Subgoal-Oriented deep reinforcement learning

Alemán González, Jorge

Curs 2017-2018

Director: Anders Jonsson

GRAU EN ENGINYERIA INFORMÁTICA

Universitat Pompeu Fabra Barcelona
Escola Superior Politècnica

Treball de Fi de Grau

Subgoal-Oriented deep reinforcement learning

Jorge Alemán González

Supervised by Anders Jonsson

June 9, 2018

Bachelor in Computer Science

Department of Information and Communication Technologies

Acknowledgements

I wish to offer my thanks:

To my supervisor Anders Jonsson, for the guidance through the fascinating field of reinforcement learning and the patience shown to teach me all those new concepts.

To Miquel Juyent, for teaching me how the neural networks work and the state of the art methods of deep reinforcement learning. I also want to thank him for sharing the source code implementing Asynchronous Advantage Actor-Critic and helping me developing the new algorithm.

To my parents, for being an economic support and letting me live five hundred kilometers away from home. In particular to my mother, for the moral support in those days which i thought i will never end my thesis.

Abstract

State-of-the-art methods for deep reinforcement learning have demonstrated the ability to learn complex game strategies. However, they perform poorly in environments with sparse rewards or with the facility to die. In both cases the algorithms are not able to learn from low probability rewards. One of the algorithms that present this problem is Asynchronous Advantage Actor-Critic (A3C) (Mnih et al., 2016).

For this project it has been developed an algorithm called Multiple Asynchronous Advantage Actor-Critic (MA3C) which uses transfer learning and hierarchial learning to take advantage of subgoals. The algorithm has been tested in three different environments, including Montezuma's Revenge, that presents both difficulties. The experiments demonstrate that MA3C explores better and learns more quickly just by using a few domain knowledge.

The proposed method is very flexible and generalizes to any environment where multiple subgoals can be defined. Future research could be focused in the modification of this algorithm to automatically discover subgoals, avoiding the usage of domain knowledge.

Contents

Abbreviations

A3C Asynchronous Advantage Actor-Critic

AC Actor Critic

AI Artificial Intelligence

ANN Artificial Neural Network

CNN Convolutional Neural Network

DL Deep Learning

DQN Deep Q-Network

MA3C Multiple Asynchronous Advantage Actor-Critic

MDP Markov Decision Process

ML Machine Learning

MR Montezuma's Revenge

ReLU rectified linear unit

RL Reinforcement Learning

SDP Sequential Decision Process

SMDP Semi-Markov Decision Process

Chapter 1

Motivation

Nature has the ability of evolving in all kinds of adverse environments. One of the consequences of this continuous evolving is us, human beings, the maximum expression of adaptive living being. By using our intelligence we improve year by year the chances of survival to this chaotic world. This kind of intelligence is composed by several cognitive abilities: learning, forming concepts, understanding, applying logic and reasoning.

Since creation of Artificial Intelligence (AI) investigators from all around the world have tried to grant machines some of this skills. One way of letting machines exhibit intelligence is by modeling how the human brain works.

The brain is composed by billions of neurons that interact with each other to create intelligence. This system is named Artificial Neural Network (ANN) and scientists have learned lots of things about how it works, but not everything. Mathematicians used this knowledge to develop an artificial version of this network called Artificial Neural Network (ANN), which models neural networks based on mathematical functions.

But, why it is important for us to have intelligent machines?. Because by teaching computers to do some tasks they will help us reducing the amount of work. We can create intelligent tools to do our work better or to make our lives easier.

In the last years an area of AI called Reinforcement Learning (RL) has become really popular. In this field, algorithms learn how to solve any kind of tasks that has a numeric reward attached. There are plenty of algorithms capables of learning abstractions of the problem to reuse its knowledge in similar ones. Most of them are based on making decisions one after the other and looking how this choices change

the environment.

The goal of this thesis is to extend a specific RL algorithm based on ANNs to help machines solve complex tasks.

Chapter 2

Background

2.1 Reinforcement Learnings

Reinforcement Learning is an area of Machine Learning (ML) that models a problem focusing on maximizing a numerical reward by interacting, in a sequential manner, with the environment. To achieve this, the agent obtains a representation of the situation of the problem, named *state*; then selects an *action*; interacts with the environment, by applying that action; and finally it finds itself a new state while obtaining some *reward*. Repeating this steps several times, the agent should explore the different states of the problem and learn how to map situations to actions with the purpose of maximizing the reward.

The learning process is usually organized in *episodes*. Each episode is composed by an initial state s_0, the first one that the environment generates; intermediate states s_t ; and a final state s_T, which establish the end of an episode. The final state belongs to the set of final states of the environment, which is composed by one or more states. The agent's goal is to maximize the total amount of reward it receives in a complete episode. When an episode ends the the environment resets and starts again from s_0 or some other initial state that belongs to the set of initial states. In some cases the agent interacts infinitely without reaching any terminal state, we call this *continual tasks*.

One of the challenging aspects of reinforcement learning is the tradeoff between *exploration* and *exploitation*. In the kind of environments this thesis is focused on, the actions applied affect not only immediate rewards but also future ones. This implies that taking actions with high immediate reward is not always the best choice,

there might be some series of actions with which it obtains higher rewards in the future. In reinforcement learning, the agent should exploit immediate rewards in order to obtain feedback of how is it performing. Nevertheless, it may also explore different actions that do not have as high immediate reward in order to find higher future rewards.

2.1.1 The environment

We will describe this as in the textbook *Reinforcement Learning: An Introduction* from Sutton and Barto, (1998, Section 3.1).

The *environment* is the main source of information about the problem that we want to model with which the *agent* is able to interact. This interactions and information about the problem must be stated in a specific way.

The *state* is the peace of information, from the environment, that our agent uses to make decisions. Interactions are transferred in the form of actions that the environment makes available for the agent. The information given about the problem must change based on the actions applied.

The agent and environment interact with each other in a sequence of discrete time steps, $t = 0, 1, 2, \ldots$ At each time step, t, the agent receives some state $s_t \in \mathcal{S}$ and on that basis selects an action $a_t \in A(s_t)$. One time step later, the agent receives a numerical reward, $r_{t+1} \in \Re$, and finds itself a new state, s_{t+1}. The state transition and reward must depend on the sequence of past actions and states, if not the agent will not be able to figure out what to do for obtaining the highest rewards. The methodology in which the agent interacts step by step with the environment is named Sequential Decision Process (SDP).

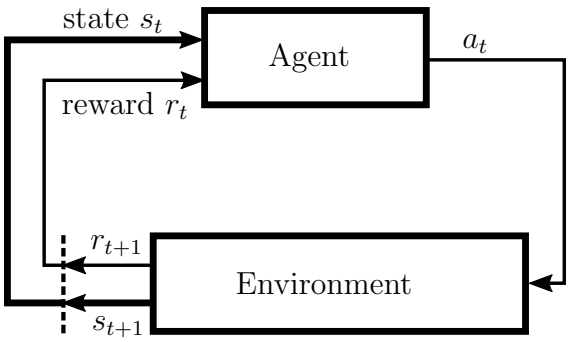

Figure 2.1: Diagram of interaction between agent and environment (Sutton and Barto, 1998, Section 3.1)

This is an abstract and very flexible framework that fits in a large variety of problems. The different components of this interface can be arbitrarily defined to model different kinds of actions, time steps, states and environments. But it restricts rewards to the real domain. For example, the state can be any kind of information about the environment, from just an image of some maze to complex market statistics. This fact is important to us because the algorithm proposed in this thesis uses domain knowledge to define a state representation of the environment composed by images and additional information.

2.1.2 Markov Decision Processes

A Markov Decision Process (MDP) is a restricted case of SDPs (Sutton and Barto, 1998, Section 3.5). In MDPs, the state s_{t+1} generated by applying action a_t in state s_t only depends on these two factors (a_t and s_t), unlike in SDPs that may depend on any previous states, rewards or actions. This fact is named the *Markov property* and is formally described as:

$$P\{s_{t+1} = s, r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\} = P\{s_{t+1} = s, r_{t+1} = r | s_t, a_t\} \tag{2.1}$$

As stated, the probability distribution over states and rewards only depends on the immediate previous state and action, not the full history. Bear in mind that the state s_t may contain some representation of previous states, actions and rewards but not the full history. This abstract could be as simple as the number of actions taken.

All concepts presented in this thesis assumes that the problems can be defined as a MDP, that is, they can be expressed as a tuple $\langle \mathcal{S}, \mathcal{A}_s, \mathcal{P}_a(s, s'), \mathcal{R}_a(s, s'), \gamma \rangle$ where:

- $\mathcal{S}$ is the state space.
- $\mathcal{A}_s$ is the set of actions available in state s.
- $\mathcal{P}_a$ is the transition probability function $\mathcal{P}_a : \mathcal{S} \times \mathcal{S} \to [0, 1]$ for action $a \in \mathcal{A}_s$. To clarify, $\mathcal{P}_a(s, s')$ is the probability that taking action a in state $s \in \mathcal{S}$ will lead to state $s' \in \mathcal{S}$.
- $\mathcal{R}_a$ is the reward function $\mathcal{R}_a : \mathcal{S} \times \mathcal{S} \to \Re$ for action $a \in \mathcal{A}_s$. To clarify, $\mathcal{R}_a(s, s')$ is the expected reward for transiting from state $s \in \mathcal{S}$ to state $s' \in \mathcal{S}$ by taking action a.
- $\gamma \in [0, 1]$ is the discount factor of future rewards used to calculate returns (Subsection 2.1.3)

$\mathcal{S}$ and $\mathcal{A}_s$ may be infinite sets which implies that $\mathcal{P}_a(s, s')$ and $\mathcal{R}_a(s, s')$ may also be infinite. If this happens the MDP is named infinite MDP.

2.1.3 Returns

We have defined the goal of the agent, that is to maximize the total reward obtained in an episode. We also said that the agent must learn to reach that goal, but we have not defined yet how the agent should select the actions to achieve it. The expected return R must be defined in a way that encourages the agent to learn, so R is some specific function of the reward sequence (Sutton and Barto, 1998, Section 7.1). The simplest way of computing it is just adding all the future rewards.

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T \tag{2.2}$$

Where r_T is the reward obtained in some terminal state. In continual tasks the time step $T = \infty$ implying that the return, which is what we are trying to maximize, will also be infinite. To avoid this problem we must introduce the *discounted return*, which adds in a discount factor γ. This factor modifies the previous formulation in the following way:

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

where γ is a parameter, $0 \leq \gamma \leq 1$. Observe that as k approaches infinity the weight applied to r decreases, if $\gamma < 1$. If the final time step T tends to infinity then this new formulation converges, not like the previous one. The discount factor is a hyperparameter related to how farther rewards should be taken into account or not. If we use a value near 0 then the algorithm is capable to rewards that are few time steps away.

Given that the value function (Subsection 2.1.6) is an approximation of the future returns, we can use it to define the total return value without the entire series of rewards. This concept is called the *n-step return* at time t, and its general formulation is:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}) \tag{2.4}$$

2.1.4 Reward shaping

Sometimes an agent is not "smart" enough to reach the environment rewards. In this scenario some new rewards can be added to guide the agent towards the original rewards.

Specifically, applying reward shaping to a MDP (Ng, Harada, and Russell, 1999) is to modify its reward function $\mathcal{R}_a(s, s')$ getting a new one $\mathcal{R}'_a(s, s')$, the difference between them is defined by the *shaping reward function*. More formally $\mathcal{R}'_a(s, s') = \mathcal{R}_a(s, s') + \mathcal{F}(s, a, s')$ where $\mathcal{F} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \Re$ is a bounded real-valued function.

The function $\mathcal{F}$ should be chosen using expert knowledge about the domain or in a general manner that works for any MDP. This choice should be made carefully, because the modification of the MDP rewards should guide the agent to reach the same optimal policy π^* (Subsection 2.1.5) and not another policy which is suboptimal for the original problem. To guarantee consistency with the optimal policy, $\mathcal{F}$ must be *potential-based*. We say $\mathcal{F}$ is potential-based if there exists a real-valued function $\phi : \mathcal{S} \to \Re$ such that for all $s \in \mathcal{S} \setminus \{s_0\}, a \in \mathcal{A}, s' \in \mathcal{S}$

$$\mathcal{F}(s, a, s') = \gamma\phi(s') - \phi(s) \tag{2.5}$$

By defining $\mathcal{F}$ in this way we can ensure that the reward shaping is robust, near-optimal policies in the original MDP remain near-optimal in the new MDP.

2.1.5 Policy

As described by Sutton and Barto, Section 1.3, a policy π defines the learning agent's way of behaving at a given time. It selects the action, given the environment state. During the learning process of the agent its policy may vary multiple times, looking for the best mapping of state-action pairs that fits the problem. This is called the optimal policy π^* and the objective of any reinforcement learning algorithm is to approximate its policy to the optimal policy.

In most cases the policy is a probability distribution over the actions that can be choosed in some state s. Is denoted as $\pi(a, s)$ where s belongs to the state space and a to the action space, fulfilling the following condition:

$$\forall s \in \mathcal{S} \sum_{a \in \mathcal{A}_s} \pi(a, s) = 1 \tag{2.6}$$

The action selected to apply to the environment is randomly selected with this probability distribution. When $\pi(s, a) = 1$, for some a, the policy is *deterministic* in state s, because action a will always be chosen in that state.

2.1.6 Value functions

As described by Sutton and Barto, Section 3.7, the *value function* is an estimate of how good it is for the agent to be in a given state. It uses the expected return to create value functions that depends on future rewards which the agent will obtain. This value will be used by the agent to choose the action. More formally, the value is the expected return for an agent that follows policy π from state s.

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\} \tag{2.7}$$

The agent will look for the policy that maximizes its values, the optimal value function V^* has the highest value for all states.

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{2.8}$$

But to select the best actions we need some formulation of V depending on the selected action. This is the Q value, defined as the value of taking action a in state s and then following policy π.

$$Q^{\pi}(s, a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\} \tag{2.9}$$

Both V and Q are useful values that the algorithm will learn from experience. As it goes taking actions and receiving rewards it must compute the returns to better approximate Q and V values. This will make converge the agent's policy to the optimal policy π^*. In fact the optimal policy is formally described in terms of the value function, satisfying:

$$V^{\pi^*}(s) \geq V^{\pi}(s) \forall \pi \forall s \in \mathcal{S} \tag{2.10}$$

Where $\mathcal{S}$ is the state space of the environment.

So we can define V^* and Q^* as the value function and *action value function* respec-

tively. that give us the expected return of following the optimal policy π^*.

2.1.7 Q-learning

Q-learning is an off-policy algorithm that uses experience to approximate the optimal action value function. It updates the Q function in each step t with the following method:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[\, r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.11}$$

(Sutton and Barto, 1998, Section 6.5)

Where α is a constant step-size parameter that controls how fast the $Q(s_t, a_t)$ value change. Note that the Q value is being updated based on rewards and the max Q of all actions in s_{t+1}, which makes that in the long term Q converges to Q^*. It is also important that this updates does not depend on the policy followed so it will converge even in the case a random action selection is being used. The only imperative is to visit all state-action pairs continually to make several updates of each Q value and continually improve its approximation to Q^*.

There is a more general version of this algorithm called *n-step Q-learning* which instead of using the reward and Q value of the next state $t + 1$ it updates taking into account $t + n - 1$ rewards and the maximum $t + n$ Q value.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[\, \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n \max_a Q(s_{t+n}, a) - Q(s_t, a_t)] \tag{2.12}$$

This make sense when the action taken in states $t, t + 1, \ldots, t + n - 1$ and their respective rewards are known.

2.1.8 Actor Critic

It is a different representation of the agent-environment interface where the agent is splitted into policy and value functions. The *actor* is the one that selects the actions (policy function does this) and the *critic* criticizes the actions taken by the actor (value function does this). The value function determines how good is a state in terms of future rewards and the states is determined by the actions taken following the policy.

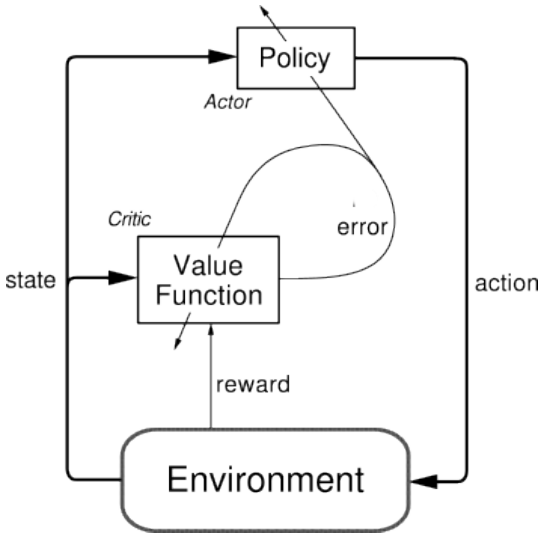

Figure 2.2: Representation of the Actor Critic interface.

In Figure 2.2 it can be seen how the error is computed based on the approximation of the value function changing how actor and critic behave. The hole learning process depend on the error and is defined as the difference between the future returns and the approximation of this value made by the algorithm. More formally:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.13}$$

By minimizing this error δ_t the policy will trend to the optimal policy. The value function will also approximate more precisely future return values.

2.1.9 Semi-Markov Decision Processes

A SMDP is a framework for *hierarchical reinforcement learning* in which actions are not atomic, in the sense that they do not last for just one time step, can be composed by several smaller actions and make decisions based on several states.

A common formalism for representing complex actions is the options framework. An option can be expressed as a tuple $\langle \mathcal{I}, \pi, \beta \rangle$ where $\mathcal{I}$ is the set of initial states in which the option is available, π is the policy that defines which actions must be taken inside the option and β is a probability distribution over states of the option being interrupted. More formally, a Markov option is made up by:

- $\mathcal{I} \subseteq \mathcal{S}$ where $\mathcal{S}$ is the set of states
- $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ where $\mathcal{A}$ is the set of available actions during the option, it

can also contain other options.

- $\beta : \mathcal{S} \to [0, 1]$

Since an option can be composed by several actions or other options, and each action lasts one time step, an option may last more time steps.

With Markov options the interrupt decision is made on the current state s_t but sometimes we may want to end an option if some specific amount of time has elapsed. For this purpose semi-Markov options were created, they use the history of states and actions taken in that option to select the next action and the end condition. So in both π and β instead of using $\mathcal{S}$ it uses Ω which is the set of all histories $h_{t\tau}$ defined as follows:

$$h_{t\tau} = \{s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, ..., r_\tau, s_\tau\} \tag{2.14}$$

Where τ is the holding time of the option, thus is, the amount of time steps it takes to complete. Finally the three components of a semi-Markov option are: $\pi : \Omega \times \mathcal{A} \to [0, 1]$, $\beta : \Omega \to [0, 1]$ and $\mathcal{I} \subseteq \mathcal{S}$

A SMDP is a variation of an MDP where the set of actions $\mathcal{A}_s$ is made up by semi-Markov options. A SMDP problem can be expressed as a tuple $\langle \mathcal{S}, \mathcal{O}_s, P_o(s, s'), R_o(s), \gamma \rangle$ where:

- $\mathcal{S}$ is the state space
- $\mathcal{O}_s$ is the set of possible options available in state s.
- $P_o(s, s')$ is the likelihood of reaching state $s' \in \mathcal{S}$ from state $s \in \mathcal{S}$ when taking option $o \in \mathcal{O}_s$, discounted depending on the time taken to reach it.

$$P_o(s, s') = \sum_{k=1}^{\infty} p(s', k)\gamma^k \tag{2.15}$$

  Where $p(s', k)$ is the probability that the option o terminates in s' after k steps.

- $R_o(s)$ is the expected reward for taking option o in state s. To define it we must introduce $\varepsilon(o, s, t)$ which denotes the event of o being initiated in state s at time t.

$$R_o(s) = E\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{k-1} r_{t+k} \mid \varepsilon(o, s, t)\right\} \tag{2.16}$$

- $\gamma \in [0, 1]$ is the discount factor.

With SMDPs we can define multiple levels of agents where, for example, some of

them control basic actions and others complex options.

2.2 Artificial Neural Networks

In this section we describe what are Artificial Neural Networks (ANNs), how they work and which are the most important variants for this thesis. Most of the definitions and figures are taken from *Neural Networks and Learning Machines* by Haykin, (2009)

An ANN is a collection of artificial neurons that process information in a parallel way and stores experiential knowledge to use it in another experiences. It continually adapts connexions between neurons to reach some goals, this process is named learning.

2.2.1 Artificial neuron

An *artificial neuron*, from now on called *neuron*, is an information-processing unit modeled as a mathematical function with these five basic elements:

- A set of connecting links with some weight associated. A signal x_j at the input of connexion j related to neuron k is multiplied by the weight w_{kj}.
- An adder for summing the weighted input signals.
- An activation function $\varphi(\cdot)$ that limits the amplitude of the output of a neuron.
- A bias b_k which has the effect of increases or reduces the input of the activation function.
- An output signal y_k which is the result of the processed information.

In mathematical terms we describe the neuron with the following equation:

$$y_k = \varphi(b_k + \sum_{j=1}^{m} w_{kj} x_j) \tag{2.17}$$

where $x_1, x_2, \ldots, x_m$ are the input signals; $w_{k1}, w_{k2}, \ldots, wk_m$ are the respective weights of neuron k; b_k is the bias; $\varphi(\cdot)$ is the activation function; and y_k is the output signal of the neuron.

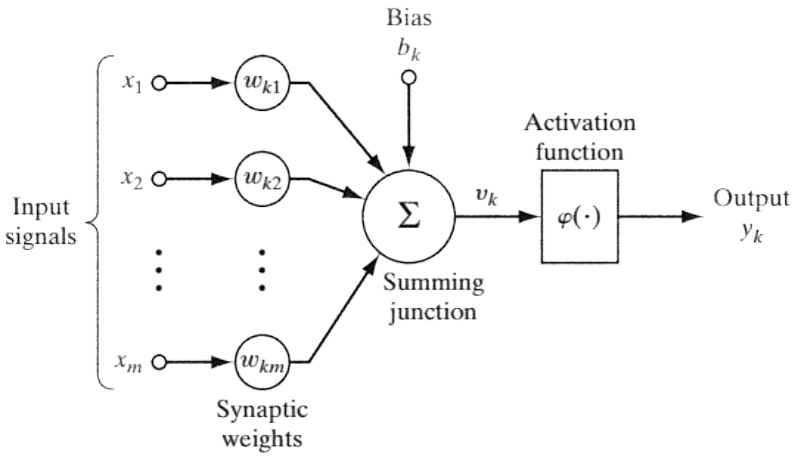

Figure 2.3: Achitecture of an artificial neuron

2.2.2 Multilayer perceptron

A *multilayer perceptron* is a kind of ANN made up by several ordered *layers* of neurons. It is characterized by having an input layer, several hidden layers and an output layer.

The input layer is a set of neurons whose inputs are not connected to other neurons and whose outputs are connected to the first hidden layer. Each of the hidden layers satisfies that the output of neurons in layer i is the input of some neurons in layer $i + 1$, except the last hidden layer that is connected with the output layer. The output layer is a set of neurons whose results form the output signal of the network. When each neuron has as input all neurons of previous layer the network is named fully connected.

In multilayer perceptrons there are two kinds of signals, *function signals* and *error signals*.

A function signal is an input signal that propagates through the network generating an output signal. The output signal is the result of the function parametrized by all the weights and biases of all neurons.

An error signal originates at the output layer and propagates backwards through the network. It approximates the difference between the output signal and the desired output signal, given that we are using the ANN to approximate some function. Each node computes its contribution to the error by estimating the gradient with respect its weights and bias.

The process of adjusting the weights and biases of each neuron to approximate

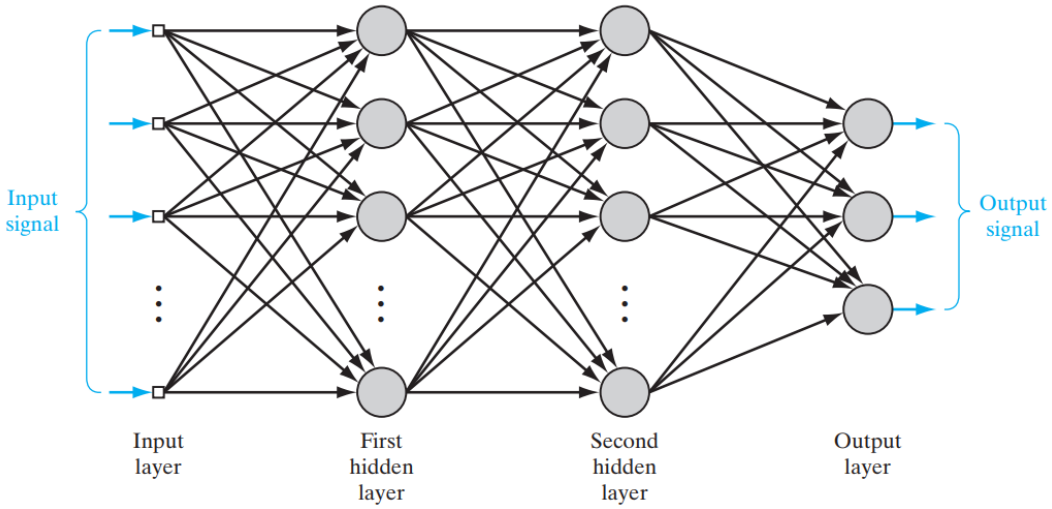

Figure 2.4: Fully connected multilayer perceptron.

a required output signal, given some input signal, is named *training*. I order to accomplish this purpose different gradient descent algorithms might be used, such as RMSprop (Hinton, 2014). The speed with which the parameters change is controlled by a fixed hyperparameter called the *learning rate*.

Fully connected layers are the most computationally expensive to train because of the amount of connexions and variables. One way of reducing complexity is to use *shared weights*, instead of having different weights for each input of each neuron, some part of the weights of a neuron are reused to be multiplied for the input of another neurons. This reduces the number of variables the network should maintain and update, but the number of mathematical computations will be the same.

Another way of reducing the number of computations is by reducing the number of nodes. *Pooling layers* do that, is a form of non-linear downsampling that applies a function to a group of nodes resuming them in just one node.

2.2.3 Transfer learning in Artificial Neural Networks

The aim of *transfer learning* techniques is to improve the learning process of a new task, given some previous knowledge about related tasks. Artificial Neural Networks, especially the deepest ones, need tons of data to adjust their weights to successfully perform some tasks.

A way to reduce the amount of training data needed and speed up the learning process is to reuse some ANN previously trained for a similar task. Then the ANN

adapts its weights to succeed in the target task. For example, if we want to recognize dogs inside images, we could use some existing ANN developed to recognize cats and then adapt its weights.

Another method is to train our network in a general problem related to the target task to later focus on it. With DNN we could train a large network on some task and then reuse just the first layers, that should contain a basic feature extraction about the problem. In the previous example we could take the layers of the network responsible of recognizing shapes and add some new nodes to train on recognizing dogs. A simpler example is to train the full network first to recognize shapes and then train on dogs.

2.3 Deep Learning

Deep Learning (DL) is a machine learning method based on high-level representations of the data. It basically extends ANNs by adding multiple hidden layers creating huge networks with lots of parameters and interrelations.

In the last years this technique has become really popular as a consequence of its great results comparing to human level performance in many tasks.

2.3.1 Convolutional Neural Network

This network is a variation of the multilayer perceptron specifically designed to recognize visual patterns. They are usually made up of four different types of layers: *convolutional layers*, *ReLU layers*, *pooling layers* and fully connected layers.

Convolutional layers apply a convolution to some previous layer, characterized by a filter. Each node defines its connections based on the filter and their position. The filter is made up by *kernel size*, *stride*, *padding* and *values*. The kernel size defines the shape and amount of nodes of the previous layer that are connected to a convolutional layer node. The stride defines how the kernel moves across the previous layer, with each move, the filter is applied to some group of nodes by connecting them to a new node of the convolutional layer. The padding (specifically zero padding) is the amount of zeros to add around the border of each dimension in the previous layer, this helps to preserve as much information about the previous layer values. Finally, the values define the weights for each input signal in each neuron. These

are the same values for each convolutional layer node. This concept is named *shared weights* and minimizes the amount of variables the ANN has to retain and update.

rectified linear unit (ReLU) layers increases the nonlinear properties of the model by applying the function $f(x) = max(0, x)$ to all outputs of the previous layer. It is common practice to apply this layer after convolutional layers. Applying ReLU functions mitigates the *vanishing gradient problem*, which is characterized of an exponential reduction of the gradient through the layers.

Pooling layers generate a downsampled version of the previous layer. It applies some function (usually a max function) to a group of outputs of the previous layer and saves the result in just one node of the pooling layer. This technique reduces the parameters of the network which decrease the memory and calculations needed, it also reduces overfitting.

2.3.2 Deep Q-Network

This algorithm was introduced by Mnih et al. in "Human-level control through deep reinforcement learning" (2015).

A Deep Q-Network, from now on DQN, is a deep convolutional neural network designed to approximate Q values of reinforcement learning. This agent learns from images to approximate the optimal action-value function (Subsection 2.1.7). The aim of DQNs is to learn successful policies directly from images of some given problem, for example, a game. For that purpose the output of the network will be the Q values of each action given some input state.

Bear in mind that when using ANNs in reinforcement learning the learning problem can become unstable or even diverge. This is caused by the weight updates, that changes not only the result of som action-value pair but all of the action-value associations; the correlation present in sequence of observations; and the correlations between the action-values and target values. For that reason, apart from the ANN, they used an experience replay (E). This structure saves last T states which the environment has gone by in the format $e_t = (s_t, a_t, r_t, s_{t+1})$. Then it picks in an uniformly random manner from this set to train the network. Training on independent and identically distributed samples helps reducing the instability generated by the ANN.

Q-learning updates are organized in iterations, containing several training steps. Each iteration has its fixed network parameters θ_i^- and applies several training

steps obtained from the experience replay to finally change them. This updates are characterized by the loss applied to the network to adjust its weights. They defined the loss function in the following way:

$$L_i(\theta_i) = E_{(s,a,r,s')\sim U(D))}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right] \tag{2.18}$$

Where θ_i and θ_i^- are the wights of two ANNs with the same architecture. θ_i is updated each training step while θ_i^- only change when the iteration finish, following $\theta_i = \theta_i^-$ after this happens.

This network has successfully surpass human-level skills in many Atari games. But in Montezuma's Revenge it was unable to obtain any score (see Subsection 3.2.3).

2.3.3 Asynchronous Advantage Actor-Critic

The algorithm Asynchronous Advantage Actor-Critic, from now on A3C, was introduced by Mnih et al. in "Asynchronous Methods for Deep Reinforcement Learning" (2016).

The main difference between asynchronous methods and the methodology used in DQN is the experience replay. In A3C instead of having an experience replay, to take uniformly random states and avoid instability, it runs in parallel multiple instances of the environment. At each time step the agent is being trained with different states of multiple environments that runs in isolation.

Another difference between DQN and A3C is the reinforcement learning approach. It uses the actor-critic interface introduced in Subsection 2.1.8. It defines two ANNs that represent the value of the state (V) and the policy (π). V is the output of just one node, but π is the output of a set of nodes, each one representing the probability of taking an action. In practice this two ANNs share almost all the layers, except the last one (V and π).

Also the loss function used to train the network differ from DQN, it uses n-step returns (Equation (2.4)) and it does not update Q values, it updates V and π. The number of steps used in the loss function is t_{max} (an hyperparameter) but if a final state is reached before that, the loss is computed with less than t_{max}. It also adds an entropy factor H based on the policy, in order to improve exploration by discouraging premature convergence to suboptimal deterministic policies. A3C uses

the following update function:

$$\rho \nabla_{\theta'} \, log \, \pi(a_t \mid s_t; \theta') \, A(s_t, a_t; \theta, \theta_v) + \eta \nabla_{\theta'} H(\pi(s_t; \theta')) \tag{2.19}$$

Where A is the advantage function of n-step returns in terms of V. θ' are the weights related to the policy and θ_v the weights related to the value function. η is an hyperparameter that controls the strength of the entropy regularization term and ρ another one which controls the strength of the advantage function. The advantage function defines at follows:

$$A_t = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v) \tag{2.20}$$

Where k can vary from state to state (if a final state is reached) and is upper-bounded by t_{max}.

This algorithm is parallelized in different threads which contain different replicas of a shared ANN. Each thread interacts with the environment using his own ANN, until t_{max} or end of game are reached, and updates the global network parameters with the update function above. The algorithm used in each thread is the following.

Algorithm 1 Asynchronous Advantage Actor-Critic - psudocode for each actor-learner thread (Mnih et al., 2016)

//Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$

//Assume thread-specific parameter vectors $\theta' and \theta'_v$

Initialize thread step counter $t \leftarrow 1$

repeat

Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.

Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$

$t_{start} = t$

Get state s_t

repeat

Perform a_t according to policy $\pi(a_t|s_t;\theta')$

Receive reward r_t and new state s_{t+1}

$t \leftarrow t + 1$

$T \leftarrow T + 1$

until terminal s_t **or** $t - t_{start} == t_{max}$

$$\mathrm{R} = \begin{cases} 0, & \text{for terminal } s_t \\ V(s_t, \theta'_v), & \text{for non-terminal } s_t \text{ // } \textit{Bootstrap from last state} \end{cases}$$

for $i \in \{t-1, \ldots, t_{start}\}$ **do**

$R \leftarrow r_i + \gamma R$

Accumulate gradients wrt θ' : $d\theta \leftarrow d\theta + \rho \nabla_{\theta'} log\pi(a_i \mid s_i;\theta')(R - V(s_i;\theta'_v)) + \eta \nabla_{\theta'} H(\pi(s_i;\theta'))$

Accumulate gradients wrt θ'_v : $d\theta_v \leftarrow d\theta_v + \rho(\partial(R - V(s_i;\theta'_v)^2)/\partial\theta'_v)$

end for

Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.

until $T > T_{max}$

Chapter 3

Methodology

3.1 Algorithms

We have modified the A3C algorithm in order to take advantage of sub-tasks with the purpose of enhancing training speed and exploration. We refer to this new algorithm as Multiple Asynchronous Advantage Actor-Critic (MA3C) and in this section it can be found specifications about the architectures of both ANNs: A3C and MA3C. In addition we will explain the MA3C algorithm.

They have been developed using tensorflow (Abadi et al., 2015), an open source machine learning framework which the authors of A3C and DQN are also using (Kavukcuoglu, 2016).

3.1.1 Asynchronous Advantage Actor-Critic

As we have explained before (Subsection 2.3.3), this algorithm uses the image of the environment to represent the state. We have used almost the same preprocessing and network architecture (Figure 3.1) than in "Asynchronous Methods for Deep Reinforcement Learning" (Mnih et al., 2016). In practice, the images of any game are resized to 84×84 greyscale pixels in order to reduce the computational cost of training and to fit the convolutions. The algorithm applies this preprocessing to 4 stacked frames allowing the ANN to implicitly calculate the movement of the different pixels/objects on the screen. We also use a frame skip of 4, i.e. only one out of four consecutive frames is taken into account and stacked. To generate the skipped frames A3C repeats 3 times the last selected action, emulating the human behaviour,

who is unable to take actions as fast as the game is rendering (60 frames per second). The state dimensionality for MR is $84 \times 84 \times 4$ which defines the input layer of the CNN, but in Simple States (Subsection 3.2.1) and Complex States (Subsection 3.2.2) the RGB channels of a single frame are used instead. The dimensionality of the input layer in that games is $84 \times 84 \times 3$.

There are several hidden layers specifically designed to obtain high level abstractions of the environment frames. The first one is a convolutional layer (Subsection 2.3.1) with 16 filters with kernel dimension 8×8 and stride 4, followed by a ReLU layer. The second layer is also a convolutional layer, but with 32 filters with kernel dimension 4×4 and stride 2, again followed by a ReLU layer. The last hidden layer intends to represent general features about the state of the game. It is a fully connected layer composed by 256 ReLU nodes.

From these 256 features the actor and critic layers, which are the output layers, decide the actions that should be taken in order to maximize the reward, as explained in (Subsection 2.1.8). The actor is made up by as many nodes as actions available in each game with values from 0 to 1, representing the probability distribution π. The critic is just 1 neuron representing the value function $V(s_t)$.

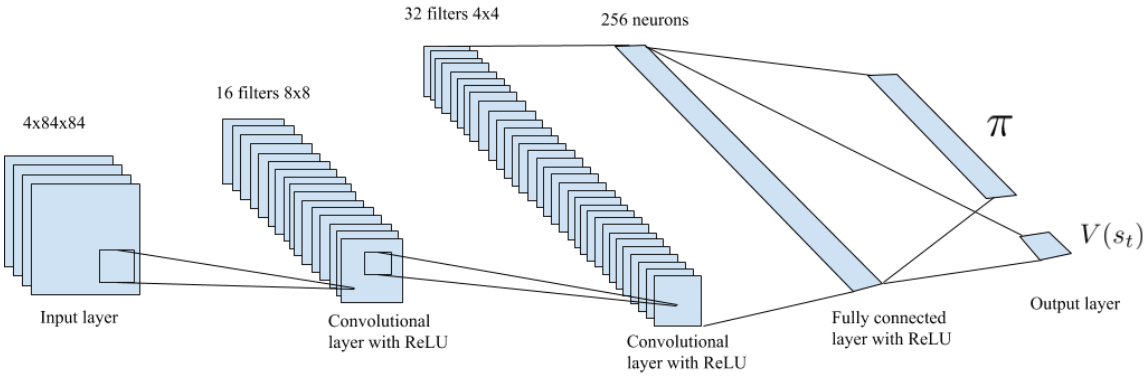

Figure 3.1: Architecture of the A3C algorithm.

3.1.2 Multiple Asynchronous Advantage Actor-Critic

This algorithm is strongly associated with hierarchial reinforcement learning, since the last layers of A3C (π and $V(s_t)$) are replicated several times in order to model different tasks inside a common bigger problem.

As we can see in Figure 3.2 the architecture is very similar to A3C. There is an input layer of size $84 \times 84 \times 4$, a convolutional layer with 16 8×8 filters, another

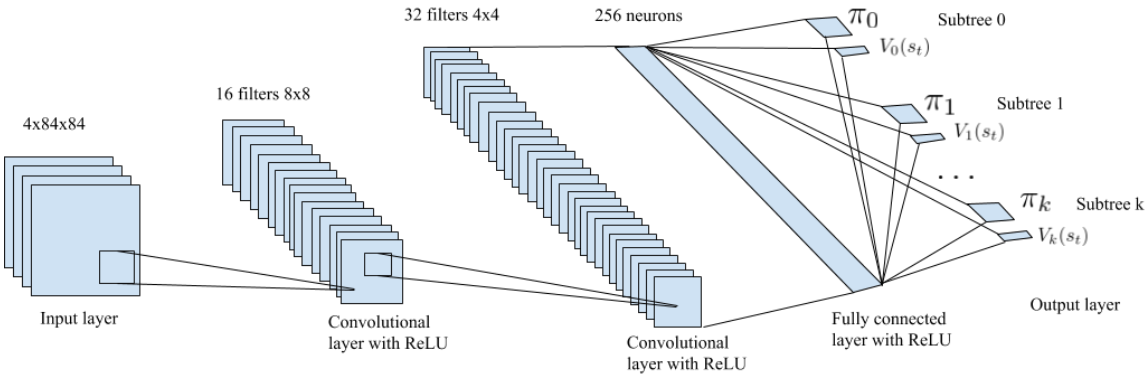

Figure 3.2: Architecture of the MA3C algorithm.

convolutional layer with 32 4×4 filters and a fully connected layer with 256 nodes. The output layer is made up by K copies of the A3C output layer, where $K = k - 1$ is an hyperparameter which might be different depending on the use case. For each of the games we have set K to a different value: with Simple States we have used $K = 3$, with Complex States $K = 2$ and with MR $K = 4$. This value depend on the number of phases (see Subsection 3.2.1 , Subsection 3.2.2 and Subsection 3.2.3). The input layer is also different depending on the game, in the same manner than A3C.

The architecture of MA3C contains multiple *subgraphs* defining different Actor Critic approaches to the given input state, each one connected to the high level features (last layer) but not between them. With this architecture the algorithm uses transfer learning (Subsection 2.2.3) to extract common features about the environment, which are useful for all AC layers. When some task is already trained and a new one starts to train, most of the network is reused, speeding up the learning process. Given that every subgraph is used to solve a different task inside a bigger problem, the common parameters will be optimized to extract as much information as possible about the general problem, while the weights of each subgraph will be optimized to succeed in the specific task.

The update function, advantage function and algorithm of MA3C remain the same ones as in A3C (Subsection 2.3.3). The only difference is that this one selects the chunk of the parameter vectors θ' and θ'_v that will be updated depending on the currently active subgraph. The method which selects one subgraph or another is arbitrary, they might be selected with an AI method, just with a couple of rules or be given by the state. For this thesis we have used two approaches: part of the state, in Simple States (Subsection 3.2.1) and Complex States (Subsection 3.2.2); and some rules about the position of the hero in Montezuma's Revenge (Subsection 3.2.3).

The full modified algorithm is shown in Algorithm 2.

Algorithm 2 Multiple Asynchronous Advantage Actor-Critic - psudocode for each actor-learner thread (Mnih et al., 2016)

//Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
//Assume thread-specific parameter vectors $\theta' and \theta'_v$
//Assume subree-specific parameter vectors $\theta'^k and \theta'^k_v$ *from* $\theta' and \theta'_v$
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $$\mathrm{R} = \begin{cases} 0, & \text{for terminal } s_t \\ V(s_t, \theta'_v), & \text{for non-terminal } s_t \text{ // Bootstrap from last state} \end{cases}$$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Select k *// Select subgraph with any criteria*
    Accumulate gradients wrt θ'^k : $d\theta^k \leftarrow d\theta^k + \rho \nabla_{\theta'^k} log\, \pi(a_i \mid s_i; \theta'^k)(R - V(s_i; \theta'^k_v)) + \eta \nabla_{\theta'^k} H(\pi(s_i; \theta'^k))$
    Accumulate gradients wrt θ'^k_v : $d\theta^k_v \leftarrow d\theta^k_v + \rho(\partial(R - V(s_i; \theta'^k_v)^2)/\partial\theta'^k_v)$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

3.2 Environments

In order to demonstrate the capabilities of MA3C over A3C we have developed two simple environments where the weaknesses of the second one come to light. We have also compared the algorithms in a real game, Montezuma's Revenge, which has

become famous because of its difficulty.

These three environments have been used through a useful toolkit called OpenAi Gym (Brockman et al., 2016), in which the different games can be accessed by a common interface facilitating the analysis of different algorithms and games.

3.2.1 Simple States

This game is made by a 6×10 grid of square blocks. Their colors define the kind of object and how they will interact with the hero (an special object). This are the different types:

- *hero*: A block that can be controlled by the actions.
- *wall*: If the hero hits a wall the game ends and he obtains a reward of value -1. When this happens we say that the hero dies.
- *checkpoint*: When the hero reach this object he obtains a reward of value 1 and enables the hidden checkpoint reward. Once the hero reaches it for the first time he will never obtain the reward again.
- *hidden checkpoint*: When the hero reaches this object the door is activated, enabling the reward of the door. It basically forces the hero to pass through, giving no reward to him.
- *door*: If the hero reach the door having gone through the different checkpoints the game ends and he obtains a reward of value 1.

The goal of the hero is to reach the door by passing through the different checkpoints without colliding with any wall, this will give to the hero the maximum score, 2 points. The game is organized in three different phases/states, each one having a different objective. The first one's objective is to reach the checkpoint, of the second one's objective is the hidden checkpoint and the third one's, the door. The information about the current phase is always available to the player.

In order to solve this game with both A3C and MA3C algorithms we must model it as a SMDP problem, describing components of the tuple $\langle \mathcal{S}, \mathcal{A}_s, \mathcal{P}_a(s, s'), \mathcal{R}_a(s, s'), \gamma \rangle$.

- Each states is a pair $s_t = (\mathbf{F}, p)$ where $p \in \{0, 1, 2\}$ is the phase of the game and $\mathbf{F}$ is the matrix of RGB pixels with dimensionality $84 \times 84 \times 3$. $\mathcal{S}$ is the set of all s which satisfies the conditions about object collisions presented above.

Figure 3.3: The hero navigating through a hostile environment trying to reach the door. The wall, hero, checkpoint, hidden checkpoint and door are represented with the following colors respectively: blue, grey, green, turquoise and violet.

- $\forall s \ \mathcal{A}_s = \{\text{UP}, \text{DOWN}, \text{LEFT}, \text{RIGHT}, \text{WAIT}\}$ and each action $a_t \in \mathcal{A}_s$ corresponds to the movement of the hero.
- This game is deterministic, which means that $\forall s \forall a \exists s' \mid \mathcal{P}_a(s, s') = 1$ implying $\forall s \forall a \forall s'' \neq s' \ \ \mathcal{P}_a(s, s'') = 0$
- The reward function is determined as explained before. Depending on the phase (p) some transition from a frame $\mathbf{F}_t$ to another frame $\mathbf{F}_{t+1}$ may come with a reward $\mathcal{R} \in \{-1, 0, 1\}$.
- The discount factor γ is 0.99

The purpose of creating this game is to prove that the MA3C algorithm (Subsection 3.1.2) has better exploration skills than A3C (Subsection 2.3.3).

3.2.2 Complex States

This game is really similar to the previous one but the dynamics are a bit different. The hero must follow a series of steps to reach the door. The order in which the hero must go through the objects is: checkpoint → hidden checkpoint → door → hidden checkpoint → checkpoint → door. We force the hero to go back on his own steps when he reaches the door. In this game some of the rewards also change. These are the changes with respect to the Simple States game, the rest remains equal:

- *checkpoint*: The first time that the hero goes through this object he obtains a reward of 1. Then he must follow the rest of the path described above to obtain again 1 of score (after the second time he visits the hidden checkpoint).
- *hidden checkpoint*: In this game both 2 times that the hero goes through this object (following the path) receives 1 of score. The first time the checkpoint object enables the reward and the second time the door enables it.
- *door*: The first time the hero reach the door it moves to the starting position of the hero (bottom left corner) and gives him 1 of score. The second time behaves as in Simple States.

Figure 3.4: The hero after reaching the door for first time. The wall, hero, checkpoint, hidden checkpoint and door are represented with the following colors respectively: blue, grey, green, turquoise and violet.

In this game there are only two phases. The first one goes from the start until the first time the hero reaches the door. The second one finishes the second time it reaches the door. As in Simple States the hero must follow the path in order to obtain the rewards and finish the game. The information about the current phase is also available to the player.

The MDP problem of this game is exactly the same as in Simple States (Subsection 3.2.1). The only thing that changes are the rules of rewards and phases which we have already defined.

The purpose of creating this game is to prove that A3C is quite bad going back on his own steps (when the image of the game is similar) while for MA3C is easy.

3.2.3 Montezuma's Revenge

We wanted to test MA3C in an environment which other authors also did a research on. Montezuma's Revenge (MR) is an Atari game created in 1984 which recently has become famous because of its difficulty. DQN and A3C algorithms get less than 100 score on average when playing this game (Mnih et al., 2016).

The hero, Panama Joe, goes into an Aztec pyramid full of treasures, traps and monsters. The game takes place inside the pyramid where there are different rooms with the treasures he must collect. In order to navigate from a room to another he must collect keys and open doors while avoiding traps and monsters. It is a relatively big game (24 rooms) where the agent starts in screen 1 (Figure 3.5) and must navigate through most of the rooms in order to collect a group of special gems which make him win the game.

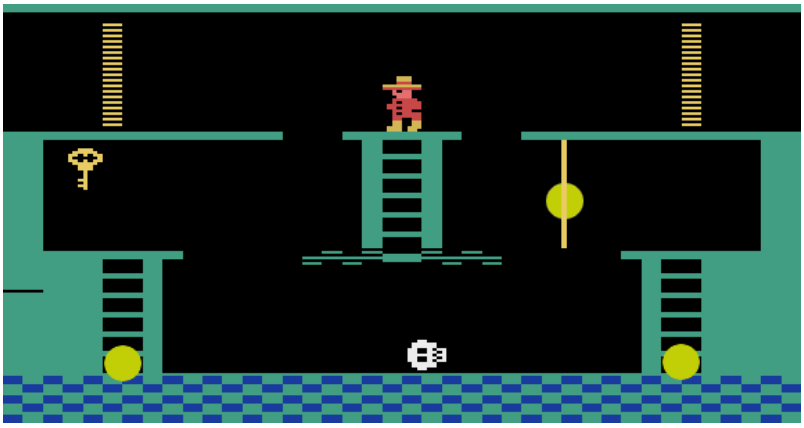

Figure 3.5: The hero is in the starting position of the game, in the first screen of Montezuma's Revenge. Three yellow bullets have been added in order to represent the checkpoints.

Since this game is really complex and computationally expensive to train, we have made some changes to it.

- In the hole game there are several screens but we will only use the first one in our experiments.
- The hero has multiple lives but we will consider he has just 1 and, if he dies, the game will be reset.
- When the hero reaches the key we will also reset the game, so the unique goal of the hero is to take that object.

Nevertheless, the procedures remain the same as in the original game.

At each time step, the player can take 8 different actions: NOOP (stay still), FIRE (jump straight up), UP, RIGHT, LEFT, DOWN, LEFTFIRE (jump to the left), RIGHTFIRE.

There are one rope and three stairs with which the hero can go up and down. If he jumps into the rope he will grab the cord, but, if he jumps into the stairs, he will fall into the floor. From now on we will refer to the right stairs as *Rstairs* and to left ones *Lstairs*. We will use later the position (in screen space) of the objects, which are the following:

- rope: from $(109, 174)$ to $(109, 212)$
- Rstairs: $(133, 148)$
- Lstairs: $(21, 148)$
- key: $(14, 209)$

In order to obtain this positions, and the agent's position, we have used the memory layout described in *Solving Montezuma's Revenge with planning and reinforcement learning* (Garriga Alonso, 2016).

The hero can die within two manners: by falling into the floor from some high position (e.g. from the rope) or by touching the skull in the bottom, which is always moving left and right.

The hero can obtain a score by collecting the key, which gives him 100 points. He can also win 10 points by passing through each of the checkpoints which we have manually added to guide the agent towards the key. Since they have been selected in order to guide the agent towards π^* this is a way of applying reward shaping (Subsection 2.1.4) to Montezuma's Revenge. The potential function $\phi(s)$ is defined in the following way:

$$\phi(s) = \begin{cases} 10, & \text{hero visits for the first time the positions: rope, Rstairs, Lstairs, key} \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

The difficulty is due to the facility to die and the remoteness of the rewards.

We have modeled this game as a SMDP (Subsection 2.1.9), with the tuple $\langle \mathcal{S}, \mathcal{O}_s, P_o(s, s'), R_o(s), \gamma \rangle$.

The state $s \in \mathcal{S}$ is a pair $s_t = (\mathbf{F}, p)$ where $p \in \{0, 1, 2, 3\}$ is the phase/option of the game and $\mathbf{F}$ is the matrix of processed pixels (as described in Subsection 3.1.1).

The different phases are described by consecutive options in which $\mathcal{I}$ and β are specified, but π must be learned by the algorithm. The consecutive options, in order of appearance, are the following:

- o_0 is the option defined by $\mathcal{I} = s_0$, the game initial state, and β being:

$$\beta = \begin{cases} 1, & \forall s \in \mathcal{S}_{rope} \\ 0, & \forall s \notin \mathcal{S}_{rope} \end{cases} \tag{3.2}$$

  Where $\mathcal{S}_{rope}$ is the set of states in which the agent is in the same screen space as the rope.

- o_1 is the option defined by $\mathcal{I} = \mathcal{S}_{rope}$ and β being:

$$\beta = \begin{cases} 1, & \forall s \in \mathcal{S}_{Rstairs} \\ 0, & \forall s \notin \mathcal{S}_{Rstairs} \end{cases} \tag{3.3}$$

  Where $\mathcal{S}_{Rstairs}$ is the set of states in which the agent is in the same screen space as the Rstairs.

- o_2 is the option defined by $\mathcal{I} = \mathcal{S}_{Rstairs}$ and β being:

$$\beta = \begin{cases} 1, & \forall s \in \mathcal{S}_{Lstairs} \\ 0, & \forall s \notin \mathcal{S}_{Lstairs} \end{cases} \tag{3.4}$$

  Where $\mathcal{S}_{Lstairs}$ is the set of states in which the agent is in the same screen space as the Lstairs.

- o_3 is the option defined by $\mathcal{I} = \mathcal{S}_{Lstairs}$ and β being:

$$\beta = \begin{cases} 1, & \forall s \in \mathcal{S}_{key} \\ 0, & \forall s \notin \mathcal{S}_{key} \end{cases} \tag{3.5}$$

  Where $\mathcal{S}_{key}$ is the set of states in which the agent is in the same screen space as the rope. Bear in mind that β is defined in a way that the option finishes in the same states as the game does.

The action set available inside each option is $\mathcal{A}_s = \{\text{NOOP}, \text{FIRE}, \text{UP}, \text{RIGHT}, \text{LEFT}, \text{DOWN}, \text{LEFTFIRE}, \text{RIGHTFIRE}\}$.

By defining the options in this way, we divide the problem in 4 phases from which, once it changes to the next, it is impossible to go back to another previous option/phase. This simplifies the game, being really similar to Simple States and Complex States games.

As far as the agent will learn how to act inside the option, we cannot accurately define $P_o(s, s')$ and $R_o(s)$. This is because the number of steps k depends on the policy of the option. But we can say that options o_0, o_1 and o_2 have a reward $r_{t+k} = 10$ while o_3 has a reward $r_{t+k} = 110$, i.e., the score given by the key and by the checkpoint.

The discount factor γ is 0.99.

We have decided to model this problem as a SMDP in order to expose the strengths of MA3C inside hierarchical learning models, in the knowledge that modeling it as a MDP may have been easier and makes more sense.

Chapter 4

Evaluation

In this chapter we will supply enough evidence about how the MA3C algorithm learns faster and explores better than A3C in environments where subgoals can be defined.

All the code needed to make the following analysis was developed inside a project of the Universidad Pompeu Fabra researcher Miquel Juyent. We have taken his A3C implementation and extended it in order to allow the use of MA3C and the games Simple States and Complex States. The seed to initialize the weights of the different networks where randomly chosen in each of the experiments and runs.

We have used multiple common hyperparameters along the three experiments, which are:

- The discount factor $\gamma = 0.99$ (Subsection 2.1.3).
- $t_{max} = 20$ (Subsection 2.3.3).
- Entropy regularization term $\eta = 0.1$ (Subsection 2.3.3).
- Value regularization term $\rho = 0.5$ (Subsection 2.3.3).
- The learning rate is 0.007 (Subsection 2.2.2).

4.1 Simple States

We have compared the performance of both algorithms A3C (Algorithm 1) and MA3C (Algorithm 2) in Simple States game (Subsection 3.2.1).

The specific hyperparameters used in this experiment which differ from the general ones are:

- The number of threads in which the algorithms have been parallelized is 8 (Subsection 2.3.3).
- Subgraphs $K = 3$ (just for MA3C, Subsection 3.1.2).

This parameters have been chosen in order to show the disadvantages of A3C as opposed to MA3C. There might be a different combination of hyperparameters in which, at the end, both algorithms converge to the same solution. Since the ultimate goal is to show how MA3C explores better, this hyperparameters are good enough.

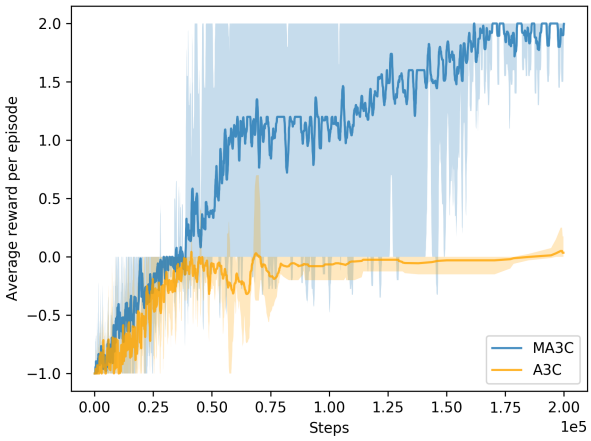

Figure 4.1: Average score over 5 runs of A3C and MA3C algorithms in Simple States environment (200k steps/actions). The graphics have been smoothed by a moving average with a 20 steps window. The shadow represents maximum and minimum values of those 5 smoothed runs.

The learning process of both algorithms measured by score over steps is shown in Figure 4.1. As can be seen, MA3C algorithm obtains score 2 (checkpoint and door) while A3C obtains score 0 because it reaches the checkpoint but then hits a wall.

This is due to the subgraph changes which the algorithm makes every time it goes through any of the checkpoints (normal and hidden). Every time the change happens the data in the common network changes in a different manner (the direction of the gradient) until it converges to common features for all subgraphs. Thanks to that, the randomness is implicitly increased every time the change happens until convergence is achieved. And, as far as more random actions lead to better exploration, this algorithm is using the subgoals (checkpoints) in order to enhance exploration only when needed, specifically when a new phase is achieved.

It is important to remember that the hidden checkpoint does not give any reward, so in some cases there will be no need of adding intermediate rewards, just subgraph changes. In general, when solving a problem with MA3C it might be no need of doing reward shaping (Subsection 2.1.4) and carefully select the $\mathcal{F}$ function to ensure near-optimal policies. The subgraph changes can be done in any state and the optimal policies will remain the same, it will just enhance exploration after reaching that states.

4.2 Complex States

Thanks to the results obtained with Simple States game, we realized that MA3C may be better than A3C learning different policies from two really similar states. We decided to test the algorithms in this game (Complex States, Subsection 3.2.2) in order to observe how they behave in environments in which it is mandatory to retrace its steps. It is important to mention that in this game there is no hidden checkpoint, so the MA3C does not have as much "advantage" as in Simple States.

The specific hyperparameters used in this experiment which differ from the general ones are:

- The number of threads in which the algorithms have been parallelized is 8.
- Subgraphs $K = 2$. (just for MA3C)

The learning process of both algorithms measured by score over steps is shown in Figure 4.2. As can be seen, MA3C algorithm obtains score 6 (all checkpoints and doors) while A3C obtains score about 1.5 on average, which means that half the time it reaches the first door and the other half it only reaches the second checkpoint (at the end both times hits the wall). The outlier drawn as a pink line is low probable case of A3C in which it succeeds, obtaining also score 6.

With this results we can observe how MA3C performs better, not only in terms of exploration but also in average succeed. Going back into his steps may be a difficult task for A3C because of the similarity of inputs. That is, if the policy learned when being next to the first checkpoint is to take the UP action, while coming back from the second checkpoint to the first one, the policy must change in order to take the DOWN action. In both cases the hero will be next to the first checkpoint, so the algorithm must figure out that, the location change of the door, means a change on the goal.

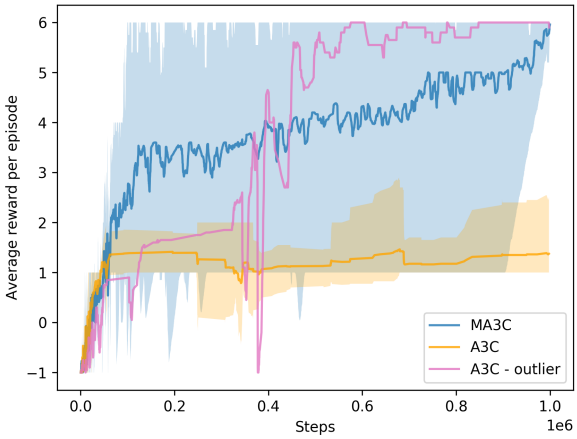

Figure 4.2: Average score over 5 runs of A3C and MA3C algorithms in Complex States environment (1M steps/actions). The best run that we obtained with A3C is shown as an outlier because it was the only run in which we were able to get reward 6. The graphics have been smoothed by a moving average with a 20 steps window. The shadow represents maximum and minimum values of those 5 smoothed runs.

Using MA3C in this kind of problems allows us to change the goal of the agent very easily, just by selecting another subgraph. Since the unique layer which changes is the last one, we will preserve all the previous knowledge about the environment while learning how to reach the new goal.

4.3 Montezuma's Revenge

The specific hyperparameters used in this experiment which differ from the general ones are:

- The number of threads in which the algorithms have been parallelized is 32 in order to speed up the learning process.
- Subgraphs $K = 4$, one for each option which the algorithm must learn. (just for MA3C)

As explained in Subsection 3.2.3, the algorithm is learning the different options of the problem (which are being executed one after the other). After doing an 80 million steps training, it was able to reach the third checkpoint (Lstairs) in many runs, while reaching the key in just a few of them. So it successfully learned the policies of options o_0, o_1 and o_2, and it was trying to learn o_3. This is not fully true because the reward of completing o_3 is 110 while the others rewards are just 10, but we can

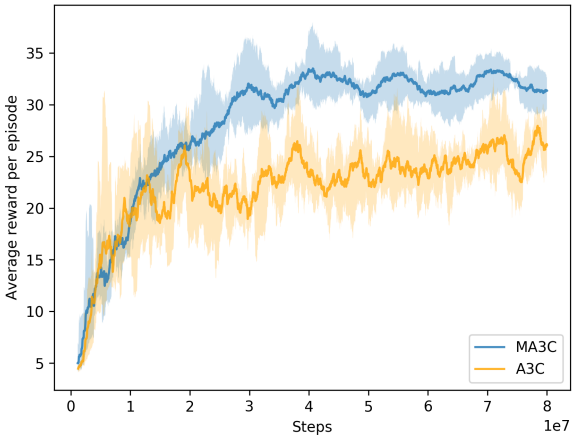

Figure 4.3: Average score over 5 runs of A3C and MA3C algorithms in Simple States environment (80M steps/actions). The graphics have been smoothed by a moving average with a 20 steps window. The shadow represents maximum and minimum values of those 5 smoothed runs.

affirm that with this training process we obtain policies which reach Lstairs with high probability.

As in the other experiments, there might be a different combination of hyperparameters which performs better than in this one, but they are good enough to show the differences between both algorithms.

As can be seen in Figure 4.3, the learning process of MA3C is faster than A3C. Even in complex environments as this one is, the main knowledge can be resumed in the first layers and changing the subgraph is useful to model different subgoals/options. The MA3C curve is also less noisy than A3C, which means that this algorithm is less susceptible to behaviour changes during the learning process.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

First we learned the basics of reinforcement learning, specifically how to model a problem as a Markov Decision Process and how different algorithms that solve this kind of problems work: Q-learning and Asynchronous Advantage Actor-Critic. Later, we did a brief introduction to Artificial Neural Networks, emphasizing on the architecture of the network and how to improve the learning process through transfer learning techniques. We also resumed the state of the art of deep reinforcement learning algorithms to finally propose a variation of those which take advantage of transfer learning methods.

We developed two simple games and modified one of the most famous Atari games (Montezuma's Revenge) to help us understanding the strengths of our new algorithm. We model these games by splitting it into smaller tasks and arranging them in a sequence. We realized that MA3C successfully reused the acquired knowledge of previous tasks in new ones, improving exploration, learning speed and learning stability.

Thanks of being a really general approach it should be easy to replicate in other games or problems, just by identifying how to split them.

5.2 Future work

We have managed subgraph changes of MA3C in a simple manner, but there are more intelligent ways of doing it. We could have any kind of AI algorithm to select

the subgraph which is activated, it would be interesting to analyse the combination of different algorithms with MA3C. For example, MLSH (Frans et al., 2017) is a very similar algorithm which automatically selects the active subgraph.

We made transfer learning through an unsupervised learning approach, but an attractive research would be to transfer the knowledge in a supervised manner. A way of doing it is to have just two subgraphs and using one to enhance exploration when the first one converges. There will be a MA3C network whose first subgraph trains in a problem until convergence and then the second subgraph starts training with all the previous knowledge. Then if this new subgraph finds another better policy we could use the outputs of this policy to train the first network in a supervised manner, i.e. by giving input-output pairs to the first subgraph.

One of the main weaknesses of A3C is the exploration, and we tried to improve it with MA3C, which enhances the exploration implicitly by promoting randomness. We realized that there might be a way of modifying the return equation (Equation (2.3)) ir order to explicitly add rewards for exploring new states.

Bibliography

Abadi, Martín et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. Last retrieved: June 9, 2018. URL: https://www.tensorflow.org/.

Brockman, Greg et al. (2016). *OpenAI Gym*. eprint: arXiv:1606.01540.

Frans, Kevin et al. (2017). "Meta learning shared hierarchies". In: *arXiv preprint arXiv:1710.09767*.

Garriga Alonso, Adrià (2016). *Solving Montezuma's Revenge with planning and reinforcement learning*. URL: https://github.com/rhaps0dy/report-TFG.

Haykin, Simon (2009). *Neural Networks and Learning Machines*. Pearson Prentice Hall.

Hinton, Geoff (2014). *Overview of mini-batch gradient descent*. Last retrieved: June 9, 2018. URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Kavukcuoglu, Koray (2016). *DeepMind moves to TensorFlow*. Last retrieved: June 9, 2018. URL: https://ai.googleblog.com/2016/04/deepmind-moves-to-tensorflow.html.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Mnih, Volodymyr et al. (2016). "Asynchronous Methods for Deep Reinforcement Learning". In: *arXiv preprint arXiv:1602.01783v2*.

Ng, Andrew Y, Daishi Harada, and Stuart Russell (1999). "Policy invariance under reward transformations: Theory and application to reward shaping". In: *ICML*. Vol. 99, pp. 278–287.

Sutton, Richard S and Andrew G Barto (1998). *Reinforcement Learning: An Introduction*. MIT Press.