UNIVERSITY OF PERUGIA
Department of Mathematics and Computer Science

Master's Degree in Computer Science

Thesis

Supervised Learning of Neural Random-Access Machines with Differential Evolution

Graduand
Valerio Belli

Supervisors
Prof.ssa Valentina Poggioni
Prof. Marco Baioletti

Academic year 2016/2017

Introduction

The success of Deep Learning is undeniable. In the past years there has been an explosion of studies aimed to search new models, such as those aimed to image recognition [**?**], parsing [**?**] and speech recognition [**?**] where the models act on inputs of fixed length. At the same time a new family of models, based on the Neural Turing Machine (NTM) [**?**] has been introduced. Such models represent an attempt to train a deep network with few parameters and large memory on new types of problems, such as algorithm recognition. The success of NTM has opened the way to many other similar models, such as Neural GPUs [**?**], Neural Programmer [**?**], Neural Programmer-Interpreters [**?**], Memory Networks [**?**], Stack-Augmented Neural Networks [**?**], Differential Neural Computer [**?**] and Pointer Network [**?**].

In this thesis we consider an interesting model that takes inspiration from the NTM, the Neural Random-Access Machine (NRAM). This model, proposed in [**?**], attempts to implement the concepts of pointer manipulation and dereferencing through primitive operations. Hence, NRAM is proposed as an alternative to NTM, holding the same expressive potential while enhancing it through the use of different powerful operations. However, this does not mean that NRAM is better than NTM or the other models based on NTM. In fact, as specified in [**?**]

> [...] *As a result, the model's core primitives have a strong effect on the set of functions that can be feasibly learned in practice, similarly to the way in which the choice of a programming language strongly affects the functions that can be implemented with an extremely small*

amount of code. [...]

The original NRAM implementation is based on backpropagation, pushing it to a new level of complexity. Starting from [**?**], we tried to approach the optimization of this model through the Differential Evolution optimization algorithm. Differently from back-propagation and gradient-based algorithms, the Differential Evolution interleaves phases of exploitation and exploration in the space of the solutions. On the basis of the DENN framework, introduced in Section **??**, we have completely re-implemented the NRAM model and tested it. We compared the results with those obtained by the original model optimized by the gradient-based algorithm ADAM.

The thesis is organized as follows: in Chapter **??** we introduce the NRAM model; in Chapter **??** we explain what Artificial Neural Networks are and how can they be trained with the Gradient Descent algorithms; in Chapter **??** we introduce Differential Evolution and DENN as alternatives to Gradient Descent algorithms; in Chapter **??** we describe the implementation and finally, in Chapter **??**, we present the results of the experiments.

Contents

Chapter 1

Neural Random-Access Machines

In this chapter we will make an overview about the aspects of Neural Random-Access Machines [**?**] that has as objective the learning of algorithms. The overview starts with a simplified version containing only the registers. It is explained how the pointers manipulation works and later it is explained how the memory-augmented version works.

1.1 Registers only model

The simplified NRAM model can be divided in three main pieces: the controller, which can be a feedforward or a Long-Short Term Memory (LSTM) neural network, the registers and the gates (modules).

Let $N = \{0,\ 1,\ 2,\ \ldots,\ I-1\}$ a set of integers, where I is a integer constant. Because the model in [**?**] should be trained with the gradient descent, the NRAM does not work directly with integers but with probability distributions $p \in \mathbb{R}^{|N|}$, which must satisfy $p_i \geq 0$ and $\sum_{i=0}^{I-1} p_i = 1$. Hence, each register or memory cell plays the role of a random variable. For instance, let $I = 3$ and r_1 a register - the probability distribution contained in r_1 associated to the integer 1 is $r_1 = \{0, 1, 0\}$. In this way, the model is fully differentiable.

The registers

The registers is a set of quick access memory cells where each of them contains a vector p. The controller does not have a write direct access to the registers, but it can manipulate their values only through the modules.

The modules

In the original paper three types of gates have been introduced: constant, unary and binary. Let $M = \{m_1, m_2, \ldots, m_Q\}$ the set of the modules, then each of them can be represented as a function as follows

$$m_i \in N \text{ (Constant modules)} \tag{1.1.0.1}$$

$$m_i : N \to N \text{ (Unary modules)} \tag{1.1.0.2}$$

$$m_i : N \times N \to N \text{ (Binary modules)} \tag{1.1.0.3}$$

We use always the same sequence of fourteen modules: Read (described in the Section **??**), Zero() = 0, One() = 1, Two() = 2, Inc(a) = (a + 1) mod I, Add(a, b) = (a + b) mod I, Sub(a, b) = (a - b) mod I, Dec(a) = (a - 1) mod I, Less-Than(a, b) = [a ≤ b], Less-Or-Equal-Than(a, b) = [a ≤ b], Equal-Than(a, b) = [a = b], Min(a, b) = $\min(a, b)$, Max(a, b) = $\max(a, b)$, Write (described in the Section **??**). Using always the same sequence is important, because a different permutation of the set M can bring the NRAM to not converge. Furthermore, as for the registers, the gates have to work over probability distributions. Hence, the **a** and **b** are vectors which contain probability distributions.

1.1.1 Execution flow of register-only model

For a better comprehension of the NRAM execution, its pseudocode is visible in the Algorithm **??**.

The execution of the NRAM starts at the Line **??** and continues for all the timesteps. At the Line **??**, the controller generates a new configuration with the input from the registers, used successively at the Lines **??** and **??**. The configuration indicates how the circuit is structured, i.e. for each gate indicates

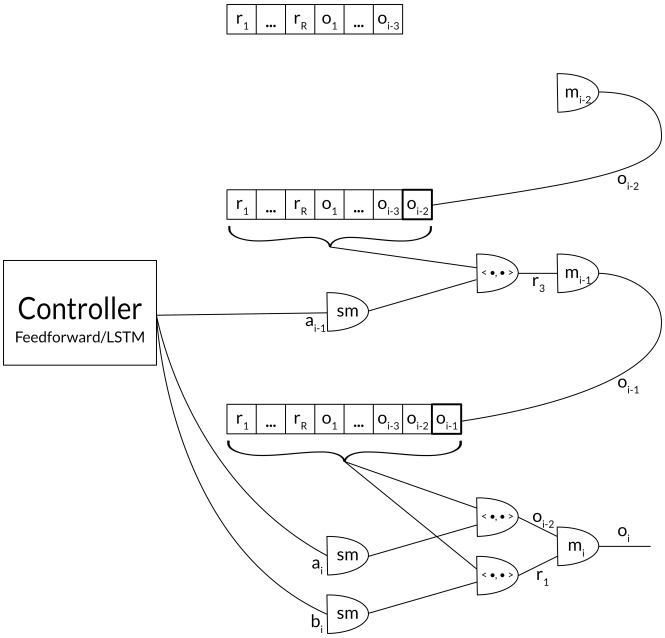


Figure 1.1: Extract of the execution of the NRAM circuit. The node with **sm** refers to the softmax activation function, the $< \cdot, \cdot >$ to a weighted average between the registers together the output of the previous modules with respect to the **softmax**(...). With the weighted average the controller selects the data indicated by the coefficient, with which the next module is fed. Here, the gates m_{i-2}, m_{i-1}, m_i are, respectively, constant, unary and binary.

the input sources. Hence, in other words, the controller learns how to connect the gates to resolve the problem on which it is trained. In other words, the inputs for the gate m_i are chosen by the controller from the set $\{r_1, \ldots, r_R, o_1, \ldots, o_{i-1}\}$ where:

- r_j is the content of the j^{th} register, $j = 1, \ldots, R$;
- o_k is the output of the k^{th} previous gate with respect to the current m_i, $k = 1, \ldots, |Gates| - 1$.

selected through a weighted average with a coefficient as follows

$$(r_1, r_2, \ldots, r_R, o_1, o_2, \ldots, o_{i-1})^T \mathbf{softmax}(s_i) \tag{1.1.1.1}$$

where $s_i \in \mathbb{R}^{R+i-1}$ is a generic vector that indicates the input source for the i^{th} gate.

Hence, let $m_i \in M$ the i^{th} module in M, m_i is executed as follows

Algorithm 1 Execution of the NRAM without the memory

Let C a controller (MLP or LSTM)
Let R the register set
Let M a set of modules
Let T timesteps
for each timestep $t \in T$ **do**
    C gets as inputs the $P(x = 0)$ of the registers
    if C is a LSTM **then**
        C updates its internal state
    end if
    C outputs one-shot the configuration of the circuit of the NRAM
    The circuit, composed by the set M, is executed
    The registers are updated
end for

- Constant gate

$$o_i = m_i() \tag{1.1.1.2}$$

- Unary gate

$$o_i = m_i((r_1, \ldots, r_R, o_1, \ldots, o_{i-1})^T \mathbf{softmax}(a_i)) \tag{1.1.1.3}$$

- Binary gate

$$o_i = m_i((r_1, \ldots, r_R, o_1, \ldots, o_{i-1})^T \mathbf{softmax}(a_i), (r_1, \ldots, r_R, o_1, \ldots, o_{i-1})^T \mathbf{softmax}(b_i)) \tag{1.1.1.4}$$

where, as stated previously, $a_i, b_i \in \mathbb{R}^{R+i-1}$ are produced by the controller and the o_i is the output that is appended to the set $\{r_1, \ldots, r_R, o_1, \ldots, o_{i-1}\}$ and used later for the subsequent modules. Since the registers contain probability distributions, the inputs of the modules are also probability distributions because they are weighted averages of probability distributions. Hence, the gates are extended to work over probability distributions and because the inputs are probability distributions also the output is a probability distribution as follows:

$$\mathbb{P}(m_i(A, B) = c) = \sum_{a,b \in N} \mathbb{P}(A = a)\mathbb{P}(B = b)[m_i(a, b) = c], \text{for } c \in N \tag{1.1.1.5}$$

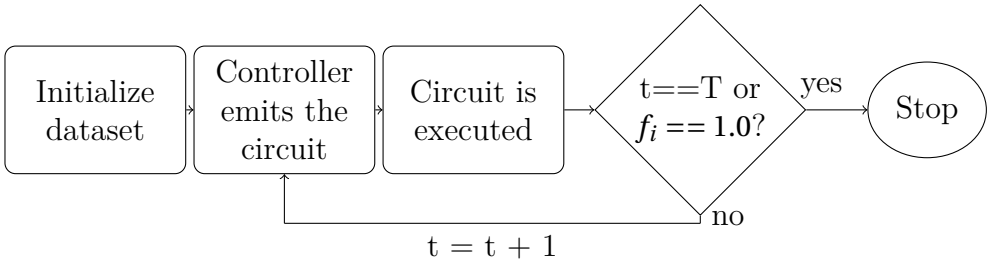


Figure 1.2: High level execution of the NRAM without the Curriculum Learning activated. Here, t is a variable which takes into account the current timestep.

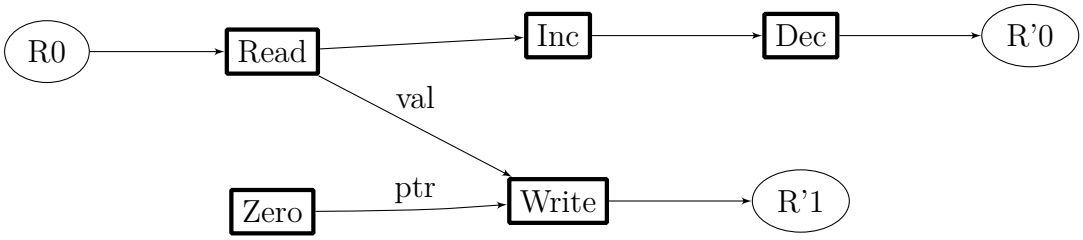


Figure 1.3: Example of NRAM circuit. With the circles are represented the registers and with the rectangles are represented the various circuits. The registers with the apex are those that are modified in the timestep. The labels x and y represent the values order for the gates (obviously if it is violated, a gate produces a different result). The modules Read and Write are presented in the next section.

At the same time of the emission of the vectors $a_i, b_i \in \mathbb{R}^{R+i-1}$, the vectors $c_i \in \mathbb{R}^{R+|M|}$ for $i = 1, 2, \ldots, R$ are also emitted, which indicate the sources, that can be the register content at timestep $t-1$ or a module output at the current timestep t, from which the new registers content is get. So after the circuit is executed, at the Line **??**, the registers content is updated as follows:

$$r_i^{(t+1)} = (r_1^{(t)}, \ldots, r_R^{(t)}, o_1, \ldots, o_{|M|})^T \mathbf{softmax}(c_i), \quad i = 1, \ldots, R \tag{1.1.1.6}$$

As stated previously, at the Line **??** the controller gets input from the registers. An easy way would be to give the entire content of the registers, recalling that each of them stores a probability distribution $\mathbb{R}^{I-1}$. However, in this way the controller cannot generalize to different memory sizes. In fact, as discussed in Section **??**, the sizes of the memory depends directly on the constant I.

Hence to resolve this inconvenient, in the original paper the controller receives from all the registers R only $P(r_i = 0)$, i.e. the probability that a register content is equal to zero. Another possibility is to give as input to the controller the

discretized registers r_i - in other words, what we is done is to discretize the probability distributions, contained in the registers, into integer, using these as input of the controller. Both the solutions limits the information available to the controller, forcing it to solve the problem with the modules instead on its own, and reducing the problem complexity.

1.2 Memory augmented model

The previously described model works only with the registers. Hence, to make that the controller learns an algorithm, the learning process starts initializing the registers with the starting input sequence of the problem. The main disadvantage of this solution is that the model is constrained to the number of the registers, unable to generalize to longer sequence of a problem because, simply, the model cannot process sequences longer than the number of the registers which is constant.

Hence to resolve this problem, the model is augmented with a variable-sized memory tape of $|N|$ memory cells, where each of them stores a distribution over the set N. Each distribution both in registers and in memory could be seen as a fuzzy address and so can used by the NRAM as a fuzzy pointer to a memory location. The memory can be formalized as a matrix $\mathcal{M} \in \mathbb{R}^{|N|}_{|N|}$, where a value $\mathcal{M}_{i,j}$ is the probability that the i^{th} memory cell contains the j^{th} integer value.

Hence, to interact with the memory the NRAM contains another two modules:

- **Read**: this module takes as input a pointer and returns as output the memory content at the location pointed by the pointer. This behaviour is the same when the pointer is an integer or is a probability distribution. More precisely, if the pointer $p \in \mathbb{R}^{|N|}$ is a probability distribution represented as a column vector, then the module returns $\mathcal{M}^T p$, while if it is an integer the result is just that cell.

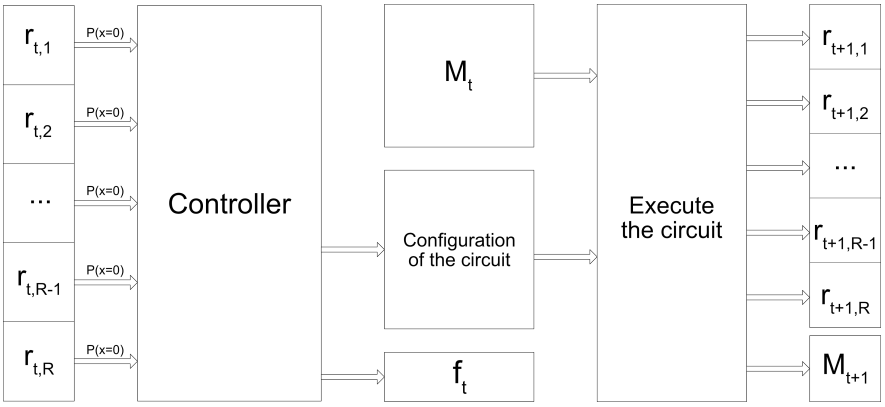


Figure 1.4: The execution of the NRAM in a timestep.

- **Write**: this module takes as input a pointer and a value and returns zero. If the inputs are probability distributions the module behave as follows:

$$\mathscr{M} = (J - p)J^T \cdot \mathscr{M} + pa^T \tag{1.2.0.1}$$

where $J \in \{1\}^{|N|}$ is a column vector and $\cdot$ denotes a coordinate-wise multiplication.

The architecture of the memory augmented NRAM is presented in the Figure **??**. Recalling the learning process of the registers-only model, in this case the memory is initialized with the problem starting input sequence used as a I/O device. In this way the controller can learn an algorithm operations over a sequence of limited size and later could apply them to a longer sequence.

The NRAM uses a system to decide if the execution can be terminated. Recalling that the execution is divided T timesteps, in each of them the controller emits along the circuit configuration a scalar $f_t \in [0,1]$[1], which represents the willingness of finish the execution in the current timestep t. Although in the paper is not explicitly specified, we decided that if $f_t = 1.0$ than the execution of NRAM is terminated. Starting from these f_i, we have the probability that the execution is not finished in the previous timestep is $\prod_{i=1}^{t-1}(1-f_i)$ and the probability

[1]The controller emits a scalar x_i, with which is produced the $f_i = \mathbf{sigmoid}(x_i)$.

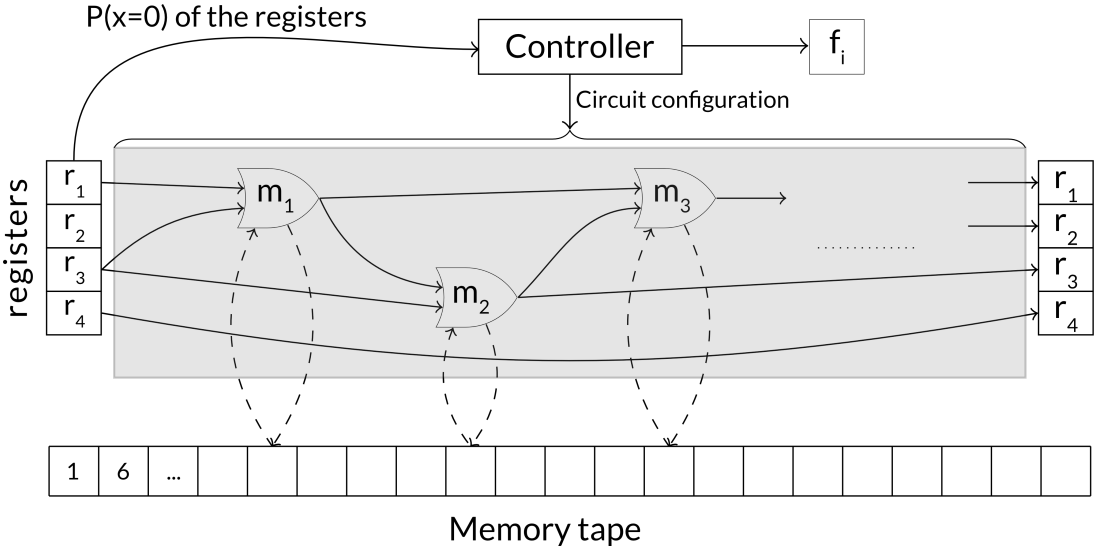


Figure 1.5: Detailed execution of the NRAM augmented with a memory with $R = 4$ registers. As can be seen the controller generates the circuit configuration and the f_i, starting from the $\mathbb{P}(r_i = 0)$ of the registers. Here, that connections are represented with solid lines, except for the interactions with the memory by the **Read** and **Write** modules that are represented with dashed lines.

that the output is produced in the current timestep is $p_t = f_t \cdot \prod_{i=1}^{t-1}(1-f_i)$. As stated previously, the execution is continued for a maximum of timesteps T, except for the previous cases, where the NRAM is forced to emits an output and, due to this, the probability to terminate is $p_T = 1 - \sum_{i=1}^{T-1} p_i$ regardless of the f_T of the last timestep.

1.2.1 Cost function

Let $\mathcal{M}^{(t)} \in \mathbb{R}^{|N|}_{|N|}$ the memory matrix at the end of the execution of the timestep t and $(x, y) \in N^{|N|}$ a couple which contains the starting input sequence and the expected output sequence, the cost function is defined, only for the memory augmented version, as the expected negative log-likelihood of producing the correct output, i.e.

$$-\sum_{i=1}^{T}\left(p_t \cdot \sum_{i=1}^{|N|} log\left(\mathcal{M}^{(t)}_{i,y_i}\right)\right) \tag{1.2.1.1}$$

where y_i is the expected integer value at the i^{th} memory cell, that here acts also as a pointer. The cost calculation is made only over the part of the memory that contains the output of interest, leaving out the other parts made available as a "free" memory which supports the NRAM execution.

Chapter 2

Artificial Neural networks

In this chapter we will make an overview about neural networks and some other concepts for a better comprehension of the following parts of the thesis.

2.1 Introduction

The Artificial Neural Networks is a family of classification technique[1], that are inspired by the human brain: in particular by the connections inside this latter. The human brain consists principally of nerve cells called **neurons**, that are connected together via the **axons** to transmit the electrical impulse by a neuron to another. This electrical impulse generated by a stimulated neuron is transferred to another one via the dendrites, a particular element in the human brain used to connect two neuron: this point of contact is called synapse.

Analogously the internal structure of a Neural Network is composed by components called **neurons**, connected together by directed links. There are many types of Neural Networks, but for the sake of simplicity and for the scope of the thesis we will focus out attention to the feedforward neural network model, explaining firstly the Perceptron model.

[1]Classification is the operation of learning a function f that map example records $x \in D$, where D is the set of (x, y), to the labels set y (the classes). This target function is called also Classification Model and is used in a descriptive or predictive way problem depending.

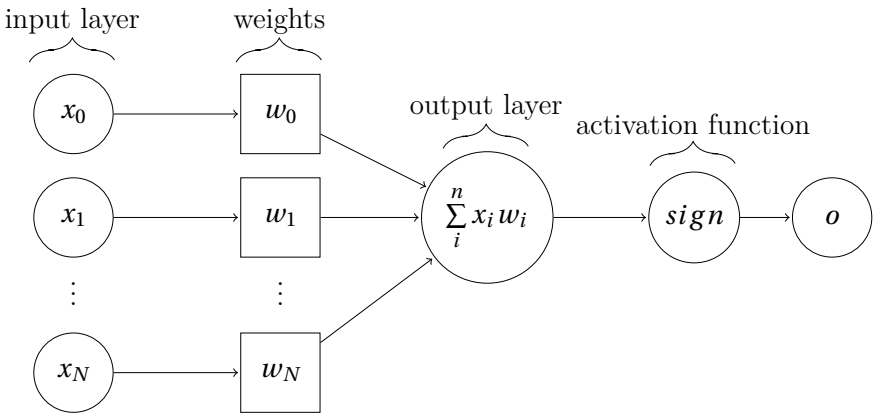


Figure 2.1: Perceptron model illustration

2.2 Perceptron

The Perceptron model is the basic and simplest type of Neural Network, firstly proposed in [**?**]. This model is composed only by two types of nodes called **input nodes** and **output node**. As we can see in the Figure **??**, the input nodes and the output node are connected by weighted links similarly to the human brain, that are used to simulate the synaptic connections strength.

This model calculates the output $\hat{y}$ as a weighted sum of the input with respect to the connections weight, to which is summed the bias (a value used to rectify, in this case, the neural network output). Hence, the output of the Perceptron model can be expressed as:

$$\hat{y} = \mathbf{g}(w_0 x_0 + w_1 x_1 + \cdots + w_N x_N + b) = \mathbf{g}(\mathbf{w} \bullet \mathbf{x} + \mathbf{b}) \tag{2.2.0.1}$$

where $i = 1, \ldots, N$ and N is the cardinality of the vector $\mathbf{x}$, $\mathbf{g}$ is called the activation function. The training of the Perceptron consists in recompute (the key passage at the step 7 of Algorithm **??**), or more precisely update, in an iterative manner the connections weight until they are able to fit the input data, i.e. the examples $(x, y) \in D$, optimizing the objective function. At the step 7 of the Algorithm **??**

$$w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i + \hat{y}_i^{(k)}) x_{ij}$$

Algorithm 2 Perceptron learning algorithm [?]

Let $D = \{(\mathbf{x}_i, y_i) \mid i = 1, \ldots, N\}$ be the set of examples.
Initialize the weight vector with random values, $\mathbf{w}^{(0)}$
repeat
  for each example $(\mathbf{x}_i, y_i) \in D$ **do**
    Compute the predicted output $\hat{y}_i^{(k)}$
    for each weight w_i **do**
      Update the weight $w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i + \hat{y}_i^{(k)})x_{ij}$
    end for
  end for
until Stopping condition is met

where $w_j^{(k)}$ is the connection weight at the step k, λ is the learning rate[2] and x_{ij} is the j^{th} feature of the i^{th} example.

2.3 Multi-Layer Perceptron (MLP)

The Multi-Layer Perceptron (MLP), known as feedforward neural network, has been principally created to solve the problems of the Perceptron because:

- It cannot handle a domain with more than two classes, because it divides the input data in only two boundaries, as it can be seen in the figure **??**
- It may does not converge if the data are not linearly separable and this leads to reduce the use of Perceptron in very few cases (trivially, when the data is linearly separable)

As for the Perceptron, the goal of MLP is to approximate a function f^* that should represents the underlying data with which the neural network is trained, e.g. a classifier where $y = f^*(\mathbf{x})$ associates an example record $\mathbf{x}$ to class y.

These models set, that include also single layered perceptrons, are called *feed-forward* because the information flows from the input layer, that contains the

[2]Learning rate is a parameter that indicates to the optimization algorithm how much quickly the neural network must abandons the "old beliefs" substituting them with the new ones, or in other words, how much the connections weight are updated.

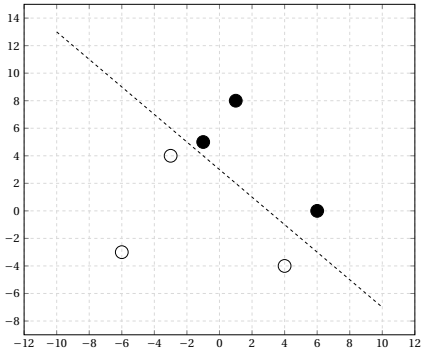


Figure 2.2: Representation of the decision separation of the Perceptron.

data of the example **x**, through all the intermediate layers called **hidden layers** and finally to the output layer **y**. So all these levels are connected only with the ones next and feedback connections that fed back the network are missing. Neural networks having this exclusive characteristic are called **recurrent neural networks**.

To all of these layers are associated different functions, called **activation functions**, that allow the layers to produce non-linear output values. Thus, in another perspective, a feedforward can be seen as a chain of activation functions. From this description we can see that the major difference between single-layered Perceptrons and feedforward neural networks is the training strategy. In fact the design of the FFN prevents to approach the training with the same strategy used with the Perceptron because we do not have *a priori* the informations regard the desired output of all the hidden layers.

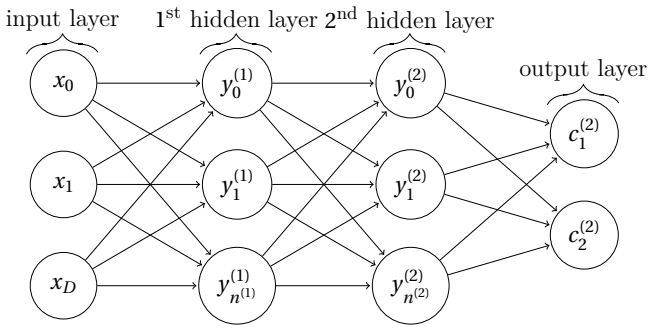
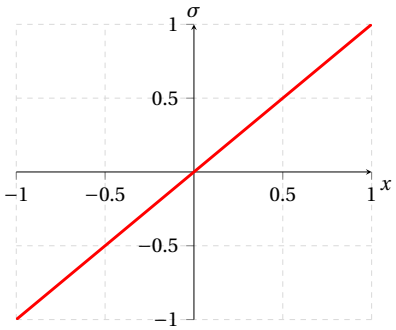


Figure 2.3: Graph representation of feedforward neural network with (2)-layer, with D input units and 2 output units that represent the classes. For simplicity of representation all the labels w_{ij} associated to the edges, that represent the weights of the neural networks, are omitted.

2.3.1 Activation functions

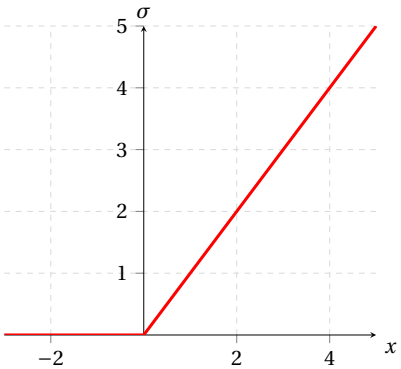
The activation functions are special functions that can be, in particular, differentiable. This last characteristic is mandatory if the used optimization algorithm is gradient-based. These functions are used in every neurons of an ANN and define how the neurons emit their values when stimulated. The following are some activation functions with their plots:

$$\sigma(x) = x$$



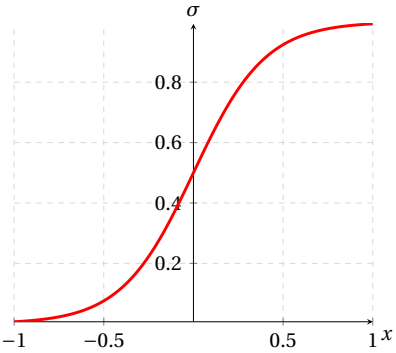
(a) Linear activation function

$$\sigma(x) = \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{for } x >= 0 \end{cases}$$



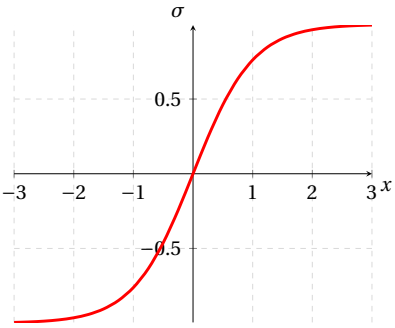
(b) ReLu activation function

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^{x}}{e^{x}+1}$$



(c) Sigmoid activation function

$$\sigma(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



(d) Tanh activation function

Figure 2.4: Some plots of activation functions.

2.4 Artificial Neural Network training

The training of ANN generally follows two main steps: the initialization of the weights of the connections and the computation of these with an iterative algorithm which is guided by the optimization of an objective function[3], searching for a global minimum[4]. The specific operations in the second step can be further split in two sub-steps: firstly the objective function is computed, then in the second step the weight of the connections are updated with some strategy/algorithm according to the objective function value.

Algorithm 3 Flow of a general algorithm based on the gradient for ANNs.

1: Let $D = \{(\mathbf{x}_i, y_i) \mid i = 1, \ldots, N\}$ be the set of examples.
2: Let λ a small real valued learning rate
3: Initialize the weights vector with random values, $\mathbf{w}^{(0)}$
4: **repeat**
5: **for** each batch of $(\mathbf{x}_i, y_i) \in D$ **do**
6: Compute the predicted output $\hat{y}_i^{(k)}$
7: Compute the gradient δ for each w_{ij} with backpropagation
8: **for** each weight w_{ij} **do**
9: Update the weight w_{ij} with some strategy using the associated δ and the learning rate λ
10: **end for**
11: **end for**
12: **until** Convergence is met

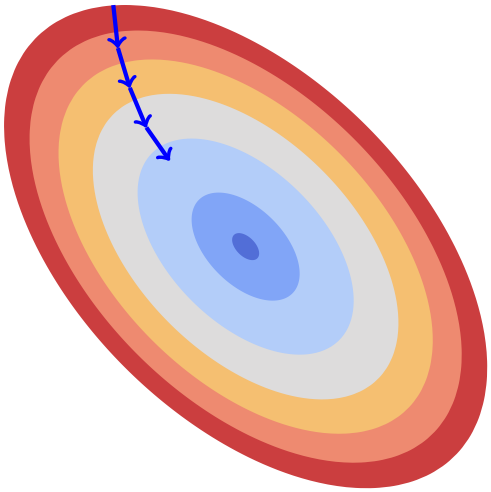
2.4.1 Gradient-based optimization algorithms

Once the gradients are computed according to the objective function, for example with the backpropagation technique presented in Section **??**, they can be used to re-compute the associated parameters. To do this it is possible to use some optimization algorithms that implement gradient descent. These algorithm have as objective the search of the global minimum of the cost function with a search

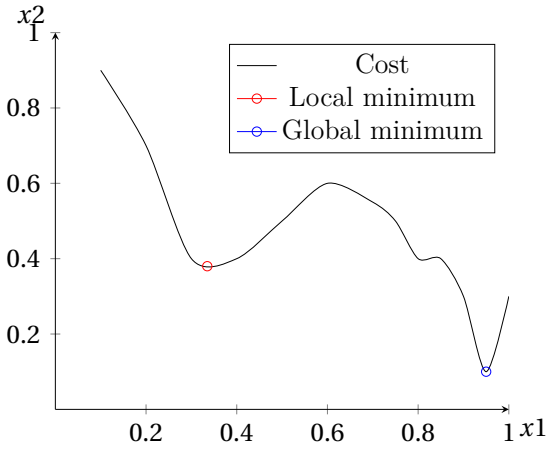
[3] In specific we speak of **cost** or **loss** function if the objective is to minimize this latter.

[4] Obviously the global minimum is not guaranteed and this depend by the problem, the ANN design and the optimization algorithm. Often what is found is a local minimum, which not guarantee that the ANN represents the underlying data.

method that is similar to the descent of a bowl (that can be associated to the complete plot of the objective function) by an hypothetical ball, where the bowl bottom corresponds to the global minimum of the function. This search of a



(a) The Gradient Descent can be seen as the descent of a bowl by a ball. The end of each arrow is a descent step of an hypothetical ball. With the red and blue are represented respectively the zones of maximum and minimum.



(b) Second example plot of gradient descent. As we can see is not certain that the global maximum can be reached in the descent.

better parameters configuration not only works with the delta, but also with a parameter λ called learning rate, that must be small for a better research. Hence, to all weights are not associated only δ but

$$\Delta w_{ij} = -\lambda \frac{\partial E}{\partial w_{ij}} \tag{2.4.1.1}$$

Stochastic Gradient Descent (SGD)

Using the equation **??** the parameters are updated as follows

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \lambda \frac{\partial E}{\partial w_{ij}} \tag{2.4.1.2}$$

Adaptive Moment (ADAM)

Let β_1 and β_2 exponential decay rates used for the momentum estimates, m_0 the 1^{st} moment vector, v_0 the 2^{nd} moment vector and ϵ a small real valued scalar we have

$$m_{ij}^{(t)} = \beta_1 m_{ij}^{(t-1)} + (1-\beta_1)\delta_{ij}^{(t)} \tag{2.4.1.3}$$

$$v_{ij}^{(t)} = \beta_2 v_{ij}^{(t-1)} + (1-\beta_2)\delta_{ij}^{2,(t)} \tag{2.4.1.4}$$

$$\hat{m}_{ij}^{(t)} = \frac{m_{ij}^{(t)}}{1-\beta_1} \tag{2.4.1.5}$$

$$\hat{v}_{ij}^{(t)} = \frac{v_{ij}^{(t)}}{1-\beta_2} \tag{2.4.1.6}$$

$$w_{ij}^{(t)} = w_{ij}^{(t)} - \lambda \frac{\hat{m}_{ij}^{(t)}}{\sqrt{\hat{v}_{ij}^{(t)}} + \epsilon} \tag{2.4.1.7}$$

Momentum The technique of momentum [**?**] takes inspiration to physical environment. It is used to accelerate learning[5], specially in case of high curvature, small and noisy gradients. Formally, the momentum is represented by a vector $\mathbf{v}$ that plays the role of velocity, i.e. the direction and speed of parameters moving through the parameters space. It is used in learning as a force to avoid that the gradients move too freely[6], conditioning too much the descent. In other words, the momentum helps the descent to move in the direction of most weight gradients push. In ADAM there are two momentum vectors, respectively the mean and the uncentered variance of the gradients.

2.4.1.1 Backpropagation

Backpropagation is an algorithm introduced in [**?**] used to optimize the weights of a neural network. For a batch of data it computes the contribution in the error of each neurons with respect to an objective function, which should capture the

[5] In Stochastic Gradient, so also in ADAM, it is used to resolve the variance of the gradient.

[6] In this case, with a physical similitude, we can see the gradients as a hockey puck sliding on a frictionless icy surface.

differences between the expected output, i.e. the y, and the generated output, i.e. the $\hat{y}$, transforming those in a real value. An example of objective function is the Total Sum of Squared errors:

$$TSS = \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{2.4.1.8}$$

The backpropagation is divided into two phases, the forward and the backward propagation.

Forward propagation Intuitively, in the forward propagation pass the network is "forward executed", i.e. starting from a batch of example, the computations for all the layers are executed up to the output layer in a forward manner. Formally, let $D = \{(\mathbf{x}_i, y_i) \mid i = 1, \ldots, N\}$ the set of examples, $L = \{(l_h^{(j)}, b_h^{(j)}) \mid j = 1, \ldots, M,\ h$ is the number of neurons of the layer$\}$ the set of FNN hidden layers and $A = \{\alpha^{(i)} \mid i = 1, \ldots, M\}$ the set of activation functions. Hence, selected the sample $(\mathbf{x}_i, y_i)$ the parameters, i.e. the weights, for the first level are computed in this way

$$l^{(1)} = \alpha^{(1)}(\sum_{i=1}^{|\mathbf{x}|} x_i l_h^{(1)} + b^{(1)}) \tag{2.4.1.9}$$

where $\alpha^{(1)}$ is the activation function of the first layer.
Subsequently, as state previously, the computation continues for all the other hidden layers as follows

$$l^{(j)} = \alpha^{(j)}(\sum_{n \in h^{(j-1)}} n h_h^{(j)} + b^{(j)}) \tag{2.4.1.10}$$

up to the output layer where we have

$$\hat{y} = \alpha^{(M)}(\sum_{n \in h^{(M-1)}} n h_{h=C}^{M} + b^{(M)}) \tag{2.4.1.11}$$

where $\hat{y}$ is the neural networks output that will be used in the backward propagation step, $h_{h=C}^{M}$ is the output level and C the classes of the problem.

Backward propagation Once the objective function is computed, the backward propagation can be executed. A strict requirement is that all the activation

functions and the objective function must be differentiable, because it uses the technique of Chain Rule of calculus which, basically, computes the derivatives of functions formed by composing other functions whose derivatives are known.

This makes it possible to calculate the error for each node starting from the output layer in a backward mode, i.e. starting from the output layer for each layers δ is calculated. Make it clearer with an example where we calculate the gradient δ for a generic parameter of the FNN. Hence, let **TSS** the loss function in **??** and w_{ij} a parameter[7] for the neuron j^{th} of l arriving from the i^{th} neuron of the previous layer $l-1$. We start by examining the partial derivative of error with respect to the parameter w_{ij} applying the chain rule

$$\frac{\partial TSS}{\partial w_{ij}} = \frac{\partial TSS}{\partial neur_j}\frac{\partial neur_j}{\partial \alpha_j}\frac{\partial \alpha_j}{\partial w_{ij}} \tag{2.4.1.12}$$

Now let's examine the parts of this partial derivative, we have for the first term

$$\frac{\partial TSS}{\partial neur_j} = \frac{\partial TSS}{\partial y} = \frac{\partial}{\partial y}\frac{1}{2}(t-y)^2 = y - t \tag{2.4.1.13}$$

if the $\text{neur}_j = y$, i.e. we are in the output layer, otherwise we must use another way if we are in an other layer. So consider all the neurons $N_i = \{u, v, \ldots, w\}$ of the $i-th$ layer that receive an input from the neuron neur_j of the previous layer, then we have a recursive expression

$$\frac{\partial TSS}{\partial neur_i} = \sum_{j \in N_i}\left(\frac{\partial TSS}{\partial \alpha_j}\frac{\partial \alpha_j}{\partial neur_i}\right) = \sum_{j \in N_i}\left(\frac{\partial TSS}{\partial neur_j}\frac{\partial neur_j}{\partial \alpha_j}w_{ij}\right) \tag{2.4.1.14}$$

Then examine the second term and we have

$$\frac{\partial neur_j}{\partial \alpha_j} = \frac{\partial}{\partial \alpha_j}\phi(\alpha_j) \tag{2.4.1.15}$$

that as we can observe it's only the derivative of the neuron with respect to the activation function α_j associated to the level.

Finally observe the third term of **??**

$$\frac{\partial \alpha_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj} neur_k\right) = \frac{\partial}{\partial w_{ij}} w_{ij} neur_i = neur_i \tag{2.4.1.16}$$

[7] The connection between two neurons.

observing that $neur_i = x_i$, if the l_j is the first layer after the input layer.

So putting all together we have

$$\frac{\partial TSS}{\partial w_{ij}} = neur_i \delta_j \tag{2.4.1.17}$$

where, considering the two case in **??** and **??**, we have

$$\delta_j = \frac{\partial TSS}{\partial n_j}\frac{\partial n_j}{\partial \alpha_j} = \begin{cases} (n_j - \hat{y}_j) n_j (1 - n_j), \text{ if } n_j \text{ is an output neuron} \\ (\sum_{r \in N_i} \delta_j w_{jr}) n_j (1 - n_j), \text{ if } n_j \text{ is a inner neuron} \end{cases} \tag{2.4.1.18}$$

2.4.2 Alternatives to Gradient Descent

Although the technique of Gradient Descent is largely used in Machine Learning for the optimization of Neural Networks, is not the unique approach to this problem. An alternative can be found in the family of Evolutionary Computation. In particular in Evolutionary Algorithms (EAs); they are a class of algorithms inspired by mechanisms of the biological evolution, used in Artificial Intelligence and Machine Learning for the research of a global optimum in a problem, i.e. a solution. This research is made evolving a set of candidates named individuals, with methods similar to which can found in Nature like reproduction, mutation, recombination and evolution. As pointed out in literature [**?**, **?**], also the EAs have been successfully used for the search and optimization of small feedforward neural networks. What is still unclear is how can be used and what results can be obtained with this algorithms in much more deep ANNs, like NRAM.

Chapter 3

Differential evolution and DENN

In the first part of this Chapter we make an overview of the iterative algorithm Differential Evolution, moving later to the mutation and crossover strategies that we have used in the experiments. Finally, we describe DENN (Differential Evolution for Neural Network) - a framework that implements and applies many of the variants, like JADE, SHADE and L-SHADE, of the algorithm Differential Evolution for training ANNs.

3.1 Differential Evolution

Differential Evolution (DE) is a metaheuristic[1] introduced in [**?**], belonging to the family of Evolutionary Algorithms (EAs), that has as objective the searching of a solution through the parallel evolution of a set of candidate solutions.

Differential Evolution is a parallel iterative direct search metaheuristic which utilizes a set, called **population**, of NP D-dimensional numerical vectors

$$x_i,\ i = 1,\ \ldots,\ NP \tag{3.1.0.1}$$

where each of them is called **individual**, **genoma** or **chromosome**. Every individual is manipulated for **G** generations, where the population is not reduced

[1] A metaheuristic is a procedure that has as objective the search, creation or selection of an heuristic that could be find a optimal solution of a problem.

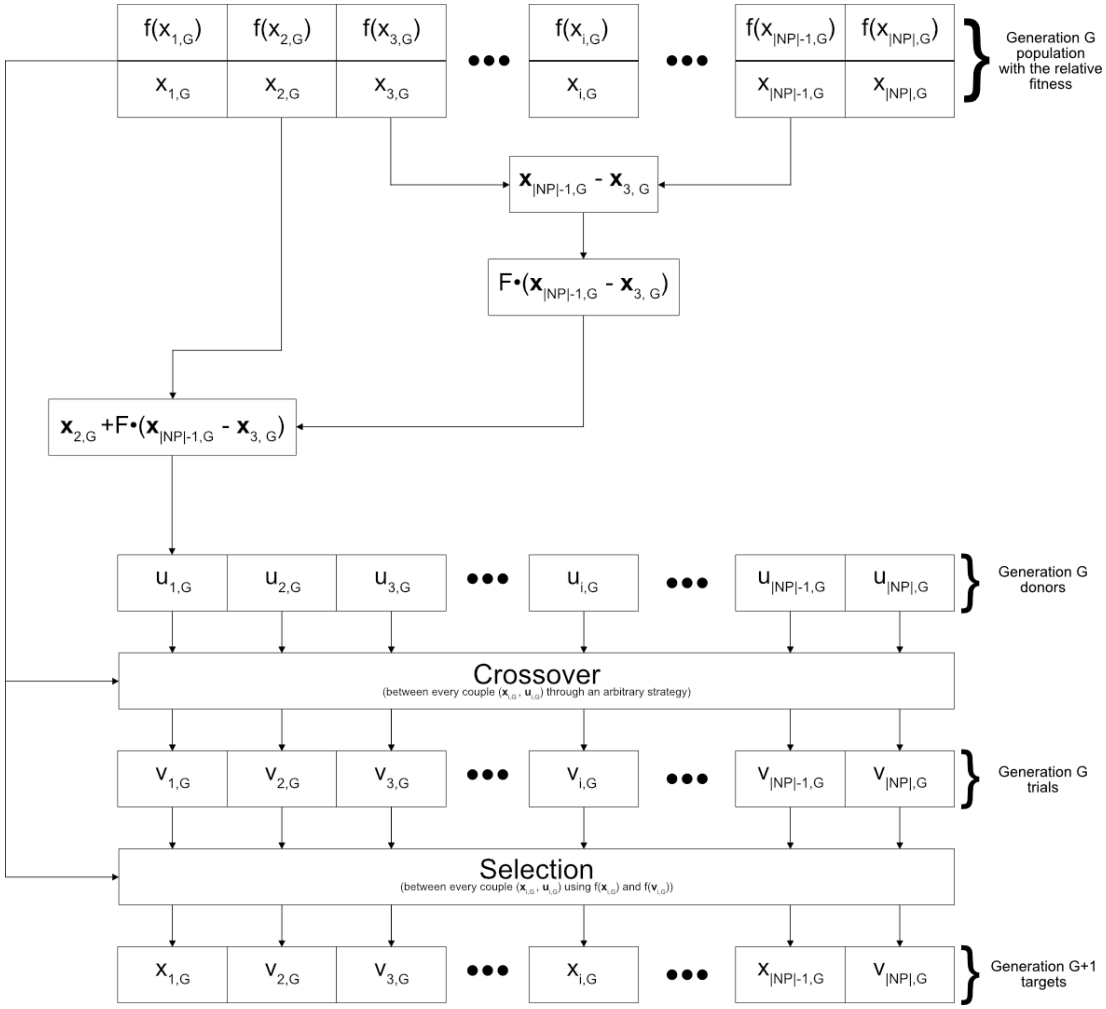


Figure 3.1: Execution flow of the Differential Evolution referred to a generation.

nor incremented, looking for one individual that can be considered as solution. The search is guided by an objective function called **fitness function**, which represents the goodness of an individual. For a better research, the individuals should be randomly initialized covering most possible the search space.

In each generation two new sets of individuals are generated, called respectively **donor** and **trial** sets. The donors set is generated mixing up individuals, called **targets**, of the population NP with a mutation operation. The targets and donors are mixed though a crossover method which creates the trials set. After this, the trials is compared one-by-one to the targets with the fitness function -

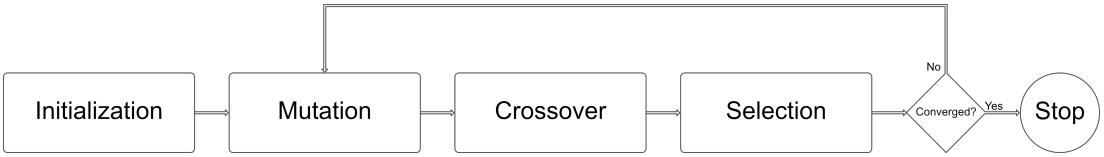


Figure 3.2: Differential Evolution flow diagram.

the better are selected creating the targets set of the next generation.

Summing up, firstly the population is initialized with some strategy - the most used is the randomly initialization of each individual. After this step, the operations executed in every DE generation are:

- **Mutation**: Let x an individual at the generation G, also called **target**. A **donor** is generated combining x with some other individuals through a differential mutation operation. For example, with the strategy **rand/1** introduced in [**?**] we have

$$v_{i,G+1} = x_{r_1,G} + F \cdot (x_{r_2,G} - x_{x_3,G}) \tag{3.1.0.2}$$

  where $r_1, r_2, r_3 \in \{1, 2, \ldots, NP\}$ are mutually exclusive indices and F is a real-valued constant defined by the user.

- **Crossover**: the crossover does nothing else than mixing up one-by-one the donor vectors components with the target vectors with some crossover methods, generating the **trials** vectors and enhancing the potential diversity of the population. For example, let the target vector $x_{i,G}$ and the mutant $v_{i,G}$ the **bin** strategy work as follows

$$u_{ji,G} = \begin{cases} v_{ji,G}, & \text{if } (randb(j) \leq CR) \text{ or } j = rnbr(i) \\ x_{ji,G}, & \text{if } (randb(j) > CR) \text{ and } j \neq rnbr(i) \end{cases} \quad i = 1, 2, \ldots, D \tag{3.1.0.3}$$

  where $randb(j)$ is a function which generate a real-valued number for the j^{th} parameter according to binomial distribution, $\mathrm{CR} \in [0, 1]$ is the global user-defined crossover constant used as a threshold and $rnbr(i)$ is a function that generate a random index which ensures that is selected at least one parameter of the mutant v_i.

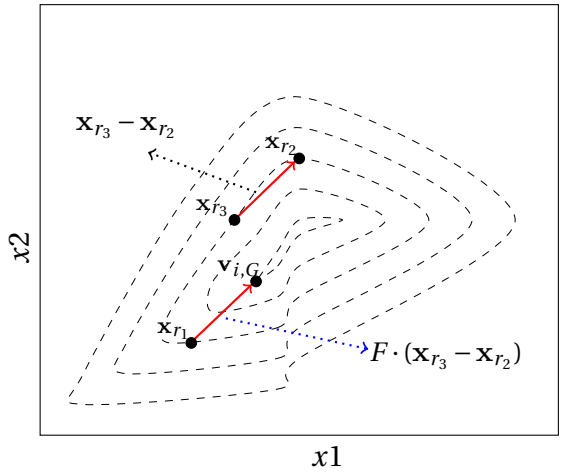


Figure 3.3: Illustration of a simple Differential Evolution mutation. $\mathbf{v}_{i,G}$ is the new donor vector, created with the scaled difference between $\mathbf{x}_{r_2}$ and $\mathbf{x}_{r_3}$ combined through summation with $\mathbf{x}_{r_1}$.

- **Selection**: Each trial is compared one-by-one to the corresponding target using the fitness function - if the target have a smaller cost with respect to the trial, than it is retained as individual of the population of the next generation and vice-versa.

In the algorithm explanation we have described one method for the mutation and one method for the crossover step. Anyway, there exist other methods of mutation and crossover. Hence, in order to classify all the variants the notation $DE/x/y/z$ is used, where:

- **DE**: indicates the optimization algorithm;
- **x**: indicates the mutation strategy;
- **y**: indicates how many targets couple are selected in the mutation step;
- **z**: indicates the crossover strategy;

Hence, for example, a possible variant is *DE/rand/1/bin*. For the thesis work have been used and tested different configurations of the DE made available by the framework DENN (**??**). Hence, about this, following are introduced some of the used algorithms and strategies.

3.1.1 Differential Evolution variants

3.1.1.1 Adaptive DE (JADE)

The base version of Differential Evolution is powerful, but one of its biggest drawback is that the constants F and CR must be selected by the user. Though [**?**] suggests to set the $F \in [0.4, 0.95]$ and $CR \in (0, 0.2)$, if the function is separable, or $CR \in (0.9, 1.0)$ when the parameters of the function are dependent, this choice remains always a problem dependent decision.Hence, JADE was introduced in [**?**] to solve this problem.

Briefly, this DE variant removes the necessity of selecting the best combination of F and CR constants. This is done with a parameter adaptation system which searches the constants best values refining them at each generation. Moreover, these constants are generated for each individual and so they are, generally, different one from the another, e.g. $CR_1 \neq CR_i$. Formally what is done is the following:

- **CR**: Let μ_{CR} be the mean of the CR values, initialized at the first generation to 0.5. For each generation the crossover probability CR_i associated to the x_i individual, is generated according to a normal distribution with mean μ_{CR} and a standard deviation 0.1

$$CR_i = \text{randn}_i(\mu_{CR}, 0.1) \tag{3.1.1.1}$$

  truncated with respect to the interval $[0, 1]$.

  After each generation, let S_{CR} be the set of the successful CR values, i.e. those values associated to the trials v which are better than the targets x. Hence, the mean μ_{CR} is recalculated as follows:

$$\mu_{CR} = (1 - c) \cdot \mu_{CR} + c \cdot \text{mean}_A(S_{CR}) \tag{3.1.1.2}$$

  where the $c \in [0, 1]$ is a positive constant and $mean_A(\cdot)$ is the arithmetic mean.

- **F**: similarly to CR, F is generated for each target with a system similar to that used for CR. Let μ_F the average of the F values, initialized at the first generation to **0.5**. Hence, for each generation and for each target individual, F_i is generated using the mean μ_F and a standard deviation **0.1** as follows:

$$F_i = \text{randc}_i(\mu_F, 0.1) \tag{3.1.1.3}$$

that is truncated if $F_i > 1$ and regenerated if $F_i \leq 0$, so that a $F_i \in (0, 1]$ and randc_i is a Cauchy random generator associated to each target individual.

As for the CR case, let S_F the set of successful mutation factors to which are added the successful Fs at the end of a generation. Then for each generation the mean μ_F is generated as follows:

$$\mu_F = (1 - c) \cdot \mu_F + c \cdot \text{mean}_L(S_F) \tag{3.1.1.4}$$

where the c is a positive constant in $[0, 1]$ and $mean_L$ is the Lehmer mean defined as:

$$\text{mean}_L(S_F) = \frac{\sum_{F \in S_F} F^2}{\sum_{F \in S_F} F} \tag{3.1.1.5}$$

Moreover, JADE has introduced an optional external memory denoted as **A** with a size equal than NP, where all the elements that have failed the selection process is stored. Once the memory is full and it is necessary to add a new failing individual, a randomly selected element is deleted. This optional memory is used in the crossover method named current-to-pbest, as said in Section **??**.

3.1.1.2 Success-History based Adaptive DE (SHADE)

The main problem of JADE is that the generation of CR_i and F_i is controlled by the memories S_{CR} and S_F, that for how are managed could contain poor settings of CR and F. Hence, this fact can leads JADE to have degraded performances.

Index	1	2	...	$H-1$	H
M_{CR}	$M_{CR,1}$	$M_{CR,2}$	...	$M_{CR,H-1}$	$M_{CR,H}$
M_F	$M_{F,1}$	$M_{F,2}$	...	$M_{F,H-1}$	$M_{F,H}$

Table 3.1: Historical memories M_{CR} and M_F

SHADE introduced in [?] aims to strengthen JADE introducing a crossover and mutation constants generation alternative system. This is done leaving out the S_{CR} and S_F memories and adding two new memories M_{CR} and M_F, named *historical memories*, where the values that have well performed in the past are stored. Formally, let M_F and M_{CR} two arrays with H cells all initialized to **0.5**. Similarly to JADE, the constants are generated in each generation for all the individuals like follows:

$$F_i = \text{randc}_i(M_{F,r_i}, 0.1) \tag{3.1.1.6}$$

$$CR_i = \text{randn}_i(M_{CR,r_i}, 0.1) \tag{3.1.1.7}$$

where M_{F,r_i} and M_{CR,r_i} are two memory cells randomly selected for each individual. Here the F_i and CR_i are managed like in JADE, i.e. are truncated or regenerated.

In every generation, after the selection is done, the successful F_i and CR_i values are recorded in temporary memories S_F and S_{CR} with which the content of the memory is updated as follows:

$$M_{F,k,G+1} = \begin{cases} \text{mean}_{WL}(S_F), & \text{if } S_F \neq 0 \\ M_{F,k,G}, & \text{otherwise} \end{cases} \tag{3.1.1.8}$$

$$M_{CR,k,G+1} = \begin{cases} \text{mean}_{WA}(S_{CR}), & \text{if } S_{CR} \neq 0 \\ M_{CR}, & \text{otherwise} \end{cases} \tag{3.1.1.9}$$

where $k \in [1, H]$ indicates the memory cell to update. k is initialized to 1 and incremented by one at the end of each generation. When $k > H$, is set again to

one. Here mean_{WL} is the weighted Lehmer mean computed as follows:

$$\text{mean}_{WL}(S_F) = \frac{\sum_{k=1}^{|S_F|} w_k \cdot S_{F,k}^2}{\sum_{k=1}^{|S_F|} w_k \cdot S_{F,k}} \tag{3.1.1.10}$$

and mean_{WA} is the weighted arithmetic mean introduced by Peng et al in [**?**], computed as follows:

$$\text{mean}_{WA}(S_{CR}) = \sum_{k=1}^{|S_{CR}|} w_k \cdot S_{CR,k} \tag{3.1.1.11}$$

$$w_k = \frac{\Delta f_k}{\sum_{k=1}^{|S_{CR}|} \Delta f_k} \tag{3.1.1.12}$$

$$\Delta f_k = |f(\mathbf{u}_{k,G}) - f(\mathbf{x}_{k,G})| \tag{3.1.1.13}$$

3.1.1.3 L-SHADE

The population size used in a EA algorithm, in this case Differential Evolution, is very important. In fact, a small population results in a faster convergence, but this can also leads the algorithm to fall in a local minimum. Conversely, a larger population increments the algorithm chance to converge to a global minimum by also increasing the computational cost. Hence, the population size is another problem dependent parameter to be optimize.

Hence, to resolve this problem, [**?**] introduced L-SHADE, a variant of SHADE that has in addition to F and CR constants optimization also a population optimization. This is done through the LSPR (Linear Population Size Reduction), a deterministic linear method that makes a population reduction using a linear function.

Let NP^{init} the initial population size and NP^{min} the minimum possible population size to use. Hence, the population for the next generation $G+1$ is calculated as follows:

$$NP_{G+1} = \text{round}\left[\left(\frac{NP^{min} - NP^{init}}{MAX\_NFE}\right) \cdot NFE + NP^{init}\right] \tag{3.1.1.14}$$

where *NFE* is the current number of fitness evaluations and *MAX_NFE* is the maximum number of fitness evaluations. The linear function at the time of population generation is not constrained to the previous population, so if $NP_{G+1} > NP_G$ then the worst $(NP_G - NP_{G+1})$ individuals are pruned from the population having also in this case a linear population reduction.

3.1.2 Mutation strategies

3.1.2.1 Rand

The Rand mutation strategy is introduced in [?] and works selecting randomly the individuals with which the donors/mutant is created. The procedure is as follows:

$$v_{i,G+1} = x_{r_1} + F \cdot (x_{r_2,G} - x_{r_3,G}) \tag{3.1.2.1}$$

$$v_{i,G+1} = x_{r_1} + F \cdot (x_{r_2,G} - x_{r_3,G}) + F \cdot (x_{r_4,G} - x_{r_5,G}) \tag{3.1.2.2}$$

where $r_1, r_2, r_3, r_4, r_5 \in \{1, 2, \ldots, NP\}$ must be mutually different and also from the related index i.

3.1.2.2 Best

The Best mutation strategy is introduced in [?] and works selecting, as the name suggests, the current best individual and some other as follows:

$$v_{i,G+1} = x_{best,G} + F \cdot (x_{r_1,G} - x_{r_2,G}) \tag{3.1.2.3}$$

$$v_{i,G+1} = x_{best,G} + F \cdot (x_{r_1,G} + x_{r_2,G} - x_{r_3,G} - x_{r_4,G}) \tag{3.1.2.4}$$

where, as for rand, $r_1, r_2, r_3, r_4 \in \{1, 2, \ldots, NP\}$ must be mutually different and also from the related index i.

3.1.2.3 DEGL

Differently from the previous two methods, DEGL, introduced in [?], makes a topological neighborhood exploration, i.e. for each generation, DEGL, creates

every individuals $\mathbf{v}_{i,G+1}$ belonging to the donors set through a convex combination between the local mutant $\mathbf{L}_{i,G}$ and the global mutant $\mathbf{g}_{i,G}$.

Formally, let the NP_G population at generation G, disposed with a ring topology. For each individual $\mathbf{x}_{i,G}$ is a neighborhood defined of $k \in [0, (NP-1)/2]$ individuals, consisting in vectors $\mathbf{x}_{i-k,G}, \ldots, \mathbf{x}_{i+k,G}$. For each individual $\mathbf{x}_{i,G}$ a local mutant $\mathbf{L}_{i,G}$ and a global mutant $\mathbf{g}_{i,G}$ are created as follows:

$$\mathbf{L}_{i,G} = \mathbf{x}_{i,G} + \alpha \cdot (\mathbf{x}_{n\_best_i,G} - \mathbf{x}_{i,G}) + \beta \cdot (\mathbf{x}_{p,G} - \mathbf{x}_{q,G}) \tag{3.1.2.5}$$

$$\mathbf{g}_{i,G} = \mathbf{x}_{i,G} + \alpha \cdot (\mathbf{x}_{g\_best,G} - \mathbf{x}_{i,G}) + \beta \cdot (\mathbf{x}_{r_1,G} - \mathbf{x}_{r_2,G}) \tag{3.1.2.6}$$

where n_best_i indicates the best individual in the neighbors of $\mathbf{x}_{i,G}$ differently to g_best which is the global best individual, $p, q \in [i-k, i+k]$, with $p \neq q \neq i$, and $r_1, r_2 \in NP_G$, with $r_1 \neq r_2 \neq i$. α and the β are real-valued scalar used as scaling factors.

After the creation of local and global donor vectors, each of them are combined through convex combination using a scalar weight $w \in (0, 1)$ in the following mode:

$$\mathbf{V}_{i,G} = w \cdot \mathbf{g}_{i,G} + (1-w) \cdot \mathbf{L}_{i,G} \tag{3.1.2.7}$$

3.1.2.4 Current-to-pbest

Current to pbest is introduced in [**?**] as an evolution of best method. In fact, best is a greedy strategy which uses prevalently the information of the best solution and this could lead the search to converge to local minimum. Instead in current-to-pbest this information can impact partially on the search, reducing in this way the possibility that the search algorithm stops in a local minimum.

This strategy exists in two versions which differentiate in the use of an auxiliary memory. The base version does not use the auxiliary memory and creates each donor $\mathbf{u}_{i,G}$ as follows:

$$\mathbf{u}_{i,G} = \mathbf{x}_{i,G} + F_i \cdot (\mathbf{x}^p_{best,G} - \mathbf{x}_{i,G}) + F_i \cdot (\mathbf{x}_{r_1,G} - \mathbf{x}_{r_2,G}) \tag{3.1.2.8}$$

where $\mathbf{x}_{best,G}$ is an individual chosen among the $100p\%$ of the current population NP_G and F_i is the mutation factor which is associated to every individual and managed as in JADE and SHADE.

Denoting by $\mathbf{A}$ the auxiliary memory where are stored the elements that have failed the selection process (e.g. if $f(\mathbf{x}_{i,G}) < f(\mathbf{v}_{i,G})$, then $\mathbf{x}_{i,G}$ is added to $\mathbf{A}$), the alternative version of Current to pbest with memory works as follows:

$$\mathbf{u}_{i,G} = \mathbf{x}_{i,G} + F_i \cdot (\mathbf{x}^p_{best,G} - \mathbf{x}_{i,G}) + F_i \cdot (\mathbf{x}_{r_1,G} - \tilde{\mathbf{x}}_{r_2,G}) \tag{3.1.2.9}$$

where $\mathbf{x}_{best,G}$, $\mathbf{x}_{r_1,G}$ and $\mathbf{x}_{i,G}$ are selected as in **??** and $\tilde{\mathbf{x}}_{r_2,G}$ is selected, instead, in $\mathbf{P} \cup \mathbf{A}$. Note that if $\mathbf{A}$ size exceeds a certain threshold, then some solutions are randomly removed.

The Current-to-pbest is further improved in association with SHADE. Here, the constant p existing in JADE versions is substituted with a variable version, i.e. to each individuals a value p_i is associated as follows:

$$p_i = rand[p_{min}, 0.2] \tag{3.1.2.10}$$

where $p_{min} = 2/NP_G$, so that at least 2 individuals are selected, and 0.2 is the maximum value as suggest in [**?**].

3.1.3 Crossover strategies

3.1.3.1 Bin

Bin is classic form of crossover and works as follows:

$$u_{ij,G} = \begin{cases} v_{ij,G}, & \text{if } randb(j) \leq CR \text{ or } j = rnbr(i) \\ x_{ij,G}, & \text{otherwise} \end{cases} \tag{3.1.3.1}$$

where $randb(i) \in [0,1]$ is a uniform random generator and $rnbr(j) \in [1, D]$ so that at least one component of mutant $\mathbf{v}_{i,G}$. The figure **??** gives a visual representation of the strategy.

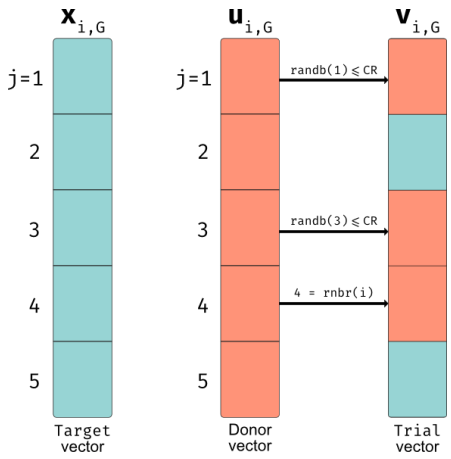


Figure 3.4: Bin crossover strategy illustration with $D = 5$.

3.1.3.2 Exp

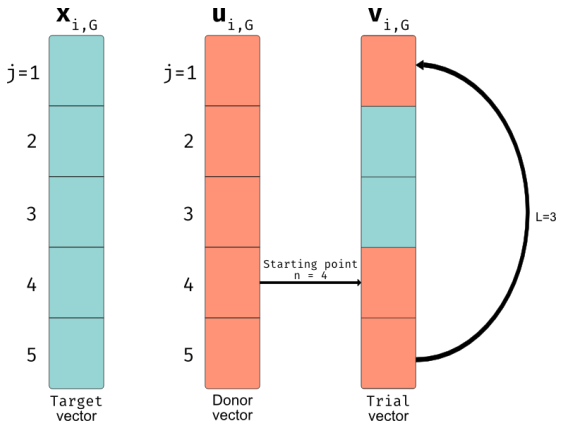


Figure 3.5: Exp crossover strategy illustration with a $N = 4$ as starting point and $L = 3$ parameters to substitute.

The crossover Exp, starts by choosing a random $n \in [1, D]$, used as an initial point in the target value where the parameters substitution starts, and a integer $L \in [1, D]$ that represents the parameters to substitute. So after choosing this two values, a trial vector is created as follows:

$$v_{ij,G} = u_{ij,G}, \text{ for } j = \{< n >_D, \ldots, < n + L - 1 >_D\} \tag{3.1.3.2}$$

$$v_{ij,G} = x_{ij,G}, \text{ otherwise} \tag{3.1.3.3}$$

where $< \cdot >_D$ is the modulo function with modulus D. An example is showed in figure **??**

3.2 DENN

DENN is a framework written in C++ by Gabriele Di Bari and Mirco Tracolli, based on their thesis work. It aims to apply the Differential Evolution concepts on the ANN training as an alternative to Gradient-based algorithms. It has been initially created as a TensorFlow extension due to the performance and simplicity offered by this latter, but now it is completely based on the C++ library Eigen due to its implementation based on the high performance library LAPACK written in Fortran 77.

3.2.1 How it works

DENN is a framework that aims to help the developer to construct a system to train Neural Networks with Differential Evolution. To achieve this goal, DENN is structured in pluggable modules through which is decided how the neural network should be trained. For example, let the Differential Evolution configuration JADE/
Rand/1/Bin - JADE, Rand/1 and Bin are three modules that can be selected for the training process at the time of configuration of DENN. The plug-in that could be used are not only for the DE, but also regard the dataset[2], how the neural network should be structured[3] and, more important, what kind of neural network you want instruct[4]. Some of the configuration parameters can be found in the Table **??**.

As said in the Chapter **??**, Differential Evolution bases its functioning also on the existence of a population of vectors, called *NP*, that is parallel evolved by the

[2]Where is it, how to manage it, etc.
[3]How many levels and the activation functions
[4]At this time, DENN can trains only feed-forward networks.

optimization algorithm aiming to arrive to the fitness function minimum value, i.e. to a solution vector which has the fitness minimum value. By the definition of DE, it is assumed that vectors are multi-dimensional, i.e. a list of features to optimize. Hence in DENN, since the population is composed by neural networks it cannot be used as is, instead the individuals must be managed as a set of weights and biases matrices, i.e. every individual is a neural network and is formed by a weights and biases matrices set. Hence, the mutation and crossover actions are not executed over the whole structure of individuals, but singularly over their subcomponents.

For a better explanation, let's examine the mutation phase through an example. Let NP the Neural Networks population, Rand/1 the mutation strategy, r_1, r_2 and r_3 three mutually exclusive indexes, the donor $\mathbf{u}_1$ is created as follows:

$$\mathbf{u}_1^{w_1} = \mathbf{x}_{r_1}^{w_1} + F \cdot (\mathbf{x}_{r_2}^{w_1} - \mathbf{x}_{r_3}^{w_1}) \tag{3.2.1.1}$$

$$\mathbf{u}_1^{b_1} = \mathbf{x}_{r_1}^{b_1} + F \cdot (\mathbf{x}_{r_2}^{b_1} - \mathbf{x}_{r_3}^{b_1}) \tag{3.2.1.2}$$

$$\dots$$

$$\mathbf{u}_1^{w_{|\mathbf{HL}|}} = \mathbf{x}_{r_1}^{w_{|\mathbf{HL}|}} + F \cdot (\mathbf{x}_{r_2}^{w_{|\mathbf{HL}|}} - \mathbf{x}_{r_3}^{w_{|\mathbf{HL}|}}) \tag{3.2.1.3}$$

$$\mathbf{u}_1^{b_{|\mathbf{HL}|}} = \mathbf{x}_{r_1}^{b_{|\mathbf{HL}|}} + F \cdot (\mathbf{x}_{r_2}^{b_{|\mathbf{HL}|}} - \mathbf{x}_{r_3}^{b_{|\mathbf{HL}|}}) \tag{3.2.1.4}$$

where w_i and b_i represent the weights and bias vectors of the i$^{\text{th}}$ level and **HL** is the hidden layers set. The same work is made also for the crossover phase, so we do not explain it with an example which is leaved to the reader.

Argument	Description
Execution args	
threads_pop	Executed threads of DENN
seed	Distribution seed
instance	Type of model (nram/default)
Batch info	
batch_size	Number of training examples
batch_offset	Examples per batch
use_validation	Activate pop validation at the end of a sub- geneneration
compute_test_per_pass	Compute the test accuracy for each pass
DE	
generations	Total number of generations
sub_gens	Number of sub-generations
number_parents	DE population size
f	DE F coefficient
cr	DE CR coefficient
evolution_method	Evolution method (JADE/SHADE/L-SHADE)
mutation	Mutation method (degl/curr_p_best)
crossover	Crossover method (bin/exp/interm)
Network	
hidden_layers	Levels size
activation_functions	Activation functions of levels
NRAM	
task	Task to execute
max_int	Max int in the set
sequence_size	The size of the input sequence
n_registers	Registers to use
time_steps	Execution timesteps
gates	Gates to use
change_difficulty_level	The lambda under of which the difficulty can be changed
step_gen_change_difficulty	Number of gen where the same difficulty is used

Table 3.2: Some arguments could be used in the configuration file of DENN.

Chapter 4

Implementation

Our thesis work is split in three different projects: a refactoring of the already existing project[1] of Andrew Gibiansky, written in Python and based on Theano[2] library, where some parts of NRAM and a parametrization system are missing, an implementation in C++ of the NRAM using DENN as optimization engine, and an additional implementation[3] of the NRAM used to test the generalization ability of the discovered ANNs.

Preface Since some of the tasks presented in Chapter **??** are too complex to be learned by a plain NRAM, we have also used some of the enhanchements described in [**?**].

Gradient clipping The size of the model can be extremely large and moreover extremely depth, due to the execution in various timesteps. In these types of networks the gradient can often explode, as noticed in [**?**]. Hence, in the

[1] https://github.com/gibiansky/experiments/tree/master/nram

[2] Theano is a Python library created at LISA labs of University of Montreal, which lets to define easily complex mathematical expressions and execute them efficiently in the CPU and GPU specially with those which involves multi-dimensional arrays. The expressions definition is made through the creation of computational graph which is then executed and, if requested, automatic differentiated with an automatic differentiator, e.g. when one would calculate the gradient.

[3] https://github.com/DrugoLebowski/denn-lite-nram-executor

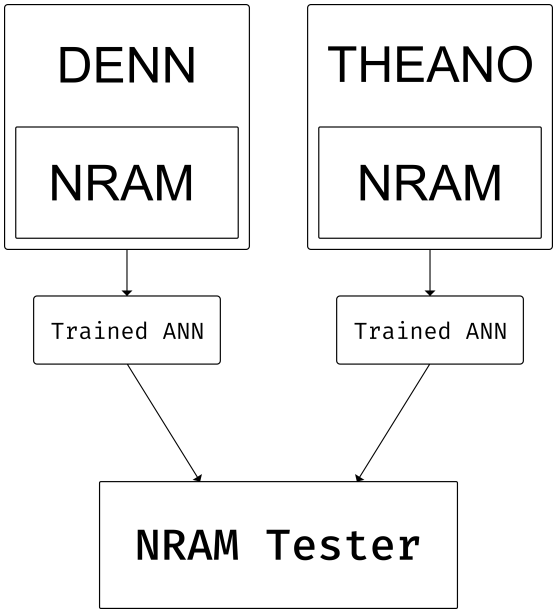


Figure 4.1: System structure: the first two blocks are the implementations used to train the neural networks. Once a controller is trained, it can be run with the NRAM Tester, i.e. the *vanilla* version without the training phase, to test the generalization ability and to visualize the produced circuits.

Theano implementation all the gradients are clipped in the range $[-C_1, C_1]$ and successively rescaled, such that its L_2 norm[4] is not larger than some constant C_2.

Noise As noticed in [**?**], in the Theano implementation a Gaussian noise, whose variance decays over times, is added to the computed gradient.

Entropy As for [**?**], we also noticed that the network can fix the registers used as pointers in some value. Although it could be an advantage in some cases, if this happens too early in the training phase, it could force the network to stay fixed on some pointer with a very small chances of change. Hence, to alleviate this condition, we give to the network a sort of a "entropy bonus" which decreases over time. This entropy is computed for each generated distribution[5] by the neural network with the Shannon entropy and is multiplied by the following coefficient

[4]Let $\mathbf{x} = [x_1, x_2, \ldots, x_n]$, the norm L_2 is defined as $|\mathbf{x}| = \sqrt{\sum_{k=1}^{n} |x_i|^2}$.

[5]We have identified these distributions be those in the memory.

that decreases over time

$$E = e * d^t \tag{4.0.0.1}$$

where e is the entropy coefficient, d is the decay of the entropy coefficient and t is the timestep. After that, the computed value is subtracted from the cost of the timestep t (Please refer to Section **??** to see how the cost is computed).

Curriculum learning As noticed by [**?**] and [**?**], curriculum learning is crucial to train very deep neural networks. Hence, as in the original paper, we decided to implement in the system the curriculum learning described in [**?**].

For each task, we have defined manually a set of problems of increasing difficulties, where the difficulty d is defined as a tuple containing the length of the working sequence and the number of timesteps of running. During the training, a difficulty tuple is selected according to the current difficulty D as follows:

- with probability 10%: pick d randomly from the set of all difficulties according to a uniform distribution.
- with probability 25%: pick d randomly from the set $[1, D+e]$ according to a uniform distribution, with e is generated from a geometric distribution with a success probability of 0.5.
- with probability 65%: set $d = D + e$ where e is sampled as above.

The difficulty is modified when $1 - \frac{c}{m} < \lambda$. $1 - \frac{c}{m}$ is the error rate, c represents the correct modified cells, m the cells that should be modified and λ is a threshold chosen by the user. Furthermore, we ensure that the difficulty is modified again at least after a given number of generations.

4.1 How we did it

The implementation of NRAM module is similar both in DENN and Theano versions, differentiating only for the training algorithm and for some specific technique (for example, Gradient Clipping is useless in the implementation with

DENN and so it is not implemented). Hence, leaving out these specifics of the framework we speak only about the NRAM implementation which is presented for simplicity as pseudocode[6].
The main block of NRAM can be seen in Algorithm **??**. The execution starts with the setting of some variables, such as cost variable and the constants used for the computing of the entropy. The main cycle is at Line **??** - in every timestep the network releases all the coefficients, i.e. the circuit connections, at Line **??**. At Line **??** is executed the circuit, with the Algorithm **??** - it returns the willingness of terminate the execution with which is computed from Line **??** to **??** the timestep cost, added later to the total cost. At Line **??**, if the willingness of terminate reaches the value **1.0** then the execution of the NRAM is terminated. The "magic" of NRAM happens in Algorithm **??**, where the circuit is executed. From Line **??** to **??**, each gate in the list **Gates** is executed, getting an input[7] and producing an output, which might be accessed later by another gate or register. From Line **??** to **??**, the registers are updated. As it can be seen in the Lines **??**, **??**, **??** and **??**, the selection of the input of a gate and the new content of the register is made through the function in the Algorithm **??** which do a weighted average. An example of gate is visible in Algorithm **??** - the **add** gate compute the summation of two number represented as two probability distributions. Finally, the functions GetGateCoefficient and GetRegisterCoefficient are purely conceptual functions which indicates a method with which the coefficients are acquired.

The pseudocode of the implementation of the NRAM used to test the generalization ability of the discovered ANNs is showed in Algorithm **??**. For each test, defined by the size of the memory and the timesteps, is executed the NRAM for each sample - with the results at Line **??** is written to file the circuits generated during the execution. At Line **??** and at Line **??**, when all the samples of a difficulty are managed, are printed to an image the memories and to a csv the error rate reached by NRAM for the current difficulty.

[6]The entire code is available at https://github.com/Gabriele91/DENN-LITE/tree/nram

[7]Depending by the type - Costant gate produce a constant output without getting an input, unary and binary gate gate get, respectively, one and two input.

4.1.1 Datasets

The datasets used during the training are automatically generated at runtime. Each batch of samples is composed by:

- initial memories: the memories which are manipulated by NRAM during the execution;
- expected memories: the memories which are compared to the modified initial memories for cost computation;
- cost masks: used to take into account only parts of the memories during the computing of the cost;
- error rate masks: used to take into account only parts of the memories during the computing of the error rate;

Remembering how a classification problem is defined, from an high level the x is formed by the Registers and "Initial memory" and the y is the "Expected Memory". The datasets are generated only at the beginning of the training of the NRAM, except when the Curriculum Learning is active in which are re-generated to every change of difficulty.

Algorithm 4 Pseudocode of the main block of NRAM.

```
1: function NRAM(NeuralNetwork, BatchSize, Regs, IMem, OMem, Cost-
   Mask, MaxInt, T, Gates)
2:     Cost = 0.0
3:     ProbIncomplete = 1.0
4:     CumProbComplete = 0.0;
5:     p_T = 1.0
6:     EntropyCoeff = 0.1
7:     EntropyDecay = 0.999
8:     for t ∈ [1,...,T] do
9:         Conf = NeuralNetwork.predict(Regs.ProbZero())
10:        {f_i, Regs, IMem} = RunCircuit(Gates, Conf, Regs, IMem, MaxInt)
11:
12:        if t == T then
13:            p_T = 1 - CumProbComplete
14:        else
15:            p_T = f_i * ProbIncomplete
16:        end if
17:
18:        CumProbComplete = CumProbComplete + p_t
19:        ProbIncomplete = ProbIncomplete * (1 − f_i)
20:
21:        EntropyWeight = EntropyCoeff ∗ EntropyDecay^t
22:        EntropyCost = EntropyCost+Entropy(IMem) * EntropyWeight
23:
24:        Cost += CalculateCost(InMem, OutMem, CostMask) - Entropy-
   Cost
25:        if f_i ≥ 1.0 then
26:            break
27:        end if
28:    end for
29:
30:    return Cost
31: end function
```

Algorithm 5 Pseudocode of the run circuit function.

```
1: function RunCircuit(Gates, Conf, Regs, IMem, MaxInt)
2:     RegsAndOutput = Copy(Regs)
3:
4:     for each g ∈ Gates do                          ▷ Run of the circuit
5:         if g.isConstant( )then
6:             GateResult = g.Value
7:         else if g.isUnary( )then
8:             Coeff = GetGateCoefficient(Conf, g)
9:             InputValue = Avg(RegsAndOutput, Coeff)
10:            GateResult = g.execute(InputValue, IMem, MaxInt)
11:        else
12:            (CoeffA, CoeffB) = GetGateCoefficient(Conf, g)
13:            InputValueA = Avg(RegsAndOutput, CoeffA)
14:            InputValueB = Avg(RegsAndOutput, CoeffB)
15:            GateResult = g.execute(GateValueA, GateValueB, IMem, Max-
    Int)
16:        end if
17:        RegsAndOutput = Concatenate(Regs, GateValue)
18:    end for

19:    for i ∈ [1,...,|Regs|] do                      ▷ Update of the registers
20:        Coeff = GetGateCoefficient(Conf, i)
21:        NewValue = Avg(RegsAndOutput, Coeff)
22:        Regs[i] = NewValue
23:    end for
24:
25:    return (Conf[Conf.size(-) 1], Regs, IMem)
26: end function
```

Algorithm 6 Pseudocode of the avg function

```
1: function Avg(RegsOut, Coefficient)
2:     return RegsOut * Coefficient
3: end function
```

Algorithm 7 Example of NRAM gate.

```
1: function ADD(A, B, IMem, MaxInt)
2:     C = ZERO(MAXINT)▷ Matrix of zeros where is stored the output of the module Add
3:     for i ∈ [0, MaxInt − 1] do
4:         for j ∈ [0, MaxInt − 1] do
5:             C[j] += A[i] * B[(j - i) % MaxInt]
6:         end for
7:     end for
8:     return C
9: end function
```

Algorithm 8 Pseudocode of the NRAM tester.

```
1: function NRAMTEST(NeuralNetwork, BatchSize, Tests, Gates)
2:     for {MaxInt, Timestep} ∈Tests do
3:         {Regs, IMem, OMem, CostMask} = GENERATEBATCH(BatchSize, MaxInt, Timestep)
4:         for s ∈ [1,...,BatchSize] do
5:             SampleIMem = IMem[s]
6:             SampleOMem = OMem[s]
7:             SampleRegs = Regs[s]
8:             for t ∈ [1,...,Timestep] do
9:                 Conf = NEURALNETWORK.PREDICT(Regs.ProbZero())
10:                {SampleRegs, SampleIMem} = RUNCIRCUIT(Gates, Conf, SampleRegs, SampleIMem, MaxInt)
11:                PRINTCIRCUITTOFILE(Conf)
12:            end for
13:        end for
14:        PRINTMEMORIESTOFILE(SampleIMem)
15:        PRINTERRORRATETOFILE(SampleIMem, SampleOMem, CostMask)
16:    end for
17: end function
```

Chapter 5

Experiments

In the first part of this chapter we present some problems on which the NRAM is trained, while in the second part, we compare the results and we show the circuits learned by the neural networks.

5.1 Tasks

The following are the description of the executed task used in our experiments. In the description, big and small letters represents respectively arrays and pointers, *NULL* denotes the value 0 and is used as an ending character or in the lists, as a placeholder for missing next element. In the experiments, along the initial and desired memories, are also generated the cost masks that used during the cost calculation as attention mechanisms.

5.1.1 Access

Given a value k and an array $\mathbf{A}$, return $\mathbf{A}[k]$. Input is given as k, $A[0]$, ..., $\mathbf{A}[n-1]$, *NULL* and the network should replace the first memory cell with $\mathbf{A}[k]$. An example is visible in Figure **??**.

Initial memory									
4	5	1	4	7	2	8	3	6	0
Desired memory									
7	5	1	4	7	2	8	3	6	0
Cost mask									
1	1	1	1	1	1	1	1	1	1
Error mask									
1	0	0	0	0	0	0	0	0	0

Table 5.1: Expected behaviour with the task Access – the first value is the pointer in the sequence A to which the NRAM should access.

5.1.2 Increment

Given an array **A**, increment all its elements by 1. Input is given as $\mathbf{A}[0]$, ..., $\mathbf{A}[n-1]$, *NULL* and the expected output is $\mathbf{A}[0]+1$, ..., $A[n-1]+1$. An example is visible in Figure **??**.

Initial memory									
5	5	9	4	7	8	0	0	0	0
Desired memory									
6	6	0	4	8	9	0	0	0	0
Cost mask									
1	1	1	1	1	1	1	1	1	1
Error mask									
1	1	1	1	1	1	0	0	0	0

Table 5.2: Expected behaviour with the task Increment – each element must be incremented by one, also considering the interval of the values N.

5.1.3 Copy

Given an array and a pointer to the destination, copy all elements from the array to the given location. Input is given as p, $\mathbf{A}[0]$, ..., $\mathbf{A}[n-1]$ where p points to one element after $\mathbf{A}[n-1]$. The expected output is $\mathbf{A}[0]$, ..., $\mathbf{A}[n-1]$ at positions p, ..., $p+n-1$ respectively. An example is visible in Figure **??**.

Initial memory									
5	5	1	4	7	0	0	0	0	0
Desired memory									
5	5	1	4	7	5	1	4	7	0
Cost mask									
0	1	1	1	1	1	1	1	1	1
Error mask									
0	0	0	0	0	1	1	1	1	0

Table 5.3: Expected behaviour with the task Copy – the first value is the pointer to the memory to which the NRAM should starts copy the sequence A.

5.1.4 Reverse

Given an array and a pointer to the destination, copy all elements from the array in reversed order. Input is given as $p, \mathbf{A}[0], \ldots, \mathbf{A}[n-1]$ where p points one element after $\mathbf{A}[n-1]$. The expected output is $\mathbf{A}[n-1], \ldots, \mathbf{A}[0]$ at positions $p, \ldots, p+n-1$ respectively. An example is visible in Figure **??**.

Initial memory									
5	5	1	4	7	0	0	0	0	0
Desired memory									
5	5	1	4	7	7	4	1	5	0
Cost mask									
0	1	1	1	1	1	1	1	1	1
Error mask									
0	0	0	0	0	1	1	1	1	0

Table 5.4: Expected behaviour with the task Reverse – the first value is the pointer to the memory to which the NRAM should starts reverse the sequence A.

5.2 The generalization problem

Generalization means is the ability of a neural network to recognize patterns in new examples which were never seen before. In a classical classification problem this means that the neural network should recognize the patterns in the features

Task	Train complexity		Reached cost 0		Train error		Generalization	
	No CL	CL	No CL	CL	No CL	CL	No CL	CL
Access	$\text{len}(A) = 8, t = 5$	$\text{len}(A) \leq 10$	✓	✓	0	0	Perfect	Perfect
Increment	$\text{len}(A) = 9, t = 4$	$\text{len}(A) \leq 10$	✓	✓	0	0	Perfect	Perfect
Copy	$\text{len}(A) = 5, t = 11$	$\text{len}(A) \leq 9$	×	×	—	—	—	—
Reverse	$\text{len}(A) = 4, t = 9$	$\text{len}(A) \leq 8$	✓	✓	0	0	Perfect	Perfect

Table 5.5: Results of the tests with DENN. The train complexity represents the maximum length of integers sequence A used in the training. The train error represents the lowest error rate reached by the trained neural networks. The generalization represents the behaviour of the neural networks with memory sequences longer and more timesteps with respect to those used in training. The evaluation is executed as in [**?**].With "—" we indicate the lack of informations due to non-convergence of controller.

presented to it, classifying them with the correct label. The same concept can be also applied to NRAM, with some differences. Remembering that the training is made using memory with limited size and a given number of timesteps, "generalization ability" means that the controller should learn to create the right circuits also for examples[1] with memory of different sizes and a different number of timesteps.

As stated previously in the Paragraph **??**, the Curriculum Learning is used to boost the training making the neural network more robust and giving it a better generalization ability. In fact, in some cases the training using a specific memory size and a given number of timesteps lead the objective function to have a value equal to zero, but the controller is too specialized (overfitting) and produces circuits unable to generalize.

5.3 Results

In the experiments we have used the same structure in [**?**], training NRAM with DENN and Theano implementations on some "easy" tasks. The main objective are to compare the performances of two versions (DENN and Theano) both with and without Curriculum Learning, focusing specially to the generalization ability of the discovered solutions.

The used variants of DE are **JADE**, **SHADE** and **L-SHADE**, which are

[1]Please refer to Section **??** to see how the examples are formed.

Task	Train complexity	Reached cost 0
Access	$\text{len}(A) = 8, t = 5$	×
Increment	$\text{len}(A) = 9, t = 4$	×
Copy	$\text{len}(A) = 5, t = 11$	×
Reverse	$\text{len}(A) = 4, t = 9$	×

Table 5.6: Results of the tests of ADAM. The contained informations follows those in Table **??**.

Step	0	1	2	3	4	5	6	7	8	9	10	11	$r0$	$r1$	$r2$	$r3$	Read	Write
1	6	4	3	6	5	2	0	0	0	0	0	0	0	0	0	0	p:0	p:0 v:6
2	6	4	3	6	5	2	0	0	0	0	0	0	6	1	0	0	p:1	p:10 v:4
3	6	4	3	6	5	2	0	0	0	0	4	0	10	2	0	0	p:2	p:9 v:3
4	6	4	3	6	5	2	0	0	0	3	4	0	9	3	0	0	p:3	p:8 v:6
5	6	4	3	6	5	2	0	0	6	3	4	0	8	4	0	0	p:4	p:7 v:5
6	6	4	3	6	5	2	0	5	6	3	4	0	7	5	0	0	p:5	p:6 v:2
7	6	4	3	6	5	2	2	5	6	3	4	0	6	6	0	0	p:6	p:5 v:2
8	6	4	3	6	5	2	2	5	6	3	4	0	5	7	0	0	p:7	p:4 v:5
9	6	4	3	6	5	2	2	5	6	3	4	0	4	8	0	0	p:8	p:3 v:6
10	6	4	3	6	5	2	2	5	6	3	4	0	3	9	0	0	p:9	p:2 v:3
11	6	4	3	6	5	2	2	5	6	3	4	0	2	10	0	0	p:10	p:1 v:4
Final	6	4	3	6	5	2	2	5	6	3	4	0	1	11	0	0	×	×

Table 5.7: Example of execution showing the memory, register and Read/Write gates states of Task Reverse with the controller discovered without the using of Curriculum Learning. The keywords **p** and **v** indicates respectively the pointer which is accessed and the value which is written.

combined with the mutation methods **DEGL** and **Current-to-pbest** and the crossover method **bin**. We used in each test a population of $[100, 400]$ individuals. For DEGL we used a neighborhood in the interval $[4, 8]$ and for Current-to-pbest we set p to 0.1. All the tests are executed with a feedforward neural network with two hidden layers,. The training set is composed by batches of 1000 examples generated at runtime. The cost calculation is done with the cost function introduced in Section **??**. The train error is computed with the expression introduced in Paragraph **??**. The generalization tests are executed with input sequences up to 1000 values. The Curriculum Learning was set up with an interval of batches in $[25, 250]$ and a threshold equal to 0.1.

Task	JADE/DEGL/1		JADE/Current-to-pbest/1		SHADE/DEGL/1		SHADE/Current-to-pbest/1		L-SHADE/DEGL/1		L-SHADE/Current-to-pbest/1	
	No CL	CL	No CL	CL	No CL	CL	No CL	CL	No CL	CL	No CL	CL
Access	×	✓	✓	✓	×	×	×	✓	×	×	×	×
Increment	×	×	✓	×	×	×	×	✓	×	×	×	×
Copy	×	×	×	×	×	×	×	×	×	×	×	×
Reverse	×	✓	✓	×	✓	✓	×	✓	×	×	×	×

Table 5.8: Convergence table. Although not specified, all the variants are associated with the crossover method **bin**. As it can be seen, L-SHADE seems unable to reach a solution in all the tasks, probably due to the linear reducing of the population. For the other cases, except for the task **Increment** with JADE/Current-to-pbest/1/bin, it can be seen that Curriculum Learning boost the training of the controller.

The training was executed using configurations of Max-int, timesteps and registers compatible with those used by Kurach et al. As showed in the Table **??**, for the easy tasks trained with Differential Evolution without the Curriculum Learning we found a set of hyperparameter which reached error 0 during the training, even using shorter input sequences compared to those used in [**?**]. Differently from which declared in Kurach at al., for Copy task the search of hyperparameters has been difficult. The same results was not reached in the training with Theano and ADAM, as showed in **??**. Due to the lack of the algorithmic details in [**?**] our implementations probably differ from the original and an exact comparison cannot be made.

For the easy tasks, the Curriculum Learning is not always necessary — in fact, for the tasks **access**, **increment** and **reverse** is possible to search a combination of hyperparameters, both for NRAM and DENN, that make the converge to a solution which generalize perfectly possible. However, we observed that the generated circuits produced by a controller trained with the Curriculum Learning, in general are more regulars or smart, like in the case of increment showed in **??**.

5.3.1 DENN variants results

In this section we show the convergence graphics of the cost function during the training. In the following charts in Figures **??**, **??** and **??** we plotted the values of the cost function obtained running several DE variants as described in the legends. For each chart we chose a different number of generations because of different problem difficulties: while 350 generations are sufficient to train a controller for the Access problem, we need 1000 generations for the Increment problem.

It is difficult to provide a final interpretation of these results because more tests and run have to be done, but considering the data obtained so far we can conclude that, a part the case of JADE/DEGL/1/bin + CL, the best performing variants are the ones with the Current-to-pbest mutation method. Despite this, it is clear that the contribution of the Curriculum Learning is crucial in some cases like in the Increment task.

Access

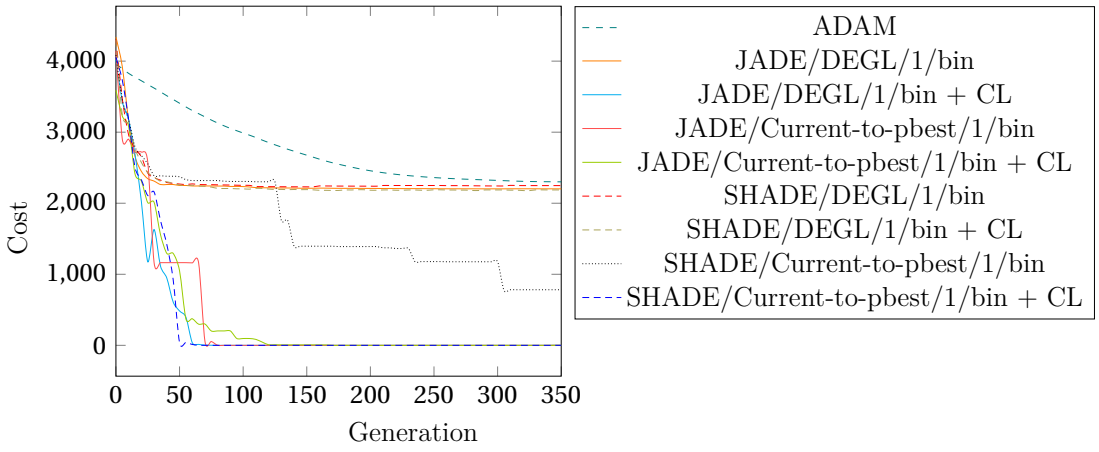


Figure 5.1: Costs convergence of DE variants with task **Access**.

Increment

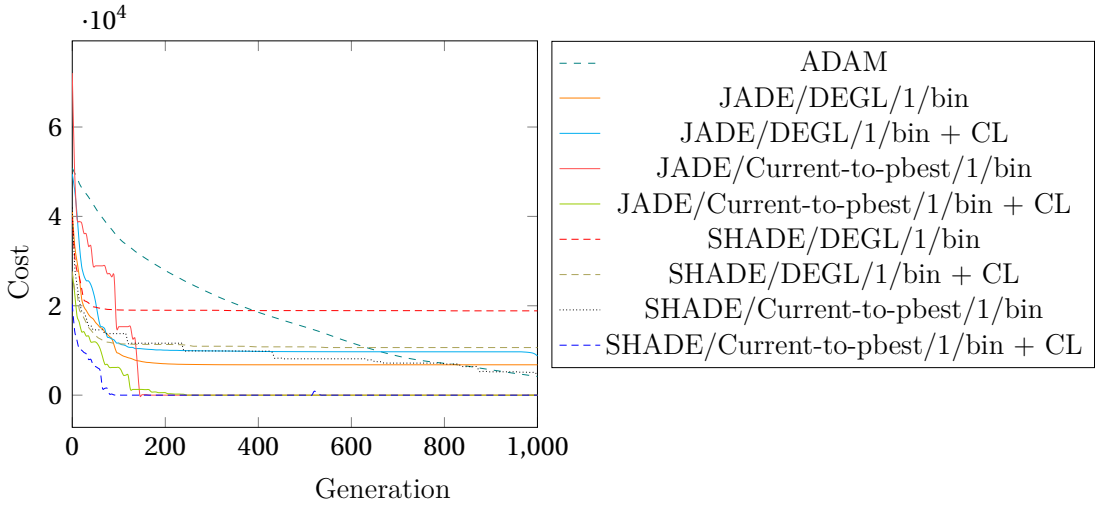


Figure 5.2: Costs convergence of DE variants with task **Increment**.

Reverse

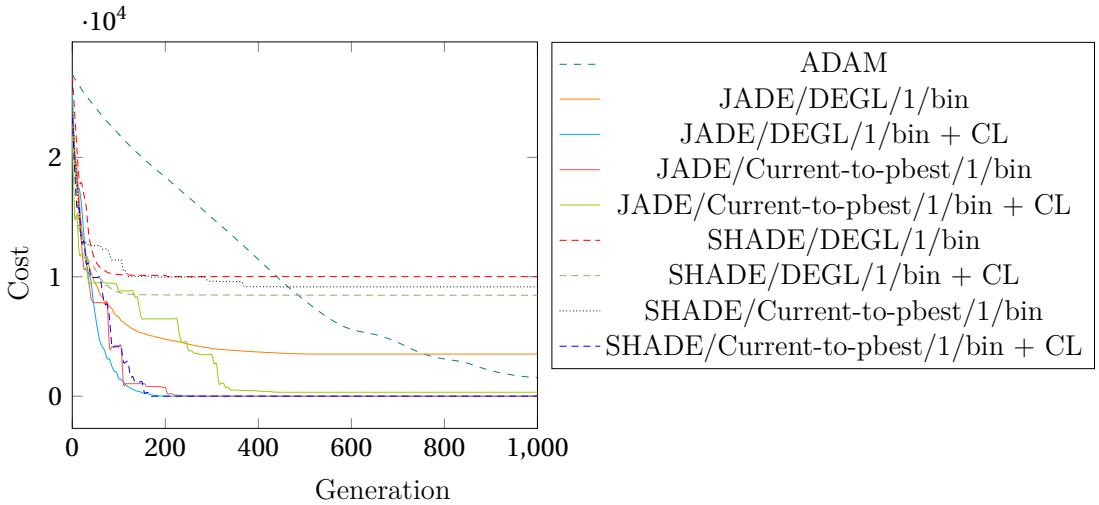


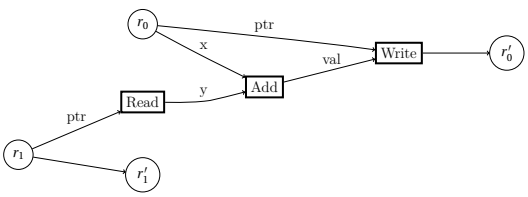
Figure 5.3: Costs convergence of DE variants with task **Reverse**.

5.3.2 Circuits

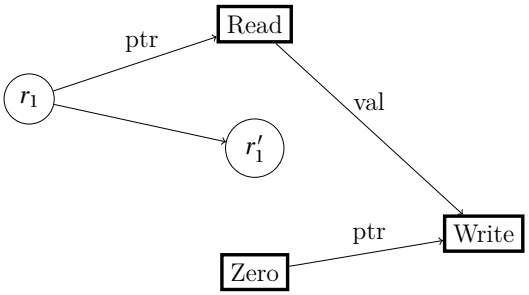
In the following sections, we present the circuits generated by the best solutions found by the system trained using DENN. For the sake of readability, only the necessary gates and registers are shown in the circuits. In the gates **Less-Than**, **Less-Equal-Than**, **Equality**, **Min** and **Max** the order of the parameters x and y is important; for the gate **Write**, the pointer and the value to write are indicated respectively with the labels *ptr* and *val*.

Overall, the circuits generated differ from those shown in [**?**]. In particular, we observed that the solutions found with different DE variants generate different circuits. The tables shown in the following contain the states of the memories and the registers at the beginning of the steps and the final memory obtained at the end of the execution.

5.3.2.1 Access



(a) The circuit generated by the controller from the steps ≥ 2, found with JADE/current-to-pbest/1/bin without the Curriculum learning.



(b) The circuit generated by the controller from the steps ≥ 2, found with JADE/current-to-pbest/1/bin with the Curriculum learning.

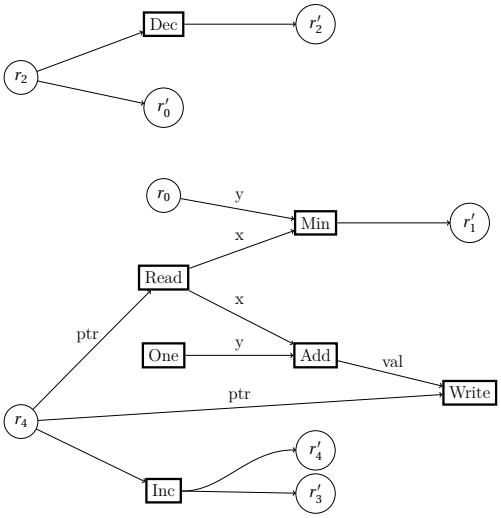
Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$r0$	$r1$	Read	Write
1	4	15	7	1	5	13	7	9	8	2	0	5	9	13	4	0	0	0	p:0	p:0 v:4
2	4	15	7	1	5	13	7	9	8	2	0	5	9	13	4	0	0	4	p:4	p:0 v:5
3	5	15	7	1	5	13	7	9	8	2	0	5	9	13	4	0	0	4	p:4	p:0 v:5
4	5	15	7	1	5	13	7	9	8	2	0	5	9	13	4	0	0	4	p:4	p:0 v:5
5	5	15	7	1	5	13	7	9	8	2	0	5	9	13	4	0	0	4	p:4	p:0 v:5
Final	5	15	7	1	5	13	7	9	8	2	0	5	9	13	4	0	0	4	×	×

Table 5.9: Behaviour obtained with the task Access with the controller found with JADE/current-to-pbest/1/bin without the Curriculum Learning.

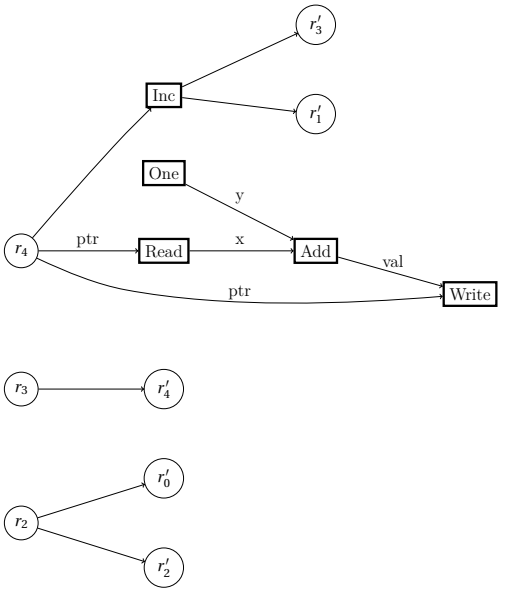
Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$r0$	$r1$	Read	Write
1	3	4	6	6	13	14	5	9	8	2	7	3	13	13	0	0	0	0	p:0	p:0 v:0
2	0	4	6	6	13	14	5	9	8	2	7	3	13	13	0	0	2	3	p:3	p:0 v:6
3	6	4	6	6	13	14	5	9	8	2	7	3	13	13	0	0	2	3	p:3	p:0 v:6
4	6	4	6	6	13	14	5	9	8	2	7	3	13	13	0	0	2	3	p:3	p:0 v:6
5	6	4	6	6	13	14	5	9	8	2	7	3	13	13	0	0	2	3	p:3	p:0 v:6
Final	6	4	6	6	13	14	5	9	8	2	7	3	13	13	0	0	2	3	×	×

Table 5.10: Behaviour obtained with the task Access with the controller found with JADE/current-to-pbest/1/bin with the Curriculum Learning.

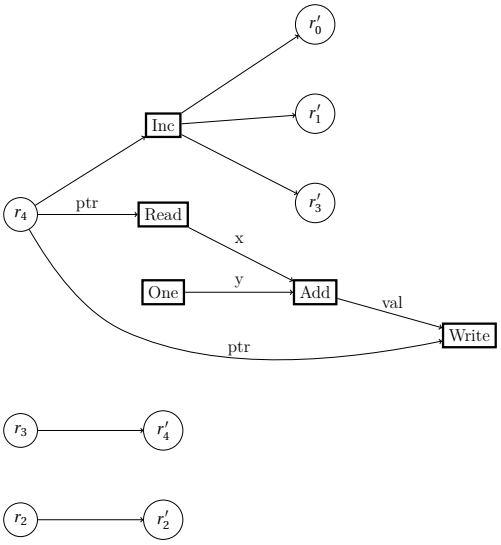
5.3.2.2 Increment



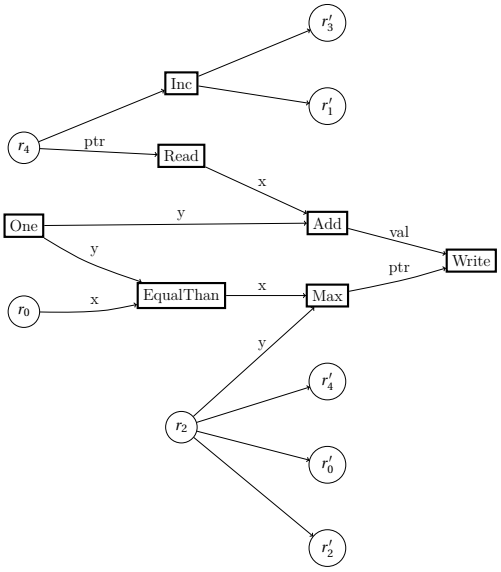
(a) The generated circuit in the first timestep.



(b) The circuit generated for every step ≥ 2 and $\leq \text{size}(A)$.



(c) The generated circuit for timestep $\geq$ size(A) and $\neq$ T.

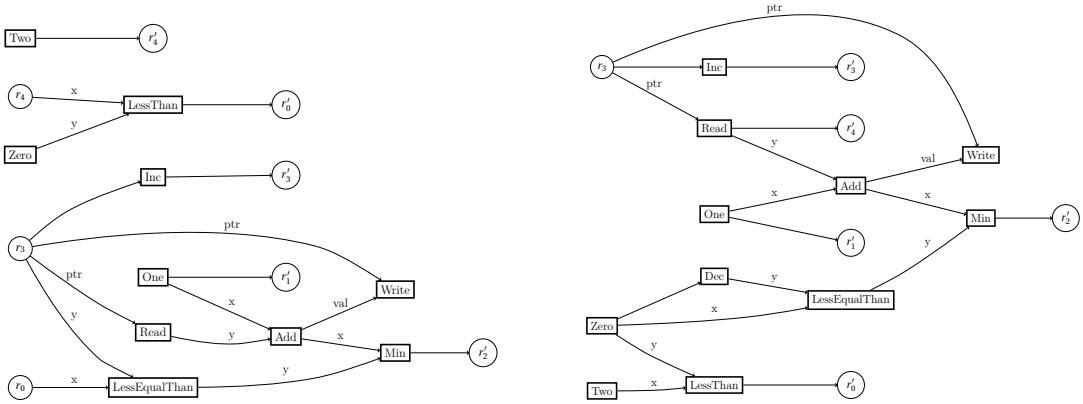


(d) The generated circuit for the step equals to T.

Figure 5.4: The circuits generated by the controller found with SHADE/current-to-pbest/1/bin with Curriculum Learning.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$r0$	$r1$	$r2$	$r3$	$r4$	Read	Write
1	4	15	12	9	14	14	9	10	6	2	0	0	0	0	0	0	0	0	0	0	0	p:0	p:0 v:5
2	5	15	12	9	14	14	9	10	6	2	0	0	0	0	0	0	0	0	15	1	1	p:1	p:1 v:0
3	5	0	12	9	14	14	9	10	6	2	0	0	0	0	0	0	15	2	15	2	2	p:2	p:2 v:13
4	5	0	13	9	14	14	9	10	6	2	0	0	0	0	0	0	3	3	15	3	3	p:3	p:3 v:10
5	5	0	13	10	14	14	9	10	6	2	0	0	0	0	0	0	4	4	15	4	4	p:4	p:4 v:15
6	5	0	13	10	15	14	9	10	6	2	0	0	0	0	0	0	5	5	15	5	5	p:5	p:5 v:15
7	5	0	13	10	15	15	9	10	6	2	0	0	0	0	0	0	6	6	15	6	6	p:6	p:6 v:10
8	5	0	13	10	15	15	10	10	6	2	0	0	0	0	0	0	7	7	15	7	7	p:7	p:7 v:11
9	5	0	13	10	15	15	10	11	6	2	0	0	0	0	0	0	8	8	15	8	8	p:8	p:8 v:7
Final	5	0	13	10	15	15	10	11	7	2	0	0	0	0	0	0	9	9	15	9	9	×	×

Table 5.11: Behaviour obtained with the task Increment in the training with SHADE/current-to-pbest/1/bin with Curriculum Learning.



(a) The generated circuit in the first timestep.

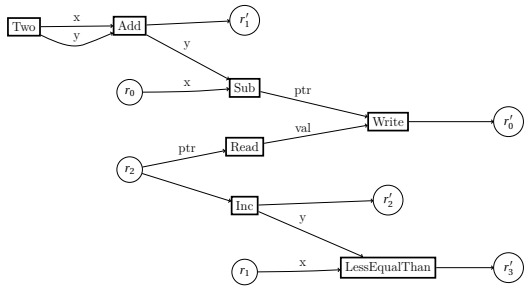
(b) The circuit generated for the timesteps ≥ 2.

Figure 5.5: The circuits generated by the controller found with JADE/current-to-pbest/1/bin without Curriculum Learning.

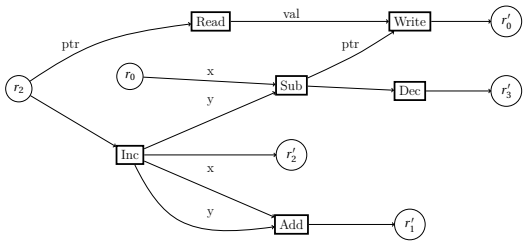
Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$r0$	$r1$	$r2$	$r3$	$r4$	Read	Write
1	5	2	5	8	1	4	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	p:0	p:0 v:6
2	6	2	5	8	1	4	11	0	0	0	0	0	0	0	0	0	0	1	1	1	2	p:1	p:1 v:3
3	6	3	5	8	1	4	11	0	0	0	0	0	0	0	0	0	0	1	1	2	2	p:2	p:2 v:6
4	6	3	6	8	1	4	11	0	0	0	0	0	0	0	0	0	0	1	1	3	5	p:3	p:3 v:9
5	6	3	6	9	1	4	11	0	0	0	0	0	0	0	0	0	0	1	1	4	8	p:4	p:4 v:2
6	6	3	6	9	2	4	11	0	0	0	0	0	0	0	0	0	0	1	1	5	1	p:5	p:5 v:5
7	6	3	6	9	2	5	11	0	0	0	0	0	0	0	0	0	0	1	1	6	4	p:6	p:6 v:12
Final	6	3	6	9	2	5	12	0	0	0	0	0	0	0	0	0	0	1	1	7	11	×	×

Table 5.12: Behaviour obtained with the task Increment in the training with JADE/current-to-pbest/1/bin.

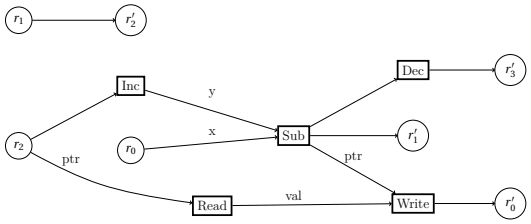
5.3.2.3 Reverse



(a) The circuit generated by the controller in the first timestep.



(b) The circuit generated for the $t \in [2, \text{size}(A))$ and for the odd timesteps $\geq \text{size}(A)$.

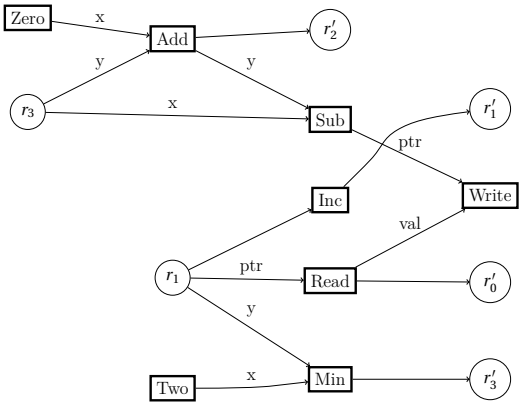


(c) The circuit generated for the even timesteps $\geq \text{size}(A)$.

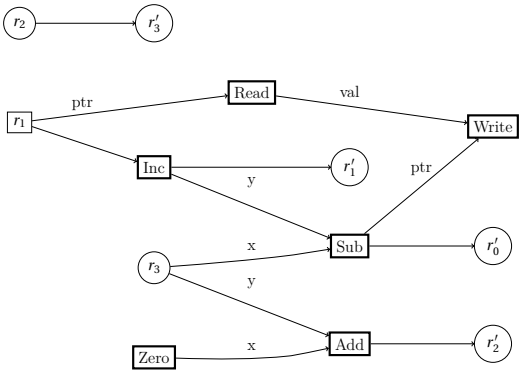
Figure 5.6: The circuits generated by the controller found with SHADE/current-to-pbest/1/bin with Curriculum Learning.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$r0$	$r1$	$r2$	$r3$	Read	Write
1	8	10	7	1	8	3	3	4	0	0	0	0	0	0	0	0	0	0	0	0	p:0	p:12 v:8
2	8	10	7	1	8	3	3	4	0	0	0	0	8	0	0	0	0	4	1	1	p:1	p:14 v:10
3	8	10	7	1	8	3	3	4	0	0	0	0	8	0	10	0	0	4	2	13	p:2	p:13 v:7
4	8	10	7	1	8	3	3	4	0	0	0	0	8	7	10	0	0	6	3	12	p:3	p:12 v:1
5	8	10	7	1	8	3	3	4	0	0	0	0	1	7	10	0	0	8	4	11	p:4	p:11 v:8
6	8	10	7	1	8	3	3	4	0	0	0	8	1	7	10	0	0	10	5	10	p:5	p:10 v:3
7	8	10	7	1	8	3	3	4	0	0	3	8	1	7	10	0	0	12	6	9	p:6	p:9 v:3
8	8	10	7	1	8	3	3	4	0	3	3	8	1	7	10	0	0	14	7	8	p:7	p:8 v:4
9	8	10	7	1	8	3	3	4	4	3	3	8	1	7	10	0	0	0	8	7	p:8	p:7 v:4
10	8	10	7	1	8	3	3	4	4	3	3	8	1	7	10	0	0	7	7	6	p:7	p:8 v:4
11	8	10	7	1	8	3	3	4	4	3	3	8	1	7	10	0	0	0	8	7	p:8	p:7 v:4
12	8	10	7	1	8	3	3	4	4	3	3	8	1	7	10	0	0	7	7	6	p:7	p:8 v:4
Final	8	10	7	1	8	3	3	4	4	3	3	8	1	7	10	0	0	0	8	7	×	×

Table 5.13: Behaviour obtained with the task Reverse with SHADE/current-to-pbest/1/bin with Curriculum Learning.



(a) The circuit generated by the controller in the first timestep.



(b) The circuit generated for the timesteps ≥ 2.

Figure 5.7: The circuits generated by the controller found with JADE/current-to-pbest/1/bin without Curriculum Learning.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$r0$	$r1$	$r2$	$r3$	Read	Write
1	8	15	3	14	13	13	14	11	0	0	0	0	0	0	0	0	0	0	0	0	p:0	p:0 v:8
2	8	15	3	14	13	13	14	11	0	0	0	0	0	0	0	0	8	1	0	0	p:1	p:14 v:15
3	8	15	3	14	13	13	14	11	0	0	0	0	0	0	15	0	14	2	0	0	p:2	p:13 v:3
4	8	15	3	14	13	13	14	11	0	0	0	0	0	3	15	0	13	3	0	0	p:3	p:12 v:14
5	8	15	3	14	13	13	14	11	0	0	0	0	14	3	15	0	12	4	0	0	p:4	p:11 v:13
6	8	15	3	14	13	13	14	11	0	0	0	13	14	3	15	0	11	5	0	0	p:5	p:10 v:13
7	8	15	3	14	13	13	14	11	0	0	13	13	14	3	15	0	10	6	0	0	p:6	p:9 v:14
8	8	15	3	14	13	13	14	11	0	14	13	13	14	3	15	0	9	7	0	0	p:7	p:8 v:11
9	8	15	3	14	13	13	14	11	11	14	13	13	14	3	15	0	8	8	0	0	p:8	p:7 v:11
Final	8	15	3	14	13	13	14	11	11	14	13	13	14	3	15	0	7	9	0	0	×	×

Table 5.14: Behaviour obtained with the task Reverse with JADE/current-to-pbest/1/bin without Curriculum Learning.

Conclusions

In this work a version of NRAM with Differential Evolution as optimization engine has been presented. In particular the DENN framework has been used instead of ADAM optimization algorithm.

We implemented two versions of the system: the first is a re-factoring and an extension of the project of Andrew Gibiansky, an implementation of NRAM written in Python with Theano and using ADAM algorithm; the second one is the new version proposed in this thesis, it is written in C++ and uses DENN as optimization framework.

As seen in the experiments, we concluded that Differential Evolution and DENN behave well with this type of machine and with large neural networks. As seen previously, in the Access, Increment and Reverse tasks, solutions that generalize perfectly are reached, also without the use of Curriculum Learning. This show that the research made by Differential Evolution is more effective respect to the one made by Gradient Descent and ADAM for this kind of problems.

Considering the data obtained so far we can conclude that, apart the case of JADE/DEGL/1/bin + CL, the best performing variants are the ones with the Current-to-pbest mutation method. Despite this, it is clear that the contribution of the Curriculum Learning is crucial in some cases like in the Increment task.

We observed also that the solutions found with DENN return more intuitive circuits compared to those in [**?**], showing that the exploitation and exploration done by DENN is more effective respect to those executed by Gradient Descent.

Future works

For the future we are going to complete all the experiments also with new tasks. Moreover, since the current structure of the gates is too stringent making the execution excessively static with the forced use of all them, one of our objective is to upgrade the NRAM with a more advanced system for a dynamic selection of the gates. Finally, we are planning to rewrite a enhanced NRAM model which does not require the differentiability condition; in this case we will loose the possibility to use the gradient descent method to train the controller, but the model could result simpler and its controller can be anyway trained with an evolutionary optimization algorithm.

List of Figures

Acknowledgements

This experience at the University and the development of this thesis were a long and hard paths, that without the support of many people would not be possible.

First of all, I would to thank my parents and all my family for being the best guidance that I could have in my life and my girlfriend Giulia for her support and for always being by my side.

I would make a big thank to my Professors and advisors Valentina Poggioni and Marco Baioletti and to my supervisor Gabriele Di Bari for their support in the creation and review of this thesis.

Finally, I would to thank my friends Simone, Eugenio, Giammarco, Lorenzo and all the others, which have had a big impact and played a big role in this journey.