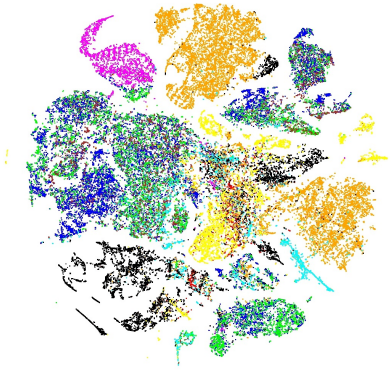
TÉCNICO LISBOA



Deep Learning Methods for Reinforcement Learning

Daniel Luis Simões Marta

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor

Professor Rodrigo Martins de Matos Ventura

Examination Committee

Chairperson: Professor Arlindo Manuel Limede de Oliveira
Supervisor: Professor Rodrigo Martins de Matos Ventura
Members of the Committee: Professor Mário Alexandre Teles de Figueiredo

November 2016

Acknowledgments

What began as an assignment of a deceptively simple concept, compressing information to be used on reinforcement learning, sooner became an exploration quest on search for deep learning methods that could enhance representation and approximation of states, opening doors to a vast scientific horizon of possibilities.

Firstly, a special thank to my adviser, Prof. Rodrigo Ventura, not only for his confidence in my potential and work as a researcher but also for his support and motivation. His insights, suggestions and guidance cleared a troublesome path of mathematical obstacles and new concepts I did not imagine before enrolling into this thesis.

Secondly, I would like to thank Prof. Isabel Ribeiro and Prof. Pedro Lima, for their enthusiasm on the Autonomous Systems course, which greatly improved my curiosity and motivation and led me to further pursue a thesis on related topics. I would also like to thank Prof. Luis Custódio in which his teachings were very paramount to start this thesis, his rigour and standards facilitated the approach. Prof. João Messias's summer course at ISR was very enlightening and concise, enabling significant discussions about Reinforcement Learning.

Although not explicit on this thesis, there was a tremendous amount of hours, I spent learning about statistics, information theory, machine learning, reinforcement learning and finally deep learning.

Finally, an honest and special thank-you to my family, mainly my parents, sisters and my grand mother. They were a quintessential support during the development of this thesis, they always encouraged me and believed in me, even on the darkest times. This work was supported by project FCT/MEC(PIDDAC) INCENTIVO/EEI/LA0009/2014.

Resumo

Esta tese foca-se no desafio de desacoplar a percepção de estados e aproximação de funções quando aplicamos *Deep Learning* a aprendizagem por reforço. Como ponto de partida, foram considerados os estados de dimensão elevada, sendo esta a razão fundamental da notória limitação da incorporação de algoritmos de aprendizagem por reforço a domínios reais. Abordando a temática da *Curse of Dimensionality*, propomo-nos reduzir a dimensionalidade dos estados através de métodos de *Machine Learning*, no sentido de obter representações sucintas e suficientes (representações internas de um sistema), que podem ser usadas como estados análogos para aprendizagem por reforço. No âmbito da mesma metodologia, foram exploradas alternativas para parametrizar a *Q-function* em tarefas com um espaço de estados bastante superior.

Várias abordagens foram usadas ao longo das duas últimas décadas, incluindo *Kernel Machines*, onde a escolha de filtros apropriados para cada problema consumia a maior parte da investigação científica. Nesta tese foram exploradas técnicas de *Machine Learning*, nomeadamente técnicas com treino não supervisionado, com foco na arquitetura de redes neuronais que deram origem ao ramo *Deep Learning*.

Uma das temáticas chave consiste na estimativa de *Q-values* para espaços de estados elevados, quando as abordagens tabulares são insuficientes. Como um meio de aproximar a Q-function foram explorados métodos de *Deep Learning*.

Os principais objetivos incluem a exposição e compreensão dos métodos propostos com vista à implementação de um controlador neuronal. Várias simulações foram efetuadas, tendo em conta diferentes métodos de otimização e funções de custo com o propósito de retirar conclusões.

Diversos procedimentos foram elaborados para aproximar a *Q-value function*. Para inferir melhores abordagens e possibilitar o aparecimento de aplicações reais, foram conduzidos testes entre duas arquiteturas distintas. Implementação de técnicas do estado da arte foram utilizadas e testadas em dois problemas clássicos de controlo.

Palavras-chave: Aprendizagem Máquina, Redes Neuronais, Deep Learning, Processos de Markov, Aprendizagem por reforço

Abstract

This thesis focuses on the challenge of decoupling state perception and function approximation when applying *Deep Learning Methods* within *Reinforcement Learning*. As a starting point, high-dimensional states were considered, being this the fundamental limitation when applying *Reinforcement Learning* to real world tasks. Addressing the *Curse of Dimensionality* issue, we propose to reduce the dimensionality of data in order to obtain succinct codes (internal representations of the environment), to be used as alternative states in a *Reinforcement Learning* framework.

Different approaches were made along the last few decades, including *Kernel Machines* with *hand-crafted features*, where the choice of appropriate filters was task dependent and consumed a considerable amount of research. In this work, various *Deep Learning* methods with unsupervised learning mechanisms were considered.

Another key thematic relates to estimating Q-values for large state-spaces, where tabular approaches are no longer feasible. As a mean to perform Q-function approximation, we search for supervised learning methods within *Deep Learning*.

The objectives of this thesis include a detailed exploration and understanding of the proposed methods with the implementation of a neural controller. Several simulations were performed taking into account a variety of optimization procedures and increased parameters to draw several conclusions.

Several architectures were used as a Q-value function approximation. To infer better approaches and hint for higher scale applications, a trial between two similar types of Q-networks were conducted. Implementations regarding state-of-the-art techniques were tested on classic control problems.

Keywords: Machine Learning, Neural Nets, Deep Learning, Markovian Decision Processes, Reinforcement Learning

Contents

List of Figures

List of Tables

List of Symbols

$[\ ]^T$	Transpose operator
α	Learning rate
$\boldsymbol{\theta}$	Parameters of the Network
$\boldsymbol{\theta}_{ij}$	Specific parameter of a connection j in layer i
ϵ	Probability of selecting action a
γ	Discount Factor
$\mathbb{E}(\cdot)$	Expected value
$\mathbf{a}$	A bold symbol stands for a multi-dimensional variable
$\mathcal{L}$	Loss function
$\mathcal{T}$	Training Criterion
∇	Gradient operator
π	Policy of an agent
π^*	Optimal Policy
a	Action of an agent
s	State of an environment

Glossary

ADAM Adaptive Moment Estimation.

AE Auto-Encoder.

ANN Artificial Neural Networks.

DBN Deep Belief Network.

DL Deep Learning.

DQN Deep Q-Network.

MDP Markov Decision Process.

NFQ Neural Fitted Q-Network.

RBM Restricted Boltzmann Machine.

RL Reinforcement Learning.

RPROP Resilient Backpropagation.

SAE Stacked Auto-Encoder.

SCG Scaled Gradient Descent.

1| Introduction

"I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted."

– Alan Turing

Contents

MACHINE learning grew as a sub-field of Artificial Intelligence with the primary goal of having machines learning from data. Researchers attempted to approach this with several methods including *Neural Networks* which mainly included perceptrons and generalized linear models.

Early *Neural Networks* models were inspired by the reverse engineering process of our biological brain when collecting, separating, and in sum, identifying interesting data structure from our perceptions.

The first breakthrough of neural networks happened when a neural network containing few hidden layers could be trained under the *back-propagation* algorithm [2]. The current wave of neural networks, rebranded as *Deep Learning* (where Deep refers to neural networks with multiple hidden layers) started around 2006 [3] and *Deep Learning* in general still grows in popularity with research and state-of-the-art results (Section. 1.2). A plethora of Neural Networks configurations emerged in the last years of research. *Stacked Auto-Encoders* are able to perform unsupervised learning and find key features in search for a joint probability. Deep Feed-Forward Networks are able to perform efficient supervised learning on large datasets, an essential element for tasks such as regression or classification.

The key objective of this work focuses on identifying the impact of that *Deep Learning* in *Reinforcement Learning*, in particular, to take advantage of the ability *Deep Learning* has when inferring complex distributions of data. This has a special role when dealing with high-dimensional inputs or wide state spaces, where a *Q-function* requires approximation.

1.1 Motivation

"The real challenge proved to be solving tasks that for us, humans, feel intuitive but in reality are hard to describe, formally these tasks include recognizing faces or spoken words, both very intrinsic to the human nature" - [1].

This thesis focused its attention on the usage of Deep Learning methods in Reinforcement Learning, as the researcher Dr.David Silver from the Royal Society University, United Kingdom, in a recent conference [1] of May 7, 2015 (ICLR) stated "one would need Deep Reinforcement Learning to solve the real AI challenges". Having an agent performing in a real environment and collecting the highest expected sum of rewards is paramount and a long goal in robotics. Applying reinforcement learning to a real world task has always been challenging due to the inherent variety and interdependence of variables necessary to circumscribe a problem. The main reason of the above challenge relates with the fact of having large state-spaces thus turning the usage of tables impracticable. Therefore, reinforcement learning did not remain as the main topic of AI research. It was believed that TD:BlackGamon was a very special case of success, since researchers could not apply the same simple neural network to another game. Neuron-dynamic programming [4] emerged as a set of methods to apply Neural Networks within Reinforcement Learning . Previous research led to the conclusion that Reinforcement Learning does not scale well to high-dimensional input data. The problem is known as the *curse of dimensionality* [5]. Traditionally, this problem was handled by using hand-crafted low-dimensional feature vectors, in order to, inferring a person's face (which accounts for a lot of detail such as the nose, eyes and other traces) [6], matching the desired details, or in the case of speech, one could distinguish vowels with a spectrogram of an audio signal by detecting the first harmonic (known as formant).

Later, we could combine hand-designed features and mapping from these, generating a richer, stronger output. Unfortunately, hand-crafted features are never ideal, even if we have means to combine these features with powerful algorithms, the possibility of approaching unseen and unpredicted samples is a real issue. Even under the hypothesis of designing appropriate features for a certain task the relevance onto different tasks would be sparse, inhibiting the possibility to *transfer learning*. Following the trend of mapping from features, early efforts were made to surpass PCA from a linear method to a non-linear one (using Restricted Boltzmann Machines) with success on a MNIST dataset [7]. The ability to gather knowledge from a large set of data and generating features "on the fly", avoiding the need for human operators to formally specify important details to look for, turned to be promising.

Therefore, we purpose the usage of Deep Learning as a complementary machine learning method to Reinforcement Learning. This is specifically important when considering a variety of problems and environments that would consume too many resources (in a tabular form) or just infeasible in practice. A recent and successful case of success on *Deep Reinforcement Learning* was published by Google Deep Mind [8] on *Nature*, where the same Deep architecture was used across a variety of Atari games. The effectiveness of *Deep Learning* relies on the efficiency of neural networks, specially when forming Deep architectures [9]. In the same work, the physicist Henry W. Lin, also attributed the success of *Deep Learning* to structural properties of our environment and

[1] `https://www.youtube.com/watch?v=EX1CIVVkWdE`

corresponding datasets used, specifically in terms of symmetry and locality.

1.2 State of the art

Deep Learning arose to popularity as a field at the ImageNet Large-Scale Visual Recognition Competition (ILSVRC) in 2012, when a convol-net had victory, with a state-of-the-art top-5 error of 15.3% from 25%, against other methods, as sparse-coding [10], with the latest competitions achieving errors close to 3%. Traffic signs classification [11], enhanced localization and detection on images benefited from the use of convolutional networks [12]. Probabilistic reasoning through classification had also been applied with success in automated medical diagnosis.

Besides image recognition, speech recognition has seen overwhelming improvements with a drop of error rates after a decade of stagnation [13]. Since then, the complexity, scale and accuracy of deep networks have increased and the complexity of tasks, that can be solved, has grown. For instance, neural networks can learn outputting chains of letters and numbers taken from images, instead of only proposing one object [14].

Another interesting neural network designed to model sequences, LSTM (Long Short-Term Memory) has arisen in popularity with the application of machine translation [15].

Creative applications of neural networks shown a neural network could potentially learn entire programs, leading to revolutionary (but still in a embryonal stage) self-programming technology[16].

Modern reinforcement learning endeavours mainly relate to either compacting high dimensional inputs (to use as succinct states) or approximating the value function with novelty machine learning algorithms [17]. Regarding value function estimation, decision tree functions were used with success on problems were table lookup would be infeasible [18].

Another interesting approach considered NEAT, an evolutionary optimization technique to automatically discover effective representations for TD function approximators. NEAT alongside several Q algorithms were applied to several benchmarks with relative success, when compared to previous methods [19].

A recent alternative to compactly represent the Q-function with fewer memory requirements consists on using low-rank and sparse matrix models. A matrix is decomposed into low-rank and sparse components by using the Robust Principal Component Analysis, hence a convex optimization problem [20].

In reinforcement learning, recent research included Deep Learning architectures to obtain succinct state representations in a data efficient manner with Deep Auto-Encoders [21]. Recently Google published in Nature a reinforcement learning framework, Deep Q-Value Network (DQN), where the same model was used to learn different ATARI games, achieving human performance in half of the tested games [8].

Research by Google DeepMind, using a Deep Neural Network and Monte Carlo tree-search, allowed the creation of an algorithm able to beat a master at the game GO [22]. Furthermore, asynchronous methods for Deep Reinforcement Learning have been used for neural controllers, namely in tasks of navigation in 3D environments with even greater success than previous methods [23]. Recent research to improve the results of DQN considered a new mechanism, Human

Checkpoint Replay. A training dataset is initialized by samples generated from a human player to sort out games that include sparse rewards as the Montezumas's Revenge [24]. An alternative architecture for partially observable MDPs emerged from DQN, altering the last fully connected layer to a LSTM (Long Short-Term Memory) to incorporate memory of t time stamps instead of just one image [25] .

In robotics, Deep Learning was successfully applied to solve classical control tasks, such as an inverted pole, directly from raw preprocessed visual inputs [26]. An example of the previous was the end-to-end training of deep visuomotor policies using a convolutional network and several fully connected layers, demonstrating higher performances when compared to previous methods [27]. More recently, Deep Learning was also applied to *self-driving car* technology [28].

1.3 Problem statement

The application of reinforcement learning rises several challenges. Information about the environment, when considering real world or complex tasks, is often overwhelming. Implementations considering raw unprocessed data from perceptions are often infeasible, due to the massive data availability, corresponding memory usage and processing power. A second concern resides upon entering the realm of large state-spaces. Identifying and classifying similar states and their properties, thus generalizing the environment with a sufficient model, is quintessential.

Therefore, this thesis approaches two reinforcement learning challenges: encoding raw input into meaningful states; and generalizing the environment from a proper model.

To tackle the previous, deep learning methods were explored as an instrument alongside with state-of-the-art reinforcement learning considerations.

The first problem was approached by considering unsupervised learning such as auto-encoders Sec. 2.5.1 and stacked auto-encoders Sec. 2.5.5 to perform dimensionality reduction on high-dimensional images from a toy problem. Several comprehensive experiments were performed Chap. 5, adjusting the number of neurons and the optimization procedure. The aim of the former mentioned tests was to verify the impact of choosing different architectures for the specific task.

The second problem was firstly tackled with two similar neural networks, depicting two Q-function approximations. An offline method (NFQ 3.8) to train both networks under the same conditions was used. A battery of tests was conducted to achieve a conclusive result on better practices. Furthermore, the best Q-function architecture was used on a much larger state-space. To accomplish this, a more efficient algorithm, DQN Sec. 3.9 was employed alongside better procedures. Results are finally drawn within two classic control tasks.

1.4 Main contributions

The main contributions of this thesis are mostly related to the exploration of newer methodologies for reinforcement learning tasks , which include efforts towards reducing the dimensionality of states under the scope of Deep Learning, and designing various architectures for high dimensional states or large state spaces.

For aerospace applications, Deep learning applications are still in a embrionary stage. In a recent conference promoted by the Langley Research center [2] several current and future applications were discussed. Anomaly detection in non-destructive evaluation of materials images is performed by detecting anomalous pixels, using convolutional neural networks to classify image data. Classifications models to predict the cognitive state of the pilots may be created with deep neural networks, using physiological data collected during flight simulations.

For radar applications, recent deep learning implementations include statistical characterization of sea clutter by modeling the parameters of a K distribution [29].

Since deep learning became a powerful tool of machine learning, the range of applications is only limited by the amount of data/computational resources and mainly by optimization procedures (which are the hardest to overcome).

A final contribution of this thesis relates to providing open-source code, to enable the replication of the final experiments for any reader `https://github.com/DanielLSM/ThesisIST`.

1.5 Thesis Outline

Following this introductory chapter, the thesis branches between two main concepts, *Deep Learning* and *Reinforcement Learning*. Both have an extensive dedicated chapter, mainly designed to present the reasoning needed to build the tested architectures later at the end of this work.

Chapter 1 An introduction gives an overview of this thesis, including the outline of the approach, an insight to previous developed research and finally the objectives the author proposes to reach as well as the motivation to do so.

Chapter 2 Discusses old and new concepts about Neural Networks. The main focus is presenting possible designs, and their algorithms, to serve as function approximations of states and values for *Reinforcement Learning*. Both supervised and unsupervised learning are approached and applied in later chapters.

Chapter 3 A generic classic approach of Reinforcement Learning is presented, aiming to describe On-line and Off-line learning. Furthermore, discussion and conjunction of hypotheses are coupled with recent research on *Deep Learning*.

Chapter 4 Several architectures are designed and defined under certain assumptions made in previous chapters.

Chapter 5 Experimental results within the proposed architectures are drawn.

Chapter 6 Conclusions gathered from the current endeavor and considerations towards future work.

[2] "Big Data Analytics and Machine Learning in Aerospace" May 17, 2016 `https://www.jlab.org/conferences/trends2016/talks/ambur.pptx`

2| Deep Learning Concepts

Contents

Deep learning was re-branded from ANN (Artificial Neural Networks) in recent years. Although this thesis revolves around the use of *Deep Learning*, several other methods are extremely popular in machine learning and statistics, such as Linear Regression, being simple and of efficient computation, performing least-squares fitting estimation over targets of interest to approximate a linear function to a dataset.

Supervised training of a feed-forward neural networks was successfully applied, with just one hidden layer [2]. Furthermore, an ANN is able to represent arbitrary complex but smooth functions

[30], given enough neurons. By adding additional layers, more complex non-linear representations are acquired, enriching our representation of an arbitrary large dataset [31].

Arguably one of the most important reasons for these methods to arise was the increment of computational run-time by performing parallel computations with GPUs on the training phase. Affording to update millions of parameters (sometimes billions with GoogleLeNet) in every iteration when performing non-linear non-convex optimization was paramount to the success of Neural Nets.

In the following sections, a plethora of fundamental Deep Learning concepts used to understand later experiments are unfold.

2.1 Neuron Model

Early biologically inspired efforts to mimic brain on an algorithmic/mathematical manner were settled under certain assumptions and restrictions [32]. The core ideas were related to acquired contemporaneous knowledge about biological activity of nervous connections, biological neuron models and corresponding experiences conducted. Assumptions focused mainly on the importance of neuronal activity, as neurons should have an "all-or-none" signal pattern since our entire brain is not active at every given moment. A neuron should have to be excited above a certain voltage to activate, e.g, the sum of input connections should lead to a signal at the output, hence creating a cause relation effect. Delays in our artificial neural model should be only considered with respect to synaptic delay (although modern delay is due to intensive matrix computation). A final remark is related to the static nature of the network, since our brains do not overwhelmingly change during each computation.

While developing an artificial neuron model, dendrites could be considered as input connections leading to a nucleus.

[1] https://commons.wikimedia.org/wiki/File:Blausen_0657_MultipolarNeuron.png

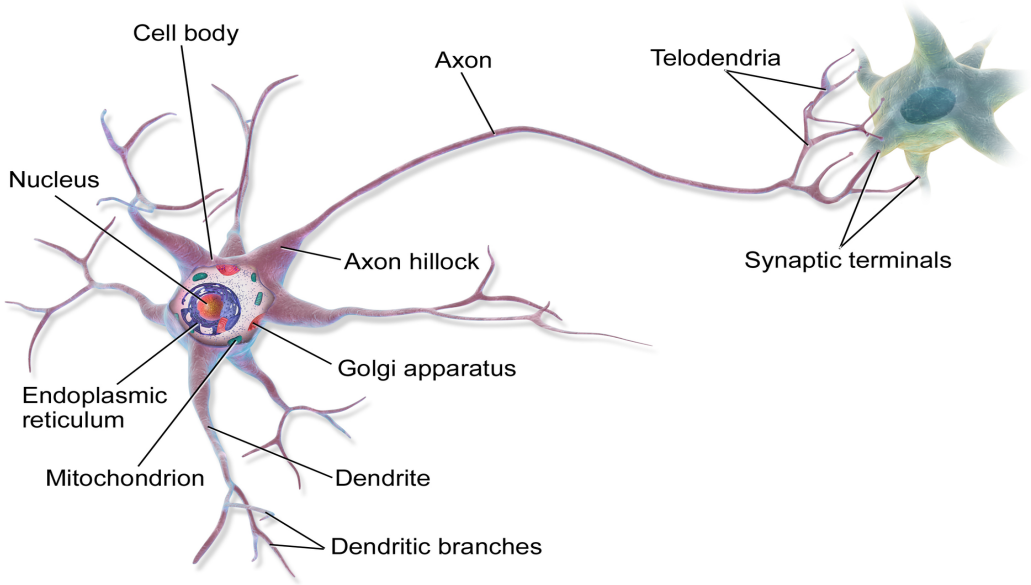


Figure 2.1: Neuron Model.[1]

On the nucleus a sum of electrical impulses could be added and passed into other neurons using an axon or axon terminals. These axons operate as an output signal. Any input transformation is allowed (except zero) but since neurons were believed to operate in a on-off manner, a step function was initially chosen.

Considering an artificial neuron, every connection is represented by w_i and every input x_i. A neuronal input is translated into:

$$a = w_1x_1 + w_2x_1 + w_3x_3 + w_ix_i + ... + b = \sum_{i=0}^{i=N}(w_ix_i) + b \tag{2.1}$$

The above expression strictly comprises a linear combination of variables, e.g., a regression model. Or in matrix form:

$$a = \mathbf{W}\mathbf{x} + b \tag{2.2}$$

In fact, there are methods to train a Neural Network using recursive least-squares algorithm [33]. Nevertheless, every output of a neuron is a non-linear affine transformation of the n^{th} input. A neuron output o is computed by:

$$o = \Phi(a) = \Phi(\mathbf{W}\mathbf{x} + b) \tag{2.3}$$

where $\mathbf{W}$ is the weight matrix, assigning weights to input connections, $\mathbf{x}$ the connections themselves and b a bias constant. On the following section, different non-linearities are described to compute the affine transformation.

2.1.1 Common Non-Linear output Functions

The early artificial neuron model output Eq. (2.3) initially considered Φ as step function. In common modern applications, Φ represents a non-linear function. Fitting non-linearities to solve a supervised learning problem enables complex data structures to be embodied into our model. If Φ is a linear function, the problem is transfigured into a *Linear Regression*, similar to the least squares method but not effective for highly non-linear data.

In a deep structure choosing a specific type of function might depend on some design choice for a specific task. Object recognition often deals with classification tasks, where evaluating the cross-entropy of the output is suitable and a *softmax* is required. It was empirically demonstrated, that the choice of hidden layers activations of deep architectures is not essential for a satisfying performance, e.g., achieving a reasonable optimization loss. In sum, we present a summarized table 2.1 of commonly non-linear taken from [1].

2.2 Neural Networks

Connecting and increasing neurons in a neural network can be seen as increasing the order of a polynomial regression or adding kernels to a kernel method.

An architecture may be shallow (Auto-Encoders), deep (Deep Belief Networks) or recurrent (Recurrent Neural Networks) but four central properties prevail across designs. Since most

Non-Linear Functions

Non-Linearity	Description
Hyperbolic tangent (tahn)	$\Phi(z) = tangenthyperbolic(z) = \frac{2}{1+e^{-2z}} - 1$.
Softmax	$\Phi(z) = softmax(z) = \frac{e^{z_i}}{\sum_j e^{z_j}}$. The softmax output can be considered as a probability distribution over a finite set of outcomes. It is mainly used as output non-linearity for predicting discrete probabilities over output categories.
Sigmoid	$\Phi(z) = sigmoid(z) = \frac{1}{(1+e^{-z})}$.
Rectifier (ReLU)	Transform of hidden layer from input to next: $\Phi(z) = max(0,z)$, or $\Phi(z) = (z)^+$.

Table 2.1: Table of popular non-linear functions used in DL layers according to [1].

conventional training procedures are gradient-based optimizations with parameterized families of functions (often with a probabilistic interpretation, in the context of conditional maximum likelihood criteria), *Deep Learning* tends to be quite flexible. Since parameters are updated recursively, a wide variety of neural structures emerge, due to this inherent modular aspect.

In this chapter, we will explore the following fundamental concepts to build a neural network:

- **Transfer functions,** non-linear functions, acting as outputs of each layer Sec. 2.1.1.
- **Loss function,** our learning objective, a function of each output and corresponding targets Sec. 2.4.
- **A training criterion and a regularizer,** the loss function coupled with a regularizer to avoid over-fitting or enhancing generalization/performance on a particular dataset Sec. 2.4.3.
- **An optimization procedure,** the method we choose to propagate the gradient of the training criterion through the parameters of our network Sec. 2.4.4.

2.3 Concept of Layer

A Neural network architecture may be designed for a specific task, but a set of internal layers is constant throughout designs. We follow the approach of the *Neural Network Design*[34]. A layer $\mathcal{L}_i$ represents a specific set of neurons sharing a common property. Denoting $\mathcal{N}_{i,t}$ as all t neurons residing in a layer i, thus:

$$\forall_i \quad \forall_t \quad \mathcal{N}_{i,t} \in \mathcal{L}_i \tag{2.4}$$

if each neuron connected to the input is represented as $\mathcal{N}_{input,t}$ then our set of neurons may be simply represented as $\mathcal{L}_{input}$. Most layers between an input and an output are referenced as *hidden layers* for representing an inner block of computation, thus not directly interacting with any outside variable. Considering a fully connected neural network, each neuron in our input layer connects to

each neuron in the following (*hidden*) layer, where each output is a function of the previous unit. If $\mathbf{x}_{input}$ is a data vector fed to the input layer and $o_{\mathcal{N}_{hidden,t}}$ an output t of our first *hidden layer* then:

$$\forall_t \quad o_{\mathcal{N}_{hidden,t}} = \Phi(\mathbf{W}_{input,t}\mathbf{x}_{input} + b_t) \tag{2.5}$$

where $\mathcal{L}_{input}$ represents the input vector $\mathbf{x}_{input}$. However, layers contain many neurons and different dimensions, we may generalize from one neuron to the full layer by considering all connections to all neurons and compacting in matrix form:

$$\mathbf{o}_{\mathcal{L}_{hidden}} = \Phi(\mathbf{W}_{input}\mathbf{o}_{\mathcal{L}_{input}} + \mathbf{b}) \tag{2.6}$$

where $\mathbf{o}_{\mathcal{L}_{hidden}}$ is a vector with each entry representing the output of a particular neuron. Propagating data forward in a *neural network* is usually referenced as a *forward-pass* and the complete algorithm is described later in Alg. 1. A *forward-pass* is a sequential process, mainly consisted in several matrix multiplications and sums. A batch of input vectors can be propagated in parallel through our network leading to increased efficiency and speed, being one of the strong points on why *Deep Learning* emerged in popularity alongside to the creation of powerful *gpus*.

Computing the second hidden layer, equation (2.6) is unrolled into:

$$\mathbf{o}_{\mathcal{L}_{hidden2}} = \Phi(\mathbf{W}_{hidden2}\Phi(\mathbf{W}_{hidden}\mathbf{x} + \mathbf{b}) + \mathbf{b}) \tag{2.7}$$

with networks following a similar dynamic being referred as *feed-forward networks*. Successive layers with all neurons from one layer connected to another are referenced in the literature as *fully-connected* layers, an image is presented Fig. 2.2. Although *Fully-connected* networks were the first generation of neural networks (multilayer perceptrons), further research explored sparsity within connections, leading to popular *convolutional networks* [35]. These types of networks are known to be translational invariant relative to the input, since specific features are convolved in every image, instead of using every neuron in a layer as a general feature detector. In the exposition of this work, whenever we refer to neural networks, fully connection is assumed.

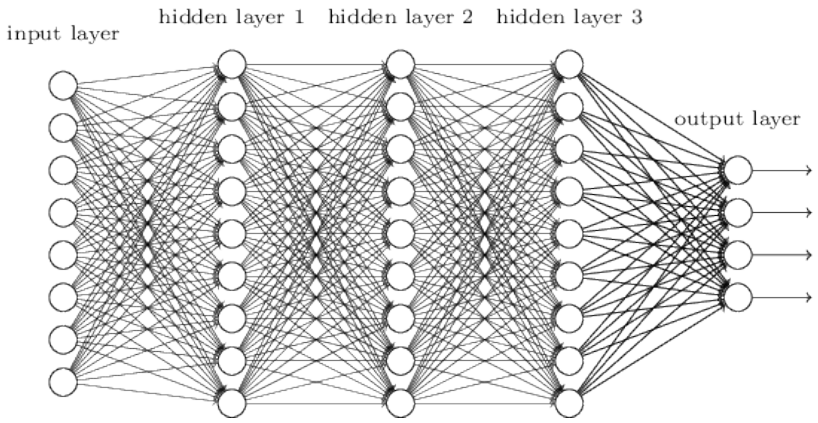


Figure 2.2: Fully-Connected Feed-Forward Network.[2]

[2] `http://neuralnetworksanddeeplearning.com/chap5.html`

2.4 Loss Function

The concept of a *loss function* is widely used across machine learning, statistics and information theory. In NN/DL literature our loss function is referenced as objective (purpose of the task) function, performance measurement or energy function when relating to energy-based models. It is often a function of the output layer, expected outputs defined by labels or targets and other variables related to training of the neural network parameters.

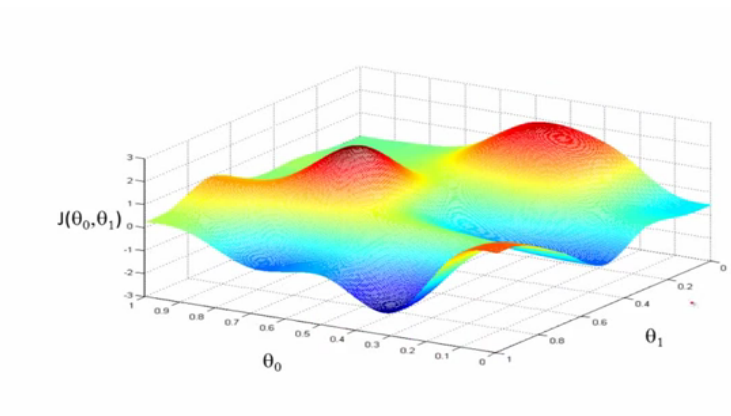


Figure 2.3: Non-Convex Loss function.[3]

Furthermore, a loss function is defined as:

$$\mathcal{L} = \Psi(o_{\mathcal{N}_{output}}, o_{training\ set}) \tag{2.8}$$

where Ψ is choosen according to our regression/classification task, $o_{\mathcal{N}_{output}}$ is our output vector and $o_{training\ set}$ a corresponding target/label. Every $o_{\mathcal{N}_{output}}$ is obtained by propagating inputs throughout our network e.g., a *forward pass*. Our targets $o_{training\ set}$ may be classes (*one-hot vectors*, a vector of zeros and a one) or real valued depending on the objective. An example of a non-convex loss function is exemplified in Fig. 2.3. Our primary goal is to adjust both parameters θ_1 and θ_2 to lower our loss $\mathcal{L}(\theta_1,\theta_2)$, and descend towards one of both minimums illustrated.

The choice of a loss function is paramount to achieve a good result in a certain task. Different loss functions will have very different local minima (and geometry), with a poor initialization of the weights, gradient descent could easily get trapped in a poor minimum. Hence, one must define an appropriate loss function. Neural Networks are applied to several domains, in this thesis we will focus on two: Classification and Regression.

2.4.1 Regression

Considering the squared error loss function, usually applied in linear regression methods such as minimum squared method, a Vanilla (shallow) MLP may be created for this example 2.4: the input corresponds to the data we feed to the model, the activations are a linear combination of weights designed to each connection and an output layer.

[3] `https://dmm613.files.wordpress.com/2014/12/nonconvex.png`

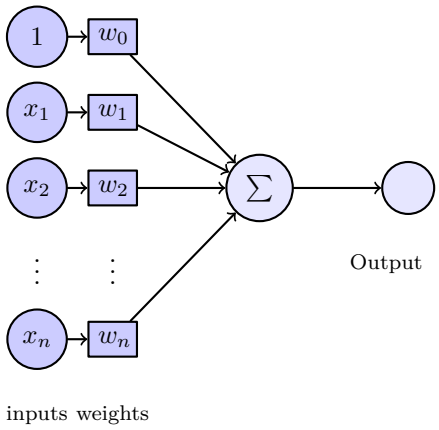


Figure 2.4: Simple MLP for Linear Regression

For example, if each dot corresponds to pairs of (x,y) and our goal is to fit a line, minimizing the squared distance error between each dot and the line (minimizing the length of each mini "spring", recalling Hooke's law from physics), each activation of our hidden unit would be purely linear.

The generalization of the linear regression to regression via any function f can be done by defining the mean squared error of f in which the loss function is [1]:

$$\mathbb{E}[\|\mathbf{y} - f(\mathbf{x};\boldsymbol{\theta})\|^2] \tag{2.9}$$

In order to estimate the expected value of the error (quadratic) we commit by approximating $\hat{\mathbf{y}} = f(\mathbf{x};\boldsymbol{\theta})$ to $\mathbf{y}$, we use a batch of N samples from our training set:

$$\mathcal{L}_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 \tag{2.10}$$

Using a neural network prediction to evaluate the loss function, $\hat{y}_i$ becomes an output of a neuron at the output layer. Furthermore, we may generalize the error for a multidimensional output layer. If $o_{\mathcal{N}_{output,j,i}}$ is a neuron j on the output layer, D is the dimension of the output layer, and i the corresponding sample, we may re-write (2.10) as:

$$\mathcal{L}_{MSE} = \frac{1}{D}\frac{1}{N}\sum_{j=1}^{D}\sum_{i=1}^{N}(o_{\mathcal{N}_{output,i,j}} - y_{i,j})^2 \tag{2.11}$$

2.4.2 Classification

In machine learning, classification translates on maximizing a probability of a certain label (class) within a set. In a neural network standpoint often equates by using *hot vectors* at the output layer. Hence, each code at the output will represent a different class. Our aim is to have the desired output for each label, e.g, obtaining a value close to one for the corresponding class.

One widely used loss function from information theory is recalled here, the logarithm of the likelihood function for a Bernoulli distribution:

$$\mathcal{L}_{log} = -\frac{1}{N}\sum_{i=1}^{N}(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)) \tag{2.12}$$

where, N is the number of samples, y_i represents the desired output and $\hat{y}_i$ a class predicted by a model. Using cross-entropy for multiple classes we attain,

$$\mathcal{L}_{log} = -\frac{1}{N}\frac{1}{M}\sum_{i=1}^{N}\sum_{j=1}^{M}(y_{i,j} log(\hat{y}_{i,j}) + (1 - y_{i,j}) log(1 - \hat{y}_{i,j})) \tag{2.13}$$

where M is the number of labels involved on the classification process.

Using a neural network prediction to evaluate the loss function, $\hat{y}_{i,j}$ becomes an output of a neuron at the output layer, hence Eq. (2.13) can be re-written as:

$$\mathcal{L}_{log} = -\frac{1}{N}\frac{1}{M}\sum_{i=1}^{N}\sum_{j=1}^{M}(y_{i,j} log(o_{\mathcal{N}_{output,j}}) + (1 - y_{i,j}) log(1 - o_{\mathcal{N}_{output,j}})) \tag{2.14}$$

where the output layer has equal dimension to the class vector we aim to minimize.

2.4.3 Training criterion and a Regularizer

The output of a *neural network* could be interpreted as a conditional probability of our outputs given data and weights of network connections (more in Section. 2.5.1). In practice and in most problems, inferring the true conditional probability is intractable (since we are sampling over a finite training set), instead of minimizing directly the loss function, we will minimize the expected value of the loss function (using our training set). This expected value is mentioned in the literature as training criterion $\mathcal{T}$. One recurrent problem most neural networks present relates to over-fitting a training set in order to reduce the training criterion, but it is often inefficient, since mapping the training set entirely is computationally expensive and without generalization one will find high performance errors on new upcoming samples.

Over-fitting is never desirable in any application (unless the objective is mapping directly inputs to outputs), for instance, in order to accentuate the problematic the reader may assume the purpose of fitting a polynomial function over a data set of points and further assume the usage of the mean square error loss function. Furthermore, the mean square error can be lowered close to zero, corresponding to a high order polynomial fitting exactly every single point of a training set. If a training set is small, the error of a loss function could be as close to zero as possible, however generalization amongst data is lost. Thus a term is added along the training criterion to avoid over fitting, this is usually referred as regularizer.

Minimizing the new loss Function improves the error amongst the training and test data, thus regularizing our model.

Weight Decay

It is basically weight decay or clearly L2 weight decay because it pushes the weights towards zero, hence penalizing the complexity of the fitting curve, imposing a smoothness restriction into

the model. In a Bayesian perspective, it is equivalent to impose a prior distribution to our model.

The regularization is propagated through the weights of the network during the backward step of the optimization procedure, in the case of L2-decay we obtain [1]:

$$\mathcal{T} = \mathcal{L} + \lambda f(|\boldsymbol{\theta}|^2) \tag{2.15}$$

Another interesting method to avoid over-fitting our network model is the *dropout* [36] method, where some neurons with probability $1-p$ are not updated during the backward pass. Nevertheless, there are several regularization terms such as the sparsity coefficient (randomly setting some weights to zero).

2.4.4 Optimization Procedure

Thus far, a collection of families of functions to apply as activations were presented, several types of layers were explored, and several loss functions designated to achieve specific objectives were mentioned. Kernel machines are non-linear but convex problems, so they are less computational expensive than Deep Networks which are non-linear, non-convex problems. Hence, we enter the realm of iterative optimization procedures monitored through a performance measure. In *Deep Learning*, optimization procedures comprise several methods to propagate training criterion gradients to parameters of our network, in order to improve performance and accelerate convergence.

By using a validation set one is allowed to generalize, this is also called *early stopping*[37].

Chain Rule and Back-Propagation

Back-Propagation is a general method for computing and propagating gradients in a neural network with an arbitrary number of layers and hidden units. The basic principle is trivially derived from Leibniz calculus, where chain rule is applied to a composite function iteratively. After defining the objective function (the learning objective) a gradient is applied to understand the rate of change to apply to each output weight, in order to descend closer to a minimum by propagating the gradient across the chain, the learning *per se*, is done by decomposing recursively the derivatives of the cost J with respect to the parameters. Amongst each layer "back-propagation" is computed from each derivative with respect to the parameters of the closest layer. The previous remains as one of the main reasons to use simple non-linearities, since the gradient becomes easier to compute. Next, a chain rule overview is performed and generalized for neural networks.

If multiple concatenated functions are considered (or interchangeably layers) and for instance a z that depends on y then, to represent the impact that a small change on x has on z, one needs to compute the small change on y first.

Furthermore, z may be our objective function $z = \mathcal{L}(g(\theta))$ and the gradient is computed with respect to specific parameters of a certain layer $x = \theta$. The reader may notice the intermediate quantities $y = g(\theta)$ where g is a non-linear activation from the type reviewed at section 2.1.1.
Fig. 2.5 follows a simple interpretation: A minor change in θ will propagate linearly on a tangent line (or hyperplane) by $\frac{\partial g(\theta)}{\partial \theta}$ on $g(\theta)$ and that small change will propagate further into our final training criterion $\mathcal{L}(g(\theta))$ by $\nabla_g(\theta)\mathcal{L}(g(\theta))$. Basically, chain rule establishes a multi-dimensional

method to compute the effects of θ on $\mathcal{L}(g(\theta))$ by recursively computing partial derivatives across all the composite functions that represent the training criterion.

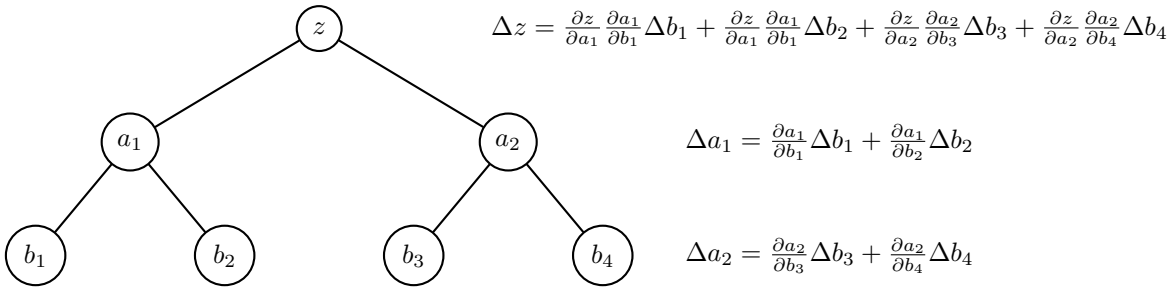


Figure 2.5: Chain rule applied to a neural network.

Back-Propagation for Deep Networks

Whereas Figure. 2.5 illustrated a shallow Network with only one hidden layer, in Deep Networks $g(\theta)$ is a composite output function of an arbitrary number of hidden layers and non-linear activations. In theory the designer (or the reader) is allowed to build a Deep architecture and apply Back-Propagation across every layer recursively (one of the reasons simple to differentiate non-linear activations were chosen), without greatly changing the hyper-parameters of the training procedure. This aspect relies as one of the strongest points of Back-Propagation methods, it is simple and can be paralleled using several GPUs.

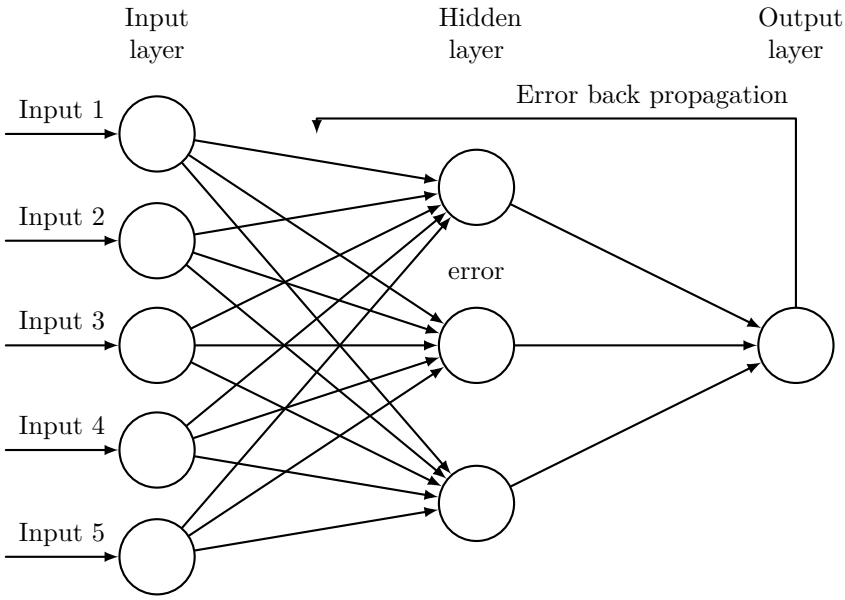


Figure 2.6: Back-Propagation on a neural network.

Back-Propagation is defined by two main algorithms [1]: a forward-pass and a backward-pass. Every iteration comprises both procedures. A forward-pass is computed, propagating the input throughout every layer and activation, in order to update the training criterion. Thereafter, a

backward-pass is performed using the gradient of $\mathcal{L}(g(\theta))$ recursively layer-wise, until all parameters are changed. The method is applied until convergence is reached or a defined maximum iteration counter is met.

Although vanilla Back-Propagation is flexible and easy-to-use, does not guarantee convergence around a minimum (due to oscillations) nor if this local minimum is good or close enough to a global minimum. The former is discussed above and the later will be exposed in Sec. 2.4.4.

Algorithm 1 Forward-pass algorithm for Neural Networks ([1], p.212). There are N layers, each mapping an input $\mathbf{l}_k$via two transformations, matrix $\mathbf{W}^{(k)}$ which suffers a transform via a non-linearity into an output $\mathbf{l}_{k+1}$. The input of the first layer corresponds to $\mathbf{l}_0$ and the predicted output $\hat{\mathbf{y}}$ represented by $\mathbf{l}_N$. The training criterion is computed by adding a regularizer to a Loss function

procedure FORWARD-PASS
  $\mathbf{l}_0 = \mathbf{x}$
  for $k = 1...,N$ **do**
    $\mathbf{z}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{l}^{(k-1)}$
    $\mathbf{l}^{(k)} = f(\mathbf{z}^{(k)})$
  end for
  $\hat{\mathbf{y}} = \mathbf{l}_N$
  $\mathcal{L} = L(\hat{\mathbf{y}},\mathbf{y}) + \lambda\Omega$

Algorithm 2 Backward-pass algorithm for a Neural Networks ([1], pag.213). Using the same nomenclature as 1, the gradient will be propagated from the output to the input in order to change the weights/biases according to a gradient-based optimization method.

procedure BACKWARD-PASS
  $\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}}\mathcal{L} = \nabla_{\hat{\mathbf{y}}}L(\hat{\mathbf{y}},\mathbf{y}) + \lambda\nabla_{\hat{\mathbf{y}}}\Omega$
  for $k = 1...,N$ down to 1 **do**
    $\mathbf{v} \leftarrow \nabla_{\mathbf{z}}\mathcal{L} = \mathbf{v} \odot f'(\mathbf{z}^{(k)})$
    $\nabla_{\mathbf{b}^{(k)}}\mathcal{L} = \mathbf{v} + \lambda\nabla_{\mathbf{b}^{(k)}}\Omega$
    $\nabla_{\mathbf{W}^{(k)}}\mathcal{L} = \mathbf{v}\mathbf{l}^{(k-1)T} + \lambda\nabla_{\mathbf{W}^{(k)}}\Omega$
    $\mathbf{v} \leftarrow \nabla_{\mathbf{l}^{(k-1)}}\mathcal{L} = \mathbf{W}^{(k)T}\mathbf{v}$
  end for

Classic Gradient Descent

The simplest optimization algorithm consists on following the steepest descent direction. Mathematically the gradient (maximum rate of change) is perpendicular to the tangent vectors of the surface $\mathcal{L}$. Using the notation in Alg. 2, $\nabla_{\mathbf{W}^{(k)}}\mathcal{L}$ is used to compute $\mathbf{W}^{(k)}$ in the following manner:

$$\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \epsilon\nabla_{\mathbf{W}^{(k)}}\mathcal{L} \tag{2.16}$$

Or for each entry:

$$w_{ij}^{(k+1)} = w_{ij}^{(k)} - \epsilon\frac{\partial\mathcal{L}}{\partial w_{ij}}^{(k)} \tag{2.17}$$

whereas ϵ is the learning rate; on the other side, a high rate of learning will lead to oscillation. A much nicer way to tackle this issue is by including a clear momentum-term:

$$\Delta w_{ij}^{k+1} = \epsilon \frac{\partial \mathcal{L}}{\partial w_{ij}}^{k} + \mu \Delta w_{ij}^{k} \tag{2.18}$$

A better algorithm to handle this issue is presented in Section. 2.4.4.

Scaled Conjugate Gradient Descent (SCG)

An alternative method to gradient descent considers not only the linear mapping around the point of interest $\mathcal{L}(\theta)$ but also its curvature. In gradient descent, the weight change is equivalent to consider a first-order taylor expansion in a given point:

$$\mathcal{L}(\theta + \Delta\theta) \approx \mathcal{L}(\theta) + \mathcal{L}' T \Delta\theta \tag{2.19}$$

However, SCG [38] uses information on the second order taylor expansion to be used, e.g.:

$$\mathcal{L}(\theta + \Delta\theta) \approx \mathcal{L}(\theta) + \mathcal{L}' T \Delta\theta + \frac{1}{2}\Delta\theta^{\mathsf{T}} \mathcal{L}'' \Delta\theta \tag{2.20}$$

The objective for each iteration consists on determining a critical point $\Delta\theta_*$ of the second order approximation (minimum of the parabolic approximation) e.g:

$$\mathcal{L}_{parabolic}(\Delta\theta) = \mathcal{L}'(\theta) + \mathcal{L}''(\theta)\Delta\theta = 0 \tag{2.21}$$

if there is a conjugate base available, the solution can be simplified considerably. If there is a $\mathbf{P} = \mathbf{p_1}, \mathbf{p_2}, \ldots \mathbf{p_N}$ then we can find the minimum point through a linear combination of this R^N basis.

$$\Delta\theta_* - \Delta\theta = \sum_{i=0}^{i=N} (\alpha_i \mathbf{p_i}) \tag{2.22}$$

Theorem 1 and 2 [38] demonstrate how the conjugate base is obtained recursively (with information of both the Jacobian and Hessian of the loss function). Furthermore, computing the Hessian of $\mathcal{L}(\theta)$ directly is infeasible due to its calculation complexity and memory usage involved. Instead, a non-symmetric approximation is used:

$$s_k = \mathcal{L}''(\theta)\mathbf{p_k} \approx \frac{\mathcal{L}'(\theta + \sigma_k \mathbf{p_k}) - \mathcal{L}'(\theta)}{\sigma_k} \tag{2.23}$$

The full algorithm [38] can be detailed, where empirical evidence shows this method to be far superior than gradient descent.

Resilient Back-Propagation (RPROP)

The RPROP learning algorithm (which stands for "Resilient Back-Propagation") [39] is a local adaptive strategy focused around the sign of the partial derivative around a certain weight w_{ij}.

It overcomes the problems discussed on Sec. 2.4.4 about classic back-propagation by accelerating the convergence process in shallow areas of the objective function and correcting the oscillation problems around a local minima.

A update-value Δ_{ij} is introduced solely for the purpose of determining the weight-update w_{ij}. The key idea is quite simple and the adaptation-rule works as follows: if the sign of the partial derivative $\frac{\partial E}{\partial w_{ij}}$ remains the same, then we assume to be in a shallow region and accelerate convergence, thus increasing the anterior Δ_{ij} by a scalar η^+ higher than one.

Otherwise, every time $\frac{\partial E}{\partial w_{ij}}$ changes its sign, a local minimum was missed, i.e. the last update was too large and $\Delta\theta_{ij}$ is decreased by η^-.

Simple put in equation form:

$$\Delta_{ij}^t = \begin{cases} \eta^+ \Delta_{ij}^{t-1} & \text{, if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^- \Delta_{ij}^{t-1} & \text{, if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \text{ where } 0 < \eta^- < 1 < \eta^+ \\ \Delta_{ij}^{t-1} & \text{, else} \end{cases} \tag{2.24}$$

Once the update-value for every weight is computed (this can be done very efficiently), another simple rule is defined for the sign of the weight-update: If the derivative is positive, then increasing the weight will lead to a higher error on the training criterion, so the weight is subtracted by the weight-update and vice-versa. The weight is added if the partial derivative is negative accordingly with equations (2.25) and (2.26) .

$$\Delta w_{ij}^t = \begin{cases} -\Delta_{ij}^t & \text{, if } \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ +\Delta_{ij}^t & \text{, if } \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \text{ where } 0 < \eta^- < 1 < \eta^+ \\ 0 & \text{, else} \end{cases} \tag{2.25}$$

$$w_{ij}^{t+1} = w_{ij}^t + \Delta w_{ij}^t \tag{2.26}$$

The update-values and weights are double-checked after a forward-pass.

ADAM

Previous methods considered all training data available when performing each step, e.g., a forward-pass and a backward-pass of considerably computational complexity and large memory usage. Several methods considering subsamples of data (specially since larger datasets are available today) were elaborated with considerable superior results. Another advantage relates to enabling distributed computation amongst several GPUs or machines to accelarate learning. Stochastic gradient descent (SGD) emerged as a stochastic method of choosing random batches from a training data set and performing gradient descent.

One of these methods is presented here: the ADAM[40] (adaptive momentum estimation) Alg. 3 algorithm. However, directly minimizing $\mathcal{L}$ is no longer possible, instead its expected value $\mathbb{E}(\mathcal{L})$ is minimized. The algorithm is considerably significant when dealing with methods of the form Sec. 3.9.

Algorithm 3 ADAM algorithm [40] with hyper-parameters α, β_1, β_2 and ϵ

procedure ADAM
  $\mathcal{L}(\theta)$ Stochastic loss function (generated with N samples, with θ parameters)
  $m_o \leftarrow 0$ (first moment vector)
  $v_o \leftarrow 0$ (second moment vector)
  $t \leftarrow 0$ (iteration counter)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta \mathcal{L}_t(\theta_{t-1})$ (current gradients to update next estimate)
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ (first moment estimate)
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ (second moment estimate)
    $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$ (bias-corrected from the first moment estimate)
    $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$ (bias-corrected from the second moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$
  end while
  return θ_t

2.5 Unsupervised Learning

In machine learning, unsupervised learning methods aim to extract features from unlabelled data, summarizing their useful properties with minimal error. Earlier approaches included *Cluster Analysis*, closely related to statistical methods as *Principal Components Analysis* [41]. Popular clustering algorithms comprise *k-means*, mixture models [42] and *Hierarchical Clustering* [43].

2.5.1 Auto-Encoder

Architectures emerged with the aim of testing whether Neural Networks could discover interesting representations in data, i.e., enable the capture of statistical regularities. *Diabolo-Networks*, *Autoassociators* or interchangeably *Auto-encoders* [44] were compared with *PCA* in research [45], showing improved results.

Extracting important features of the agent's sensors, hence reducing their dimensionality, transfigures any decision-making task into a more tractable form, solving in part the *Curse of Dimensionality* problematic. If the encoding is successful, the obtained code may be employed as a succinct representation of the environment.

Auto-encoders were initially designed to outperform PCA on complex (non-linear) data distributions. An under-complete architecture possessing a bottleneck is considered to infer meaningful representations by having a lower dimension code as stated above. Although an under-complete (hidden layer with a lower dimension than the input) architecture is very useful (and used in our work to reduce the dimensionality of Markovian states). Recent research covered over-complete auto-encoders often including a sparsity element as a mean to regularize and force the auto-encoder to learn meaningful features.

2.5.2 Auto-Encoder Description

An Auto-Encoder is a Multilayer Neural Network designed to reconstruct input data, with the purpose of further identifying key features, enabling intermediate representations to other endeavours. Auto-encoders present the following form:

- **X,** a vectorized input, represents common data, e.g, images, electric signals, text.
- **an encoder function** $F : \mathbf{X} \rightarrow \mathbf{Z}$**,** transforming input data into a intermediate representation, **Z**, by forward passing **X** through each *hidden layer*.
- **an internal coded representation Z,** a code produced by the encoder function F, the layer producing **Z** is often referred as *bottleneck*.
- **a decoder function** $G : \mathbf{Z} \rightarrow \tilde{\mathbf{X}}$**,** consisting of a forward pass from all *hidden layers* succeeding the encoded representation.
- **The reconstruction of the input** $\tilde{\mathbf{X}} = (G \circ F)\mathbf{X}$**,** to be used as a mean of testing our *Auto-encoder*.
- **a loss function** $\mathcal{L}(\mathbf{x},\hat{\mathbf{x}})$**,** underlying our intent to reconstruct the output Sec. 2.4 . MSE may be used, leading to $\arg\min_{G,F}(||X - (F \circ G)X||^2)$

A statistical effort to estimate parameters is partially converted into a *Deep Learning* optimization problem of minimizing a loss function. Schematically, *Auto-encoders* can be represented by the following Fig. 2.7.

Since a probability distribution is indirectly inferred from our training data, one might want to capture interesting specific characteristics, when the distribution has a lot of variance or is very

[4]http://ufldl.stanford.edu/tutorial/images/Autoencoder636.png

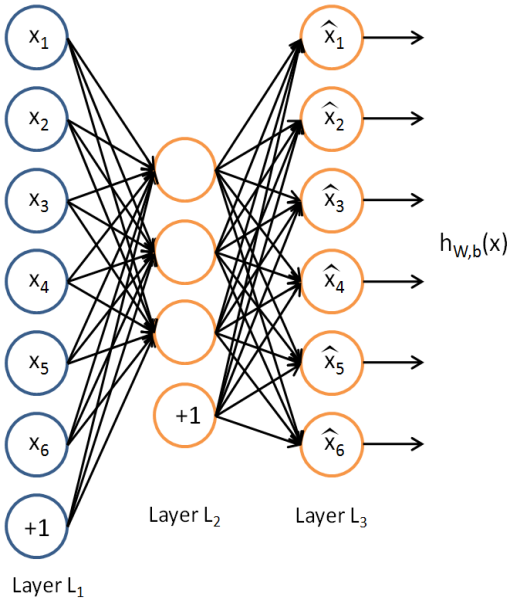


Figure 2.7: Illustration of an Auto-Encoder,**x** represents the input vector and $\hat{\mathbf{x}}$ a reconstruction.[4]

non-linear and should not be limited by the input dimension. As any neural network, it benefits from the same previous properties mentioned in Chapter. 2. Deep (or Stacked) Auto-Encoders surpassed shallow ones [3], being able to represent complex, highly non-linear distributions, which is reflected on the final converged training error (we also attest this later on Chapter. 5). Besides the bottleneck constraint, alternative constraints or regularization methods (such as the sparsity coefficient) have been used.

2.5.3 Restricted Boltzmann Machine

A Boltzmann Machine is a generative stochastic recurrent neural network that allows to solve difficult combinatorial problems, given enough time. One learning algorithm [46] allowed efficient training, although it was still slow and impractical for most applications. To potential the feasibility of training a *Boltzmann Machine* one could restrict connections within layers, forming a *bipartite graph* [47]. The two sets of nodes are usually referred as *visible units* and *hidden units* (or latent units).

A *Restricted Boltzmann machine* is an energy-based model, attributing a scalar energy to each possible configuration. In matrix form, we define energy as:

$$E(\mathbf{x},\mathbf{h}) = -\mathbf{h}^T\mathbf{W}\mathbf{x} - \mathbf{c}^T\mathbf{x} - \mathbf{b}^T\mathbf{h} \tag{2.27}$$

where, $\mathbf{x}$ is the input and $\mathbf{h}$ are the hidden layer units. The biases attached to the visible and hidden layer are $\mathbf{c}$ and $\mathbf{b}$, correspondingly.

The joint probability of a configuration is defined in terms of energy as [48]:

$$p(\mathbf{x},\mathbf{h}) = \frac{1}{\mathcal{Z}}\exp(-E(\mathbf{x},\mathbf{h})) \tag{2.28}$$

where $\mathcal{Z}$ is the partition function to normalize our distribution.

$$\mathcal{Z} = \sum_{\mathbf{v},\mathbf{h}} \exp(-E(\mathbf{x},\mathbf{h})) \tag{2.29}$$

The computation of the probability of each node simplifies to a marginal:

$$p(\mathbf{x}) = \frac{1}{\mathcal{Z}}\sum_{\mathbf{h}} \exp(-E(\mathbf{x},\mathbf{h})) \tag{2.30}$$

Considering a sigmoid 2.1 as a non-linearity is used, the probability of a hidden unit activation can be written as [48]:

$$p(h_j = 1|\mathbf{x}) = \Psi(b_j + \sum_i (x_i w_{ij})) \tag{2.31}$$

or for a visible unit:

$$p(x_i = 1|\mathbf{h}) = \Psi(a_i + \sum_j (h_j w_{ij})) \tag{2.32}$$

where $\Psi = sigmoid(a) = \frac{1}{(1+\mathrm{e}^{-a})}$. Hence, the objective is to maximize the product of probabilities

assigned to a training data $\mathbf{X}$ e.g,:

$$\arg\max_{W} \prod_{\mathbf{x}\in\mathbf{X}} p(x) \tag{2.33}$$

as in *maximum likelihood estimation*. Considering the log operator to ease computation, we can maximize the expected log probability as follows [49]:

$$\arg\max_{W} \mathbb{E}[log\, p(\mathbf{x})] = \arg\min_{W} \mathbb{E}[-log\, p(\mathbf{x})] \tag{2.34}$$

furthermore, the derivative of the log probability with respect to the parameters (weights) as a mean to perform stochastic gradient descent is determined [49]:

$$\frac{\partial}{\partial w}(-log\, p(x)) = \mathbb{E}[\frac{\partial E(x,h)}{\partial w}|x] - \mathbb{E}[\frac{\partial E(x,h)}{\partial w}] \tag{2.35}$$

the first term of the right side of the above equation is often referenced as *positive phase contribution* and the right one *negative phase contribution*.

The equation. (2.35) is used to update weights of the network, according to some criteria and a multiplying constant (just as in learning rate).

The training of a *RBM* was still known to be slow, despite simplifying the general Boltzmann machine. A faster method emerged, *Contrastive Divergence* [47]. The algorithm performs *Gibbs sampling* to perform gradient descent with a single sample.

2.5.4 Deep Belief Network

A *Deep Belief Network* is a generative neural network composed of several layers. Each layer represents a successively trained building block under an unsupervised learning algorithm.

Therefore, higher degrees of complexity are learned for each trained block. The first layers aim to identify edges in images or fundamental details on data. Further layers combine these features in a non-linear manner to identify traits in a higher level of abstraction. An example of the previous can be found in reference [3].

Several layers of *Restricted Boltzmann Machines* can be trained and stacked to perform dimensionality reduction by inferring statistical attributes on data. The first *RBM* is trained on input data. Afterwards, the second *RBM* is trained on features generated by the first, as if they were visible units.

Lastly, the *Deep Belief Network* is trained to tweak the weights closer to a better *minima* according to [50] and [51].

The *Deep Belief Network* can also be coupled with a soft-max layer for classification, e.g., to classify numbers with the MNIST digits dataset [3]. Nevertheless a *Deep Belief Network* may be further optimized under any supervised learning algorithm to achieve a good accuracy on a given benchmark [13].

[5] `http://doc.okbase.net/kemaswill/archive/17466.html`

2.5.5 Stacked Auto-Encoder

Instead of training a *Deep Belief Network* with *Restricted Boltzmann Machines* to perform *Unsupervised Learning* on a training data set, with the objective of dimensionality reduction, *Auto-Encoders* may be used instead. There are variations of *Stacked Auto-Encoders*, such as *Denoising Stacked Auto-Encoders* or *Variational Stacked Auto-Encoders*.

The pseudo-algorithm described in [3], also found [52] is presented in Alg. 4.

Algorithm 4 Greedy Layer-Wise Algortihm to train a Stacked Auto-Encoder from Auto-Encoders

procedure STACKED AUTO-ENCODER
  Transform each sample from the data-set $\mathcal{D}$ into a input vector by concatenating collumns
  Initialize vector $\mathbf{v}$ containing dimensions for each layer of the SAE
  $\mathbf{z_{aux}} \leftarrow \text{vertcat}(D)$
  for $i = 0$ to $dim(\mathbf{v})$ **do**
    Train auto-encoder AE_i, on input data $\mathbf{z_{aux}}$ with hidden layer of dimension $\mathbf{v}_i$
    Forward-pass input $\mathbf{z_{aux}}$ onto the fully trained auto-encoder AE_i and store
    each output from the hidden layer (or bottleneck) into $\mathbf{h}_{aux}$
    $\mathbf{z}_{aux} \leftarrow \mathbf{h}_{aux}$
    Copy the first matrix of weights/biases from AE_i into a SAE
  end for
  Fine tune SAE with $\mathbf{z}_{aux}$
  Return SAE

The stacked auto-encoder could be then coupled with a softmax-layer for classification (such as the case of handwritten numbers) or unfolded to obtain reconstructions and verify the accuracy of the model. In order to obtain a succinct representation of data, an experimental setup was created and tested on a simple dataset Chap. 5. The objective of the pre-training is setting the initial weights of the Stacked Auto-Encoder closer to a good local minimum Fig. 2.10.

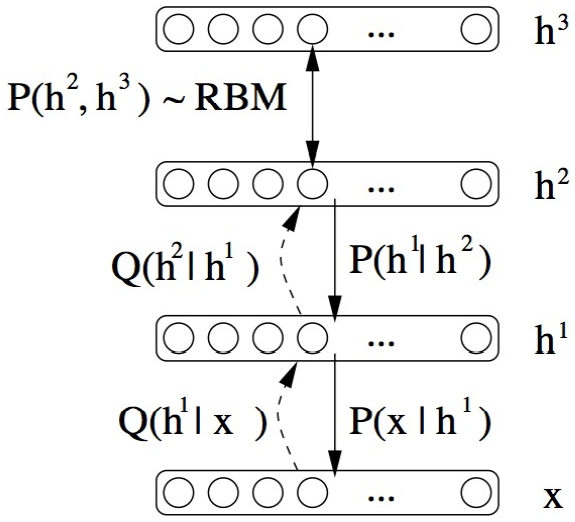


Figure 2.8: Deep Belief Network, pre-trained and stacked with a succession of RBM's.[5]

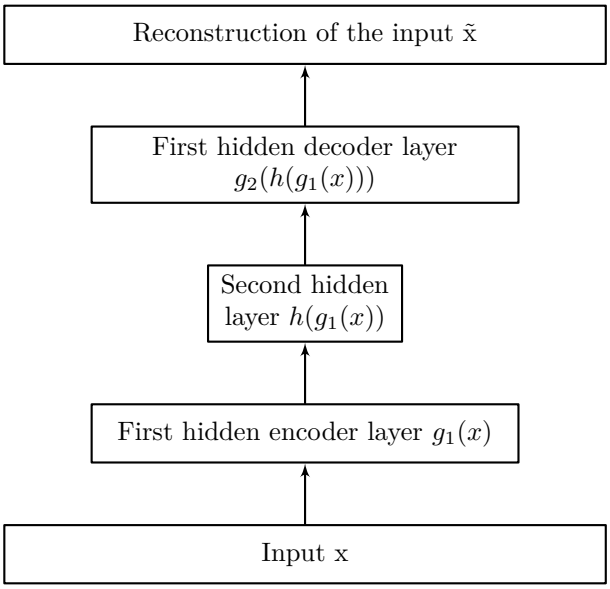


Figure 2.9: Stacked Auto-Encoder.

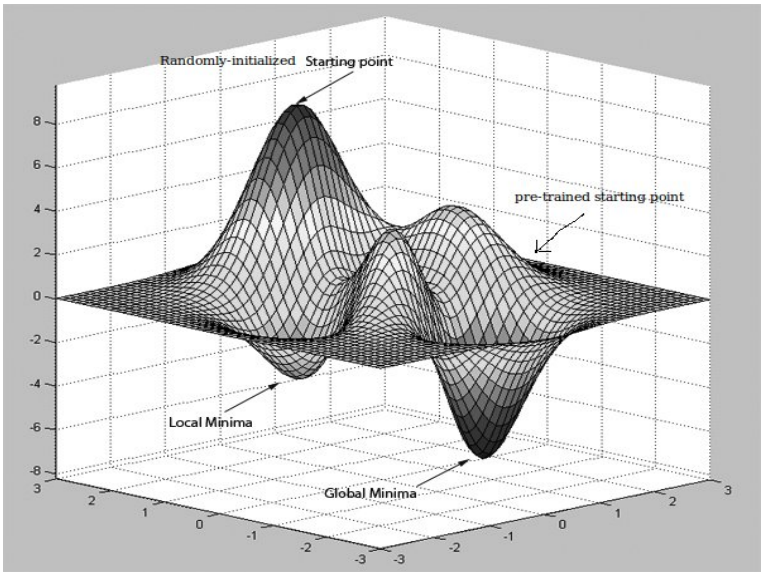


Figure 2.10: Effects of pre-training in neural networks.[6]

Although earlier research suggested that *Deep Belief Networks* had an advantage over SAE [3], recent research suggested that inference in energy-models is very similar to back-propagation [53]. Since a *Stacked Auto-Encoder* is easier to train then a *Deep Belief Network* (mainly for the reason RBM's are harder and slower to train even with contrastive divergence), this algorithm to perform dimensionality reduction was used.

2.5.6 Pre-training with RBM's vs AE's

One question emerges: "When creating a DBN, how exactly a RBM compares to a Auto-encoder in the training phase?". The question encompasses two main points: Which one is faster? Which one is more reliable (translating in better learnt features)?

In terms of training, contrastive divergence is still slower than training a feed-forward neural network like an auto-encoder to build a generative model.

The mathematical foundations of both gather several differences. RBM's are energy-based models benefiting from Boltzmann learning (like the Boltzmann factor for pattern recognition). AE's compute a reconstruction error and add a regularization term. In the case of a sparsity term, the network aims to reduce the KL divergence of our model.

Both RBM and auto-encoder, solve an optimization problem of the form:

$$\arg\min_{\boldsymbol{\theta}}(\mathcal{L} + regularization) \tag{2.36}$$

Both structures aim to minimize a reconstruction error. The fundamental mathematical approach for a RBM would require a full evaluation of the $\mathcal{Z}$ partition function. Unfortunately, since it is unpractical, *Contrastive Divergence* uses an approximation to perform gradient descent. This approximation can be seen, in a certain degree, as a regularization [54].

[6] http://stackoverflow.com/questions/34514687/how-does-pre-training-improve-classification-in-neural-networks

3| Reinforcement Learning

Contents

FOCUSING on the central decision-making concepts, elements required to understand the architectural and experimental work are described in this chapter. A recurrent problem in machine learning relates to teaching an agent to perform actions in order to achieve a certain goal or task in a specific environment.

A method to tackle decision-making problems comprises *Reinforcement Learning*, which has been inspired by behavioral psychology [55], using *temporal credit assignment*. Later, research developed several frameworks such as *Adaptive Heuristic Critic* algorithms [56], culminating with the formulation of *Q-learning* [57] explored later in section. 3.4.1.

3.1 Reinforcement Learning Problem

Reinforcement Learning as described in [58], is a learning framework where an agent/learner interacts with the environment in a trial and error manner. In contrast with other machine learning methods, the agent is not told the proper actions to take. Instead, the agent explores the environment to achieve the maximum amount of future rewards (or statistically the highest sum of expected rewards), usually in search for a goal/objective (or a target space) represented numerically by a large reward.

Therefore, this approach slightly differs from supervised learning, when, as knowledgeable designers, directly sample interactions and apply statistical pattern recognition. Since an agent is actuating and learning simultaneously, better methods need to be considered. Additionally, considering the hypothesis of a zero *a priori* knowledge about the environment a proper exploration/exploitation strategy is required. The final goal translates into progressively improve the sequence of actions given states of the environment, ultimately attaining the best policy/behavior for our agent.

Every learning problem portrays similar modules: a learner (agent), a teacher (reward function), a performance measurement of how well the learning agent is behaving (usually tests represented numerically by rewards) and a couple more variables to be considered; several properties comprising a *Reinforcement Learning* problem are unravelled. These function as a reference to untangle later procedures.

3.1.1 Environment

An environment constitutes a world for our agent to act and learn. For a real world task, as a robotic arm, a pole to balance, our environment could be the whole surrounding 3D space or the 2D images from a camera. It could encompass an entirely virtual world, posted messages (twitter/facebook) or a virtual game from an emulator (ATARI, OpenAI Gym). To clearly define a particular event within an environment, this thesis mathematically describes a state as a vector. A state of an environment can be represented by a space of dimension N contained within the state space of an environment D, e.g.,

$$\mathbf{s}_t \subset \mathbf{D}, \qquad \mathbf{s}_t \in \mathrm{R}^N \tag{3.1}$$

where $\mathbf{s}_t$ is a particular state at time t and $\mathbf{D}$ the state-space encompassing our world. The dimension of the state R^N is problem dependent. An appropriate dimensional representation of our environment is paramount with an impactful effect on our agent's learning. For each problem, there can be an overwhelming amount of different possible representations, identifying the key features enables for an easier endeavour.

For example, concerning the robotic arm problem, for most tasks, fully multidimensional track of every surrounding point is not required. Instead, a camera set-up generating 2D frames may fully describe the dynamics and variables of interest in question. Within each frame, dimensionality can be reduced by considering just one channel instead of four.

3.1.2 Rewards/Goal

In a Reinforcement Learning framework, an agent learns by reinforcement (as in psychology). A negative reward leads to an undesirable behaviour. Conversely, a sequence of positive rewards lead towards a appreciable policy. As studied in [58], the aim of our agent is to maximize the accumulate sum of rewards:

$$\mathbf{R}_t = \mathbf{r}_{t+1} + \mathbf{r}_{t+2} + \mathbf{r}_{t+3} + ... + \mathbf{r}_T \tag{3.2}$$

where $\mathbf{R}_t$ represents the return, t a particular time step up to T. In general, for stochastic environments, the goal is to maximize the expected value of the return within a task. Eq. 3.2 can be re-written as follows:

$$\mathbf{R}_t = \sum_{k=0}^{T} \mathbf{r}_{t+k+1} \tag{3.3}$$

However, a future reward may have a different present value. This rate is referred in literature as discount factor. A discount factor can be considered as a learning parameter, varying from $0 \leq \gamma \leq 1$. Every future reward at time stamp t is discounted by γ^{k-1}. If γ approaches zero, our agent considers rewards near the present state as much more valuable. In contrast, a γ is close to one changes our agent to behave greedily and consider future rewards as equally important. The return for a specific policy can be simply put as:

$$\mathbf{R}_t = \sum_{k=0}^{T} \gamma^k \mathbf{r}_{t+k+1} \tag{3.4}$$

3.2 Markovian Decision Processes

In order to mathematically formalize any decision-making effort, a common framework is required to compare real world tasks or theoretical problems. To further simplify our endeavour, memoryless processes are considered by using the *Markov Property* [59]. Extending *Markov chains*, adding actions to generate other possible outcomes, and rewards as a motivational input for differentiating higher quality states, we recall the definition of *Markov Decision Processes* [60]. Throughout this thesis, experiments were tested within environments able to be formulated by MDP's.

The standard Reinforcement Learning set-up can be described as a MDP, consisting of:

- **A finite set of states** D**,** comprising all possible representations of the environment.
- **A finite set of actions** A**,** containing all possible actions available to the agent at any given time.
- **A reward function** $r = \psi(s_t,a_t,s_{t+1})$**,** determining the immediate reward of performing an action a_t from a state s_t, resulting in s_{t+1}.

- **A transition model** $T(s_t,a_t,s_{t+1}) = p(s_{t+1}|s_t,a_t)$**,** describing the probability of transition between states s_t and s_{t+1} when performing an action a_t.

3.3 Functions to Improve Behaviour

Behaviour on the part of our agent was not yet mathematically specified. An agent is expected to progressively collect more rewards within each episode, thus actively learning by reinforcement. Each state is followed by an action, which leads to another state and corresponding reward. The agent's behaviour becomes a simple premise: "What action should I take for each state?".

3.3.1 Policy

A behaviour translates into a control *policy*, a map determining what action to be taken at each state. A policy can be deterministic when the execution of an action is guaranteed, or stochastic, when considering a certain probability involved. Therefore, a deterministic policy is mathematically represented by the following:

$$\pi(s_t) = a_t \quad , \quad s_t \subset D \quad a_t \subset A \tag{3.5}$$

If the actions are stochastic, the policy transforms into a probability distribution of a_t given s_t then the information has to be included in the policy:

$$\pi(a_t|s_t) = p_i \quad , \quad s_t \subset D \quad a_t \subset A \quad 0 \leq p_i \leq 1 \tag{3.6}$$

Theoretically, the policy gathering the most amount of rewards for a particular world is considered an optimal policy, denoted by π^*. Moreover, after applying a learning algorithm coupled with a proper exploration strategy until convergence, given sufficient episodes, the policy obtained can be considered "optimal" or "sub-optimal". The later is also referenced as *Bellman's Principe of Optimality* [61].

3.3.2 Value Function

A Value function evaluates the usefulness of a policy, given a state, $s_t \in D$ and following the same policy π thereafter. Usefulness of a policy comprises a gathered discounted sum of rewards Eq. 3.4.

$$V : V^\pi \rightarrow \mathbb{R}, \qquad V^\pi(s) = \mathbb{E}_\pi\{R_t|s_t = s\} = \mathbb{E}_\pi\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1|s_t=s}\} \tag{3.7}$$

Furthermore, a value function V^π may be estimated by "trial-and-error", since the expected value unfolds from experience samples within the task. One of the main advantages of unrolling a value function comprises with benefiting from dynamic programming properties, i.e., can be calculated recursively. Unfolding equation (3.7), a value function given a certain state is equal to

the following value plus a reward [58]:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\{R_t|s_t = s\} = \mathbb{E}_{\pi}\{\sum_{i=0}^{\infty}\gamma^i r_{t+i+1|s_t=s}\} = \mathbb{E}_{\pi}\{r_{t+1} + \sum_{i=0}^{\infty}\gamma^i r_{t+i+2|s_t=s}\} \tag{3.8}$$

If we consider the case of a stochastic policy π then equation 3.8 unravels into the *Bellman equation* of V^{π} [58]:

$$\begin{aligned} V^{\pi}(s) &= \sum_a \pi(a|s) \sum_{s_{t+1}} p(s_{t+1}|s_t,a)[r(s,a,s_{t+1}) + \gamma E_{\pi}[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_{t+1} = s_{t+1}]] \\ &= \sum_a \pi(a|s) \sum_{s_{t+1}} p(s_{t+1}|s_t,a)[r(s,a,s_{t+1}) + \gamma V^{\pi}(s_{t+1})] \end{aligned} \tag{3.9}$$

3.3.3 Quality function

There are several methods to obtain a (close to optimal) policy empirically. One of this methods uses a quality function [58] (interchangeably Q-value) to evaluate the quality (estimated accumulated rewards following a certain policy) of applying an action to a specific state, discussed on Section. 3.4.1. Q-function has a similar definition to the value function but also takes into consideration an action. It comprises the long term rewards of applying an action to a state and following the considered policy π thereafter [58]:

$$Q : S \times A \to \mathbb{R} \quad Q(s,a)^{\pi} = \mathbb{E}_{\pi}\{R_t|s_t = s,a_t = a\} = \mathbb{E}_{\pi}\{\sum_{i=0}^{\infty}\gamma^i r_{t+i+1|s_t=s,a_t=a}\} \tag{3.10}$$

If an optimal policy π^* is considered, the value of a state $V^{\pi^*}(s_t)$ is equal to the Q-value $Q^{\pi^*}(s_t,a_t)$ when taking the optimal action [58]:

$$s_t \subset D \quad a_t \subset A \quad V^{\pi^*}(s_t) = Q^{\pi^*}(s_t,a_t) = \arg\max_a Q^{\pi}(s_t,a_t) \tag{3.11}$$

3.4 Temporal Difference Learning

Temporal-difference learning [58] (or TD) interpolates ideas from Dynamic Programming (DP) and Monte Carlo methods. TD algorithms are able to learn directly from raw experiences without any particular model of the environment. Whether in Monte Carlo methods, an episode needs to reach completion to update a value function, Temporal-difference learning is able to learn (update) the value function within each experience (or step).

The price paid for being able to regularly change the value function is the need to update estimations based on other learnt estimations (recalling DP ideas).

Whereas in DP a model of the environment's dynamic is needed, both Monte Carlo and TD approaches are more suitable for uncertain and unpredictable tasks. Since TD learns from every transition (state, reward, action, next state, next reward) there is no need to ignore/discount some episodes as in Monte Carlo algorithms.

3.4.1 Q-Learning (Off-Policy)

A quintessential TD off-policy control algorithm is the one-step Q-learning by Watkins [57]. The algorithm is very similar to Sec. 3.4.2, except on the update step, where the maximum possible future value is used instead of using the whole experience tuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. Q-learning is considered an off-policy algorithm since we approximate the optimal Q-value function, Q^*, independently of the current policy. This algorithm presents itself as a slighter greedier version of the next one, and further implications are discussed in Sec. 3.5.3. The update step is as follows:

$$Q : S \times A \rightarrow \mathbb{R} \tag{3.12}$$

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_{t+1} + \gamma max_a Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)) \tag{3.13}$$

The α in Eq.(3.13) is the learning rate ($0 < \alpha \leqslant 1$) and determines the rate at which new information will override the old Q-value. γ is the discount factor ($0 < \gamma \leqslant 1$) which decreases the estimated Q-values for future states.

Algorithm 5 Q-learning (Off-Policy)

```
1:  procedure Q-Learning
2:      Initialize Q(s,a) = 0 for all a ∈ A and s ∈ S
3:      Repeat until the end of the episode:
4:      s_t ← InitialState
5:      for each episode step do
6:          Select a_t, based on a exploration strategy from s_t
7:          Take action a_t, observe r_{t+1}, s_{t+1}
8:          Q(s_t,a_t) ← Q(s_t,a_t) + α(r_{t+1} + γ max_a Q(s_{t+1},a_{t+1}) − Q(s_t,a_t))
9:          s_t ← s_{t+1}
10:         if then s == terminal then
11:             Q(s_{t+1},a_{t+1}) = 0
12:         end if
13:     end for
```

3.4.2 SARSA (On-Policy)

SARSA is a On-Policy temporal-difference reinforcement learning method to estimate a quality function. It heritages the name from: (State, Action, Reward, Next State, Next Action). The update step is as follows:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)) \tag{3.14}$$

The Q value is updated through interactions with the environment, thus updating the policy depends on the action taken. The Q-value for a state-action pair is not directly updated, but gradually adjusted with a learning rate α. As with the Q-learning algorithm, SARSA will keep on improving its policy towards a better solution as long as all state-action pairs continue to be updated. The SARSA-learning algorithm found in literature [58] is presented at algorithm 6 .

Algorithm 6 SARSA-learning

```
1: procedure On-Policy
2:     Initialize Q(s,a) arbitrarily for all a ∈ A and s ∈ S
3:     Repeat until the end of the episode:
4:     s_t ← InitialState
5:     Select a_t based on a exploration strategy from s_t
6:     for each episode step do
7:         Take action a_t, observe r_{t+1}, s_{t+1}
8:         Select a_{t+1} based on a exploration strategy from s_{t+1}
9:         Q(s_t,a_t) ← Q(s_t,a_t) + α(r_{t+1} + γQ(s_{t+1},a_{t+1}) − Q(s_t,a_t))
10:        s_t ← s_{t+1}
11:        a_t ← a_{t+1}
12:        if then s == terminal then
13:            Q(s_{t+1},a_{t+1}) = 0
14:        end if
15:    end for
```

3.5 Exploration vs Exploitation

The exploration vs exploitation dilemma is a recurring theme in reinforcement learning and AI in general. Should we exploit acquired knowledge i.e., should we follow a known high-reward path? Or should we explore unknown states in search for a better new policy?

The balance between both highly improves our agent's learning performance. One plausible answer comes to surface: First an agent is required to explore the largest amount of states, following by exploiting gathered knowledge to acquire better results upon the agent being confident enough of having deeply explored the world.

For uncertain/dynamic environments, it is very complex to acquire whether sufficient exploration has been employed. Nevertheless, a couple of methods to select actions through a quality function will be described.

3.5.1 Epsilon-Greedy Strategy Selection

A possible, simple, yet effective method to select an action at each time step is presented as the $\epsilon-$greedy selection strategy. Given a quality function $Q(s,a)$, the best action is selected (one that maximizes a Q-value for a given state) with a probability of $(1-\epsilon)$. Furthermore, ϵ is contained within an interval of zero and one ($0 \leq \epsilon \leq 1$), since it represents a probability. With probability ϵ an action is taken randomly from the action set. A description of the algorithm [58] is found below:

If our Q-value function returns a vector instead of a scalar, we trade the linear loop of actions for a simple *max* operator. Remaining ϵ constant over time, when calling procedure Alg. 7, limits our ability to exploit knowledge gathered from experiences. A slightly better strategy would be to decrease ϵ in order to highly explore at the beginning of the task and decrease exploration over exploitation over time. The previous is mentioned in literature as a ϵ-greedy decreasing strategy.

A variety of family functions could be used to picture ϵ's decrease. Furthermore, an initial

Algorithm 7 ϵ Greedy Strategy

```
1: procedure ACTION = ε-GREEDY(ε, s = state)
2:     Initialize a_aux ∈ A_s
3:     if then ε ≥ rand(0,1)
4:         Select a random action a_i ∈ A_s from the action space
5:         a_aux = a_i
6:     else
7:         Initialize Q_aux = 0
8:         for each action a_i ∈ A_s do
9:             Compute Q(s,a_i) based on s = state
10:            if then Q(s,a_i) ≥ Q_aux
11:                Q_aux = Q(s,a_i)
12:                a_aux = a_i
13:                Break
14:            end if
15:        end for
16:    end if
17:    Return a_aux
```

ϵ_0 is chosen to be close to one (full exploration strategy) at the beginning of the experience. A discount ϵ factor can be computed to obtain dr_ϵ. Finally, a new ϵ is computed after each call of Alg. 7.

$$\epsilon_t = \frac{\epsilon_0}{1 + t dr_\epsilon} \tag{3.15}$$

3.5.2 Boltzmann Exploration

One major intrinsic flaw of ϵ greedy selection relies on considering one action as the best (highest Q-value), and considering all other *bad* actions as equiprobable. While drawing a *bad* action from an uniform distribution, information about the relative quality of each action is discarded (Q-values not considered) loosing potential exploitation gains.

Furthermore, the problem highly intensifies if the gap between Q-values is large and *second-best* actions are severely penalized. Using the Boltzmann distribution (thus Boltzmann exploration [62]), also referenced as the soft-max selection method, knowledge about every $Q(s_t,a_t)$ is used in the following equation:

$$p(a_t|s_t,Q_t) = \frac{\exp^{Q(s_t,a_t)/\tau}}{\sum_{b \in A_D} \exp^{Q(s_t,b)/\tau}} \tag{3.16}$$

where τ is recalled as *temperature* from thermodynamics. If τ presents high values, each numerator is pushed to one leading to equiprobable actions, thus rewarding exploration. Decreasing τ with the number of episodes, exploration becomes more *greedy*, giving higher probability to more promising actions (instead of just the most promising one).

3.5.3 On-Policy versus Off-Policy Methods

Whereas On-Policy methods use the action chosen to update the action-value-function on the same policy, Off-Policy choses a different policy to update the same action-value function.

Both approaches have their merit and the gap between them also relates to the action selection method we choose to use, often a dilemma referred as exploration/exploitation problem. If the simplest exploration strategy is considered, i.e, $\epsilon-$ greedy Sec. 3.5.1 performs within an On-Policy method, the action selection will be immediately introduced in the update step [58]. In contrast, an Off-Policy methods as *Q-learning*, opts for the "greediest", "optimal" estimate, and discards the weight of a bad move on part of the agent converging faster towards a sub-optimal/optimal policy. However, On-Policy methods as SARSA avoid states which imply a greater amount of risk of obtaining a negative reward.

3.6 Function Approximation

Applying *Reinforcement Learning* algorithms to usual control tasks or decision-making problems, always arose several challenges, especially in continuum domains. Earlier RL problems were tested and solved in a tabular manner. Each (*state*,*action*) pair had a well defined tabular form. Gathered state-action values were accessed and stored during algorithm updates, turning rather complex tasks into large decision trees, often infeasible, due to memory and time limitations. Specifically when dealing with high-dimensional states, a problem known as *Curse of Dimensionality*. However, new challenges have to be dealt with. Problems inherent to optimization raise several concerns, such as over-fitting, over-training, initialization of parameters amongst many.

In order to apply *Reinforcement Learning* to *real-world* tasks, a set of known inherent issues have to be considered:

- **The world is not a discrete place,** specifically in robotics, scientists often have to deal with continuum spaces and times.
- **We introduce a strong bias towards what we, as researchers add, when elaborating states,** by handcrafting features/filters, or creating optimistic heuristics, self-limiting learning.
- **For high dimensional states,** specifically in dynamic environments, tracking all environmental variables is infeasible.

Instead of using tables, one could use several *machine learning* algorithms to generalize information from an arbitrary quantity of samples. Since *function approximation* can be performed through a variety of *supervised learning* procedures, a plethora of methods are available such as *pattern recognition*, *statistical curve fitting* and many others. This thesis aims to explore *Deep Learning* methods alongside *Reinforcement Learning* algorithms.

Deep Learning can be applied in multiple domains such as: transcription, machine translation, anomaly detection, synthesis and sampling, de-noising amongst many other. The most popular ones consist of classification and regression:

- **Classification,** discrete output, a given input is classified as belonging to a discrete group. An example of this can be face identification, object recognition and the classification of handwritten digits.

- **Regression,** continuous output, the required output is real valued. An example of this can be building a model to estimate house-pricing and generally fitting a multi-dimensional surface on a dataset.

3.7 Deep Learning approach to Reinforcement Learning

This thesis aims to explore a solution started with algorithms inspired in the works of recent Deep Learning applications ([21] and [63]) with efficient practices elaborated regarding the genesis of a Q-value function [8]. Deep learning will enhance reinforcement learning, solving (in some degree) the *Curse of Dimensionality* by doing unsupervised learning on input data to obtain state representations and supervised learning on a Q-value function approximation. Starting with a raw data input vector, which could be black and white pixels from an image, a music, or electromagnetic signals of any kind. In sum, information our agent receives from the world:

$$\mathbf{x}_{input} \in \mathbb{R}^D \tag{3.17}$$

where $\mathbb{R}^D$ is the dimension of the input vector. The analysis proceeds by extracting properties with a unsupervised learning algorithm, where we extract the components with the highest variance from the data.

Auto-Encoders are typically a feed-forward neural network. A Linear Factor model only specifies a parametric encoder, whereas an auto-encoder specifies a decoder and an encoder.

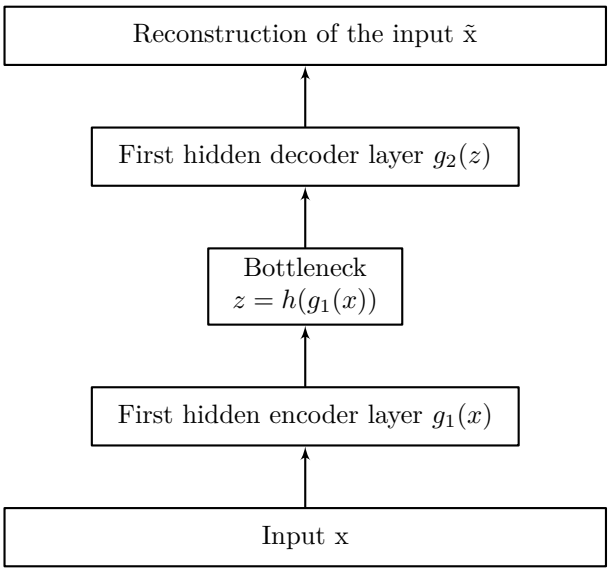


Figure 3.1: Stacked Auto-Encoder.

We will use Neural Networks with non-linear activations, *Stacked Auto-Encoders* and denote each hidden layer as a composite function of other hidden layers, by the following notation:

$$\begin{aligned} \mathbf{z} &= h(g_1(\mathbf{x}_{input})) \in \mathbb{R}^B \\ \widetilde{\mathbf{x}} &= g_2(\mathbf{z}) \in \mathbb{R}^D \end{aligned} \tag{3.18}$$

k

where g_1 represents the hidden layers up to the intermediate representation according to Fig. 3.1, h the non-linearity of the bottleneck layer, $\mathbf{z}$ the code we obtain from $h(g_1(\mathbf{x}_{input}))$ and $\tilde{\mathbf{x}}$ the reconstruction of the input with g_2 being a composite function of several hidden layers from the bottleneck to the output layer. $B > D$ the Stacked Auto-Encoder is said to be over-complete. Figure. 3.1 represents a simple Stacked Auto-Encoder but both g_1 and g_2 could be a composite of several hidden layers, trained in a similar fashion 2.5.5. In this thesis an under-complete Auto-Encoder is used to compress raw data into more manageable states. This is specially important for reinforcement learning, since data efficient methods to perform a large amount of iterations and deal with learning in real-time are paramount.

Our states are vectors of features with a finite action space, and actions will be deterministic, hence:

$$s_t = \mathbf{z} \in \mathbb{R}^B \quad \pi(s_t) = a_t \quad \text{with} \quad a_t \in \mathbb{R}^A \tag{3.19}$$

Starting with the Bellman expectation equation, we may compute a value function Q^π for a certain policy π, thus:

$$\begin{aligned} Q^\pi(s_t,a_t) =& \mathbb{E}_{s_t}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|(s_t,a_t)] \\ =& \mathbb{E}_{s_{t+1}}[r_{t+1} + \gamma\ Q^\pi(s_{t+1},a_{t+1})|(s_t,a_t)] \end{aligned} \tag{3.20}$$

or the Bellman optimality equation, unrolling an optimal value function Q^*:

$$Q^*(s_t,a_t) = \mathbb{E}_{s_{t+1}}[r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1},a_{t+1})|s_t,a_t] \tag{3.21}$$

From here, one could directly search for the optimal policy, on the policy space, solving iteratively the Equation. (3.20), known as policy iteration. In a deterministic environment the expectation simplifies to:

$$Q_{i+1}(s_t,a_t) = r_{t+1} + \gamma\ Q_i(s_{t+1},a_{t+1}) \tag{3.22}$$

or directly solve Equation. (3.21) by value iteration:

$$Q_{i+1}(s_t,a_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q_i(s_{t+1},a_{t+1}) \tag{3.23}$$

Since our state space might still be very large, an approximation for $Q^\pi(s_t,a_t)$ is considered. We will use a neural network, which might be refereed as a "Deep Value function", in analogy to Deep Neural networks, thus obtaining a non-linear value iteration:

$$Q^*(s_t,a_t;\boldsymbol{\theta}) \approx Q^*(s_t,a_t) \tag{3.24}$$

the parameters of the neural network, e.g, weights and biases are represented by $\boldsymbol{\theta}$. Finally, after stipulating a neural network to represent our value function, one needs to define an objective function (loss function) to define train. To accomplish this, tuples of experiences $(s_t,a_t,s_{t+1},r_{t+1})$ from the gathered data set (also referenced as replay memory) $\mathcal{D}$ have to be sampled. Since it is a regression problem, we will use mean square error to propagate parameter changes, on the following

manner:

$$\mathcal{L}_i(\boldsymbol{\theta}_i) = \mathbb{E}_{(s_t,a_t,s_{t+1},r_{t+1})\sim\mathcal{D}}[(r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1};\boldsymbol{\theta}_{i-1}) - Q(s_t,a_t;\boldsymbol{\theta}_i))^2] \tag{3.25}$$

therefore $r + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1};\theta_{i-1})$ is our network target (critic term) and $Q(s_t,a_t;\theta)$ is the current output of the network (actor term) after a forward-pass. Our goal is to lower the expectation of them being different after every iteration. The gradient of Eq. (3.25) is presented:

$$\nabla_{\boldsymbol{\theta}_i}\mathcal{L}_i(\boldsymbol{\theta}) = \mathbb{E}_{(s_t,a_t,s_{t+1},r_{t+1})\sim\mathcal{D}}[(r_{t+1}+\gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1};\boldsymbol{\theta}_{i-1})-Q(s_t,a_t;\boldsymbol{\theta}))\nabla_{\boldsymbol{\theta}_i}Q(s_t,a_t,\boldsymbol{\theta}_i)] \tag{3.26}$$

The objective function is then optimized end-to-end, according to an optimization procedure, using $\nabla_{\theta_i}\mathcal{L}_i(\theta_i)$.

A Neural Fitted Q-funcition [63] was elaborated and a small variation [8] considered a fully connected network of different output layer dimension (one Q-value per action to increase overall computational performance).

Both Networks (*Stacked Auto-Encoders* and *Feed-Forward Network*) are connected, although they are pre-trained and trained separately. The final architecture is tested on a mini-game.

In addition, the fully connected network regarding Q-function approximation is also tested (same hyper-paramaters, optimization procedures, non-linearites but different dimensions) [8] in two classic control problems at Chapter 5.

3.8 Neural Fitted Q Iteration (NFQ)

A possible approach of fitted Q-learning and neural networks was first suggested by Riedmiller [63]. Whereas a neural network needed to be re-trained after every action on an online methodology (expensive and impracticable method), the NFQ approach allows for major computational gains by enabling the possibility of training a *Q-function* approximation within a training set on a supervised manner using fast converging optimization procedures. To accomplish this, a set of tuples is gathered (memory set), which represents the acquired information (state, action, following state and resulting reward) of each interaction from each episode.

Afterwards a Memory set is used (using an heuristic [64] or a sampling method [8]) as data to do supervised learning on our neural architecture. After each episode, a new *Q-function* is re-trained by fitted Q-learning iterations, where the past Q-values are obtained by doing a forward-pass on the previous *Q-function*. Finally, end-to-end training is performed by adding the respective rewards to the obtained Q-values.

After reaching convergence, the new *Q-function* approximation is used to generate a new/-better episode. A Neural Fitted Q pseudo algorithm is presented Alg. 8

From DQN [8] algorithm, was acknowledge a change in the Neural Fitted Q-Iteration from one output to one output per action using only states as inputs to lower the computational requirements (since a cycle of forward-passes proportional to the number of actions would have to be performed). Following the previous, to enable a possibly better approach, several tests were conducted at later experiments in Chapter. 5. The pseudo NFQ algorithm is similar for both Q-networks except on

training and obtaining Q-values. For the Q-network with one single output, a loop is required to obtain the best action. For the second Q-network a max-operator is suficient to obtain the best action from **G** Alg. 8.

Algorithm 8 Pseudo Neural Fitted Q Iteration with one Q-value for each action at the output

Initialize Neural Network
Initialize Memory Set D
Initialize Pattern Set P
Repeat N times:
$s_t \leftarrow InitialStateVector$
for each episode do
    Select a_t **according to** ϵ**-greedy(ForwardPropagate(nnetQ,**s_t**,**ϵ**))**
    Take action a_t**, observe** r_{t+1}**,** s_{t+1}
    Store Experience:
    $D.s_t \leftarrow s_t$
    $D.s_{t+1} \leftarrow s_{t+1}$
    $D.r_{t+1} \leftarrow r_{t+1}$
    $D.a_t \leftarrow a_t$
    if $s == terminal$ **then**
        Break
    end if
    Train after every k^{th} **step:**
    if $mod(N,k) == 0$ **then**
        for number of NFQ iterations do
            for number of experiences j do
                $\mathbf{P}_{1,j} \leftarrow (1 - \alpha)ForwardPropagate(nnetQ,D.s_t) + \alpha(D.r_{t+1} + \gamma max_a ForwardPropagate(nnetQ, D.s_{t+1}))$
                $\mathbf{P}_{1,j} \leftarrow D.s_t$
            end for
            while nnetQ.error $<$ **threshold do**
                $nnetQ.error \leftarrow training_step(nnetQ,P)$
            end while
            Reset Pattern Set P
        end for
    end if
end for

3.9 Brief description of the DQN Algorithm

One recent successful approach of applying *Deep Learning* with *Reinforcement Learning* was introduced by a Google research group [8]. The employed architecture was able to achieve high-performances (i.e, above human-level performance) on a variety of Atari games. The algorithm used a single channel of pre-processed frames as an input, and had only access to the score (functioning as a reward).

The method employs as training criterion $\mathcal{T}$ the mean squared error between our current Q-value estimate and our targets of interest (the Q-learning update). Eq .3.25 also changes to

accommodate our target network in the following manner: [65]

$$\mathcal{L}(\theta_{\mathbf{i}}) = \mathbb{E}_{s_{t+1},a_{t+1}}[(r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1},\theta_{\mathbf{i-1}}) - Q(s_t,a_t,\theta_i))^2] \tag{3.27}$$

where $r + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1},\mathbf{w_{i-1}}) = y_i$ represents a constant *critic* term of our Q-update. The errors are then propagated to the current Q-network where the first Jacobian (derived from the output layer) is[65]:

$$\nabla_{\theta_i}\mathcal{L}(\mathbf{w}) = \mathbb{E}_{s_{t+1},a_{t+1}}[(y_i - Q(s_t,a_t,\theta_i))\nabla_{\mathbf{w}}Q(s_t,a_t,\theta_i)] \tag{3.28}$$

therefore, the whole network maps states to actions while using a variant of batch learning to decorrelate observations, smoothing oscillations while iterating.

Algorithm 9 DQN Pseudo-Code

```
Initialize DQN
Initialize Replay Memory D
Repeat M episodes
s_t ← InitialStateVector
for each episode step do
    Select a_t according to ε-greedy(ForwardPropagate(DQN,s_t,ε))
    Take action a_t, observe r_{t+1}, s_{t+1}
    Store Experience:
    D.s_t^i ← s_t
    D.s_{t+1}^i ← s_{t+1}
    D.r_{t+1}^i ← r_{t+1}
    D.a_t^i ← a_t
    Sample a mini batch of K experiences from D
    Compute Q-update for each experience using DQN and the target DQN
    Do a single training step on the DQN with the batch K
    Every k^th steps transfer paramaters from DQN to target DQN:
    if mod(N,k) == 0 then
        targetDQN.θ ← DQN.θ
    end if
    if thens == terminalthen
        Break
end for
```

In [8], two key features supported by empirical evidence enabled improvements compared to previous methods:

- **Replay Memory,** storing experience tuples (s_t,a,s_{t+1},r_{t+1}). These tuples are randomly sampled in a batch one step learning manner for the train of DQN.

- **Target Network,** a copy of weights and biases every n^{th} iteration is performed. Targets are later used to compute the *critic* term of the *Q-learning iteration.*

3.10 Deep Reinforcement Learning Scheme and Discussion

An endeavored effort towards searching for alternative *Reinforcement Learning* frameworks was made in order to draw conclusions about *Deep Learning* applications. Despite *Deep Learning*'s rapid metamorphose as a field due to active research, new (as modern as possible) and old concepts were reviewed as a starting point to understand the validity of our initial hypothesis.

In this thesis, several architectures for decoupling state-perception from state-action generalization behaviour were briefly explored. In the thematic of state-perception, there is plenty of research to be made. Collecting and sparsing data towards a better deep understanding, namely in the form of feature vectors seem quintessential in real-time applications. In this particular regard, interesting research has been published [28], where variational auto-encoders were used alongside other methods to extract meaningful features to be efficiently applied on self-driving technology. Therefore, unsupervised learning methods, present and future (not restricted within neural networks) may emerge with increasing efficiency to deal with the *Curse of Dimensionality*.

Another interesting element in supervised learning relates with generalization, useful for large state-spaces. Enabling models of sufficient statistics to circumscribe our agents behavior, specifically on continuum-domain tasks, is paramount if one wants to implement an agent in real-time, low-memory, portable devices. In this regard, interesting work has been developed with *Deep Learning* in robotics [27], having continuum torques as targets of the network, directly applied into the motors.

Research will appear to test different neural configurations, blocks, activations, recursive approaches to deal with probability models for every transition or, in a near future, LSTMs to deal with long term dependencies. This thesis aims to test some implementations and variations inspired in similar works. The Architectures used here were mainly inspired by [21] and [8].

The state reduction in [8] is performed by a conv-net (convolutional network and a standard method for object recognition) known to be efficient in an Atari game type of environment, since it is locally and translational invariant, with the state reduction being performed by filters (specifically max pooling layers) detecting each moving object on the screen. Additionally, we aim to further test the following:

- **A Stacked-Autoencoder generates succinct states and a reconstruction,** retrieved by a decoder from each code. Auto-encoders are also flexible towards the type of data.
- **Generalization within our Q-function,** regarding a fully connected Q-function approximations (fitted-Q, online or off-line approach), several methods were explored.

In this thesis, a Stacked Auto-Encoder was selected, as suggested in [21] and for the Q function-aproximation [63] with a small alteration inspired on [8] to lower the computational costs by a factor of the number of actions (each time we access the maximum Q-value for every state in every iteration). In sum, procedure 2.5.5 was applied for the state reduction and the altered version of NFQ described in Sec. 3.8. Furthermore, a partial DQN Sec. 3.9 (the fully connected structure) is presented as a less computational expensive alternative to NFQ. Nevertheless, other supervised/unsupervised methods might emerge in the forthcoming years. A general scheme Fig. 3.2

and an algorithm Alg. 10 are both presented. Moreover, specific implementations are designed and described in Chap. 4 and finally tested at Chap. 5.

Algorithm 10 Deep Reinforcement Learning Pseudo Algorithm

```
Initialize Neural Network
Initialize Pattern Set P
Initialize Memory Replay D
for each episode step do
    Observe o_t and convert to s_t
    Select a_t according to ε-greedy(ForwardPropagate(nnetQ,s_t,ε)
    Take action a_t, observe r_{t+1}, o_{t+1}
    Store Experience in D
    Use unsupervised learning method with o observations from D
    Use encoder F : X → Z to convert observations o into s = z
    P.s_t^i ← s
    P.s_{t+1}^i ← s_{t+1}
    P.r^i ← r
    P.a^i ← a
    if s == terminal then
        Break
    end if
    run(SupervisedLearningMethod(nnetQ,P))
    Reset Pattern Set P
end for
```

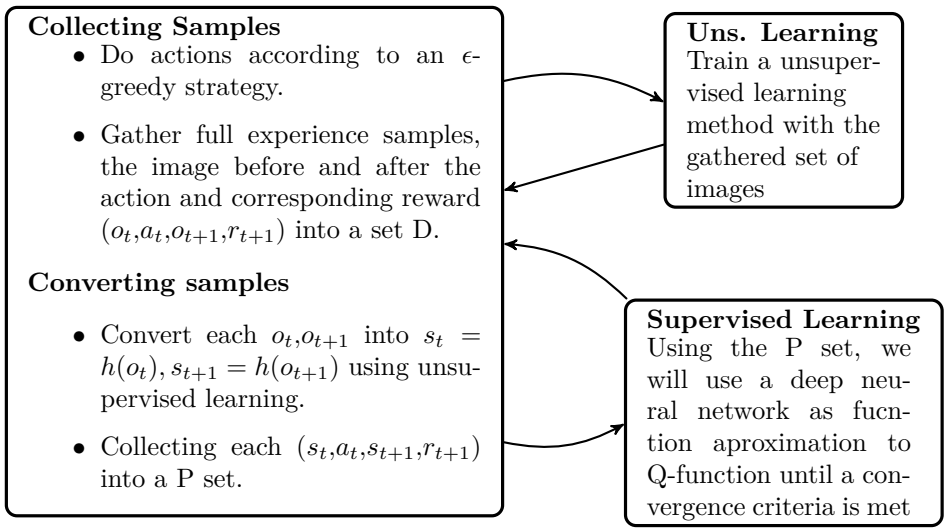


Figure 3.2: Deep Reinforcement Learning Scheme.

4| Experimental Architectures

Contents

Based on the presented concepts thus far, our goal relies on substantiating *Reinforcement Learning* implementations while integrating *Deep Learning* techniques. Despite the wide range of possible applications, specially since prolific modern research is being held, this thesis narrowed down to a handful of illustrative experiments, able to be iterated in future research.

From the discussion in Sec. 3.10, several concepts will be further tested to gather a meaningful confirmation and plausible conclusion. Neural networks will be used as a vehicle to perceive and decompose high-dimensional inputs. For complex *Reinforcement Learning* applications, a succinct internal representation of the environment is paramount. The previous increases in relevance for future subjects as *transfer learning* and *curiosity mechanisms*. In a preliminary phase, recalling Sec. 2.5 several blocks are built, namely *Auto-Encoders* and *Stacked Auto-Encoders*.

To accomplish the previous, a simple Maze is implemented with the purpose of outputting 10.000 pixel images. Several Auto-encoders are designed and trained to verify the importance of pre-training a larger neural network, and to access the representational power of both shallow and stacked auto-encoders. Comprehensive experiments are conducted with several dimensions in Chapter. 5.

Another relevant matter consists on using neural networks to perform Q-learning updates from gathered interactions with the environment. Two similar structures are built and tested in order to achieve a better understanding of "what and why" are the best practices when employing this methods on future problems.

To achieve the former, two Q-function approximations $Q(s,a;\boldsymbol{\theta})$, one with states and actions as input and Q-values and the output, and another with states as input and one Q-value for each action at the output layer. To perform a fair comparison between both, an equal number of parameters was considered.

To attest the validity of a Q-function approximation for larger state-spaces, two classic control problems were considered: Mountain Car problem; and the Cart Pole problem.

A third architecture is built with Sec. 3.9 as a main reference, to lower the memory and overall computational resources required. The objective of the last architecture is to verify whether the regression capacity of a similar structure would perform in a much larger state-space, as an effort to point out future engineering applications.

4.1 Proposed Environments

4.1.1 Toy Example: Simple Maze

To validate our architecture an accessible environment was chosen with a known optimal policy such as a maze, similar to [21]. The maze's dynamics, are described as follows:

- **A finite set of states** S**,** each image (100×100 pixels used, resulting in 10000-dimensional input vectors) produced by the interaction between the maze and our learning agent.
- **A finite set of actions** A**,** four actions, $(a_t = 0 : left)$, $(a_t = 1 : right)$, $(a_t = 2 : up)$ and $(a_t = 3 : down)$.
- **A reward function** $r = R(s_t,a_t,s_{t+1})$**,** the immediate reward of performing a given action a_t in the maze: $r = 0$, if the agent moves to an empty space; $r = -1$, if a wall is met; and $r = 1$, if the agent reaches the goal.

Each episode will be the result of the interaction between the game and our learning agent until either it stumbles on a wall or reaches the goal. Making the goal protected by walls will ensure a slightly better testing. Despite the maze's simplicity, the game's main creation purpose was to test unsupervised learning methods Sec. 2.5 and supervised learning with the Neural Fitted Q-Iteration Sec. 3.8.

4.1.2 Problem: Mountain Car

This classic problem was first described in [66]. A car starting from the bottom of a two-hill mountain aims to reach the top of the right hill. The car is not able to reach the top by only applying one action (right velocity), it has to balance to the left first and gain momentum on the descent.

The dynamics of this problem have the following equations for each time step:

$$\begin{aligned} x_{t+1} &= x_t + v_{t+1}\tau \quad , x \in [-1.2,0.6](m) \\ v_{t+1} &= v_t + (a-1)0.001 + cos(3x_t)(-0.0025)\tau \quad , v \in [-0.07,0.07](m/s) \end{aligned} \tag{4.1}$$

where $\tau = 1 second/step$ and a varies from (0,1,2).

- **Environment dynamics** $\mathbf{s}_t : (x = position, v = velocity) \in S$**,** which represents a vector of position and velocity, following the dynamics presented above Eq. 4.1. The starting position is $x_0 = -0.5(m)$ at the middle of the hill. The goal is met if the cart is able to reach $x > 0.5(m)$.

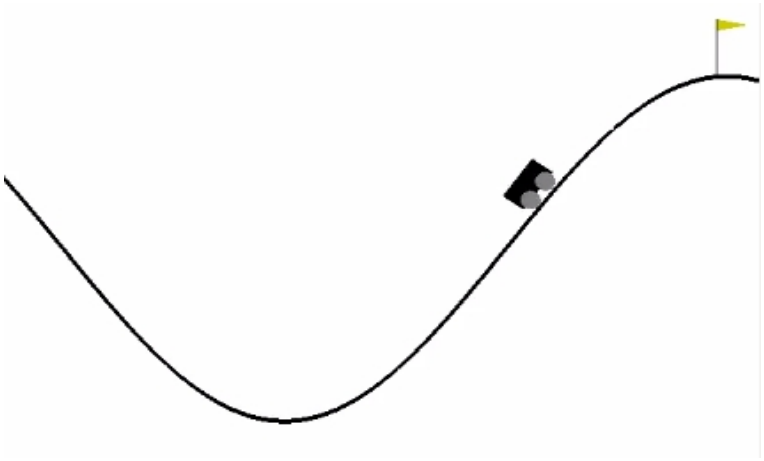


Figure 4.1: Image of the Mountain Car problem, retrieved from the OpenAI gym emulator.

- **A finite set of actions** A**,** three actions, apply left thrust ($a_t = 0 : left$), no action ($a_t = 1 : NoAction$) and right thrust($a_t = 2 : right$).

- **A reward function** $r = R(s_t,a_t,s_{t+1})$**,** the reward is $r = -1$ across all the state-space.

4.1.3 Problem: Cart Pole

Simplified cart pole problem without friction [67] (between cart/pole and cart/track). The angle between the pole and the verical axis is θ and x is the horizontal position of the cart.

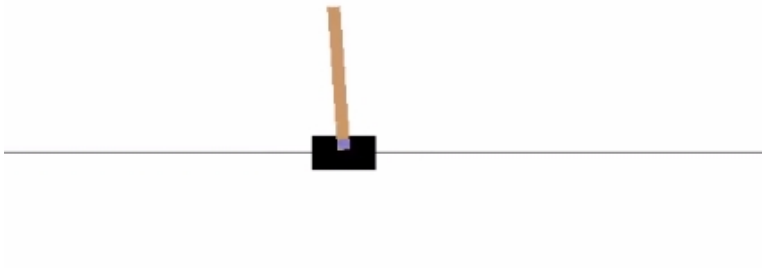


Figure 4.2: Image of the Cart Pole problem, retrieved from the OpenAI gym emulator.

The problem dynamics have the following equations:

$$\begin{aligned} x_{t+1} &= x_t + \tau \dot{x}_t \\ \dot{x}_{t+1} &= \dot{x}_t + \tau \ddot{x}_t \\ \ddot{x}_t &= \frac{F + ml(\dot{\theta}_t^2 sin(\theta_t) - \ddot{\theta}_t cos(\theta t))}{m + M} \\ \theta_{t+1} &= \theta_t + \tau \dot{\theta}_t \\ \dot{\theta}_{t+1} &= \dot{\theta}_t + \tau \ddot{\theta}_t \\ \ddot{\theta}_t &= \frac{gsin(\theta_t) + cos(\theta_t)(\frac{-F - ml\dot{\theta_t}^2 sin(\theta_t)}{m+M})}{l(\frac{4}{3} - \frac{mcos(\theta_t)^2}{m+M})} \end{aligned} \tag{4.2}$$

where $\tau = 0.02(seconds/step)$, the length of the pole $l = 0.5(m)$, $F = \pm 10$ is the magnitude of the force applied by our agent. The mass of the pole is $m = 0.1(kg)$, mass of the cart $m = 1.0(kg)$ and gravity $g = 9.8(ms^{-2})$

- **Environment dynamics** $\mathbf{s}_t : (x_t, \dot{x}_t, \theta_t, \dot{\theta}) \in S$**,** consisting of a state for the agent, following the dynamics presented above Eq. 4.2. The starting position is s_0, a uniform randomized 4-D vector with boundaries for each variable of $[-0.05, 0.05]$. The goal is to balance the pole as long as possible. Each episode stops when $||x_t|| > 2.4(m)$ or $||\theta_t|| > \frac{24}{360}(rad)$
- **A finite set of actions** A**,** two actions, apply left thrust $(a = -1 : F = -10(N))$, or right thrust$(a = 1 : F = 10(N))$.
- **A reward function** $r = R(s_t, a_t, s_{t+1})$**,** in the used model, the reward is $r = 1$ in every state.

4.2 State Reduction Architecture

To test an implementation of the Stacked Auto-Encoder, samples of the maze were retrieved from the dataset $\mathcal{D}$, gathered from the interaction between the game and our learning agent. An example of an episode can be seen Fig. 4.3.

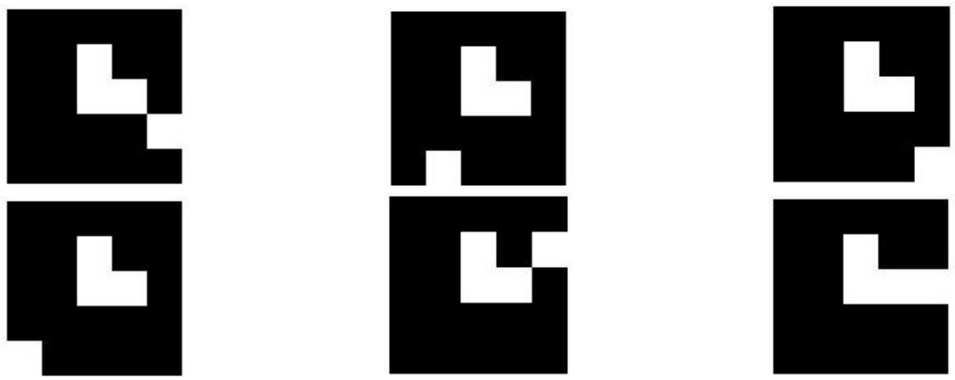


Figure 4.3: Training images of the Stacked Auto-Encoder.

The primary step is to convert each image to a vector by concatenating each column of pixels. Each vector represents a sample and their collection form a training set in its wholesome. Next,

the training set is used to train a Stacked Auto-Encoder. Whereas a shallow auto-encoder would be straightforward to train, the methodology will be followed as described in Sec. 2.5.5 to train a Stacked one. Using the concatenated vector, the training of the first shallow auto encoder is proceeded using the subsequent composition:

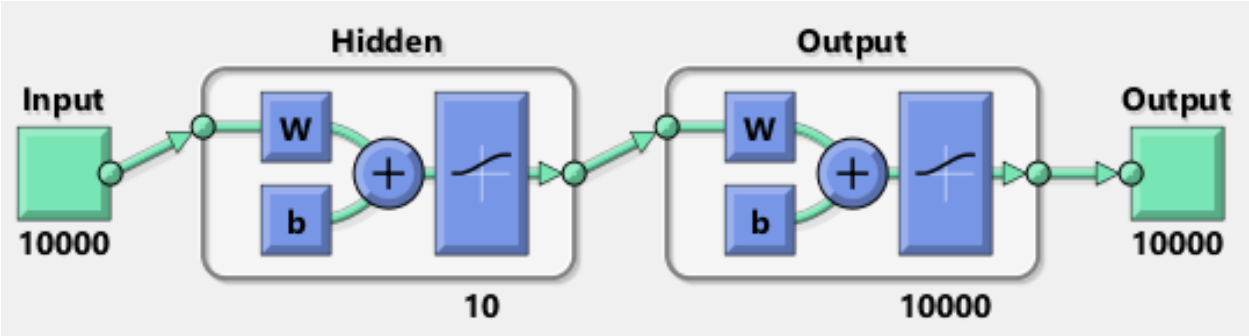


Figure 4.4: First trained shallow encoder from raw images. The input represents a $10000by1$ vector, **W** represents a matrix of weights and **b** represents the bias of each block. Each number bellow a box represents its quantity within each layer.

- **One hidden Layer and one output Layer,** containing 10 hidden units. Each hidden unit is a $\Phi(a) = sigmoid(a) = \frac{1}{(1+e^{-a})}$ non-linearity, since each pixel varies in the interval of [0;1]. The output layer contains 10000 sigmoids .

- **Loss function,** Mean squared error Sec. 2.4.1where $o_{\mathcal{N}_{output,i,j}}$ is the resulting output of a forward-pass, and $y_{i,j}$ represents each pixel of the original image (vector equal to the input).

- **Training criterion and regularizer,** the ten units at the bottleneck (hidden layer) serve as a self-regularizer.

- **Optimization procedure,** SCG and Rprop optimization algorithms, to be tested and compared in Chap. 5.

A set of features are extracted from the input vector, by performing a forward-pass on the encoder (hidden layer), hence converting every image into a ten dimensional vector (this is only possible due to the fairly low variance amongst the dataset). The second shallow auto-econder is trained in a similar manner and described Fig. 4.5:

- **One hidden Layer and one output Layer,** containing 2 hidden units, each hidden unit is a $\Phi(a) = sigmoid(a) = \frac{1}{(1+e^{-a})}$ non-linearity, since each pixel varies in the interval of [0;1]. The output layer comprises 10 sigmoids.

- **Loss function,** mean squared error Sec. 2.4.1 where $o_{\mathcal{N}_{output,i,j}}$ is the resulting output of a forward-pass, and $y_{i,j}$ represents each feature vector generated from the primary auto-encoder.

- **Training criterion and regularizer,** the two units at the bottleneck (hidden layer) serve as a self-regularizer.

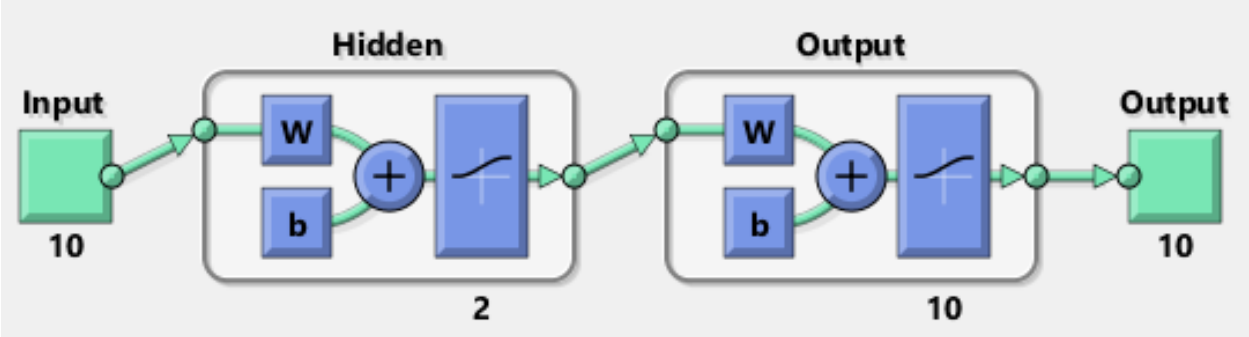


Figure 4.5: Second shallow encoder trained from features.

- **Optimization procedure,** SCG and Rprop optimization algorithm, to be tested and compared in Chap. 5.

This method is commonly referenced in the literature as pre-training. The goal of pre-training is to start at a much lower point in the loss surface, before training the new network end-to-end.

Finally, a Stacked Auto-Encoder is created by stacking both shallow Auto-Encoders according to Sec. 2.5.5 and re-training the whole structure. The final encoder is presented in Fig. 4.6:

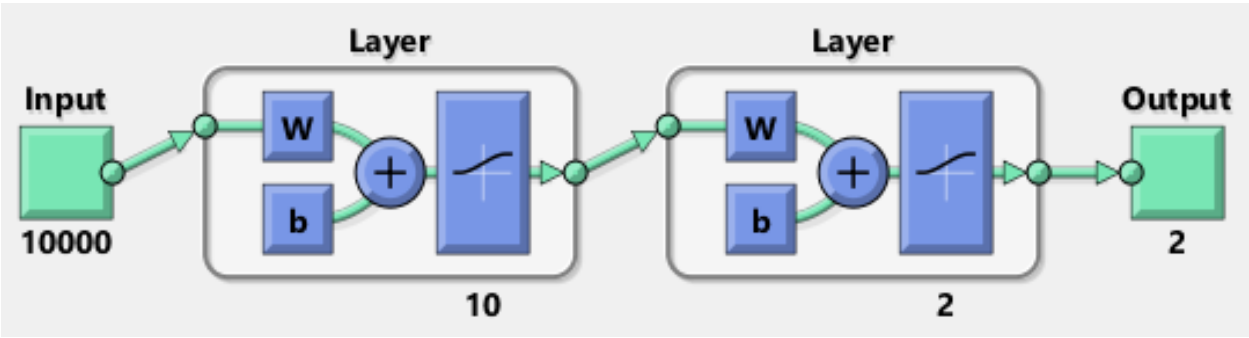


Figure 4.6: Final stacked encoder.

Although we will only use the encoder, the decoder could be used for other purposes. The final trained encoder performs a meaningful coding of our image, enabling a succinct state representation to be used in a Q-learning algorithm.

4.3 Learning Architecture for the Maze Problem

This section aims to embody the proposed algorithm in Sec. 3.8 by conjugating the Stacked encoder and a Feed forward network to estimate Q-values under the maze MDP. The main advantage of using a Stacked-Encoder relies on converting the original dataset D into a feature set D_z by using $F : \mathbf{X} \rightarrow \mathbf{Z}$.

The encoded states produced by the Stacked Auto-Encoder allow for major computational speed, since the structure is expected to iterate thousands of times during the training of our Q-value neural network (3000×12 for convergence and twelve fitted Q iterations). The starting structures were initially thought with [64] in mind.

Furthermore, experimental results in Chapter. 5, from the two structures presented in this section, are used for comparison and further validation/understanding of different valid options. Both of them have advantages and disadvantages. This thesis aims to design similar neural structures (equal amount of weights, biases and non-linearities), to elaborate an impartial review.

4.3.1 First NFQ variant (NFQ1, one Q-value at the output)

The first DNN to approximate the function $Q(s,a;\boldsymbol{\theta})$ considers the case of a single output. Alternatively, this DNN will be refered as NFQ1. The DNN alongisde the properties described bellow, will be used as a supervised learning method within the pattern set Sec. 3.8.

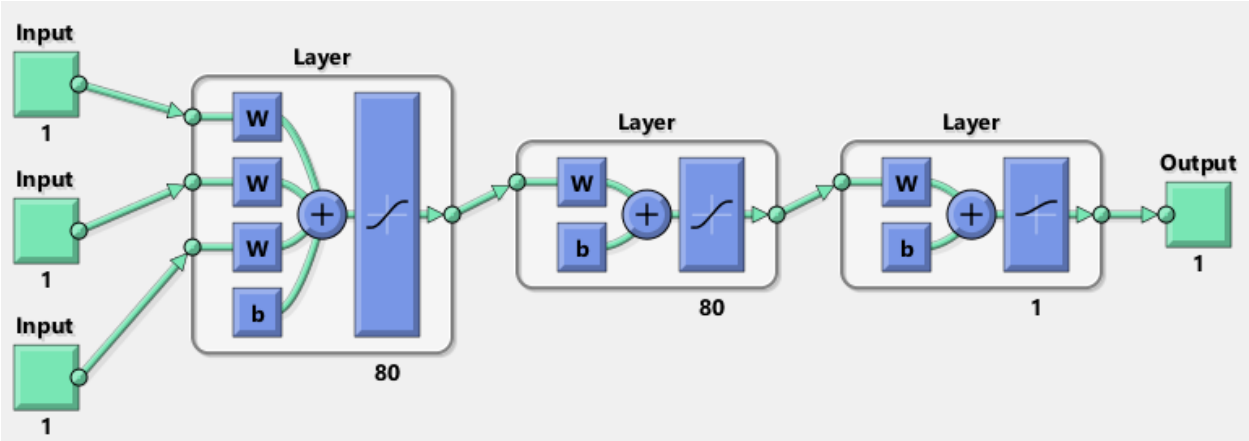


Figure 4.7: Neural Fitted Q Network with states and actions as the input and one Q-value output.

- **3 inputs, 2 hidden Layers and 1 output,** the inputs correspond to the states and actions and the output corresponds to $Q(s,a;\boldsymbol{\theta})$. Each hidden layer has 80 units, and each hidden unit is composed by $\Phi(a) = tanh(a) = \frac{2}{1+e^{-2a}} - 1$ non-linearity. Our output layer has a sigmoid activation, since the rewards are clipped from [0;1].
- **Loss function,** mean squared error Sec. 2.4.1, between $r + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1};\boldsymbol{\theta}_i)$ and the output $Q(s_t,a_t,\boldsymbol{\theta}_i)$.
- **A training criterion and regularizer,** simply the loss function, since the state-space is small.
- **Define an optimization procedure,** RPROP optimization algorithm explained in detail Sec. 2.4.4.

4.3.2 Second NFQ variant (NFQ2, one Q-value per action at the output)

This second DNN version is similar to the DQN fully connected layers [8]. Alternatively this variant will be referenced as NFQ2 alongside the bellow properties. Both architectures are compared within the same environmental setup in Sec. 5.

- **2 inputs, 2 hidden Layers and 4 outputs,** the inputs correspond to the states, and actions correspond to four outputs $Q(s,a;\boldsymbol{\theta})$. Each hidden layer has 80 units, each hidden

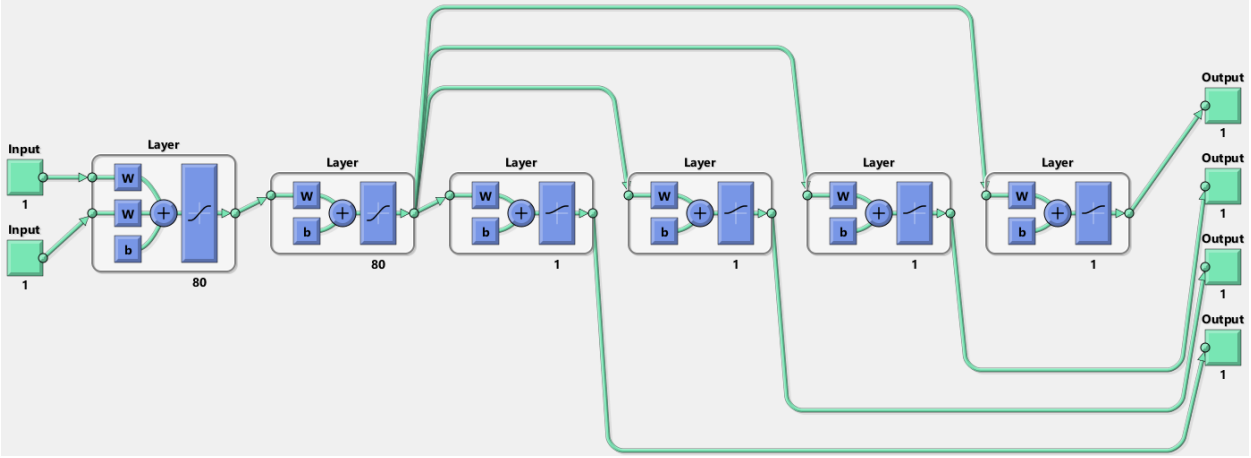


Figure 4.8: Neural Fitted Q Network with states as the input and one Q-value output per action.

unit represents a $\Phi(a) = tanh(a) = \frac{2}{1+e^{-2a}} - 1$ non-linearity. Our output layer has four sigmoid activations, since the rewards are clipped from [0;1].

- **Loss function,** mean squared error Sec. 2.4.1 between $r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}, \boldsymbol{\theta}_i)$ and our output $Q(s_t, a_t, \boldsymbol{\theta}_i)$.
- **A training criterion and regularizer,** simply the loss function, since the state-space is small.
- **Define an optimization procedure,** RPROP optimization algorithm explained in detail Sec. 2.4.4.

4.4 Learning Architecture for the Mountain Car/Cart Pole Problem

Changes to the original Q-learning architecture were inspired by the Google's DQN and briefly approached in Sec. 3.9. The previous architectures comprise an alternative reinforcement learning framework requiring higher computational resources. Re-learning a Q-function is demanding and not particular efficient. Considering batch-learning, specifically recent optimization procedures as ADAM requiring low-memory usage, unveil a possible solution. However, batch-learning has an intricacy, specifically in Reinforcement Learning.

One of the problem relates to over-fitting the Q-function approximation into a biased state-space, thus compromising gravely the agent's performance. For example, if a batch sample of our $\mathcal{D}$(replay memory) corresponds to a small sub-set of the whole state space (thus a poor statistical representation of $\mathcal{D}$) full training until convergence would highly corrupt every other state information, due to the inherent fully connected nature of a DNN. To avoid this fact, mini-batches are taken according to a uniform distribution from $\mathcal{D}$, and a single training step is performed, back-propagating the gradient of the loss function only once across the network.

The second problem has a similar cause with a different effect. By updating our Q-function every mini-batch training step, the targets for the Q-learning step also change. For example, if the

network is trained successfully in a poor statistical representation, the targets for a different state space moderately change. It does not disable learning, but highly oscillates the Q-function compromising our well defined exploration/exploitation strategy. The oscillation will overly explore/exploit according to the acquired biased mini-batches and greatly influence either our exploration or exploitation. The previous influence relates to: either by gathering too many rewards in the biased batch and greatly exploit a particular sub-set; or by gathering too many negative rewards and explore a completely different state space, even though there could be an optimal solution around the biased state space.

To increase robustness in learning (less oscillations), in addition to randomly sample mini-batches, a fixed target-network is created. Since it is fixed, each training step does not affect targets when training the whole model. The previous is accomplished since target updates are made *offline*, with the downside of turning the learning procedure slower. A description of the partial DQN structure is elaborated below, to be tested in the following chapter.

- **N inputs, 2 hidden Layers and N outputs,** the inputs correspond to the states values for each problem and one output $Q(s,a;\boldsymbol{\theta})$ per action. Each hidden layer has M units, each unit has a $\Phi(a) = ReLU(a)$. Outputs are linear activations, once rewards are no longer clipped.
- **Loss function,** mean squared error for each mini-batch Sec. 2.4.1 between our output $Q(s_t,a_t,\theta_i)$ and $r + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1};\boldsymbol{\theta}_{i-1})$.
- **Training criterion and regularizer,** coupled with a L2 weight decay $\lambda = 0.001$.
- **Optimization procedure,** ADAM [40] as a faster method than RPOP, used in DQN Sec. 3.9. The ADAM hyper-parameters used were: $\alpha = 0.001$; $\beta_1 = 0.9$; $\beta_2 = .999$; $\epsilon = 1 \cdot 10^{-}8$.
- **Fixed Target-Network,** a replica of weights and biases of the current network every N iterations.

5| Experimental Results

Contents

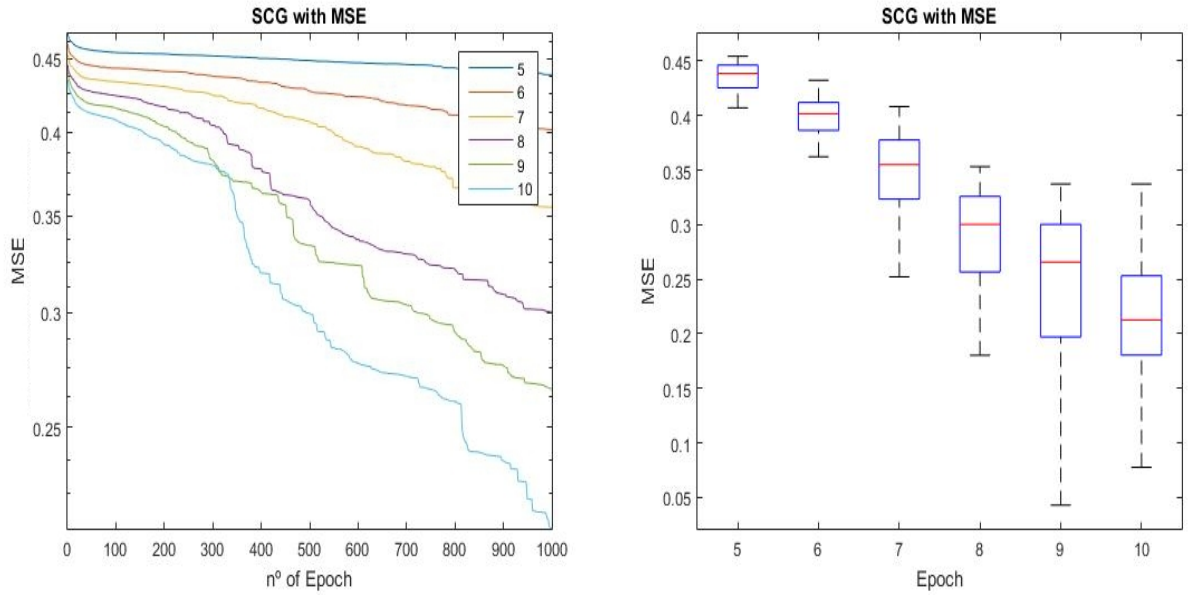
INFERENCE about the performance and validity of proposed architectures is conducted with a set of experiments described in this chapter. Each experiment is throughly described with a disclosed reasoning on its purpose. Following the directives and path we choose to pursue, experiments are conducted with an emphasis on handling the environment and our Q-learning function on a data efficient, flexible, robust and feasible manner.

5.1 State Reduction Framework

This section aims to validate our architectures performing state reduction at Sec. 4.2. A set of features is extracted from the input vector, by doing a forward-pass on the chosen encoder (architecture presenting the lowest absolute loss), hence converting every image into a lower dimensional vector. The second shallow auto-encoder is trained in a similar manner.

This method is commonly referenced in the literature as pre-training. Finally a Stacked Auto-Encoder is created by stacking both shallow encoders according to Sec. 2.5.5 and re-training the whole structure.

Comprehensive tests varying the number of neurons were performed. A dataset of experiences was provided (simulating dataset $\mathcal{D}$) with one hundred images from different frames of the maze shown in Fig. 4.3. In sum, the tests verify better designs, leading to the selection of the final architecture. Based on this results, the final architecture will be coupled with the NFQ for Deep Reinforcement Learning as stated in Section 4.3.



(a) Scaled Conjugate Gradient Descent with MSE. (b) Corresponding Boxplot of the final iteration.

Figure 5.1: Tests varying the number of neurons.

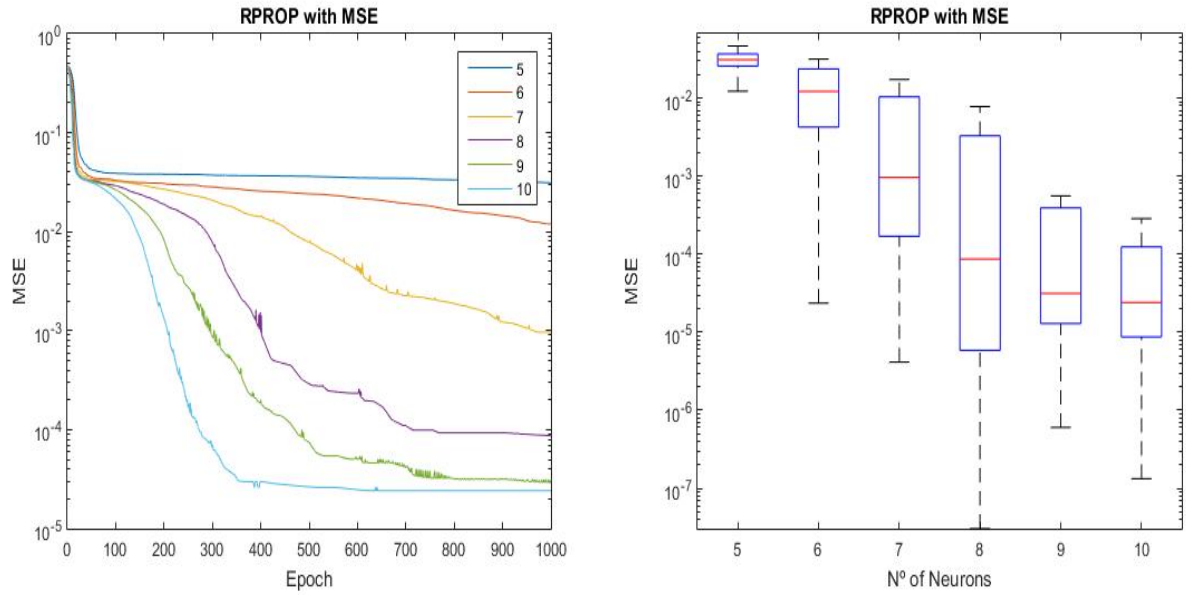
Concerning the optimization, the default parameters for each optimization procedures were used. For SCG, the parameter to determine the change in weight for second derivative approximation is $\sigma = 5 \cdot 10^{-6}$ and the parameter for regulating the indefiniteness of the Hessian is $\lambda = 5 \cdot 10^{-7}$. For RPROP the learning rate is $\alpha = 0.01$, the increment to weight change $\Delta_{inc} = 1.2$, the decrement is $\Delta_{dec} = 0.5$, the initial weight change is $\Delta_0 = 0.07$ and the maximum weight change $\Delta_{max} = 50$.

Auto-Encoder Performance Measurements vs. Different Number of Neurons at the bottleneck

The number of neurons at the bottleneck have a strong impact in the reconstruction error. Each neuron increases the complexity of our model by adding more degrees of freedom to the reconstruction. Many neurons could be added to the bottleneck (middle low dimensional hidden layer) as in the case of an over-complete auto-encoder, requiring increased computational and time efforts. Approaching the Curse of Dimensionality [5] problematic, a reduced number of neurons at the bottleneck is desirable. Starting with only five neurons at the first shallow auto-encoder, which will then be increased up to ten neurons on the pre-training phase. In order to have statistic relevance within these experiments, weights are randomly initialized. One hundred full trainings were preformed until convergence (low absolute value of the training criterion) or with a maximum of one thousand epochs (forward and backward-propagation sweep), to reveal robustness in the optimization procedures.

Increasing the number of neurons has a strong impact in the performance measured (as seen in Fig 5.1 and Fig 5.2). Adding additional neurons give our model extra degrees of freedom, capturing the data distribution accurately.

In the following section, a comparison between optimization procedures is conducted.



(a) RPROP with Mean Square Error.

(b) Corresponding Boxplot of the final iteration.

Figure 5.2: Tests varying the number of neurons.

Auto-Encoder Performance Measurements vs. Training Procedure

In this section, a comparison between both training procedures used, RPROP and SCG, was elaborated, when applied to the ten neuron shallow Auto-Encoder, which had the lowest reconstructions errors on the previous section.

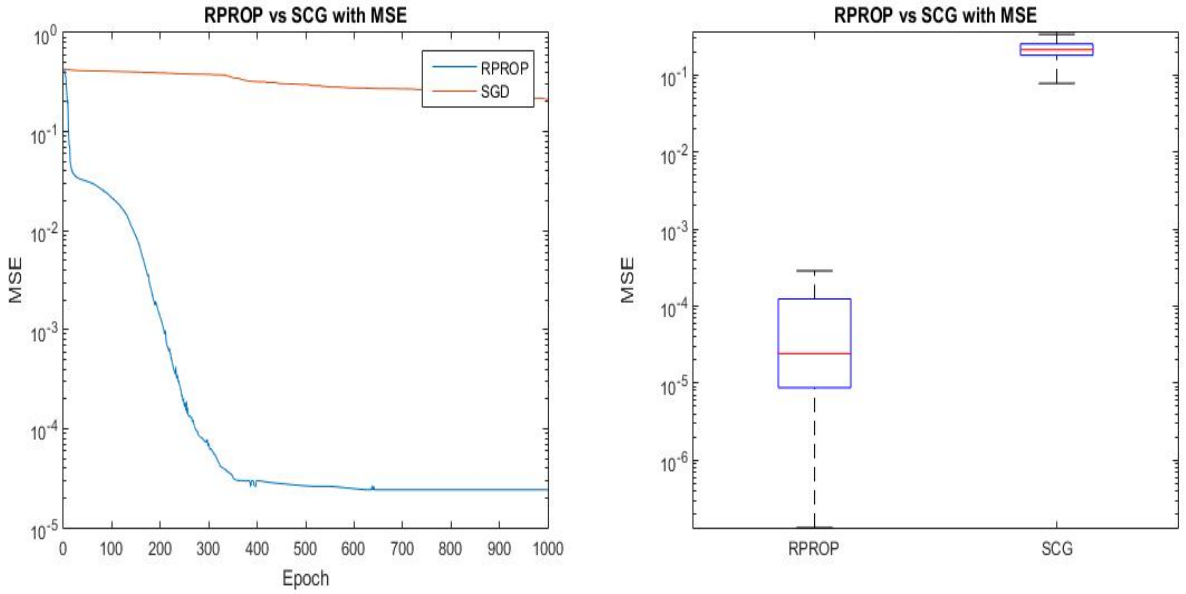
As illustrated in Fig 5.3 and stated at [39], the convergence is accelerated by using RPROP. A limit of one thousand epochs was kept, since most RPROP tests converged below this number.

Furthermore, the results obtained here, corroborate [68], in which was stated that RPROP is superior to SCG (in achieving a small MSE error amongst the training data set), when the objective function contains discontinuities or the number of patterns is small.

Stacked Auto-Encoder Reasoning

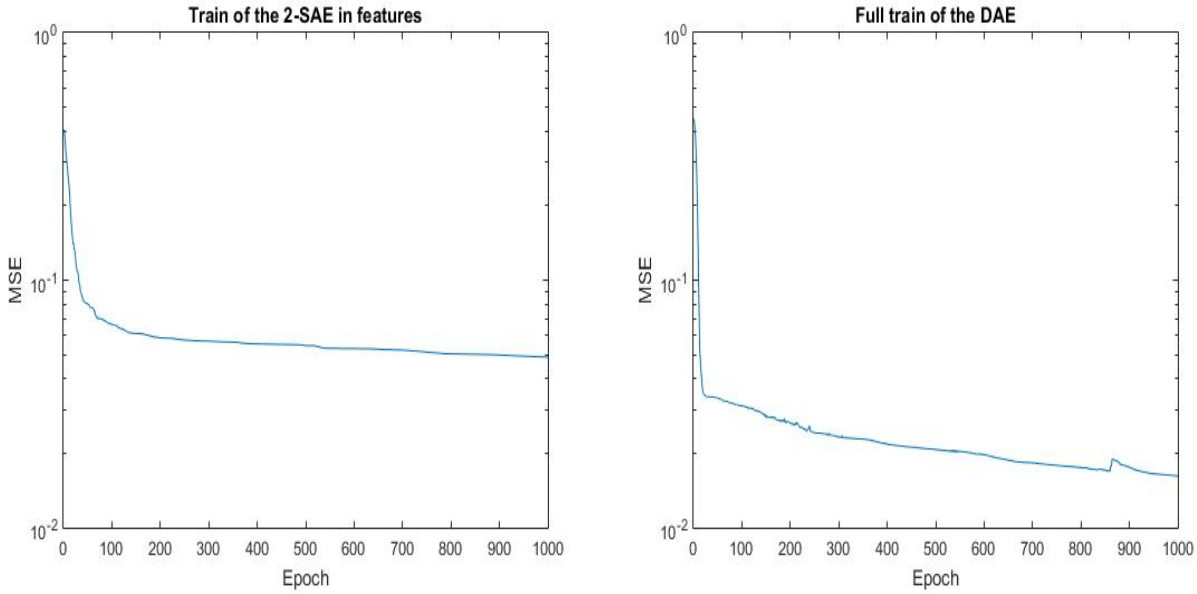
It was mathematically demonstrated [30] and empirically put in practice with [3], that a properly trained Stacked auto-encoder compresses deeper information. Deeper architectures are indeed exponentially more prolific in linear regions [31] than shallow ones. As stated before, a short internal representation of each state is desirable in order to increase our Q-value approximation performance. Since our test bed environment is rather simple Sec. 4.1 the hypothesis of a two neuron seems sound (as an analogy of (x,y) coordinates to describe the position of our agent). A stacked auto-encoder is obtained according to Sec. 2.5.5. A second shallow auto-encoder was trained with the features (forward-pass on the input) generated by the first auto-encoder Fig. 5.4a chosen in the previous sections and the final train of the Stacked Auto-Encoder was obtained Fig. 5.4b in order to guarantee a meaningful two-dimensional code.

Both the feature training on the second auto-encoder and the full Stacked Auto-Encoder training revealed satisfying results when comparing Fig. 5.6 to the shallow auto-encoder . From



(a) RPROP vs SCG with MSE. (b) Corresponding Boxplot of the final iteration.

Figure 5.3: Comparing different optimization procedures.



(a) Two neuron Auto-Encoder trained on features. (b) Final train on the Stacked Auto-Encoder.

Figure 5.4: Pre-Training and Full Training of the SAE. The ten neuron AE is used to convert the observations into features by using the resulted encoder. Afterwards, a two-neuron AE is trained in features and full training is performed by copying the weights of the two-neuron AE into the hidden layer of the SAE. Finally full training is executed and the results shown.

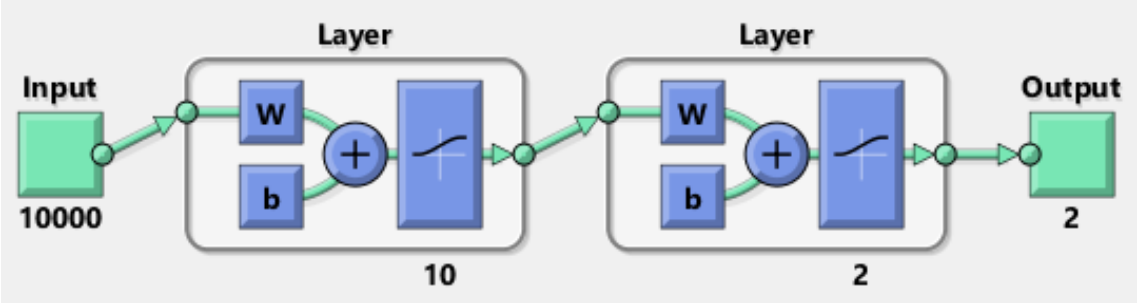
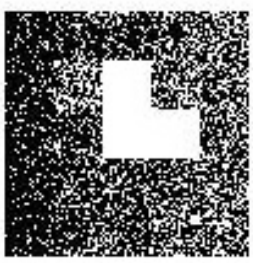
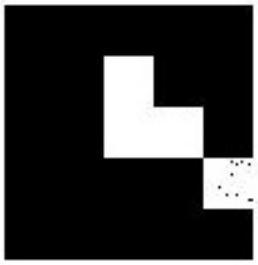


Figure 5.5: Deep Encoder used to code our input into a two dimensional vector. Each point represents a product, the input is represent by a $10000by1$ vector, $\mathbf{W}$ represents a matrix of weights, $\boldsymbol{b}$ represents the bias of each block (neuron) and each number bellow a box represents its quantity



(a) Image retrieved from a 2-neuron bottleneck Auto-Encoder.



(b) Image retrieved from a 2-neuron bottleneck Stacked Auto-Encoder.

Figure 5.6: Comparison of the same image retrieved after a forward-pass.

Fig 5.4 the necessity of a Stacked Auto-Encoder was acknowledged in order to achieve a meaningful two dimensional code of every image Fig. 5.6 within dataset $\mathcal{D}$.

5.2 Deep Reinforcement Learning results

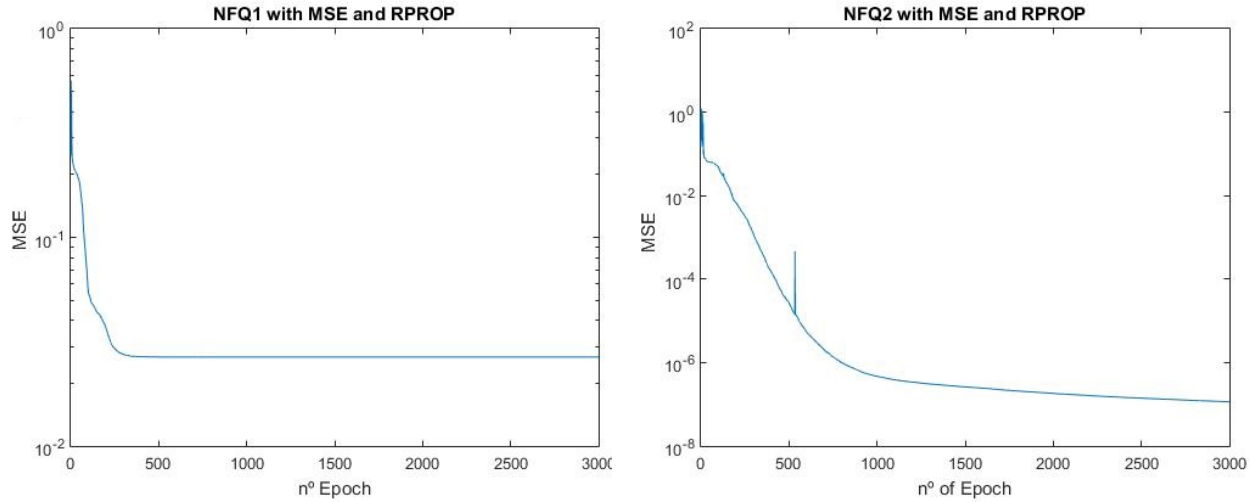
One of the main concerns when applying Deep Learning as our function approximation is the impact of a bad initialization has on the Q-function itself. However, in DNN's there is an exponential number of good local minima due to the existence of saddle points [69], close to the global minimum.

A concern rises from clipping the rewards between [0,1], turning impactful rewards indistinguishable from average ones. Also, if the back-propagated error by the network is large at some point, the policy diverges from the optimal policy.

Another issue is related to a known Q-value oscillation (thus policy) caused by a biased data set $\mathcal{D}$. An imbalanced set of transitions across the state space will lead to suboptimal policies or overwhelm the impact of some transitions across the state space in the training phase. This is partially solved by a sampling strategy (as for example an sampling heuristic from the goal [64]) or a better exploration algorithm.

The generalization of Q-values for different regions of the state space that were not initially intended, might also be problematic, since the network is fully connected.

Therefore, an architecture flexible enough is required to deal with the above mentioned prob-



(a) Training error on Fig. 4.7 architecture. (b) Training error on Fig. 4.8 architecture.

Figure 5.7: Comparison between NFQ1 and NFQ2.

lems. Such a design has to keep a low error across every iteration in order to converge to an optimal policy. Furthermore, experimental results from the two structures presented are used for comparison and further validation and understanding of different valid options.

5.2.1 NFQ Tests

Each episode will be the result of the interaction between the game and our learning agent until we stumble on either a wall or the goal. Making the goal protected by walls will ensure proper testing and validation of either our exploration method, unsupervised learning methods with the Stacked Auto-Encoder and supervised learning with the Neural Fitted Q-Network (NFQ). Both structures were announced in Sec. 4.3.1 and Sec. 4.3.2, and tested on the same benchmark $\mathcal{D}$.

5.2.2 Final notes regarding the NFQ approach

Both architectures converged to an optimal policy, leading from the starting point to the goal. The last structure (NFQ2 Sec. 4.3.2) clearly surpassed the first one (NFQ1 Sec. 4.3.1) in a performance perspective, illustrated by Fig 5.8 and Fig 5.7.

Furthermore, the variance in the first structure NFQ1 is also higher when comparing to the variance of the second structure NFQ2, pointing towards a more stable version of the non-linear Q-value approximation. Since finding $max_a Q(s_{t+1},a_{t+1})$ for every Q-value update requires a linear amount of cycles proportional to the number of actions for the first structure NFQ1, there are far more computational costs to be considered.

Nevertheless, the first Q-approximation is fairly interesting, once it does not limit the number of actions. If the number of actions was unknown or the same structure needed to be used for different problems, the aforementioned architecture would be the only solution. However, the fitted Q-algorithm has an inherent disadvantage: re-learning the Q-function, each time more information is gathered, becomes costly. In the next section, a highly efficient method to perform Deep Reinforcement Learning with batch-learning on an increasing replay memory is addressed.

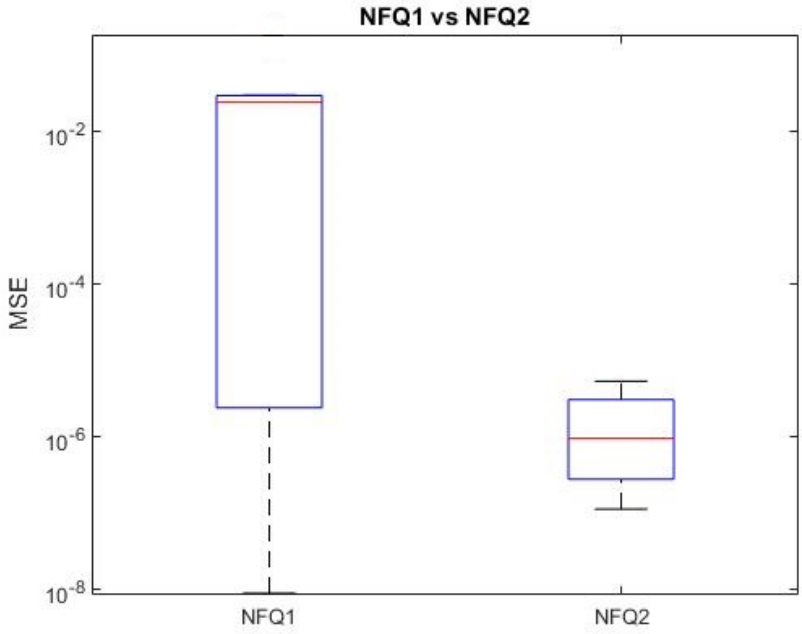


Figure 5.8: Comparison between NFQ designs with a Boxplot for the last NFQ iteration in 100 tests.

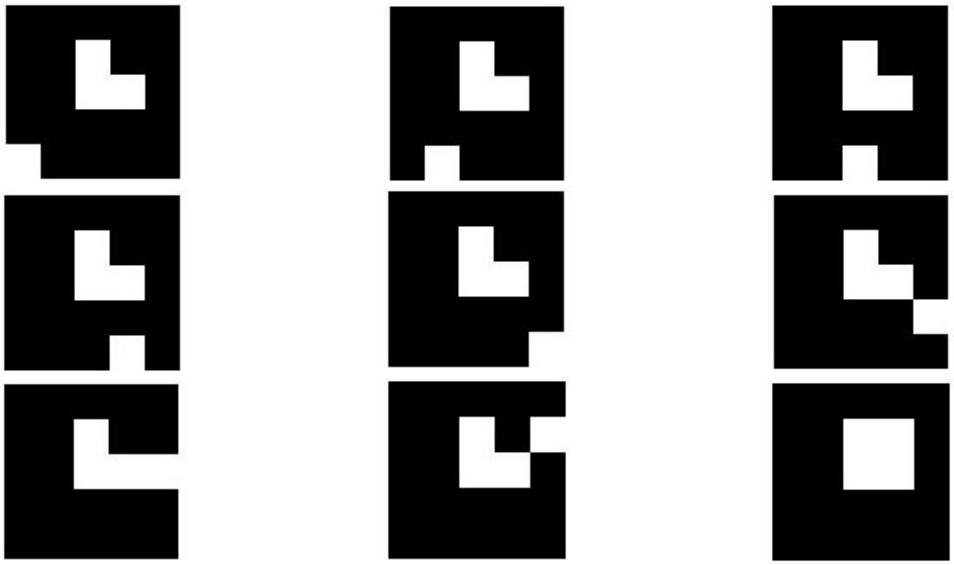


Figure 5.9: Sequence of images to the goal, under the optimal policy, starting state at left top and reaching goal at bottom right, after the NFQ converged.

5.3 Partial DQN Experiments

To further test the hypothesis, a deep feed-forward network is able to perform supervised learning on a larger dataset, for Q-learning purposes, several classic control problems Sec. 3.2 were chosen.

Since a certain precision on Q values has been reached in the former experiments, by both NFQ1 and NFQ2, an architecture of similar dimensions was employed. Moreover, compact states are used (states from the emulator) for this experiment, instead of using the whole perceptual information (frames).

Although batches of frames could have been used as a higher-dimensional input representation, the Stacked Auto-Encoders tests were considered to be successful and the employment in this setting would rise the computational resources much further. Nevertheless, the effectiveness of Stacked Auto-Encoders and variations has been recently researched and employed with a high degree of success [28].

The reasoning behind this experiment is to verify how the *Q-approximation* (fully connected network) stands on a much larger state space. Leveraging the former proposed architecture, a simplified version of DQN was elaborated, approximately corresponding to the final fully connected layers of the Google's DQN [8] (double hidden layer of 256 units). In addition, the amount of hidden units was lowered to be closer to our former architectures (one hundred units each hidden layer) shown in Sec. 4.4. The first convolutional layers (of the DQN architecture) aimed to compress pre-processed frames into meaningful features, serving a similar purpose of a *Stacked Auto-encoder*, but with a much higher efficiency in the case of images.

In this particular set-up, all figures were obtained with the same hyper-parameters, under the same computational tests. The choice of hyper-parameters is partially learning independent, meaning the algorithm is robust to changes, once it will only hinder the speed/efficiency of learning. Nevertheless, the most impactful changes are enumerated as follows:

- **Replay Memory $\mathcal{D}$,** the repository of gathered tuples (s_t,a_t,r_{t+1},s_{t+1}) should contain a large portion of the state-space represented. If the Replay Memory is small, learning is infeasible, five hundred thousand samples of capacity were used, a similar number to the DQN structure.

- **Target-update Network rate,** defines the rate that the parameters of our current Q-network are transferred to the target-network. The reasoning behind the target network was discussed in Sec. 3.9, one thousand steps were chosen between parameter transfer to assure a smooth learning.

- **Mini-batch size $b \in \mathcal{D}$,** containing a uniform sample of tuples from $\mathcal{D}$. A batch of eight samples was chosen due to computational restrictions. A larger batch translates in more computational operations within each training step.

5.3.1 Simplified DQN applied to the Mountain Car Problem

After two thousand episodes, the Q-agent obtained an average rewards of -332 per last 100 episodes Fig. 5.10. According to [66] the Mountain Car problem is considered solved when an agent achieves -110 rewards per last 100 episodes. Despite the agent's inability to solve the task in two thousand episodes, the result was considered to be a reasonable starting point of a free-model architecture. Nevertheless, further tests and more computational resources could be spent to guarantee resolvability.

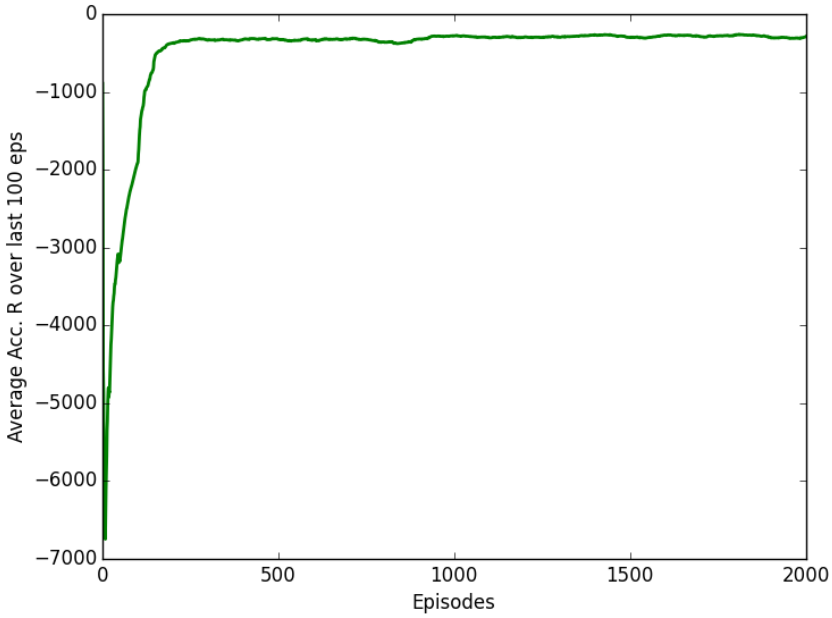


Figure 5.10: Average Rewards over every 100 episodes on the MC problem

The cost at each training step was gathered for further inference of our agents performance Fig. 5.11. The median and minimum cost within each episode are considerably low (mainly due to the impact ADAM has on accelerating propagation) and the maximum cost was highlighted at red. The red spikes reflect each cost immediately before the transfer of parameters between the current and target-network, when predictions differ the most. A slight decrease in these jumps is expected, since the Q-network parameters should change in a slower rate as learning progresses.

The Q-values at the beginning of each episode reflect the agent's initial policy around the starting state. From Fig. 5.12 there is no clear choice from our agent, corroborating the belief that an optimal policy was not found (further propagation should be performed to guarantee convergence, since $Q_i = Q^*$ when $i \to \infty$, but the former would also be no guarantee since the approximation error would hinder this convergence). Nevertheless, a mean value of -68 for the three Q-values was reached, forcing our agent to balance from the bottom of the two-sided hill.

An average of the last one hundred Values (max operation of the output after a forward pass) at the end of each episode was recorded for further inspection Fig. 5.13. At the beginning, around the first one hundred episodes, our Q-values are further bellow -1. This correlates with the fact of our agent first 5-20 episodes end without the goal being met. After a certain number of concrete

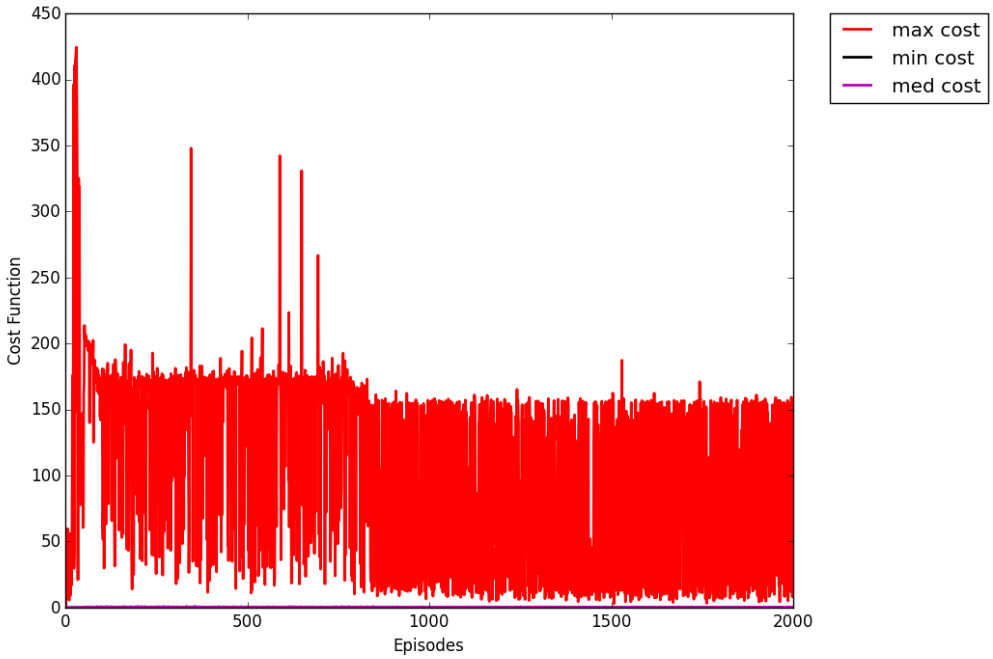


Figure 5.11: Cost over each episode on the MC problem

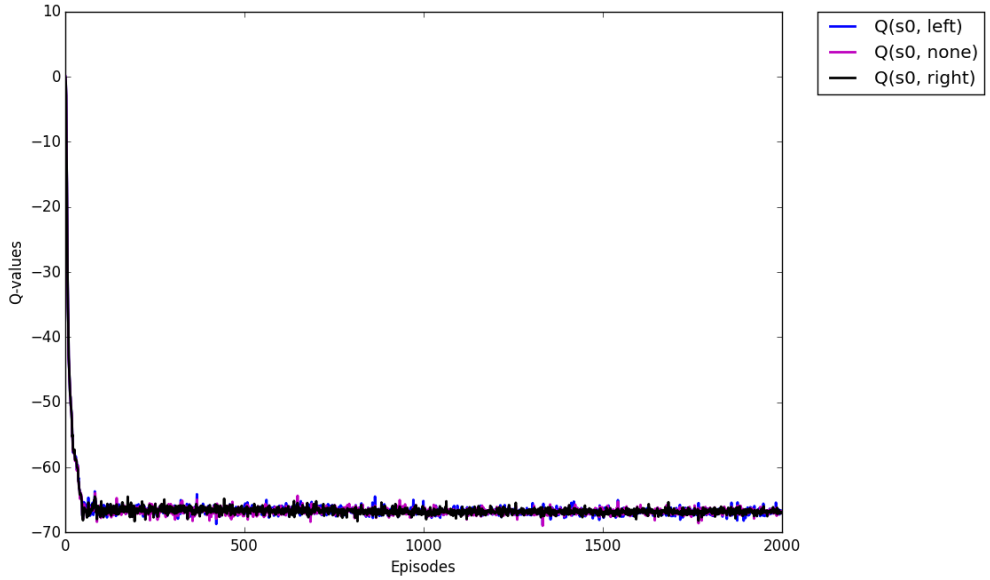


Figure 5.12: Q-values at the start of each episode on the MC problem

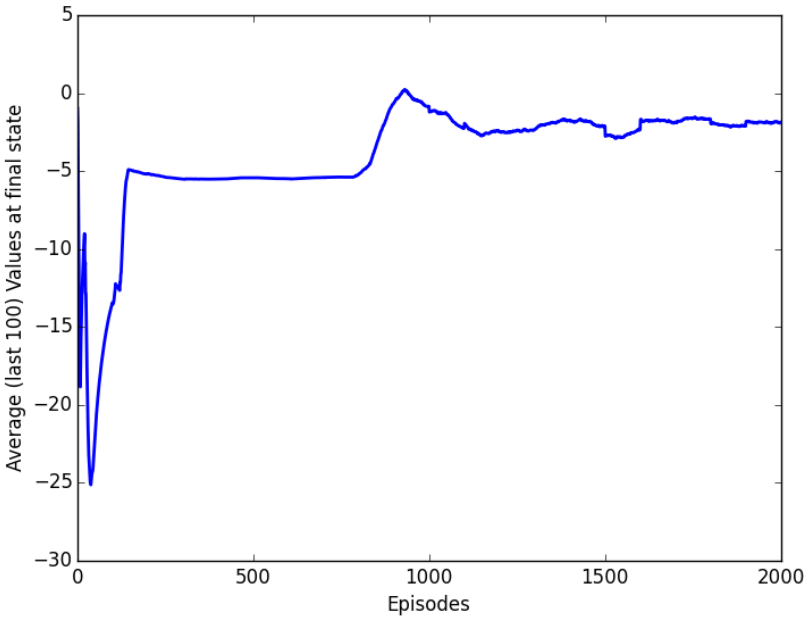


Figure 5.13: Value function of the last episode on the MC problem

episodes ending with a goal met, our Network prediction approaches to -1. The former leads to the conclusion that the agent learns the optimal policy around the goal and correctly propagates change through the rest of the state-space.

5.3.2 Simplified DQN applied to the Cart Problem

As a generalization test of the former Q-learning framework, a second task: Cart Pole Problem Sec. 4.1.3 was chosen to access the validity of the approach. According to [70], the problem is considered "solved" after achieving an average reward of 195. The code and hyper-parameters are equal to the Mountain Car task and the code is exactly the same, except in the definition of the OpenAI gym environment and the target-network transfer rate. Since each episode has a random starting point (initial angle), early episodes ended with an average of 10-20 steps. A target network rate of one thousand would make for an extremely slow learning. The rate was changed to ten to achieve a higher average reward earlier.

One noticeable change when comparing Fig. 5.14 to the former problem is the increased oscillation of our agent's performance. As mentioned in Sec. 3.9 this oscillation is due to our stochastic optimization procedure, considering different state-spaces within each step. Nevertheless, the trend (considering a mean of the average) gathered rewards per episode was increasing, achieving an average of 311 rewards after one thousand episodes. The cost within each episode has the same interpretation of the Mountain Car Problem, but the decrease in maximum cost is more abrupt, this fact is also related with the reduced target-network update. Q-values from both actions are gathered and presented in Fig. 5.16. The starting state in each episode is not static as in the previous problem. The Cart-Pole spawns from a random position, generating different starting dynamics each time. Nevertheless, our Q-network predicts an average of two hundred steps until

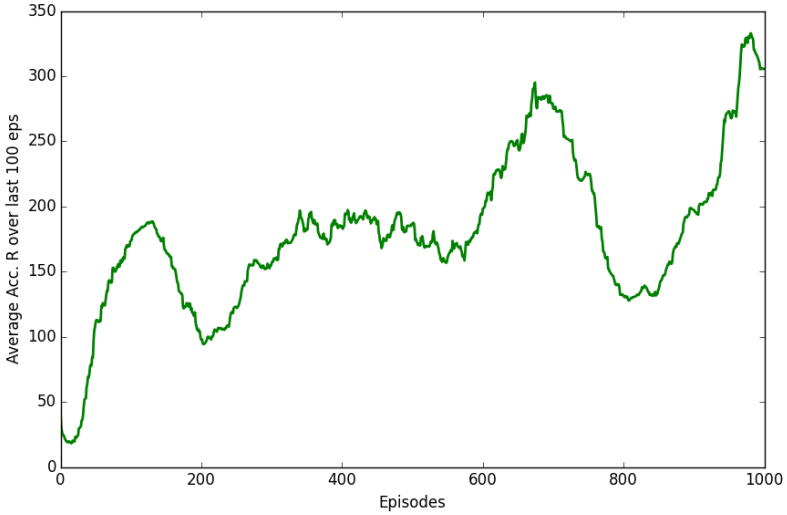


Figure 5.14: Average Rewards over every 100 episodes on the CP problem

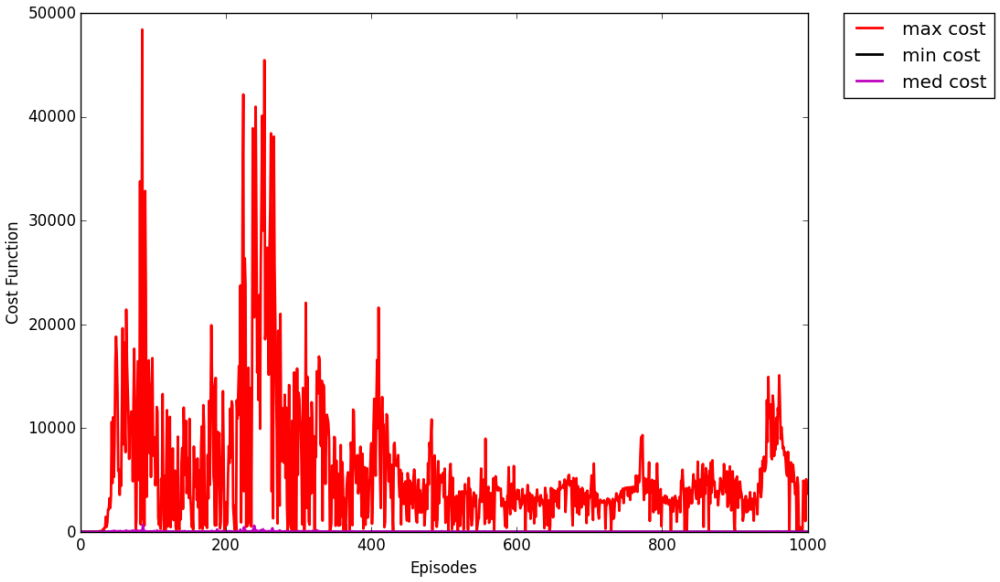


Figure 5.15: Cost over each episode on the CP problem

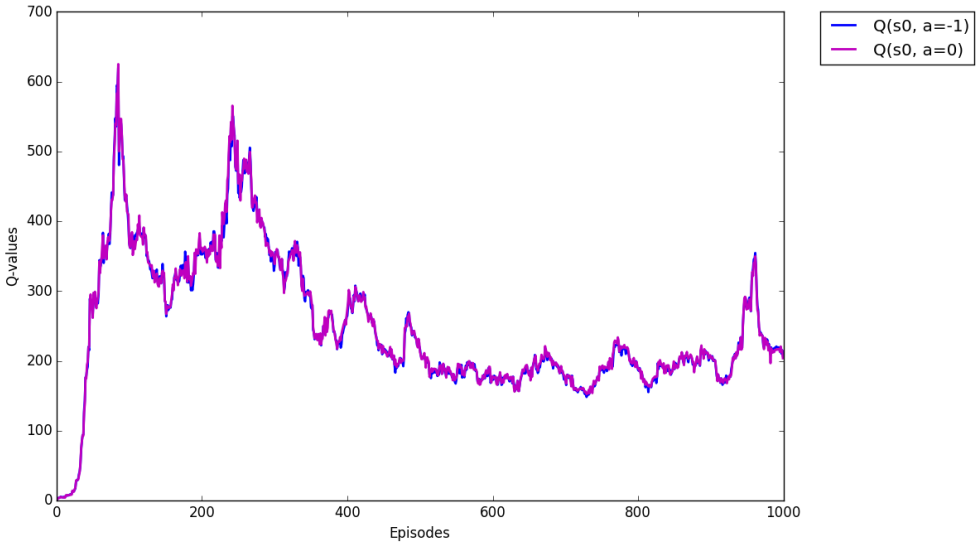


Figure 5.16: Q-values at the start of each episode on the CP problem

failure, which is still above the "solved" flag consideration.

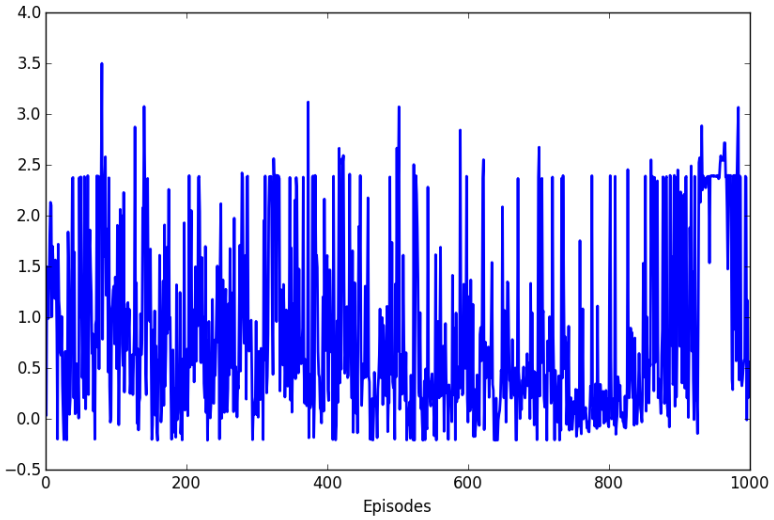


Figure 5.17: Value function of the last episode on the CP problem

From Fig. 5.17 the value function varies from 0 to 2.5 (the correct value should be around 1). The oscillation is also due to the update-rate and the fact the optimization procedure derives from a stochastic sampling.

6| Conclusions

The aim of this thesis was to explore novel reinforcement learning methodologies for general purpose tasks alongside state-of-the-art machine learning neural networks, also re-branded as Deep Learning. Validation for the previous was explored with experimental results obtained in Chapter 5. The impact of a larger bottleneck on shallow auto-encoders was verified with experimental evidence and became a starting point on the state reduction hypothesis. Varying the optimization procedure aided the selection of better practices and achieved a milestone in this work. A Stacked Auto-Encoder was trained to meet our expectations towards a short, yet meaningful code that could be used as an input of a Decoder neural network to reconstruct the initial images. For our reinforcement learning framework, we started with the Neural Fitted Q-iteration algorithm 3.8 and built several Deep Feed Forward networks to perform supervised learning on a data set $\mathcal{D}$.

Two main architectures are studied, one with states and actions as inputs and a Q-value at the output, and another configuration having the coded states as input and one Q-value per action at the output. Tests were performed under the same benchmark $\mathcal{D}$ and results drawn, leading to the latter approach as possibly the best.

Furthermore, a Q-network based in [8] was implemented and tested on two classic control problems: Mountain Car and Cart Pole. Two main components to improve performance were added: a target-network to prevent oscillation; and a mini-batch learning step to further increase efficiency and generalization on these types of methods (instead of re-learning a Q-function within each episode). The results proved to be model-free and quite effective, possibly leading to further experiments and different configurations.

This thesis approached the challenge of designing methods to improve from classic reinforcement learning to "Deep" reinforcement learning by allowing different state representations and function approximations. Unsupervised learning methods were used on high dimensional inputs, shifting the problem from designing relevant features to satisfiable general purpose networks. Several architectures to approximate Q-function and consequently Q-iteration algorithms were designed, specifically to apply supervised learning alongside a replay memory $\mathcal{D}$.

Future work will include mainly the incorporation of memory to deal with long term dependencies in a reinforcement learning framework as a whole. One of the solutions might be introducing a LSTM (Long Short Term Memory neural network) already being very popular in translation. An attention window mechanism could also be incorporated to look at only a part of the high dimensional input and process sequences of actions-states instead of just one at a time. The second solution could be using a LSTM as an approximation of the Q-value function, which could interpret long term dependencies and enhance correlations within sequences of states, creating a deep interpretation (a history rich one).

Bibliography

[1] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2015.

[2] David E. Rumelhart and James L. McClelland. *Parallel distributed processing: explorations in the microstructure of cognition.*, volume 1. MIT Press, Reading, MA, 1986.

[3] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

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

[5] David L Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS Math Challenges Lecture*, pages 1–32, 2000.

[6] Timothy F Cootes, Gareth J Edwards, and Christopher J Taylor. Active appearance models. In *Computer Vision—ECCV'98*, pages 484–498. Springer, 1998.

[7] GE Hinton. Reducing the dimensionality of data with neural. *IEEE Trans. Microw. Theory Tech*, 47:2075, 1999.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[9] Henry W Lin and Max Tegmark. Why does deep and cheap learning work so well? *arXiv preprint arXiv:1608.08225*, 2016.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[11] Dan Cirecsan, Ueli Meier, Jonathan Masci, and Jurgen Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, 2012.

[12] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[13] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[14] Ian J Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. *arXiv preprint arXiv:1312.6082*, 2013.

[15] Minh-Thang Luong, Ilya Sutskever, Quoc V Le, Oriol Vinyals, and Wojciech Zaremba. Addressing the rare word problem in neural machine translation. *arXiv preprint arXiv:1410.8206*, 2014.

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

[17] Xin Xu, Lei Zuo, and Zhenhua Huang. Reinforcement learning algorithms with function approximation: Recent advances and applications. *Information Sciences*, 261:1–31, 2014.

[18] Larry D Pyeatt, Adele E Howe, et al. Decision tree function approximation in reinforcement learning. In *Proceedings of the third international symposium on adaptive systems: evolutionary computation and probabilistic graphical models*, volume 2, pages 70–77, 2001.

[19] Shimon Whiteson and Peter Stone. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7(May):877–917, 2006.

[20] Hao Yi Ong. Value function approximation via low-rank models. *arXiv preprint arXiv:1509.00061*, 2015.

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

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

[23] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*, 2016.

[24] Ionel-Alexandru Hosu and Traian Rebedea. Playing atari games with deep reinforcement learning and human checkpoint replay. *arXiv preprint arXiv:1607.05077*, 2016.

[25] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*, 2015.

[26] Jan Mattner, Sascha Lange, and Martin Riedmiller. Learn to swing up and balance a real pole based on raw visual input data. In *International Conference on Neural Information Processing*, pages 126–133. Springer, 2012.

[27] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[28] Eder Santana and George Hotz. Learning a driving simulator. *arXiv preprint arXiv:1608.01230*, 2016.

[29] José Raúl Machado Fernandez, Jesús de la Concepción Bacallao Vidal, Anilesh Dey, DK Bhattacha, DN Tibarewala, Nilanjan Dey, Amira S Ashour, Dac-Nhuong Le, Evgeniya Gospodinova, Mitko Gospodinov, et al. Improved shape parameter estimation in k clutter with neural networks and deep learning. *International Journal of Interactive Multimedia and Artificial Inteligence*, 3(Regular Issue), 2016.

[30] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[31] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pages 2924–2932, 2014.

[32] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[33] Mahmood R Azimi-Sadjadi and R-J Liou. Fast learning process of multilayer neural networks using recursive least squares method. *IEEE Transactions on signal processing*, 40(2):446–450, 1992.

[34] Martin T Hagan, Howard B Demuth, Mark H Beale, and Orlando De Jesus. *Neural network design*, volume 20. PWS publishing company Boston, 1996.

[35] Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[36] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[37] Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998.

[38] Martin Fodslette Moller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural networks*, 6(4):525–533, 1993.

[39] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference On*, pages 586–591. IEEE, 1993.

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

[41] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[42] Neil E Day. Estimating the components of a mixture of normal distributions. *Biometrika*, 56(3):463–474, 1969.

[43] Lior Rokach and Oded Maimon. Clustering methods. In *Data mining and knowledge discovery handbook*, pages 321–352. Springer, 2005.

[44] Herv Bourlard and Yves Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, 59(4-5):291–294, 1988.

[45] Nathalie Japkowicz, Stephen Jose Hanson, and Mark A Gluck. Nonlinear autoassociation is not equivalent to pca. *Neural computation*, 12(3):531–545, 2000.

[46] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

[47] Geoffrey Hinton. A practical guide to training restricted boltzmann machines. *Momentum*, 9(1):926, 2010.

[48] Deeplearning.net rbm tutorial. `http://deeplearning.net/tutorial/rbm.html`.

[49] Montreal research group on deep belief networks. `http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/DBNEquations`.

[50] Dumitru Erhan, Pierre-Antoine Manzagol, Yoshua Bengio, Samy Bengio, and Pascal Vincent. The difficulty of training deep architectures and the effect of unsupervised pre-training. In *AISTATS*, volume 5, pages 153–160, 2009.

[51] Ilya Sutskever, James Martens, George E Dahl, and Geoffrey E Hinton. On the importance of initialization and momentum in deep learning. *ICML*, 28:1139–1147, 2013.

[52] Stanford stacked auto-encoder tutorial. `http://ufldl.stanford.edu/wiki/index.php/Stacked-Autoencoders`.

[53] Yoshua Bengio. Early inference in energy-based models approximates back-propagation. *arXiv preprint arXiv:1510.02777*, 2015.

[54] Quora: Rbm vs. auto-encoder. `http://tinyurl.com/zo4pleh`.

[55] Richard Stuart Sutton. *Temporal credit assignment in reinforcement learning*. PhD thesis, University of Massachusetts Amherst, 1984.

[56] CJC Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, 1989.

[57] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

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

[59] RM Blumenthal. An extended markov property. *Transactions of the American Mathematical Society*, 85(1):52–72, 1957.

[60] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[61] Richard Bellman and Robert E Kalaba. *Dynamic programming and modern control theory*, volume 81. New York: Academic Press, 1965.

[62] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[63] Martin Riedmiller. Neural fitted q iteration-first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005*, pages 317–328. Springer, 2005.

[64] Martin Riedmiller. 10 steps and some tricks to set up neural reinforcement controllers. In *Neural Networks: Tricks of the Trade*, pages 735–757. Springer, 2012.

[65] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[66] Andrew William Moore. Efficient memory-based learning for robot control. Technical report, University of Cambridge, 1990.

[67] António E Ruano. *Intelligent control systems using computational intelligence techniques*, volume 1. IEEE Press Piscataway, NJ, USA, 2005.

[68] Rprop-scg comparison. `http://www.ra.cs.uni-tuebingen.de/SNNS/SNNS-Mail/96/0123.html`.

[69] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2933–2941. Curran Associates, Inc., 2014.

[70] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, pages 834–846, 1983.

[71] Guillaume Alain and Yoshua Bengio. What regularized auto-encoders learn from the data-generating distribution. *The Journal of Machine Learning Research*, 15(1):3563–3593, 2014.

[72] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[73] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[74] Ian Goodfellow, Honglak Lee, Quoc V Le, Andrew Saxe, and Andrew Y Ng. Measuring invariances in deep networks. In *Advances in neural information processing systems*, pages 646–654, 2009.

[75] Christopher Poultney, Sumit Chopra, Yann L Cun, et al. Efficient learning of sparse representations with an energy-based model. In *Advances in neural information processing systems*, pages 1137–1144, 2006.

[76] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408, 2010.

[77] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems*, pages 2933–2941, 2014.

[78] Razvan Pascanu, Yann N Dauphin, Surya Ganguli, and Yoshua Bengio. On the saddle point problem for non-convex optimization. *arXiv preprint arXiv:1405.4604*, 2014.

[79] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[80] Holger Schwenk and Maurice Milgram. Transformation invariant autoassociation with application to handwritten character recognition. *Advances in neural information processing systems*, pages 991–998, 1995.

[81] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends in Machine Learning*, 2(1):1–127, 2009.

[82] Jehoshua Bruck and Joseph W Goodman. A generalized convergence theorem for neural networks. *IEEE Transactions on Information Theory*, 34(5):1089–1092, 1988.