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, Matteo Pirotta, Ph.D.

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

Academic Year 2016-2017

Contents

1 Background **3**

1.1 Deep Learning . . . 3

1.1.1 Artificial Neural Networks . . . 4

1.1.2 Backpropagation . . . 5

1.1.3 Convolutional Neural Networks . . . 8

1.1.4 Autoencoders . . . 9

1.2 Reinforcement Learning . . . 11

1.2.1 Markov Decision Processes . . . 12

1.2.2 Optimal Value Functions . . . 15

1.2.3 Value-based optimization . . . 16

1.2.4 Dynamic Programming . . . 17

1.2.5 Monte Carlo Methods . . . 18

1.2.6 Temporal Difference Learning . . . 19

1.2.7 Fitted Q-Iteration . . . 21

2 State Of The Art **23**

2.1 Value-based Deep Reinforcement Learning . . . 23

2.2 Other approaches . . . 26

2.2.1 Memory architectures . . . 26

2.2.2 AlphaGo . . . 27

2.2.3 Asynchronous Advantage Actor-Critic . . . 28

2.3 Related Work . . . 29

References **31**

List of Figures

1.1 Graphical representation of the perceptron model 4
1.2 A neural network with one hidden layer 5
1.3 Linear separability with perceptron 6
1.4 Visualization of SGD on a space of two parameters 7
1.5 Effect of the learning rate on SGD updates 7
1.6 CNN for image processing 8
1.7 Shared weights in CNN 9
1.8 Max pooling 9
1.9 Schematic view of an autoencoder 10
1.10 Reinforcement learning setting 11
1.11 Graph representation of an MDP 13
1.12 Policy iteration 17

2.1 Some of the games available in the Atari environments 25
2.2 Architecture of NEC 26
2.3 Neural network training pipeline of AlphaGo 27
2.4 The asynchronous architecture of A3C 28

List of Tables

List of Algorithms

1 SARSA 20
2 Q-Learning 21
3 Fitted Q-Iteration 22
4 Deep Q-Learning with Experience Replay 24

Chapter 1

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.

1.1 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 [18].

Deep learning is at the heart of modern machine learning research, where deep models have revolutionized many fields like computer vision [16, 31], machine translation [36] and speech synthesis [33]. 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.

1.1.1 Artificial Neural Networks

Feed-forward Artificial Neural Networks (ANNs) [3] are universal function approximators inspired by the connected structure of neurons and synapses in biological brains.

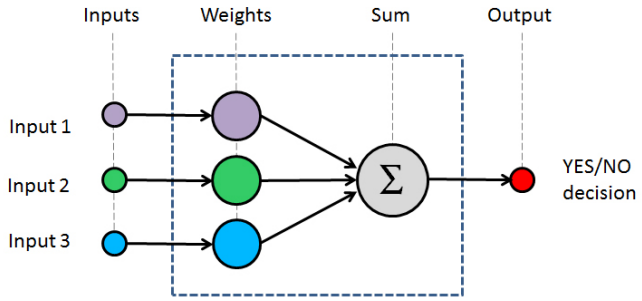

Figure 1.1: Graphical representation of the perceptron model

ANNs are based on a fairly simple computational model called *perceptron* (Figure 1.1), which is a transformation of an n-space into a scalar value

$$z = \sum_{i=1}^{n} (w_i \cdot x_i) + b \tag{1.1}$$

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 1.2):

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:

$$z^{(i)} = W^{(i)} \cdot \sigma(z^{(i-1)}) + B^{(i)} \tag{1.2}$$

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.

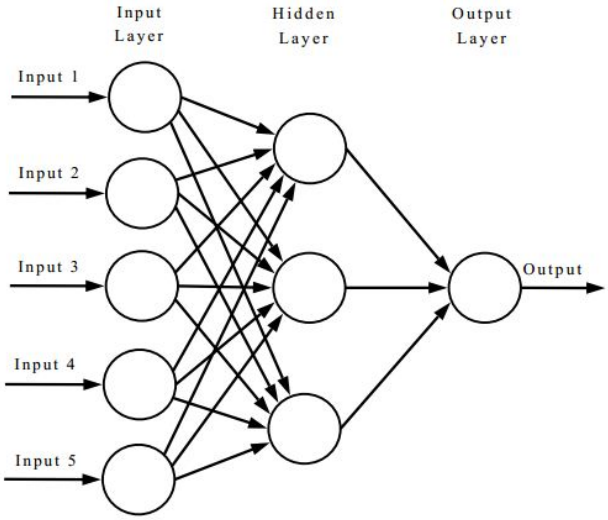

Figure 1.2: A neural network with one hidden layer

1.1.2 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*:

$$w_i^{new} = w_i^{old} - \eta(\hat{y} - y)x_i, \forall i = (1, ..., n) \tag{1.3}$$

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 1.3.

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

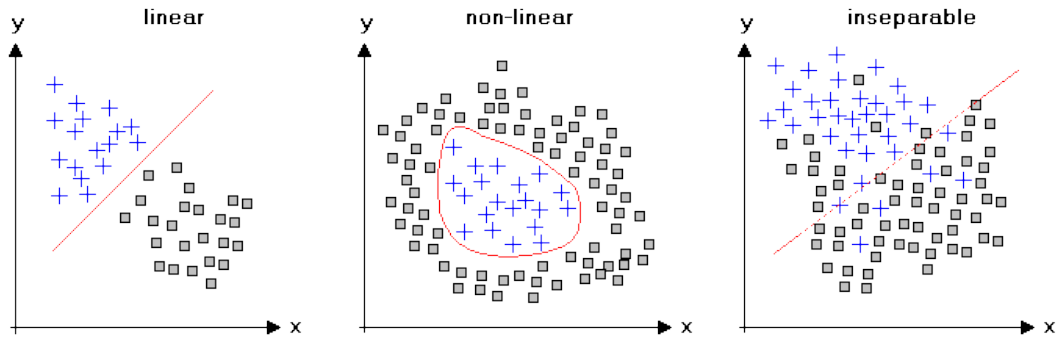

Figure 1.3: Different 2D classification problems, respectively linearly, non-linearly and non separable. The perceptron would be able to converge and correctly classify the points only in the first setting

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:

$$W_i^{new} = W_i^{old} - \eta \frac{\partial L(y, \hat{y})}{\partial W_i^{old}} \tag{1.4}$$

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 is to use *stochastic gradient descent* (SGD) [3], 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 1.5).

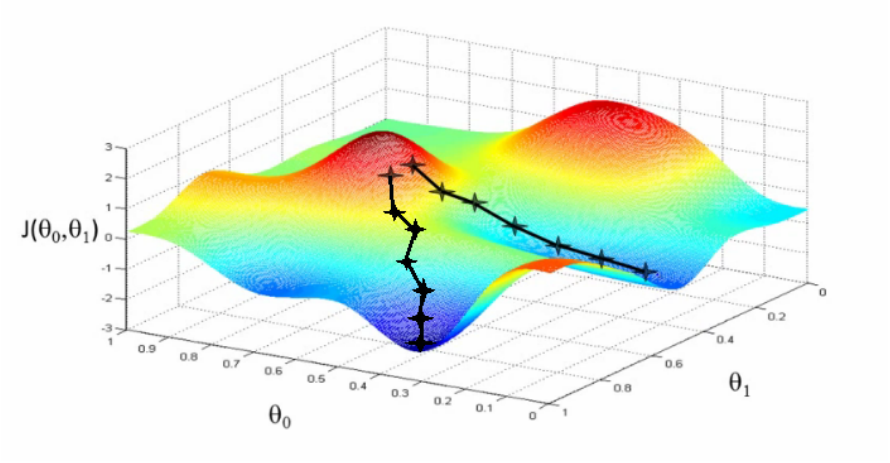

Figure 1.4: Visualization of SGD on a space of two parameters

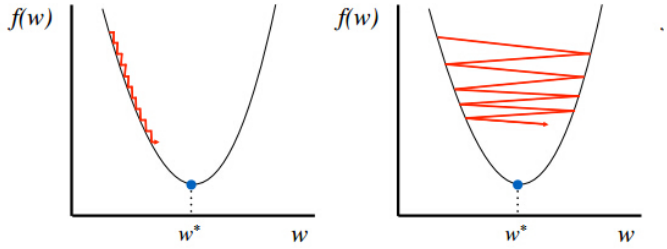

Figure 1.5: Effect of the learning rate on SGD updates. Too small (left) may take longer to converge, too big (right) may overshoot the optimum and even diverge

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:

$$W_i^{(j+1)} = W_i^{(j)} - \gamma\eta \frac{\partial L(y^{(j-1)}, \hat{y}^{(j-1)})}{\partial W_i^{(j-1)}} - \eta \frac{\partial L(y^{(j)}, \hat{y}^{(j)})}{\partial W_i^{(j)}} \tag{1.5}$$

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* [6] and *Adadelta* [37] optimization algorithms), and a similar approach which uses an adaptive momentum term (called *Adam* [15]).

1.1.3 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).

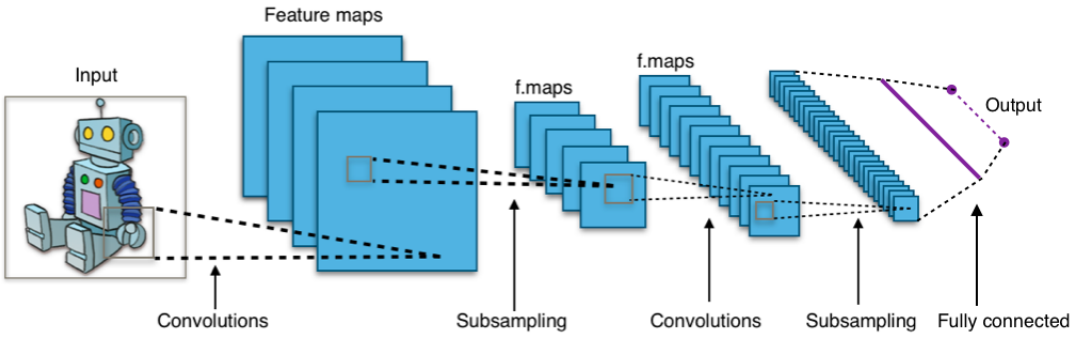

Figure 1.6: Typical structure of a deep convolutional neural network for image processing, with two convolutional hidden layers and a dense section at the end (for classification or regression)

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 1.7. 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, 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.

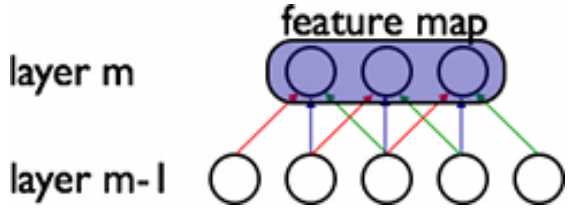

Figure 1.7: simple representation of shared weights in a 1D CNN. Each neuron in the second layer applies the same receptive field of three weights to three adjacent neurons of the previous layer. The filter is applied with a stride of one element to produce the feature map

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 1.8).

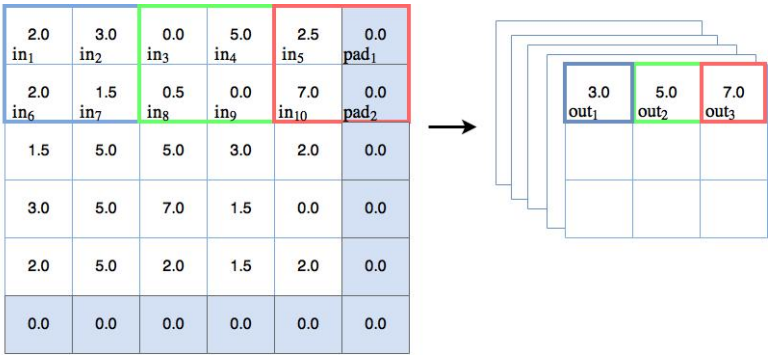

Figure 1.8: Example of max pooling, where only the highest activation value in the pooling window is kept

Finally, typical applications of CNNs in the literature use mixed architectures composed of both convolutional and fully connected layers. In tasks like image classification [29, 31], 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 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.

1.1.4 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 1.9). 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.

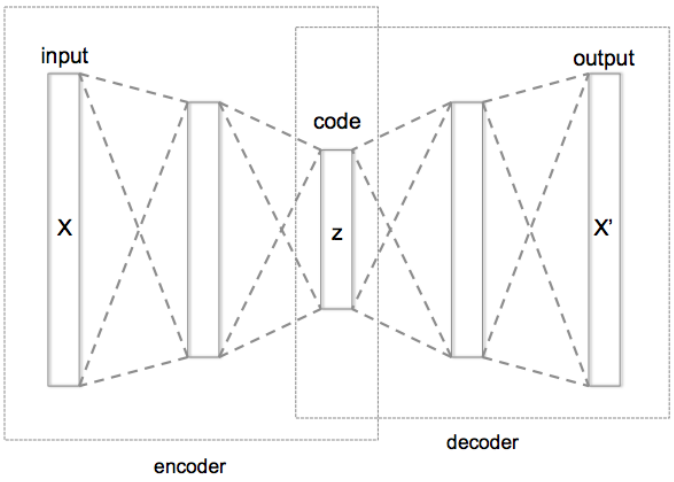

Figure 1.9: Schematic view of an autoencoder, with the two main components highlighted

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 1.1.3 [7]. 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) [20]. This approach in building the decoder, however, can sometimes cause blurry or inaccurate reconstructions due to the upscaling operation which simply replicates in-

formation 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 [38]. 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.

1.2 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.

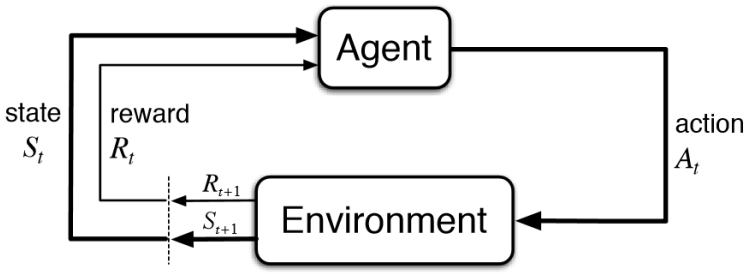

Figure 1.10: The reinforcement learning setting, with the agent performing actions on the environment and in turn observing the state and reward

[1] Or *transposed convolutions*.

1.2.1 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 1.10).

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 the exploration of the environment in a given state.

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 $s_n \in S^T$, 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.

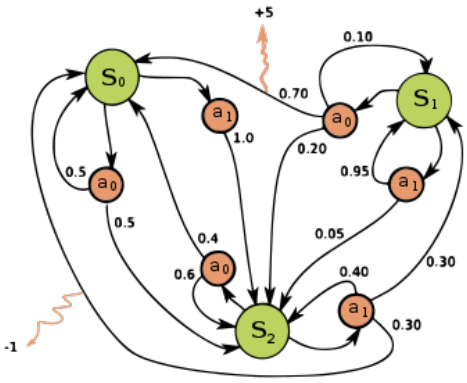

Figure 1.11: Graph representation of an MDP. Each node represents a state, each arc is a transition from a state to another; note that actions may have probability distributions associated to the following states

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:

$$V^\pi(s) = E_\pi[R_t|s_t = s] \tag{1.6}$$

$$= E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s] \tag{1.7}$$

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:

$$Q^\pi(s, a) = E_\pi[R_t|s_t = s, a_t = a] \tag{1.8}$$

$$= E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a] \tag{1.9}$$

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:

$$\begin{aligned} V^\pi(s) &= E_\pi[R_t|s_t = s] \\ &= E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s] \\ &= E_\pi[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_t = s] && (1.10) \\ &= \sum_{a\in A} \pi(s, a) \sum_{s'\in S} P(s, a, s')[R(s, a) + \\ &\quad + \gamma E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_{t+1} = s']] && (1.11) \\ &= \sum_{a\in A} \pi(s, a) \sum_{s'\in S} P(s, a, s')[R(s, a) + \gamma V^\pi(s')] && (1.12) \end{aligned}$$

Intuitively, relation (1.12) 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.

1.2.2 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 finite 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 1.2.3).

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

$$V^*(s) = \max_{\pi} V^{\pi}(s), \forall s \in S \tag{1.13}$$

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^*:

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) \tag{1.14}$$

$$= E[r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a] \tag{1.15}$$

Notice that equivalence (1.15) 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 (1.12), 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

[2] We make this clarification for formality, but we do not expand the details further in this work. Refer to [30] for more details on the subjet of non-finite MDPs.

that the agent can take in that state:

$$V^*(s) = \max_a Q^*(s,a) \tag{1.16}$$
$$= \max_a E_{\pi^*}[R_t|s_t = s, a_t = a] \tag{1.17}$$
$$= \max_a E_{\pi^*}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a] \tag{1.18}$$
$$= \max_a E_{\pi^*}[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_t + k + 2|s_t = s, a_t = a] \tag{1.19}$$
$$= \max_a E_{\pi^*}[r_{t+1} + \gamma V^*(s_{t+1})|s_t = s, a_t = a] \tag{1.20}$$
$$= \max_a \sum_{s' \in S} P(s,a,s')[R(s,a) + \gamma V^*(s')] \tag{1.21}$$

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

$$Q^*(s,a) = E[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a')||s_t = s, a_t = a] \tag{1.22}$$
$$= \sum_{s'} P(s,a,s')[R(s,a) + \gamma \max_{a'} Q^*(s', a')] \tag{1.23}$$

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.

1.2.3 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 (1.16). 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 Section **??** and Chapter 2).

1.2.4 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

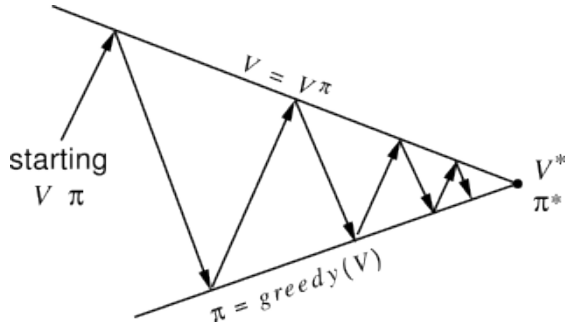

Figure 1.12: Classical representation of the policy iteration algorithm, which highlights the relation between policies and their associated value functions. Each pair of arrows starting from a policy and ending on a greedy policy based on the value function is a step of the algorithm

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 1.12), such that:

$$\pi'(s) = \arg\max_{a \in A} Q^{\pi}(s, a) \tag{1.24}$$

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

$$Q^{\pi}(s, \pi'(s)) = \max_{a \in A} Q^{\pi}(s, a) \geq Q^{\pi}(s, \pi(s)) = V^{\pi}(s) \tag{1.25}$$

This continuous improvement is applied until the inequality in (1.25) becomes an equality, i.e. until the improved policy satisfies the Bellman optimality equation (1.16). 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*:

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} P(s, a, s')[R(s, a) + \gamma V(s')] \tag{1.26}$$

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.

1.2.5 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 (1.24) 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.

1.2.6 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:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{1.27}$$

In general, TD methods have several advantages over MC as they allow for an *on-line* (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).

[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.

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:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{1.28}$$

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

Algorithm 1 SARSA

```
Initialize Q(s, a) arbitrarily
Initialize π as some function of Q (e.g. greedy)
repeat
  Initialize s
  Choose a from s using π
  repeat
    Take action a, observe r, s'
    Choose a' from s' using π
    Update Q(s, a) using rule (1.28)
    if π is time-variant then
      Update π towards greediness
    end if
    s ← s'; a ← a'
  until s is terminal or Q did not change
until training ended or Q did not change
```

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

Defined by Sutton and Barto [30] 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

[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.

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:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{1.29}$$

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 π arbitrarily
repeat
  Initialize s
  repeat
    Choose a from s' using π
    Take action a, observe r, s'
    Update Q(s, a) using rule (1.29)
    s ← s'
  until s is terminal or Q did not change
until training ended or Q did not change
```

1.2.7 Fitted Q-Iteration

Having introduced a more classic set of traditional RL algorithms in the previous sections, we now present a more modern approach 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 approach 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) approach 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 1.2.4, 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 π; 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.

Chapter 2

State Of The Art

The integration of RL and neural networks has a long history. Early RL literature [26, 32, 2] 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 [5] 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.

2.1 Value-based Deep Reinforcement Learning

In 2015, Mnih et al. [22] 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 *hyperparameter 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*.

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

$$\frac{\partial L}{\partial W_i^{old}} = E[(r + \gamma \max_{a'} Q(s', a'; \theta') - Q(s, a)) \frac{\partial Q(s, a; \theta)}{\partial W_i^{old}}] \tag{2.1}$$

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* [19] 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 4.

Algorithm 4 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 θ, θ'
for *episode* $= 1, M$ **do**
  for $t = 1, T$ **do**
    Select a random action a_t with probability ε
    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 θ
    Every C steps, set $\theta' = \theta$
  end for
end for

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) [34] 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 (1.29)). This general issue was addressed by Van Hasselt (2010) with *Double Q-learning* [13], a learning algorithm which keeps two separate estimates of the action-value function

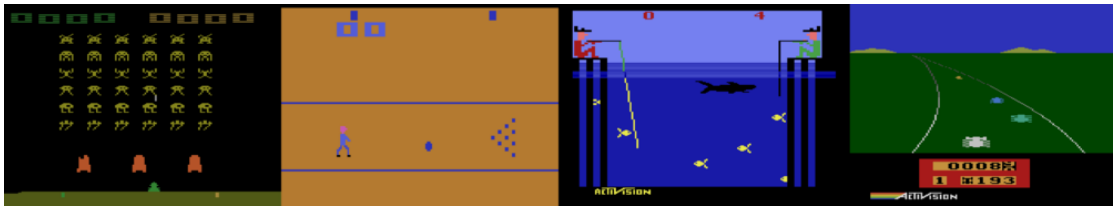

Figure 2.1: Some of the games available in the Atari environments

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

$$Q^A(s,a) \leftarrow Q^A(s,a) + \alpha[r + \gamma Q^B(s', \arg\max_a Q^A(s',a)) - Q^A(s,a)] \tag{2.2}$$

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:

$$y_j = \begin{cases} r_{j+1}, & \text{if } s_{j+1} \text{ is terminal} \\ r_{j+1} + \gamma Q(s_{j+1}, \arg\max_a Q(s_{j+1}, a; \theta); \theta'), & \text{otherwise} \end{cases} \tag{2.3}$$

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) [27] 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* [35], 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:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{2.4}$$

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]:

$$Q^\pi(s,a) = V^\pi(s) + (A^\pi(s,a) - \max_{a'} A^\pi(s,a')) \tag{2.5}$$

[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...).

Several other extensions of the DQN algorithm have been proposed in recent years. Among these, we cite Osband et al. (2016) [24] 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) [14] 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) [23] and Harutyunyan et al. (2016) [11] 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.

2.2 Other approaches

2.2.1 Memory architectures

Graves et al. (2016) [10] 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* [9] on a variety of query-answering and natural language processing tasks, and was used to solve a simple *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).

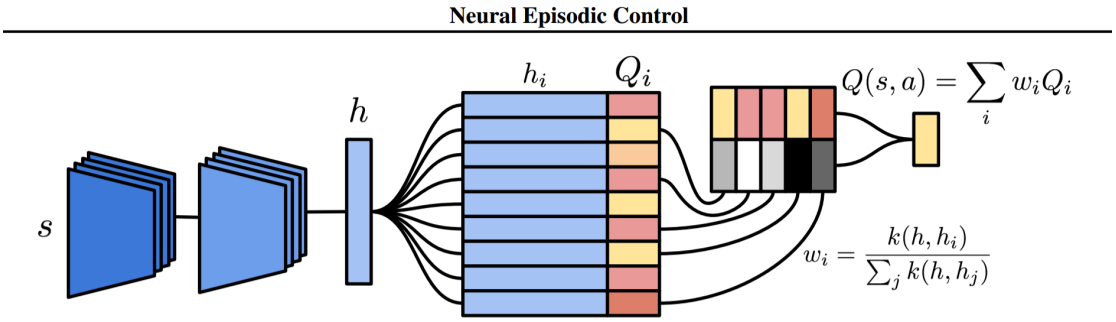

Figure 2.2: Architecture of NEC

Pritzel et al. (2017) [25] 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 2.2). 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.

2.2.2 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.

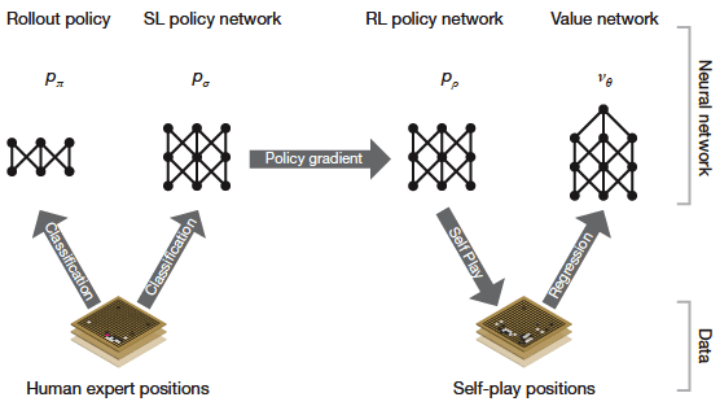

Figure 2.3: Neural network training pipeline of AlphaGo

Silver et al. (2016) [28] 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

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

(the company behind AlphaGo) decided to retire the program from official competitions and released a dataset containing 50 self-play games [12].

AlphaGo is a complex architecture which combines deep CNNs, reinforcement learning, and Monte Carlo Tree Search (MCTS) [4, 8]. 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 2.3). 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.

2.2.3 Asynchronous Advantage Actor-Critic

Actor-critic algorithms [30] are TD 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. Mnih et al.

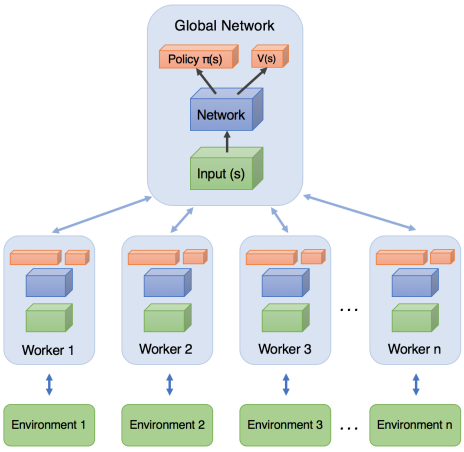

Figure 2.4: The asynchronous architecture of A3C

(2016) [21] 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 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.

2.3 Related Work

Lange and Riedmiller (2010) [17] 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*) with good sample efficiency.

Bibliography

[1] Oron Anschel, Nir Baram, and Nahum Shimkin. Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning.

[2] 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.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] Alex Graves and Greg Wayne. Neural turing machines. 2014.

[10] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[11] 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.

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

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

[14] 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.

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

[16] 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.

[17] 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.

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

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

[20] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. *Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction*, pages 52–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[21] 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.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–33, 2015.

[23] 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.

[24] 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.

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

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

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

[28] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

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

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

[31] 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.

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

[33] 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.

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

[35] 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.

[36] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. 2016.

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

[38] 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.