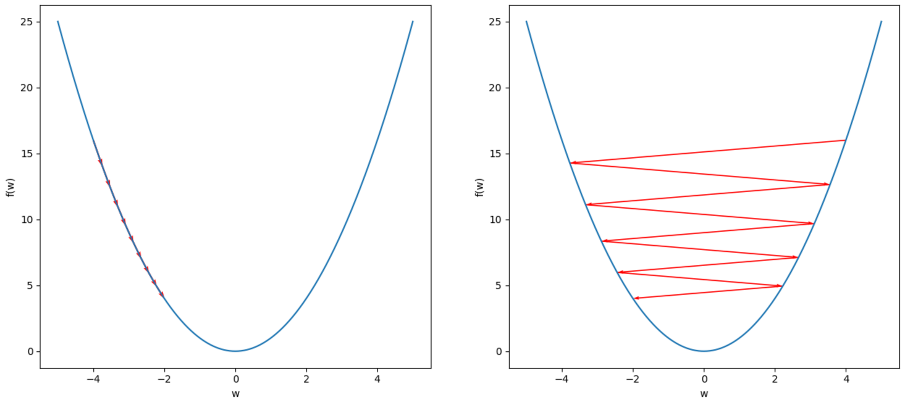


Figure 2.5: Effect of the learning rate on SGD updates. Too small (left) may take longer to converge, too big (right) may overshoot the optimum and even diverge.

In order to improve the accuracy and speed of SGD, some additional tweaks are usually added to the optimization algorithm. Among these, we find the addition of a *momentum* term to the update step of SGD, in order to avoid oscillating in irrelevant directions by incorporating a fraction of the previous update term in the current one:

$$W_i^{(j+1)} = W_i^{(j)} - \gamma\eta \frac{\partial L(y^{(j-1)}, \hat{y}^{(j-1)})}{\partial W_i^{(j-1)}} - \eta \frac{\partial L(y^{(j)}, \hat{y}^{(j)})}{\partial W_i^{(j)}} \tag{2.5}$$

where (j) is the number of updates that have occurred so far. In this approach, momentum has the same meaning as in physics, like when a body falling down a slope tends to preserve part of its previous velocity when subjected to a force. Other techniques to improve convergence include the use of an adaptive learning rate based on the previous gradients computed for the weights (namely the *Adagrad* [8] and *Adadelta* [44] optimization algorithms), and a similar approach which uses an adaptive momentum term (called *Adam* [20]).

2.1.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of ANN inspired by the visual cortex in animal brains, and have been widely used in recent literature to reach state-of-the-art results in fields like computer vision, machine translation, and, as we will see in later sections, reinforcement learning.

CNNs exploit spatially-local correlations in the neurons of adjacent layers by using a *receptive field*, a set of weights which is used to transform a local subset of the input neurons of a layer. The receptive field is applied as a *filter* over different locations of the input, in a fashion that resembles how a signal is *strided* across the other during convolution. The result of this operation is a nonlinear transformation of the input space into a new space (of compatible dimensions) which preserves the spatial information encoded in the input (e.g. form the $n \times m$ pixels of a grayscale image to a $j \times k$ matrix that represents subgroups of pixels in which there is an edge).

While standard ANNs have a *fully connected* (sometimes also called *dense*) structure, with each neuron of a layer connected to each neuron of the previous and following layer, in CNNs the weights are associated to a filter and *shared* across all neurons of a layer, as shown in Figure 2.6. This *weights sharing* has the double advantage of greatly reducing the number of parameters that must be updated during training, and of forcing the network to learn general abstractions of the input that can be applied to any subset of neurons covered by the filter.

In general, the application of a filter is not limited to one per layer and it is customary to have more than one filter applied to the same input in parallel, to create a set of independent abstractions called *feature maps* (also referred to as *channels*, to recall the terminology of RGB images for which a 3-channel representation is used for red, green,

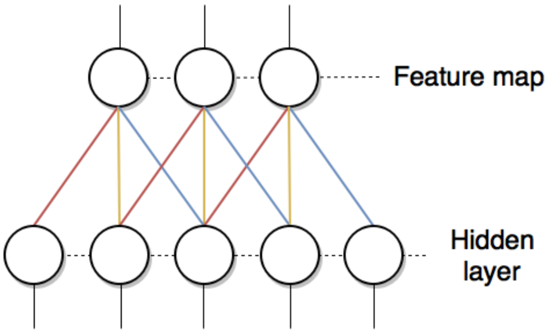


Figure 2.6: Simple representation of shared weights in a 1D CNN. Each neuron in the second layer applies the same receptive field of three weights to three adjacent neurons of the previous layer. The filter is applied with a stride of one element to produce the feature map.

and blue). In this case, there will be a set of shared weights for each filter. When a set of feature maps is given as input to a convolutional layer, a multidimensional filter is strided simultaneously across all channels.

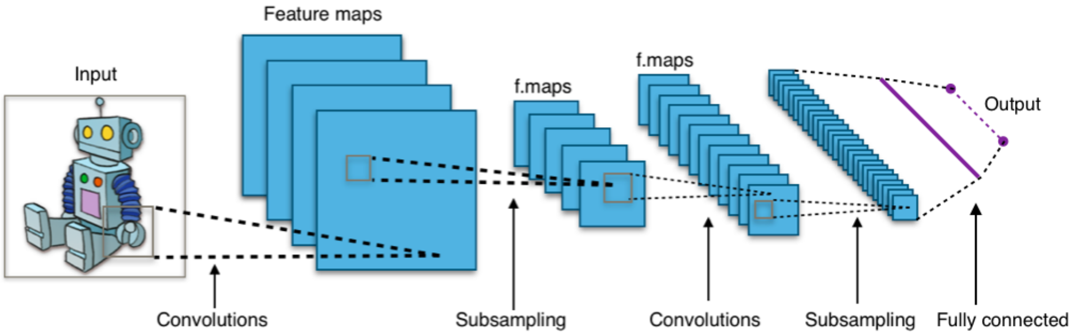
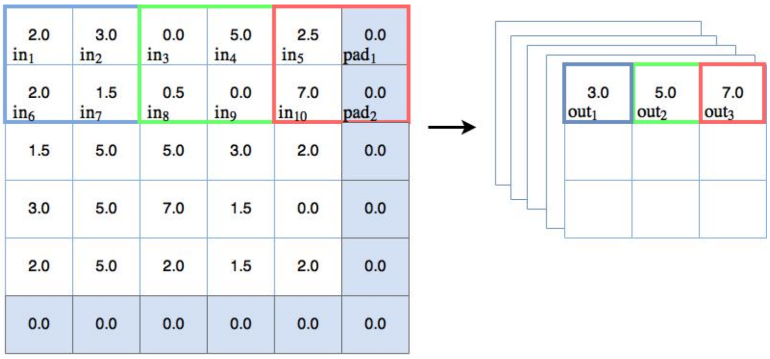


Figure 2.7: Typical structure of a deep convolutional neural network for image processing, with two convolutional hidden layers and a dense section at the end (for classification or regression).

At the same time, while it may be useful to have parallel abstractions of the input space (which effectively enlarges the output space of the layers), it is also necessary to force a reduction of the input in order to learn useful representations. For this reason, convolutional layers in CNNs are often paired with *pooling layers* which reduce the dimensionality of their input according to some criteria applied to subregions of the input neurons (e.g. for each two by two square of input neurons, keep only the maximum activation value as shown in Figure 2.8).

Finally, typical applications of CNNs in the literature use mixed architectures composed of both convolutional and fully connected layers. In tasks like image classification [35, 37], convolutional layers are used to extract significant features directly from the images, and dense layers are used as a final classification model; the training in this case



2.0 in_1	3.0 in_2	0.0 in_3	5.0 in_4	2.5 in_5	0.0 pad_1
2.0 in_6	1.5 in_7	0.5 in_8	0.0 in_9	7.0 in_{10}	0.0 pad_2
1.5	5.0	5.0	3.0	2.0	0.0
3.0	5.0	7.0	1.5	0.0	0.0
2.0	5.0	2.0	1.5	2.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

3.0 out_1	5.0 out_2	7.0 out_3

Figure 2.8: Example of max pooling, where only the highest activation value in the pooling window is kept.

is done in an end-to-end fashion, with the classification error being propagated across all layers to *fine-tune* all weights and filters to the specific problem.

2.1.4 Autoencoders

Autoencoders (AE) are a type of ANN which is used to learn a sparse and compressed representation of the input space, by sequentially compressing and reconstructing the inputs under some sparsity constraint.

The typical structure of an AE is split into two sections: an *encoder* and a *decoder* (Figure 2.9). In the classic architecture of autoencoders these two components are exact mirrors of one another, but in general the only constraint that is needed to define an AE is that the dimensionality of the input be the same as the dimensionality of the output. In general, however, the last layer of the encoder should output a reduced representation of the input which contains enough information for the decoder to invert the transformation.

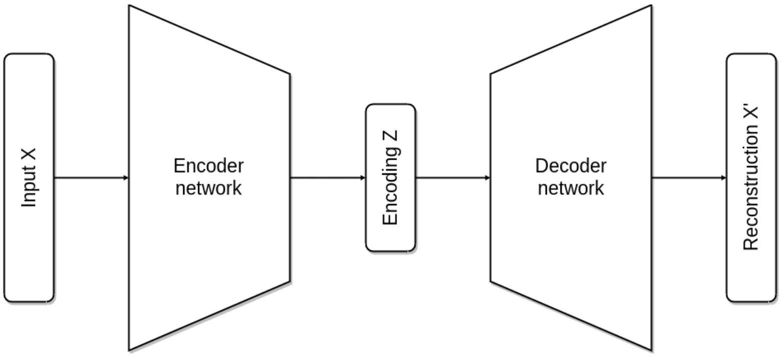


Figure 2.9: Schematic view of an autoencoder, with the two main components represented separately.

The training of an AE is done in an unsupervised fashion, with no explicit target required as the network is simply trained to predict its input. Moreover, a strong regularization constraint is often imposed on the innermost layer to ensure that the learned representation is as abstract as possible (typically the $L1$ norm of the activations is minimized as additional term to the loss, to enforce sparsity).

Autoencoders can be especially effective in extracting meaningful features from the input space, without tailoring the features to a specific problem like in the end-to-end image classification example of Section 2.1.3 [9]. An example of this is the extraction of features from images, where convolutional layers are used in the encoder to obtain an abstract description of the image's structure. In this case, the decoder uses convolutional layers to transform subregions of the representation, but the expansion of the compressed feature space is delegated to *upscaling layers* (the opposite of pooling layers) [25]. This approach in building the decoder, however, can sometimes cause blurry or inaccurate reconstructions due to the upscaling operation which simply replicates information rather than transforming it (like pooling layers do). Because of this, a more sophisticated technique has been developed recently which allows to build purely convolutional autoencoders, without the need of upscaling layers in the decoder. The layers used in this approach are called *deconvolutional*[1] and are thoroughly presented in [45]. For the purpose of this thesis it suffices to notice that image reconstruction with this type of layer is incredibly more accurate, down to pixel-level accuracy.

2.2 Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning which studies how to optimize the behavior of an agent in an environment, in order to maximize the cumulative sum of a scalar signal called *reward* in a setting of *sequential decision making.* RL has its roots in optimization and control theory but, because of the generality of its characteristic techniques, it has been applied to a variety of scientific fields where the concept of *optimal behavior in an environment* can be applied (examples include game theory, multi-agent systems and economy). The core aspect of reinforcement learning problems is to represent the setting of an agent performing decisions in an environment, which is in turn affected by the decisions; a scalar reward signal represents a time-discrete indicator of the agent's performance. This kind of setting is inspired to the natural behavior of animals in their habitat, and the techniques used in reinforcement learning are well suitable to describe, at least partially, the complexity of living beings.

In this section we introduce the basic setting of RL and go over a brief selection of the main techniques used to solve RL problems.

[1] Or *transposed convolutions.*

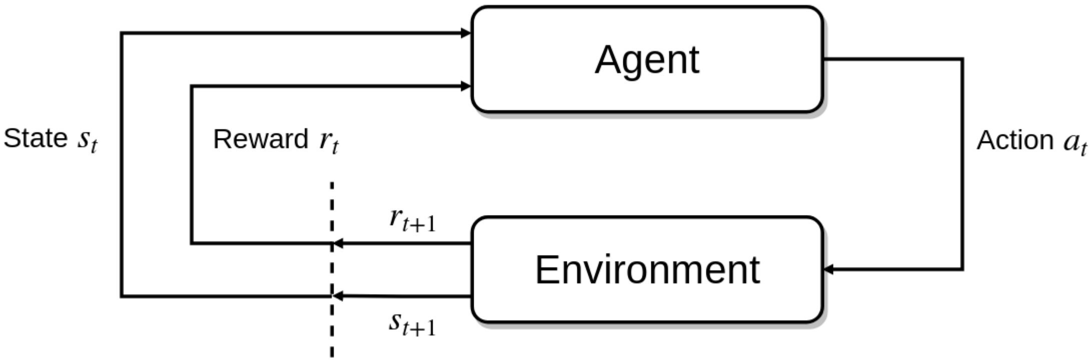


Figure 2.10: The reinforcement learning setting, with the agent performing actions on the environment and in turn observing the state and reward.

2.2.1 Markov Decision Processes

Markov Decision Processes (MDPs) are discrete-time, stochastic control processes, that can be used to describe the interaction of an *agent* with an *environment* (Figure 2.10).

Formally, MDPs are defined as 7-tuples $< S, S^T, A, P, R, \gamma, \mu >$, where:

- S is the set of observable states of the environment. When the set if observable states coincides with the true set of states of the environment, the MDP is said to be *fully observable*. We will only deal with fully observable MDPs without considering the case of *partially observable* MDPs.

- $S^T \subseteq S$ is the set of *terminal states* of the environment, meaning those states in which the interaction between the agent and the environment ends. The sequence of events that occur from when the agent observes an initial state until it reaches a terminal state is usually called *episode*.

- A is the set of actions that the agent can execute in the environment.

- $P : S \times A \times S \to [0, 1]$ is a *state transition function* which, given two states $s, s' \in S$ and an action $a \in A$, represents the probability of the agent going to state s' by executing a in s.

- $R : S \times A \to \mathbb{R}$ is a *reward function* which represents the reward that the agent collects by executing an action in a state.

- $\gamma \in [0, 1]$ is a *discount factor* which is used to weight the importance of rewards during time: $\gamma = 0$ means that only the immediate reward is considered, $\gamma = 1$ means that all rewards have the same importance.

- $\mu : S \to [0, 1]$ is a probability distribution over S which models the probability of starting an episode in a given state.

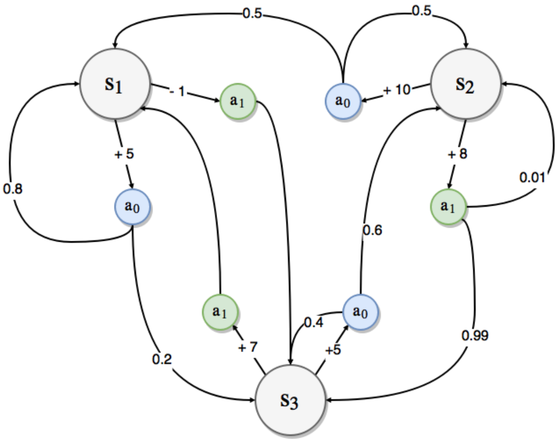


Figure 2.11: Graph representation of an MDP. Each gray node represents a state, each arc leaving a state is associated to an action and its corresponding reward, an the action is in turn associated to the following state; note that actions may have probability distributions over the outgoing arcs.

Episodes are usually represented as sequences of tuples

$$[(s_0, a_0, r_1, s_1), ..., (s_{n-1}, a_{n-1}, r_n, s_n)]$$

called *trajectories*, where $s_n \in S^T$, $s_0 \sim \mu$, and $(s_i, a_i, r_{i+1}, s_{i+1})$ represents a transition of the agent to state s_{i+1} by taking action a_i in s_i and collecting a reward r_{i+1}.

In MDPs the modeled environment must satisfy the *Markov property*, meaning that the reward and transition functions of the environment must only depend on the current state and action, and not on the past state-action trajectory of the agent. In other words, an environment is said to satisfy the Markov property when its one-step dynamics allow to predict the next state and reward given only the current state and action.

Policy

The behavior of the agent in an MDP can be defined as a probability distribution $\pi : S \times A \to [0, 1]$ called a *policy*, which given $s \in S, a \in A$, represents the probability of selecting a as next action from s. An agent which uses this probability distribution to select its next action when in a given state is said to be *following* the policy.

A common problem when defining policies is the *exploration-exploitation* dilemma. An agent following a policy may end up observing the same trajectories in all episodes (e.g. when following a deterministic policy in a deterministic MDP), but there may be cases in which a better behavior could be had if the agent *explored* other states instead of simply *exploiting* its knowledge. It is therefore common to add a probabilistic element to policies (irrespectively of their determinism), in order to explicitly control the exploration degree of the agent. Common techniques to control the *exploration-exploitation* tradeoff are:

- ε-greedy policies: actions are selected using a given policy with probability $1-\varepsilon$, and randomly the rest of the time;

- softmax action selection: improves on ε-greedy policies by reducing the number of times a suboptimal action is randomly selected. To do so, a probability distribution (commonly a *Boltzmann distribution*) dependent on the expected return from the successor states (something called the *value* of the states, which is introduced in the next Subsection) is used.

Value Functions

Starting from the concept of policy, we can now introduce a function that evaluates how good it is for an agent following a policy π to be in a given state. This evaluation is expressed in terms of the expected return, i.e. the expected discounted sum of future rewards collected by an agent starting from a state while following π, and the function that computes it is called the *state-value function for policy* π (or, more commonly, just *value function*).

Formally, the state-value function associated to a policy π is a function $V^\pi : S \to \mathbb{R}$ defined as:

$$V^\pi(s) = E_\pi[R_t | s_t = s] \tag{2.6}$$

$$= E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s] \tag{2.7}$$

where $E_\pi[\cdot]$ is the expected value given that the agent follows policy π, and t is any time step of an episode $[s_0, ..., s_t, ..., s_n]$ where $s_t \in S, \forall t = 0, ..., n$.

Similarly, we can also introduce a function that evaluates the goodness of taking a specific action in a given state, namely the expected reward obtained by taking an action $a \in A$ in a state $s \in S$ and then following policy π. We call this function the *action-value function for policy* π denoted $Q^\pi : S \times A \to \mathbb{R}$, and defined as:

$$Q^\pi(s, a) = E_\pi[R_t | s_t = s, a_t = a] \tag{2.8}$$

$$= E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a] \tag{2.9}$$

The majority of reinforcement learning algorithms is based on computing (or estimating) value functions, which can then be used to control the behavior of the agent. We also note a fundamental property of value functions, which satisfy particular recursive relationships like the following *Bellman equation for V^π*:

$$\begin{aligned}
V^\pi(s) &= E_\pi[R_t|s_t = s] \\
&= E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s] \\
&= E_\pi[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_t = s] \qquad (2.10)\\
&= \sum_{a\in A} \pi(s,a) \sum_{s'\in S} P(s,a,s')\Big[R(s,a) + \\
&\quad + \gamma E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_{t+1} = s']\Big] \qquad (2.11)\\
&= \sum_{a\in A} \pi(s,a) \sum_{s'\in S} P(s,a,s')[R(s,a) + \gamma V^\pi(s')] \qquad (2.12)
\end{aligned}$$

Intuitively, Equation (2.12) decomposes the state-value function as the sum of the immediate reward collected from a state s to a successor state s', and the value of s' itself; by considering the transition model of the MDP and the policy being followed, we see that the Bellman equation simply averages the expected return over all the possible (s, a, r, s') transitions, by taking into account the probability that these transitions occur.

2.2.2 Optimal Value Functions

In general terms, *solving* a reinforcement learning task consists in finding a policy that yields a sufficiently high expected return. In the case of MDPs with discrete state and actions sets[2], it is possible to define the concept of *optimal policy* as the policy which maximizes the expected return collected by the agent in an episode.

We start by noticing that state-value functions define a partial ordering over policies as follows:

$$\pi \geq \pi' \iff V^\pi(s) \geq V^{\pi'}(s), \forall s \in S$$

From this, the *optimal policy* π^* of an MDP is a policy which is better or equal than all other policies in the policy space. It has also been proven that among all optimal policies for an MDP, there is always a deterministic one (see Section 2.2.3).

The state-value function associated to π^* is called the *optimal state-value function*, denoted V^* and defined as:

$$V^*(s) = \max_\pi V^\pi(s), \forall s \in S \qquad (2.13)$$

[2] We make this clarification for formality, but we do not expand the details further in this work. Refer to [36] for more details on the subjet of non-finite MDPs.

As we did when introducing the value functions, given an optimal policy for the MDP it is also possible to define the *optimal action-value function* denoted Q^*:

$$\begin{align} Q^*(s,a) &= \max_{\pi} Q^{\pi}(s,a) \tag{2.14} \\ &= E[r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a] \tag{2.15} \end{align}$$

Notice that Equation (2.15) in this definition highlights the relation between Q^* and V^*.

Since V^* and Q^* are value functions of an MDP, they must satisfy the same type of recursive relations that we described in Equation (2.12), in this case called the *Bellman optimality equations.* The Bellman optimality equation for V^* expresses the fact that the value of a state associated to an optimal policy must be the expected return of the best action that the agent can take in that state:

$$\begin{align} V^*(s) &= \max_{a} Q^*(s,a) \tag{2.16} \\ &= \max_{a} E_{\pi^*}[R_t | s_t = s, a_t = a] \tag{2.17} \\ &= \max_{a} E_{\pi^*}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a] \tag{2.18} \\ &= \max_{a} E_{\pi^*}[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_t + k + 2 | s_t = s, a_t = a] \tag{2.19} \\ &= \max_{a} E_{\pi^*}[r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a] \tag{2.20} \\ &= \max_{a} \sum_{s' \in S} P(s,a,s')[R(s,a) + \gamma V^*(s')] \tag{2.21} \end{align}$$

The Bellman optimality equation for Q^* is again obtained from the definition as:

$$\begin{align} Q^*(s,a) &= E[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') || s_t = s, a_t = a] \tag{2.22} \\ &= \sum_{s'} P(s,a,s')[R(s,a) + \gamma \max_{a'} Q^*(s', a')] \tag{2.23} \end{align}$$

Notice that both Bellman optimality equations have a unique solution independent of the policy. If the dynamics of the environment (R and P) are fully known, it is possible to solve the system of equations associated to the value functions (i.e. one equation for each state in S) and get an exact value for V^* and Q^* in each state.

2.2.3 Value-based optimization

One of main algorithm classes for solving reinforcement learning problems is based on searching an optimal policy for the MDP by computing either of the optimal value functions, and then deriving a policy based on them. From V^* or Q^*, it is easy to determine an optimal, deterministic policy:

- Given V^*, for each state $s \in S$ there will be an action (or actions) which maximizes the Bellman optimality equation (2.16). Any policy that assigns positive probability to only this action is an optimal policy. This approach therefore consists in performing a one-step forward search on the state space to determine the best action from the current state.

- Given Q^*, the optimal policy is that which assigns positive probability to the action which maximizes $Q^*(s, a)$; this approach exploits the intrinsic property of the action-value function of representing the *quality* of actions, without performing the one-step search on the successor states.

In the following sections we will describe some of the most important value-based approaches to RL, which will be useful in the following chapters of this thesis. We will not deal with equally popular methods like *policy gradient* or *actor-critic* approaches, even though they have been successfully applied in conjunction with DL to solve complex environments (see Chapter 3).

2.2.4 Dynamic Programming

The use of dynamic programming (DP) techniques to solve reinforcement learning problems is based on recursively applying some form of the Bellman equation, starting from an initial policy π until convergence to π^*. In this class of algorithms, we identify two main approaches: *policy iteration* and *value iteration*.

Policy iteration

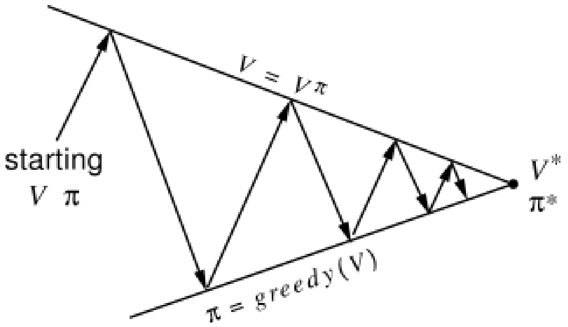


Figure 2.12: Classical representation of the policy iteration algorithm, which highlights the relation between policies and their associated value functions. Each pair of arrows starting from a policy and ending on a greedy policy based on the value function is a step of the algorithm.

Policy iteration is based of the following theorem:

Theorem 1 (Policy improvement theorem) *Let π and π' be a pair of deterministic policies such that*

$$Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s), \forall s \in S$$

Then, $\pi' \geq \pi$, i.e.

$$V^{\pi'}(s) \geq V^{\pi}(s), \forall s \in S$$

This approach works by iteratively computing the value function associated to the current policy, and then improving that policy by making it act greedily with respect to the value function (as shown in Figure 2.12), such that:

$$\pi'(s) = \arg\max_{a \in A} Q^{\pi}(s, a) \tag{2.24}$$

For Theorem 1, the expected return of the policy is thus improved because:

$$Q^{\pi}(s, \pi'(s)) = \max_{a \in A} Q^{\pi}(s, a) \geq Q^{\pi}(s, \pi(s)) = V^{\pi}(s) \tag{2.25}$$

This continuous improvement is applied until Inequality (2.25) becomes an equality, i.e. until the improved policy satisfies the Bellman optimality equation (2.16). Since the algorithm gives no assurances on the number of updates required for convergence, some stopping conditions are usually introduced to end the process when the new value function does not change substantially after the update (*ε-convergence*) or a certain threshold number of iterations has been reached.

Value iteration

Starting from a similar idea, the *value iteration* approach computes the value function associated to an initial policy, but then applies a contraction operator which iterates over sequentially better value functions without actually computing the associated greedy policy. The contraction operator which ensures convergence is the *Bellman optimality backup*:

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} P(s, a, s')[R(s, a) + \gamma V(s')] \tag{2.26}$$

As with policy iteration, convergence is ensured without guarantees on the number of steps, and therefore it usual to terminate the iteration according to some stopping condition.

2.2.5 Monte Carlo Methods

Dynamic programming approaches exploit the exact solution of a value function which can be computed starting from a policy, but in general this requires to have a perfect knowledge of the environment's dynamics and may also not be tractable on sufficiently complex MDPs.

Monte Carlo (MC) methods are a way of solving reinforcement learning problems by only using *experience*, i.e. a collection of *sample trajectories* from an actual interaction of an agent with the environment. This is often referred to as a *model-free* approach because, while the environment (or a simulation thereof) is still required to observe the sample trajectories, it is not necessary to have an exact knowledge of the transition model and reward function of the MDP.

Despite the differences with dynamic programming, this approach is still based on the same two-step process of policy iteration (evaluation and improvement). To estimate the value of a state $V^\pi(s)$ under a policy π with Monte Carlo methods, it is sufficient to consider a set of episodes collected under π: the value of the state s will be computed as the average of the returns collected following a *visit* of the agent to s, for all occurrences of s in the collection[3].

This same approach can be also used to estimate the action-value function, simply by considering the occurrence of state-action pairs in the collected experience rather than states only.

Finally, the policy is improved by computing its greedy variation (2.24) with respect to the estimated value functions and the process is iteratively repeated until convergence, with a new set of trajectories collected under each new policy.

2.2.6 Temporal Difference Learning

Temporal Difference (TD) learning is an approach to RL which uses concepts from both dynamic programming and Monte Carlo techniques. TD is a *model-free* approach which uses experience (like in MC) to update an estimate of the value functions by using a previous estimate (like in DP). Like MC, TD estimation uses the rewards following a visit to a state to compute the value functions, but with two core differences:

1. Instead of the average of all rewards following the visit, a single time step is considered (this is true for the simplest TD approach, but note that in general an arbitrary number of steps can be used; the more steps are considered, the more the estimate is similar to the MC estimate).

2. Estimates of the value functions are updated by using in part an already computed estimate. For this reason, this approach is called a *bootstrapping* method (like DP). Specifically, the iterative update step for the value function is:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.27}$$

[3] Note that a variation of this algorithm exists, which only considers the average returns following the *first* visit to a state in each episode.

In general, TD methods have several advantages over MC as they allow for an *online* (i.e. they don't require full episode trajectories to work), bootstrapped, *model-free* estimate, which is more suitable for problems with long or even infinite time horizons. Moreover, TD is less susceptible to errors or exploratory actions and in general provides a more stable learning. It must be noted, however, that both TD and MC are guaranteed to converge given a sufficiently large amount of experience, and that there are problems for which either of the two can converge faster to the solution.

We will now present the two principal control algorithms in the TD family, one said to be *on-policy* (i.e. methods that attempt to evaluate and improve the same policy that they use to make decisions) and the other *off-policy* (i.e. methods with no relations between the estimated policy and the policy used to collect experience).

SARSA

As usual in *on-policy* approaches, *SARSA*[4] works by estimating the value $Q^{\pi}(s,a)$ for a current behavior policy π which is used to collect sample transitions from the environment. The policy is updated towards greediness with respect to the estimated action-value after each transition (s,a,r,s',a'), and the action-value is in turn updated step-wise with the following rule:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)] \tag{2.28}$$

The training procedure of SARSA can be summarized with Algorithm 1.

Convergence of the SARSA method is guaranteed by the dependence of π on the action-value function, as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (e.g. a time-dependent ε-greedy policy with $\varepsilon = 1/t$).

Q-learning

Introduced by Watkins in 1992 [], and considered by Sutton and Barto [36] as one of the most important breakthroughs in reinforcement learning, *Q-learning* is an *off-policy* temporal difference method that approximates the optimal action-value function independently of the policy being used to collect experiences. This simple, yet powerful idea guarantees convergence to the optimal value function as long as all state-action pairs are continuously visited (i.e. updated) during training.

The update rule for the TD step in Q-learning is the following:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)] \tag{2.29}$$

[4] Originally called *on-line Q-learning* by the creators; this alternative acronym was proposed by Richard Sutton and reported in a footnote of the original paper in reference to the *State, Action, Reward, next State, next Action* tuples which are used for prediction.

Algorithm 1 SARSA

```
Initialize $Q(s, a)$ arbitrarily
Initialize $\pi$ as some function of $Q$ (e.g. greedy)
repeat
  Initialize $s$
  Choose $a$ from $s$ using $\pi$
  repeat
    Take action $a$, observe $r$, $s'$
    Choose $a'$ from $s'$ using $\pi$
    Update $Q(s, a)$ using Rule (2.28)
    if $\pi$ is time-variant then
      Update $\pi$ towards greediness
    end if
    $s \leftarrow s'$; $a \leftarrow a'$
  until $s$ is terminal or Q did not change
until training ended or Q did not change
```

As we did for SARSA, an algorithmic description of the Q-learning algorithm is provided in Algorithm 2.

Algorithm 2 Q-Learning

```
Initialize $Q(s, a)$ and $\pi$ arbitrarily
repeat
  Initialize $s$
  repeat
    Choose $a$ from $s'$ using $\pi$
    Take action $a$, observe $r$, $s'$
    Update $Q(s, a)$ using Rule (2.29)
    $s \leftarrow s'$
  until $s$ is terminal or Q did not change
until training ended or Q did not change
```

2.2.7 Fitted Q-Iteration

Having introduced a more classic set of traditional RL algorithms in the previous sections, we now present a more modern algorithm to solve MDPs with the use of supervised learning algorithms to estimate the value functions.

As we will see later in this thesis, the general idea of estimating the value functions with a supervised model is not an uncommon approach, and it has been often used in the literature to solve a wide range of environments with high-dimensional state-action spaces. This is especially useful in problems for which the closed form solutions of DP, or the guarantees of visiting all state-action pairs required for MC and TD are not feasible.

Here, we choose the *Fitted Q-Iteration* (FQI) [10] algorithm as representative for this whole class, because it will be used in later sections of this thesis as a key component of the presented methodology. FQI is an *off-line*, *off-policy*, *model-free*, *value-based* reinforcement learning algorithm which computes an approximation of the optimal policy from a set of four-tuples (s, a, r, s') collected by an agent under a policy π. This approach is usually referred to as *batch mode* reinforcement learning, because the complete amount of learning experience is fixed and given a priori.

The core idea behind the algorithm is to produce a sequence of approximations of Q^π, where each approximation is associated to one step of the *value-iteration* algorithm seen in Section 2.2.4, and computed using the previous approximation as part of the target for the supervised learning problem. The process is described in Algorithm 3.

Algorithm 3 Fitted Q-Iteration

```
Given: a set $F$ of four-tuples $(s \in S, a \in A, r \in \mathbb{R}, s' \in S)$ collected with some policy $\pi$; a regression algorithm;
$N \leftarrow 0$
Let $\hat{Q}_N$ be a function equal to 0 everywhere on $S \times A$
repeat
  $N \leftarrow N + 1$
  $TS \leftarrow ((x_i, y_i), i = 0, \ldots, |F|)$ such that $\forall (s_i, a_i, r_i, s'_i) \in F$:
    $x_i = (s_i, a_i)$
    $y_i = r_i + \gamma \max_{a \in A} \hat{Q}_{N-1}(s'_i, a)$
  Use the regression algorithm to induce $\hat{Q}_N(s, a)$ from $TS$
until stopping condition is met
```

Note that at the first iteration of the algorithm the action-value function is initialized as a 0 constant, and therefore the first approximation done by the algorithm is that of the reward function. Subsequent iterations use the previously estimated function to compute the target of a new supervised learning problem, and therefore each step is independent from the previous one, except for the information of the environment stored in the computed approximation.

A more practical description on how to apply this algorithm to a real problem will be detailed in later sections of this thesis. For now, we limit this section to a more abstract definition of the algorithm and we do not expand further on the implementation details.

2.3 Additional Formalism

In this section we briefly introduce some additional algorithms and concepts which will be used in later chapters as secondary components of our approach.

2.3.1 Decision Trees

Decision trees are a non-parametric supervised learning method for classification and regression. A decision tree is a tree structure defined over a domain (*attributes*) and co-domain (*labels*), in which each internal node represents a boolean test on an attribute and each leaf node represents a label. Given a set of attributes for a data point, the tests are applied to the attributes starting from the root until a leaf is reached, and the corresponding label is output by the model.

To learn a decision tree, the input space is partitioned recursively by splitting a subset of the space according to a binary condition, in a greedy procedure called *Top-Down Induction of Decision Trees* (TDIDT) based on *recursive binary partitioning* (see Figure 2.13). This recursive procedure is iterated over the input space until all training samples belonging to a partition have the same label, or splitting the domain further would not add information to the model. At each step, the attribute which best splits the training samples is selected for splitting, where the quality of the split is determined according to different criteria such as:

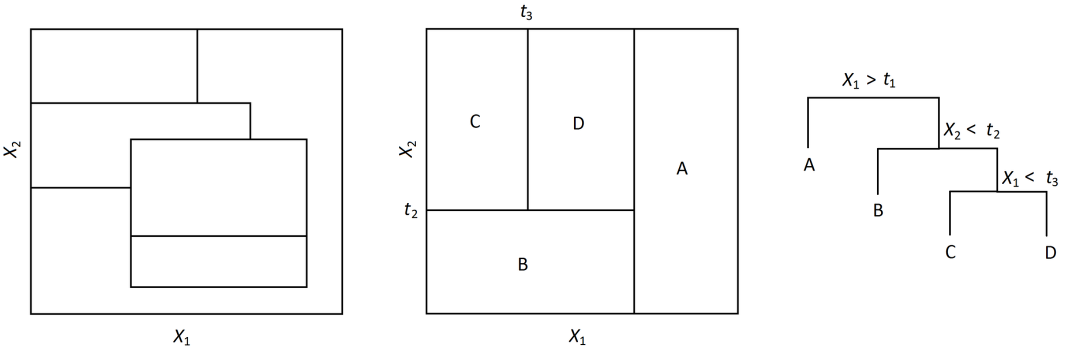


Figure 2.13: A 2D space partitioned with a random splitting algorithm (left) and recursive splitting (center); the recursively partitioned feature space can then be converted to a decision tree (right).

- *gini impurity*, the probability of incorrectly labeling a random training sample, if it was randomly labeled according to the distribution of labels in the partition. Attributes with low Gini impurity are selected with a higher priority. The measure is defined for a set as:

$$I_G(S) = \sum_{i=1}^{K} p_i(1 - p_i) \tag{2.30}$$

where S is a set of samples with labels $\{1, ..., K\}$ and p_i is the fraction of samples with label i in the set;

- *information gain*, the expected change in information entropy resulting form the split. To build a tree, the information gain of each possible split is computed and the most informative split is selected. The process is then repeated iteratively until the tree is complete. The measure is defined as the difference of entropy between a father node and a weighted sum of the children's entropy, where entropy is:

$$H(S) = -\sum_{i=1}^{K} p_i \log p_i \tag{2.31}$$

and the information gain is formalized as:

$$I_I(S, a) = H(S) - H(S|a) \tag{2.32}$$

for a set S and a split a;

- *variance reduction*, typically used in regression trees (with a continuous co-domain), it quantifies the total reduction of the variance in the target after the split. Attributes with higher variance reduction are selected for splitting with higher priority. The metric is computed as:

$$\begin{aligned} I_V(S) = & \frac{1}{|S^{All}|^2} \sum_{i \in S^{All}} \sum_{j \in S^{All}} \frac{1}{2}(x_i - x_j)^2 + \\ & - \frac{1}{|S^T|^2} \sum_{i \in S^T} \sum_{j \in S^T} \frac{1}{2}(x_i - x_j)^2 + \\ & - \frac{1}{|S^F|^2} \sum_{i \in S^F} \sum_{j \in S^F} \frac{1}{2}(x_i - x_j)^2 \end{aligned} \tag{2.33}$$

where S^{All} is the set of all sample indices in the set S, S^T is the partition of indices for which the attribute test is true, and S^F is the partition of indices for which the attribute test is false.

2.3.2 Extremely Randomized Trees

Extremely Randomized Trees (Extra-Trees) [13] is a tree-based ensemble method for supervised learning, which consists in strongly randomizing both attribute and cut-point choice while splitting a decision tree node. The Extra-Trees algorithm builds an ensemble of M unpruned decision trees according to the classical top-down procedure, but differently from other tree induction methods it splits nodes by choosing cut-points fully at random. Extra-Trees can be applied to both classification and regression by building an ensemble of trees for either task.

The procedure to randomly split a node when building the ensemble is summarized in Algorithm 4.

The procedure has three main parameters:

- K, the number of attributes randomly selected at each node;

- n_{min}, the minimum sample size for splitting a node.

In the prediction phase, the output of each tree in the ensemble is aggregated to compute the final prediction, with a majority voting in classification problems and arithmetic average in regression problems.

Algorithm 4 Extra-Trees node splitting

```
Split_node($S$):
  Input: the local learning subset $S$ corresponding to the node we want to split
  Output: a split $[a < a_c]$ or nothing
  if Stop_split(S) is True then
    Return nothing
  else
    Select $K$ attributes $\{a_1, ..., a_k\}$ among all non constant candidate attributes in $S$
    Draw $K$ splits $\{S_1, ..., S_k\}$ where $S_i =$Pick_random_split($S, a_i$)
    Return a split $S_*$ such that $Score(S_*, S) = \max_i Score(S_i, S)$
  end if

Pick_random_split(S, a):
  Input: a subset $S$ and an attribute $a$
  Output: a split
  Let $a^S_{max}$ and $a^S_{min}$ denote the maximal and minimal value of $a$ in $S$
  Draw a random cut-point $a_c$ uniformly in $[a^S_{min}, a^S_{max}]$
  Return the split $[a < a_c]$

Stop_split($S$):
  Input: a subset $S$
  Output: a boolean
  if $|S| < n_{min}$ then
    return True
  end if
  if All attributes are constant in $S$ then
    return True
  end if
  if The output is constant in $S$ then
    return True
  end if
  Return False
```

Chapter 3

State Of The Art

The integration of RL and neural networks has a long history. Early RL literature [32, 38, 3] presents *connectionist* approaches in conjunction with a variety of RL algorithms, mostly using dense ANNs as approximators for the value functions from low-dimensional (or engineered) state spaces. The recent and exciting achievements of DL, however, have caused a sort of RL renaissance, with DRL algorithms outperforming classic RL techinques on environments which were previously considered intractable. Much like the game of Chess was believed out of the reach of machines until IBM's Deep Blue computer [6] won against world champion Garry Kasparov in 1997, DRL has paved the way to solve a wide spectrum of complex tasks which were previously considered a stronghold of humanity.

In this chapter we present the most important and recent results of DRL research, as well as some work related to the method proposed in this thesis.

3.1 Value-based Deep Reinforcement Learning

In 2015, Mnih et al. [27] introduced the *deep Q-learning* (DQN[1]) algorithm which basically ignited the field of DRL. The important contributions of DQN consisted in providing an end-to-end framework to train an agent on the *Atari* environments starting from the pixel-level representation of the states, with a deep CNN (called *deep Q-network*) used to estimate the Q function and apply greedy control. The authors were able to reuse the same architecture to solve many different games without the need for *hyper-parameter tuning*, which proved the effectiveness of the method.

The key idea of DQN is to embed the update step of Q-learning into the loss used

[1] Acronym of *Deep Q-Network*.

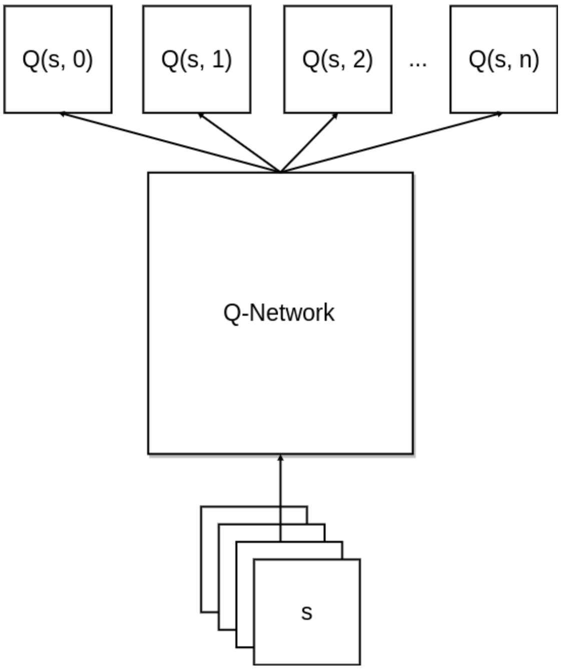


Figure 3.1: Structure of the Deep Q-Network by Mnih et al..

for SGD to train the deep CNN, resulting in the following gradient update:

$$\frac{\partial L}{\partial W_i^{old}} = E[(r + \gamma \max_{a'} Q(s', a'; \theta') - Q(s, a)) \frac{\partial Q(s, a; \theta)}{\partial W_i^{old}}] \tag{3.1}$$

where θ, θ' indicate two different sets of parameters for the CNN, which are respectively called the *online network* (θ) to select the action for the collection of samples, and the *target network* (θ') to produce the update targets. The online network is continuously updated during training, whereas the target network is kept fixed for longer time intervals in order to stabilize the online estimate. Moreover, a sampling procedure called *experience replay* [24] is used to stabilize training. This consists in keeping a variable training set of transitions collected with increasingly better policies (starting from a fully random ε-greedy policy and decreasing ε as the Q estimate improves), from which training samples are randomly selected. The full training procedure of DQN is reported in Algorithm 5.

From this work (which we could call introductory), many improvements have been proposed in the literature. Van Hasselt et al. (2016) proposed *Double DQN* (DDQN) [40] to solve an over-estimation issue typical of Q-learning, due to the use of the maximum action value as an approximation for the maximum expected action value (see Equation (2.29)). This general issue was addressed by Van Hasselt (2010) with *Double Q-learning* [18], a learning algorithm which keeps two separate estimates of the action-value function

Algorithm 5 Deep Q-Learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with two random sets of weights $\theta, \theta'$
for $episode = 1, M$ do
    for $t = 1, T$ do
        Select a random action $a_t$ with probability $\varepsilon$
        Otherwise, select $a_t = \arg\max_a Q(s_t, a; \theta)$
        Execute action $a_t$, collect reward $r_{t+1}$ and observe next state $s_{t+1}$
        Store the transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$
        Sample minibatch of transitions $(s_j, a_j, r_{j+1}, s_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_{j+1}, & \text{if } s_{j+1} \text{ is terminal} \\ r_{j+1} + \gamma \max_{a'} Q(s_{j+1}, a'; \theta'), & \text{otherwise} \end{cases}$
        Perform a gradient descent step using targets $y_j$ with respect to the online parameters $\theta$
        Every $C$ steps, set $\theta' \leftarrow \theta$
    end for
end for
```

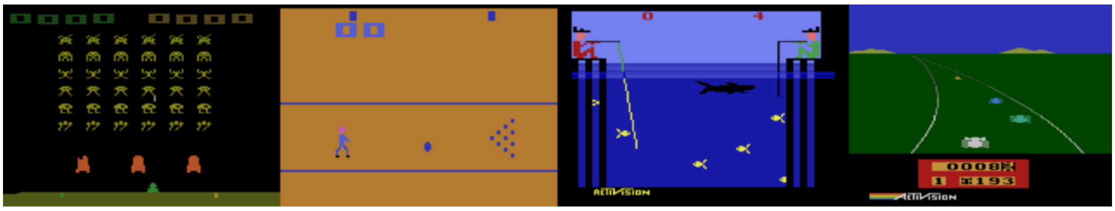


Figure 3.2: Some of the games available in the Atari environments.

Q^A and Q^B, and uses one to update the other as follows:

$$Q^A(s,a) \leftarrow Q^A(s,a) + \alpha[r + \gamma Q^B(s', \arg\max_a Q^A(s',a)) - Q^A(s,a)] \tag{3.2}$$

and vice-versa for Q^B. DDQN uses a similar approach to limit over-estimation in DQN by evaluating the greedy policy according to the online network, but using the target network to estimate its value. This is achieved with a small change in the computation of the update targets:

$$y_j = \begin{cases} r_{j+1}, & \text{if } s_{j+1} \text{ is terminal} \\ r_{j+1} + \gamma Q(s_{j+1}, \arg\max_a Q(s_{j+1}, a; \theta); \theta'), & \text{otherwise} \end{cases} \tag{3.3}$$

DDQN performed better (higher median and mean score) on the 49 Atari games used as benchmark by Mnih et al. (2015), equaling or even surpassing humans on several games.

Schaul et al. (2016) [33] developed the concept of *prioritized experience replay*, which replaced DQN's uniform sampling strategy from the replay memory with a sampling strategy weighted by the *TD errors* committed by the network. This improved the performance of both DQN and DDQN.

Wang et al. (2016) introduced a slightly different end-to-end *dueling architecture* [41], composed of two different deep estimators: one for the state-value function V and one for the *advantage function* $A : S \times A \to \mathbb{R}$ defined as:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{3.4}$$

In this approach, the two networks share the same convolutional layers but use two separate dense layers. The two streams are then combined to estimate the optimal action-value function as[2]:

$$Q^\pi(s,a) = V^\pi(s) + (A^\pi(s,a) - \max_{a'} A^\pi(s,a')) \tag{3.5}$$

Several other extensions of the DQN algorithm have been proposed in recent years. Among these, we cite Osband et al. (2016) [30] who proposed a better exploration strategy based on Thompson sampling, to select an exploration policy based on the probability that it is the optimal policy; He et al. (2017) [19] added a constrained optimization approach called *optimality tightening* to propagate the reward faster during updates and improve accuracy and convergence; Anschel et al. (2017) [1] improved the variance and instability of DQN by averaging previous Q estimates; Munos et al. (2016) [28] and Harutyunyan et al. (2016) [16] proposed to incorporate on-policy samples to the Q-learning target and seamlessly switch between off-policy and on-policy samples, which again resulted in faster reward propagation and convergence.

3.2 Other approaches

3.2.1 Memory architectures

Graves et al. (2016) [15] proposed *Differentiable Neural Computer* (DNC), an architecture in which an ANN has access to an external memory structure, and learns to read and write data by gradient descent in a goal-oriented manner. This approach outperformed normal ANNs and DNC's precursor *Neural Turing Machine* [14] on a variety of query-answering and natural language processing tasks, and was used to solve a simple

[2] In the original paper, the authors explicitly indicate the dependence of the estimates on different parameters (e.g. $V^\pi(s,a;\phi,\alpha)$ where ϕ is the set of parameters of the convolutional layers and α of the dense layers). For simplicity in the notation, here we report the estimates computed by the network with the same notation as the estimated functions (i.e. the network which approximates V^π is indicated as V^π, and so on...).

moving block puzzle with a form of reinforcement learning in which a sequence of instructions describing a goal is coupled to a reward function that evaluates whether the goal is satisfied (a set-up that resembles an animal training protocol with a symbolic task cue).

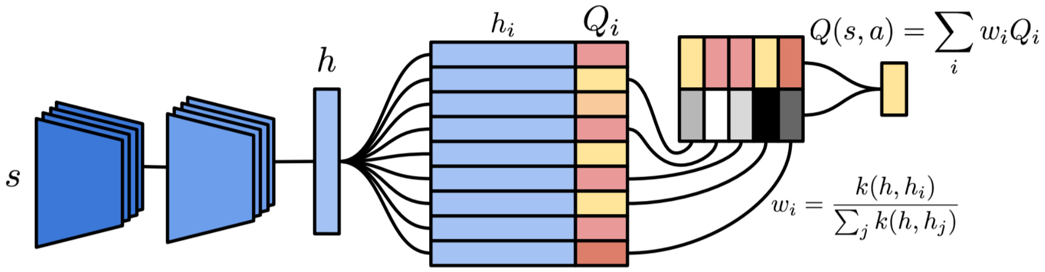

s
h
h_i
Q_i
$Q(s,a) = \sum_i w_i Q_i$
$w_i = \frac{k(h,h_i)}{\sum_j k(h,h_j)}$

Figure 3.3: Architecture of NEC (image taken from [31]).

Pritzel et al. (2017) [31] extended the concept of differentiable memory to DQN with *Neural Episodic Control* (NEC). In this apporach, the DRL agent consists of three components: a CNN which processes pixel images, a set of memory modules (one per action), and a dense ANN which converts read-outs from the action memories into action-values. The memory modules, called *differentiable neural dictionaries* (DNDs), are memory structures which resemble the dictionary data type found in computer programs. DNDs are used in NEC to associate the state embeddings computed by the CNN to a corresponding Q estimate, for each visited state: a read-out for a key consists in a weighted sum of the values in the DND, with weights given by normalized kernels between the lookup key and the corresponding key in memory (see Figure 3.3). DNDs are populated automatically by the algorithm without learning what to write, which greatly speeds up the training time with respect to DNC.

NEC outperformed every previous DRL approach on Atari games, by achieving better results using less training samples.

3.2.2 AlphaGo

Traditional board games like chess, checkers, Othello and Go are classical test benches for artificial intelligence. Since the set of rules which characterizes this type of games is fairly simple to represent in a program, the difficulty in solving these environments stems from the complexity of the state space. Among the cited games, Go was one of the last board games in which an algorithm had never beaten top human players, because its characteristic 19 × 19 board which allows for approximately 250^{150} sequences of moves[3] was too complex for exhaustive search methods.

[3] Number of legal moves per position elevated to the length of the game.

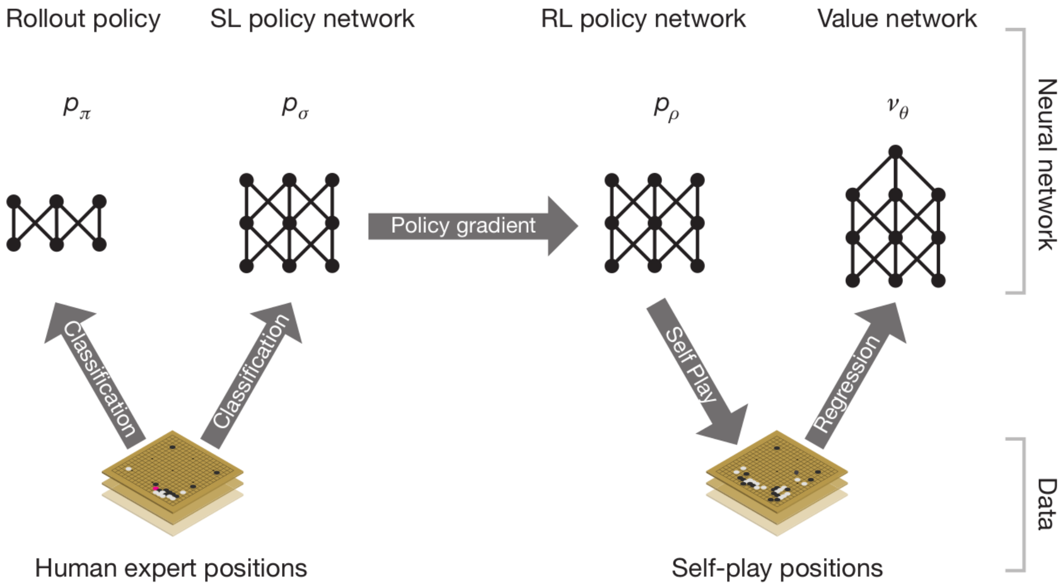


Figure 3.4: Neural network training pipeline of AlphaGo (image taken from [34]).

Silver et al. (2016) [34] introduced *AlphaGo*, a computer program based on DRL which won 5 games to 0 against the European Go champion in October 2015; soon after that, AlphaGo defeated 18-time world champion Lee Sedol 4 games to 1 in March 2016, and world champion Ke Jie 3 to 0 in May 2017. After these results, Google DeepMind (the company behind AlphaGo) decided to retire the program from official competitions and released a dataset containing 50 self-play games [17].

AlphaGo is a complex architecture which combines deep CNNs, reinforcement learning, and Monte Carlo Tree Search (MCTS) [5, 12]. The process is divided in two phases: a neural network training pipeline and MCTS. In the training pipeline, four different networks are trained: a *supervised learning* (SL) policy network trained to predict human moves; a *fast* policy network to rapidly sample actions during MC rollouts; a *reinforcement learning* policy network that improves the SL network by optimizing the final outcome of games of self-play; a *value* network that predicts the winner of games (see Figure 3.4). Finally, the policy and value networks are combined in an MCTS algorithm that selects actions with a lookahead search, by building a partial search tree using the estimates computed with each network.

3.2.3 Asynchronous Advantage Actor-Critic

Actor-critic algorithms [36] are RL methods that have a separate memory structure to explicitly represent the policy independent of the value function. The policy structure is known as the actor, because it is used to select actions, and the estimated value function is known as the critic, because it criticizes the actions made by the actor. Learning is always on-policy, as the critic must learn about and critique whatever policy is currently being followed by the actor. The critique usually takes the form of a TD error which is used to drive learning in both the actor and the critic, evaluated as:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{3.6}$$

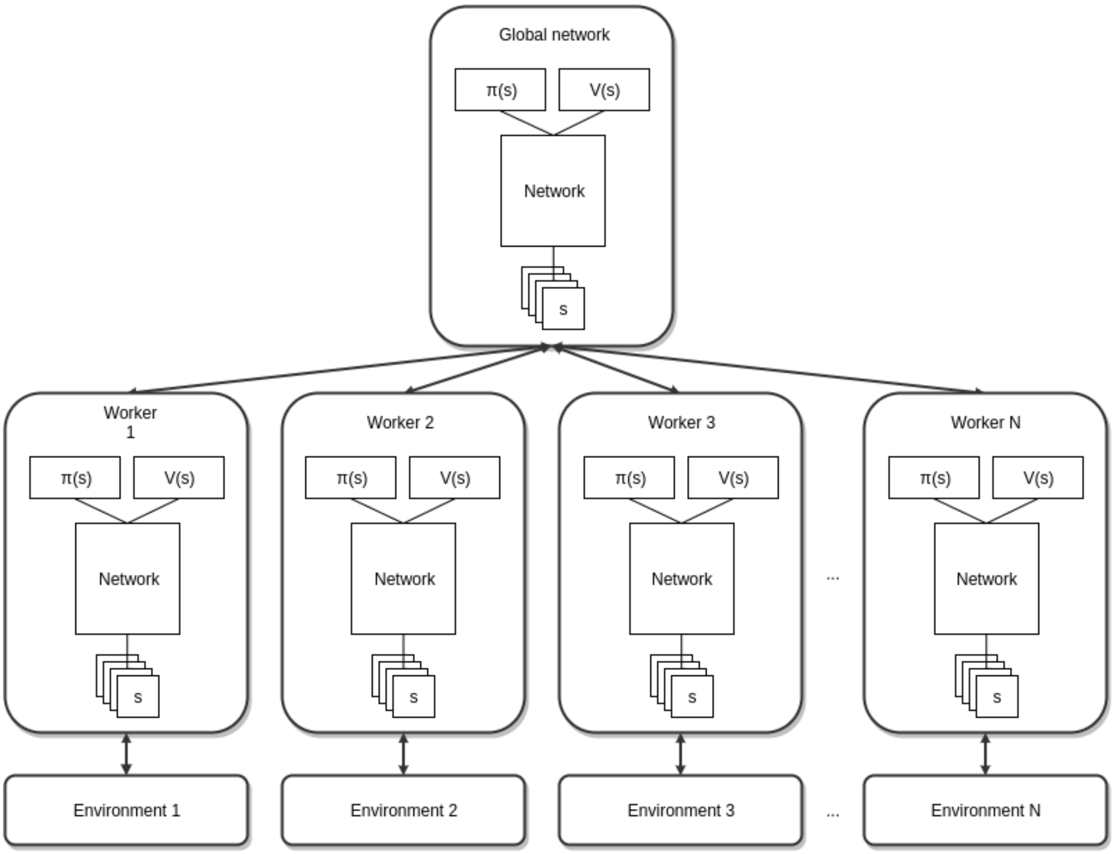


Figure 3.5: The asynchronous architecture of A3C.

Mnih et al. (2016) [26] presented a deep variation of the actor-critic algorithm, called *Asynchronous Advantage Actor-Critic* (A3C). In this approach, different instances of actor-critic pairs are run in parallel (Figure 3.5) to obtain a lower-variance estimate of the value function, without the need of a replay memory to stabilize training. Each *worker* consists in a deep CNN with a unique convolutional section and two separate dense networks on top, one for the value function and one for the policy.

This asynchronous methodology was applied to other classical RL algorithms in the same paper; we only report the actor-critic variant as it was the best performing, with notably shorter training times and performance comparable to DQN and its variations.

3.3 Related Work

Lange and Riedmiller (2010) [22] proposed the *Deep Fitted Q-iteration* (DFQ) algorithm, a batch RL method which used deep dense autoencoders to extract a state representation from pixel images. In this algorithm, a training set of (s, a, r, s') transitions is collected with a random exploration strategy, where s, s' are pixel images of two consecutive states. The samples are then used to train a dense autoencoder with two neurons in the innermost layer, which in turn is used to encode all states in the training set. This encoded dataset is then passed as input to FQI, which produces an estimate for the Q function using a kernel based approximator. A new policy is then computed from the estimated Q and the encoder, and the process is repeated starting with the new policy until the obtained Q is considered satisfactory. The authors applied DFQ to a simple *Gridworld* environment with fixed size and goal state, and were able to outperform other image-based feature extraction methods (like *Principal Components Analysis* [42]) with good sample efficiency.

Chapter 4

Fitted Q-Iteration with Deep State Features

As central contribution of this thesis, we propose a DRL algorithm which combines the feature extraction capabilities of unsupervised deep CNNs with the fast and powerful batch RL approach of FQI. Given a control problem with high-dimensional states and a mono-dimensional discrete action space, we use a deep convolutional autoencoder to map the original state space to a compressed *feature space* which accounts for information on the states and the nominal dynamics of the environment (i.e. those changes not directly influenced by the agent). We reduce the representation further by applying the *Recursive Feature Selection* (RFS) algorithm to the extracted state features, and this minimal state space is then used to run the FQI algorithm. We repeat this procedure iteratively in a *semi-batch* approach to bootstrap the algorithm, starting from a fully random exploration of the environment.

In this chapter we give a formal description of the method and its core components, whereas technical details of implementation will be discussed in the next chapter.

4.1 Motivation

The state-of-the-art DRL methods listed in the previous chapter are able to outperform classic RL algorithms in a wide variety of problems, and in some cases are the only possible way of dealing with high-dimensional control settings like the Atari games. However, the approaches cited above tend to be grossly *sample-inefficient*, requiring tens of millions of samples collected on-line to reach optimal performance. Several publications successfully deal with this aspect, but nonetheless leave room for improvement (lowering at most by one order of magnitude the number of samples required). The method introduced by Lange and Riedmiller (2010) [22] is similar to ours but their dense architecture predates the more modern convolutional approaches in image processing and is less suited than our AE for complex tasks.

The method that we propose tries to improve both aspects of information content of the compressed feature space and sample efficiency. We extract general features from the environments in an unsupervised fashion, which forgoes the task-oriented representation of the current state-of-the-art methods in favor of a description which relates more closely to the original information content of the states; because of the generality of this description, the compressed state space can be used with a variety of RL algorithms in their default setting (like we show with FQI), without having to adapt the deep architecture to solve the problem end-to-end like in deep Q-learning. Moreover, since deep features can be difficult to interpret in an abstract way and in general there is no assurance against redundancy in the representation, we use a feature selection technique on the extracted space in order to really minimize the description while retaining all information required for control. Finally, our sample-efficient methodology allows our agent to learn non-trivial policies using up to two orders of magnitude less samples than the state-of-the-art approaches.

4.2 Algorithm Description

The general structure of this algorithm is typical of DRL settings: we use a deep ANN to extract a representation of an environment, and use that representation to control an agent with standard RL algorithms. We also add an additional step after the deep feature extraction to further reduce the representation down to the essential bits of information required to solve the problem by using the *Recursive Feature Selection* (RFS) algorithm [7].

We focus exclusively on environments with a discrete and mono-dimensional action space $A = \{0, 1, ..., a\}$, where actions are assigned a unique integer identifier starting from 0 with no particular order. We also assume to be operating in a three dimensional state space (*channels* × *height* × *width*) for consistency with the experimental setting on which we tested the algorithm (with pixel-level state spaces), although in general the algorithm requires no such assumption and could be easily adapted to higher or lower dimensional settings.

The algorithm uses a modular architecture with three different components which are combined after training to produce an approximator of the action-value function. The main components of the algorithm are:

1. a deep convolutional autoencoder which we use to extract a representation of the environment; the purpose of the AE is to map the original, pixel-level state space S of the environment into a strongly compressed feature space $\tilde{S}$, which contains information on both the states and part of the transition model of the environment;

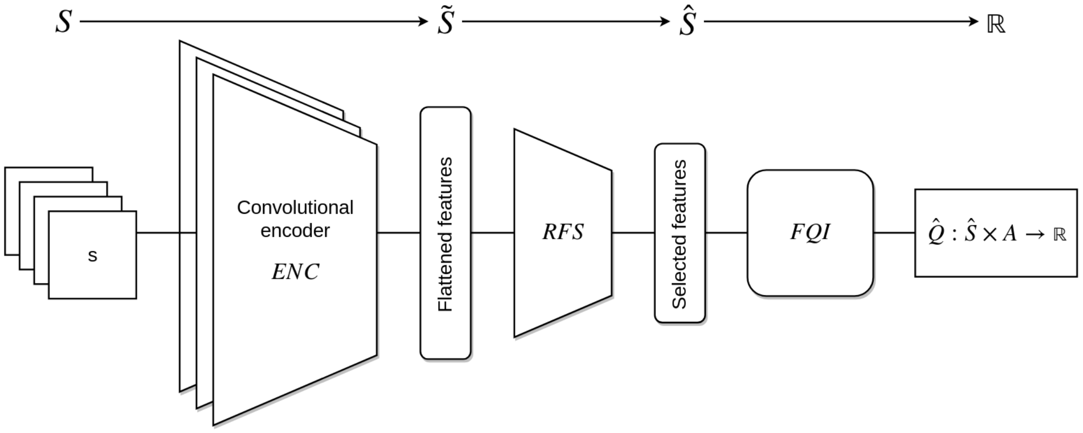


Figure 4.1: Composition of the three main modules of the algorithm to produce the end-to-end function $Q : S \times A \to \mathbb{R}$. Pixel-level states are transformed by the encoder to the 1D feature space $\tilde{S}$, which in turn is filtered by RFS into space $\hat{S}$. The final representation is used to train FQI which outputs an action-value function $\hat{Q}$ on $\hat{S} \times A$. The same preprocessing pipeline is then required to use $\hat{Q}$, and the full composition of the three modules effectively consists in an action-value function on the original space $S \times A$.

2. the *Recursive Feature Selection* (RFS) technique to further reduce the state representation $\tilde{S}$ and keep only the truly informative features extracted by the AE, effectively mapping the extracted feature space to a subspace $\hat{S}$.

3. the *tree-based* FQI learning algorithm which produces an estimator for the action-value function, with $\hat{S}$ as domain.

The full procedure consists in alternating a training step and an evaluation step, until the desired performance is reached. ,

A training step of the algorithm takes as input a training set $\mathcal{TS}$ of four-tuples $(s \in S, a \in A, r \in \mathbb{R}, s' \in S)$ to produce an approximation of the action-value function, and consists in sequentially training the three components from scratch to obtain the following transformations, respectively:

- $ENC : S \to \tilde{S}$, from the pixel representation to a compressed feature space;
- $RFS : \tilde{S} \to \hat{S}$, from the compressed feature space to a minimal subspace with the most informative features;
- $\hat{Q} : \hat{S} \times A \to \mathbb{R}$, an approximation of the optimal action-value function on $\hat{S}$.

After training, we simply combine the three functions (see Figure 4.1) to obtain the full action-value function $Q : S \times A \to \mathbb{R}$ as follows:

$$Q(s, a) = \hat{Q}(RFS(ENC(s)), a) \tag{4.1}$$

To collect the training set $\mathcal{TS}$, we define a greedy policy π based on the current approximation of Q as:

$$\pi(s) = \arg\max_a Q(s, a) \tag{4.2}$$

and we use an ε-greedy policy π_ε based on π (cf. Section 2.2.1) to collect $\mathcal{TS}$. We initialize the ε-greedy policy as fully random (which means that we do not need an approximation of Q for the first step), and we decrease ε after each step down to a fixed minimum positive value ε_{min}. This results in a sufficiently high exploration at the beginning of the procedure, but increasingly exploits the learned knowledge to improve the quality of the collected samples after each training step, as the agent learns to reach states with a higher value. The lower positive bound on ε is kept to minimize overfitting and allow the agent to explore potentially better states even in the later steps of the algorithm. A similar approach, called *ε-annealing*, was used by Mnih et al. (2015) [27] for the online updates in DQN.

Each training step is followed by an evaluation phase to determine the quality of the learned policy and eventually stop the procedure when the performance is deemed satisfactory.

A general description of the process is reported in Algorithm 6, and details on the training phases and evaluation step are given in the following sections. Note that in general the *semi-batch* approach, which starts from a training set collected under a fully random policy and sequentially collects new datasets as performance improves, is not strictly necessary. It is also possible to run a single training step in pure batch mode, using a dataset collected under an expert policy (albeit with some degree of exploration required by FQI) to produce an approximation of the optimal Q function in one step. This allows to exploit the sample efficiency of the algorithm in situations where, for instance, collecting expert samples is expensive or difficult.

4.3 Extraction of State Features

The AE used in our approach consists of two main components, namely an *encoder* and a *decoder* (cf. Section 2.1.4). To the end of explicitly representing the encoding purpose of the AE, we keep a separate notation of the two modules; we therefore refer to two different CNNs, namely $ENC : S \to \tilde{S}$ that maps the original state space to the compressed representation $\tilde{S}$, and $DEC : \tilde{S} \to S$ which performs the inverse transformation. The full AE is the composition of the two networks $AE : DEC \circ ENC : S \to S$ (Figure 4.2). Note that the composition is differentiable end-to-end, and basically consists in *plugging* the last layer of the encoder as input to the decoder. Both components (in our specific implementation, discussed further in the next chapter) are convolutional neural networks specifically structured to reduce the representation down to an arbitrary number of features in the new space.

Algorithm 6 Fitted Q-Iteration with Deep State Features

```
Given:
  $\varepsilon_{min} \in (0,1)$
  $\varphi : [\varepsilon_{min}, 1] \to [\varepsilon_{min}, 1)$ s.t. $\varphi(x) < x, \forall x \in (\varepsilon_{min}, 1]$ and $\varphi(\varepsilon_{min}) = \varepsilon_{min}$;
Initialize the encoder $ENC : S \to \tilde{S}$ arbitrarily;
Initialize the decoder $DEC : \tilde{S} \to S$ arbitrarily;
Initialize $Q$ arbitrarily;
Define $\pi(s) = \arg\max_{a} Q(s, a)$;
Initialize an $\varepsilon$-greedy policy $\pi_\varepsilon$ based on $\pi$ with $\varepsilon = 1$;
repeat
  Collect a set $\mathcal{TS}$ of four-tuples $(s \in S, a \in A, r \in \mathbb{R}, s' \in S)$ using $\pi_\varepsilon$;
  Train the composition $DEC \circ ENC : S \to S$ using the state elements $(s \in S)$ of $\mathcal{TS}$ as input and target;
  Build a set $\mathcal{TS}_{ENC}$ of four-tuples $(\tilde{s} \in \tilde{S}, a \in A, r \in \mathbb{R}, \tilde{s}' \in \tilde{S})$ by applying the encoder to the first and last elements of each four-tuple in $\mathcal{TS}$ s.t. $\tilde{s} = ENC(s)$;
  Call the RFS feature selection algorithm on $\mathcal{TS}_{ENC}$ to obtain a space reduction $RFS : \tilde{S} \to \hat{S}$;
  Build a set $\mathcal{TS}_{RFS}$ of four-tuples $(\hat{s} \in \hat{S}, a \in A, r \in \mathbb{R}, \hat{s}' \in \hat{S})$ by applying $RFS$ to the first and last elements of each four-tuple in $\mathcal{TS}_{ENC}$ s.t. $\hat{s} = RFS(\tilde{s})$;
  Call FQI on $\mathcal{TS}_{RFS}$ to produce $\hat{Q} : \hat{S} \times A \to \mathbb{R}$;
  Combine $\hat{Q}$, $RFS$ and $ENC$ to produce $Q : S \times A \to \mathbb{R}$:

      $Q(s, a) = \hat{Q}(RFS(ENC(s)), a)$

  Set $\varepsilon \leftarrow \varphi(\varepsilon)$;
  Evaluate $\pi$;
until stopping condition on evaluation performance is met;
Return $Q$
```

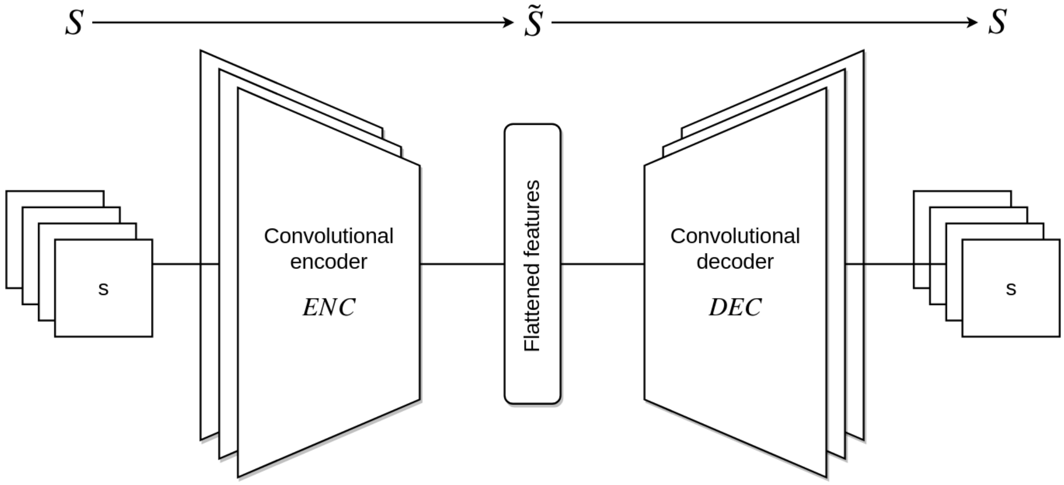


Figure 4.2: Schematic view of the AE.

We train the AE to reconstruct its input in an unsupervised fashion, using the a dataset $\mathcal{TS}$ of four-tuples $(s \in S, a \in A, r \in \mathbb{R}, s' \in S)$ collected with the ε-greedy policy and taking the first element $(s \in S)$ of each sample as input and target. The output of this phase is the trained encoder $ENC : S \to \tilde{S}$, which takes as input the three-dimensional states of the environment and produces a one-dimensional feature vector representing a high-level abstraction of the state space.

We refer to $\tilde{S}$ as a representation of both the state and the nominal environment dynamics because we allow for the original state space S to contain both such types of information, accounting for changes in the environment which are not directly (or even indirectly) caused by the user. An example of this is the *MsPacman* environment in the Atari suite, implementation of the popular arcade game, in which the *ghosts* are able to move autonomously, not necessarily influenced by the player; in such a setting, the environment presents a non-trivial nominal dynamic which must be taken into account in the extracted representation (Figure 4.3). Details on how we accomplish this are provided in the following chapter, when dealing with the specific implementation of our experimental setting.

4.4 Recursive Feature Selection

The *Recursive Feature Selection* (RFS) [7] algorithm is a dimensionality reduction technique for control problems proposed by Castelletti et al. (2011). The algorithm identifies which state and action features (i.e. elements of the state and action vector spaces) are most relevant for control purposes, reducing the dimensionality of both spaces by removing the less important features. The core of the algorithm consists in recursively

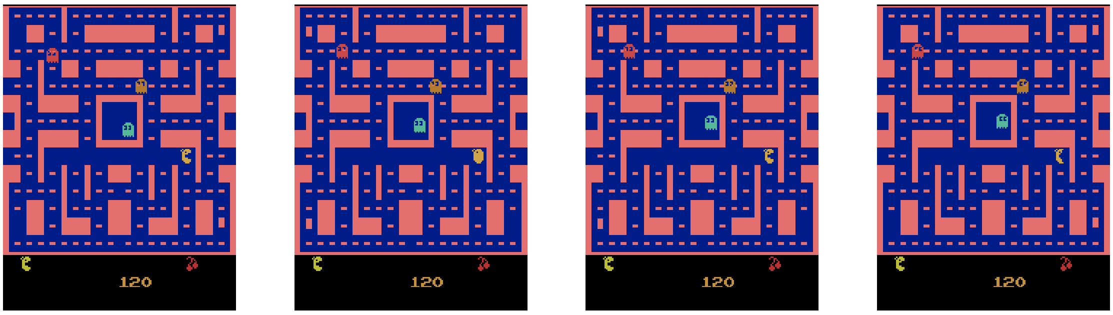


Figure 4.3: Example of non-trivial nominal dynamics in the MsPacman *environment, with the ghosts moving autonomously across the four consecutive observations.*

selecting the features which best explain the dynamics (i.e. the transition model) of the features already selected, starting from the subset of features needed to explain the reward model, using the *Iterative Feature Selection* (IFS) [11] algorithm to recursively build a dependency tree of features.

Algorithm 7 Recursive Feature Selection (RFS)

```
Given: a dataset $\mathcal{D} = \langle s \in S, a \in A, s' \in S \rangle$, a target feature $F_0^i$ from the set of features of $S \cup A$, a set $\mathcal{F}_{sel}^i$ of previously selected features;
$\mathcal{F}_{F_0}^i \leftarrow IFS(\mathcal{D}, F_0^i)$
$\mathcal{F}_{new}^i \leftarrow \mathcal{F}_{F_0}^i \setminus \mathcal{F}_{sel}^i$
for all $F_j^{i+1} \in \mathcal{F}_{new}^i$ do
  $\mathcal{F}_{F_0}^i \leftarrow \mathcal{F}_{F_0}^i \cup RFS(\mathcal{D}, F_j^{i+1}, \mathcal{F}_{sel}^i \cup \mathcal{F}_{F_0}^i)$
end for
return $\mathcal{F}_{F_0}^i$
```

The main procedure (summarized in Algorithm 7) takes as input a dataset $\mathcal{D}$ of observed transitions from an environment, the values of a target feature F_0^i in $\mathcal{D}$, and a set $\mathcal{F}_{sel}^i$ of previously selected features. The dataset and target feature are given as input to IFS, which returns a subset of features $\mathcal{F}_{F_0}^i$ which best explain the dynamics of F_0^i. RFS is then recursively called on each feature in the set $\mathcal{F}_{new}^i = \mathcal{F}_{F_0}^i \setminus \mathcal{F}_{sel}^i$ of new features selected by IFS. At the first step, the algorithm is usually run to identify the most important features to explain the reward R by setting $F_0^0 = R$ and $\mathcal{F}_{sel}^0 = \emptyset$, so that the final output of the procedure will be a set of features which describe the dynamics of the reward and of the environment itself (a simple example of RFS is given in Figure 4.4).

The IFS procedure called at each step of RFS is a feature selection technique based on a *feature ranking* algorithm, but in general can be replaced by any feature selection method which is able to account for non-linear dependencies and redundancy between

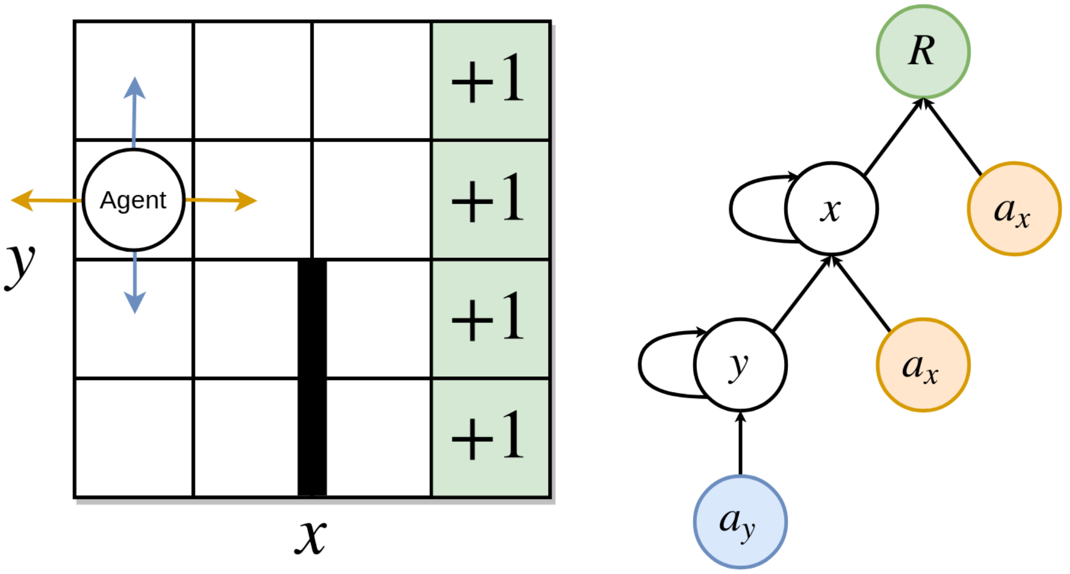


Figure 4.4: In this simple example we show the output of RFS in a simple Gridworld environment with two state features, x and y, to represent the agent's position and two action features, a_x and a_y valued in $\{-1, 0, 1\}$, to control the corresponding state features and move the agent on the discrete grid (the movement is the sum of the two effects). We use a setting with null reward everywhere except in the four goal states (green) which yield a positive reward, and a wall which cancels the effect of action a_x if the agent tries to move through it. To the right we see the dependency tree produced by RFS, which in this case selects both state features and both action features. Intuitively, the reward is explained exclusively by the state feature x and the action that controls it, a_x, because the agent gets a positive reward in each of the four cells to the far right, independently of its y coordinate, whenever it reaches one. At the same time, feature x is explained by its value and by the action feature a_x (because the next value of x is a function of both, namely $x' = x + a_x$), but also by feature y, because given the x coordinate of the agent and a direction along x we still have to know whether the agent is next to the wall in order to compute the following value of x. Finally, feature y is explained by itself and its associated action, without restrictions given by the wall.

features (as real-world control problems are usually characterized by non-linear dynamic models with multiple coupled features). Here we use IFS for coherence with the paper by Castelletti et al., and because it is a computationally efficient feature selection algorithm which works well with our semi-batch approach. IFS takes as input a dataset $\mathcal{D}$ of observed (s, a, r, s') transitions from the environment and the values of a target feature F_0 in $\mathcal{D}$. The algorithm starts by globally ranking the features of the state space according to a statistical level of significance provided by a feature ranking method FR, which takes as input the dataset and the target feature; the most significant feature F^* according to the ranking is added to the set of selected features $\mathcal{F}_{sel}$. A supervised model $\hat{f}$ is then trained to approximate the target feature F_0, taking $\mathcal{F}_{sel}$ as input. The algorithm then proceeds by repeating the ranking process using as new target feature for FR the residual feature $\hat{F}_0 = F_0 - \hat{f}(\mathcal{F}_{sel})$ (the difference between the target value and the approximation computed by $\hat{f}$ for each sample in $\mathcal{D}$). The procedure continues to perform these operations until either the best variable in the feature ranking is already in $\mathcal{F}_{sel}$, or the accuracy of the model built upon $\mathcal{F}_{sel}$ does not improve. The accuracy of the model is computed with the coefficient of determination R^2 between the values of the target feature F_0 and the values $F_{pred} = \hat{f}(\mathcal{F}_{sel})$ predicted by the model:

$$R^2(F_0, F_{pred}) = 1 - \frac{\sum\limits_k (f_{0,k} - f_{pred,k})^2}{\sum\limits_k (f_{0,k} - \frac{1}{|\mathcal{D}|} \sum\limits_{i=1}^{|\mathcal{D}|} f_{0,i})} \tag{4.3}$$

The full IFS procedure is summarized in Algorithm 8.

To the extent of our algorithm, we run the RFS procedure on a dataset $\mathcal{TS}_{ENC}$ of four-tuples $(\tilde{s} \in \tilde{S}, a \in A, r \in \mathbb{R}, \tilde{s}' \in \tilde{S})$ built by applying the transformation ENC to the first and last elements of each four-tuple in $\mathcal{TS}$. We run the algorithm on all the state and action features but, since we assume to be working in a mono-dimensional action space, we force the action feature to always be part of the representation, so that the procedure is effectively working only on $\tilde{S}$.

At the end of the procedure, we define a simple filtering operation $RFS : \tilde{S} \to \hat{S}$ that consists in keeping only the features of $\tilde{S}$ which have been selected by the algorithm. The output of the training phase is this transformation.

4.5 Fitted Q-Iteration

The last component of our training pipeline is the Fitted Q-Iteration algorithm (cf. Section 2.2.7). We provide as input to the procedure a new training set $\mathcal{TS}_{RFS}$ of four-tuples $(\hat{s} \in \hat{S}, a \in A, r \in \mathbb{R}, \hat{s}' \in \hat{S})$, obtained by applying the RFS transformation to the first and last elements of each four-tuple in $\mathcal{TS}_{ENC}$. We train the model to output a multi-dimensional estimate of the action-value function on $\mathbb{R}^{|A|}$, with one value for each

Algorithm 8 Iterative Feature Selection (IFS)

```
Given: a dataset $\mathcal{D} = \langle s \in S, a \in A, s' \in S \rangle$, a target feature $F_0$ from the set of features of $S \cup A$;
Initialize: $\mathcal{F}_{sel} \leftarrow \emptyset, \hat{F}_0 \leftarrow F_0, R^2_{old} \leftarrow 0$
repeat
    $F^* \leftarrow \arg\max_F FR(\mathcal{D}, \hat{F}_0, F)$
    if $F^* \in \mathcal{F}_{sel}$ then
        return $\mathcal{F}_{sel}$
    end if
    $\mathcal{F}_{sel} \leftarrow \mathcal{F}_{sel} \cup F^*$;
    Build a model $\hat{f} : \mathcal{F}_{sel} \to F_0$ using $\mathcal{D}$;
    $F_{pred} = \hat{f}(\mathcal{F}_{sel})$
    $\hat{F}_0 \leftarrow F_0 - F_{pred}$
    $\Delta R^2 \leftarrow R^2(\mathcal{D}, F_0, F_{pred}) - R^2_{old}$
    $R^2_{old} \leftarrow R^2(\mathcal{D}, F_0, F_{pred})$
until $\Delta R^2 < \epsilon$
return $\mathcal{F}_{sel}$
```

action that the agent can take in the environment, and we restrict the output to a single value on $\mathbb{R}$ using the action identifier as index (e.g. $Q(s, 0)$ will return the first dimension of the model's output, corresponding to the value of action 0 in state s). The output of this training phase is the transformation $\hat{Q} : \hat{S} \times A \to \mathbb{R}$, which is then combined with ENC and RFS as per Equation (4.1) to produce the approximation of Q. This phase also concludes the training step.

Chapter 5

Technical Details and Implementation

In this chapter we show the implementation details of the architecture used to perform experiments. We try to provide a complete description of the parametrization of the components and of the training procedure to ensure reproducibility of the experimental results.

5.1 Atari Environments

The *Arcade Learning Environment* (ALE) [2] is an evaluation platform for RL agents. ALE offers a programmatic interface to hundreds of game environments for the *Atari 2600*, a popular home video game console developed in 1977 with more than 500 games available, and is simply referred to as *Atari environments* (or *games*). We use the implementation of ALE provided by the *Gym 0.9.2* package for Python 2.7, developed by OpenAI and maintained as an open source project. This implementation provides access to the game state in the form of $3 \times 110 \times 84$ RGB frames, produced at an internal frame-rate of 60 frames per second (FPS). When an action is executed in the environment, the simulator repeats the action for four consecutive frames of the game and then provides another observation, effectively lowering the frame rate from 60 FPS to 15 FPS and making the effects of actions more evident (Figure 5.1). We perform a preprocessing operation on the states similar to that performed by Mnih et al. in DQN [27], in order to include all necessary information about the environment and its nominal dynamics in the new state representation. First we convert each RGB observation to a single-channel greyscale representation in the discrete 8-bit interval $[0, 255]$ using the

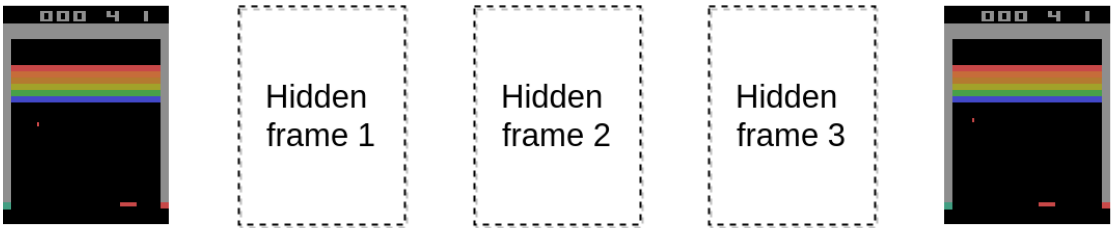


Figure 5.1: Sampling frames at a frequency of $\frac{1}{4}$ in Breakout.

ITU-R 601-2 luma transform:

$$L = \frac{299}{1000}R + \frac{587}{1000}G + \frac{114}{1000}B \tag{5.1}$$

where R, G, B are the 8-bit *red, green* and *blue* channels of the image. We then normalize these values in the $[0, 1]$ interval via *max-scaling* (i.e. dividing each pixel by 255). Moreover, we define a rounding threshold v_{round} specific to each game and we round all pixel values above or below the threshold to 1 or 0 respectively, reducing the images to a 1-bit color space in order to facilitate the training of the AE. We also reduce the height of the image by two pixels in order to prevent information loss due to a rounding operation performed by the convolution method used in the AE. Finally, we concatenate the preprocessed observation to the last three preprocessed frames observed from the environment, effectively blowing up the state space to a $4 \times 108 \times 84$ vector space (Figure 5.2). The initial state for an episode is artificially set as a repetition of the first observation provided by the simulator.

Moreover, in order to facilitate comparisons and improve stability, we remain loyal to the methodology used for DQN and perform the same clipping of the reward signal in a $[-1, 1]$ interval.

We also add a tweak to the Gym implementation of ALE in order to fix a requirement of some environments of performing some specific actions in order to start an episode (e.g. in *Breakout* it is required that the agent takes action 1 to start the game). We automatically start each episode by randomly selecting one of the initial actions of the game and forcing the agent to take that action at the beginning of the episode.

Finally, for those games in which the agent has a lives count, we monitor the number of remaining lives provided by the simulator in the *ale.lives* parameter. For those (s, a, r, s') transitions in which the agent loses a life, we consider s' as a terminal state, so that the FQI update will only propagate the reward rather than the full value.

5.2 Autoencoder

We use the autoencoder to extract a high level representation $\tilde{S}$ of the state space in an unsupervised fashion. We structure the AE to take as input the preprocessed

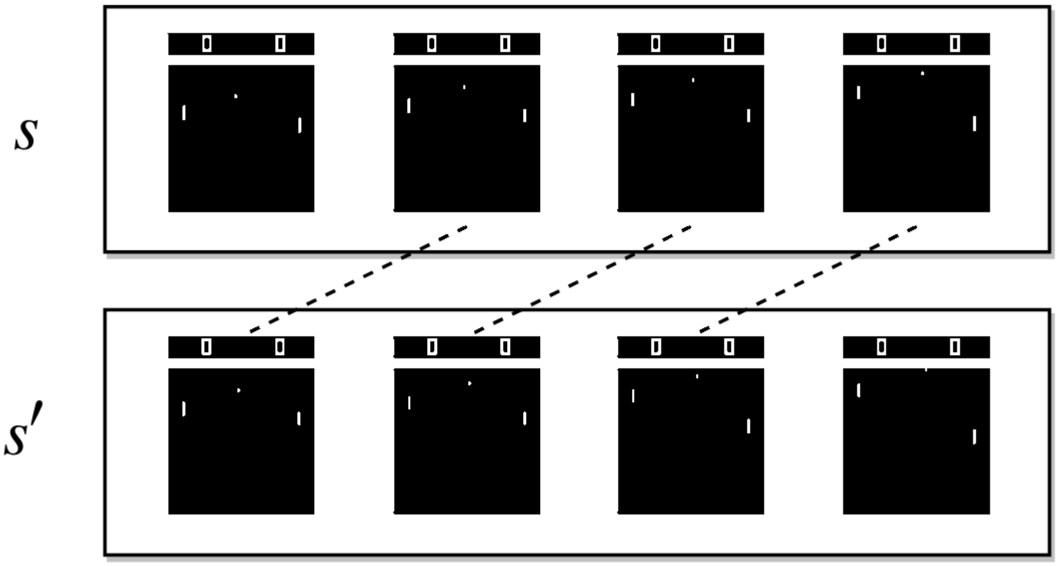


Figure 5.2: Two consecutive states s, s' in Pong, after the preprocessing operation. Each binary image in a sequence of four is an observation of 108×84 pixels, and the full $4 \times 108 \times 84$ tensor is given as input and target to the AE. Notice that between the two consecutive states there are three frames in common, with the newest observation appended as last frame in s' and the oldest being pushed out.

observations from the environment and predict values on the same vector space. The first four convolutional layers make up the encoder and perform a 2D convolution with the *valid* padding algorithm such that the input of each layer (in the format $channels \times height \times width$) is reduced automatically across the last two dimensions (height and width) according to the following formula:

$$output_i = \lfloor (input_i - filter_i + stride_i)/stride_i \rfloor \tag{5.2}$$

Since the main purpose of pooling layers is to provide translation invariance to the representation of the CNN (meaning that slightly shifted or tilted inputs are considered the same by the network), here we choose to not use pooling layers in order to preserve the precious information regarding the position of different elements in the game; this same approach was adopted in DQN. A final *Flatten* layer is added at the end of the encoder to provide a 1D representation of the feature space, which is reversed before the beginning of the decoder. The decoder consists of deconvolutional layers with symmetrical filter sizes, filter numbers and strides with respect to the encoder. Here the *valid* padding algorithm is inversed to expand the representation with this formula:

$$output_i = \lfloor (input_i \cdot stride_i) + filter_i - stride_i \rfloor \tag{5.3}$$

Type	Input	Output	# Filters	Filter	Stride	Activation
Conv.	$4 \times 108 \times 84$	$32 \times 26 \times 20$	32	8×8	4×4	ReLU
Conv.	$32 \times 26 \times 20$	$64 \times 12 \times 9$	64	4×4	2×2	ReLU
Conv.	$64 \times 12 \times 9$	$64 \times 10 \times 7$	64	3×3	1×1	ReLU
Conv.	$64 \times 10 \times 7$	$16 \times 8 \times 5$	16	3×3	1×1	ReLU
Flatten	$16 \times 8 \times 5$	640	-	-	-	-
Reshape	640	$16 \times 8 \times 5$	-	-	-	-
Deconv.	$16 \times 8 \times 5$	$16 \times 10 \times 7$	16	3×3	1×1	ReLU
Deconv.	$16 \times 10 \times 7$	$64 \times 12 \times 9$	64	3×3	1×1	ReLU
Deconv.	$64 \times 12 \times 9$	$64 \times 26 \times 20$	64	4×4	2×2	ReLU
Deconv.	$64 \times 26 \times 20$	$32 \times 108 \times 84$	32	8×8	4×4	ReLU
Deconv.	$32 \times 108 \times 84$	$4 \times 108 \times 84$	4	1×1	1×1	Sigmoid

Table 5.1: Layers of the autoencoder with key parameters

A deconvolutional layer is added at the end to reduce the number of channels back to the original four, without changing the width and height of the frames (i.e. using unitary filters and strides). All layers in the AE use the *Rectified Linear Unit* (ReLU) [29, 21] nonlinearity as activation function, except for the last layer which uses *sigmoids* to limit the activations values in the same $[0, 1]$ interval of the input (Figure 5.3). Details of the AE layers are summarized in Table 5.1.

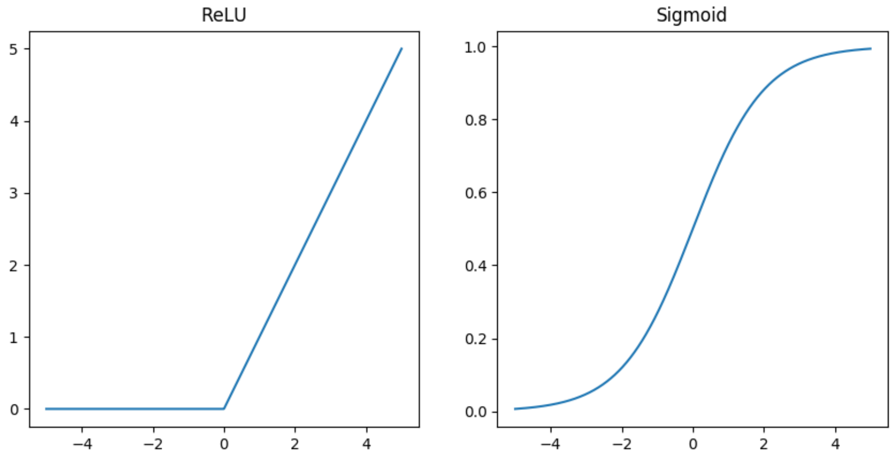


Figure 5.3: The ReLU and Sigmoid activation functions for the AE.

We train the AE with the Adam optimization algorithm [20] (see Table 5.2 for details on the hyperparameters) set to minimize the *binary crossentropy* loss defined as:

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{n=1}^{N} [y_n log(\hat{y}_n) + (1 - y_n) log(1 - \hat{y}_n)] \tag{5.4}$$

where y and $\hat{y}$ are vectors of N target and predicted observations in the state space. The dataset used for training is a subset of the dataset collected for the whole learning procedure described in Algorithm 6, namely the first elements of the four-tuples $(s, a, r, s') \in \mathcal{TS}$ (which are used as input and targets). We prevent *overfitting* of the training set by monitoring the performance of the AE on a held-out set of validation samples which amounts to roughly 10% of the training data, and stopping the procedure when the validation loss does not improve by at least 10^{-5} for five consecutive training epochs.

Parameter	Value
Learning rate	0.001
Batch size	32
Exponential decay rate (β_1)	0.9
Exponential decay rate (β_2)	0.999
Fuzz factor (ε)	10^{-8}

Table 5.2: Optimization hyperparameters for Adam

5.3 Tree-based Recursive Feature Selection

We use the RFS algorithm to reduce the state space representation computed by the AE down to the feature space $\hat{S}$. We base both the feature ranking method FR and the model $\hat{f}$, used for computing the descriptiveness of the features, on the supervised Extra-Trees algorithm (cf. Section 2.3.2).

The feature ranking approach with Extra-Trees is based on the idea of scoring each input feature by estimating the variance reduction produced anytime that the feature is selected during the tree building process. The ranking score is computed as the percentage of variance reduction achieved by each feature over the M trees built by the algorithm.

At the same time, Extra-Trees is a sufficiently powerful and computationally efficient supervised learning algorithm to use as $\hat{f}$. Note that in general FR and $\hat{f}$ could be different algorithms with different parametrizations, but Castelletti et al. use the same regressor for both tasks (as does the code implementation of RFS and IFS that we used for experiments), and we therefore complied with this choice. The parametrization of Extra-Trees for FR and $\hat{f}$ is reported in Tables 5.3[1] and 5.4, respectively for the parameters of the base trees used to build the ensemble and the Extra-Trees algorithm itself (cf. Section 2.3.2). We use the R^2 metric defined in Equation (4.3) to compute the ability of the selected features to describe the target, in K-fold cross validation over the training set $\mathcal{D}$. We compute a confidence interval over the scores of the validation predictions as:

Parameter	Value
Scoring method	Variance reduction
Max tree depth	None
n_{min} (node)	5
n_{min} (leaf)	2

Table 5.3: Parameters for base estimators in Extra-Trees (RFS)

Parameter	Value
M (Number of base estimators)	50
K (Number of randomly selected attributes)	All available attributes

Table 5.4: Parameters for Extra-Trees (RFS)

$$CI = \sqrt{\frac{1}{|\mathcal{D}|}\sum_{i=1}^{|\mathcal{D}|}(R^2)^2 \cdot (\frac{1}{|\mathcal{D}|}\sum_{i=1}^{|\mathcal{D}|}(R^2))^2} \tag{5.5}$$

and we adapt the stopping condition of RFS by setting $\epsilon = CI + CI_{old}$ (where the suffix *old* has the same meaning as in Algorithm 8) so that the condition becomes $R^2 - CI < R^2_{old} - CI_{old}$. We also multiply ϵ by a *significance* factor ξ in order to control the amount of variance that a feature must explain in order to be added to the selection: higher values of ξ means that the selection will yield a smaller subset composed exclusively of very informative features (even if the overall amount of variance explained is not necessarily the whole possible amount). The hyperparameters used for RFS are reported in Table 5.5.

5.4 Tree-based Fitted Q-Iteration

We use the FQI algorithm to learn an approximation of the Q function from the compressed and reduced state space extracted by the previous two modules. For consistency

[1] We use two different values for the minimum number of samples required to split an internal node or a leaf.

Parameter	Value
K (for K-fold cross-validation)	3
ξ (significance)	0.5

Table 5.5: Parameters for RFS

with the feature selection algorithm, we use the Extra-Trees learning method as function approximator for the action-value function. The model is trained to map the 1D compressed feature space $\hat{S}$ to the $|A|$-dimensional action-value space $\mathbb{R}^{|A|}$, and we use the action identifiers to select the single output value of our approximated $\hat{Q}$ function. The parametrization of the decision trees built for the ensemble is reported in Table 5.6 and the parametrization specific to Extra-Trees is reported in Table 5.7 (cf. Section 2.3.2).

Parameter	Value
Scoring method	Variance reduction
Max tree depth	None
n_{min} (node)	2
n_{min} (leaf)	1

Table 5.6: Parameters for base estimators in Extra-Trees (FQI)

Parameter	Value
M (Number of base estimators)	100
K (Number of randomly selected attributes)	All available attributes

Table 5.7: Parameters for Extra-Trees (FQI)

Since the FQI procedure introduces a small bias to the action-value estimate at each iteration (due to approximation error and a similar over-estimation issue to that described in Section 3.1 for Q-learning), we implement an *early stopping* procedure based on the evaluation of the agent's performance. At each iteration i, we evaluate the performance of an ε-greedy policy (with $\varepsilon = 0.05$) based on the current partial approximation $\hat{Q}_i$, by composing all the modules in the pipeline to obtain the full Q as per Equation (4.1). If the agent's performance does not improve for five consecutive iterations, we stop the training and produce the best performing estimation as output to the training phase. The performance is evaluated by running the policy for five episodes and averaging the clipped cumulative return of each evaluation episode.

Finally, we consider a discount factor $\gamma = 0.99$ for the MDP in order to give importance to rewards in a sufficiently large time frame.

5.5 Evaluation

An evaluation step is run after each training step of the full procedure. Similarly to what we do to evaluate the agent's performance during the training of FQI, here we use an ε-greedy policy with $\varepsilon = 0.05$ based on the full Q composition of Equation (4.1). The reason for using a non-zero exploration rate during evaluation is that ALE provides a fully deterministic initial state for each episode, and by using a deterministic policy we would always observe the same trajectories (thus leading to overfitting to the best policy from the initial state, rather than a generic good policy from any state). Using a non-zero exploration rate allows us to assess the agent's capability of playing effectively in any state of the game, and of correcting its behavior after an (albeit artificial) mistake.

We let the agent experience N separate episodes under the ε-greedy policy, and for each episode we consider the cumulative clipped return and the number of steps occurred. The mean and variance of these two metrics across the N episodes provide us with insights on the agent's performance: a high average return obviously means that the algorithm has produced a good policy, but at the same time a low variance in the number of steps could indicate that the agent is stuck in some trivial policy (e.g. take always the same action) which causes the episodes to be essentially identical, even accounting for the non-zero exploration rate. The latter aspect, while not negative in general, can help in the initial steps of experiments to detect potential problems in the implementation.

Evaluation parameters are summarized in Table 5.8.

Parameter	Value
Exploration rate ε	0.05
N	10

Table 5.8: Parameters for evaluation

Chapter 6

Experimental Results

In this chapter we show experimental results of our algorithm on some Atari games. We show that the representation of the states learned by our feature extraction pipeline is suitable for the semi-batch control approach with FQI, and that our agent is able to learn non-trivial policies with good sample efficiency on different Atari games.

6.1 Premise

Our algorithm is based on optimizing sample efficiency, but does not account for the computational complexity of training and evaluating the full pipeline. All three main components of the algorithm benefit a great deal from parallel computing, to the point where the hardware itself is a key component in making the algorithm treatable at all. We were thus strongly limited in our experimental phase by our access to hardware for heavy parallel computing, with even costs for renting servers on cloud providers like *Amazon AWS* quickly surpassing the hundreds of Euros per experiment. We therefore had to be extremely selective in what experiments to run, each of which took days at a time, and well balanced between exploring new possibilities (hyperparameters, network architectures, environments) and evaluating current configurations. In this chapter we report the best findings across all our experiments and we try to outline the thought process that led us to favor some choices over others. The hardware configurations of the three servers that we used for experiments is reported in Table 6.1.

Server ID	vCPU count	GPU	RAM (GB)
#1	64	None	252
#2	8	1 x Nvidia Tesla K40C	32
#3	4	1 x Nvidia Tesla K40C	32

Table 6.1: Server configurations for experiments

The code implementation of the algorithm was based on Python 2.7 and its standard libraries for general machine learning, GPU-based deep learning, and linear algebra. We trained and used the AE on an *Nvidia Tesla K40C* GPU, using the *Keras 2.0.6* API as interface to the *Tensorflow 1.3.0* library for deep learning, whereas the training and evaluation of Extra-Trees for RFS and FQI was run on CPU using the *Scikit-Learn 0.19.0* library. We used the implementations of FQI and RFS provided in the open source *IFQI* library by Matteo Pirotta. Finally data manipulation and any other algebraic operations were done with *Numpy 1.13.1*.

Chapter 7

Conclusions and Future Developments

Bibliography

[1] Oron Anschel, Nir Baram, and Nahum Shimkin. Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning. In *International Conference on Machine Learning*, pages 176–185, 2017.

[2] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013.

[3] Dimitri P. Bertsekas and John N. Tsitsiklis. Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE, 1995.

[4] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[5] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[6] Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[7] Andrea Castelletti, Stefano Galelli, Marcello Restelli, and Rodolfo Soncini-Sessa. Tree-based variable selection for dimensionality reduction of large-scale control systems. In *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2011 IEEE Symposium on*, pages 62–69. IEEE, 2011.

[8] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[9] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

[10] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(Apr):503–556, 2005.

[11] S. Galelli and A. Castelletti. Tree-based iterative input variable selection for hydrological modeling. *Water Resources Research*, 49(7):4295–4310, 2013.

[12] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012.

[13] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[14] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[15] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, AdriÃ Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[16] A. Harutyunyan, M. G. Bellemare, T. Stepleton, and R. Munos. Q (λ) with off-policy corrections. In *International Conference on Algorithmic Learning Theory*, pages 305–320. Springer, 2016.

[17] D. Hassabis and D. Silver. Alphago's next move (deepmind.com/blog/alphagos-next-move/), 2017.

[18] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[19] F. S. He, Y. Liu, A. G. Schwing, and J. Peng. Learning to play in a day: Faster deep reinforcement learning by optimality tightening. In *International Conference on Learning Representations (ICLR)*, 2017.

[20] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[22] Sascha Lange and Martin Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8. IEEE, 2010.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[24] Long-H Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3/4):69–97, 1992.

[25] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. In *Artificial Neural Networks and Machine Learning – ICANN 2011: 21st International Conference on Artificial Neural Networks, Espoo, Finland, June 14-17, 2011, Proceedings, Part I*, pages 52–59, 2011.

[26] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Dan Wiestra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–33, 2015.

[28] R. Munos, T. Stepleton, A. Harutyunyan, and M. Bellemare. Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1054–1062, 2016.

[29] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[30] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. In *Advances in Neural Information Processing Systems*, pages 4026–4034, 2016.

[31] Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adrià Puigdomènech, Oriol Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. Neural episodic control. 2017.

[32] Gavin A. Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering, 1994.

[33] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In *International Conference on Learning Representations (ICLR)*, 2016.

[34] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[35] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[36] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[37] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[38] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[39] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. In *9th ISCA Speech Synthesis Workshop*, pages 125–125.

[40] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[41] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Duel- ing network architectures for deep reinforcement learning. In *International Conference on Machine Learning (ICML).*, 2016.

[42] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[43] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol

Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. 2016.

[44] Matthew D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[45] Matthew D. Zeiler, Dilip Krishnan, Graham W. Taylor, and Rob Fergus. Deconvolutional networks. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2528–2535. IEEE, 2010.