FEDERAL UNIVERSITY OF RIO DE JANEIRO
MATHEMATICAL INSTITUTE

HUGO SIQUEIRA GOMES

Towards Deep Q-Caching

Prof. Daniel Sadoc Menasche, Ph.D.
Advisor

Prof. Wouter Caarls, Ph.D.
Coadvisor

Rio de Janeiro, Dezembro 2017

Towards Deep Q-Caching

Hugo Siqueira Gomes

Projeto Final de Curso submetido ao Departamento de Ciência da Computação do Instituto de Matemática da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Bacharel em Informática.

Apresentado por:

____

Hugo Siqueira Gomes

Aprovado por:

____

Prof. Daniel Sadoc Menasche, Ph.D.

____

Prof. Wouter Caarls, Ph.D.

____

Prof. João Carlos Pereira da Silva, Ph.D.

____

Prof. Fabrício Firmino de Faria, M. Sc.

RIO DE JANEIRO, RJ - BRASIL

Dezembro 2017

Agradecimentos

Nesta importante etapa da minha vida, gostaria de fazer alguns agradecimentos à pessoas que me ajudaram a chegar até aqui.

À minha mãe Tríccia e ao meu pai José Antonio, pelo carinho e dedicação que sempre me deram e completo apoio nas principais decisões da minha vida.

À minha irmã Maitê, minha eterna dupla e parceira para o que der e vier, pela paciência e motivação durante a vida acadêmica, .

À Mariana, minha companheira e melhor amiga que me deu força e ânimo me acompanhando por toda essa jornada.

Aos meus amigos do curso de Ciência da Computação, em especial aos da turma 2013.2, com quem convivi e aprendi ao longo da graduação e que me proporcionaram a melhor experiência possível em minha formação acadêmica.

Aos professores do Curso de Ciência da Computação que fizeram parte da minha formação acadêmica. Em especial, os Prof. João Carlos e Collier que sempre se mostraram mais do que disponíveis nas principais horas de questionamentos. Gostaria de destacar o Prof. João Carlos que me introduziu em IA.

Agradeço aos Prof. Wouter Caarls e Daniel Sadoc por todo o apoio durante a elaboração deste trabalho. Em particular, o Daniel por ter iniciado a ideia do presente trabalho comigo e de me introduzir na área de estudos de Redes. E, o Prof. Wouter por sua paciência nas suas explicações, por ter me dado a abertura para assistir as aulas na PUC Rio e, por ter me guiado no entendimento do GRL para o desenvolvimento desse trabalho.

RESUMO

Towards Deep Q-Caching

Hugo Siqueira Gomes

Dezembro/2017

Advisor: Daniel Sadoc Menasche, Ph.D.

O aprendizado de reforço profundo atraiu atenção significativa de pesquisadores e profissionais devido à sua eficiência e robustez na busca de boas políticas para agentes de inteligência artificial. No entanto, ainda falta um bom entendimento de como diferentes parâmetros interagem para obter boas soluções. Em particular, pouco se sabe sobre a causa dos algoritmos não funcionarem ou não convergirem para certos problemas. Depurar soluções de aprendizado por reforço profundo é um desafio significativo.

Neste trabalho, o objetivo é estudar o impacto de diferentes parâmetros na qualidade das soluções obtidas através da aprendizagem de reforço profundo. Para isso, foram integradas soluções de aprendizado de reforço profundo em uma biblioteca de aprendizado de reforço geral (GRL). Então, foi investigado como a topologia da rede, a taxa de exploração e os tamanhos do lote influenciam no tempo de convergência. Finalmente, foi modelado o problema de roteamento em cache comum como um problema de aprendizado de reforço profundo e iniciou-se o processo para descobrir como o GRL pode ajudar na obtenção de boas políticas para a futura Internet, sob o paradigma de Redes Orientadas a Conteúdo.

ABSTRACT

Towards Deep Q-Caching

Hugo Siqueira Gomes

Dezembro/2017

Advisor: Daniel Sadoc Menasche, Ph.D.

Deep reinforcement learning has attracted significant attention from researchers and practitioners due to its efficiency and robustness in finding good policies for artificial intelligence agents. Nonetheless, there still lacks a good understanding of how different parameters interact to obtain good solutions. In particular, little is known about why the algorithms do not work or do not converge for certain problems. Debugging deep reinforcement solutions is poses significant challenges.

In this work, our goal is to study the impact of different parameters on the quality of the solutions obtained through deep reinforcement learning. To this aim, we integrate deep reinforcement learning solutions into a general reinforcement learning (GRL) framework. Then, we investigate how network topology, exploration rate and batch sizes impact convergence time. Finally, it was modeled the joint caching-routing problem as a deep reinforcement learning problem, and point towards how GRL can help in obtaining good policies for the future Internet, under the Information Centric Networks (ICN) paradigm.

List of Figures

Figure 2.1: A Neuron Structure 8

Figure 2.2: Examples of Activation Functions 9

Figure 2.3: Deep Feedforward Network Topology 10

Figure 2.4: Simplified scheme of reinforcement learning 14

Figure 2.5: Naive formulation of Deep Q-Network 18

Figure 2.6: Optimized architecture of Deep Q-Network 18

Figure 3.1: GRL configurator 21

Figure 3.2: Control of a pendulum with limited torque 24

Figure 3.3: Control of a pendulum with limited torque 25

Figure 3.4: DNNs models for Pendulum and Cart-Pole tasks: models are ordered from left to right, from (A) to (F). (F) and (E) [rightmost models] are the less complex DNN models. (A) [leftmost model] is the most deeper DNN model. **(A)** is used as **reference model**. 28

Figure 3.5: Inverted Pendulum tested with 6 DDNs architectures 29

Figure 3.6: Cart-Pole tested with 6 DDNs architectures 29

Figure 3.7: Cart-Pole tested with 3 DNNs architectures 30

Figure 3.8: Inverted Pendulum tested with 4 different ϵ-policy 30

Figure 3.9: Cart-Pole tested with 4 different ϵ-policy 30

Figure 3.10: Inverted Pendulum tested with 6 different Minibatch Size 31

Figure 3.11: Cart-Pole tested with 5 different Minibatch Size 31

Figure 3.12: Inverted Pendulum tested with 4 different Update Intervals 32

Figure 3.13: Cart-Pole tested with 3 different Update Intervals 32

Figure 4.1: Current Internet Architecture: ISPs are interconnected with each other, and there are big service providers connected to them. End-users are attached to various ISP networks. 34

Figure 4.2: Example of LRU strategy: The access sequence is ABCDEDF, so when E is accessed, it is a miss. Then, it replaces A because A has the lowest time. 40

Figure 4.3: Network topology. 43

List of Tables

Table 3.1: Basic Parameters 27

Table 3.2: Testing Parameters 28

List of Listings

3.1 Example of Keras code to generate DNNs for GRL 22

List of Abbreviations and Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
ANN	Artificial Neural Network
BP	Backpropagation
CCN	Content-Centric Network
CNN	Convolutional Neural Network
DNNs	Deep Neural Networks
GRL	Generic Reinforcement Learning Library
ICN	Information-Centric Network
INFORM	Interest Forwarding Mechanism
IoT	Internet of Things
ISP	Internet Service Provider.
LRU	Least Recently Used
MEC	Minimum Expected Cost
MDP	Markov Decision Process
ML	Machine Learning
PARC	Palo Alto Research Center
RL	Reinforcement Learning

RNN	Recurrent Neural Network
TCP	Transmission Control Protocol
IP	Internet Protocol
WSN	Wireless Sensor Networks

Contents

Agradecimentos **i**

Abstract **ii**

Abstract **iii**

List of Figures **iv**

List of Tables **vi**

List of Listings **vii**

List of Abbreviations and Acronyms **viii**

1 Introduction **1**

1.1 Contributions 2

1.2 Roadmap 3

2 Background on Machine Learning and Deep Reinforcement Learning **4**

2.1 Machine Learning 4

2.2 Artificial Neural Networks 7

2.2.1 The Single Neuron Model 8

2.2.2 The Multilayer Model 9

2.2.3 The Learning Problem 10

2.2.4 Deep Learning 12

2.3 Reinforcement Learning 13

2.3.1 Q-Learning 15

2.3.2 Deep Q-Network 18

3 Generic Reinforcement Learning Library **20**

3.1 GRL 20

3.2 Deep Q-Learning in GRL 22

3.3 Evaluating Deep Q-Learning 23

3.3.1 The Inverted Pendulum 23

3.3.2 The Cart-Pole Swing-Up 25

3.4 Learning Performance 26

3.4.1 Standard Strategy 27

3.4.2 *DNNs models* 28

3.4.3 *Epsilon* 30

3.4.4 *Minibatch Size* 31

3.4.5 *Update Interval* 31

4 Routing-Caching Problem **33**

4.1 Internet Overview 33

4.2 Information Centric Networking 34

4.3 Q-Routing 36

4.4 Q-Caching 37

4.5 Deep Q-Caching 38

4.5.1 Motivation and Goals 38

4.5.2 Simulator Overview 40

4.6 Preliminary Results 43

5 Conclusion **44**

References **46**

Chapter 1

Introduction

One of the first developments of Artificial Intelligence was achieved by Alan Turing on his paper "Computing machinery and intelligence" [30] when he started to define what is the meaning of "machine" and "think". For decades, groups of different people – computer scientists, engineers, psychologists, biologists, linguists, philosophers - have been trying to establish the principles and concepts that make intelligence possible to a computer. Given the great development of computational and mathematical areas such as Statistics, Computer Science and Optimization, a subfield of Artificial Intelligence, known as Machine Learning, takes a step toward the idea of solving intelligence and has successfully improved performance in several types of tasks (e.g. computer vision and natural language processing).

In the course of further research in Machine Learning, scientists have combined Deep Learning - a broad set of machine learning methods, and Q-Learning - a reinforcement learning algorithm. Deep Q-Learning - as it was named by its creators - was used in different tasks and demonstrated a very optimistic result, especially when applied in digital games [23].

There are two fronts in the present work. In the first one, the Deep Q-Learning algorithm was implemented and integrated into the Generic Reinforcement Learning Library (GRL – Wouter Caarls, Ph.D.) so that its performance could be studied and analyzed in different environments using different parameters, since it's relatively

new and its implementation is not yet consolidated in the literature. This integration was indispensable for it to be tested in a huge range of different conditions instead of being limited to case-by-case implementation. There are many new environments and tasks that Deep Q-Learning can still contribute.

A new paradigm which could be optimized by Deep Q-Learning arose as the second part of this work: Information-Centric Networks (ICN). ICN is an approach to enhance the Internet's infrastructure, which has been focused on the end-to-end principle. It consists of requisition of contents by unique names, without the use of original server location (i.e., IP address), hence allowing caching and replication of the content in the network. With that in mind, it was thought to implement the environment and adapt a new version of Q-Caching [9] using Deep Q-Learning in GRL. Further details will be provided in Chapter Chapter 4.

1.1 Contributions

In summary, the key contributions of this work are the following:

1. **Development and testing of a generic reinforcement learning library (GRL):** we tested and participated in the development of a generic reinforcement learning library which allows easy tuning of parameters to perform experiments using Deep Q-Learning. The library allows to perform sensitivity analysis and to understand the impact of multiple parameters on convergence time and accuracy of reinforcement learning results. In particular, two reinforcement learning problems was evaluated in details using GRL: *Inverted Pendulum* and *Cart-Pole Swing-Up*;

2. **Mapping of the joint caching-routing problem into the Deep Q-Learning framework:** it is show how to map the problem of jointly determining caching and routing decisions in an information centric network into a problem in the framework of Deep Q-Learning. In particular, it is indicated that Q values (i.e., the value function) can be used both for routing and caching decisions. It is also indicated that deep neural networks can be used

to relate decisions on similar contents, avoiding the curse of dimensionality in determining an individual decision per content in the catalog. The proposed solution is referred to as *Deep-Q-Caching*;

3. **Use of GRL to solve instances of the caching-routing problem:** we use GRL to experiment with the joint cache-routing problem. To this aim, we implemented a network simulator under the GRL framework, and executed preliminary experiments with the goal of understanding different network topologies, content request distributions (workloads) and reinforcement learning parameters. Our preliminary results indicate that Deep Q-Caching is a promising solution to cope with the caching and routing of contents in networks oriented by contents.

1.2 Roadmap

The remainder of this work is organized as follows. In Chapter 2, it is described fundamental concepts of machine learning, deep and shallow neural networks as well as how to train them with backpropagation and concepts of reinforcement learning. Then, in Chapter 3 it is evaluate two reinforcement learning tasks. After, in Chapter 4, it is introduced Deep Q-Caching as well as related algorithms. Finally, Chapter 5 concludes.

Chapter 2

Background on Machine Learning and Deep Reinforcement Learning

This Chapter provides a brief description of core elements and terminologies in the current Artificial Intelligence field. It aims to define the purpose of Machine Learning, to show what kind of tasks Reinforcement Learning proposes to solve and understand why Neural Networks became so popular in the last few years. In addition, Backpropagation, Q-learning and Deep Q-learning algorithms will be introduced and explained in details. Some of recent papers about Machine Learning [13], Neural Networks and Deep Learning [25], and Reinforcement Learning [20] are good references to this chapter.

2.1 Machine Learning

Machine Learning (ML) is a subfield of Artificial Intelligence (AI) and is growing to become critical for all fields of AI. The interest in ML has recently surged due to increased computational processing power and the current available data. In the literature, ML usually refers to algorithms which goal is to generalize their current data in order to predict another that it has not seen yet. The idea behind it comes from algorithms that learn from experiences/examples rather than fixed programs that have hard-coded rules. The ultimate goal of all kind of ML algorithms is to

find optimal parameters, i.e, parameters that minimize an evaluation function (also called *objective function*). For this purpose, there are three steps usually used to perform a ML task: data exploration, setting up a model and model evaluation a model.

Data exploration consists of acquiring and treating the data to be used in a specific task. There can be two types of data: Structured Data – generally consists of numerical information, usually text files, which can be easily organized, processed in titled columns and rows, and Unstructured Data – usually binary data which has no easily identifiable internal structure (e.g. audios, images, videos and text messages). The data is usually represented as a vector $X \in \mathbb{R}^n$ where each dimension of this vector is considered a feature for the ML method and each of them should be relevant information depending on the task. In this step, it is necessary to set the kind of data that will be represented or predicted and define the type of the task. This task can be, for example, a Classification Problem – prediction of discrete/categorical variable, or a Regression Problem – prediction of real number/continuous variable. After this, the data is analysed, treated and organized in a dataset with many optional approaches, for example: Missing Values Treatment – complete missing values or balance classes of the dataset, and Outlier Detection – remove observations that appears far away and diverges from an overall pattern in the data. In conclusion, Feature Engineering – algorithms to extract more useful information about your data, is applied.

Setting up a model consists of choosing a learning algorithm and define a structure to embody the data representation. This structure - known as model, is dependent on the data type and the given task. There are a lot of models to different tasks in the literature such as: Support Vector Machines [15], Random Forests [7] and Neural Networks [25]. When the model is chosen, it processes the *training data* - a subset of the dataset. This procedure is called Training Phase where the model is optimized to be used for future predictions. In this phase, a *loss function* is defined to measure the model performance according to its accuracy, for example: a classification error rate. During the training phase, parameters of the model are improved via minimization of the training measured error, just as an optimization

problem.

Model evaluation consists of checking and maybe improve the model based on a measurement. There are another two subsets of the dataset: *validation* and *testing*. The former is used to provide an unbiased evaluation of the model training while tuning model hyperparameters - parameters of the model that can not be changed during the training phase. The latter is used to provide an unbiased evaluation of a final trained model. Using the generalization error - accuracy from the testing dataset, it is possible to measure the error on a new input data. The goal of the machine algorithm is to minimize the error of the training data and the gap between the training error and generalization error. In conclusion, this last step checks the reasons why the algorithm cannot converge to an optimal result and makes changes in the model hyperparameters or even change the model in order to maximize the accuracy if it is necessary.

In addition to these general steps, algorithms or methods of ML can be classified into four different groups that share the same principles: supervised, unsupervised, semi-supervised learning and reinforcement learning. This classification is based on how the training data is provided to the model. In **supervised learning**, there is a pre-defined group of pairs $\langle x_i, y_i \rangle$. Each x represents a feature and each y is called *label* - a value that indicates the output of this configuration of features. Using these pairs, the goal is to learn a mapping from x to y. In mathematical terms, a function h needs to be optimized so that $h(x_i) \approx y_i$. This optimization is related to an error function (e.g mean-squared error):

$$c(\theta) = \sum_i (y_i - f(x_i; \theta))^2 \tag{2.1}$$

Linear Regression, Support Vector Machine, Naive Bayes are example of Supervised Algorithms. They can be used for Image or Speech Recognition and Forecasting, for example.

In **unsupervised learning**, there are no pre-defined labels. We have $\vec{x} = (x_1, \ldots, x_n)$ being a set of i examples (or points) that will be used to achieve the goal of finding interesting structure or representations in the data. The error to be

minimized is the error of reconstruction:

$$c(\theta) = \sum_i (x_i - f^{-1}(f(x_i; \theta); \theta))^2 \quad (2.2)$$

Those algorithms are classified in different groups, such as Clustering Algorithm (e.g. K-means) and Dimensionality Reduction (e.g. PCA, SVD and Autoenconders).

In **semi-supervised learning**, there are labeled and unlabeled data for training. It is halfway between supervised and unsupervised learning methods since the algorithm, besides unlabeled data, is provided with some supervision information for only a (small) subset of your data. Those algorithms are classified in different groups, such as Generative Models, Low-density separation and Graph-based methods.

In **reinforcement learning**, the task is to optimize the agent's behavior in a environment. There are no labels in the data, but the agent is provided with a numerical value feedback known as reward signal. The environment is typically formulated as a Markov decision process (MDP). Those algorithms are classified in different groups, such as Value-based, Monte Carlos and Temporal Difference methods

2.2 Artificial Neural Networks

Automatically learning from data seems promising. In opposition to conventional approach in programming, in Machine Learning there is no need to precisely define rules that the computer can easily accomplish. Rather, the algorithms learn from observational data, optimize its parameters and try to figure out the target numbers. With all that, there is a model called Artificial Neural Networks that tries to take a step forward as we will see in this discussion.

Artificial Neural Networks (ANNs) have been around since the '50s, but until 2006, with the discovery of techniques for learning in Deep Neural Networks (DNNs), it was not known how to train this model to surpass most traditional approaches, just for a few specialized problems that did not require as much data and computing

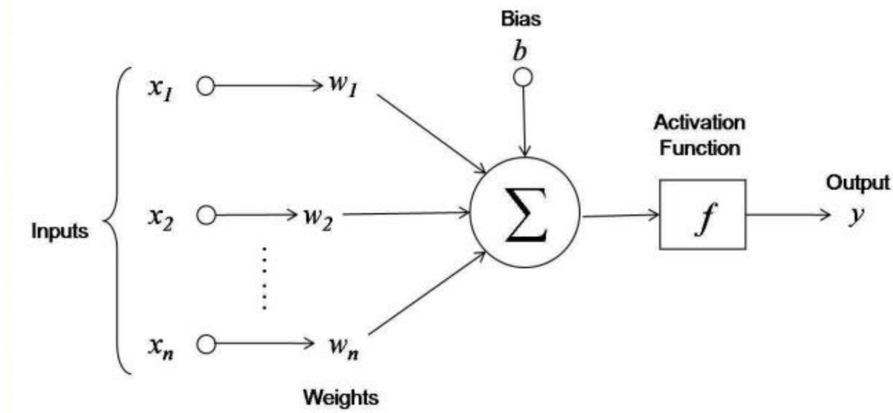

Figure 2.1: A Neuron Structure

power as other problems do now. In consequence of the discovery of many techniques to train them and an increased processing power, it was achieved best solutions to many problems in image, speech recognition and natural language processing - important fields of AI.

2.2.1 The Single Neuron Model

ANNs try to mimic the human brain process. It defines a *neuron* - processing unit consisting of numerical inputs, a body where the inputs are processed and an output, and their *connections* - analogy for biological synapses that has no power of processing, they are only responsible for connecting an output of a neuron with an input of another. A neuron represents a function that maps an input vector $x = (x_1, x_2, \ldots, x_n)$ to a scalar output y via a weight vector $(w = w_1, w_2, \ldots, w_n)$, a bias b and a nonlinear function f (Figure 2.1). The function f takes the weighted sum of the inputs plus the w_0, usually called bias b and returns y:

$$z = b + \sum_{i=1}^{N} w_i x_i \tag{2.3}$$

$$y = f(z) \tag{2.4}$$

where N is the total numbers of inputs and f is a function to provide non-linearity between the input and output known as *activation function.* There are many of them that could be used, but some are represented in Figure 2.2.

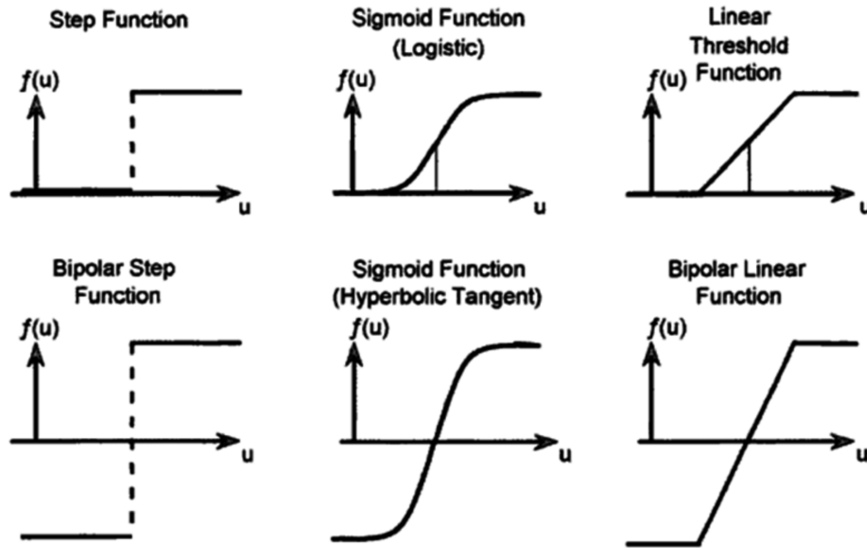

Figure 2.2: Examples of Activation Functions

2.2.2 The Multilayer Model

The way that neurons are configured in a Neural Network is known as network topology. In this work, it is used a topology that is called **feedforward neural networks** where the network is divided in layers where each neuron of each layer is connected with all the neurons in the next layer. Figure 2.3 represents this kind of neural network.

For the purpose of clarity, it is used matrix notation: a_i^k is the output of the neuron i in layer k, w_{ij}^k is the weight j (up-down) of the neuron i associated with the corresponded neuron at layer $k-1$, b_i^k is the bias of the neuron i in layer k and z_i^k is the output of a neuron i in layer k without applying the activation function. Therefore, a^k is a column vector of elements a_i^k. w^k is a matrix of elements w_{ij}^k. b^k is a column vector of elements b_i^k. z^k is a column vector of elements z_i^k.

Using this type of neural network, it is possible to process the input vector throughout all layers, i.e., calculate the final output of it. This process is known as forward pass and can be generalized as:

$$z^k = w^k a^{(k-1)} + b^k \tag{2.5}$$

$$a^k = f(z^k) \tag{2.6}$$

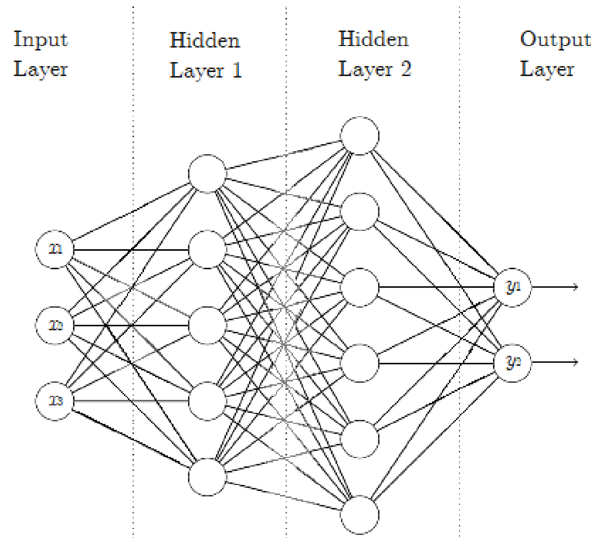

Figure 2.3: Deep Feedforward Network Topology

2.2.3 The Learning Problem

In the feedforward model, each neuron of a ANNs produces an output, using the activation function, based on a set of weights and the outputs of the previous layer. This information flows from left to right, so when a network is feeded with an input vector it processes it and generates an output vector. The learning problem of ANNs consists of finding the optimal combination of weights so that the processing function represents a good approximation of the dataset. For this, it is needed to find a way to measure how good or bad is a set of weights. Thus, it is defined a **cost function**, for example:

$$C(w) = \frac{1}{N} \sum_{i=1}^{N} E(y_i, h_w(x_i)) \tag{2.7}$$

where $a_n(x_i)$ is the output of the last layer of the network with weights w an input x_i, N is the total number of training examples (x_i, y_i) and $E(y_i, h_w(x_i))$ is the error function between y_i and $h_w(x_i)$. Like any optimization problem, the goal is to minimize an objective function:

$$w^* = \arg\min_{w} C(w) \tag{2.8}$$

The first approach could be to find every minimum points in the objective function by solving the derivative of it equals to zero. However, compute the whole

expression for this derivative and solving it is not trivial. The multidimensional parameters makes the process harder since it is needed to find the points where all of those derivatives are zero and there are a lot of minimums points throughout the function. This makes this approach computationally expensive, specially when the size of ANN scales up.

Another approach is *iterative optimization methods.* This class of algorithms do not solve it analytically, they try to follow the derivative of the function in order to find the optimal weight configuration for the network. A basic algorithm for this is **gradient descent**. Each iteration updates the weights w on the basis of the gradient as the follows equations:

$$w_{ij}^{k\prime} = w_{ij}^{k} - \eta \frac{\partial C(w)}{\partial w_{ij}^{k}} \tag{2.9}$$

$$b_{ij}^{k\prime} = b_{ij}^{k} - \eta \frac{\partial C(w)}{\partial b_{ij}^{k}} \tag{2.10}$$

where $\eta > 0$ is the learning rate of the algorithm and $w_{ij}^{k'}$ and $b_{ij}^{k'}$ are the new values of the weights and bias. There is a variation of this method - known as stochastic gradient descent, that is more common to use. It uses mini-batches of the dataset to update the weights rather than calculating the error of all examples as states in Equation 2.7 [5].

The partial derivatives of $C(w)$ is calculated via **Backpropagation** (BP) [16]. The goal with BP is to update the weights of each layer step by step from right to left using partial derivatives, thereby minimizing the error for each output neuron and the network as a whole. From the chain rule, it can be obtained the partial derivative of E - total error of the network, with respect to the last layer L of the network:

$$\frac{\partial E}{\partial w_{ij}^{L}} = \frac{\partial E}{\partial a_i^{L}} \frac{\partial a_i^{L}}{\partial z_i^{L}} \frac{\partial z_i^{L}}{\partial w_{ij}^{L}} \tag{2.11}$$

The second term, equal to $\frac{\partial E}{\partial a_i^L}$ of Equation 2.11 is straightforward to evaluate if the neuron is in the output layer, because $a_i^L = h_w(x)$. However, finding the derivative E with respect to a_i^k of a arbitrary layer k is not so trivial. Then, using

chain rule one more time, it can be defined a relation with the layer $L-1$:

$$\frac{\partial E}{\partial a_i^{L-1}} = \sum_{l \in L} \frac{\partial E}{\partial a_l^L} \frac{\partial a_l^L}{\partial z_l^L} \frac{\partial z_l^L}{\partial w_i^{L-1}} \tag{2.12}$$

Therefore, the derivative can be calculated if all the derivatives with respect to the outputs a^L (next layer) are known. Then, to update the weight w_{ij} or bias b_{ij}, using gradient descent, is just add the product of the learning rate and the gradient, multiplied by -1 (required in order to update in the direction of a minimum of the error function) as mentioned in Equation 2.9 and Equation 2.10:

$$\Delta w_{ij} = \eta \frac{\partial E}{\partial w_{ij}} \tag{2.13}$$

$$\Delta b_{ij} = \eta \frac{\partial E}{\partial b_{ij}} \tag{2.14}$$

To sum up, the final algorithm of BP is:

- Use the *feedforward pass* with the data to get the network output (Equation 2.5).
- for each output node, compute Equation 2.11.
- for each hidden node, compute Equation 2.12.
- update the weights and biases as Equation 2.13 and Equation 2.14.

2.2.4 Deep Learning

The reason why Deep Learning differs from shallow learning is about how the model processes the data. Many of machine learning algorithms have an input and an output layer, and the inputs may be transformed with manual feature engineering before training. Deep Learning adds one or more *hidden layers* - layers between the input and output layer to allow the model to learn hierarchies of concepts and building up abstraction on those layers.

Many applications of deep learning use feedforward neural network architecture as represented in Figure 2.3, which learns a function f to map input X to output Y

with minimal error on the data. The BP algorithm explained before is used to find the weights and bias the same way. These methods have improved the state-of-art in speech recognition, visual object recognition, object detection and many other application such as automatic machine translation and image caption generation.

There are many variations of these typologies such as Convolutional Neural Network (CNN) - usually used in multiple array data (e.g. images), and Recurrent Neural Network (RNN) - usually used in sequential inputs (e.g. texts). Also, there exists a huge number of concepts and techniques to train these deep models. Dropout [26] and Batch Normalization [17] are examples of them.

2.3 Reinforcement Learning

Reinforcement Learning (RL) is a learning problem that refers to optimize a controller of a system, i.e., optimize its behavior in some environment, to achieve a maximum numerical value which represents a long-term objective. RL differs from from supervised learning which there is a target label for each training example, and unsupervised learning which has no labels in the training data. It uses sparse and time-delayed signals - the **rewards**, for learning and the online performance is also important, i.e., the evaluation of the system is often concurrent with learning. A standard RL model example is shown in Figure 2.4: On each time step, the **agent** receives an observation as input, $o(s)$; the agent then chooses an **action**, $a(t)$, to interact with the environment and receives a numerical value, reward $r(t)$.

The challenge of RL is to propose algorithms for learning to behave optimally. The agent should optimize a policy $\pi(s)$ that maps the state s to action a. In particular, approximate $\pi(s)$ to $\pi^*(s)$ which is defined as the optimal policy. The environment is considered non-deterministic, that is, when the agent takes the same action at the same state on different occasions, it may result in different next states and/or different rewards.

In order to achieve an optimal behavior, it should be defined how the agent should take the future into account to decide for its action. Because of environment

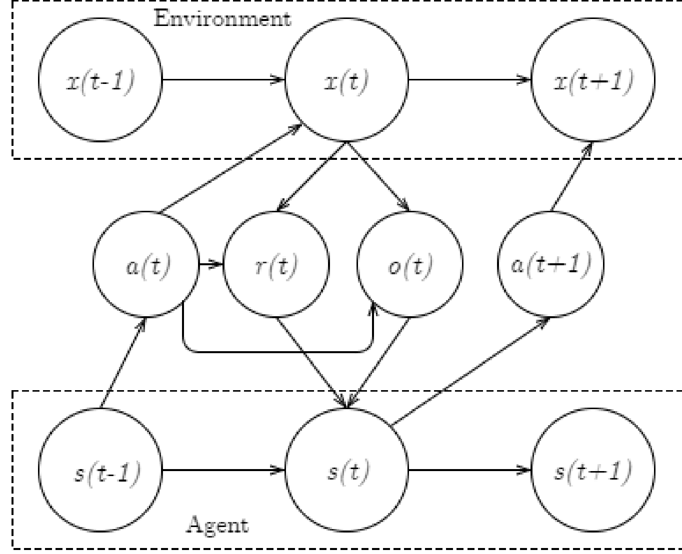

Figure 2.4: Simplified scheme of reinforcement learning

is *stochastic*, the agent can never be sure that it will get the same rewards the next time it takes this action. Therefore, the more into the future the prediction is, the more it may diverge. Hence, in the present work, it will be consider the *infinite-horizon discounted model.* It takes the sum of the rewards of the agent with a geometrically discounted rate according to discount factor $0 \leq \gamma < 1$:

$$\begin{aligned} R_t &= \sum_{t=0}^{n} \gamma^t r_t \\ &= r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + ...)) \\ &= r_t + \gamma R_{t+1} \end{aligned} \tag{2.15}$$

The Equation 2.15 is used to define the *value function* $V(s)$. It returns a numerical value for a specific state and it is defined as the expected total reward that the agent will receive starting from the particular state s. It depends on the policy π that the agent follows. Therefore, the formula of value function is given by:

$$V^{\pi}(s) = E_{\pi}\left(\sum_{t=0}^{\infty} \gamma^{i-1} r_i\right), \forall s \in S \tag{2.16}$$

Using the Equation 2.15, it can be used to establish a relation with the optimal policy $\pi^*(s)$. Since the agent needs to achieve the greatest values of accumulative rewards, it corresponds to the value function that has higher value than other

functions for all states:

$$\forall s \in S : \pi^*(s) = \arg\max_{\pi} V^{\pi}(s) \tag{2.17}$$

There are different groups of algorithms to deal with RL. This present work considers the Deep Q-Learning algorithm [23] because of its optimistic results in learning a control policies in high-dimensional environments. It will be explained in further details in the following sections. However, there are many approaches to solve reinforcement learning problems that are not considered in this work because they fall out of the scope of the work. There are many optimization methods such as *genetic algorithms* and *simulated annealing* that have been used to approach reinforcement learning tasks without using the concept of value function. Value-based algorithms (e.g. Deep Q-learning) requires a Markov Decision Problem (MDP) to model the problem and there are important classes of methods to solve it: Dynamic Programming, Monte Carlo methods and Temporal-Difference learning. For further details, the book [27] is recommended.

2.3.1 Q-Learning

In Watkins' work [31], he proposed a general computational approach based in rewards and punishments for the agent. Q-Learning was conducted considering two main ideas: to minimize the cost of behavior and trial-and-error learning.

The former is related to **Optimal Control Theory** - a field of study which deals with methods to find a control law for a system with a specific optimal rule. One approach to solve this kind of problems was developed in the mid-1950s by Richard Bellman and others [21]. The group of algorithms for solving optimal control problems by Bellman equations (Equation 2.19) is also known as *dynamic programming* It is commonly the only adopted way of solving general stochastic optimal control problems even if it suffers from *the curse of dimensionality* as explained in Bellman's work [4].

The approach of **trial-and-error learning** is related to psychology of animal learning. In Edward Thorndike's work [29], he introduces the idea of *Law of Effect.*

It investigates animals behaviors and concludes that moves who were followed by gratifying consequences tend to happen more often and those that produce undesirable consequences are less likely to occur.

Q-learning is a solution method for an MDP using statistical techniques and dynamic programming methods. Since it directly estimates the utility of taking actions without modeling the environment, it is classified as *model-free* algorithm.

An MDP consists of:

- S = a set of states, with start state $s_{\text{start}} \in S$
- $A(s)$ = a set of actions from state s
- $T(s, a, s')$ = transition function which is the probability of s' if take action a in state s
- $R(s, a, s')$ = reward of the transition (s, a, s')

It is considered that MDP has memoryless property known as **Markov Property**: the probability distribution of next state s_{t+1} of the process depends only upon the current state s_t and action a_t, not on the sequence of events that precede it. One **episode** of the process is defined as a finite sequences of states, actions and rewards:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, ..., s_{n-1}, a_{n-1}, r_n, s_n$$

As a solution in MDP, the agent needs to optimize a policy $\pi(s)$ which maps each $s \in S$ to an action $a \in A$. Intuitively, to find a good aproximation of the optimal policy $\pi^*(s)$, Q-learning is a technique that tries to measure the quality Q of a certain action a in a given state s. It is needed to define a function $Q(s, a)$ that will return the estimate of the maximum discounted future reward (Equation 2.15) that the agent can receive after performing the action a in the state s. It is defined as:

$$Q(s_t, a_t) = \max R(t + 1) \tag{2.18}$$

This function may be counter-intuitive since it returns the final accumulated reward of the episode just knowing the current state and action. But it is possible to define a recursive function:

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a') \tag{2.19}$$

which states that the value Q of the pair $\langle s, a \rangle$ is the immediate reward r - reward that the agent will receive in state s if it performs the action a, plus the maximum future discounted reward that the agent can receive if it chooses the action a' which has the best estimate of future rewards. The γ factor is the discount factor as mentioned before. Using this formula, it is possible to iteratively approximate the Q-function to the real value using an update rule [22].

A simple algorithm for Q-Learning in one episode following the policy $\pi(s)$ that selects the action with maximum Q-value is:

Algorithm 1 Q-Learning

1: initialize Q-table with random values
2: initialize s
3: **while** $s \neq s_n$ **do** ▷ s_n is a terminal state
4: action $a \leftarrow \arg\max_{a \in A(s)} Q(s,a)$
5: reward r, state $s' \leftarrow$ environment.next(a) ▷ act in the environment
6: $Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
7: $s \leftarrow s'$

In this algorithm, the Q-function is a table, with states as rows and actions as columns; α represents the learning rate of the algorithm and γ represents the discounted factor. Note that the the values of each cell in the Q table may be random or without a reasonable meaning leading to wrong values. However, with more episodes and iterations, the estimates of Q-values get more accurate as a consequence of the update rule [22].

The behavior of the policy is very important since it is directly related to the problem in RL: **exploration** vs. **exploitation**. When the policy always chooses the best estimated action, it may not be covering all possible states. It is said the the agent is exploiting the environment. To discover new states, the agent should

use random actions. It is said that the agent is exploring the environment. This is a trade-off relation: if the actions are too random, the agent may not get great rewards very often and if the actions are very greedy, the agent may not find better actions. A simple solution for this is using **ϵ-greedy policy**:

$$\pi_{\epsilon-greedy}(s) = \begin{cases} \arg\max_{a \in A(s)} Q(s,a) & \text{with probability } (1-\epsilon) \\ \text{random action from } A(s) & \text{with probability } \epsilon \end{cases} \tag{2.20}$$

2.3.2 Deep Q-Network

Deep Reinforcement Learning is obtained when Deep Neural Networks is used to approximate the value function, the policy or the model of a reinforcement learning task. In the present work, Neural Networks will be used to approximate the Q-value function of the Q-learning algorithm. It exploits the fact that NNs are very good at coming up with relevant features for highly structured data and it will be essential for Deep Q-Caching. There are two main organization for the DNNs. The first one takes the state and action as input and the corresponding Q-value as output (Figure 2.5). The second one uses only the state as input and outputs the Q-value for each possible action (Figure 2.6).

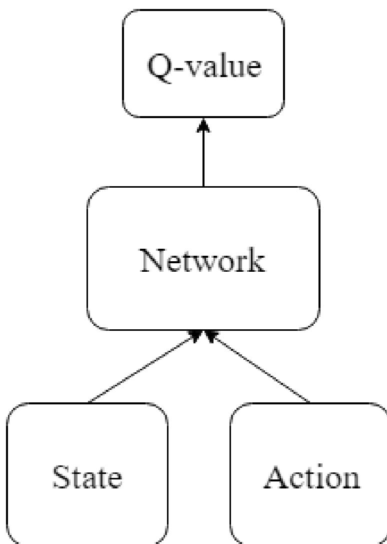

Figure 2.5: Naive formulation of Deep Q-Network

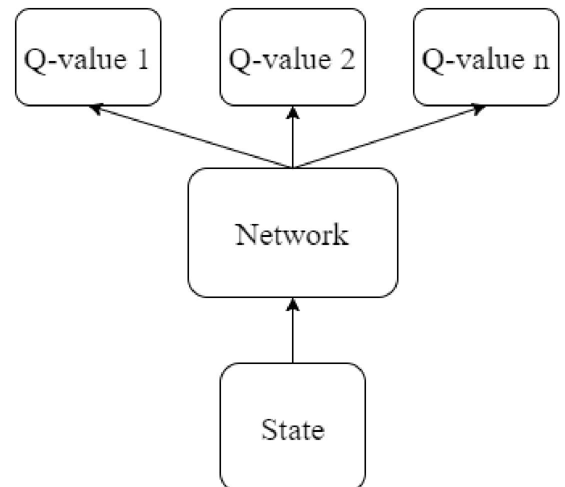

Figure 2.6: Optimized architecture of Deep Q-Network

Algorithm 2 Deep Q-Network

1: initialize Q-network with random weights
2: **while** $s \neq s_n$ **do** ▷ s_n is a terminal state
3: select $a \in A(s)$:
4: with probability ϵ random action
5: otherwise $a \leftarrow \arg\max_{a \in A(s)} Q(s, a)$
6: reward r, state $s' \leftarrow$ environment.next(a) ▷ act in the environment
7: $Q_{\text{target}} = r + \gamma max_{a'} Q(s', a')$
8: train the DNN using Q_{target} as output
9: $s \leftarrow s'$

In Algorithm 2, it is shown a simplified Deep Q-Network algorithm under a ϵ-greedy policy during one episode. Given a state s, it is needed to apply a feedforward pass for every combination (s, a), where $a \in A(s)$ (line 5). Then, it is chosen the action with greatest value when not acting greedily. With the chosen action and moving to next state s', it is possible to get the best estimate Q-value applying feedforward pass for all $a' \in A(s')$ (line 7). Then, using Q-learning update rule and backpropagation, update a new estimate for the pair (s, a).

Deep Q-Networks are very powerful however Q-learning does not converge with such representation. Therefore, Deep Q-learning introduces some differences to maintain convergence in practice: **experience replay** and **separate target network**. The former is a strategy to prevent the network from learning about what it is immediately doing in the environment and allow it to learn a more varied past experience. It stores a fixed size buffer of recent memories while acting in the environment. During the training it draws a uniform batch of random memories from the buffer and train the network with them. The second strategy uses a second network during the training procedure. It is used to generate the target Q-values that will be used to compute the loss for every action during training. The target network's weights are fixed, and only periodically or slowly updated to the primary Q-networks values. Using both strategy, the training can proceed in a more stable manner [24].

Chapter 3

Generic Reinforcement Learning Library

This Chapter provides a brief high-level overview of organization, purpose and structure of the Generic Reinforcement Learning Library (GRL) as well as some implementation details of Deep Neural Networks in GRL. Furthermore, it will get in details of two benchmark problems in the field of reinforcement learning that were used to check the power of Deep Q-Learning in different situations and the correctness of its implementation.

3.1 GRL

Generic Reinforcement Learning Library is a free software developed by Prof. Wouter Caarls during his postdoc at Federal University of Rio de Janeiro in 2015. It is written in C++ programming language and the main purpose of this library is to easily allow different parameters, algorithms and environments to be connected and tested using Reinforcement Learning. It provides a declarative configuration interface (Figure 3.1), letting the user set up a specific experiment.

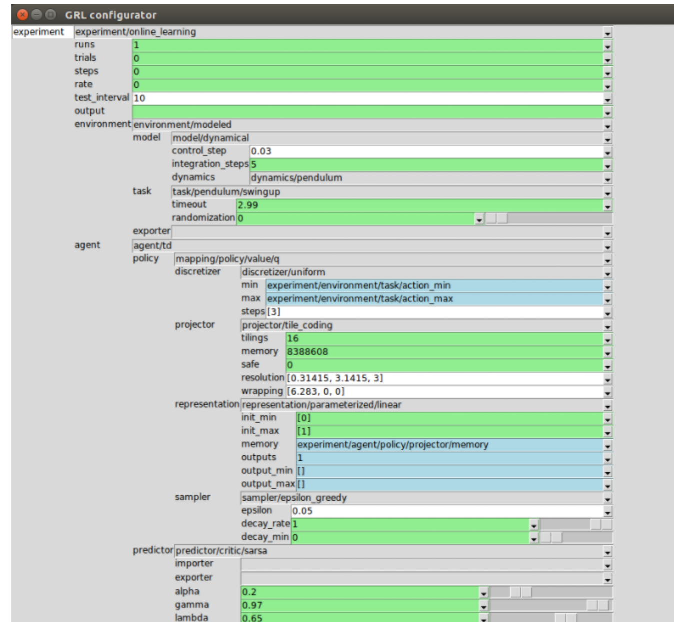

Figure 3.1: GRL configurator

Most classes and interfaces in GRL are organized using the agent-environment interface specified by RLGlue [28] that allows main entities of reinforcement learning or even experiment programs written in different languages to be connected. The implementation and more information about GRL can be accessed in github [1]. More relevant details will be provided in the next sessions.

[1]`https://github.com/wcaarls/grl`

Listing 3.1: Example of Keras code to generate DNNs for GRL

```
import numpy as np
import tensorflow as tf

from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import Adam
from keras.backend import get_session

model = Sequential()
model.add(Dense(30, input_shape=(5,)))
model.add(Activation('relu'))
model.add(Dense(1))
model.add(Activation('linear'))

model.summary()

model.compile(loss='mean_squared_error', optimizer=Adam())

# Must be in this order to enable automatic discovery
model.model._make_train_function()
model.model._make_predict_function()

# Make sure weight assign placeholders are created
weights = model.get_weights()
model.set_weights(weights)

tf.train.write_graph(get_session().graph.as_graph_def(), './', '
    cache_tf.pb', as_text=False)
```

3.2 Deep Q-Learning in GRL

The library seems promising since it provides an easy way to vary parameters and analyse the impact of these changes in the final result of the algorithm. For this reason, Deep Q-Learning was implemented in the GRL. The GRL configurator

provided a faster manner to evaluate this method in different environments. For this integration, two main libraries were necessary: **Tensorflow** [1] and **Keras** [12].

TensorFlow is an open source software library for numerical computation in data flow graphs. It is an ideal programming environment to set up a DNNs models, and its C++ Application Programming Interface (API) contributed to generate computation graphs that GRL could use along with all its elements which have already been implemented.

Keras is a high-level API that enables fast experimentation of different parameters of neural networks. It provided a fast prototyping in Python programming language of all ANNs used in this work. Figure 3.1 shows an example of a script that generates a file graph that could be used in GRL.

3.3 Evaluating Deep Q-Learning

Deep Q-Learning GRL was evaluated within two benchmark problems included in GRL: **Inverted Pendulum** and **Cart-Pole Swing-Up**. They are low-dimensional and continuous control problems that are based on well-defined dynamics of real physical systems.

3.3.1 The Inverted Pendulum

In the Pendulum task [14], the dynamics of the environment (i.e., equations of motion) are unknown to the agent, and it must learn how to behave in the environment only through selecting control actions and the receipt of rewards or penalties.

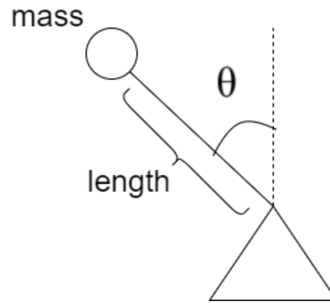

Figure 3.2: Control of a pendulum with limited torque

The Inverted pendulum is a classic problem in Dynamics and Control Theory, and it is used as a test benchmark for control strategy. This task is established by connecting a point of mass m, to the end of a massless rigid rod, of length l, attached to a fixed pivot point at the opposite end to the point. This rod can rotates in a vertical plane and is driven by a rotary electrical motor in the pivot point (this scheme is simplified in Figure 3.2).

The ultimate goal of this task is to put the pendulum at its unstable equilibrium, i.e. the upright position. Since the inverted pendulum is inherently unstable, it is necessary to always activate the control to keep the inverted state. For this application, the pendulum is fixed to only move around a certain axis of rotation so that the degree of freedom is limited to one.

The control voltage is purposely restricted. Therefore the motor does not provide enough torque to push the pendulum up in a single swing. So, the controller has to adapt itself to make the pendulum swing back and forth (destabilized) to gather energy, before being pushed up and stabilized. In the view of Reinforcement Learning (and Control Theory), this creates a nonlinear control problem; furthermore it provides an interesting environment to test Deep Q-Learning.

Since the main purpose of this work does not include presenting the structure of the simulation, as well as its implementation and physical parameters used, it will be indicated reading for further details. All the particulars about this task in GRL is based on the Session 4.5.3 of [8].

Apart from that, there are meaningful elements which are part of the process in optimizing a RL agent. It is necessary to define states, actions, the reward function and the discount factor for this task:

States: The state signal is composed of 2-tuple $\langle$angle θ, angular velocity$\dot{\theta}\rangle$. The angle θ is limited to the interval [-π, π] rad, where θ starts in -π meaning the stable position of the pendulum (down position) and goes up to 0 when it reaches the unstable position (up position). The velocity $\dot{\theta}$ is defined by the interval [-15π, 15π] rad/s using saturation.

Actions: The action is composed of 1-tuple $\langle$voltage $u\rangle$. The action is discrete for the simplification of the problem and it was set as -3V, 0V or 3V.

Reward Function: $R(\theta, \dot{\theta})$ = - 5θ^2 - 0.1$\dot{\theta}$ - u

Discount Factor: The discount factor is γ = 0.97. This discount factor is large since the values of early states during the episode should be influenced by rewards near the goal state.

3.3.2 The Cart-Pole Swing-Up

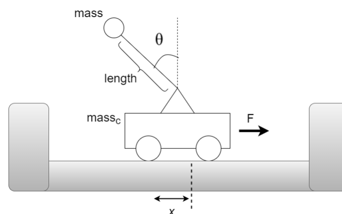

Figure 3.3: Control of a pendulum with limited torque

Next, Deep Q-Learning was applied to a more challenging task: Cart-Pole Swing-Up (Figure 3.3), which is a strongly nonlinear extension to the common Cart-Pole Balancing task [3]. This task is similar to the previous one, except for the pivot point which is not fixed. It is attached to a cart that can move left or right. The pendulum is indirectly actuated by the cart via its acceleration.

The ultimate goal of the cart-pole balancing task is to keep the pole vertically

up oriented by applying left or right directed forces to the cart. The cart has to be moved back and forth in such a way that the pendulum can rotate and achieve a balanced position. As mentioned in the other experiment description, a discussion about the simulation processes and implementation falls outside the scope of this work. Our steps proceed equally in the same way as what is indicated in [11]. The relevant definitions of this task for reinforcement learning agent are:

States: The state signal is composed of 4-tuple $\langle$position x, velocity $\dot{x}$, angle θ, angular velocity $\dot{\theta}\rangle$, where x and $\dot{x}$ are the position and the velocity of the cart. The angle θ is limited to the interval [-π, π] rad, where θ starts in -π meaning the stable position of the pendulum (down position) and goes up to 0 when it reaches the unstable position (up position). The velocity $\dot{\theta}$ is defined by the interval [-15π, 15π] rad/s using saturation.

Actions: The action is composed of a value force F. The action is continuous and $F \in \{$-10, 10$\}$ N.

Reward Function: - θ^2 - $0.1\dot{\theta}^2$ - $2x^2$ - $0.1\dot{x}^2$

Discount Factor: The discount factor is $\gamma = 0.97$.

3.4 Learning Performance

In general, a RL algorithm can be evaluated considering two results: how good the final policy is and how much reward the agent receives while acting in the world.

The final policy may be important when there is sufficient time for the agent to learn safely before being deployed. However, there are many RL applications that suffers from real-world problems (e.g. robotics). Applying reinforcement learning in robotics demands safe exploration due to high cost which becomes a key challenge of the learning process. Due to this fact, the reward received while learning may be what the agent wants to maximize.

In this work, it is evaluated those two results. Inverted Pendulum and Cart-Pole Swing-Up as presented are safe tasks even when executed in the real world. So, the

performance of Deep Q-Learning for these tasks can be evaluated considering the final policy. But the learning time of the algorithm in those environments is also important to infer how it will behavior in more complex environments.

In this section, it is reported the evaluation of Deep Q-Learning in Inverted Pendulum and Cart-Pole Swing-Up environments. A standard strategy was used to make all experiments and it will be described in Section 3.4.1. The following subsections show the impact between different parameters of the training.

3.4.1 Standard Strategy

In all cases, the learning trials were interrupted by regular test trials, in which exploration was set to zero. Those test trials are analyzed in a graph that represents the accumulate reward as a function of time shown as the mean and 95% confidence interval over 10 independent runs.

They were compared to two another shallow RL algorithms: SARSA and Q-Learning. The former is an algorithm for learning a MDP policy as well as Q-learning which was explained in Section 2.3.1. The end performance is characterized as the average over the last 10 test trials of all runs. In Pendulum task, the end performance of SARSA is -813.94 and Q-learning is -848.87. In Cart-Pole task, the end performance of SARSA is -3211.42 and Q-Learning is -3579.09.

The basic parameters for the pendulum and cart-pole were derived from the originating papers and are summarized in Table 3.4.1. It was chosen three parameters to vary in each simulation (epsilon, minibatch size, update interval) and 6 DNNs models. Those models are represented in Figure 3.4 and each parameter in Table 3.4.1.

Task	α	γ	λ
Inverted Pendulum	0.02	0.97	0.65
Cart-Pole Swing-Up	0.02	0.97	0.65

Table 3.1: Basic Parameters

Parameters						Reference Values
Epsilon ϵ	0.05	0.1	0.3	decay [a]	-	**decay**
Minibatch Size	16	32	64	128	512	**128**
Update Interval	100	500	1000	2000	-	**1000**

Table 3.2: Testing Parameters

[a]epsilon starts as 1 with decay of 0.995 and min limit of 0.01

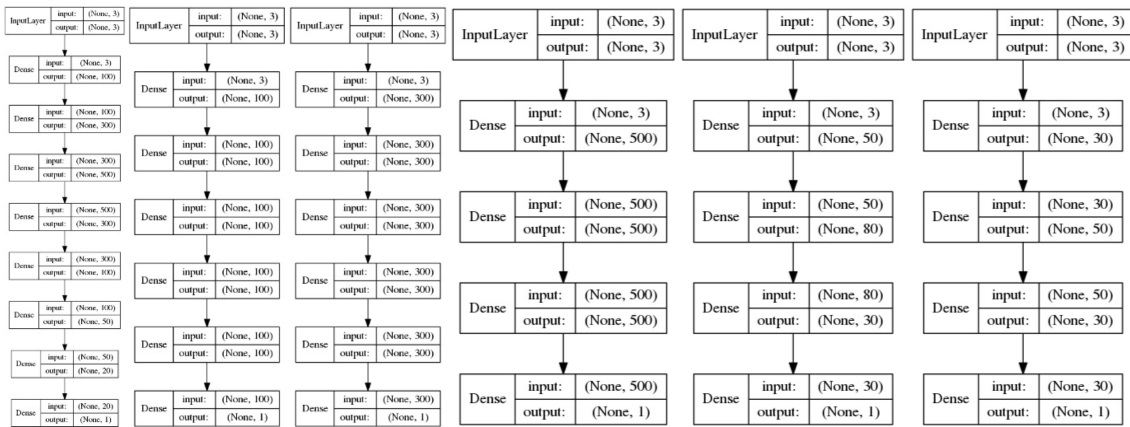

Figure 3.4: DNNs models for Pendulum and Cart-Pole tasks: models are ordered from left to right, from (A) to (F). (F) and (E) [rightmost models] are the less complex DNN models. (A) [leftmost model] is the most deeper DNN model. **(A)** is used as **reference model**.

3.4.2 *DNNs models*

It was compared the six architectures presented in Figure 3.4. The last two architectures (Figure 3.4, two rightmost architectures) are the simplest requiring more computation time to achieve the desired levels of accuracy. From left to right, the models in Figure 3.4 are roughly ordered in terms of decreasing complexity.

Figure 3.5 and 3.6 report the accumulated reward as a function of training time. For the simpler problem (inverted pendulum), it was tested the six network topologies and verified that indeed the leftmost architecture took longer to converge. This

is because it is very challenging to make such a simple network converge to an optimal policy.

For the most challenging problem (cart-pole problem), it was again tested all topologies, and observed the same behavior as in the inverted pendulum problem. To simplify visualization, Figure 3.7 shows result from the first three leftmost models of Figure 3.4. For such three models, it was verified that the first topology took longer to converge. Now, because these three topologies have somewhat the same complexity, it was observed that the simpler topology (which has less parameters to be optimized) converged faster.

This simple exercise serves to illustrate a very fundamental tradeoff in the choice of the DNN architecture subsumed by deep reinforcement learning solutions: for the highly simplified DNN architectures, with the larger the generalization power, it takes longer for the training phase to converge (if there is convergence at all). For the more complex architectures, the networks with fewer parameters may converge faster.

In Pendulum task, the end performances were: -813.81, -848.55, -831.16, -890.64, -848.19, -959.64. In Cart-Pole task, the end performances were: -357.67, -483.68, -488.59, -425.79, , -1352.03, -1744.40.

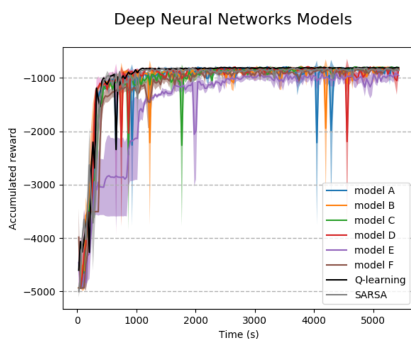

Figure 3.5: Inverted Pendulum tested with 6 DDNs architectures

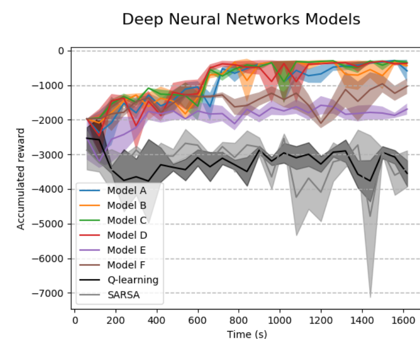

Figure 3.6: Cart-Pole tested with 6 DDNs architectures

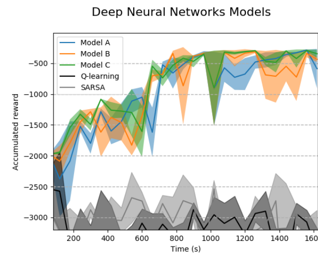

Figure 3.7: Cart-Pole tested with 3 DNNs architectures

3.4.3 *Epsilon*

Next, it was consider the impact of impact of ϵ (the exploration rate) on the time to convergence. Figures 3.8 and 3.9 show the impact of ϵ for the inverted pendulum and cart-pole problems, respectively. While most of the experiments used a fixed value of ϵ, we also experimented with a decaying ϵ value (denoted by the label "decay" in the figures). It is interesting to note that the values of ϵ with which we experimented did not show a clear impact on the time to convergence. This indicates a level of robustness of the solution with respect to this meta parameter. Note that even though it was observed this small sensitivity of the results with respect to ϵ, it was kept a decaying ϵ for the remainder of the experiments, as this was also adopted in previous works in the literature, e.g., [23].

In Pendulum task, the end performances were: -853.11, -861.52, -854.72, -822.07. In Cart-Pole task, the end performances were: -426.09, -377.05, -434.35, -416.14.

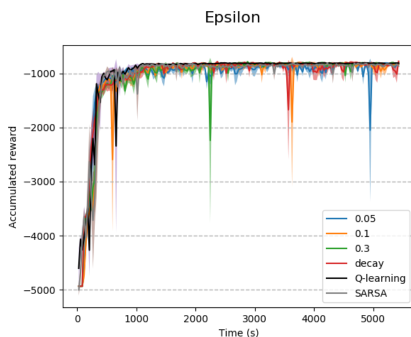

Figure 3.8: Inverted Pendulum tested with 4 different ϵ-policy

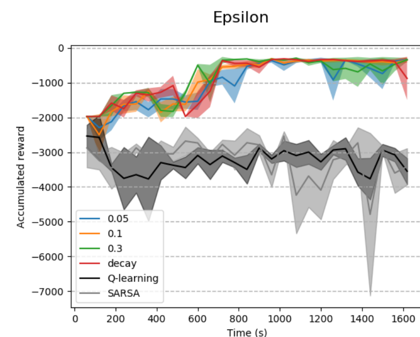

Figure 3.9: Cart-Pole tested with 4 different ϵ-policy

3.4.4 *Minibatch Size*

Next, we consider the impact of the size of the minibatch on the convergence rate. Figures 3.10 and 3.11 show the convergence time for minibatches of size 16, 64, 512 and 1024, under DNN topology F, for the inverted pendulum and the cart-pole problems, respectively. It was noted that as the minibatch size increases, the time to convergence decreases. This monotonicity may be due to the fact that, for the problems considered, it was always possible to extract information from previous samples as the training evolves, without leading to overtraining. It was envisioned that a further increase in the batch size may eventually decrease performance, but it was left this analysis as subject for future work.

In Pendulum task, the end performances were: -955.28, -837.38, -854.72, -817.85, -797.26, -804.80. In Cart-Pole task, the end performances were: -1049.36, -659.20, -493.85, -347.02, -398.90.

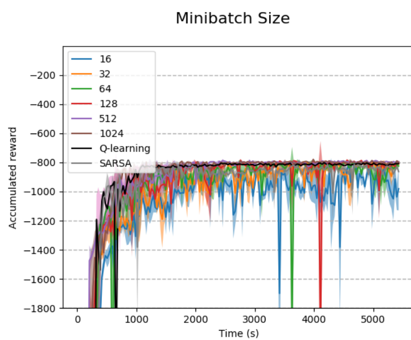

Figure 3.10: Inverted Pendulum tested with 6 different Minibatch Size

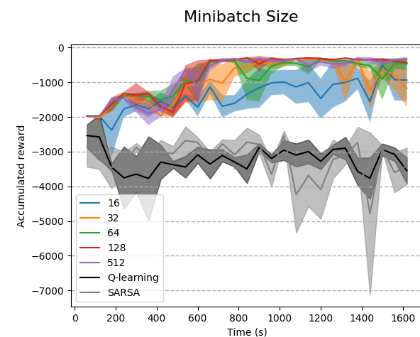

Figure 3.11: Cart-Pole tested with 5 different Minibatch Size

3.4.5 *Update Interval*

Next, it was considered the impact of the time between updates on convergence. It was simulated considering updates every 100, 500, 1000 and 2000 episodes. As shown in Figures 3.12 and 3.13, the less frequent the updates, the faster the convergence. The comments made in the previous section are applicable here as well.

For the problems considered, it was always possible to extract information from the collected samples inbetween two training executions, without leading to overtraining. However, it was envisioned that further increasing the training frequency may degrade performance, specially if it is accounted for the training cost (in terms of CPU), which has not been considered here, and is left a subject for future work.

In Pendulum task, the end performances were: -809.43, -799.76, -823.24, -848.87. In Cart-Pole task, the end performances were: -416.26, -486.39, -350.18.

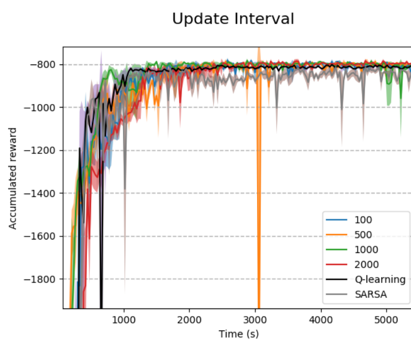

Figure 3.12: Inverted Pendulum tested with 4 different Update Intervals

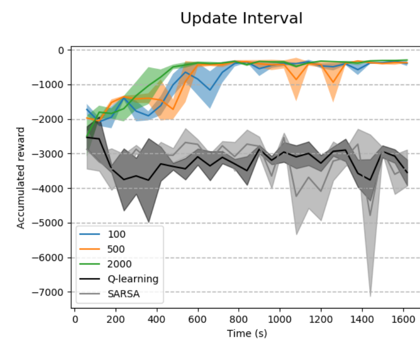

Figure 3.13: Cart-Pole tested with 3 different Update Intervals

Chapter 4

Routing-Caching Problem

There are substantial changes happening in the way we use the Internet today. New methods and algorithms are emerging to bypass inefficient operations implemented in the traditional Internet organization. In this chapter, a new algorithm named Deep Q-Caching, which combines the concepts of reinforcement learning and deep neural networks, is proposed to solve routing and cache challenges in an integrated and scalable fashion.

4.1 Internet Overview

The principles of the Internet in the 1970s were made to set up one-to-one connections between two machines - one possessing resources and the other needing access to those resources. It has been done by giving addresses to end-points allowing them to locate and connect to the destination address (i.e., IP address). The essential architecture of today's internet are rooted in those principles and is represented in Figure 4.1.

However, the purpose of the Internet went through many changes. It has become not only professional but a personal tool in our lives: sending e-mails, booking hotels, streaming videos, looking for photos, and accessing your social media. Those contents are in general a huge amount of videos, web pages and images flowing from

content providers to viewers. All these activities would call for an Internet as a large content distribution network.

The current host-driven Internet architecture presents imminent problems when the goal is to transmit multimedia content. In particular, the Internet has to cope with multiple host redirections and inefficient operations to resolve a number of queries. In a long a term, such inefficiencies may make the system non-sustainable, producing an increasing amount of overhead in terms of processing power to keep it working. As a consequence, new network environments have appeared where the traditional TCP/IP communication model is not always the best solution. The idea of a source-destination communication does not fit in scenarios like Internet of Things (IoT) or Wireless Sensor Networks (WSN), for example. This is the point where Information-Centric Networking (ICN) steps in.

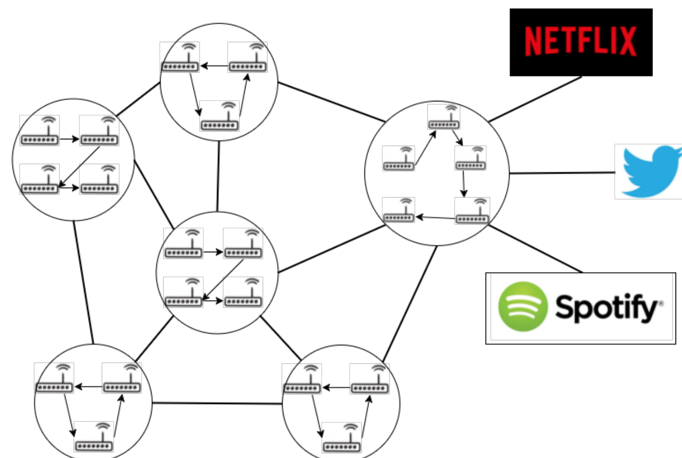

Figure 4.1: Current Internet Architecture: ISPs are interconnected with each other, and there are big service providers connected to them. End-users are attached to various ISP networks.

4.2 Information Centric Networking

The current Internet is more content-centric than location-centric, i.e., Internet users are more interested in what is the content instead of where it is from. ICNs use this idea to redesign the current outdated Internet's organization. Therefore, ICNs have received much attention and have shown optimistic results regarding

performance and scalability on Internet infrastructure [2].

ICNs are based on two ideas: each piece of content has a unique name and caching is universal in the network. The network functions are operated in terms of those named piece of contents instead of locations. Thus, in ICN paradigm, each network node is a content-provider. These nodes – routers, can store replicas of contents which are identified by its unique name, in their cache memory. Every router receives, sends to the next hop until it reaches its destination and may also store the data in its cache. The advantage is clear since a client can be served faster by an intermediate router than a final server.

Although many different architectures have been proposed for ICNs, the chosen architecture implementation was done by using a simplified idea of Content-Centric Network (CCN) architecture: a project introduced by Van Jacobson in his Google Tech Talk [18] and initialized as a project by Palo Alto Research Center (PARC) [19]. Even though all aspects of CCNs are very important in a real-world scenario, some of them are not relevant for the scope of the present work: achieving a good content name design, the security of user's content, transport-layer functionalities that considers realistic use cases are example of that. Besides them, the GRL caching simulator uses the same core idea in CNNs: deliver contents to users not specifying the location from where it was obtained, but instead it replicates the content on multiple servers within the network system. The implementation details will be described in following sections.

With those requirements and the final version of simulator, it is possible to tackle two well-known problems in ICNs: finding the optimal routing policy and the optimal cache replacement policy. Routing problem is a decision making process of selecting the optimal path to transmit a data (packets) from a source to its destination. In ICN case, it answers the question: to which adjacent router should the current router send its content to get it as fast as possible to its eventual destination? Caching problem is a decision making process of selecting what content will be removed (or not) from the cache memory when it is full. In ICN case, it answers the question: what content this router should remove from its cache memory in order

to minimize the download time of future requests?

ICN turns out to be extremely dynamic since the location and availability of the contents change over time due to those caching and routing decisions made by the routers. This intrinsic characteristic gave rise to many approaches for those two problems aforementioned. An extension of Q-routing [6] was proposed to address the problem of routing in this kind of network [10] and Q-caching [10], built on top of Q-routing, uses the information that is already collected by the routing algorithm to optimize the caching problem. Since this present work aims to extend Q-caching algorithm with deep neural networks, those strategies will be described in further details in next sessions.

4.3 Q-Routing

Q-routing [6] is an algorithm which attempts to find an optimal routing policy for a dynamically changing traffic and topology network using reinforcement learning. It is a variant of Q-Learning [31], which uses Q-functions to learn a state representation of the entire network in an asynchronous way: each node n in the network maintains a routing table with all estimated delivery time of a packet towards other nodes. Those estimates are represented as Q-values, where $Q_x(d, y)$ represents the delivery time of a packet from node x to a destination node d if the packet is forwarded via node $y \in \text{neighbours}(n)$, the set of all neighbors of node x. The forwarding actions may be done by using a *locally greedy* routing decision, i.e., when a node x receives a package to a specific destination d, it chooses the neighboring node y^* for which $Q_x(d, y^*)$ is the lowest value compared to all possible Q-values of x's neighbors. Upon sending a package P to y^*, x immediately gets back the best Q-estimation t for P by y^*, namely:

$$t = \min_{z \in \text{neighbours}(y)} Q_{y^*}(d, z) \tag{4.1}$$

When the node x receives the time t from y, it can update the related Q-value to a new estimate as follows:

$$Q_x(d,y) \leftarrow Q_x(d,y) + \eta \left(\overbrace{s+t}^{\text{new estimate}} - \overbrace{Q_x(d,y)}^{\text{old estimate}} \right) \tag{4.2}$$

where η is a *learning rate* parameter and s is the units of time in transmission between node x and node y.

In recent work, [10] proposed an extension version of Q-routing modeled for ICNs known as INterest FORwarding Mechanism - *INFORM*. Instead of estimating a Q-value for each destination, each router (node) of the network estimates the delivery time for each content. It replaces the previous table by one that holds a set of $Q_r(c,y) \forall y \in neighbours(r)$, where neighbours($r$) denotes the interfaces (neighbors) of router r, which represents the cost (reward) in terms of residual delay to the first hitting cache for a content c sent by router r via y. The updated rule is analogous to the non-extended Q-routing version.

4.4 Q-Caching

Q-caching [10] is a caching strategy that aims to reduce the average download time experienced by users in ICNs. It is built on top of Q-routing taking advantage of the estimated delivery time of a content, i.e., the related Q-value. They are used not only as a routing decision but also as a caching decision.

In Q-routing work [10], least recently used (LRU) policy was implemented as a cache replacement algorithm. This algorithm is briefly explained in Figure 99. LRU presents a good performance although the use of the estimated Q-values can bring improvements in the cache decision. It is established in Q-caching work [10] using the following strategy: the most difficult items to be obtained (highest Q-values) are cached in order to minimize the waiting time for a content. Hence, the router sorts all contents according to their expected cost and cache them based on the highest values. As a consequence, items that has a *minimum expected cost* (MEC) are evicted. MEC calculates the expected cost by multiplying the waiting time (Q

value) with the probability of receiving a request for that item, obtained through request counting.

It is important to highlight that routing and caching decisions influence each other in this scheme. When caching a content, the route that is followed by a specific content is adjusted, then, it changes the requests seen by other routers in the network and, consequently, their cache is altered. Therefore, using the same Q-values to routing and caching, Q-caching provides a better flexibility compared to LRU, which is limited to the content hit rate. For example, while LRU stores all contents that passes through the cache, Q-caching controller might not store it. Those options can lead to a better global optimization of the network.

4.5 Deep Q-Caching

As a next step towards a global optimization of an ICN-based network, this work presents not only a flexible environment modeled in GRL but also a new step for Q-caching. To develop Deep Q-Caching, it was exploited the content diversity feature of Q-caching, some disadvantages of Q-caching were improved and a real-world based environment were implemented.

4.5.1 Motivation and Goals

There are a sequential evolution of researches related to routing and caching optimization which were exposed in past sections of this work. Q-routing was introduced as a reinforcement learning approach to optimize a routing policy. In this algorithm, each router learns to estimate the time to reach a specific destination given the connected interface. INFORM [10] adapted Q-routing for ICNs: each router learns to estimate the delay time for each requested content, without specifying a location, given the connected interface. It also combines this adapted version of Q-routing with LRU for cache decision. Lastly, Q-caching uses the calculated Q values of INFORM to make better cache decisions, integrating routing and caching in a simple but effective way. To sum up, every work has made some improvements

in all the previous proposals.

Deep Q-caching is no exception. It is built on top of Q-caching and shares the same goal: to find an optimal routing and caching policy. However, even if Q-caching is able to move toward this goal, it might not be scalable in a real-world scenario. It estimates the cost to obtain each content independently and, consequently, each new content in the network has to be learned. The experience required to fill that space is prohibitive since each router needs to accumulate experience for each content. Because of this, it seems reasonable to think about a better representation of the contents in order to generalize the data. It would turns possible the estimation of delay time of new incoming contents.

Another relevant feature of Q-caching has not been discussed yet: *Content Diversity*. It concerns about how the contents will be distributed over the network after a period of time. Considering LRU strategy (Figure 4.2), a cache can only be affected with the missing contents of its neighbors: a router x can only send a request for content c to its neighbor router y if it does not have c in its current cache. The reason of this limitation is that this strategy always stores the most frequently requested contents and, therefore, the caching decision is not directly affected by the states of the neighbors. It differs from Q-caching strategy. In this case, the Q-value for caching a content c represents the distribution in upstream routers. Thus, the caching decisions are affected by the states of close routers since close contents has low cost so the router might not store this content. Such a difference increases content diversity reducing the expected download times.

Deep Q-caching, inspired by Q-Caching, faces the same challenge of promoting different content in different caches as a way to decrease the average download time. As aforementioned, the distribution of a content used as weight is not scalable. So Deep Q-Caching takes a different path with Deep Neural Networks. This model often generalize the input data very well and ICNs provide a infrastructure that permit to treat all requested contents as features.

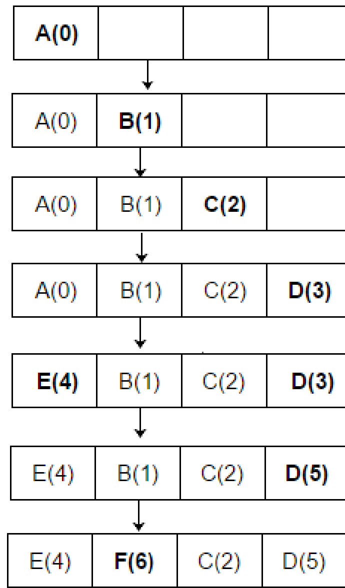

Figure 4.2: Example of LRU strategy: The access sequence is ABCDEDF, so when E is accessed, it is a miss. Then, it replaces A because A has the lowest time.

4.5.2 Simulator Overview

Algorithm 3 Event Simulator

1: **function** SIMULATE ▷ Main function (discrete event simulator)
2: network ← build network from "network_topology.txt"
3: queueOfEvents ← read events from "event_list.txt"
4: **while** queueOfEvents $\neq \emptyset$ **do**
5: event e ← queueOfEvents.top() ▷ an event comprises an interface ▷ and a content
6: queueOfEvents.pop()
7: interface i ← e.interface()
8: interface.process(e)

There are three main ideas which the simulator uses: event queue, ICN and Deep Q-Caching. The first one is briefly explained in Algorithm 3. The simulator reads the network topology using a text file which is an graph adjacency matrix (Line 2). After this, there is another file that handle all the events (Line 3). Each event is a line in the text file with numbers: the first part of the numbers is the features of the contents and the second part is the router id that will create this request. So, [245 623 123 2] in *event_list.txt* can be read as "client which is connected in router

2 want the content [245 623 123]". After this, the simulator enters in a while loop and gets the next event each iteration until all the requests finishes (Line 4). Each iteration, it processes the event in the interface specified (line 7 and 8).

The second and third idea are represented together in Algorithm 4. Each router is composed of interfaces, tables and cache memory. It uses a table to remember what requests from others routers it received and sent to another one. So, if router A requests a content to router B and B sends this requests to C, B needs to know by which interface this content has been sent (in this case, A). This table is known as PIT (Pending Interest Table). There is another table called requisitions that stores requests which started in this router. Every router has a list of interfaces to connected routers. So, it will only send or receive a message through the interfaces. There are three types of messages: request, acknowledge and data. The first step of the router is to check the type of the message to process. Each type of message has a different process function. They are described in Algorithm 4.

processRequestArrival happens when another router sends a request. First, the router checks if it has the content in its cache (Line 4). Then, if there is the content, it creates a new message with the content and add in the simulator queue returning via same interface (Line 7). If there is not, the router needs to make a routing decision (Line 9). It gets the interface with minimum estimate delay time (i.e. minimum Q-values) and add in the simulator queue a request message in the chosen interface (Line 11). As a consequence of receiving a request content, the router needs to send an acknowledge message via the same interface (Line 13).

processContentArrival happens when another router sends a content. First, router checks if it had made the requisition (Line 4). If positive, the router just delete the requisition (because it earned the content) and return. If negative, it looks to pit table to see which interface made this requisition and send through them (Line 7, 14 and 15). During this time, it has to make a caching decision (Line 10). If there is a content in cache that has smaller delay time than the current content, then it is added in cache (Line 12). Note that when the data arrives, the router can calculate the true q-value of this content via the pit table which has the

time it made the prediction, so it can train the DNN with those values (Line 13).

processAckArrival happens when another router sends the q-value. The router gets the q-value and train the DNN with respect to the content that was requested to another router.

Algorithm 4 Deep Q-Caching

```
 1: function processRequestArrival(event)          ▷ Content request received
 2:     message ← event.message
 3:     interface ← event.interface
 4:     if message ∈ router.cache then
 5:         new_message ← create message as "data" message
 6:         new_event ← (new_message, interface)
 7:         queue.add(new_event)
 8:     else
 9:         destination, q-value ← minQ()
10:         new_event ← (message, destination)
11:         queue.add(new_event)
12:         new_message ← create message as "ack" message using q-value
13:         new_event ← (new_message, interface)

 1: function processContentArrival(event)    ▷ Data received from another
    router
 2:     message ← event.message
 3:     content ← message.data
 4:     if content ∈ router.requisitions then
 5:         router.requisitions.delete(message.id)
 6:     else
 7:         destination ← router.pit(message)
 8:         router.pit.delete(message.id)
 9:         prediction q ← agent.predict(content)
10:         if q > cache.min() then               ▷ Check if content goes to cache
11:             cache.remove()
12:             cache.add(content)
13:         agent.fit(content, true q-value)
14:         new_event ← (message, destination)
15:         queue.add(new_event)

 1: function processAckArrival(event)▷ Q value received from another router
 2:     qvalue ← event.q
 3:     agent.fit(content, qvalue)
```

4.6 Preliminary Results

To appreciate the performance of the proposed Deep Q-Caching algorithm, it was ran a sample simulation of a simple network, with 7 routers, where each of the 7 routers has a corresponding agent which makes caching and routing decisions. In addition, there is a custodian server which stores all contents. The topology is illustrated in Figure 4.2.

At each time slot, there is a single content request. Each router estimates the cost-to-go to obtain a content from the catalog. Such estimates evolve over time, and converge as illustrated in a video made available at Youtube: `http://tinyurl.com/hugorlvideo`

As our next steps, we intend to reproduce the results presented in the previous chapter under this new domain. In particular, we plan to study the impact of different parameters on the convergence rate, and to compare Deep Q-Caching against other proposals in the literature.

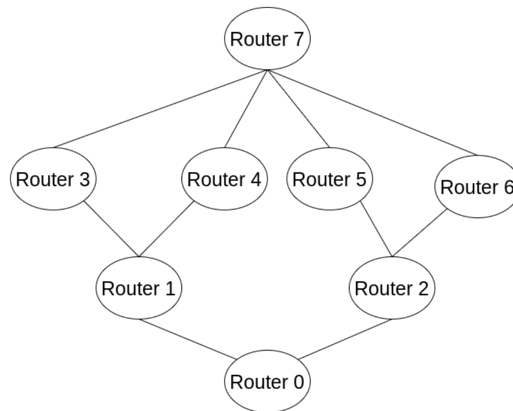

Figure 4.3: Network topology.

Chapter 5

Conclusion

In this work, it was investigated different factors that impact the performance of deep reinforcement learning solutions. First, it was developed and tested in a generic reinforcement learning library (GRL). The library allows to perform sensitivity analysis and to understand the impact of multiple parameters on convergence time and accuracy of reinforcement learning results. In particular, it was evaluated two reinforcement learning problems in details using GRL: *Inverted Pendulum* and *Cart-Pole Swing-Up*. Then, the problem of joint caching-routing problem was mapped into the Deep-Q-Learning framework, giving rise to Deep-Q-Caching. In particular, it was indicated that Q values (i.e., the value function) can be used both for routing and caching decisions. We also show that deep neural networks can be used to relate decisions on similar contents, avoiding the curse of dimensionality in determining an individual decision per content in the catalog. Finally, we gave a first step towards using GRL to solve instances of the caching-routing problem. Our preliminary results indicate that Deep-Q-Caching is a promising solution to cope with the caching and routing of contents in networks oriented by contents.

This work opens up several avenues for future research.

1. **Further analysis of Deep-Q-Caching:** we plan to reproduce the sensitiviy analysis results of *Inverted Pendulum* and *Cart-Pole Swing-Up*, now in the domain of Deep-Q-Caching.

2. **Expand investigation of parameter space:** we plan to expand the parameter space on top of which we investigate textitInverted Pendulum and *Cart-Pole Swing-Up*, to find tradeoffs, e.g., in the choice of the minibatch size (does the performance degrade as the minibatch size becomes too large? or is it always monotone? what about the training frequency?)

3. **Automatic parameter tuning:** we plan to use the knowledge gained in this work to give recommendations on how to set meta-parameters for reinforcement learning solutions.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., e Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Ahlgren, B., Dannewitz, C., Imbrenda, C., Kutscher, D., e Ohlman, B. A survey of information-centric networking. *IEEE Communications Magazine 50*, 7 (2012).

[3] Barto, A. G., Sutton, R. S., e Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, 5 (1983), 834–846.

[4] Bellman, R. The theory of dynamic programming. Relatório técnico, RAND CORP SANTA MONICA CA, 1954.

[5] Bottou, L. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[6] Boyan, J. A., e Littman, M. L. Packet routing in dynamically changing networks: A reinforcement learning approach. In *Advances in neural information processing systems* (1994), pp. 671–678.

[7] Breiman, L. Random forests. *Machine learning 45*, 1 (2001), 5–32.

[8] Busoniu, L., Babuska, R., De Schutter, B., e Ernst, D. *Reinforcement learning and dynamic programming using function approximators*, vol. 39. CRC press, 2010.

[9] Caarls, W., Hargreaves, E., e Menasché, D. S. Q-caching: an integrated reinforcement-learning approach for caching and routing in information-centric networks. *arXiv preprint arXiv:1512.08469* (2015).

[10] Caarls, W., Hargreaves, E., e Menasché, D. S. Q-caching: an integrated reinforcement-learning approach for caching and routing in information-centric networks. *arXiv preprint arXiv:1512.08469* (2015).

[11] Caarls, W., e Schuitema, E. Parallel online temporal difference learning for motor control. *IEEE transactions on neural networks and learning systems 27*, 7 (2016), 1457–1468.

[12] Chollet, F., e others. Keras. `https://github.com/fchollet/keras`, 2015.

[13] Domingos, P. A few useful things to know about machine learning. *Communications of the ACM 55*, 10 (2012), 78–87.

[14] Doya, K. Reinforcement learning in continuous time and space. *Neural computation 12*, 1 (2000), 219–245.

[15] Hearst, M. A., Dumais, S. T., Osuna, E., Platt, J., e Scholkopf, B. Support vector machines. *IEEE Intelligent Systems and their applications 13*, 4 (1998), 18–28.

[16] Hecht-Nielsen, R., e others. Theory of the backpropagation neural network. *Neural Networks 1*, Supplement-1 (1988), 445–448.

[17] Ioffe, S., e Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning* (2015), pp. 448–456.

[18] Jacobson, V. A new way to look at networking. *Google Tech Talk 30* (2006).

[19] Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., e Braynard, R. L. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies* (2009), ACM, pp. 1–12.

[20] Kaelbling, L. P., Littman, M. L., e Moore, A. W. Reinforcement learning: A survey. *Journal of artificial intelligence research 4* (1996), 237–285.

[21] Kappen, H. J. Optimal control theory and the linear bellman equation.

[22] Melo, F. S. Convergence of q-learning: A simple proof. *Institute Of Systems and Robotics, Tech. Rep* (2001), 1–4.

[23] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., e Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[24] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., e others. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[25] Schmidhuber, J. Deep learning in neural networks: An overview. *Neural networks 61* (2015), 85–117.

[26] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., e Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research 15*, 1 (2014), 1929–1958.

[27] Sutton, R. S., e Barto, A. G. *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[28] Tanner, B., e White, A. RL-Glue : Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research 10* (September 2009), 2133–2136.

[29] Thorndike, E. L. The law of effect. *The American Journal of Psychology 39*, 1/4 (1927), 212–222.

[30] Turing, A. M. Computing machinery and intelligence. *Mind 59*, 236 (1950), 433–460.

[31] Watkins, C. J., e Dayan, P. Q-learning. *Machine learning 8*, 3-4 (1992), 279–292.