POLITECNICO DI MILANO
Master's Degree in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

FAST REINFORCEMENT LEARNING USING DEEP STATE-ACTION FEATURES

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

Supervisor: Prof. Marcello Restelli
Co-supervisor: Matteo Pirotta, Ph.D.

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

Academic Year 2016-2017

Contents

Chapter 1

Background

Introduction to the section...

1.1 Deep Learning

Deep Learning (DL) is a branch of machine learning which exploits *Artificial Neural Networks* (ANN) with more than one hidden layer to learn an abstract representation of the input space [1].

Deep learning techniques can be applied to the three main classes of problems of machine learning (supervised, semi-supervised, and unsupervised), and have been used to achieve state-of-the-art results in a variety of learning tasks.

Deep learning is closely inspired by the general structure of the animal brain, made up by neurons and synapses; for this, it is often referred to as a *connectionist* approach.

The field of DL sees its origins in the early 1940s, with algorithms which mimicked the neural process being proposed by [Pitts and McCulloch]. Progress in the field was characterized by slow progress due to the great computational complexity of backpropagation, and the history of deep learning is marked by two period of complete lack of research called the *AI winters*, one in the 1970s and one in the late 1980s.

Eventually, with the advent of the information era and the large scale distribution of computing power, deep learning algorithms started to become more and more tractable, with the definitive explosion in DL research arriving in the early 2010s and fueled by the power of *Graphical Processing Units* (GPU).

In this section we give a brief overview of the basic concept behind DL and introduce the principal ideas that will be used in later sections of this thesis.

1.1.1 Artificial Neural Networks

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

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

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

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 input 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:

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:

$$z^{(i)} = W^{(i)} \cdot \sigma(z^{(i-1)})$$

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

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.

1.1.2 Backpropagation

Having defined the most basic architecture used in DL, we now present the methodology to use it in a learning setting. This is not meant to be an exhaustive description of the learning process in neural networks, so we invite the reader to see [Bishop] for more details on the subject.

Deep learning 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 set of weight matrices $W^{(i)}$ 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 a training set according to the following *update rule*:

$$w_i^{new} = w_i^{old} - \eta(\hat{y} - y)x_i, \forall i = (1, ..., n) \tag{1.1}$$

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).

The simple update rule of the perceptron 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 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 update rule for the weights of a layer is:

$$W_i^{new} = W_i^{old} - \eta \frac{\partial E}{\partial W_i^{old}} \tag{1.2}$$

where E is a differentiable function of the target and predicted values that quantifies the error made by the model on the training samples.

Notice that the loss can be computed over any number of training samples, to perform what is called a *batch update*.

1.1.3 Convolutional Neural Networks

Convolutional Neural Networks (CNN) 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 through the use of 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 the way in which a signal is *strided* across the other during the convolution operation.

The result of this operation is to obtain a nonlinear transformation of the input space of a layer into a new space (of compatible dimensions) which preserves the spatial correlation of the information coming from the input. This transformation therefore acts as an abstraction of the spatial information encoded in the original input, while maintaining the underlying structure which is proper of the data (e.g. form the $3 \times n \times m$ pixels of an RGB image to a $j \times k$ matrix that represents those groups of pixels in which there is an edge).

Differently from ANNs (which are sometimes referred to as *fully connected* or *dense* networks, 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. This 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 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 case of 2D images for which a 3-channel representation is used for red, green, and blue). In this case, there will exist a set of shared weights for each filter.

When a set of feature maps is given as input to a convolutional layer then it is usually treated as a single input, with multidimensional filters strided simultaneously across all channels to produce the following set of feature maps.

At the same time, while it may be useful to have different abstractions of the same input (which effectively enlarges the output space of the layers), it is also necessary to force a reduction of the input representation in order to learn useful transformations that encode the spatial information in an abstract way.

For this reason, convolutional layers in CNN 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).

Finally, mixed architectures which make use of both convolutional and fully connected layers are often found in the literature. This applies, for instance, to tasks like image classification, for which convolutional layers are used to extract significant fea-

tures 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* the weights and the 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*. 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. However, the encoder's usual function is to reduce the dimensionality of the input, whereas the decoder performs the inverse transformation, using the encoded representation to rebuild the image which generated it.

Autoencoders are typically designed as sequences of hidden layers for which the output's dimensionality is lower that the input's in the first half of the network, with the innermost hidden layer producing the smallest representation of the input before inverting the process and going back to the original representation. 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 added to the loss associated to the layer).

Autoencoders can be especially effective when one desires to extract meaningful features from the input space, without tailoring the features to a specific problem (like for the end-to-end image classification example in 1.1.3).

AEs have proven particularly effective in the extraction of features from images, where convolutional layers are used in the encoder to obtain a natural reduction of the inputs. In this case the decoder would still use convolutional layers to apply a non linear transformation, but the expansion of the compressed feature space would be delegated to *upscaling layers* (the opposite of pooling layers).

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 case are called *deconvolutional*[1] and are thoroughly presented in [??]. For the purpose of this thesis it suffices to notice that image reconstruction with this type of layer is incredibly more accurate in reconstructing even the smallest details of the input (down to a single pixel).

1.2 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning which is centered on optimizing the behavior of an agent in an environment 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 due to 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 in turn is affected by the decisions, with a scalar reward signal representing 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 perfectly suitable to describe, at least partly, the complex life of living creatures.

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.

1.2.1 Markov Decision Processes

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

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 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. A sequence of states observed by an agent during an interaction with the environment and ending in a terminal state is usually called an *episode*.

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

[1] Or *transposed convolutions*

- $P : S \times A \times S \to [0, 1]$ 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* with which the rewards collected by the agent are diminished at each step, and can be interpreted as the agent's interest for rewards further in time rather than immediately.

- $\mu : S \to [0, 1]$ 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 $(s_i, a_i, r_{i+1}, s_{i+1})$ represent 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}, and $s_n \in S^T$.

In Markov Decision Processes 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, rather than 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.

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:

$$V^\pi(s) = E_\pi[R_t|s_t = s] \tag{1.3}$$

$$= E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s] \tag{1.4}$$

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:

$$Q^\pi(s, a) = E_\pi[R_t|s_t = s, a_t = a] \tag{1.5}$$

$$= E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a] \tag{1.6}$$

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:

$$\begin{aligned} V^\pi(s) &= E_\pi[R_t|s_t = s] \\ &= E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s] \\ &= E_\pi[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_t = s] \qquad (1.7) \\ &= \sum_{a\in A} \pi(s, a) \sum_{s'\in S} P(s, a, s')[R(s, a) + \\ &\quad + \gamma E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_{t+1} = s']] \qquad (1.8) \\ &= \sum_{a\in A} \pi(s, a) \sum_{s'\in S} P(s, a, s')[R(s, a) + \gamma V^\pi(s')] \qquad (1.9) \end{aligned}$$

Intuitively, relation (1.9) 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, *to solve* a reinforcement learning task is to identify 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 function defines 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.

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

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

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^*:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{1.10}$$
$$= E[r_{t+1} + \gamma V^*(s_{t+1}|s_t = s, a_t = a] \tag{1.11}$$

Notice that equivalence (1.11) 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 (1.9), 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

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

agent can take in that state:

$$V^*(s) = \max_a Q^*(s,a) \tag{1.12}$$
$$= \max_a E_{\pi^*}[R_t|s_t = s, a_t = a] \tag{1.13}$$
$$= \max_a E_{\pi^*}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a] \tag{1.14}$$
$$= \max_a E_{\pi^*}[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_t + k + 2|s_t = s, a_t = a] \tag{1.15}$$
$$= \max_a E_{\pi^*}[r_{t+1} + \gamma V^*(s_{t+1})|s_t = s, a_t = a] \tag{1.16}$$
$$= \max_a \sum_{s' \in S} P(s,a,s')[R(s,a) + \gamma V^*(s')] \tag{1.17}$$

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

$$Q^*(s,a) = E[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a')||s_t = s, a_t = a] \tag{1.18}$$
$$= \sum_{s'} P(s,a,s')[R(s,a) + \gamma \max_{a'} Q^*(s', a')] \tag{1.19}$$

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 trying to compute 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 policy:

- Given V^*, for each state $s \in S$ there will be an action (or actions) which maximizes the Bellman optimality equation (1.12). 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 *goodness* of actions, without performing the one-step search on the successor states.

In this section we will describe some of the most important value-based approaches to reinforcement learning, which will be useful in the following sections of this work.

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 deep learning to solve complex environments (see 1.3 and **??**).

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 any initial policy π until convergence to π^*.

In this class of algorithms, we identify two main approaches: *policy iteration* and *value iteration*.

Policy iteration

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 functions associated to the current policy, and then improving that policy by making it act greedily with respect to the value functions, such that:

$$\pi'(s) = \arg\max_{a \in A} Q^{\pi}(s, a) \tag{1.20}$$

which, for Theorem 1, improves the expected return of the policy because:

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

.

This continuous improvement is applied until the inequality in the previous equation becomes an equality, i.e. until the improved policy satisfies the Bellman optimality equation (1.12). 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*:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} P(s, a, s')[R(s, a) + \gamma V(s')] \tag{1.21}$$

Similarly to policy iteration, convergence is ensured but 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 an 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 in 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 centered 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 (1.20) 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.

[3]This is not always true: a variation of this algorithm exists, which only considers the average returns following the *first* visit to a state in each episode.

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:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{1.22}$$

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 problem 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.

[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.

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:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{1.23}$$

The training procedure of SARSA can be summarized with the following algorithm:

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 (1.23)
    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 $\epsilon = 1/t$).

Q-learning

Defined by [Sutton, Barto] 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:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{1.24}$$

As we did for SARSA, an algorithmic description of the Q-learning algorithm is:

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 (1.24)
        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 work, 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, 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 (admittedly ignoring the differences that are obviously present between the various methods), because it will be used in later sections of this thesis as a key component of the methodology being presented.

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 algorithm is described in detail as follows:

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.

1.3 Deep Reinforcement Learning

Bibliography

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