UNIVERSITÀ DEGLI STUDI DI PERUGIA

Dipartimento di Matematica ed Informatica

Corso di Laurea Magistrale in Informatica

Tesi Di Laurea

Supervised Learning of Neural Random-Access Machines with Differential Evolution

Laureando

Valerio Belli

Matricola 283514

Relatore

Prof.ssa Valentina Poggioni

Prof. Marco Baioletti

Anno Accademico 2016/2017

Introduction

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

This is also translated in the model taking into account in this thesis, the Neural Random-Access Machine (NRAM) [**?**]. This model attempts to translate the concepts of pointer manipulation and dereferencing used in modern computers, through the addition of primitive operations. Obviously, this fact does not means that NRAM is better respect to other computational models. In fact, as specified in [**?**]

> [...] the number of timesteps that a given model has is highly limited, as extremely deep models are very difficult to train. 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 on new level of complexity. Due to the interesting results, we have tried to approach the optimization of this model in another way: trough Differential Evolution. Differently to backpropagation and gradient-based algorithm, Differential Evolution take into account the optimization problem in a different way. In fact, respects to backpropagation, DE takes in account the optimization problem interleaving phases of exploitation and exploration. So using the framework DENN, introduced in Section 3.2, we have completely re-developed the NRAM model and tested it, comparing the results with those resulting from the execution of ADAM.

The thesis is organized as follows: in Chapter 1 we introduce the NRAM model; in Chapter 2 we explain what is an Artificial Neural Network and how they can be trained with the Gradient Descent algorithms; in Chapter 3 we introduce Differential Evolution and DENN as alternatives to Gradient Descent algorithms; in Chapter 4 we speak quickly about the implementation and finally, in Chapter 5 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 the 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, the registers and the gates (modules). The central part is the controller which can be a feedforward or a Long-Short Term Memory neural network, the registers and the gates (named also **modules**) - the neural network is the only trainable part.

Let $N = \{0, 1, 2, \dots, I-1\}$ a set of integers, for an integer constant I, over the NRAM should work. Because the model in the [**?**] 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 $\sum\limits_{i=0}^{|N|} p_i = 1$. Hence, each register or memory cell plays the role of 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 the direct access in write to the registers, but 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

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

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

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

It is used always the same sequence of fourteen modules: Read (described in the Section 1.2), 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 1.2). 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 1.

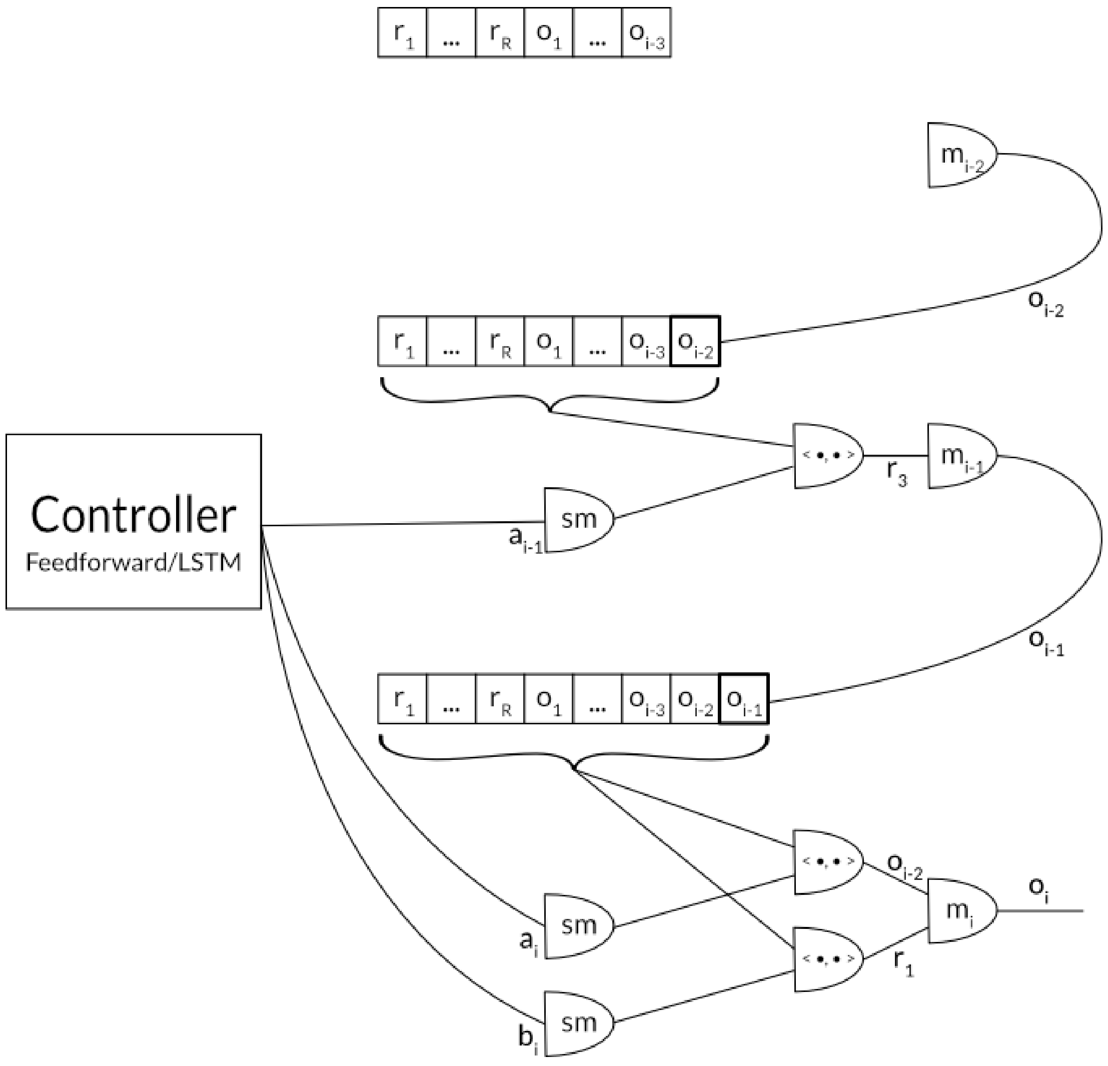


Figure 1.1: Extract of execution of NRAM circuit. The node with **sm** refers to a softmax activation function, the $< \cdot, \cdot >$ to a weighted average between the registers plus previous modules output with respect to the **softmax**$(\ldots)$. With the weighted average the controller selects the 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 execution of the NRAM starts at the Line 5 and continue for all the timesteps. At the Line 10, after the controller gets the input from the registers, this latter is executed generating a new configuration that is used at the Lines 11 and 12. The configuration indicates how the circuit is structured, i.e. for each gate indicates the input sources. Hence, in other words, the controller learns how to connect the gates

Algorithm 1 Execution of the NRAM without the memory

```
1: Let a controller (FFN or LSTM)
2: Let $R$ register
3: Let $M$ a set of modules
4: Let $T$ timesteps
5: for each timestep $t \in T$ do
6:     Controller gets as inputs the registers content
7:     if controller is a LSTM then
8:         Controller update its internal state
9:     end if
10:    Controller outputs one-shot the configuration of the NRAM
11:    The "fuzzy circuit" is executed
12:    The values of the registers is updated
13: end for
```

for resolve the problem on which it is trained. In other words, the inputs for a gates m_i is chosen by the controller from the set $\{r_1, \dots, r_R, o_1, \dots, o_{i-1}\}$ where:

- r_j is the content of the j^{th} register
- o_j is the content of the output of the j^{th} previous gate with respect to the current m_i

selected through a weighted average with a coefficient as follows

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

where $\mathrm{s}_i \in \mathbb{R}^{R+i-1}$ is a generic vector that indicates the input source for the i^{th} gate and the $\mathbf{softmax}(s_i)$ is called coefficient.
Hence, let $m_i \in M$ the i^{th} module in M, m_i is executed as follows

- Constant gate

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

- Unary gate

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

- Binary gate

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

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 $\{r_1,\dots,r_R,o_1,\dots,o_{i-1}\}$ and used later for the subsequent modules. Since the registers contain probability distributions, the

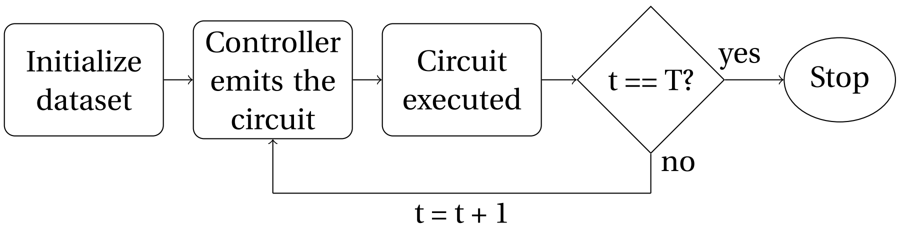


Figure 1.2: High level execution of the NRAM without the Curriculum Learning activated. Here, t is a variable which takes into account the current timestep.

inputs of the modules are also probability distributions because 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:

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

At the same time of the emission of the vectors $a_i, b_i \in \mathbb{R}^{R+i-1}$ are also emitted the vectors $c_i \in \mathbb{R}^{R+|M|}$ for $i = 1,2,\dots,R$, 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 12, the registers content is updated as follows:

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

As stated previously, at the Line 6 the controller get some input from the registers. This input could be all the registers content, but in this way the controller's parameters would depend directly on M. This case is undesirable because the neural

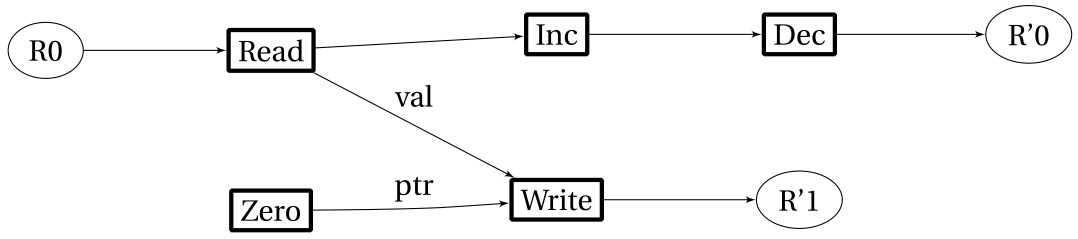


Figure 1.3: Example of NRAM circuit. With the circles are represented the registers and with the rectangles are represented the various circuits. The registers with the apex are those that are modified in the timestep. The labels x and y represent the values order for the gates (obviously if it is violated,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

network could learn the specific problem on which it trains, preventing it to generalize to different size of M.

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 $\mathscr{M} \in \mathbb{R}^{|N|}_{|N|}$, where a value $\mathscr{M}_{i,j}$ 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 both the pointer is an integer or a probability distribution, what change is the system with which the module access the memory cell. More precisely, if the pointer $p \in \mathbb{R}^{|N|}$ is a probability distribution represented as a column vector, then the module returns $\mathscr{M}^T p$.
- **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.

Moreover, the NRAM use a system with which can decide if the execution can be terminated. Recalling that the execution is divided T timesteps, in each of them the

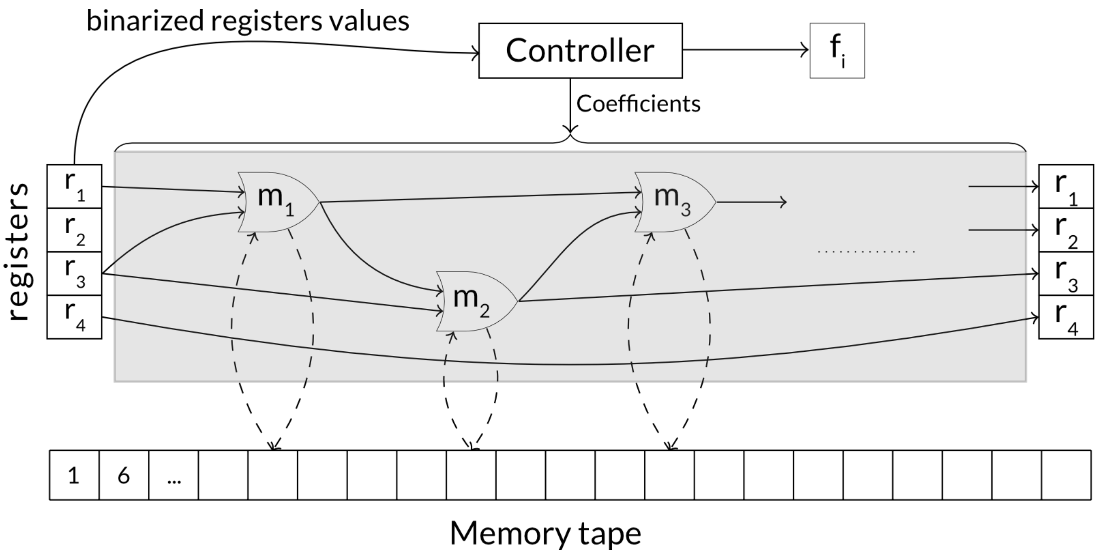


Figure 1.4: Execution of the NRAM memory augmented architecture with $R = 4$ registers. As can be seen the controllers get a registers "binarized" version, with which coefficients and the f_i are produced. As stated previously in subsection 1.1.1, the coefficients are used to select connections that here are represented with solid lines. Instead, with the dashed connections are represented the interactions by the **Read** and **Write** modules with the memory.

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. Thus, starting from this we have that the probability that the execution is not finished in the previous timestep is $\prod_{i=1}^{t-1}(1-f_i)$ and the probability 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 timestep 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 ver-

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

sion, as the expected negative log-likelihood of producing the correct output, i.e.

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

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** used 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 elements 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

[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.

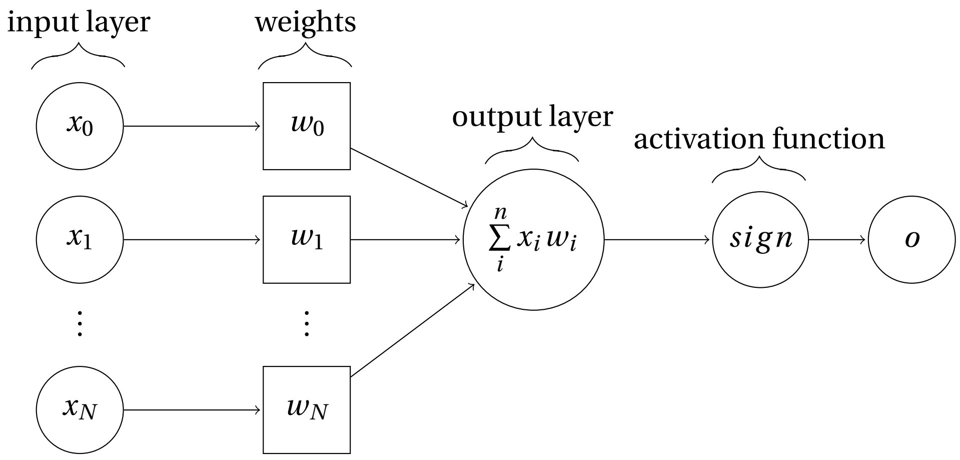


Figure 2.1: Perceptron model illustration

focus the attention to feedforward neural network model, explaining firstly the Perceptron model.

2.2 Perceptron

The Perceptron model is the basic and simplest type of Neural Network, proposed firstly in [**?**]. This model is composed only by two types of nodes called **input nodes** and **output node**. As we can see in the Figure 2.1, 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). So recalling some math, the output of the Perceptron model can be expressed in the following way:

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

where $i = 1, \ldots, N$ and N is the cardinality of the vector $\mathbf{x}$ and the $\mathbf{g}$ acts as the activation function. The training of the Perceptron consists in recompute (the key passage at the step 7 of Algorithm 2), 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 2

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

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.

Algorithm 2 Perceptron learning algorithm [**?**]

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

2.3 Multi-Layer Perceptron (MLP)

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

- It cannot handles a domain $(\mathbf{x}, y) \in D$ with more than two classes, because it divides the input data in only two boundaries, as it can be seen in the figure 2.2, so more dimensions can not be handled

[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.

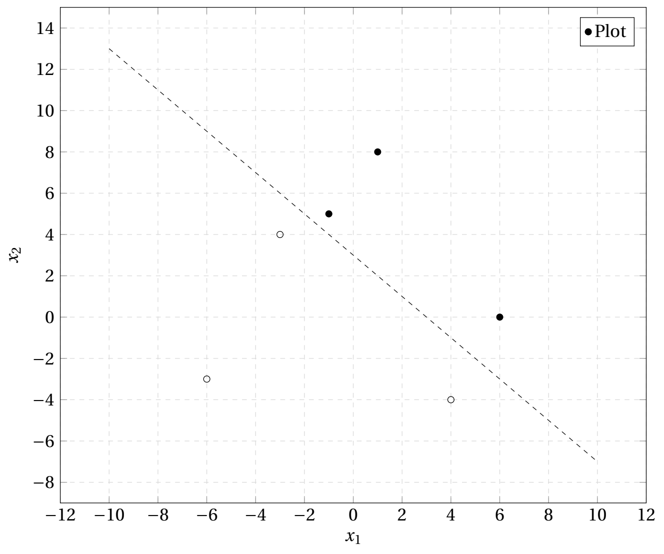


Figure 2.2: Representation of perceptron's decision separation.

- It can not converge if the data is not linearly separable and this leads to reduce the use of Perceptron in many few case (trivially, when the data is linearly separable)

As 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 perceptron, are called *feedforward* because the information flows through the input layer, that contains the data from 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 layer are associated different functions, called **activation functions**, that allow the layers to produce non-linear output values. Thus, in an other perspective, a feedforward can be seen as a chain of activation functions. From this

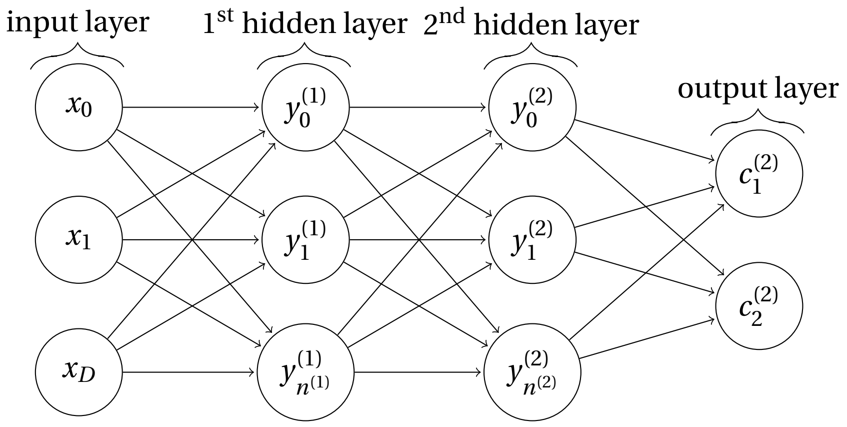


Figure 2.3: Graph representation of feedforward neural network with (2)-layer, with D input units and 2 output units (class). For simplicity of representation all the labels w_{ij} associated to the edges, that represent the weights/parameters of the neural networks, are omitted.

description we can see that the major difference between single-layered Perceptron 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 does not have *a priori* the informations regard the desired output of all the hidden layers, so we can not update the weights as we have seen in the Perceptron.

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:

$$\sigma(x) = x$$

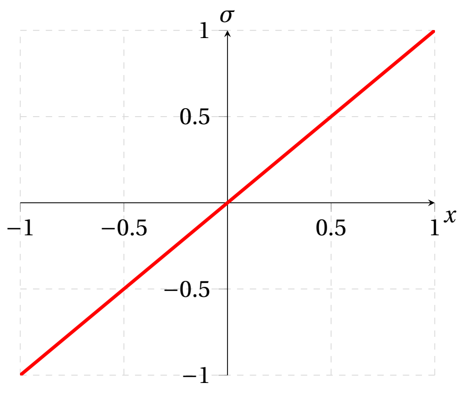


Figure 2.4: Linear activation function.

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

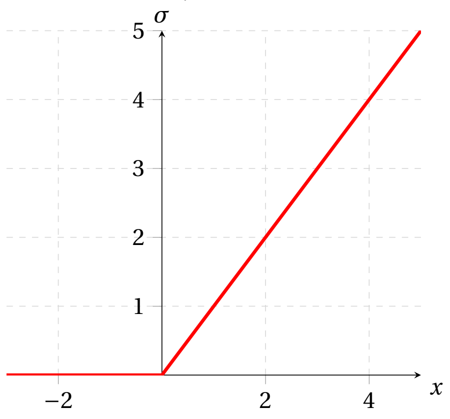


Figure 2.5: ReLu activation function.

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

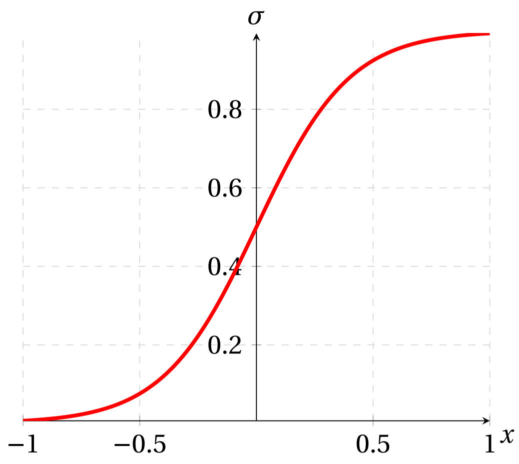


Figure 2.6: Sigmoid activation function.

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

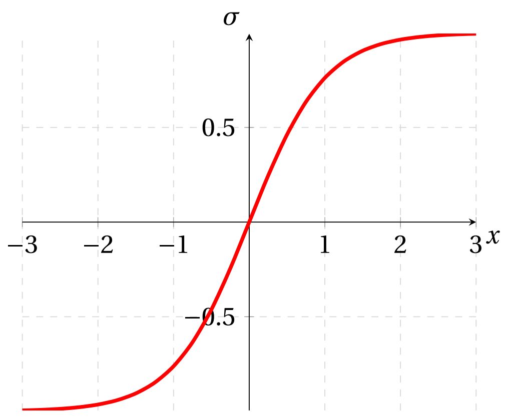


Figure 2.7: Tanh activation function.

2.4 Artificial Neural Network training

The training of ANN generally follows two main steps: the initialization of the weight of the connections and the re-compute 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 $D = \{(\mathbf{x}_i, y_i) \mid i = 1, \dots, N\}$ be the set of examples.
 2: Let $\lambda$ 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 $\delta$ 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 $\delta$ and the learning rate $\lambda$
10:         end for
11:     end for
12: until Convergence is met
```

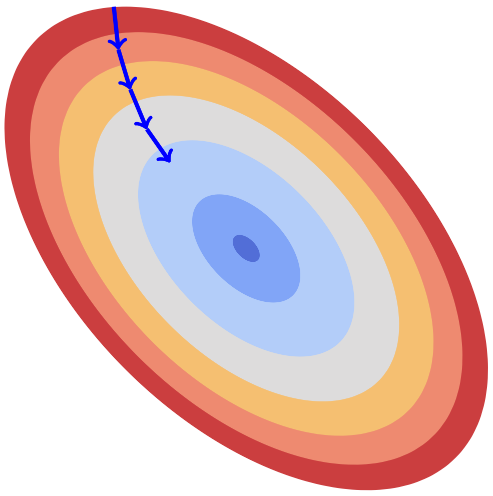
2.4.1 Gradient based optimization algorithms

Once the gradients are computed according to the objective function, for example with the backpropagation technique presented in Section 2.4.1.1, they can be used to re-compute the associated parameters. To do this exist some optimization algorithms that implement gradient descent. These algorithm have as objective the

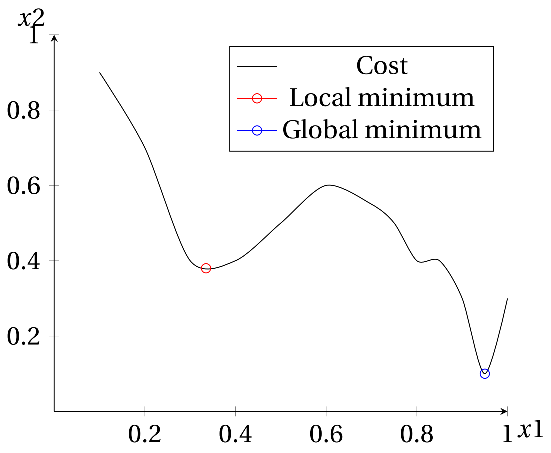
[3]In specific we speak of **cost** or **loss** function if the objective is to minimize this latter.

[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.

research of the global minimum of the cost function with a research that, with a similitude, 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 research of a better pa-



(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. (Yes, this figure seems more similar to the planet Saturn.)



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

rameters configuration not only works with the discovered 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 2.4.1 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 **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 dif-

[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.

ferences 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 neurons of the layer$\}$ the set of FNN hidden layers and $A = \{\alpha^{(i)} \mid i = 1, \ldots, M\}$. Then selected the sample $(\mathbf{x}_i, y_i)$ the parameters, i.e. the weights, for the first level are computed in this way

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

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

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

where the $h^{(j-1)}$ are the computed weights of the previous layer of the layer $h^{(j)}$, up to the output layer where we have

$$\hat{y} = \alpha^{(M)}(\sum_{n \in a^{(M-1)}} a_n^{(M-1)} o + b^{(M)}) \tag{2.4.1.11}$$

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

Backward propagation Once the objective function is computed the backward propagation can be executed. A strictly 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 make it possible to calculate the error for each node starting from the output layer in a backward mode, i.e. for each layers, starting from the output layer, it is calculated the δ for all the weights which is contribute to the ANN errors, named also gradient. Make it clearer with an example where we calculate the gradient δ for a generic parameter of the FNN, so let **TSS** the loss function in 2.4.1.1 and w_{ij} a parameter[7] for the neuron j^{th} of l arriving from the i^{th} neuron of the previous layer $l-1$. Starting 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 $N_i = \{u, v, \dots, w\}$ 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

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

[7]The connection between two neurons.

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 2.4.1.1

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

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 2.4.1.1 and 2.4.1.1, 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 and greater ANNs, like NRAM.

Chapter 3

Differential evolution and DENN

In the first part of the Chapter we make an overview to the iterative algorithm Differential Evolution, moving later to the mutation and crossover strategies that we have used in the experiments. Finally, we speak about 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 the training of 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.

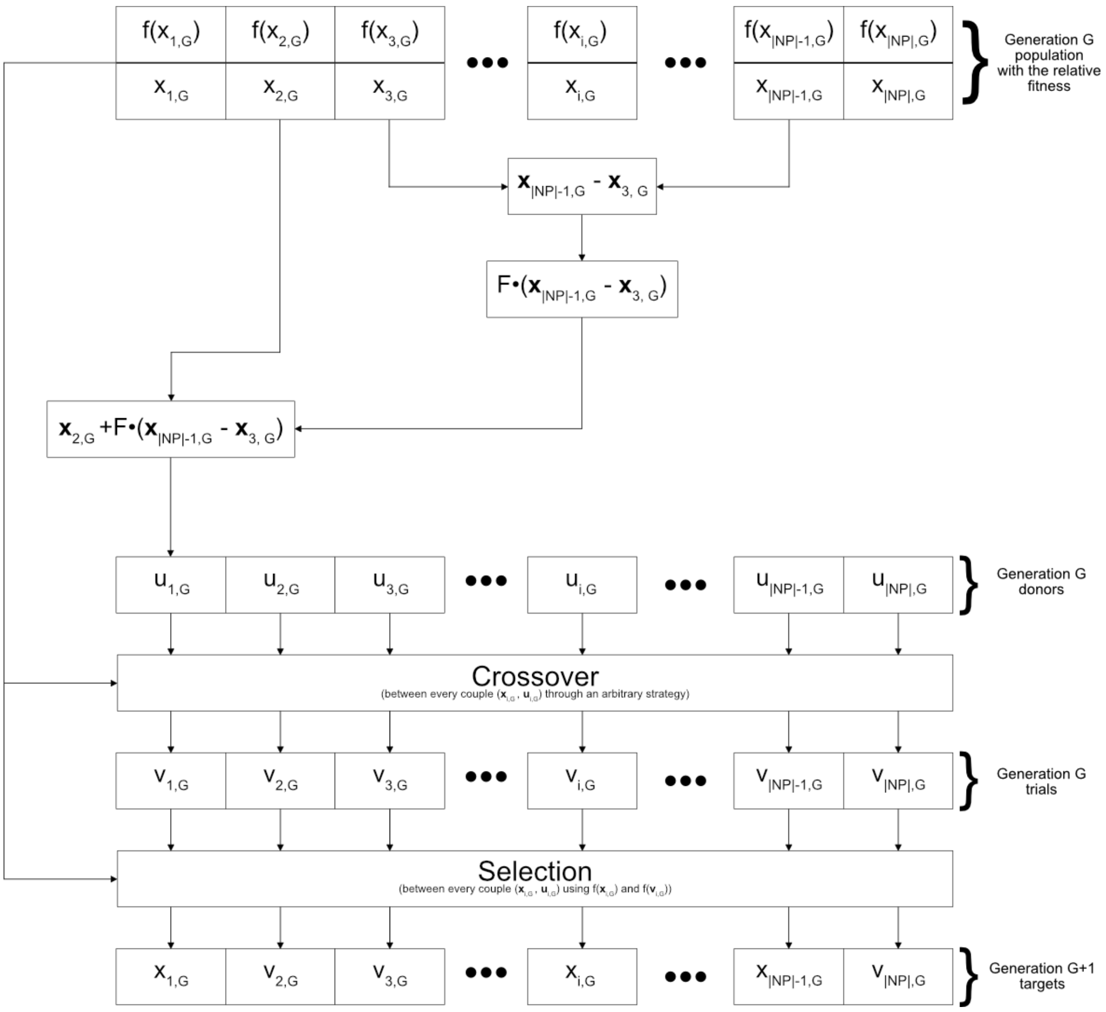
Differential Evolution is a parallel iterative direct search metaheuristic which utilizes NP D-dimensional numerical vectors

$$x_{i,G},\ i = 1,\ \ldots,\ NP \qquad (3.1.0.1)$$

[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.

called **population** formed by **individuals**. Every individual is manipulated for **G** generations, where the population is not reduced 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.

As stated previously, the algorithm goes on for G generations where in each of these



f(x1,G) f(x2,G) f(x3,G) f(xi,G) f(x|NP|-1,G) f(x|NP|,G)
x1,G x2,G x3,G xi,G x|NP|-1,G x|NP|,G
Generation G population with the relative fitness
x|NP|-1,G - x3,G
F•(x|NP|-1,G - x3,G)
x2,G +F•(x|NP|-1,G - x3,G)
u1,G u2,G u3,G ui,G u|NP|-1,G u|NP|,G
Generation G donors
Crossover
(between every couple (xi,G , ui,G) through an arbitrary strategy)
v1,G v2,G v3,G vi,G v|NP|-1,G v|NP|,G
Generation G trials
Selection
(between every couple (xi,G , ui,G) using f(xi,G) and f(vi,G))
x1,G v2,G v3,G xi,G x|NP|-1,G v|NP|,G
Generation G+1 targets

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

are generated three set of individuals called, respectively, **targets**, **donors** and **trials**.

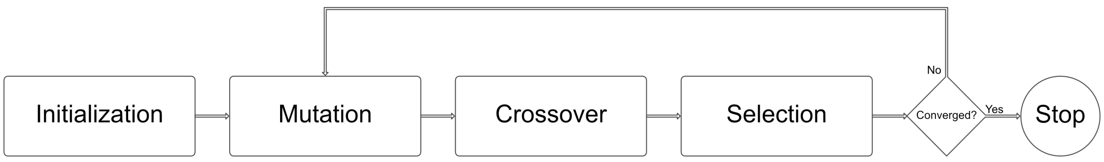


Figure 3.2: Differential Evolution flow diagram.

The targets set is formed by individuals which represent the candidate solutions and from which is created the donors (mutants) set mixing the targets. The targets and donors sets are mixed with some crossover method which creates the trials set. After this, the trials is compared one-by-one to the targets with the fitness function - the better are selected creating the targets set of the next generation.

Summing up, after the targets set initialization, the step executed in every DE generation are:

- **Mutation**: Let target x at the generation G, a mutant is generated combining x through a summation with some other targets combined with some strategy and scaled through a global real valued user defined constant F. As 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 $r_1, r_2, r_3 \in \{1, 2, \ldots, NP\}$ and must be each other different.

- **Crossover**: Because the mutation itself could leads to the creation of equal mutants, this step is introduced. As stated previously, the crossover does nothing else than mixing up the targets with the mutants (donors) with some strategy, generating the trials. For example, let the target vector $x_{i,G}$ and the mutant $v_{i,G+1}$ the **bin** strategy work as follows

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

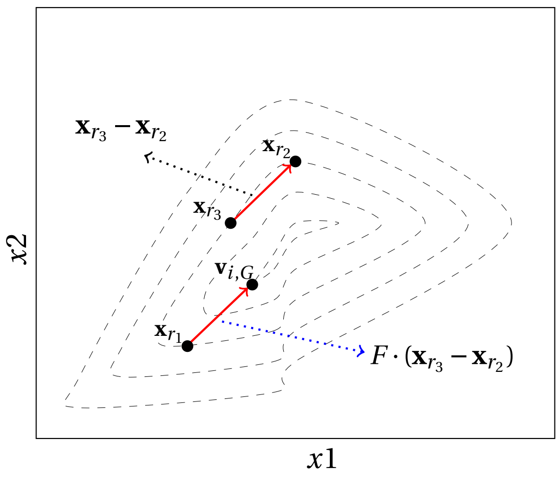

$\mathbf{x}_{r_3} - \mathbf{x}_{r_2}$
$\mathbf{x}_{r_2}$
$\mathbf{x}_{r_3}$
$\mathbf{v}_{i,G}$
$\mathbf{x}_{r_1}$
$F \cdot (\mathbf{x}_{r_3} - \mathbf{x}_{r_2})$
$x2$
$x1$

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 $\mathbf{x}_{r_1}$.

for $i = 1,2,\dots,D$. The function *randb*(j) generate a real valued number for the j^{th} parameter according to binomial distribution, CR $\in [0,1]$ is the global user defined crossover constant used as a threshold and *rnbr*(i) is a function that generate randomly an index which ensures that is selected at least one parameter of the mutant v_i.

- **Selection**: After the trials set is generated is made a comparison with respect to the targets set for each vector, i.e. the target $x_{i,G}$ is compared with respect to the trial $u_{i,G+1}$ using the fitness function. Hence, if the target have a smaller cost with respect to the trial, than it is retained for the next generation targets set and vice-versa.

During the algorithm explanation is introduced one strategy both for the mutation and crossover step, but they are not the unique even regard the optimization algorithm. Hence, in order to classify all the variants it's used the notation $DE/x/y/z$ 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 DE's different configurations made available by the framework DENN (3.2). 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 is suggest in [**?**] to set the $F \in [0.4, 0.95]$ and $CR \in (0, 0.2)$ if the function is separable and $CR \in (0.9, 1.0)$ when the function's parameters are dependent, remains always the fact that is a problem dependent decision.

Hence, trying to resolve this problem in [**?**] is introduced JADE. Briefly, this DE variant lift the user from the duty of selecting the best combination of F and CR constants. This is done with a parameter adaptation system which research 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} the CR's mean, initialized previous 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 the generation ending the best CR_is, i.e. the CRs associated to the trials v better than the targets x, are added to the set S_{CR}, i.e. the set containing the CR_i that is successful. After this, 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 is a positive constant between 0 and 1 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 F's mean, initialized previous the first generation to 0.5. Hence, for each generation and for each target individual, is generated a F_i 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 $F_i \le 0$, 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 is stored all the targets that have failed the selection process. Once the memory is full and is necessary to add a new failing individual, a randomly one is deleted.

3.1.1.2 Success-History based Adaptive DE (SHADE)

The JADE's main problem 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.

SHADE introduced in [?] aims to toughen 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, named *historical memories,* M_{CR} and M_F where are stored the constants that have well performed in the past. Formally,

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

let M_F and M_{CR} two memories with H cells initialized to 0.5. Similarly to JADE, the constants are generated in each generation for all the individuals like follows:

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

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

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, is initialized to 1 and incremented by one at the end of each generation. When $k > H$ is set already 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 used in a EA algorithm, in this case Differential Evolution, is very important. In fact, a small population result in a faster convergence, but this can also leads the algorithm to sake in a local minimum. Contrarily, a bigger population increments the algorithm chance to converge to a global minimum, but more slowly due to the searching space increment. Hence, the population is another problem dependent parameter to be optimize.

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

Let NP^{init} the initial population and NP^{min} the minimum possible population 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 respect the previous two methods, DEGL, introduced in [?], make 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 defined a neighbors 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 $r_1 \neq r_2 \neq i$. The 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, since being a greedy strategy which use prevalently the information of the best solution could leads the search to converge to local minimum. Instead in Current to pbest this is information can impacts partially on the search, reducing in this way the possibility about search algorithm to stop in a local minimum.

This strategy exists in two version which differentiate in the use of an auxiliary memory. The base version 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 between the $100p\%$ of the current population NP_G with $p \in (0,1]$ and F_i is the mutation factor which is associated to every target individual and managed as in JADE and SHADE.

Denote $\mathbf{A}$ the auxiliary memory where are stored the targets that fail 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 3.1.2.4 and $\tilde{\mathbf{x}}_{r_2,G}$ is selected, instead, in $\mathbf{P} \cup \mathbf{A}$. To note that if $\mathbf{A}$ size exceed a certain user defined 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 is associated a p_i who is set 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 by Zhang and Sanderson in [**?**].

3.1.3 Crossover strategies

3.1.3.1 Bin

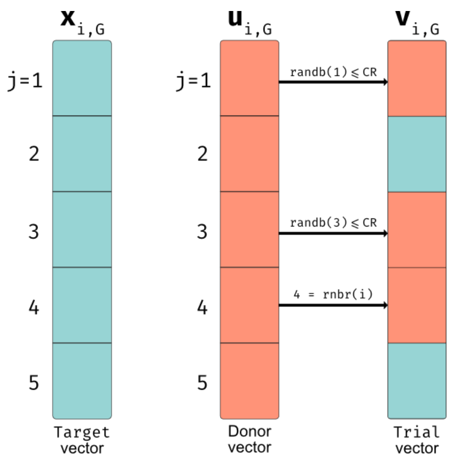


Figure 3.4: Bin crossover strategy illustration with $D = 5$.

Bin is classic form of crossover and works as follows:

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

where $randb(i) \in [0,1]$ is a uniform random generator and $rnbr(j) \in [1, D]$ so that at least one parameter of mutant $\mathbf{v}_{i,G}$. the figure 3.4 gives a visual representation of the strategy.

3.1.3.2 Exp

Exp is an alternative of the classic Bin. It starts choosing a random $n \in [1, D]$, used as an initial point in the target value where the parameters substitution starts, and a

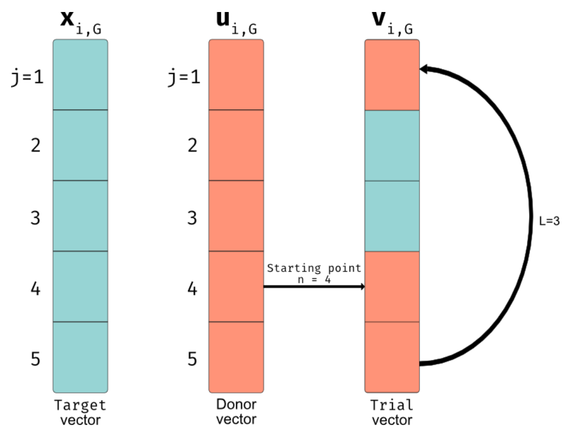


Figure 3.5: Exp crossover strategy illustration with a $N = 4$ as starting point and $L = 3$ parameters to substitute.

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 3.5

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 is initially created as a TensorFlow extension due to the performance and simplicity offered by this latter, but now 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 does it work

DENN is a framework that aims to help the developer to construct a system for 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 things, 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 3.2.

However, what really changes in DENN with respect to the standard Differential Evolution is how the population evolves. As stated in the Chapter 3, Differential Evolution bases its functioning also on the existence of a population of vectors, called *NP*, that is paralleled evolved by the 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 bi-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

[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.

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^{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
cumpter_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
n_registers	Registers to use
time_steps	Execution timesteps
gates	Gates to use
step_gen_change_difficulty	Number of gen where the same difficulty is used

Table 3.2: Some arguments used in a DENN's configuration file.

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, to which some parts of NRAM and a parametrization system are missing, an implementation in C++ of the NRAM using DENN and additional implementation[3] of the NRAM without the training phase, used to test the generalization capacity of the discovered ANNs.

Preface Because some of the tasks presented in Chapter 5 are too complex to be learn by NRAM, we have also used some of the technique that have been used in the original paper [**?**].

Gradient clipping The size of the model can be extremely huge 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 Theano im-

[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

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

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

Entropy As for [?], we also noticed that the network can fix the pointer in some value. Although it could be an advantage in some cases, however, if this happens too early in the training, it could forces 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 for the following coefficient that decreases over time

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

where e is the entropy coefficient, d is the entropy coefficient's decay and t is the timestep. After that, the computed value is subtracted from timestep cost.

Curriculum learning As noticed by [?] and [?], curriculum learning is crucial for train very deep neural networks. Hence, as in the original paper we used the curriculum learning from [?].

For each task we have defined manually a set of increasing difficulties, where a difficulty d is defined as a tuple containing the length of working sequence and the number of timesteps of running. During the training, a difficulty 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.

[4]Let $\mathbf{x} = [x_1, x_2, \dots, x_n]$, 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.

- with probability 25%: pick d randomly from the set $[1, D+e]$ according to a uniform distribution, with e 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 increasing of difficulty happens when the error rate $1 - \frac{c}{m}$, where c represents the correct modified cells and m the cells that should be modified, goes below an hyperparameter λ, decided by the user. Furthermore, we ensure that the successive increasing of the difficulty is separated by some number of generations.

4.1 How we did it

The implementation of NRAM where it is trained is similar between the two solutions, differentiating only for the training algorithm and for some specific technique (for example, Gradient Clipping that in the implementation with DENN is useless). Hence, leaving out these specifics of the framework we speak only about the NRAM implementation which is presented for simplicity as pseudocode[6].
The principal block of NRAM can be seen in Algorithm 4. 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 8 - in every timestep the network releases all the coefficients, i.e. the circuit connections, at Line 9. At Line 10 is executed the circuit, with the Algorithm 5 - it returns the willingness of terminate the execution with which is computed from Line 12 to 24 the timestep cost, added later to the total cost. The "magic" of NRAM happens in Algorithm 5, where the circuit is executed. From Line 4 to 18, 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 19 to 23, the registers are updated. As it can be seen in the Lines

[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.

9, 13, 14 and 21, the selection of the input of a gate and the new content of the register is made through the function in the Algorithm 6 which do a weighted average. An example of gate is visible in Algorithm 7 - 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 capacity of the discovered ANNs is showed in Algorithm 8. 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 11 is written to file the circuits generated during the execution. At Line 14 and at Line 15, 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.

4.1.0.1 Datasets

The datasets used during the training are generated programmatically for each tasks with a procedure. In each procedure are generated the input dataset, i.e. the initial memory manipulated during the execution of the NRAM, the expected dataset, i.e. the desired memory, and the cost mask, i.e. a vector which has the integer one in the positions of interest of the memory and zero in the others. 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 “Desired Memory”. The datasets are generated only at the start 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.

```
function NRAM(NeuralNetwork, BatchSize, Regs, IMem, OMem, CostMask, MaxInt, T, Gates)
    Cost = 0.0
    ProbIncomplete = 1.0
    CumProbComplete = 0.0;
    $p_T$ = 1.0
    EntropyCoeff = 0.1
    EntropyDecay = 0.999
    for $t \in [1, \ldots, T]$ do
        Conf = NeuralNetwork.predict(Regs.ProbZero())
        {$f_i$, Regs, IMem} = RunCircuit(Gates, Conf, Regs, IMem, MaxInt)

        if t == T then
            $p_T$ = 1 - CumProbComplete
        else
            $p_T$ = $f_i$ * ProbIncomplete
        end if

        CumProbComplete = CumProbComplete + $p_t$
        ProbIncomplete = ProbIncomplete * $(1 - f_i)$

        EntropyWeight = EntropyCoeff $*$ EntropyDecay$^t$
        EntropyCost = EntropyCost+Entropy(IMem) * EntropyWeight

        Cost += CalculateCost(InMem, OutMem, CostMask) - EntropyCost
    end for

    return Cost
end function
```

Algorithm 5 Pseudocode of the run circuit function.

```
function RunCircuit(Gates, Conf, Regs, IMem, MaxInt)
    RegsAndOutput = Copy(Regs)

    for each $g \in$ Gates do                                  ▷ Run of the circuit
        if g.isConstant( )then
            GateResult = g.Value
        else if g.isUnary( )then
            Coeff = GetGateCoefficient(Conf, g)
            InputValue = Avg(RegsAndOutput, Coeff)
            GateResult = g.execute(InputValue, IMem, MaxInt)
        else
            (CoeffA, CoeffB) = GetGateCoefficient(Conf, g)
            InputValueA = Avg(RegsAndOutput, CoeffA)
            InputValueB = Avg(RegsAndOutput, CoeffB)
            GateResult = g.execute(GateValueA, GateValueB, IMem, MaxInt)
        end if
        RegsAndOutput = Concatenate(Regs, GateValue)
    end for

    for $i \in [1, \dots, |\text{Regs}|]$ do                  ▷ Update of the registers
        Coeff = GetGateCoefficient(Conf, i)
        NewValue = Avg(RegsAndOutput, Coeff)
        Regs[i] = NewValue
    end for

    return (Conf[|Conf| - 1], Distributions)
end function
```

Algorithm 6 Pseudocode of the avg function

```
function Avg(RegsOut, Coefficient)
    return RegsOut * Coefficient
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 Add's output
3:     for $i \in [0, \text{MaxInt} - 1]$ do
4:         for $j \in [0, \text{MaxInt} - 1]$ 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 $s \in [1, \ldots, \text{BatchSize}]$ do
5:             SampleIMem = IMem[s]
6:             SampleOMem = OMem[s]
7:             SampleRegs = Regs[s]
8:             for $t \in [1, \ldots, \text{Timestep}]$ 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 this chapter we present in the first part the problems on which the NRAM is trained, finishing by comparing the results and by presenting 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 $k,\ A[0],\ \ldots,$
$\mathbf{A}[n-1],\ NULL$ and the network should replace the first memory cell with $\mathbf{A}[k]$. An example is visible in Figure 5.1.

Initial memory									
4	5	1	4	$\underline{7}$	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

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 **A**[0], …, **A**[$n-1$], *NULL* and the expected output is **A**[0] + 1, …, $A[n-1]+1$. An example is visible in Figure 5.2.

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

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, **A**[0], …, **A**[$n-1$] where p points to one element after **A**[$n-1$]. The expected output is **A**[0], …, **A**[$n-1$] at positions p, …, $p+n-1$ respectively. An example is visible in Figure 5.3.

Initial memory									
5	5	1	4	7	<u>0</u>	0	0	0	0

Desired memory									
5	5	1	4	7	5	1	4	7	0

Cost mask									
0	0	0	0	0	1	1	1	1	1

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 5.4.

Initial memory									
5	5	1	4	7	<u>0</u>	0	0	0	0

Desired memory									
5	5	1	4	7	7	4	1	5	0

Cost mask									
0	0	0	0	0	1	1	1	1	1

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.

5.1.5 Swap

Given two pointers p, q and an array $\mathbf{A}$, swap elements $\mathbf{A}[p]$ and $\mathbf{A}[q]$. Input is given as $p, q, \mathbf{A}[0], \ldots, \mathbf{A}[p], \ldots, \mathbf{A}[q], \ldots, \mathbf{A}[n-1], 0$. The expected modified array $\mathbf{A}$ is: $\mathbf{A}[0], \ldots, \mathbf{A}[q], \ldots, \mathbf{A}[p], \ldots, \mathbf{A}[n-1]$. An example is visible in Figure 5.5.

Initial memory									
5	**7**	1	4	7	3	6	8	1	0

Desired memory									
5	**7**	1	4	7	8	6	3	1	0

Cost mask									
0	0	1	1	1	1	1	1	1	1

Table 5.5: Expected behaviour with the task Swap - the first two value is the pointer in the sequence A that should be swapped.

5.1.6 Permutation

Given two arrays of n elements: P (contains a permutation of numbers 0, $\ldots$, $n-1$) and **A** (contains random elements), permutate **A** according to P. Input is given as a, $P[0]$, $\ldots$, $P[n-1]$, **A**[0], ..., **A**[$n-1$], where a is a pointer to the array **A**. The expected output is **A**[$P[0]$], $\ldots$, **A**[$P[n-1]$], which should override the array P. An example is visible in Figure 5.6.

Initial memory									
5	2	1	0	3	3	6	8	1	0

Desired memory									
5	8	6	2	1	3	6	8	1	0

Cost mask									
0	1	1	1	1	0	0	0	0	0

Table 5.6: Expected behaviour with the task Swap - the first two value is the pointer in the sequence A that should be swapped.

5.1.7 Sum

Given pointers to 2 arrays **A** and **B**, and the pointer to the output o, sum the two arrays into one array. The input is given as: a, b, o, **A**[0], $\ldots$, **A**[$n-1$], G, **B**[0], $\ldots$, **B**[$m-1$], G, where a points to first element of **A**, b points to the first element of **B**, o points to first element of output array and G is a special guardian value. The **A** + **B** array should be written starting from position o. An example is visible in Figure 5.7.

Initial memory											
3	**6**	**9**	$\underline{3}$	5	0	$\underline{4}$	6	0	$\underline{0}$	0	0

Desired memory											
3	**6**	**9**	$\underline{3}$	5	0	$\underline{4}$	6	0	$\underline{7}$	11	0

Cost mask											
0	0	0	0	0	0	0	0	0	1	1	0

Table 5.7: Expected behaviour with the task Sum - the values of the two arrays are summed up and copied starting from the index *o*.

5.2 The generalization problem

Generalization means the capacity of a neural network to recognize patterns never seen before in new examples. In a classic classification problem this means that the neural network should recognize the patterns in the features 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 over memory with limited size and timesteps, with "generalization capacity" means that the controller should learns to create the right circuit also for examples[1] with memory of different sizes and timesteps. As stated previously in the Paragraph 4, the Curriculum Learning is used to boost the training making the neural network more robust - in fact, the training over a specific memory size and timesteps could leads the objective function to have a value equal to zero, but this does not means that the neural network has learnt to generalize.

5.3 Results

In the experiments we have follow what is do in the paper [**?**], training of NRAM with DENN and Theano on "easy" and "hard" tasks. The objectives are to compare the performances of DENN and Theano without Curriculum Learning and the performances between DENN and DENN with Curriculum Learning, focusing specially to

[1]Please refer to Section 4.1.0.1 to see how the examples are formed.

Task	**Train complexity**		**Train error**		**Generalization**	
	No CL	CL	No CL	CL	No CL	CL
Access	$\text{len}(A) = 6$ $t = 3$	$\text{len}(A) \leq 10$	0	0	Semi -perfect	Perfect
Increment	$\text{len}(A) = 9$ $t = 9$	$\text{len}(A) \leq 11$	0	0	Perfect	Perfect
Copy	$\text{len}(A) = 5$ $t = 12$	$\text{len}(A) \leq 9$	0	0	_	Perfect
Reverse	$\text{len}(A) = 4$ $t = 9$	$\text{len}(A) \leq 8$	0	0	Perfect	Perfect
Swap	$\text{len}(A) = 7$ $t = 6$	$\text{len}(A) \leq 7$	0	40%	Semi -perfect	No
Permutation	$\text{len}(A) = 5$	$\text{len}(A) \leq 20$	0	0	Perfect	Perfect
ListK	$\text{len}(list) = \_$	$\text{len}(list) \leq \_$	_	_	_	_
ListSearch	$\text{len}(list) = \_$	$\text{len}(list) \leq \_$	_	_	_	_
Merge	$\text{len}(A) + \text{len}(B) = \_$	$\text{len}(A) + \text{len}(B) \leq \_$	_	_	_	_
WalkBST	$\text{size}(tree) = \_$	$\text{size}(tree) \leq \_$	_	_	_	_
Sum	$\text{len}(A) + \text{len}(B) = 6$	$\text{len}(A) + \text{len}(B) \leq 10$	0	0	Perfect	Perfect

Table 5.8: 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 [**?**], except for the keyword **Semi-Perfect** which means that the found solution generalize well only with input sequences and timesteps compatible to those used in the training.

the generalization capacity of the discovered solutions.

The used variants of DE are **JADE**, **SHADE** and **L-SHADE**, which are combined with the mutation methods **DEGL** and **Current to p best** and the crossover method **bin**. We used in each tests a population of 100 individuals. For DEGL we used a neighborhood in the interval $[4,8]$ and for Current to p best a $p = 0.1$. All the tests are executed with a feedforward neural network with two hidden levels, each containing 260 neurons. The training set is composed of batches of 1000 examples. The cost calculation has been done with the cost function showed in the Section 1.2.1. The train error is calculated with the expression introduced in Paragraph 4.

Task	Train complexity	Train error	Generalization
Access	len$(A) = 9$ $t = 3$	_	No
Increment	len$(A) = 9$ $t = 3$	_	No
Copy	len$(A) = 5$ $t = 3$	_	No
Reverse	len$(A) = 4$ $t = 3$	_	No
Swap	len$(A) = 7$ $t = 3$	_	No
Permutation	len$(A) =$ _ $t =$ _	_	_
ListK	len$(list) =$ _ $t =$ _	_	_
ListSearch	len$(list) =$ _ $t =$ _	_	_
Merge	len$(A) +$ len(B) $=$ _ $t =$ _	_	_
WalkBST	size$(tree) =$ _ $t =$ _	_	_
Sum	len$(A) +$ len(B) $=$ _ $t =$ _	_	_

Table 5.9: Results of the tests of Adam. The contained informations follows those in Table 5.8.

The generalization tests are executed with input sequences up to 1000 values. The Curriculum Learning was set up with an interval of batches between $[25, 250]$ and a threshold equal to 0.1.

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 5.8, 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

Step	0	1	2	3	4	5	6	7	8	9	10	11	*r*0	*r*1	*r*2	*r*3	*r*4	*r*5	Read	Write
1	6	3	1	2	10	11	0	0	0	0	0	0	1	1	0	0	0	1	p:0	p:6 v:6
2	6	3	1	2	10	11	6	0	0	0	0	0	2	2	0	0	0	2	p:1	p:10 v:3
3	6	3	1	2	10	11	6	0	0	0	3	0	3	3	0	0	0	3	p:2	p:9 v:1
4	6	3	1	2	10	11	6	0	0	1	3	0	4	4	0	0	0	4	p:3	p:8 v:2
5	6	3	1	2	10	11	6	0	2	1	3	0	5	5	0	0	0	5	p:4	p:7 v:10
6	6	3	1	2	10	11	6	10	2	1	3	0	6	6	0	0	0	6	p:5	p:6 v:11
7	6	3	1	2	10	11	11	10	2	1	3	0	7	7	0	0	0	7	p:6	p:5 v:11
8	6	3	1	2	10	11	11	10	2	1	3	0	8	8	0	0	0	8	p:7	p:4 v:10
9	6	3	1	2	10	11	11	10	2	1	3	0	9	9	0	0	0	9	p:8	p:3 v:2
10	6	3	1	2	10	11	11	10	2	1	3	0	10	10	0	0	0	10	p:9	p:2 v:1
11	6	3	1	2	10	11	11	10	2	1	3	0	11	11	0	0	0	11	p:10	p:1 v:3
Final	6	3	1	2	10	11	11	10	2	1	3	0	11	11	0	0	0	11	×	×

Table 5.10: 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.

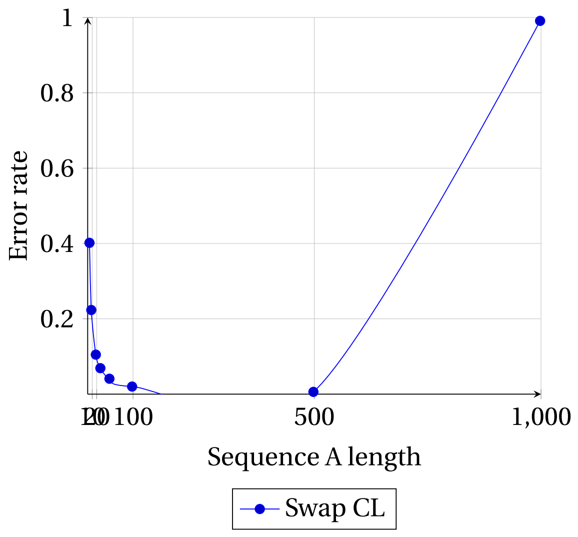


Figure 5.1: Error rate of best controllers in generalization tests.

using shorter input sequences compared to those used in the paper. Differently from which declared in Kurach at al., for tasks Copy and Swap the search of hyper-parameters has been difficulty. The same results was not reached in the training with Theano, as showed in **??**. Regard the generalization, we observed that the solutions that reached error zero, except with task Reverse and Increment, generalize well only if the input sequences and timesteps are compatible with those used in the training. For example, the training with the task Access has been made with an input

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	*r*0	*r*1
1	4	7	8	3	11	13	13	6	5	2	0	4	1	5	0	1	4
2	4	7	8	3	11	13	13	6	5	2	0	4	1	5	0	14	0
3	11	7	8	3	11	13	13	6	5	2	0	4	1	5	0	14	11
Final	11	7	8	3	11	13	13	6	5	2	0	4	1	5	0	14	11
Expected	11	7	8	3	11	13	13	6	5	2	0	4	1	5	0	14	11

Table 5.11: Right behaviour obtained with the task Access using only 3 timesteps. Once the controller has accessed the value in the input sequence, "stop" the execution, i.e. generates a circuit that always writes the accessed value in the first position.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	*r*0	*r*1
1	10	4	9	6	14	1	1	10	0	11	6	7	3	0	0	1	10
2	10	4	9	6	14	1	1	10	0	11	6	7	3	0	0	14	0
3	6	4	9	6	14	1	1	10	0	11	6	7	3	0	0	14	6
4	6	4	9	6	14	1	1	10	0	11	6	7	3	0	0	12	0
5	1	4	9	6	14	1	1	10	0	11	6	7	3	0	0	14	1
Final	1	4	9	6	14	1	1	10	0	11	6	7	3	0	0	14	1
Expected	6	4	9	6	14	1	1	10	0	11	6	7	3	0	0	14	1

Table 5.12: Wrong behaviour obtained with the task Access using only 5 timesteps. The controller continues to access the values indicated in the position 0 of the memory.

sequence of 6 values and 3 timesteps, as it can be seen in the Table **??**. Perhaps, if we execute a controller that reached in the training cost zero for more timesteps compared to those used in the training, keeping the same input sequence, the resulting memory is not that expected, like it can be seen in the Table **??**. This shows that the controller learned the patterns that arise in the registers only for 3 timesteps and not for more of these, generating later circuits that have unexpected behaviours.
The hard tasks...

As observed in Kurach et al., the Curriculum Learning is necessary in some cases. For example, the controller discovered with DENN that reached cost zero with the task Access generalize perfectly.

5.4 Circuits

Following are presented the circuits generated by the best solutions found with the training using DENN. For needs of space, in the circuits are showed only the necessary gates and registers. In the gates **Less-Than**, **Less-Equal-Than**, **Equality**, **Min** and **Max** are important the parameters order, indicated with x and y. Instead, for the gates **Read** and **Write**, the pointer and the value to write are indicated, respectively, with **ptr** and **val** labels.

Overall, the circuits generated differ from those in the paper. Moreover, we observed that the solutions found with different DE variants generate differents circuits. As did in [**?**], we present only the "main" circuits for the best solutions - for the differences please refer to the original paper. The tables shown in the following examples contains the shots of the memories and registers at the beginning of the steps and the final memory when the NRAM end to execute.

5.4.1 Access

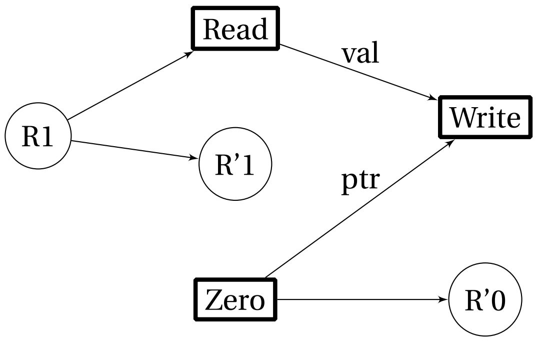


Figure 5.2: The circuit generated for all the steps for the task Access in the training with SHADE/DEGL/1/bin and Curriculum learning.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	*r*0	*r*1
1	6	10	13	11	13	13	9	4	1	12	3	6	11	4	0	0	6
2	6	10	13	11	13	13	9	4	1	12	3	6	11	4	0	0	6
3	9	10	13	11	13	13	9	4	1	12	3	6	11	4	0	0	6
4	9	10	13	11	13	13	9	4	1	12	3	6	11	4	0	0	6
5	9	10	13	11	13	13	9	4	1	12	3	6	11	4	0	0	6
Final	9	10	13	11	13	13	9	4	1	12	3	6	11	4	0	0	6

Table 5.13: Behaviour obtained with the task Access.

5.4.2 Increment

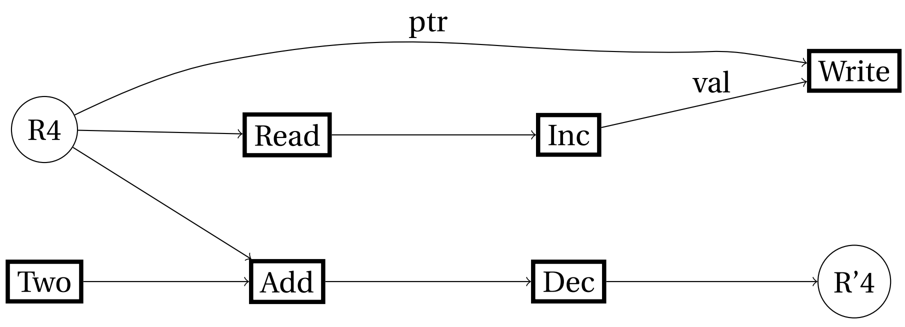


Figure 5.3: The circuit generated for every step for the task Increment found with SHADE/current to p best/1/bin with Curriculum Learning.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	*r*0	*r*1	*r*2	*r*3	*r*4
1	12	12	12	10	1	12	10	1	10	1	13	14	4	11	0	0	0	13	0	1
2	13	12	12	10	1	12	10	1	10	1	13	14	4	11	0	13	0	0	0	2
3	13	13	12	10	1	12	10	1	10	1	13	14	4	11	0	0	12	13	12	3
4	13	13	13	10	1	12	10	1	10	1	13	14	4	11	0	13	10	11	10	4
5	13	13	13	11	1	12	10	1	10	1	13	14	4	11	0	11	1	2	1	5
6	13	13	13	11	2	12	10	1	10	1	13	14	4	11	0	2	12	13	12	6
7	13	13	13	11	2	13	10	1	10	1	13	14	4	11	0	13	10	11	10	7
8	13	13	13	11	2	13	11	1	10	1	13	14	4	11	0	11	1	2	1	8
9	13	13	13	11	2	13	11	2	10	1	13	14	4	11	0	2	10	11	10	9
10	13	13	13	11	2	13	11	2	11	1	13	14	4	11	0	11	1	2	1	10
11	13	13	13	11	2	13	11	2	11	2	13	14	4	11	0	2	13	14	13	11
12	13	13	13	11	2	13	11	2	11	2	14	14	4	11	0	14	14	0	14	12
Final	13	13	13	11	2	13	11	2	11	2	14	0	4	11	0	14	14	0	14	12

Table 5.14: Behaviour obtained with the task Increment.

5.4.3 Copy

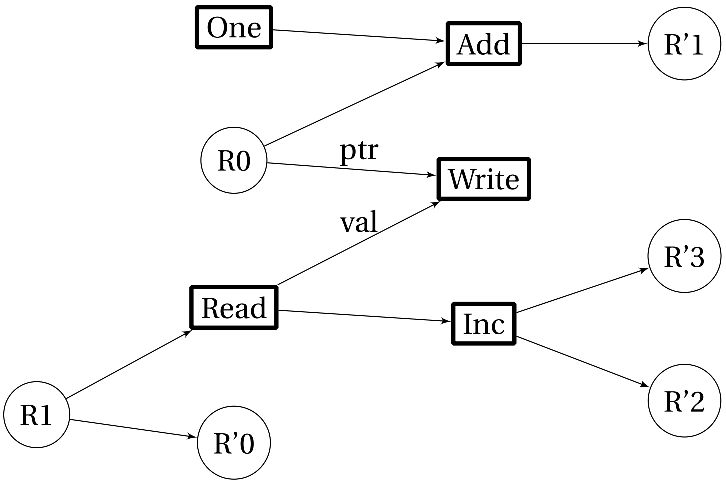


Figure 5.4: The only circuit generated for all the steps for the task Copy found in the training of the ANN with JADE/curr to p best/1/bin.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	*r*0	*r*1	*r*2	*r*3
1	8	11	15	5	8	13	8	3	0	0	0	0	0	0	0	0	0	0	0	0
2	8	11	15	5	8	13	8	3	0	0	0	0	0	0	0	0	8	1	2	1
3	8	11	15	5	8	13	8	3	11	0	0	0	0	0	0	0	1	9	12	12
4	8	0	15	5	8	13	8	3	11	0	0	0	0	0	0	0	9	2	1	1
5	8	0	15	5	8	13	8	3	11	15	0	0	0	0	0	0	2	10	0	0
6	8	0	0	5	8	13	8	3	11	15	0	0	0	0	0	0	10	3	2	1
7	8	0	0	5	8	13	8	3	11	15	5	0	0	0	0	0	3	11	6	6
8	8	0	0	0	8	13	8	3	11	15	5	0	0	0	0	0	11	4	1	1
9	8	0	0	0	8	13	8	3	11	15	5	8	0	0	0	0	4	12	9	9
10	8	0	0	0	0	13	8	3	11	15	5	8	0	0	0	0	12	5	1	1
11	8	0	0	0	0	13	8	3	11	15	5	8	13	0	0	0	5	13	14	14
12	8	0	0	0	0	0	8	3	11	15	5	8	13	0	0	0	13	6	1	1
13	8	0	0	0	0	0	8	3	11	15	5	8	13	8	0	0	6	14	9	9
14	8	0	0	0	0	0	0	3	11	15	5	8	13	8	0	0	14	7	1	1
15	8	0	0	0	0	0	0	3	11	15	5	8	13	8	3	0	7	15	4	4
Final	8	0	0	0	0	0	0	0	11	15	5	8	13	8	3	0	15	8	1	1

Table 5.15: Behaviour obtained with the task Reverse.

5.4.4 Reverse

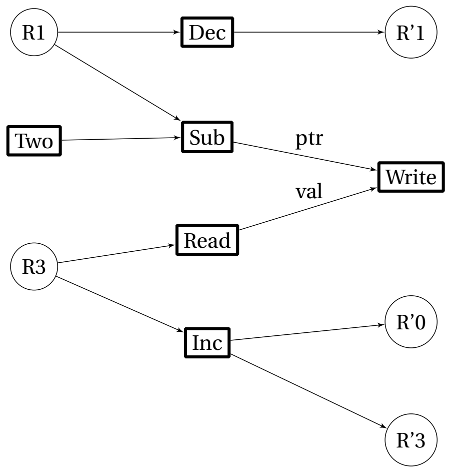


Figure 5.5: The circuit generated by the second step with the solution found in the training with JADE/current to p best/1/bin with Curriculum Learning.

Step	0	1	2	3	4	5	6	7	8	9	10	11	$r0$	$r1$	$r2$	$r3$
1	6	11	10	10	4	3	0	0	0	0	0	0	1	0	0	1
2	6	11	10	10	4	3	0	0	0	0	6	0	2	11	0	2
3	6	11	10	10	4	3	0	0	0	0	11	0	3	10	0	3
4	6	11	10	10	4	3	0	0	0	10	11	0	4	9	0	4
5	6	11	10	10	4	3	0	0	10	10	11	0	5	8	0	5
6	6	11	10	10	4	3	0	4	10	10	11	0	6	7	0	6
7	6	11	10	10	4	3	3	4	10	10	11	0	7	6	0	7
8	6	11	10	10	4	3	3	4	10	10	11	0	8	5	0	8
9	6	11	10	10	4	3	3	4	10	10	11	0	9	4	0	9
10	6	11	10	10	4	3	3	4	10	10	11	0	10	3	0	10
11	6	11	10	10	4	3	3	4	10	10	11	0	11	2	0	11
Final	6	11	10	10	4	3	3	4	10	10	11	0	11	2	0	11

Table 5.16: Behaviour obtained with the task Reverse.

5.4.5 Swap

TODO

Conclusions

List of Figures

Acknowledgements