Institut für Parallele und Verteilte Systeme

Abteilung Bildverstehen

Universität Stuttgart

Universitätstrasse 38

D-70569 Stuttgart

High Performance Computing Center Stuttgart

(HLRS)

Abteilung Applications, Models and Tools

Universität Stuttgart

Nobelstrasse 19

D-70569 Stuttgart

Diplomarbeit Nr. 3353

Optimization of Back-propagation Learning Algorithm on MLP Networks

Daniel del Hoyo Rodríguez

Studiengang:	Hauptdiplom Informatik
Prüfer:	Prof. Dr. rer. nat. habil. Paul Levi
Betreuer:	Dr. rer. nat. Oliver Zweigle
	Dr. sc. ETH Zürich Colin W. Glass
Begonnen am:	06.07.2012
Beendet am:	02.11.2012
CR-Klassifikation:	D.1.3, I.2.6

Abstract

In order to generate more efficient neural networks, the configuration of the ANN itself has to be optimized, specially refering to its parameters and architecture. To do so, this problem will be approached from the learning and training process point of view, realizing different tests. These evaluations will lead us to determine which are the most optimum parameters for this processes. At the same time, the importance of the input pattern and the data used will be studied, observing how these influences on the learning process, not only from a runtime point of view, but also measuring the obtained error in the trained network.

On the other side, the implementation itself will be optimized, doing this by executing the learning algorithm in parallel, using different nodes, meassuring the time needed for completing the trainning, and comparing it with the time needed in a sequential execution.

Contents

List of Figures

List of Algorithms

List of Equations

Chapter 1

Introduction

Artificial Neural Networks (ANN) are a mathematical model of the nervous system for information processing. It mimics the structure of the brain, and takes a set of input and generates output based on knowledge gained by prior experience through a learning process. This learning process is determined by many different parameters, such as the learning rate, the number of hidden layers and its configuration.

GPUs can be used for parallel computation, especially since NVIDIA launched its Compute Unified Device Architecture(CUDA), which makes the development process easier for the programmer. With large data sets, execution speed becomes an important factor. Implementing the learning algorithm on GPGPU and optimizing it using OpenMP produces faster networks.

The main goal of this thesis is to study and optimize ANN performance, both from execution speed and from a result quality point of view, thus, identifying configuration parameters that produce better results and speeding up the whole process.

The objetive of this work is to analyze and study the parameters for an optimal net topology and parametrization, and also do a reparallelization of the previous software using OpenMP to optimize it even for small input data-sets (having multiple kernels runs at the same time).

Chapter 2

Neural networks

Artificial neural network (ANN) are computational models based on the biological neuron. These models are able to solve a problem having a representative example pattern of the problem. There are different types of networks, depending on how they learn, or how its architecture is.

Some of the fields where ANN can be applied are:

- Medical diagnosis
- Machine diagnostics
- Quality control
- Explosives detection
- Target recognition for military
- Intrusion detection in computer network security
- Optical character recognition (OCR)
- Voice recognition
- Evolutionary and evolutive robotics
- Financial forecasting
- Credit card fraud detection

2.1 Biological perspective

From a biological point of view, a neuron is an electrically excitable cell that processes and transmits information by electrical and chemical signalling. They were recognized as primary functional unit of the nervous system by the anatomist Santiago Ramon y Cajal.

Neurons have a typical morphological characteristics, which define their function:

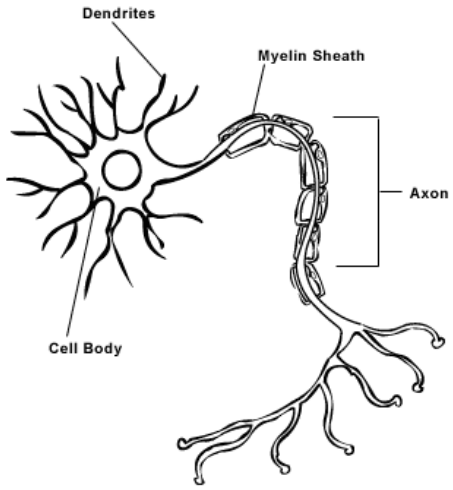


Figure 2.1: Structure of an human neuron

- Soma: cell body. It contains the cell nucleus
- Dendrite: branched projections of a neuron. They conduct the received stimulation from another neuron to the cell body.
- Axon: nerve cell. It conducts the electrical impulse away from the cell body.
- Synapse: structure that permits a neuron to pass an electrical signal to another cell.

Therefore, the learning and knowledge obtained is stored as synapses between networks. Although this is still not well known, it's thought that neurons can store both digital and analogical information [7].

2.2 Artificial perspective

An artificial neuron is a mathematical function which tries to model a human neuron. It's the basic unit of artificial neural networks. The artificial neuron receives one or more

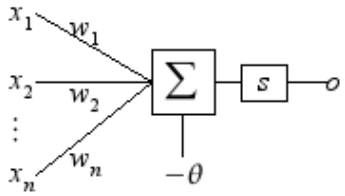


Figure 2.2: McCulloch-Pitts neuron

inputs (dendrites), and sums them, producing one output (axon). Usually the sums of each node are weighted (synapses), and the sum is passed through a non-linear function known as activation or transfer function.

The interconnection of this artificial neurons produces neural networks.

2.3 Types of neural networks

According to the number of layers of the network, we could have the following classification:

- Single-layer: This networks only have one layer, which receives the input, and produce the output.
- Multi-layer: Multi-layer networks are composed by various layers. The layer which receives the network is different than the one who receives the input.

According to the topology of the network, we could have the following classification:

- Feed-forward ANN:
  The connections between neurons do not form a directed cycle. Therefore, the information only moves in one direction, from the input nodes, through the hidden ones, until the output layer, making use of a series of weights. There could be many different intermediate layers (multilayer).
- Recurrent ANN:
  The connections between neurons form a directed cycle, which means that this type of networks have at least one closed loop of neural activation.
  Therefore, the information doesn't only travel from the input to the output, but it

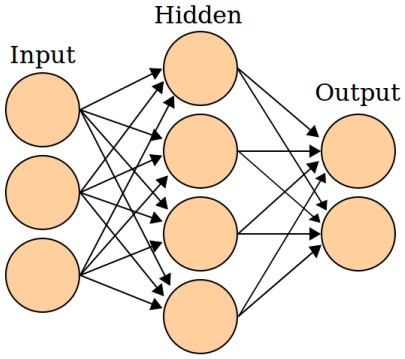


Figure 2.3: Multi-layer ANN

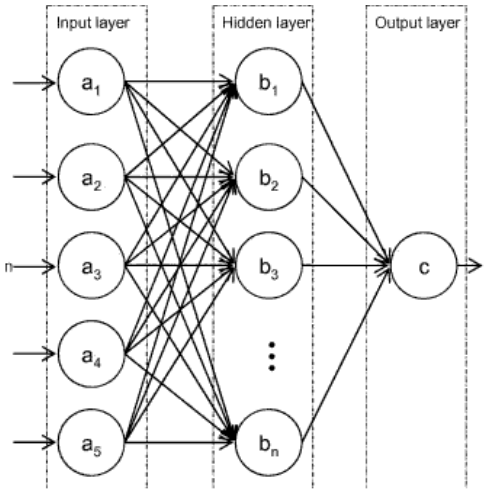


Figure 2.4: Feed-forward ANN

also can travel in both ways. As they have memory elements to store the information of several time steps, they can have a dynamic behaviour, and use this feature to process arbitrary sequences of inputs.
This characteristic make them suitable for many different applications, such as handwriting character recognition (OCR).

Unfortunately, this types of networks can need a long training time to compute and produce stable outputs.

According to the input, we could have the following schema:

- Continuous networks: Process continuous input data sets. Most of the networks are of this type.
- Discrete networks: Process discrete input data sets, usually boolean values.

A classification according to the learning paradigm is defined in chapter 4.

Some of the most common neural networks are:

- Perceptron:
  This network is associated to a neuron. It's the simplest feed-forward supervised

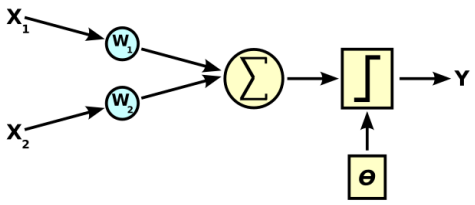


Figure 2.5: Feed-forward ANN

  learning ANN. Perceptron is only able to solve linear problems.

- Multi-layer percetron:
  It's a modification of the standard linear perceptron that maps sets of input data onto a set of appropriate output. An MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one, and it's able to solve non-linear problems.
  It consists of an input layers, various hidden layers, and an output layer.

- ADALINE:
  Modification of the MLP network, which not only is able to classify as MLP does, but also to estimate a real output. It is usually formed by only one layer of neurons.

- Kohonen:
  It is also known as self-organizing map. Unsupervised learning net which usually produces a map as result. The training process builds the map according to the input parameters while the mapping process classifies the input (vector).

- Hopfield:
  Recurrent ANN used as associative memory, with binary threshold units. They are designed to converge to a local minimum, but this convergence is not guaranteed.

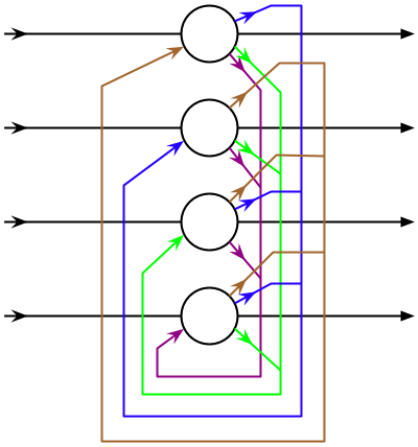


Figure 2.6: Hopfield ANN

In this master thesis, every simulation and optimization will be realized on MLP networks.

Chapter 3

Related work

Neural networks have a high number of applications in many different fields. Due to their nature, they're suitable for many problems, like classification problems or automatic recognition.

Focusing on the optimum configuration problem, Subhash C. Kalk [18] studied an approach that is based on the nature of the data. This approach trains the network by isolating the corner in the n-dimensional cube of the inputs represented by the input networks. This techniques show how a good generalization performance can be obtained, even just by visual inspection. Unfortunately, this technique is not usefull when working with many parameters.

S. Narain and A. Jain [18] studied how the learning rate influenced in the learning process. To do so, they realized different tests, using learning rates from 0.0001 to 0.05, and a limited number of iterations. Results found that small LRs were not able to achieve the desired convergence. As the learning rate is increased, the number of iterations needed to achieve the acceptable error first decrease and then increase after an optimal plateau.

On the parallel implementation side, Xavier Sierra, Francisco Madera, and H. Victor Cetina [19] have implemented the back propagation algorithm on GPU, using CUBLAS library. Also some CUDA kernels were implemented to overcome the unavailable function in CUBLAS. The comparison done with different set of benchmark datasets have shown up to 63 times faster computations than the sequential single threaded CPU implementation.

Chapter 4

Learning algorithm optimization

The output of a neuron depends on the weights parameters, therefore our aim is to adjust these parameters in such a manner that it produces the desired output, as well as extrapolate results of those input patterns which will be applied later. This adjustment of neural network parameter is done by learning algorithms.

Learning is the process where the free parameters of a neural network are adapted. Therefore, the weights of the different neurons will be adapted, and connections between neurons will be created, modified, or destroyed.

The way this adaptation is realized defines the learning algorithm itself. Corresponding to this, we can provide the following classification in three major categories:

- **Supervised learning**
  The function (i.e network configuration) will be deduced from the training data set. This set consist of two parts: the input data, and the desired output, which could be a numeric value or a label (classification). The algorithm should be able to generalize in order to predict the output value of any input in a reasonable way after the training process. An example of its configuration is pictured on figure 4.1

- **Unsupervised learning**
  Also known as self-organization learning [2]. No a priori knowledge is available, therefore, is based only upon local information (i.e. desired output is unknown). The input pattern is usually processed as a set of random variables, and classified into different patterns. This type of learning is without external teacher.
  Self organized maps (Kohonen networks) and Hebbian networks are some examples of unsupervised learning.

- **Reinforcement learning**
  Having as input a set of actions, a set of transitions between actions, and a set of rewards, the idea behind is to maximize some notion of cumulative reward. Correct

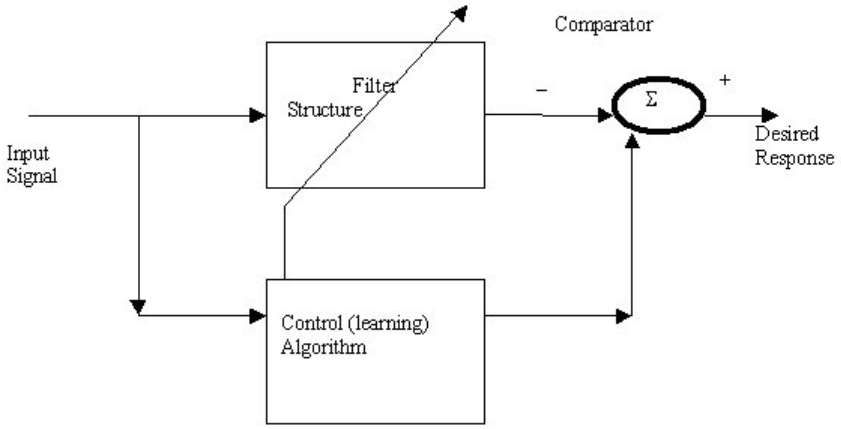


Figure 4.1: Supervised learning [1]

input/output patterns are never used. Mathematically, it is presented as a Markov decision process.
There should be a trade-off between exploration (i.e choosing action) and exploitation (i.e. current knowledge) [3].

We will work with a train data set (input and desired output), and therefore, this master thesis is focused on the supervised learning methods.

4.1 Forward pass

It consists in the propagation of the input signals to the output. Also, the calculation of the error made is done, comparing the obtained output with the desired output.

The idea behind is to transform the incoming inputs to one value usually by summation over product of input and weight values.

The product between the weights and the input is passed into the activation function which produces the output of the neuron. In this way the neuron output of the current layer passes as input to the neuron in the next layer. This process eventually will reach the output layer, producing the final result of the neural network.

Different activation functions can be used. The most common ones are:

- Step function:

$$\varphi(x) = \begin{cases} 1 & if\ x \geq 0 \\ 0 & if\ x < 0 \end{cases} \tag{4.1}$$

- Linear ramp:

$$\varphi(x) = \begin{cases} 1 & if\ x \geq \frac{1}{2} \\ x & if\ x\ \frac{1}{2} > x \geq -\frac{1}{2} \\ 0 & if\ x \leq -\frac{1}{2} \end{cases} \tag{4.2}$$

- Sigmoid function:

$$\varphi(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{4.3}$$

The activation function used in this scope of work is an hyperbolic tangent function (i.e. sigmoid function).

Also, many other different functions (like a Fourier series transformation) could be used, producing different results.

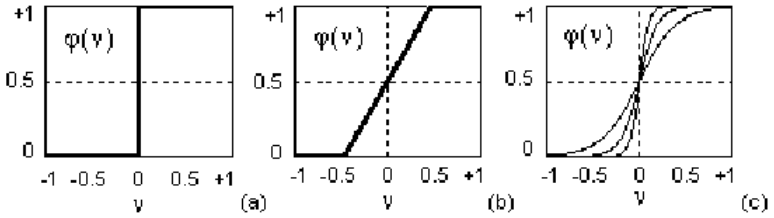


Figure 4.2: Different activation functions (a) Step. (b) Linear. (c) Sigmoid or hyperbolic

4.2 Backward pass

This procedure starts in the output layer, propagating the error signals to the input layer, and calculating recursively the local gradients for each neuron. The output error is used to alter weights on the output units. Then the error at the hidden nodes is calculated (by back-propagating the error at the output units through the weights), and the weights on the hidden nodes altered using these values.

The forward pass gave us the output. The next step is to perform a correction, which is done by adjusting the weight parameters of our neural network. The error values are calculated using equation 4.5, as it was already explained in the previous work [6].
This error is propagated backwards from the output layer to the input layer through all hidden layers. The weight delta is calculated using equation 4.4. After completing an epoch of the training set, the weight delta values are added to the weight list, in order to adjust the neural network.

$$\Delta W_{ji}(n+1) = \eta o_i \sigma_j + \alpha \Delta W_{ji}(n) \tag{4.4}$$

where

η is the learning rate which represent gradient descent step width.

α is the momentum which tell the amount of weight change.

σ is the error value of jth unit calculated by using the equation 4.5

$$\sigma_j = \begin{cases} (f'(net_j) + c)(desired - actual) & if\ j\ is\ output\ layer \\ (f'(net_j) + c) \sum_k \sigma_k W_{jk} & if\ j\ is\ hidden\ layer \end{cases} \tag{4.5}$$

where

c is the flat spot elimination value

$f'(net_j)$ is the derivative of the activation function

4.3 Problem description

A summarized pseudo code for the Back propagation algorithm is depicted below:

Algorithm 4.1 Back Propagation Algorithm

Initialize weights of every neuron with random values
Forward pass:
Apply input values to the neural network in order to calculate the output (i.e. actual output)
Compare the actual output with the desired output
Backward pass:
Propagate backwards from output to input layers for corrections
Compute weight deltas using equation 4.4
Repeat from step 2 to step 7 to cover all patterns
Update weight values of the neural network by adding weight deltas

Observing the algorithm 4.1 we can infer the following possible optimization parameters and procedures:

- Activation function:
  Already explained in 4.1, it's used to calculate the output of each neuron during the forward pass. In our simulations, a sigmoid activation function is used.

- Weight initialization:
  In order to avoid the network saturation, weights usually are initialized randomly, between a small range (usually between [-0.05, 0.05] or [-0.5,+0.5]). The essential idea to solve this problem is to try to approach to the linear regions of the activation function.
  In our simulations, weights are initialized randomly.

- Learning rate:
  The smaller the learning rate is, the smaller the changes will be. On the other hand, if we make this parameter too big in order to accelerate the convergency, an instability could be produced, which means an oscillating behaviour of the network. In order to avoid this instability, the momentum was added, producing the generalized delta rule.

- Momentum:
  To ensure the converge of the network, momentum value should be between the values of 0 and 1: $0 < |\alpha| \leq 1$
  This value prevents that convergency ends up in a local minimum.

- Flat spot elimination:
  It's a constant value which is added to the derivative of the activation function to enable the network to pass flat spots of the error surface.

- Neuron count and hidden layers:
  The number of neurons and the hidden layers highly determines the behaviour of the network. Find the optimal neural count of a network and an optimal hidden layers configuration is still an open problem.

  Smaller neural networks are preferable rather than bigger ones, due to various reasons: the number of parameters is smaller, the training is faster, and usually have a better generalization ability when using new patterns. Therefore, in order to determine the best neural configuration we have different options:

  - Start with a big size neural network, and prune it, eliminating neurons and synapses, until the network configuration is optimal.
  - Start with a small size neural network and increase its size and processing units, until the network configuration is optimal.
  - Start with a medium size neural network, and prune the connections and units that might be considered unnecessary. As next step, add random neurons with random weights, train it again, and repeat the process until the network configuration is optimal.

  Although there are some pruning algorithms that could be used for this task (i.e. Optimal Brain Damage, Levenberg-Marquardt), a naive approach will be used to optimize the network layout (start with a small number of neurons and keep adding processing units).

Therefore, the parameters that will be optimized are the network configuration (i.e. hidden layers and neuron count), the learning rate, the momentum and the flat spot elimination rate.

Once the parameters are already optimized, the focus will be set in optimizing the code itslef, by reparallelization (Chapter 5

4.4 Optimization methods

Optimization is a branch of applied mathematics and numerical analysis that deals with finding the best parameters for a function or a set of functions according to certain criteria. In the presented case, the criteria used for the optimization is the generated error, which we want to minimize. The runtime will be considered as second criteria (in case we have two outputs with the same generated error, we will choose the one with the minimum running time.

In order to achieve this, different methods can be used, such as deterministic methods (i.e. Branch and Bound), stochastic methods (Montecarlo) or heuristic methods (i.e. evolutionary algorithms).
An heuristic method and an evolutionary strategy will be used in order to evaluate different evaluation techniques results.

4.4.1 Grid-based optimization

Grid-based optimization is an heuristic method for numerical optimization. Every parameter that has to be optimized is divided using a step size. If we call i, j, k to the parameters we are dealing with(range 0 to 1), and $step_i, step_j, step_k$ to the step size of parameter i, j and k respectively, the pseudo-code to do that would be the following:

Algorithm 4.2 Grid-based optimization algorithm

```
...
for (int i=0;i<1;i=i+step_i) {
  for (int j=0;j<1;j++=step_j) {
    for (int k=0;k<1; k++=step_k) {
      evaluation=f(i,j,k);
      ...
    }
  }
}
```

The complexity of this method is determined by

$$O(N^p) \tag{4.6}$$

where p is the number of parameters.

In our learning algorithm, we wanted to optimize 3 parameters (learning rate, momentum, and flat spot elimination), and the number of neurons in each layer (tests have been done for one, two, and three hidden layers),

4.4.2 CMA-ES optimization

The CMA-ES optimization [4] stands for Covariance Matrix Adaptation Evolution Strategy. It is an evolutionary strategy for non-convex continuous optimization problems.

It's based on the biological evolution principle, working with mutations, recombinations, and selection methods. Future generations (iterations) are generated via small mutation (usually using stochastic methods) and recombinating previous generation members. Later, using a selection function, individuals which produce better results are selected for future recombinations, producing more optimal future generations.

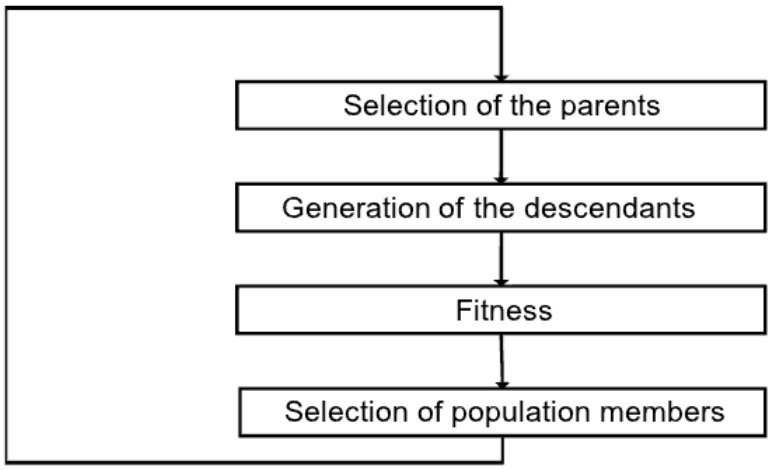


Figure 4.3: Evolutionary strategies

In this method, the mean of the distribution is updated such that the likelihood of the previously successful candidate solutions is maximized. This is done in order to increase the probability of successful candidate solutions and search step.
Also, two different tracks are recorded (evolution paths), that contains different information about the correlation between consecutive steps. This tracks help the algorithm to evolve in a favourable direction, and avoid it to converge prematurely. t

The CMA-ES algorithm used is based on modifications on the source code of Nikolaus Hansen [4], in order to use not a concrete function, but the results of executing the neural network with our desired parameters.

4.5 Evaluation

The method followed in order to evaluate the performance and the different adjustment level of the presented parameters is the following:

- First of all, and using as learning rate, momentum, and flat spot elimination, the values presented on the literature as the most optimal ones, different hidden layers configurations has been tested, each one with a different number of neurons. One, two and three hidden layers were used.
- As next step, a grid-based test was executed, changing not only the configuration of the hidden layers and the neuron count, but also the learning algorithm parameters.
- With the previous steps, we had an idea about which values of the learning algorithm produces better and optimized results. Also, a CMA-ES method will be used in order to determine the best neuron count for the given parameters.
- These tests will be realized for two different input patterns: sinus and mail; and for three different input pattern sizes: small, medium, and big.

As the mail pattern is much more complex, the training time is also higher. We will show in the results that the obtained results regarding training parameters or network configuration are quite similar to the ones obtained when training the network with a simple pattern (i.e. sinus).

Therefore, the mail pattern will only be used when needed in some test cases, in order to avoid time and resources consumption.

The evaluation will be realized in a NEC Nehalem Cluster platform. It consists of several front-end nodes for interactive access (for access details see Access) and several compute nodes for execution of parallel programs, with the following architecture:

- 700 compute nodes are of type NEC HPC-144 Rb-1 Server
  - dual CPU compute nodes: 2x Intel Xeon X5560 "Gainestown" (5000 Sequence specifications)
    - 4 cores, 8 threads
    - 2.80 GHz (3.20 Ghz max. Turbo frequency)
    - 8MB L3 Cache
    - 1333 MHz Memory Interface, 6.4 GT/s QPI
    - TDP 95W, 45nm technology
    - "Nehalem" micro-architecture

  - compute node RAM: triple-channel memory
    - standard: 12 GB RAM
    - 36 nodes upgraded to 24GB, 48GB, 128GB or 144GB RAM
  - 32 compute nodes have additional Nvidia Tesla S1070 GPU's installed.

Its features are the following:

- Operating System: ScientificLinux 5.3 (internal test on Windows HPC Server 2008)
- Batchsystem: Torque/Maui/Moab
- node-node interconnect: Infiniband + GigE
- Global Disk 60 TB (lustre)
- OpenMPI
- Compiler: Intel, GCC, Java

Three different execution times (read pattern, load pattern, train the network) will be measured. The mean square error will be used to analyse the network, which measures the average squared error between the network's output and the desired value

4.5.1 Learning rate

One hidden layer

- Grid-based evaluation values
  - Sinus pattern

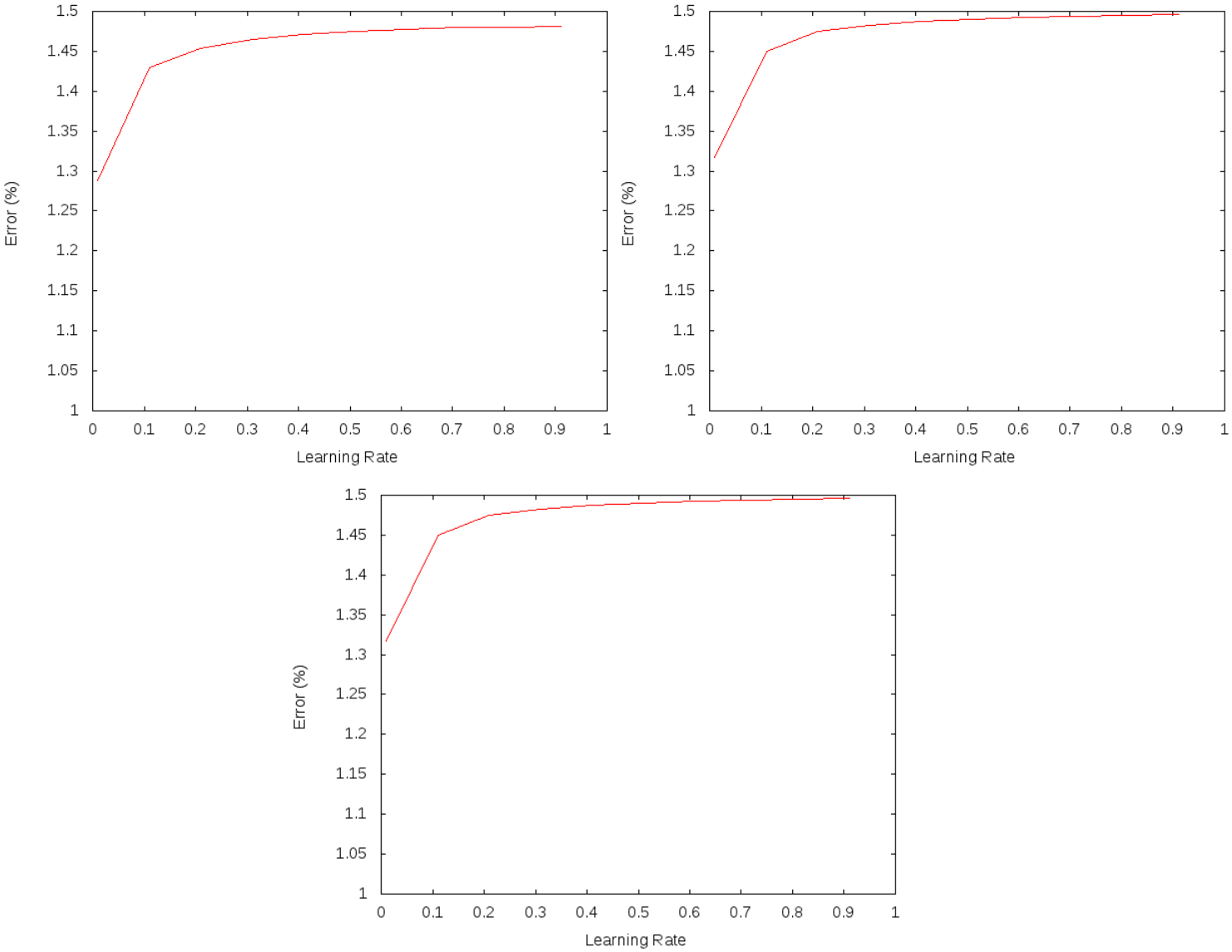


Figure 4.4: Learning rate vs error (%) with different size patterns: small(upper left), medium(upper right) and big (center)

Best results are obtained when using a learning rate of 0.01 for small, medium and big input patterns size. Learning rates of 0.11 and 0.21 also produce an accurate training.

– Mail pattern

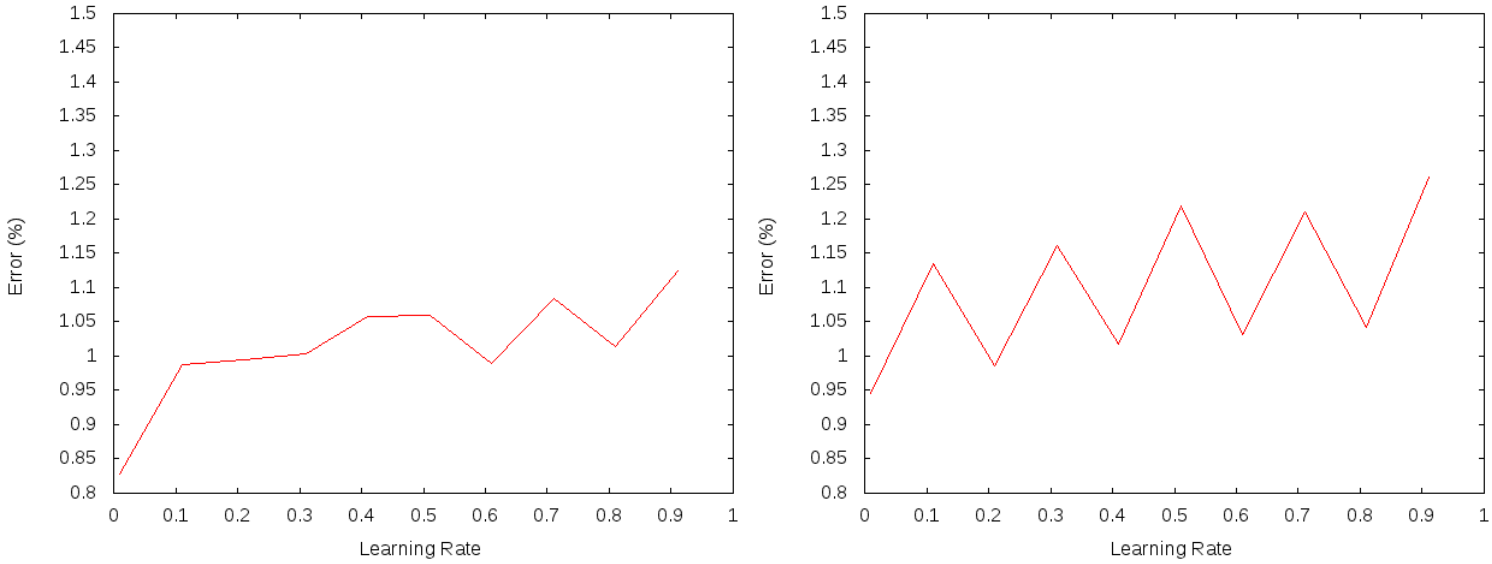


Figure 4.5: Learning rate vs error (%) with different size patterns: small(upper left), medium(upper right)

For a big pattern, it takes a lot of time (even more than 8 hours) to run each simulation. Because of this time-consuming process, results won't be calculated.

Best results are obtained when using a learning rate of 0.01 and 0.21 for small and medium size input patterns.

Two hidden layers

- Grid-based evaluation values

  – Sinus pattern

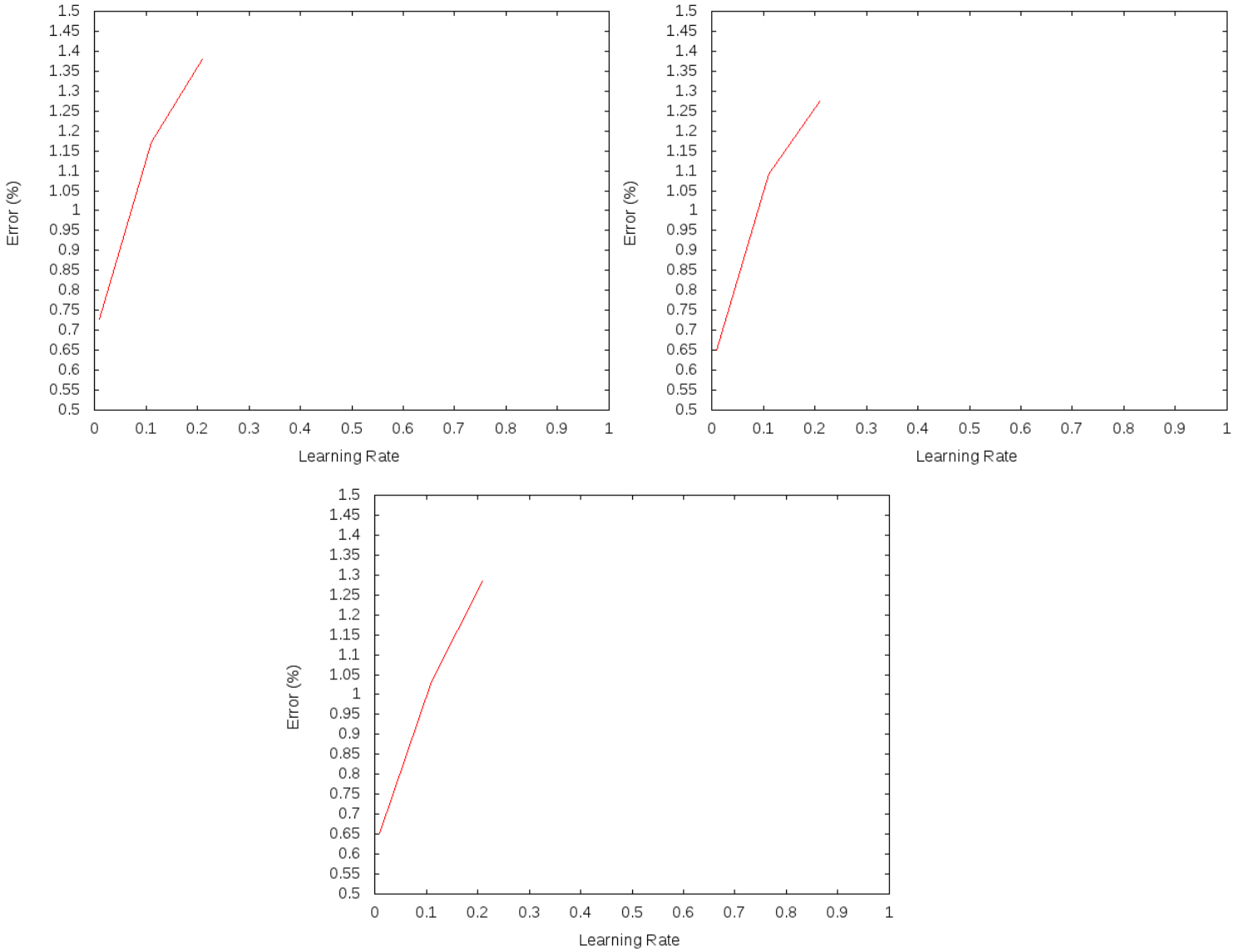


Figure 4.6: Learning rate vs error (%) with different size patterns: small(upper left), medium(upper right) and big (center)

For this tests, only the best LR configurations (obtained in the one layer benchmarks) have been evaluated.

Best results are obtained when using a learning rate of 0.01 for small, medium and big input patterns size.

Most of the simulations give high error percentages after the training process. However, specially when using a LR of 0.01, also a high number of network topology configurations can be defined in order to obtain a low error rate.

4.5.2 Momentum

One hidden layer

- Grid-based evaluation values
  - Sinus pattern

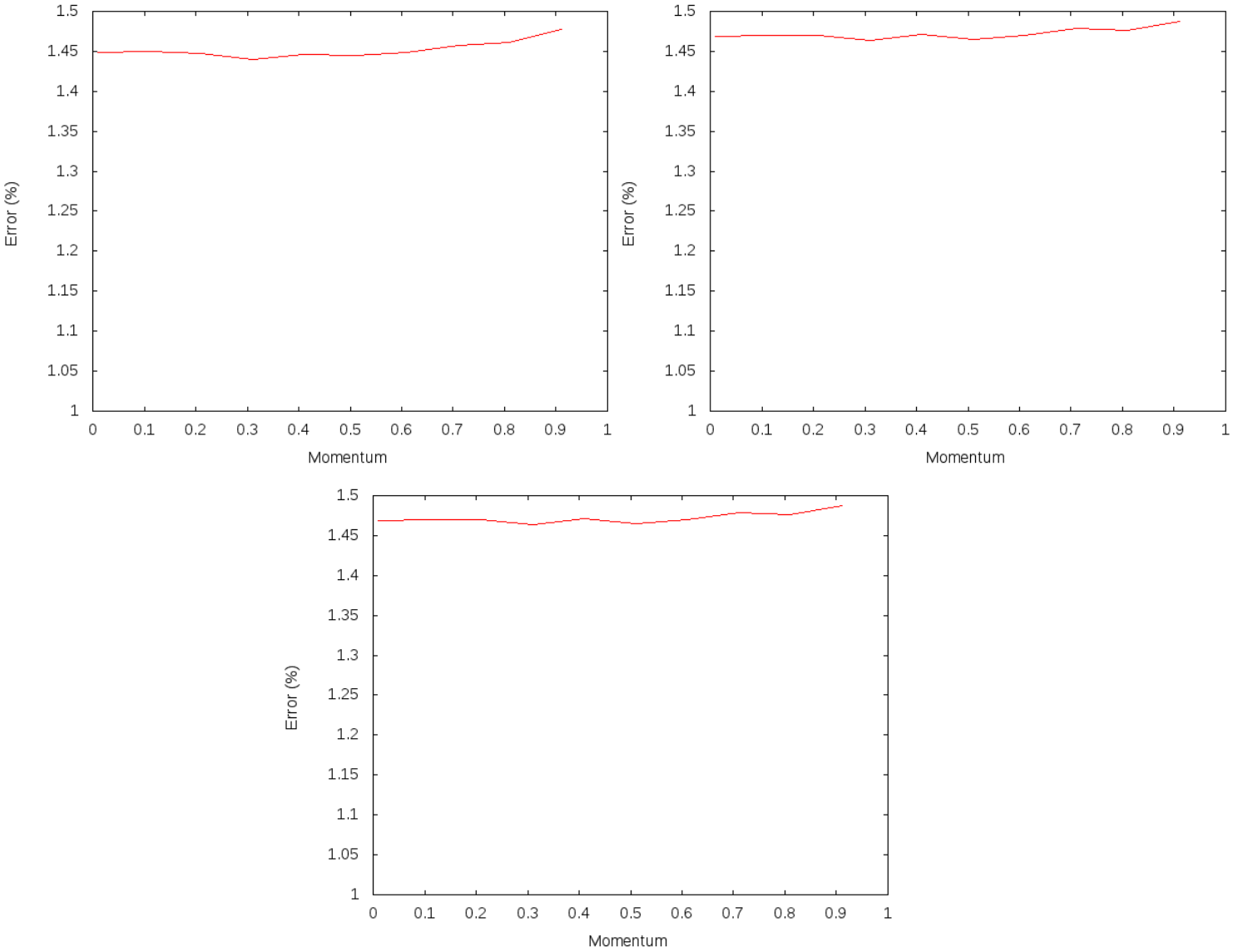


Figure 4.7: Momentum vs error (%) with different size patterns: small(upper left), medium(upper right) and big (center)

Best results (are obtained when using a momentum of 0.5 for small input patterns, and 0.1 and 0.6 for medium and big pattern size. Momentum values between 0.1 and 0.6 also produce an accurate training.

– Mail pattern

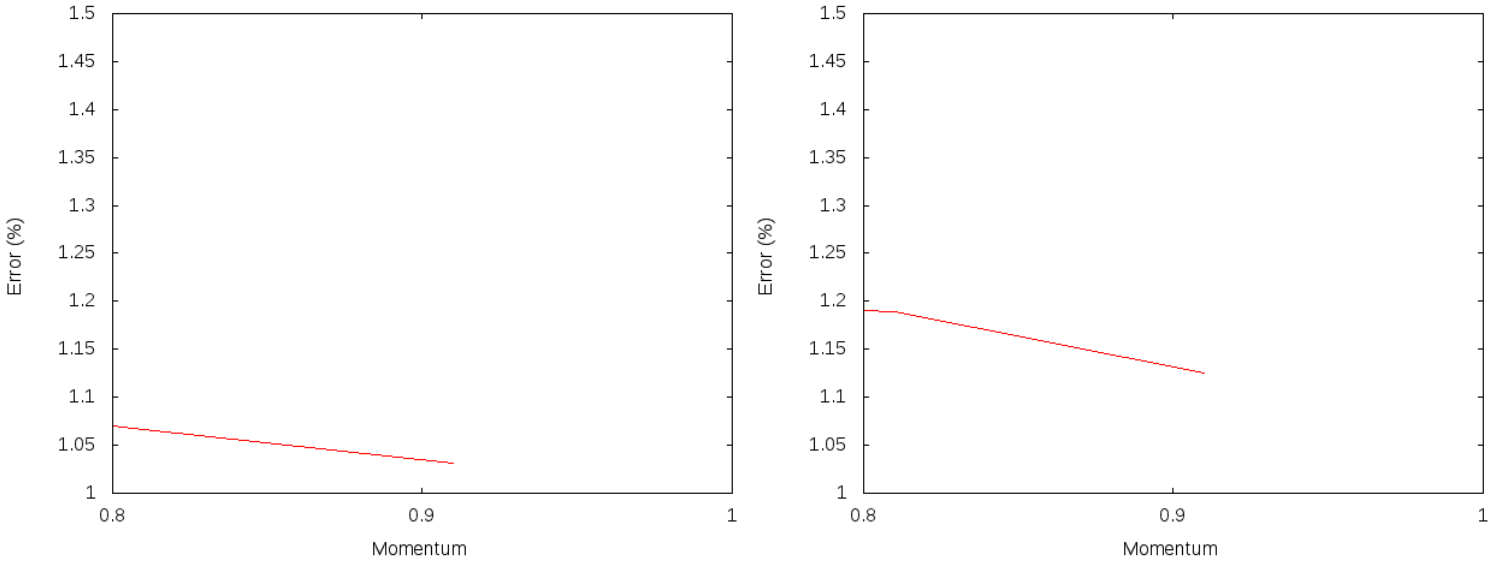


Figure 4.8: Momentum vs error (%) with different size patterns: small(upper left), medium(upper right)

For a big pattern, it takes a lot of time (even more than 8 hours) to run each simultation. Because of this time-consuming process, further results won't be calculated.

Best results are obtained when using a momentum of 0.01 for small and medium input patterns size. Momentum values between 0.3 and 0.6 also produce an accurate training.

Increasing the momentum rate for values further than 0.6 highly increases the error rate, producing much worse trained networks.

Two hidden layers

- Grid-based evaluation values
  - Sinus pattern

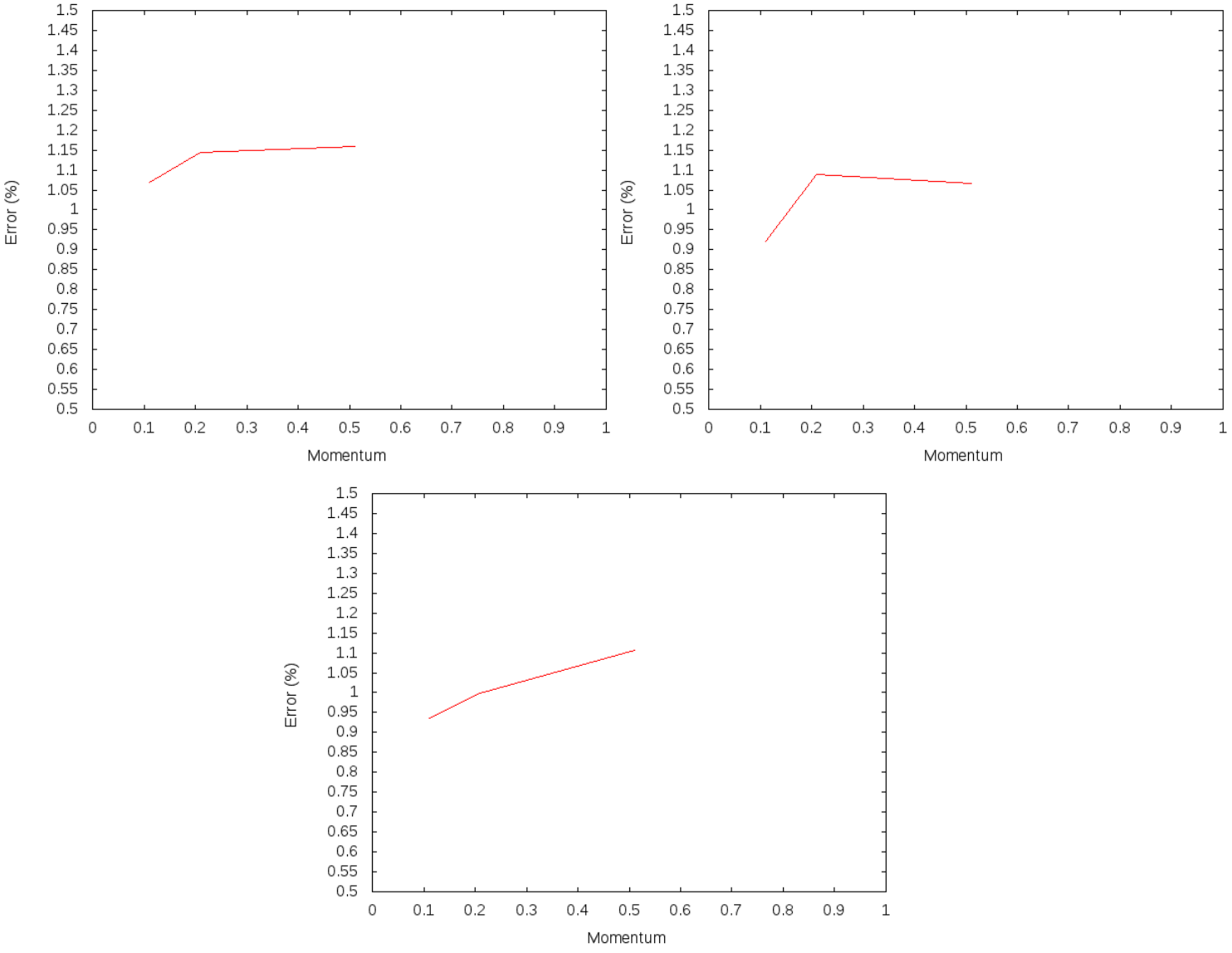


Figure 4.9: Momentum vs error (%) with different size patterns: small(upper left), medium(upper right) and big (center)

Again, only some specific momentum values have been evaluated for two layers. Best results are obtained when using a momentum of 0.1 for small and medium input patterns size.

Differences on the learning process are not significant when using different momentum values.

4.5.3 Flat spot elimination

One hidden layer

- Grid-based evaluation values
  - Sinus pattern

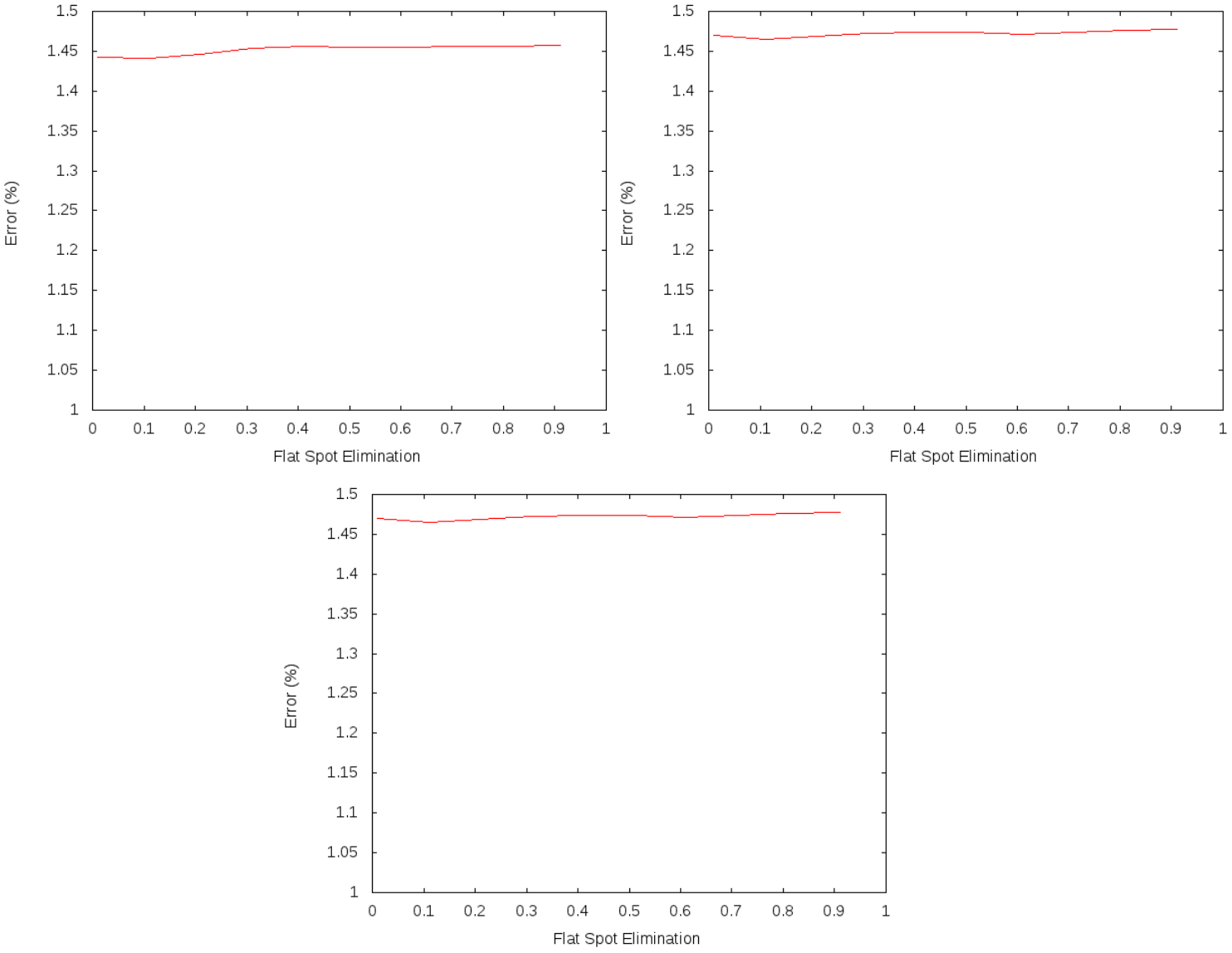


Figure 4.10: Learning rate vs error (%) with different size patterns: small(upper left), medium(upper right) and big (center)

Best results are obtained when using a flat spot elimination rate of 0.1 for small, medium and big size patterns. Flat spot elimination between 0.1 and 0.5 also produce accurate results.

– Mail pattern

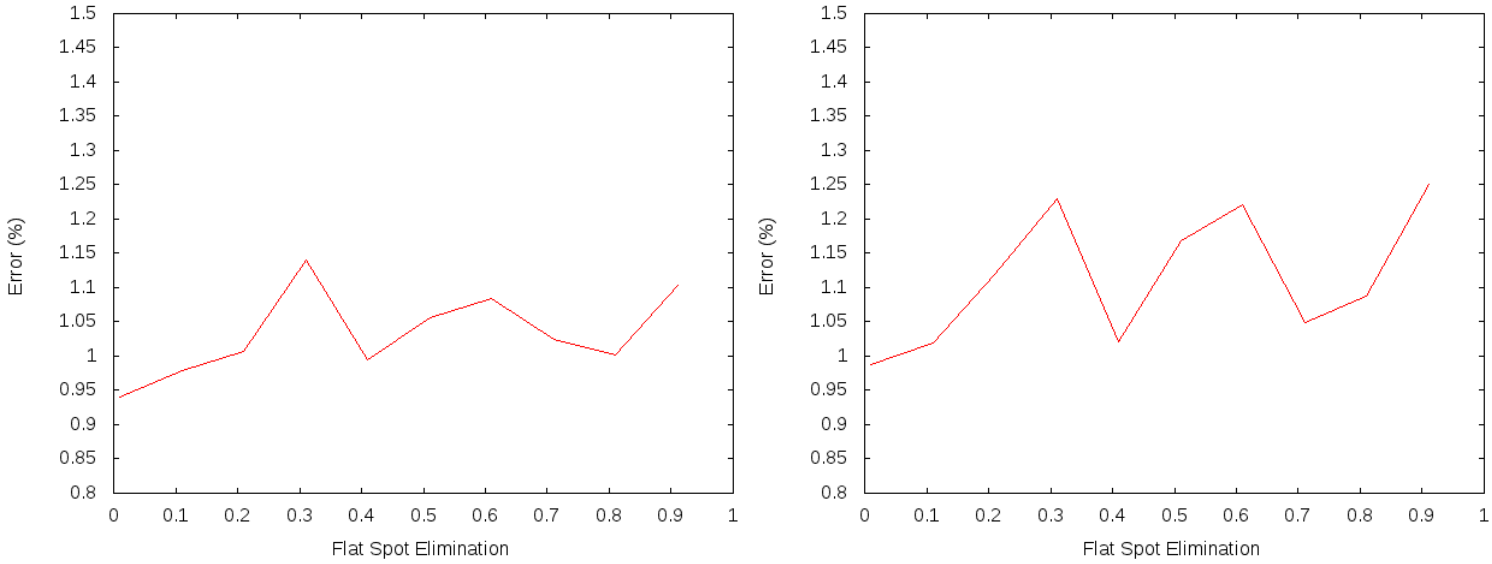


Figure 4.11: Flat spot elimination vs error (%) with different size patterns: small(upper left), medium(upper right)

For a big pattern, it takes a lot of time (even more than 8 hours) to run each simultation. Because of this time-consuming process, further results won't be calculated.

Best results are obtained when using a flat spot elimination rate of 0.1 for small and medium size patterns. Many other FSE rates between 0.01 and 0.5 also produce accurate results.

Two hidden layers

- Grid-based evaluation values
  - Sinus pattern

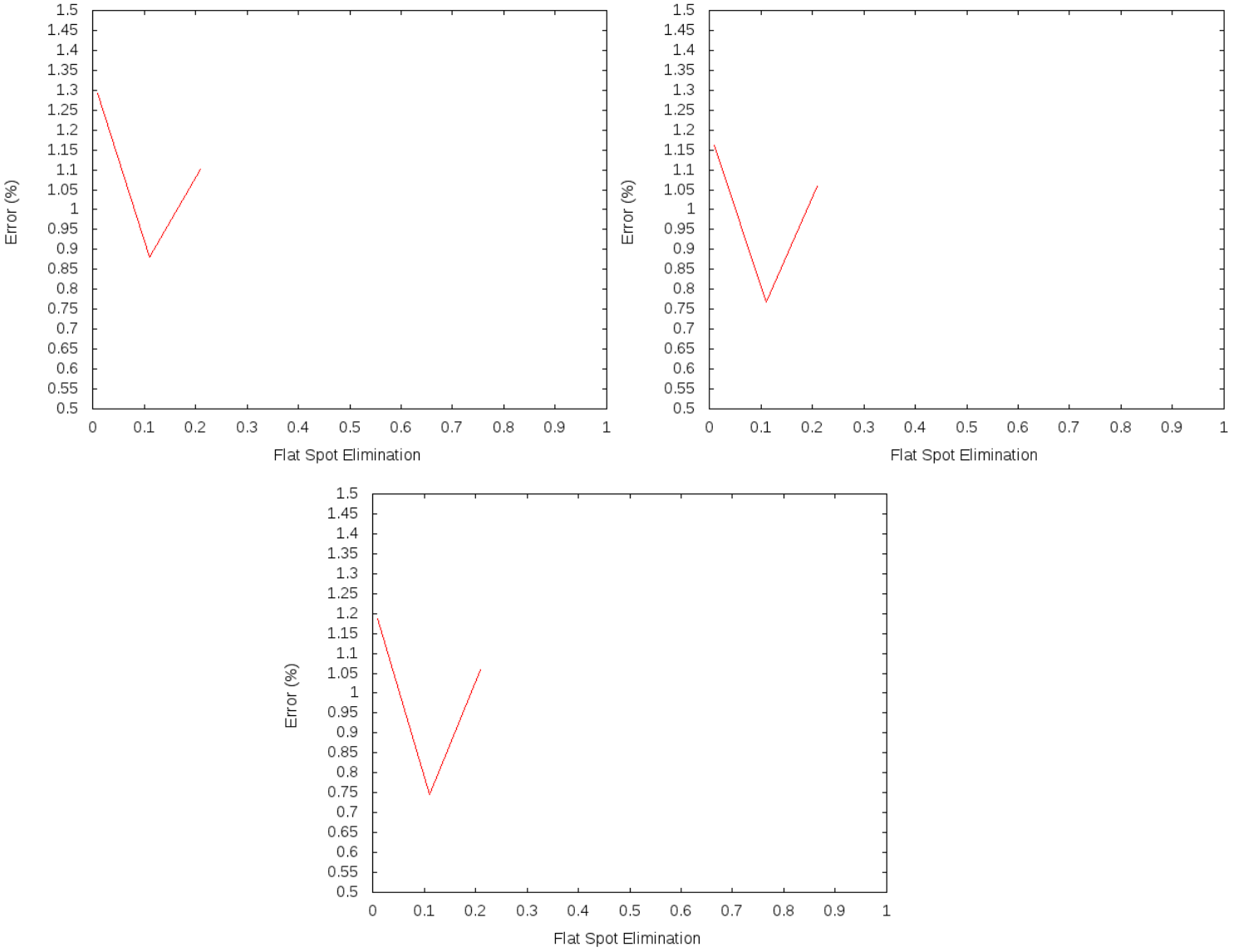


Figure 4.12: Flat spot elimination vs error (%) with different size patterns: small(upper left), medium(upper right) and big (center)

Only best FSE rates have been evaluated. Again, the differences between a FSE of 0.1 and other rates are not relevant.

4.5.4 Neuron count

One hidden layer

- **Default parameters**
  - Sinus pattern

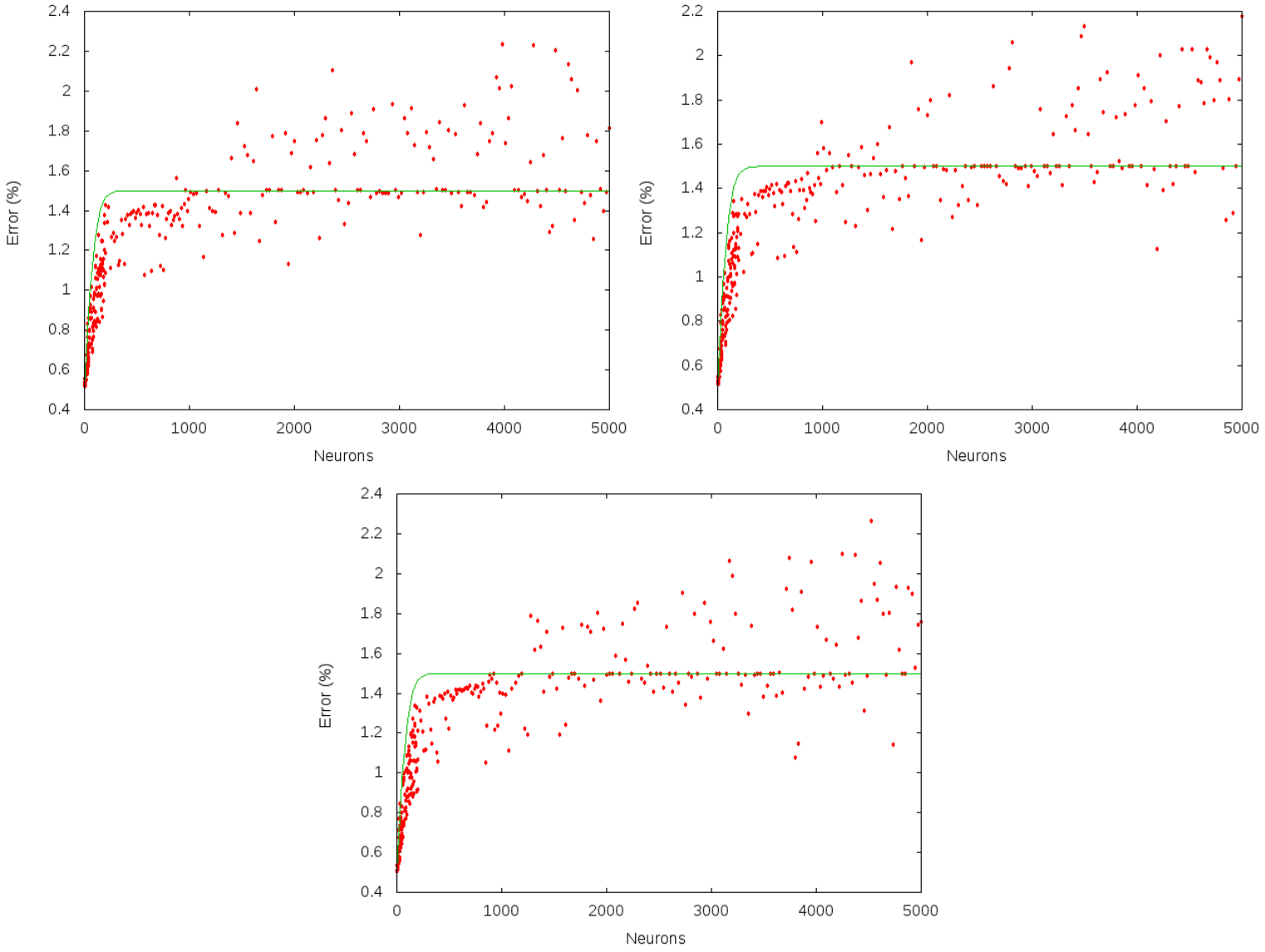
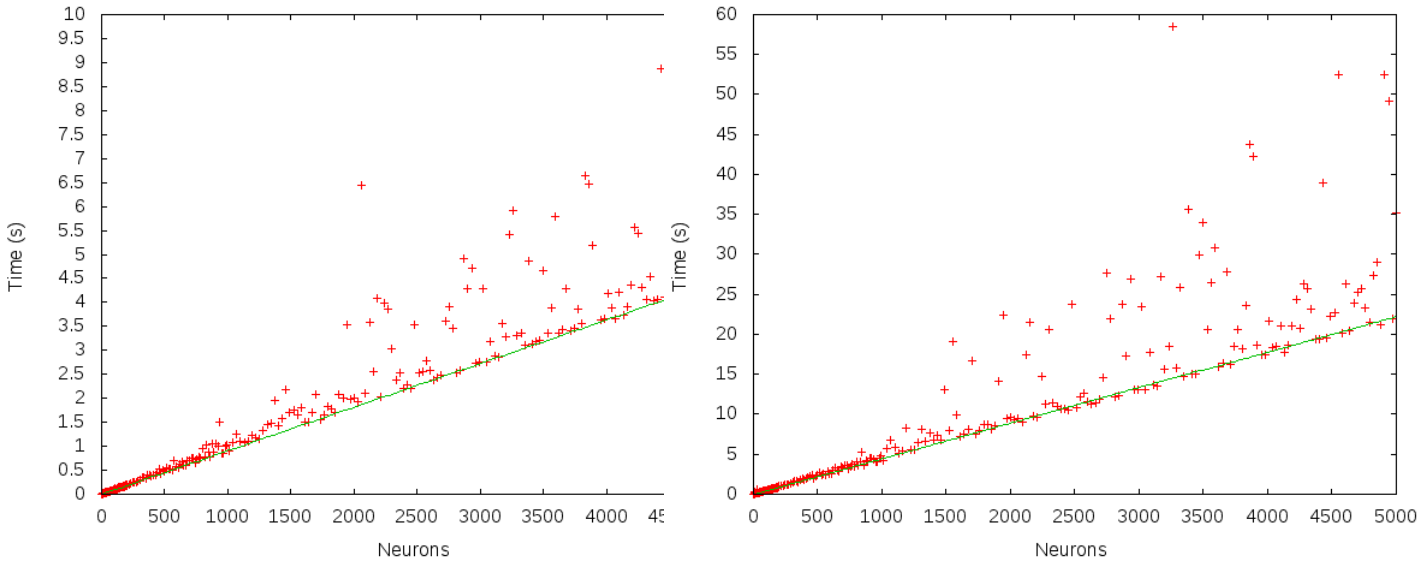


Figure 4.13: Neuron count vs time (ms) with different size patterns: small(upper left), medium(upper right) and big (center)

As it can be appreciated on the above figure, the size of the pattern doesn't really

affect the generated error, obtaining better results when working with a small number of neurons, except in some extreme cases.
The distribution is adjusted to a hyperbolic tangent function.

[First



evaluation: NC, One Layer, Sinus pattern, Default parameters]me

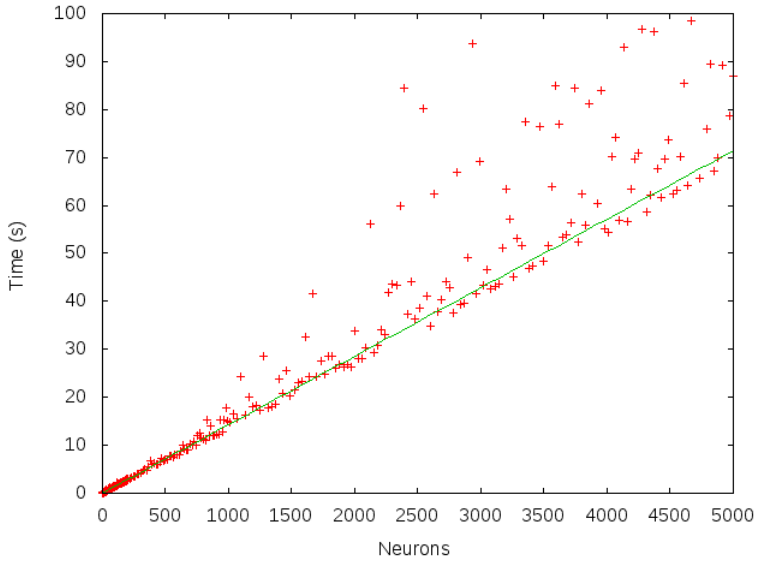


Figure 4.14: Time needed to train the network when using different size patterns: small(upper left), medium(upper right) and big (center)

As it can be appreciated on the above figure, the size of the pattern increase the learning process needed time, with a factor of 50x for a large data set.
The distribution is adjusted to a linear function: increasing the number of neurons would increase linearly the time needed for the learning process.

For the mail pattern, very similar results were obtained.

- **Grid values**
  - Sinus pattern As it can be appreciated on the above figure, the size of the pattern

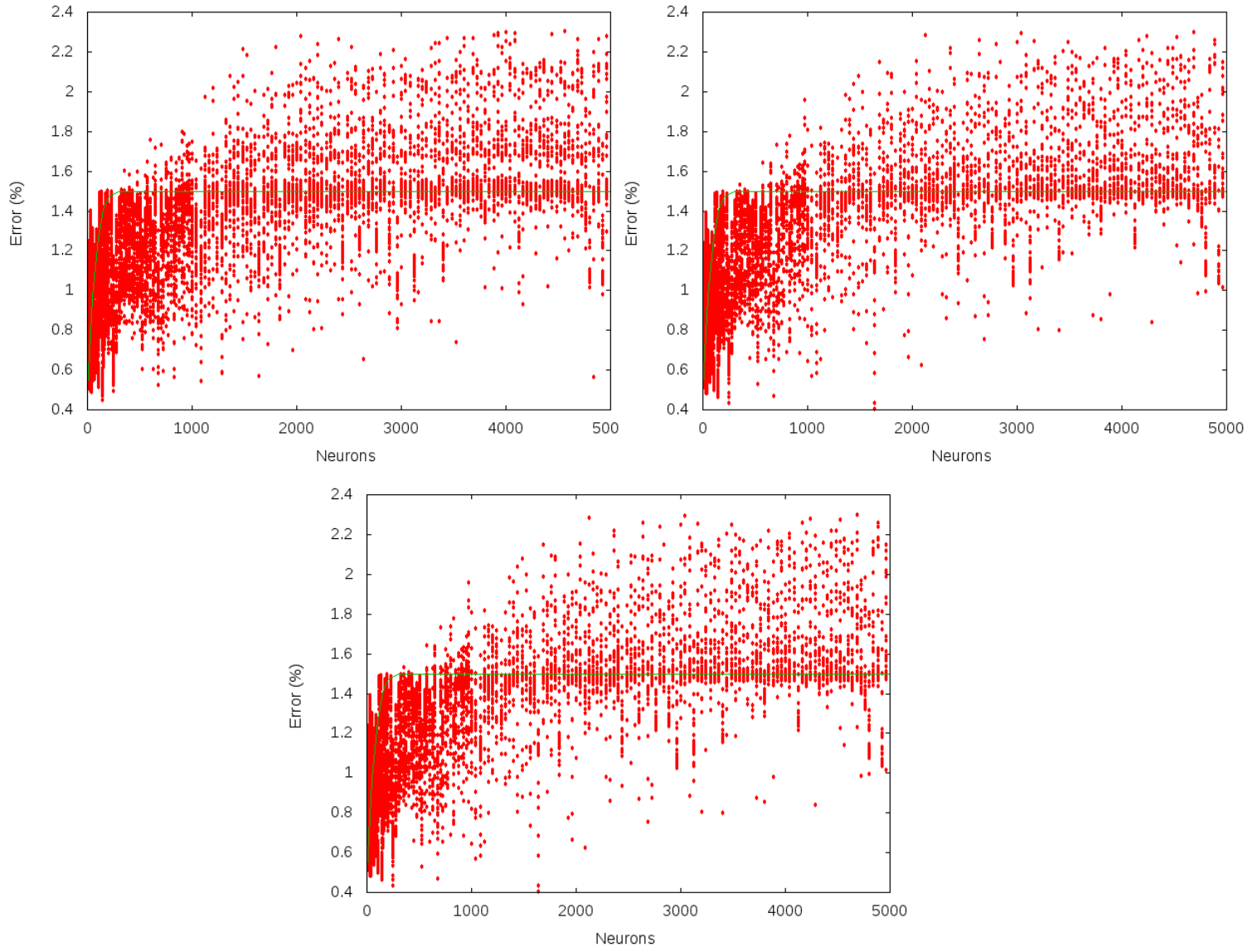


Figure 4.15: Neuron count vs time (ms) with different size patterns: small(upper left), medium(upper right) and big (center)

doesn't really affect the generated error, obtaining better results when working with a small number of neurons, except in some extreme cases.
The distribution is adjusted to a hyperbolic tangent function.

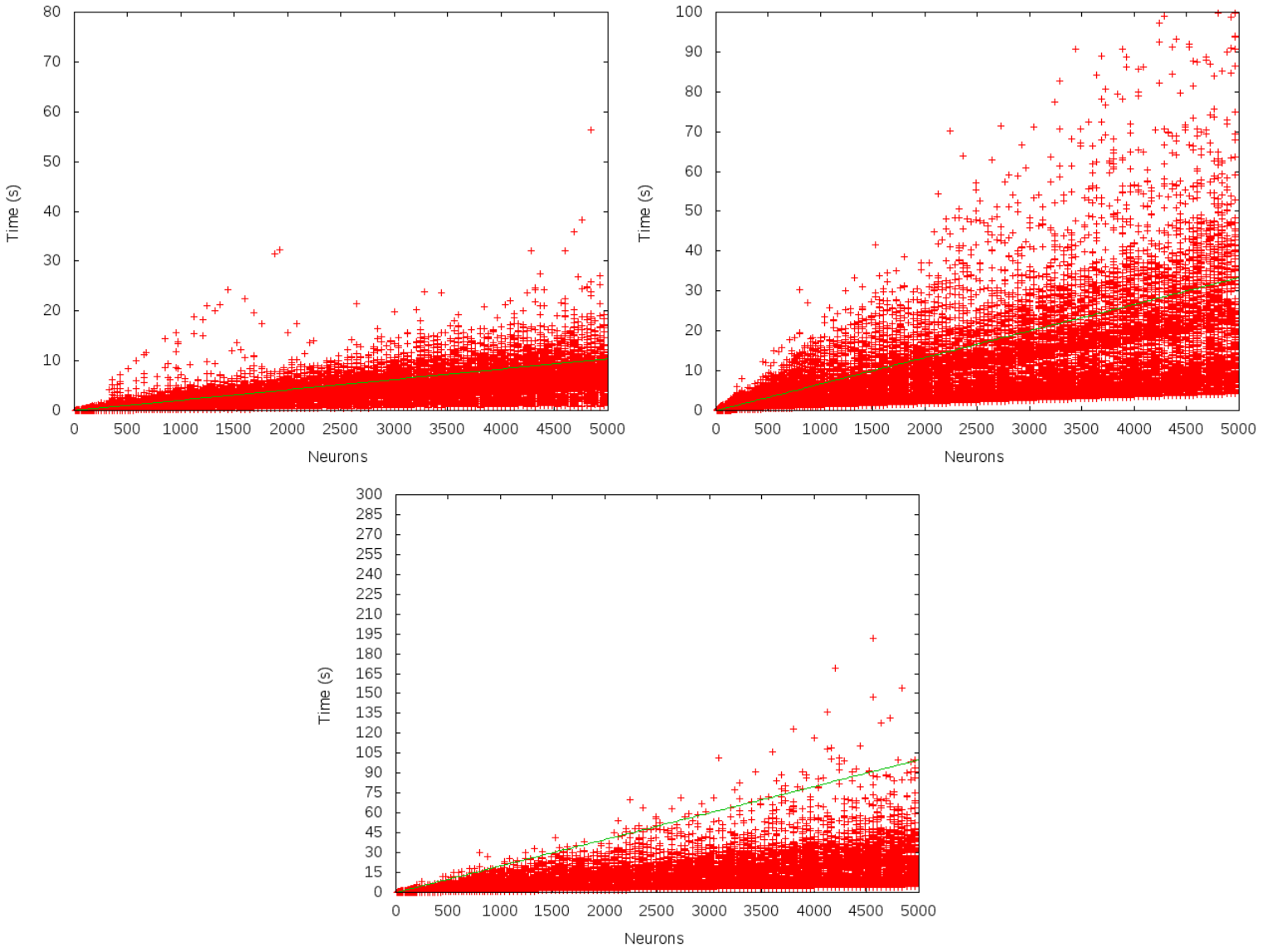


Figure 4.16: Time needed to train the network using different size patterns: small(upper left), medium(upper right) and big (center)

As it can be appreciated on the above figure, the size of the pattern increase the learning process needed time, with a factor of 3x for a large data set. The differences between larger data in mail patterns set are not as important as the differences when using a sinus pattern.
The distribution is adjusted to a linear function: increasing the number of neurons would increase linearly the time needed for the learning process.

- Mail pattern

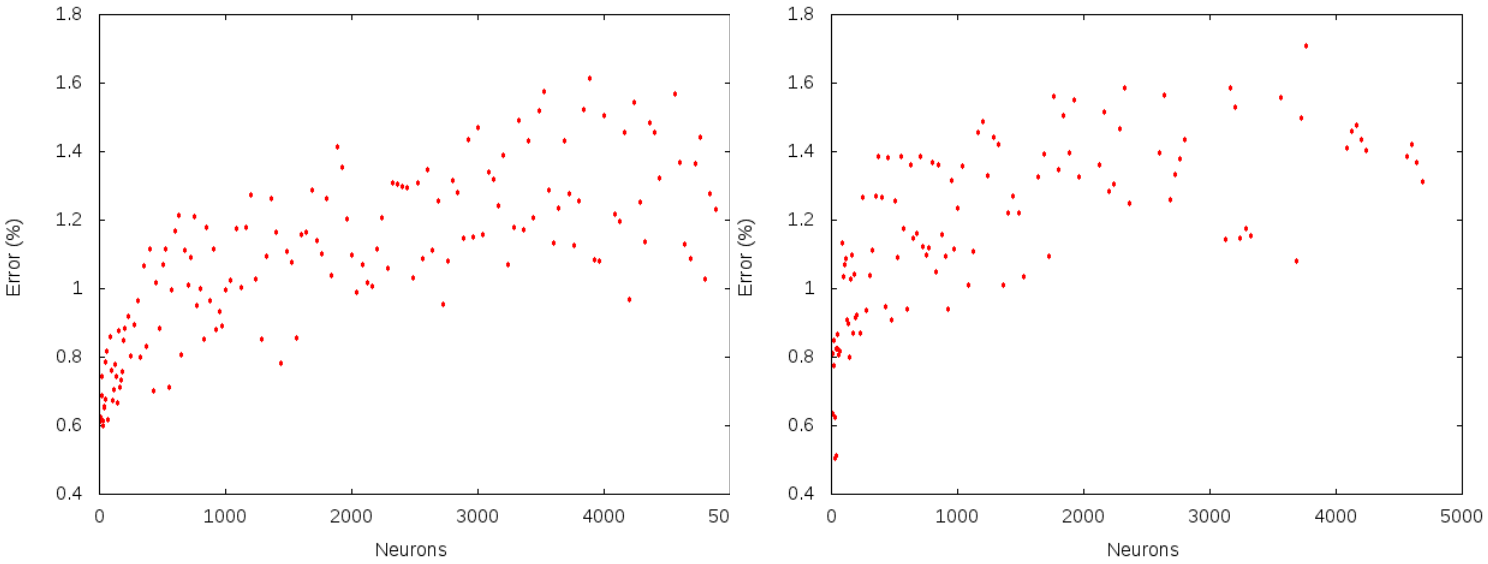


Figure 4.17: Neuron count vs time (s) with different size patterns: small(upper left), medium(upper right)

For a big pattern, it takes a lot of time (even more than 8 hours) to run each simultation. Because of this time-consuming process, results won't be calculated.
As it can be appreciated on the above figure, the size of the pattern doesn't really affect the generated error, obtaining better results when working with a small number of neurons, except in some extreme cases. However, the median of the calculated error is higher when using a bigger pattern, but the minimum can be found when training the network with a bigger pattern.
The distribution seems to be adjusted to a logarithmic function.

- **CMA-ES**
  Sinus pattern

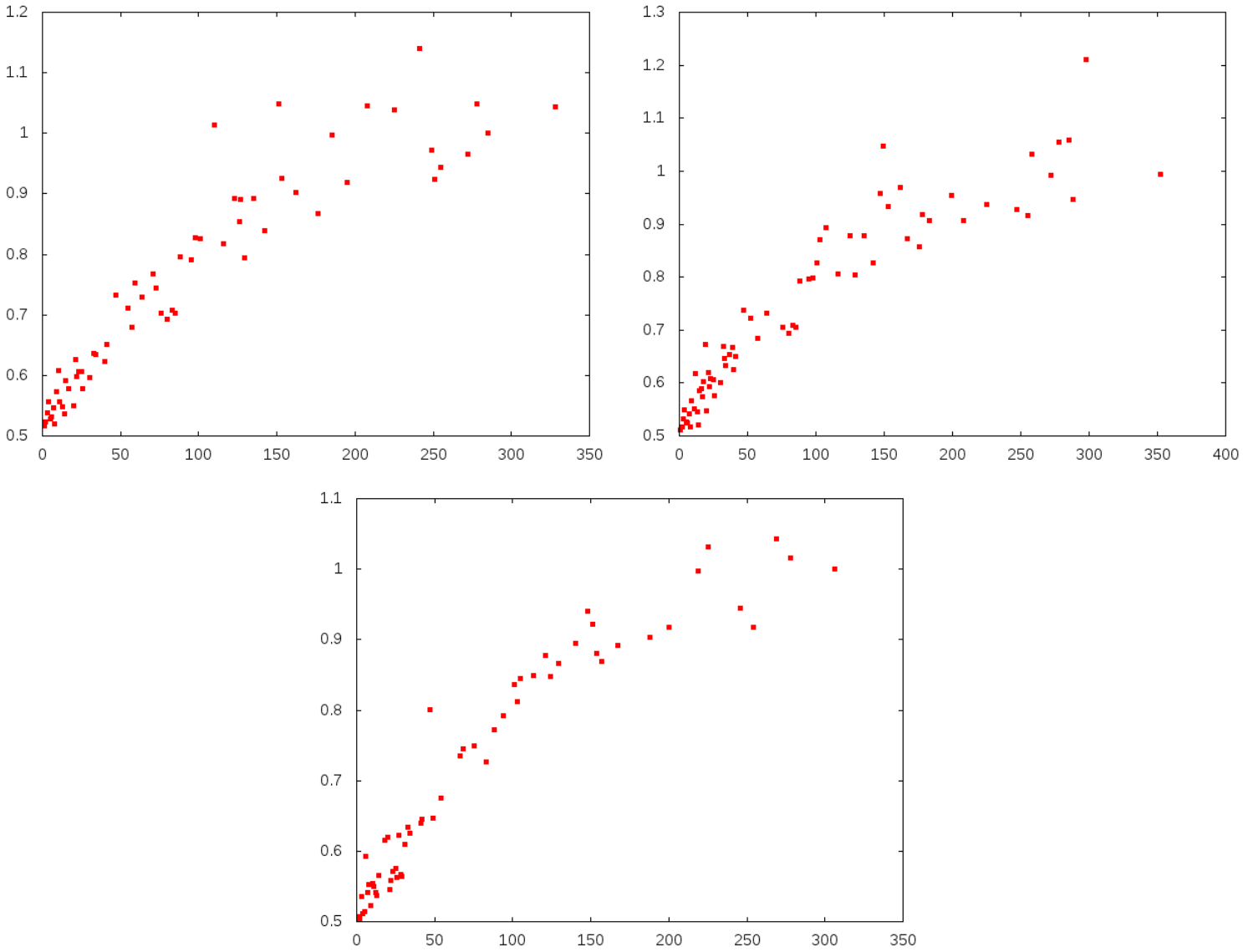


Figure 4.18: Network configuration for minimum error using different patterns: small(upper left), medium(upper right) and big (center)

Mail pattern

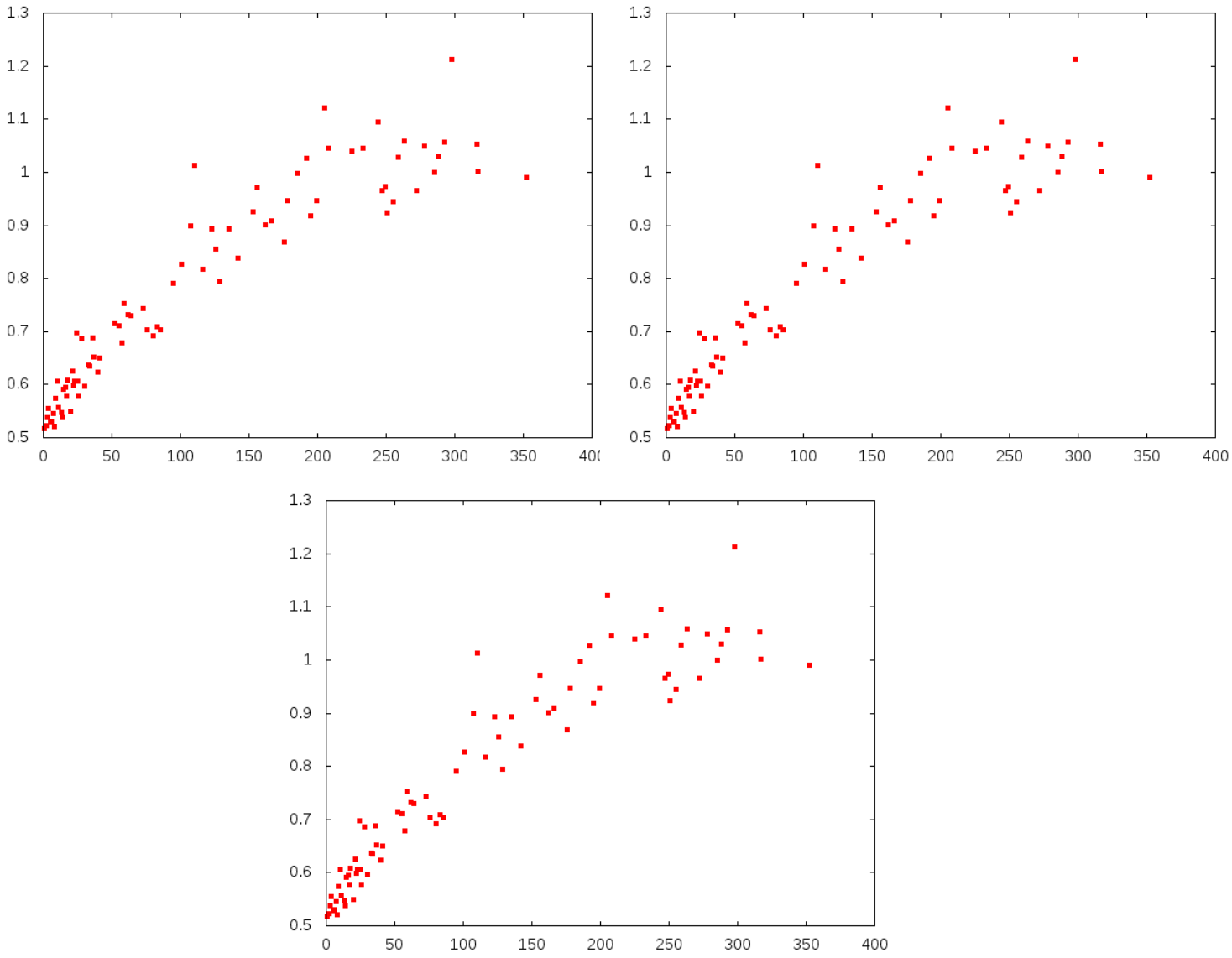


Figure 4.19: Network configuration for minimum error using different patterns: small(upper left), medium(upper right) and big (center)

Two hidden layers

- **Default parameters**
  - Sinus pattern

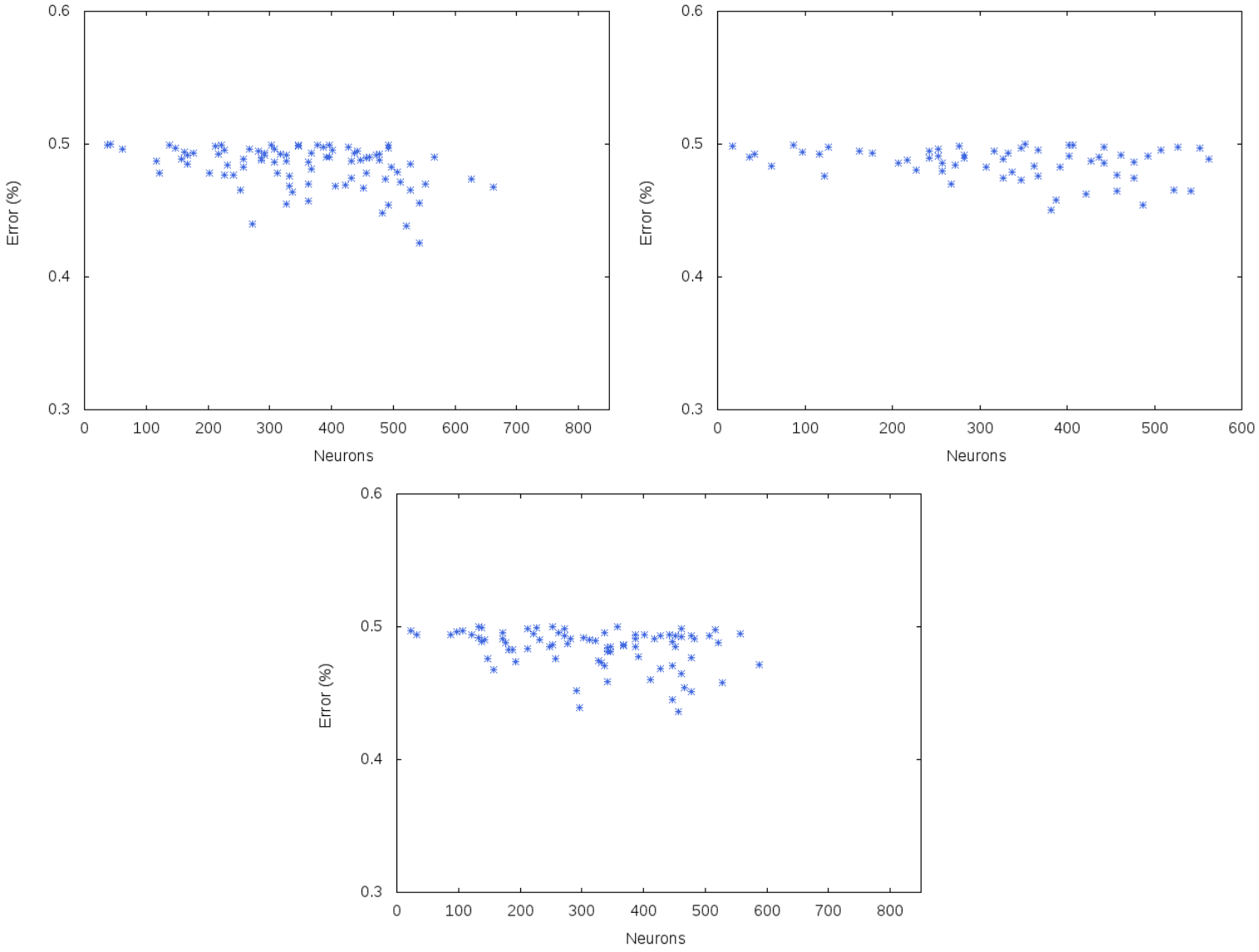


Figure 4.20: Neurons configuration where the calculated error is less than 0.5, using different size patterns: small(upper left), medium(upper right) and big (center)

As it can be appreciated on the above figure, the size of the pattern doesn't really affect the results, which are even similar when using different pattern sizes.

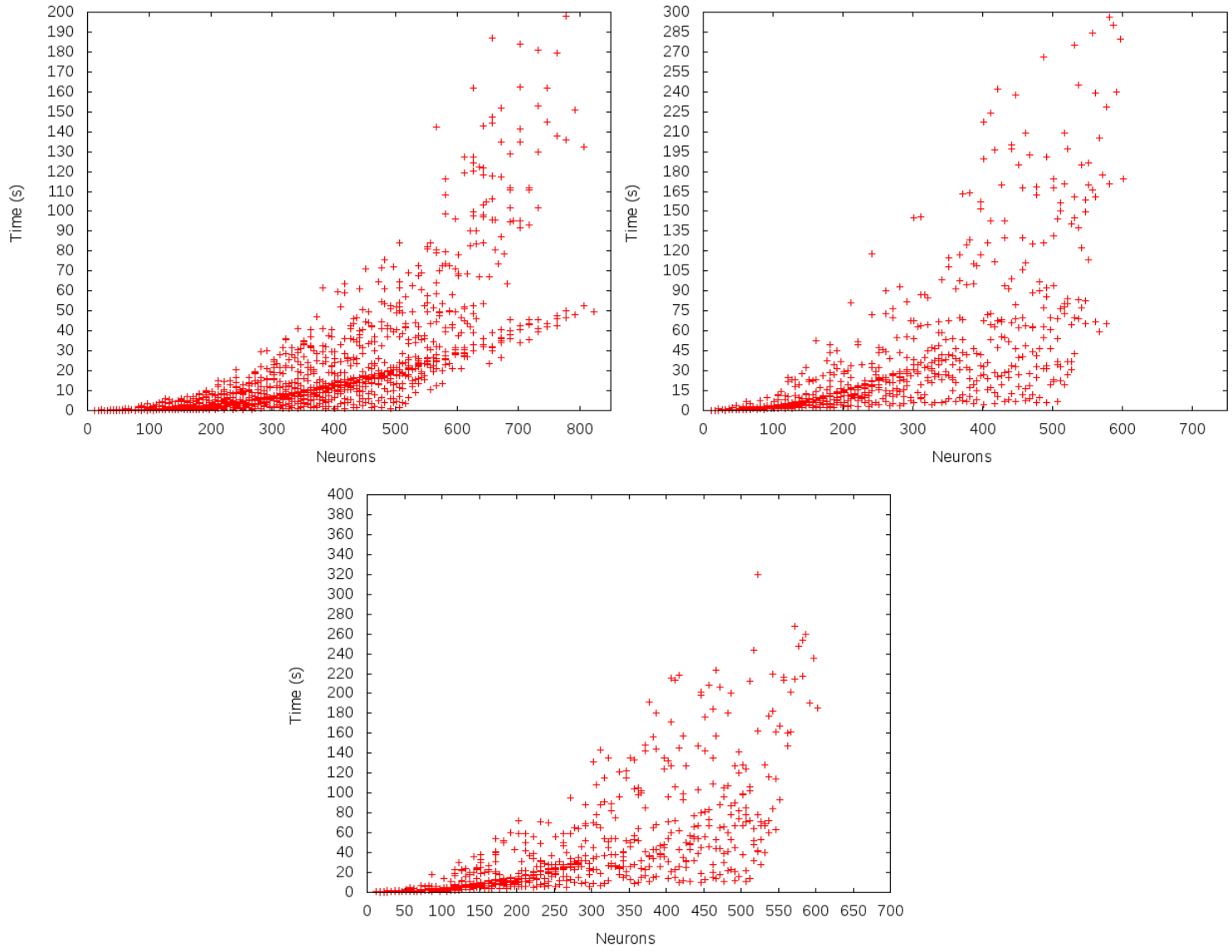


Figure 4.21: Time required to train the network when using different size patterns: small(upper left), medium(upper right) and big (center)

As it can be appreciated on the above figure, the size of the pattern increase the learning process needed time, with a factor of 2x for a large data set.
The distribution is adjusted to an exponential function: increasing the number of neurons would increase exponentially the time needed for the learning process.

Very similar results are obtained when repeating the tests for three hidden layers, with different evaluation methods (grid, CMA-ES), and with different patterns (sinus, mail).

4.5.5 First analysis

- Learning rate:
  If we observe the results, we can appreciate that best network configurations (regarding runtime and error) are obtained when working with low learning rates, i.e. 0.01. Up to 0.1 values, the network configuration doesn't seem to be really affected. Unfortunately, in the previous tests we cannot appreciate how the learning rate parameter affects the network between 0.01 and 0.1.

- Momentum:
  Values between 0.3 and 0.6 produce good network configurations, as it can be appreciated in the previous tests. In the literature [1],[3] we have seen that best results are obtained when using momentum values of 0.5. This corresponds with the results we presented here.

- Flat spot elimination:
  Although it doesn't exist a big improvement, slightly better results are obtained when working with FSE rates of 0.1 and 0.5, according to the literature [1],[3].

- Neuron count:
  According to the previous graphs, increasing the number of neurons doesn't seem to produce an improvement in the network. We have the opposite result as the one expected: increasing the number of neurons (or layers) make the training process less efficient, that means, higher error rates, and a higher training time. In fact, even for the low error rates obtained, the error is still too big, with values 5 times above 0.1, which should be the optimum.

All these studies regarding to the neuron count lead us to the conclusion that something's not working as expected.

4.6 Reformulating the problem

Different factors could be the source of the results obtained:

- Errors on the implementation:
  The backtracking algorithm could be implemented incorrectly. After some tests, and checking it step by step, tests have shown that it's working exactly as expected.

- Input pattern misconfiguration:
  Choosing a bad pattern could lead to a bad training. After checking the sinus pattern, mainly used on the previous tests, it was appreciated that a more optimum pattern could be used. Repeating some tests showed an improvement on the training network process.

- Iterations number:
  If the iteration number is not big enough, the training process could stop before an optimum network has been obtained. Increasing the number of iterations proved that the error was drastically reduced.

Therefore, the tests were redesigned according to this reformulation, and to the different evaluation cases we want to check now:

- Different input patterns:
  Sinus pattern will be redesigned in order to have better results. Samples between 0 and 2π will be more granulated. Also, the network will be trained to resolve the two spirals problem, as explained in the following section (4.6.2).

- Grid evaluation instead of CMA-ES:
  Previous results showed how different learning parameters affect the training process. That results are still valid, therefore, the optimization will be focused on that values (i.e. LR between 0.01 and 0.1, Momentum around 0.5, FSE of 0.1), and CMA-ES is not useful in this case.

- Different number of iterations:
  Increasing the number of iterations can lead to an accurate network, but also increases the learning time. Therefore, the number of iterations is also a parameter to take into account when optimizing a network.

4.6.1 Sinus and Mail pattern

One hidden layer

- **Default parameters**
  - Sinus pattern: 10.000 iterations

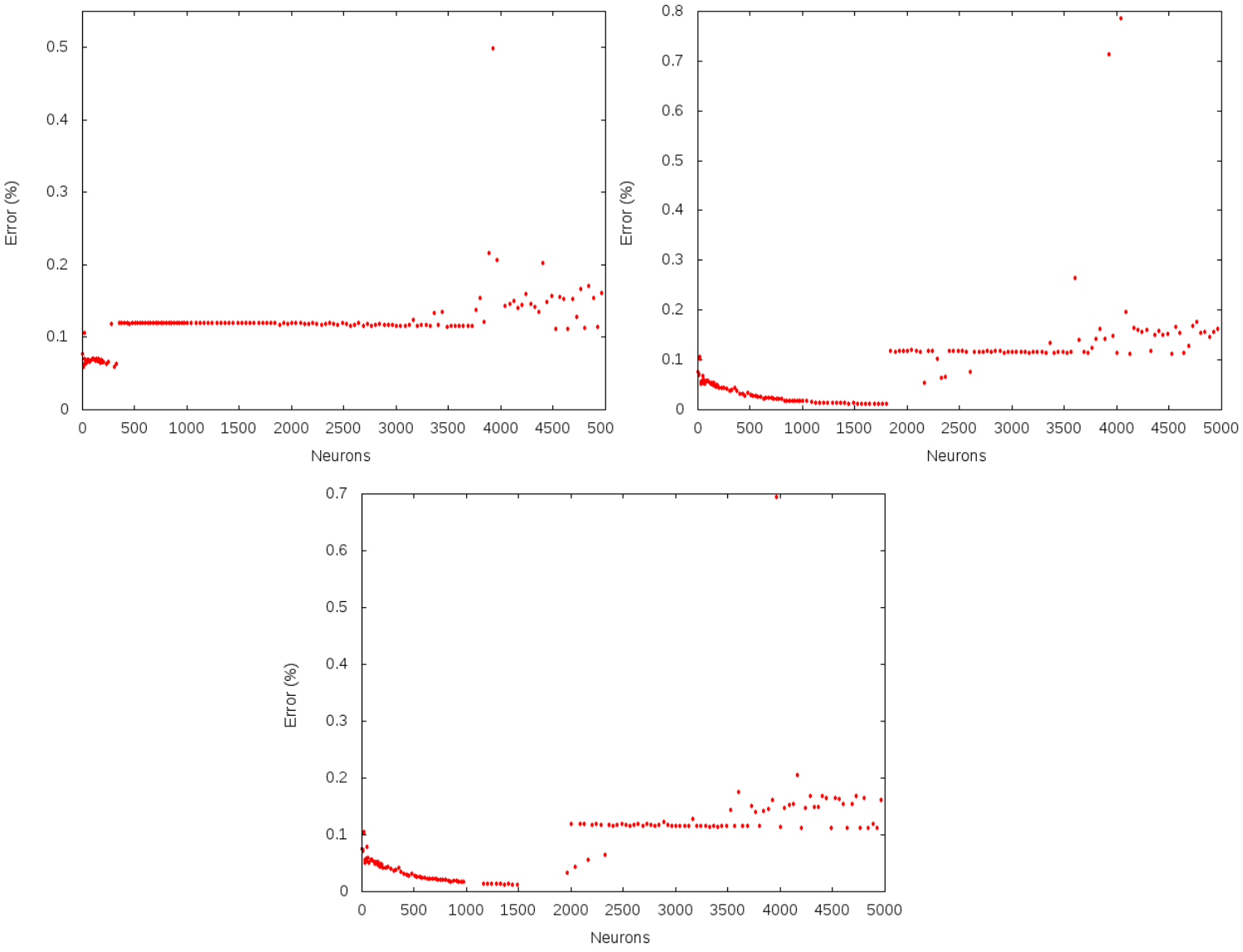


Figure 4.22: Neuron count vs time (ms) with different size patterns: small(upper left), medium(upper right) and big (center)

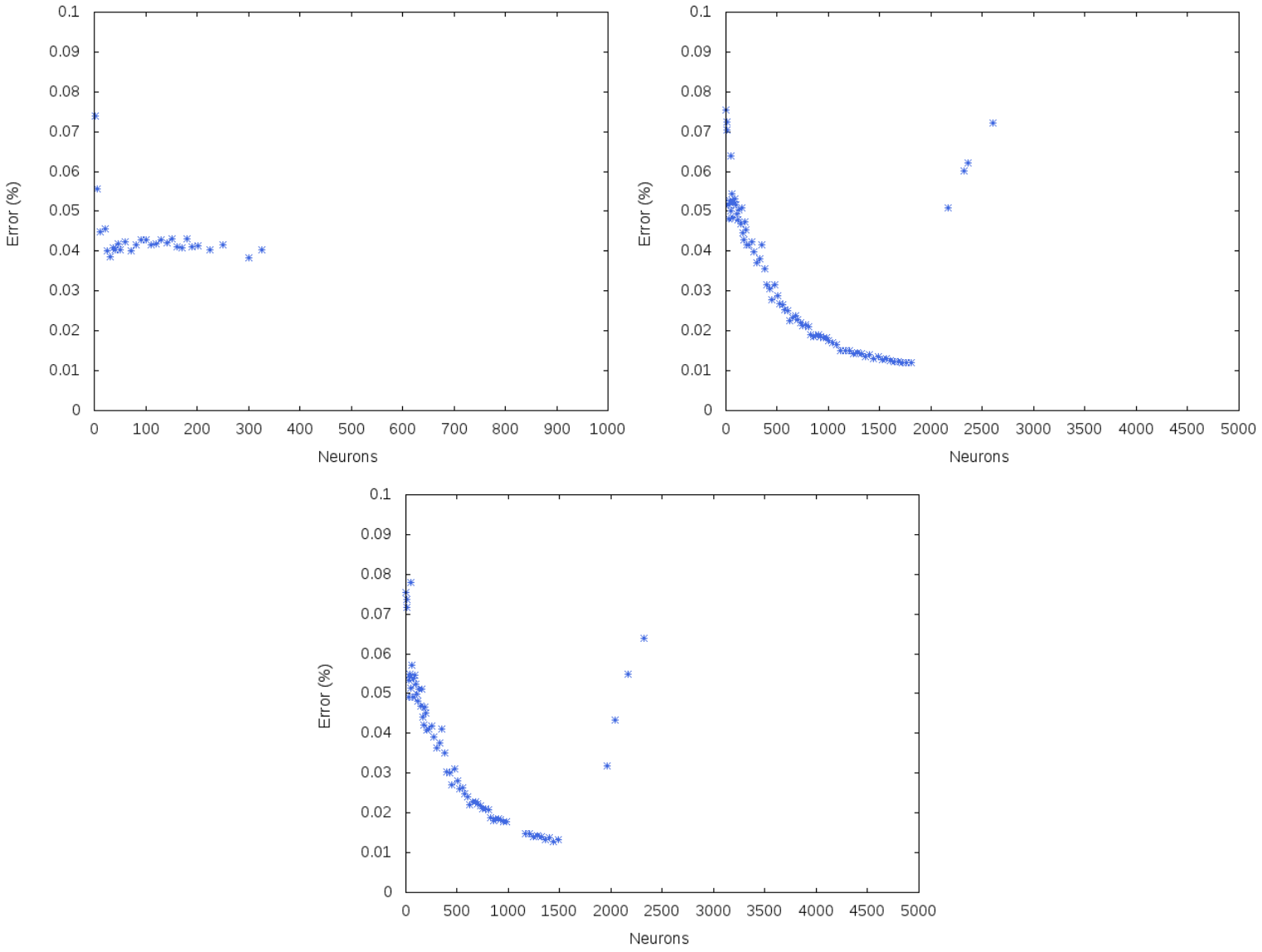


Figure 4.23: Neurons configuration where the calculated error is less than 0.1, using different size patterns: small(upper left), medium(upper right) and big (center)

In the previous graph it's pictured how the size of the pattern influences the training. With an input size big enough, we can generate an accurate trained network. It seems that up to 2000 neurons, the network doesn't produce better results, due to over-fitting.

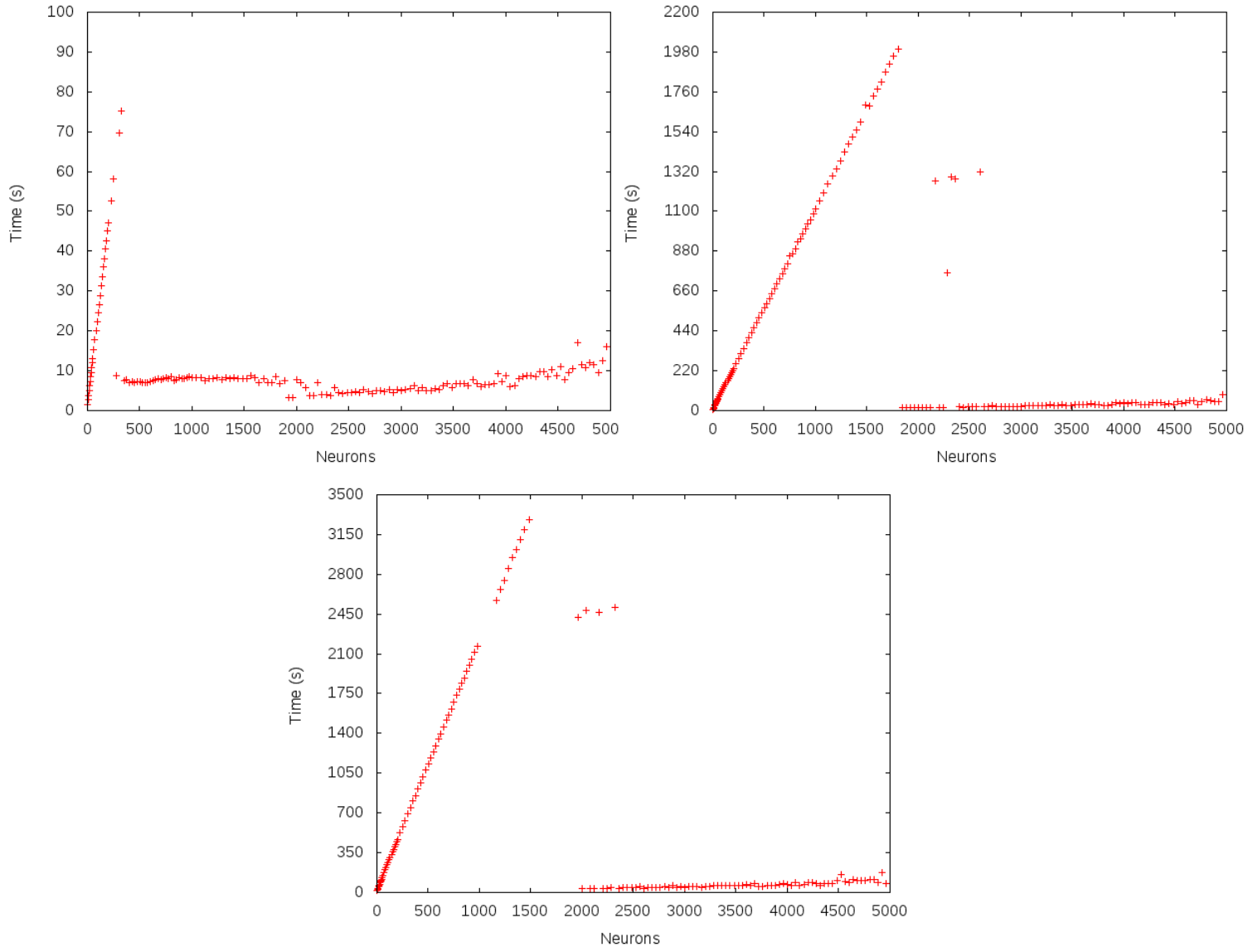


Figure 4.24: Time needed to train the network when using different size patterns: small(upper left), medium(upper right) and big (center)

The training time of the network increases linearly, and also depends of the size of the input pattern: a higher input sizes implies a slower training process.

- Sinus pattern: 20.000 iterations

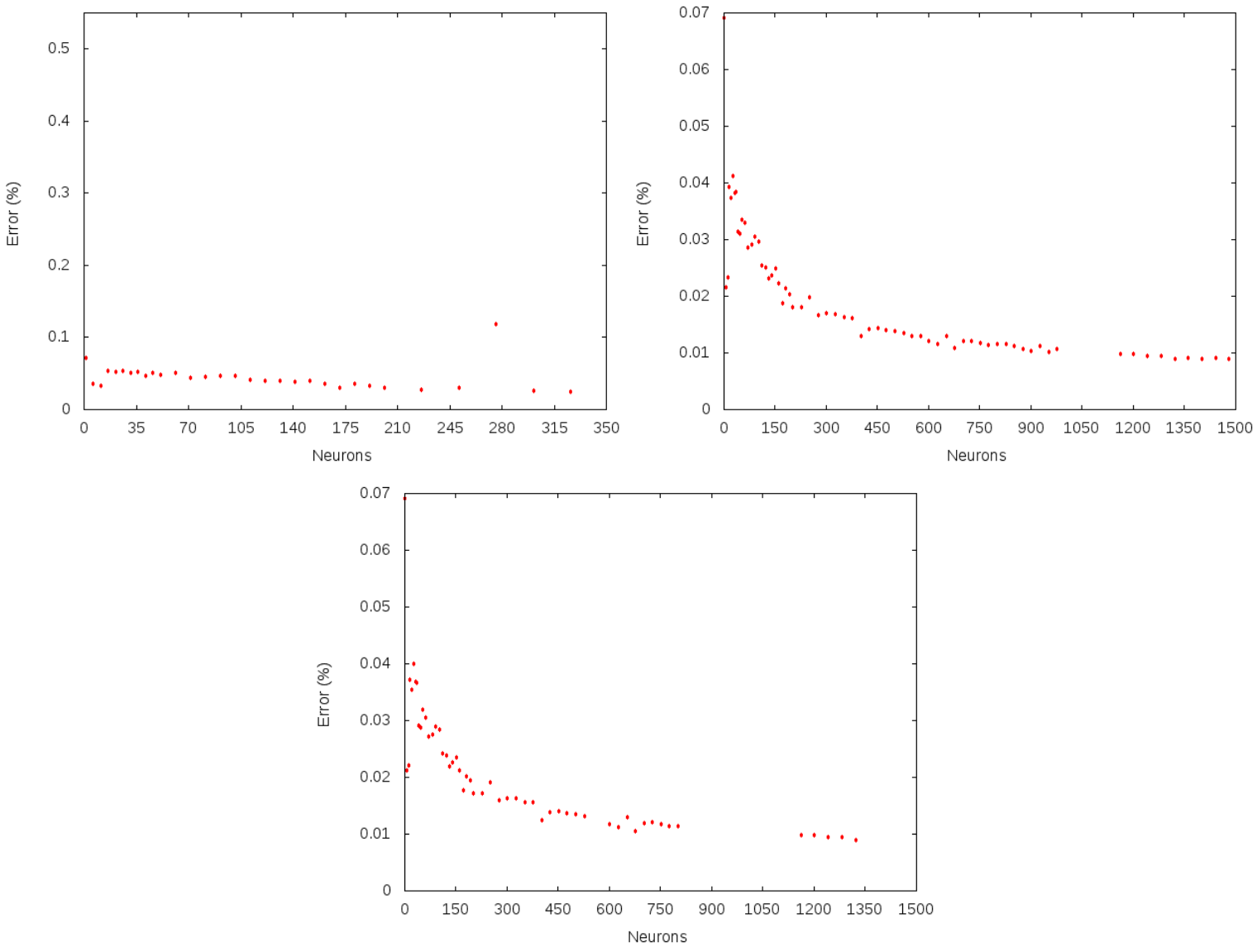


Figure 4.25: Neuron count vs time (ms) with different size patterns: small(upper left), medium(upper right) and big (center)

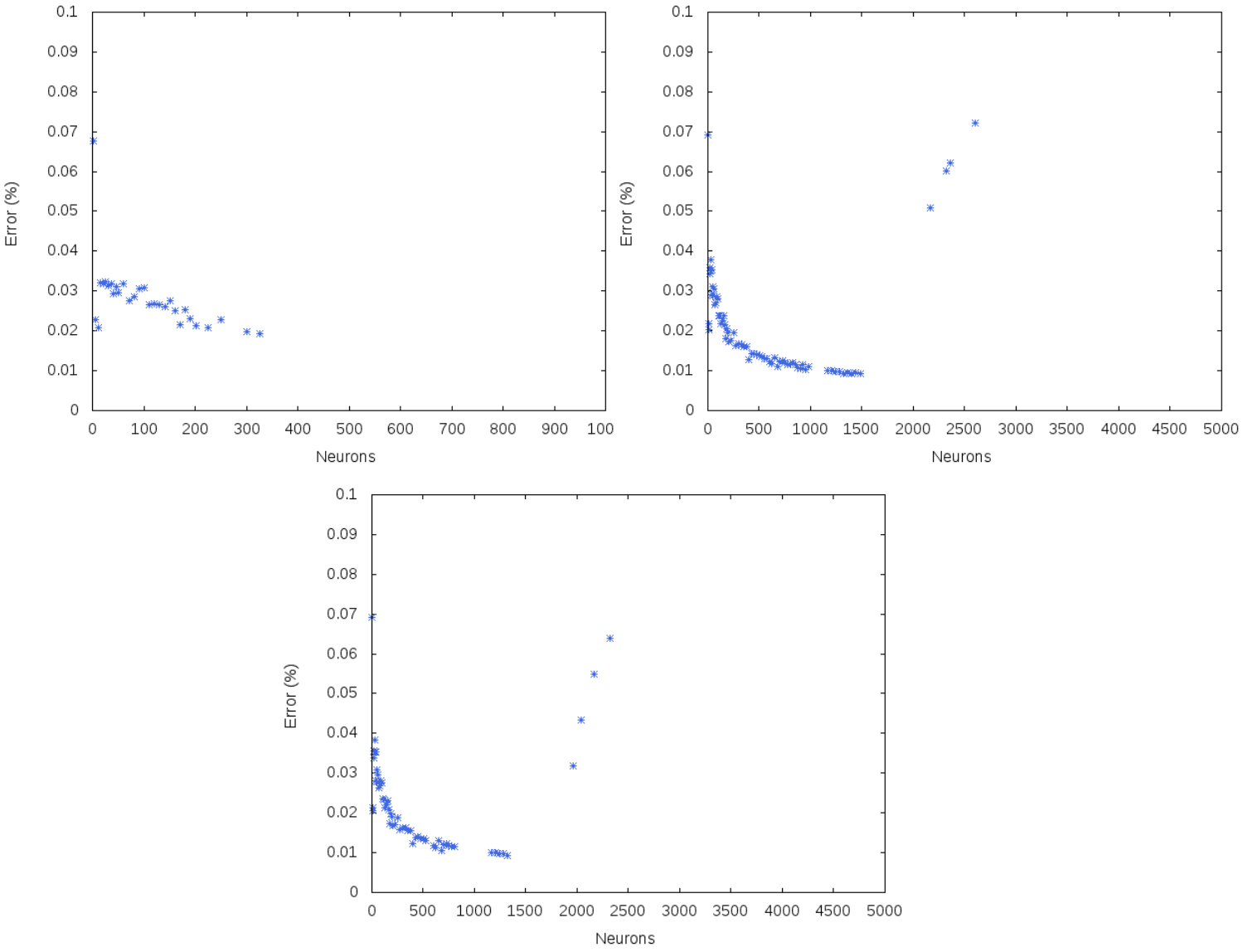


Figure 4.26: Neurons configuration where the calculated error is less than 0.1, using different size patterns: small(upper left), medium(upper right) and big (center)

In the previous graph it's pictured how the size of the pattern influences the training. With an input size big enough, we can get an accurate trained network. It seems that up to 2000 neurons, the network doesn't produce better results, due to over-fitting. Compared with a network that uses a lower number of epochs, although the error rates are the same, the neurons needed to reach that rate are fewer.

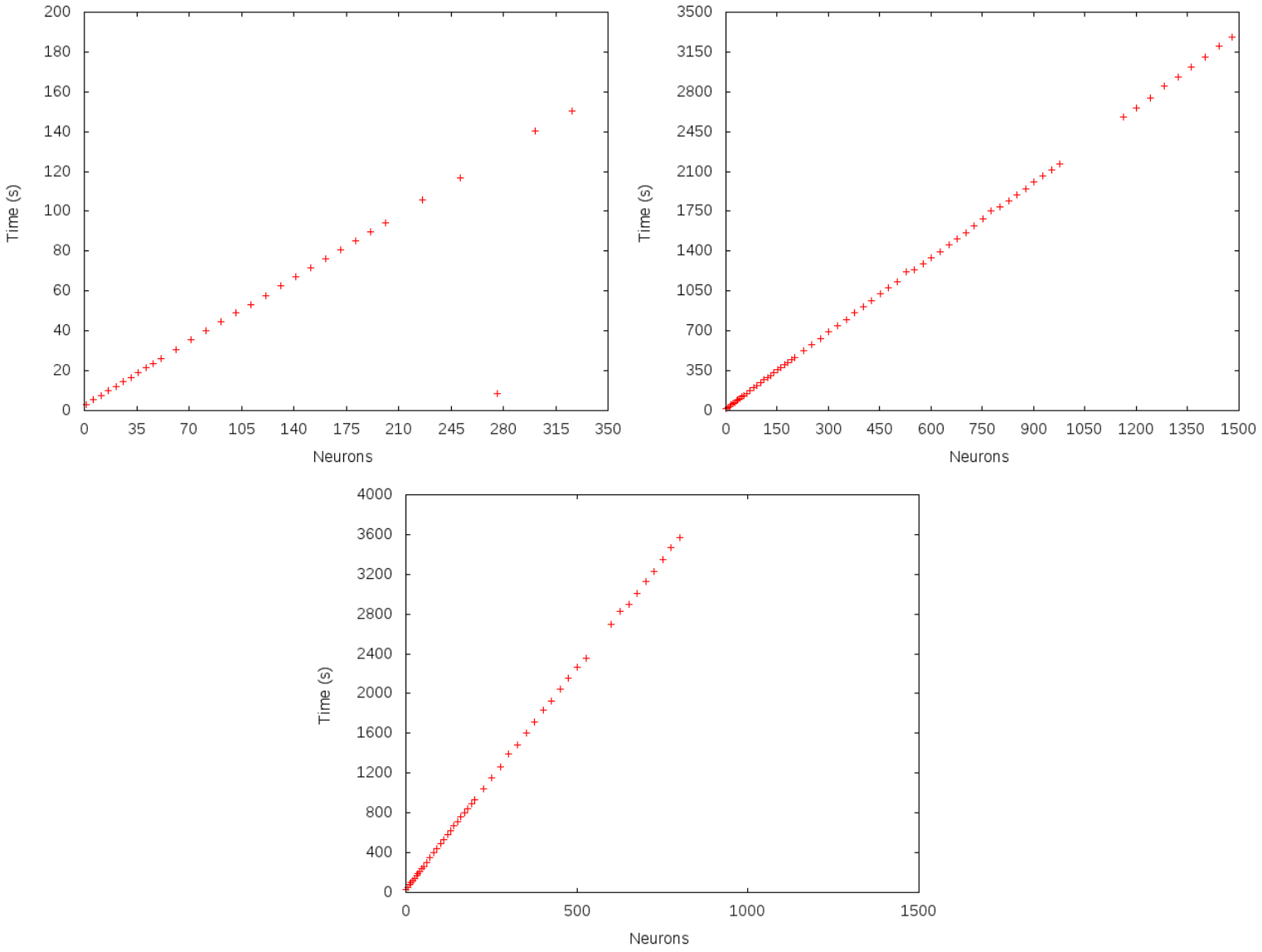


Figure 4.27: Time needed to train the network when using different size patterns: small(upper left), medium(upper right) and big (center)

As previously pictured, the bigger the input pattern is, the higher the time needed for the training is.

- Sinus pattern: 50.000 iterations

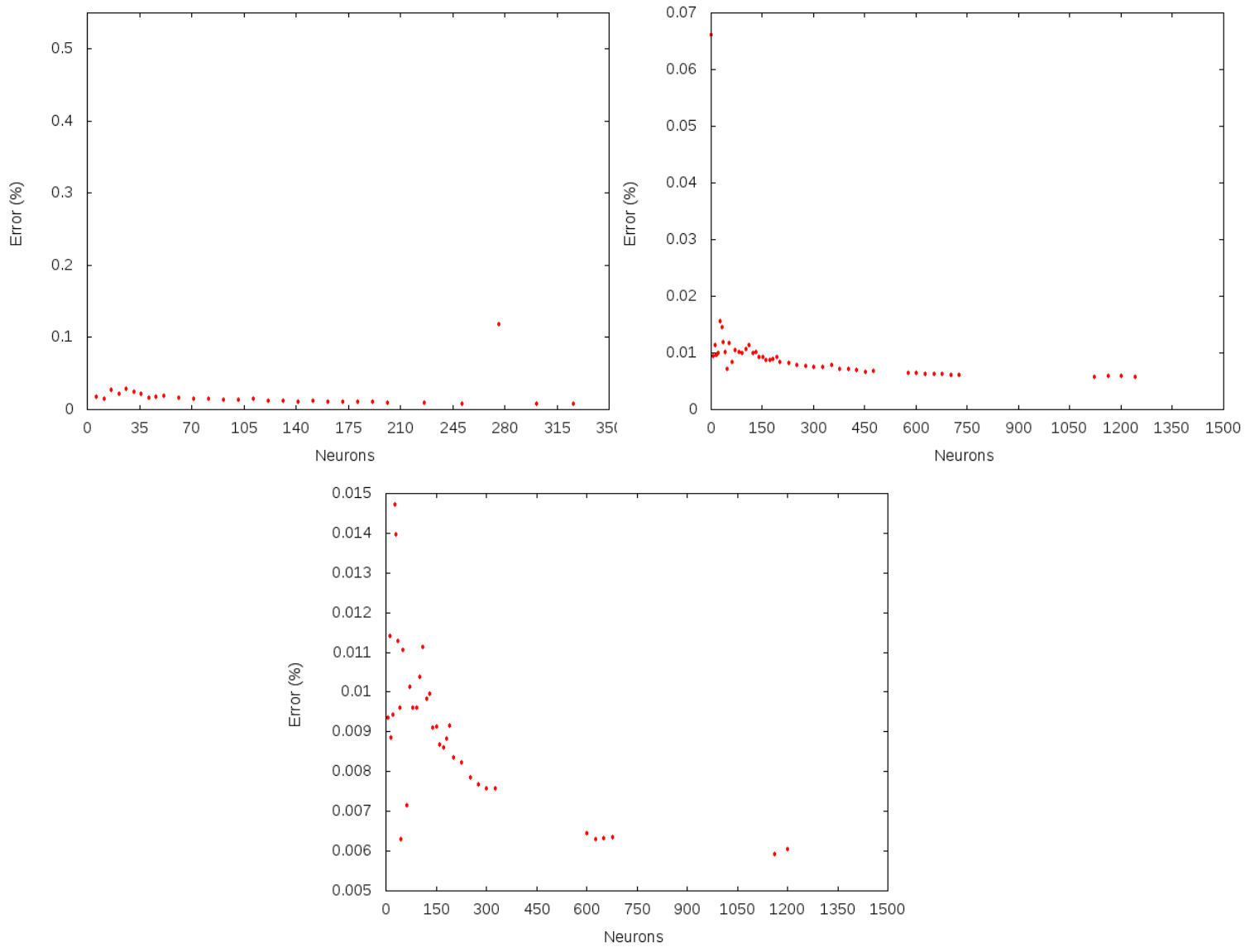


Figure 4.28: Neuron count vs time (ms) with different size patterns: small(upper left), medium(upper right,) and big (center)

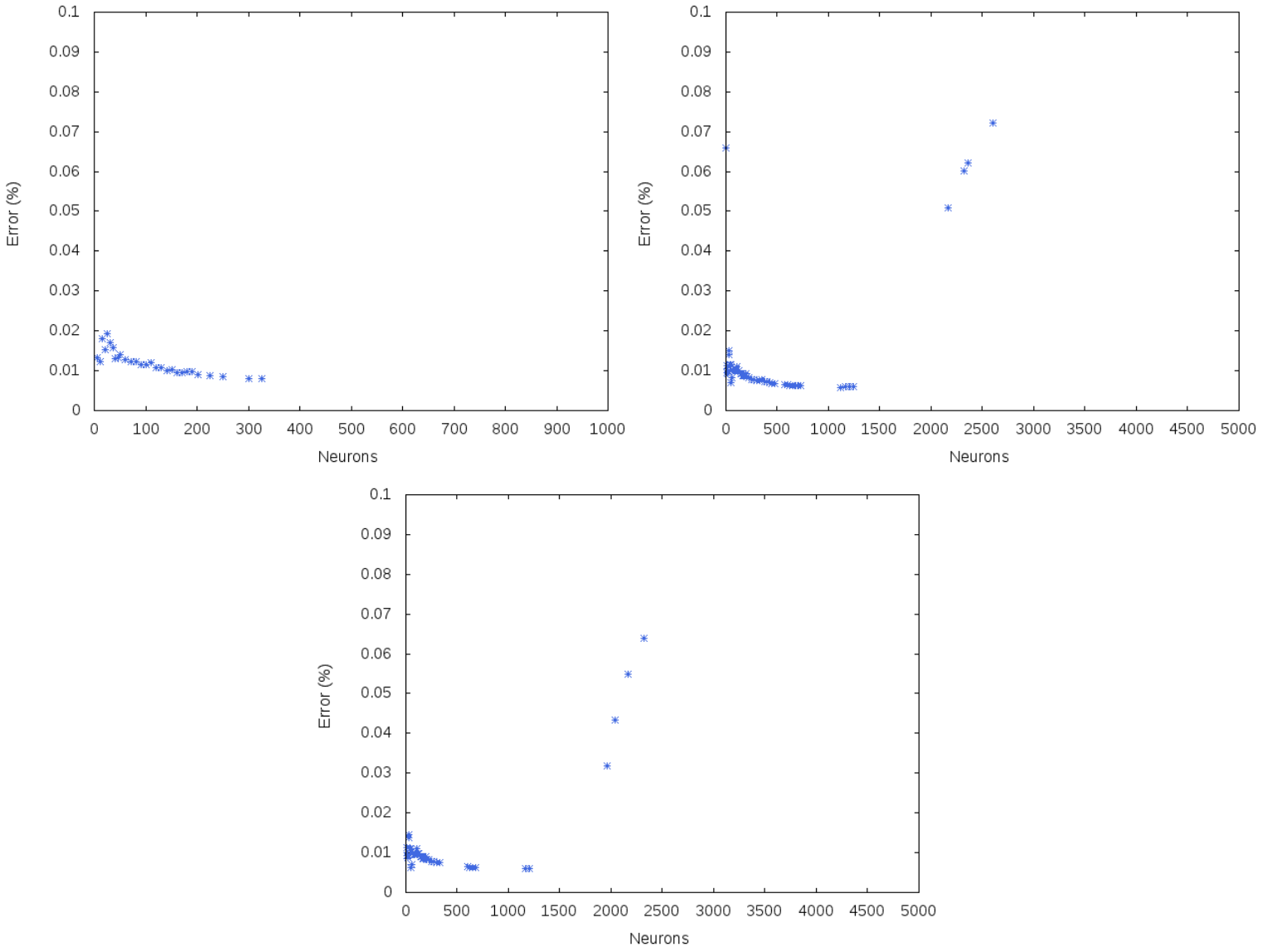


Figure 4.29: Neurons configuration where the calculated error is less than 0.1, using different size patterns: small(upper left), medium(upper right) and big (center)

In the previous graph it's pictured how the size of the pattern influences the training. With an input size big enough, we can get an accurate trained network. Up to 2000 neurons the network stops producing better results due to over-fitting. Compared with a network that uses a lower number of epochs, although the error rates are the same, the neurons needed to reach that rate are fewer.

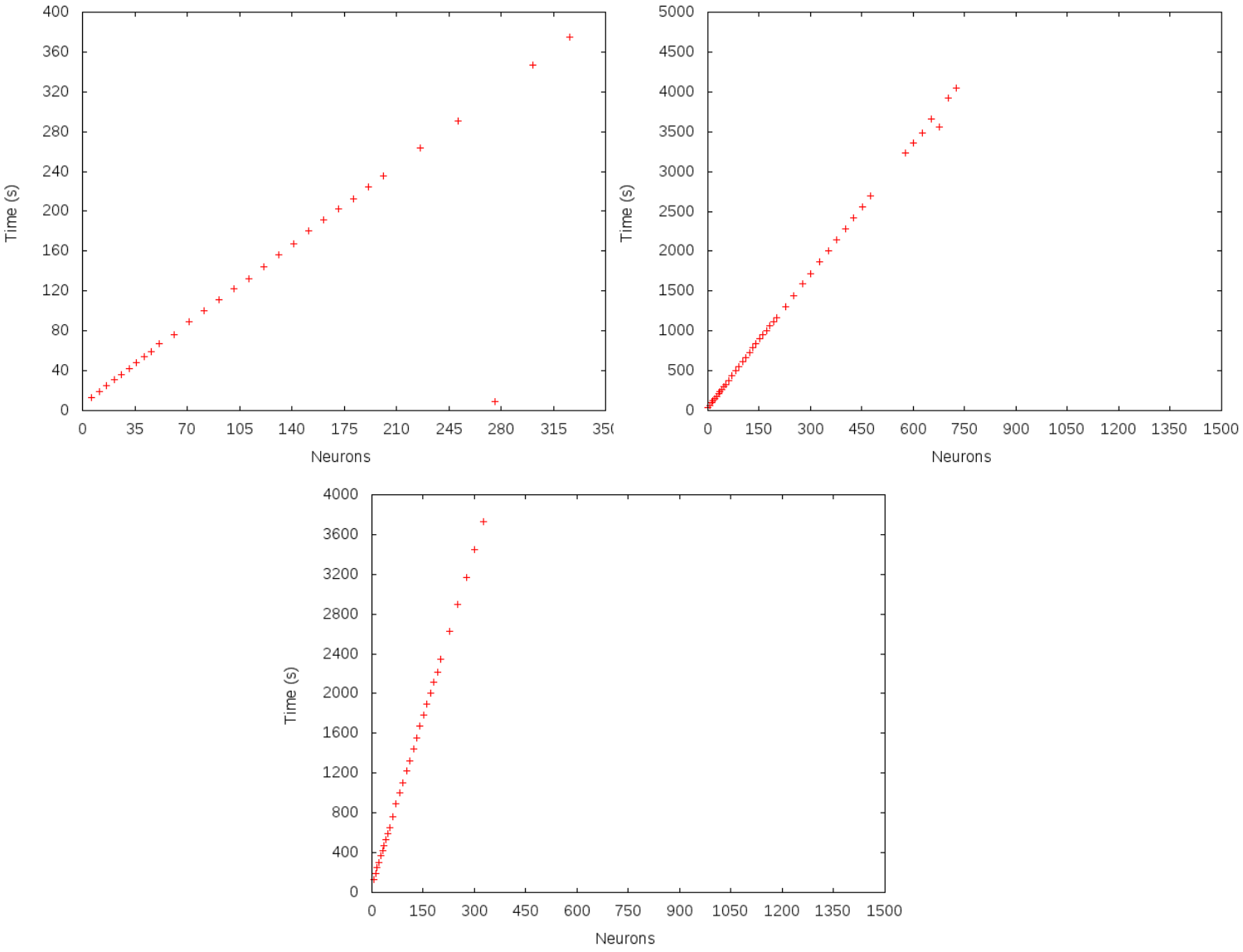


Figure 4.30: Time needed to train the network when using different size patterns: small(upper left), medium(upper right) and big (center)

As previously pictured, the bigger the input pattern is, the higher the time needed for the training is.

- Mail pattern: medium size, 20.000 iterations

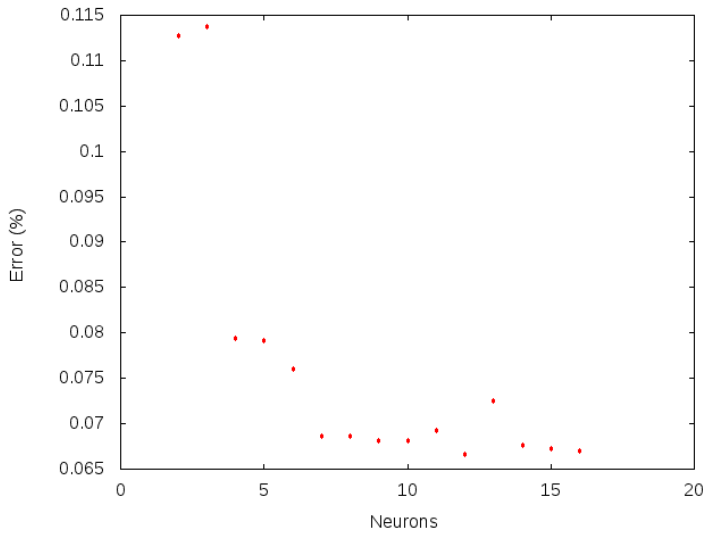


Figure 4.31: Generated error depending on the neurons number for default learning parameters, 20k iterations

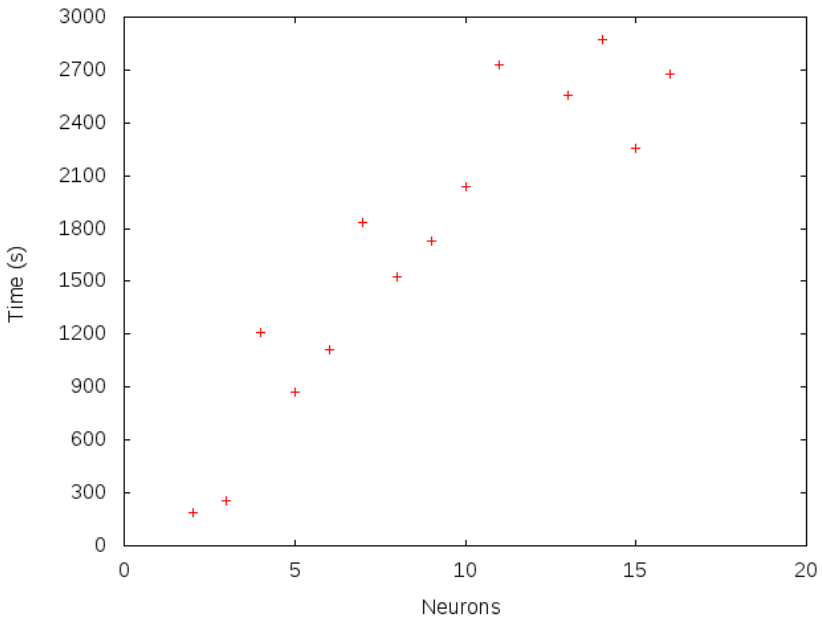


Figure 4.32: Time spent for training, according to the neuron count, 20k iterations

Same conclusions explained before can be applied to this pattern.

- **Grid evaluation**
  - Sinus pattern: medium size, 10.000 iterations

  – **Learning rate**

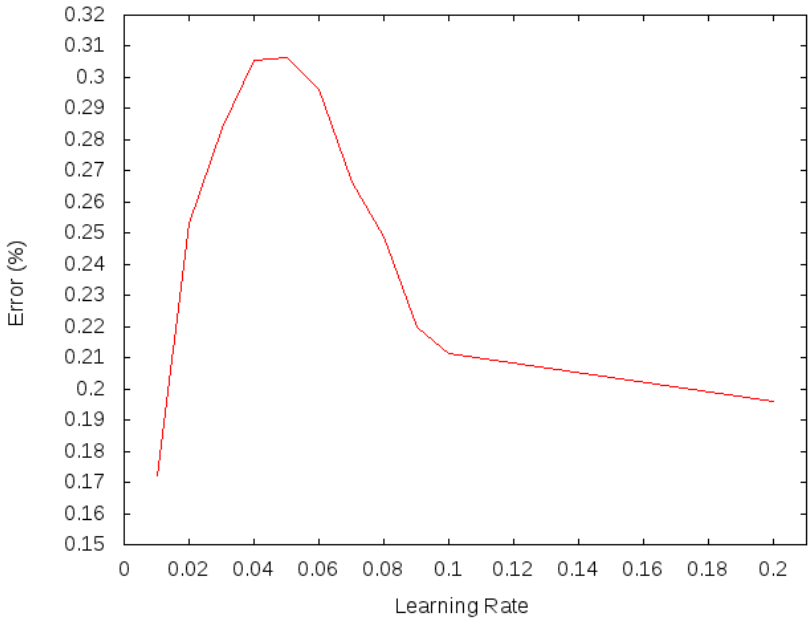


Figure 4.33: Generated error depending on the learning rate

As expected, best learning results are obtained when using a LR between 0 and 0.1, being a special proper one a LR of 0.01

– **Momentum**

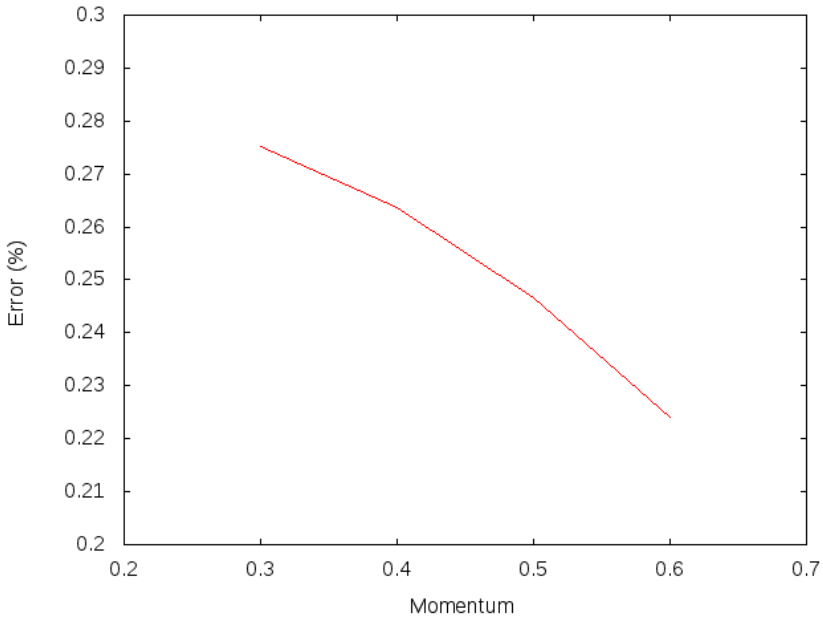


Figure 4.34: Generated error depending on the momentum

As expected due to previous tests, best training results are obtained when working with a momentum of 0.5.

– **Flat spot elimination**
As expected due to previous tests, best training results are obtained when

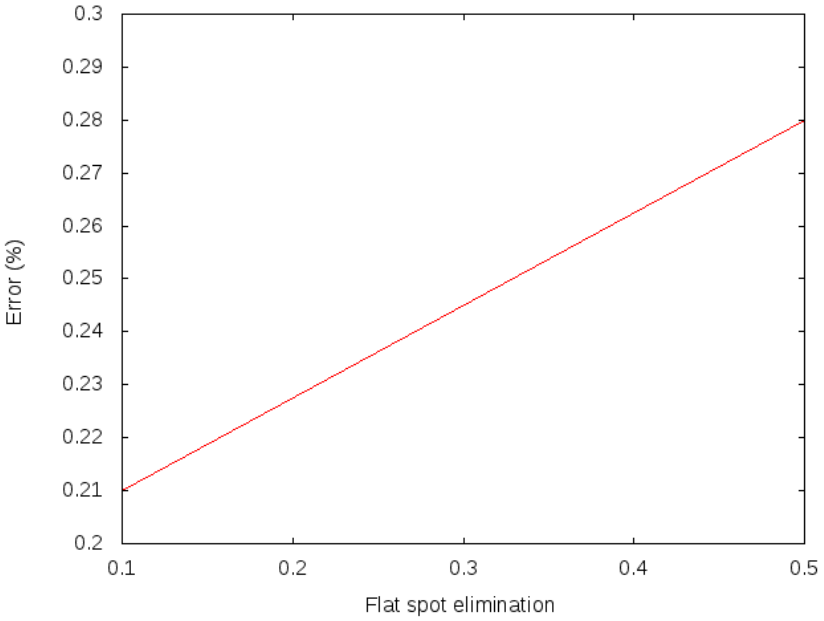


Figure 4.35: Generated error depending on the flat spot elimination rate

working with a FSE of 0.1

– **Neuron count**

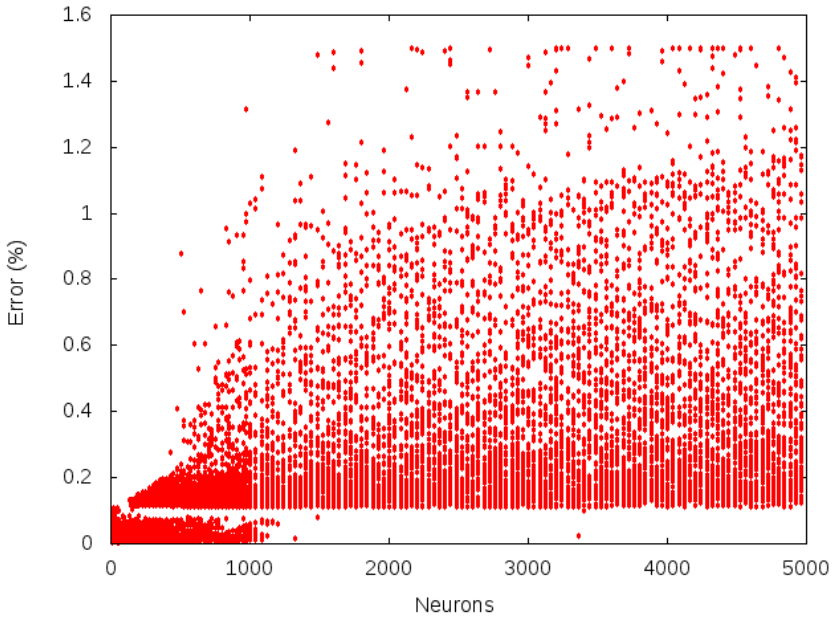


Figure 4.36: Generated error depending on the neurons number for different learning parameters

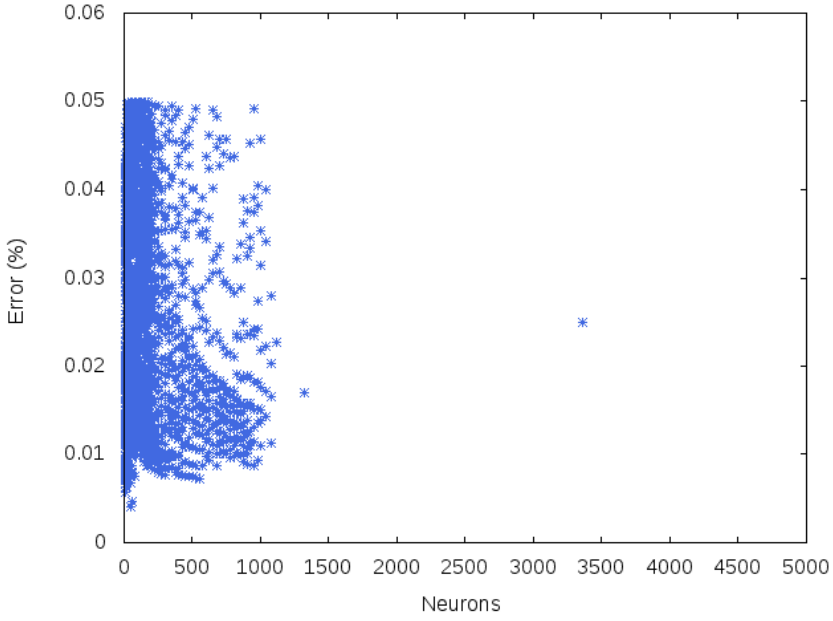


Figure 4.37: Neuron count where the calculated error is smaller than 0.05%

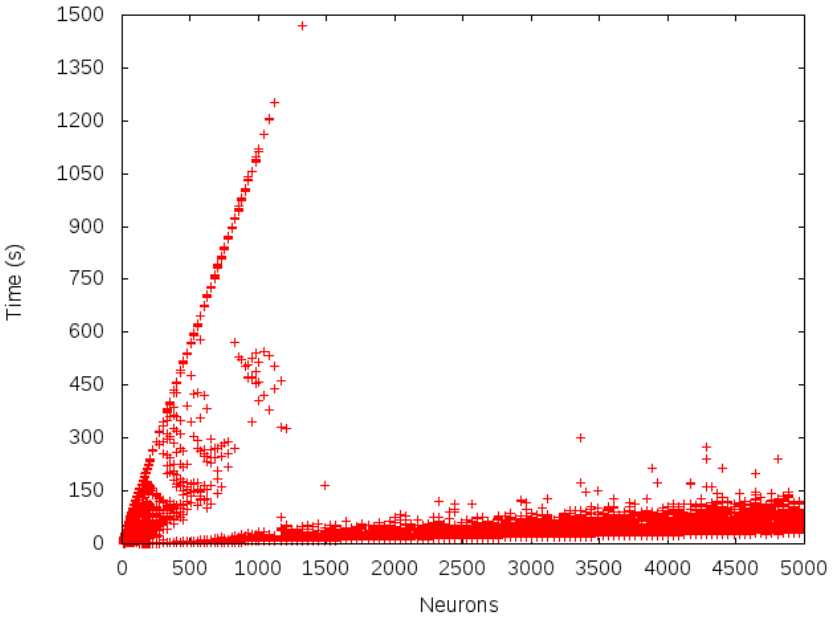


Figure 4.38: Time spent for training, according to the neuron count

Here we can clearly appreciate two separated groups: the first one, for a neuron count under 2000, and optimum network parameters (which corresponds to LR of 0.01, Momentum of 0.5 and FSE of 0.1) and the other group, where either the parameters are not the optimum, or where the network is over-fitted.

Two hidden layers

- **Default parameters**
  - Sinus pattern

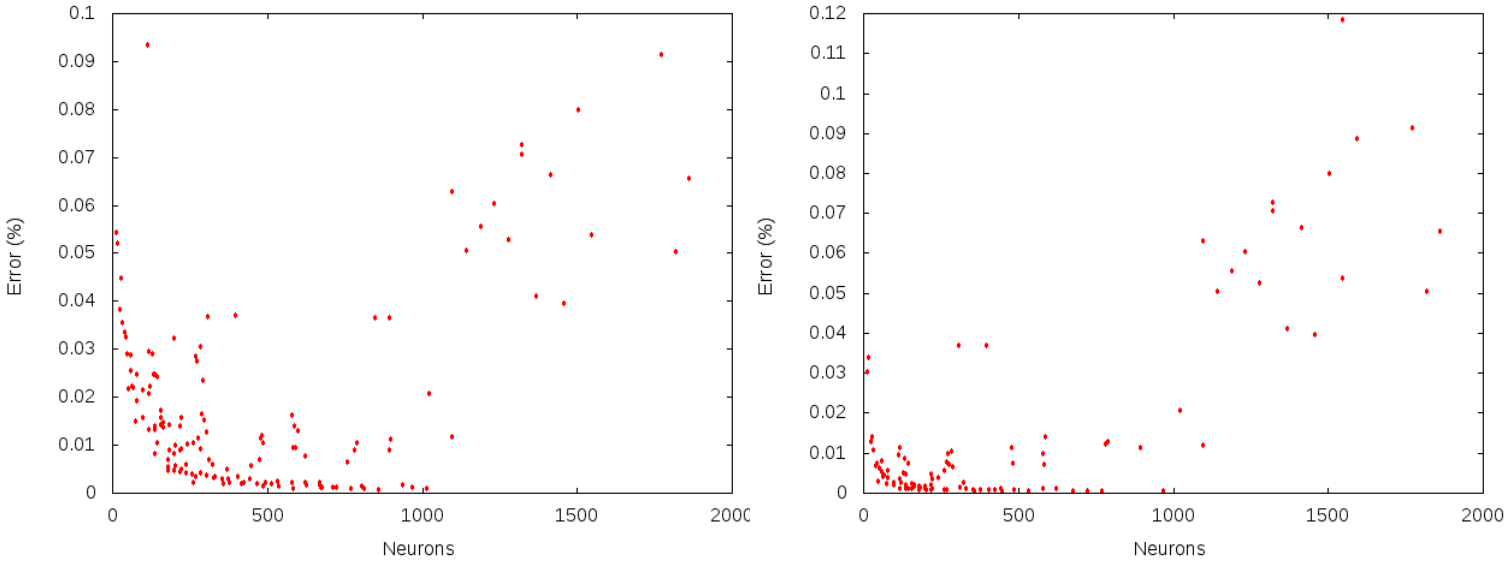


Figure 4.39: Measured error when using different number of iterations: 10.000(left), and 20.000(right)

Increasing the number of iterations produce a faster training (less number of neurons produce an accurate network). Comparing these results with the network trained using one hidden layer, we can appreciate a slightly better behaviour when using two hidden layers instead of one.

As it can easily be concluded, a higher number of hidden networks implies a higher number of weights, and therefore, more calculations. All these produce a sower training process.

Depending on the learning configuration, the network will produce better results, and keep iterating over the network, therefore, more time for training will be needed, as it's pictured in the graph.

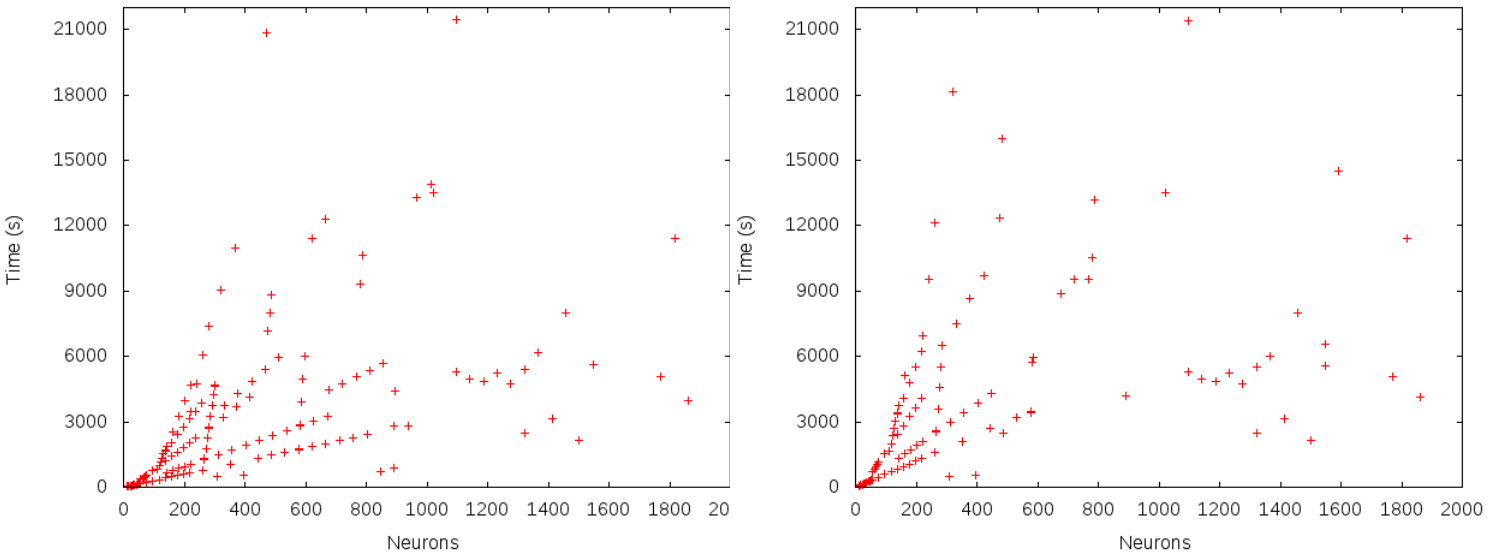


Figure 4.40: Time spent for training when using different number of iterations: 10.000(left), and 20.000(right)

- **Grid evaluation**
  - Sinus pattern: 10.000 iterations

  – **Learning rate**

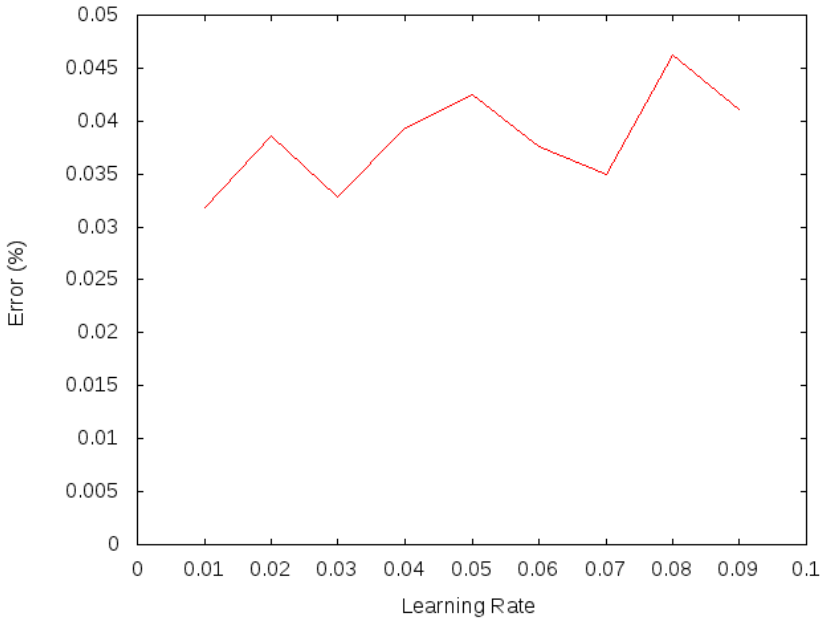


Figure 4.41: Generated error depending on the learning rate

Best results are obtained when working with learning rates of 0.01. Increasing the LR to values higher than 0.1 highly increase the obtained error.

– **Momentum**
Best results are obtained when working with momentum values between 0.4

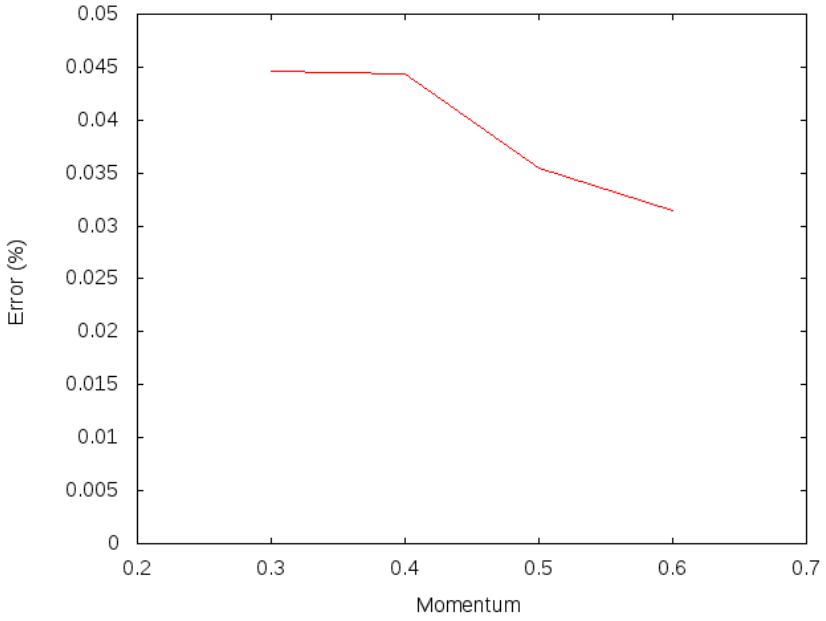


Figure 4.42: Generated error depending on the momentum

and 0.6. However, there's not a big difference between the error rates when using different momentum values.

– **Flat spot elimination**
Differences between error times when working with different flat spot elimina-

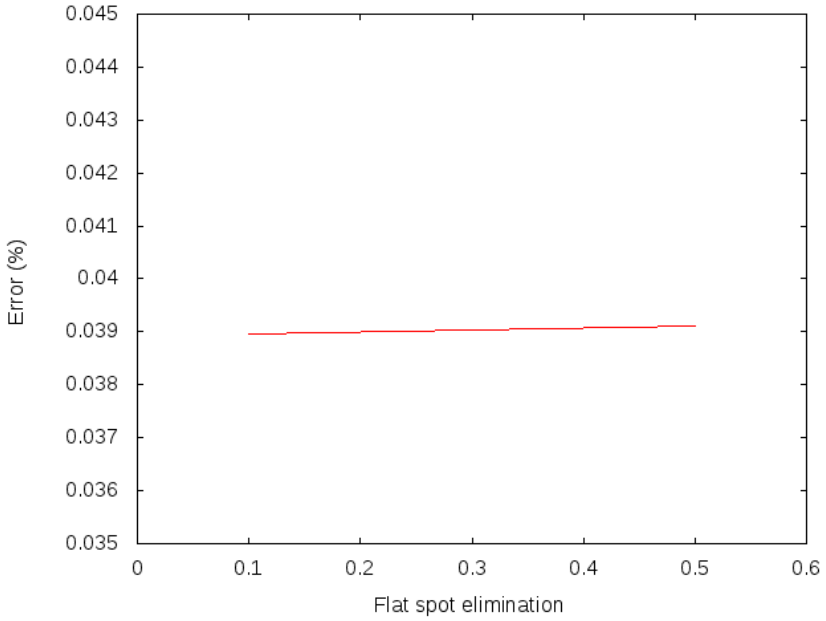


Figure 4.43: Generated error depending on the flat spot elimination rate

tion rates are not relevant, as it can be appreciated on the graph above.

– **Neuron count**

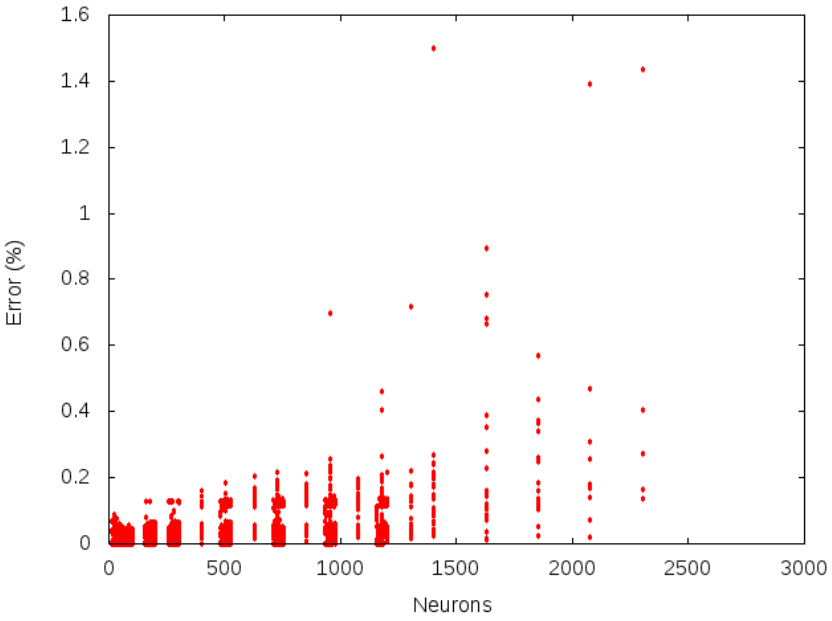


Figure 4.44: Generated error depending on the neurons number for different learning parameters

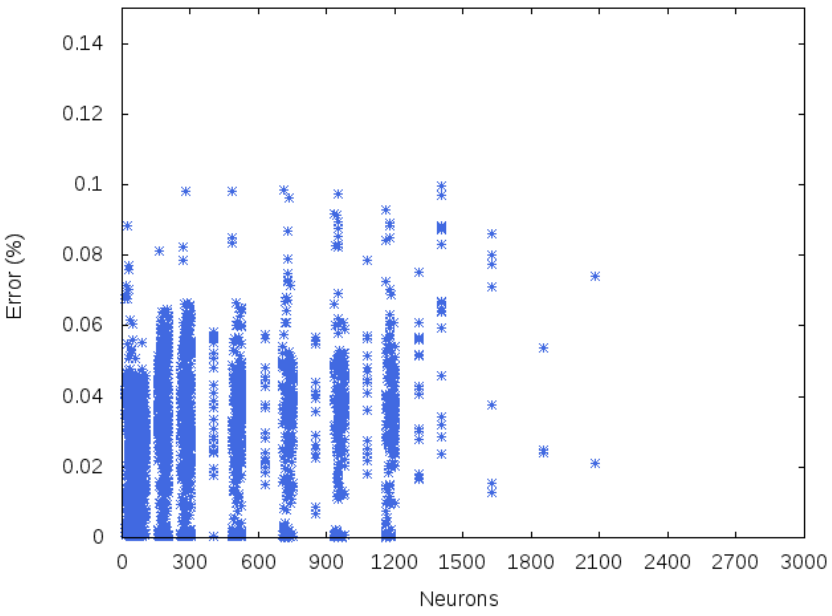


Figure 4.45: Neuron count where the calculated error is smaller than 0.1%

Increasing the number of neurons does not affect the error rates in a relevant way. Training time needed is higher when working with more neurons, due to the fact that more weights need to be recalculated in every iteration. A low number of neurons can produce better results that a network with many neurons using an optimum topology.

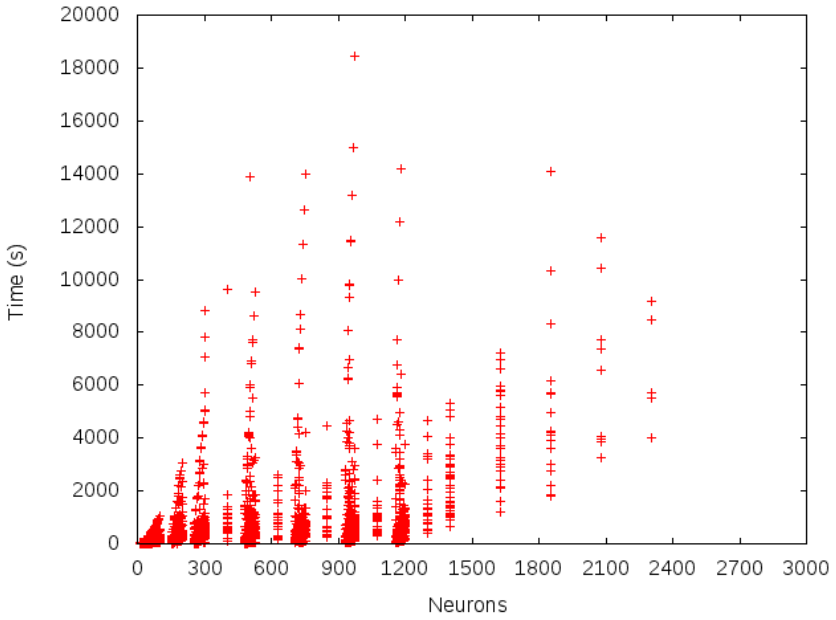


Figure 4.46: Time spent for training, according to the neuron count (adjusted to a linear function)

4.6.2 Two spirals problem

In order to aboard the training process and optimization from a different point of view, a different input pattern will be used. One common problem used in the neural network training is the two spirals problem. The idea behind is to learn a mapping that distinguishes between points in two intertwined spirals. Any MLP network able to resolve this

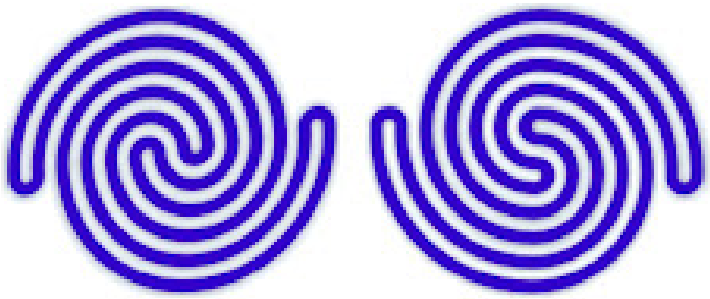


Figure 4.47: Two spirals problem

problem will have two inputs for the x and y coordinate, and two possible outputs, to distinguish between both two regions.

According to the literature [14], and as previously results have shown on the sinus case, how the pattern is chosen, and also how the input is coded, highly affects the training of the network. This is specially true when working with more complex problems. In our case, we decided to sample the input according to a classical data sheet [15].

Due to its complexity, this problem has been studied and used to test different neural networks. According to different studies ([16], [17]), best networks configurations are obtained when using a small number of hidden layers, with a small number of neurons in each layer, having the same neuron count in all of them.

Therefore, we have designed the tests for one up to four hidden layers, with a neuron count between 1 and 30 in each one. As learning parameters, LR of 0.01, Momentum of 0.5 and FSE of 0.1 have been used, due to the fact that not only literature, but also all the tests realized in our network have shown that these parameters produce the best trained networks.

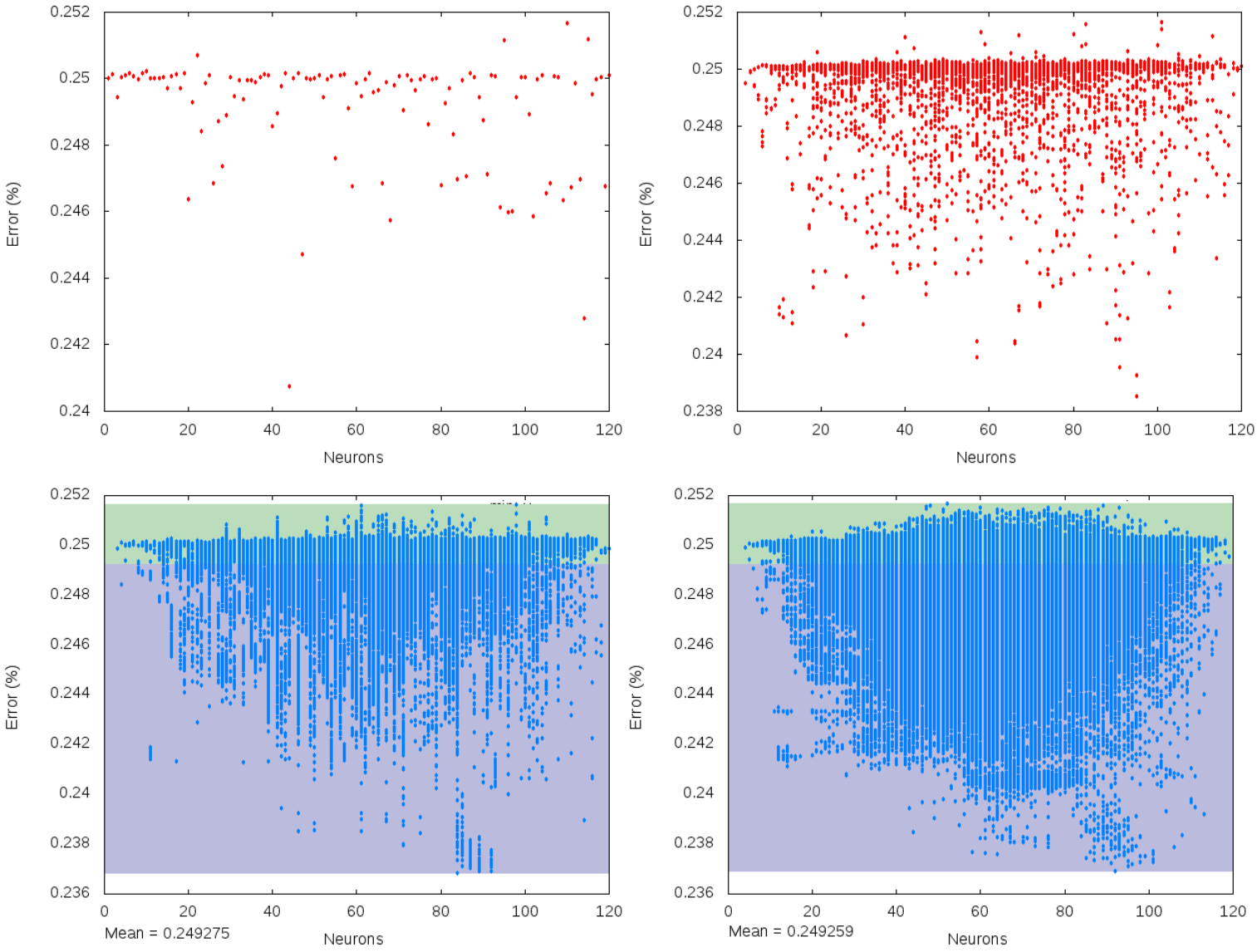


Figure 4.48: Neuron count vs Error(%) for one (upper left), two (upper right), three (bottom left) and four (bottom right) hidden layers

In this figure we can appreciate that increasing the number of neurons doesn't imply an improvement of the network. Otherwise, we can see that generally the best results are obtained when working with more neurons. This is because the training depends not only of the neuron count, but also of how the hidden layers are balanced.

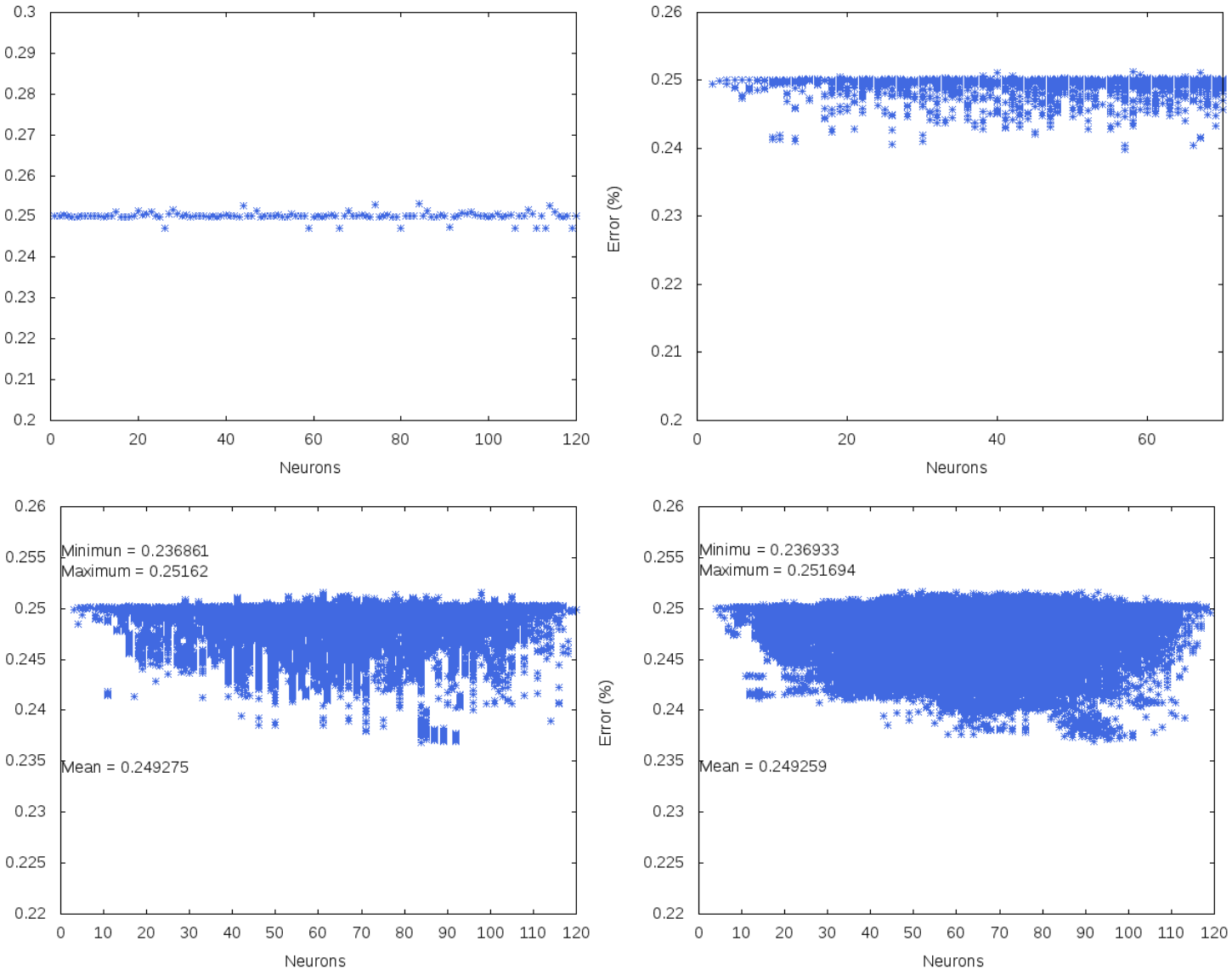


Figure 4.49: ,
scaledNeuron count for error rates under 0.25, for one (upper left), two (upper right), three (bottom left) and four (bottom right) hidden layers

The best 10 networks configurations for each layer are the followings:

- One layer:
  [26]; [18]; [25]; [5]; [16]; [12]; [1]; [17]; [6]; [13];
- Two layers:
  [26, 27]; [23, 21]; [26, 26]; [27, 19]; [28, 23]; [23, 14]; [23, 23]; [12, 29]; [25, 26]; [30, 20]

- Three layers:
  $[23, 27, 29]$; $[21, 29, 15]$; $[22, 23, 28]$; $[30, 29, 27]$; $[22, 19, 29]$; $[20, 14, 28]$; $[29, 28, 27]$; $[12, 10, 26]$; $[18, 17, 28]$; $[23, 23, 27]$
- Four layers:
  $[22, 22, 28, 29]$; $[13, 25, 29, 24]$; $[27, 21, 30, 30]$; $[16, 22, 30, 17]$; $[23, 27, 26, 28]$; $[24, 30, 29, 17]$; $[28, 25, 28, 29]$; $[29, 29, 30, 18]$; $[29, 30, 22, 30]$; $[26, 21, 29, 26]$

It doesn't seem to be a pattern, except that the difference between neuron count in each layer is never more than 6 when using three and four layers, and never more than 10 when working with two hidden layers. Also, neuron count is always around 20.

There's no a priori knowledge that can be used to decide which network configuration will be better. However, this information can be used to design test in order to obtain the optimum configuration. It should be taken into account that the higher the number of neurons is, the slower the training process gets.

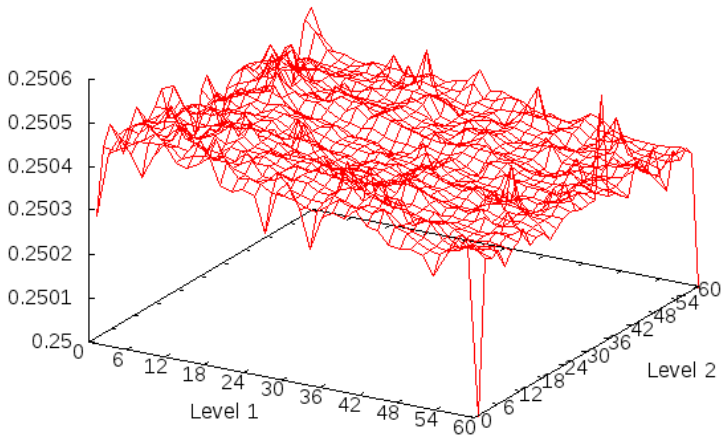


Figure 4.50: Level 1 and Level 2 count vs time, 20k iterations

Again, if we take a look to the previous graph, it doesn't seem to exist a big difference (except extreme cases) when working with different neuron counts and two hidden layers, obtaining values of ± 0.01 between the lowest and the highest error rates.

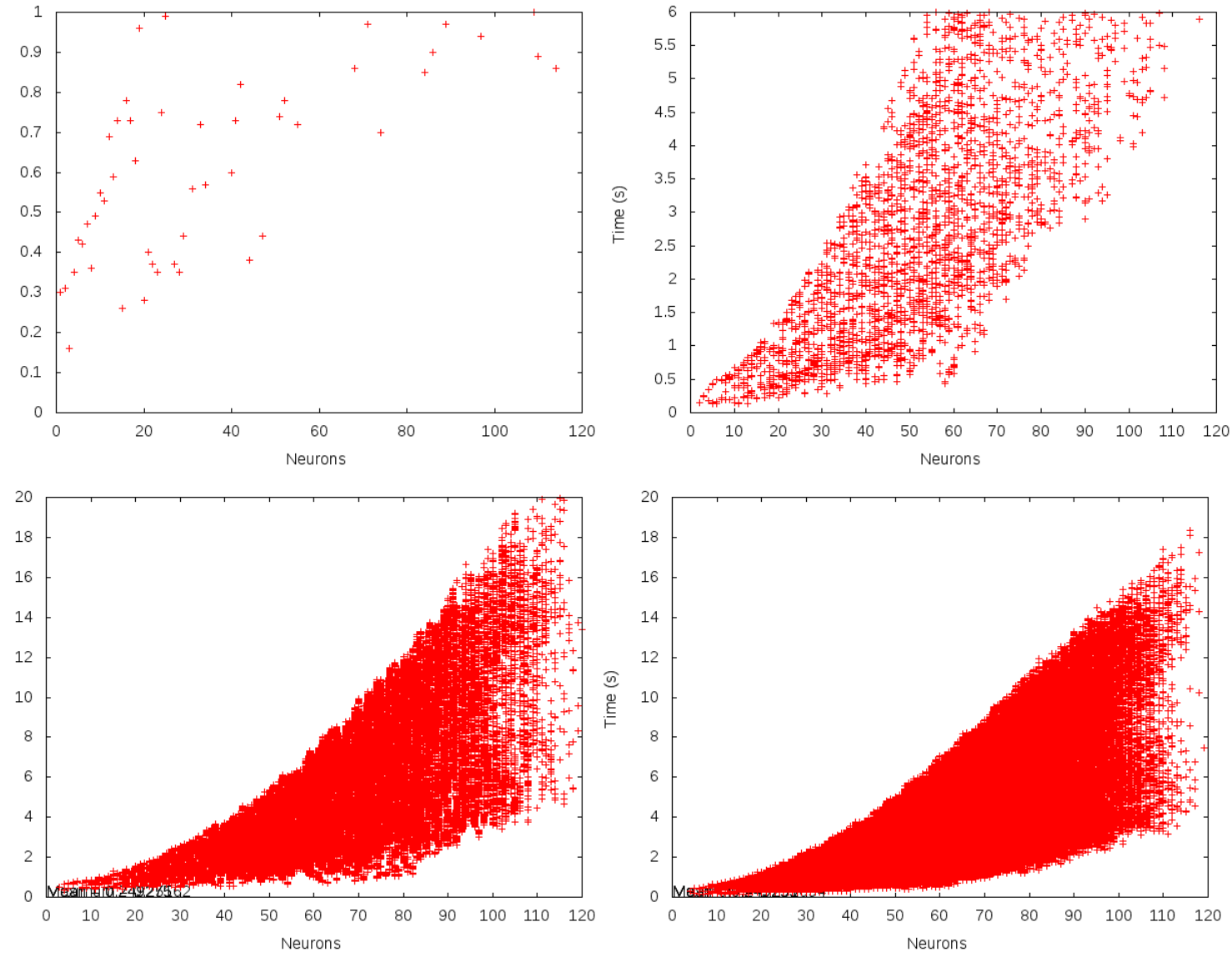


Figure 4.51: Neuron count vs Time for one (upper left), two (upper right), three (bottom left) and four (bottom right) hidden layers

The above graph shows how increasing the neuron count increase the time needed for the training process. This can be easily explained: a higher neuron count implies a higher number of weight lists, and therefore, more calculations needed in the back propagation process.

4.6.3 Non-linear classification problem

As the previous problem is a linear one, there is no relevant difference between the training with one neural network, and with multiple neurons, altought it exists, of course, a general improvement.

In order to test how the neural count highly influences in the trainning process, the simulator has been tested with a more complex problem. In this case, a linear mapping of the form $y = Ax$, where A corresponds to a $2x2$ matrix, with values between $[-1, 1]$, and x, y are 2-dimensional vectors.

Results are pictured in the following graph: As it's shown, there exists a big difference

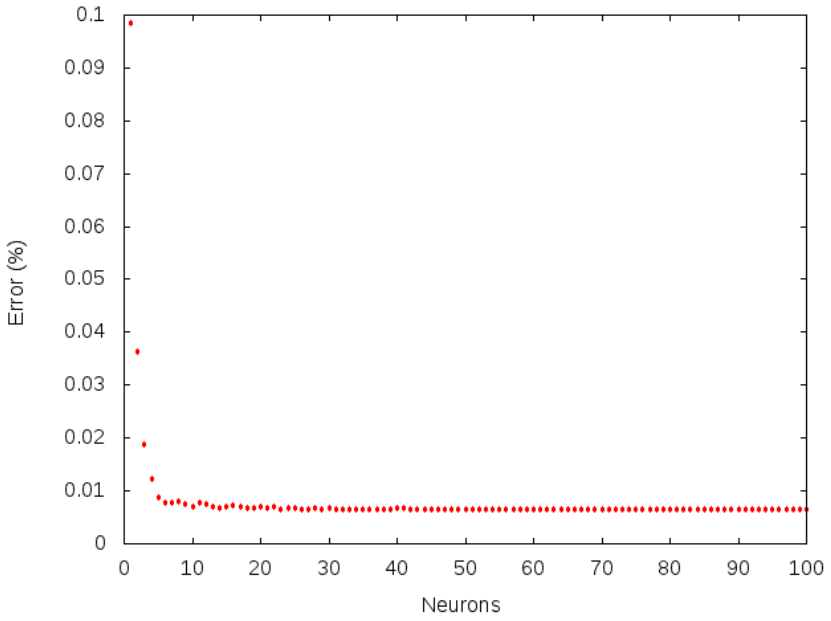


Figure 4.52: Trained network: Linear mapping

between the training with one neuron (0.58 MSE) or with ≈ 100 neurons (0.006 MSE).

4.7 Conclusions

The results obtained show how the neural network is influenced by different parameters. One of the most important ones, and which is usually ignored, is the input pattern. How to sample it, in order to train a proper network, should be studied for each case. This process will lead to an optimum training process.

The number of iterations also determine how the network is trained. When training an ANN using the back propagation algorithm, tests regarding configuration parameters (i.e LR, Momentum and FSE) should be realized. The learning rate highly influences on the training process. Although Momentum and Flat Spot Elimination also do, their influence is not so significant.

Increasing the number of neurons or hidden layers also produce better trained networks, but it increase the time needed to generate them. The optimal configuration of layers and neural counts depends specifically of the problem the network is solving.

It should be taken into account that the ANN will eventually over-fit. This means that no better results will be obtained after the over-fitting. Therefore, the network should be tested with input parameters that it has never seen before. The training process can be done automatic, in order to stop when the mean square error is higher than a threshold value.

ANN should be tested individually for each specific problem, in order to know their best topology and their best parameters configuration.

Chapter 5

Reparallelization using OpenMP

In computer science, parallel computing is a technique in which different calculations are carried out simultaneously. Therefore, large problems can be divided into smaller ones. This method is widely used in high performance computing.

There are different types of parallelism:

- **Bit-level parallelism:**
  It's based on increasing processor word size. These reduces the number of instructions needed to perform an operation where results are greater than the length of the word.
- **Instruction-level parallelism:**
  It consists in overlapping instructions, which can be done by hardware or by software (dynamically). The instructions which are going to be overlapped have to be independent, meaning that they shouldn't depend on the result of any other operation.
- **Data parallelism:**
  This is achieved when each processor performs the same task on different pieces of distributed data. It's based on the distributed (parallelized) nature of the data.
- **Task parallelism:**
  It's based on distributing execution processes (threads) across different parallel computing nodes. It's achieved when each processor executes a different thread on the same or different data. Communication is used to pass information from one thread to another.

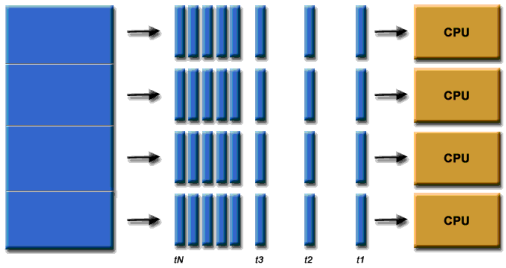


Figure 5.1: Parallel computing example, Lawrence Livermore National Security [1]

These new type of computation make programs and source code harder to write and debug, because new type of errors are introduced. One of the most common one is called race condition. Two processes (threads) are in race condition if the final result depends on the order in which the processes (threads) are executed. It usually happens when two or more processes access at the same time into a shared resource(i.e a variable) changing its value, and producing unexpected results.

5.1 Parallel programming models

There are many different tools, techniques and programming languages available in order to achieve this parallelization. Some of the most common ones are:

- **Programming languages and extensions:**
  Many different languages provide definitions in order to implement concurrent programming. ADA, for example, provides the *task* object, *processes* are available in C, and *threads* in Java, among others.
  However, as it has already been mentioned, concurrent programming can be a very complex process, and some extensions might be needed in order to be able to implement our source code easier:

  - OpenAcc:
    Available for standard C, C++, and Fortran. *The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.* [8]
  - OpenMP:
    Also available for C, C++, and Fortran. It is a *portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for*

[1] www.llnsllc.com

developing parallel applications for platforms ranging from the desktop to the supercomputer. [9]
This library is explained in detail in section 5.2.

- Ateji PX:
It's an extension of Java that can express most idioms of parallelism. It requires a JVM 1.6 or higher. [10]

- **Message Passing Interface:**
Message Passing Interface (MPI) is the standard which defines the syntax and semantics of the message passing to be used when needed in concurrent programs, which are able to run in more than one processor.
Basically, it's a communication protocol. The main advantage is that these libraries are portable and fast, because they have been optimized.

- **GPGPU programming:**
This method tries to use a Graphical Processing Unit (GPU) to perform computations. Graphics processing has a parallel nature. The use of multiple graphics cards in one computer, or large numbers of graphics chips, parallelizes even more this process.

In this section the focus will be set on OpenMP, because of the flexibility which it provides, and because the Neural Network source code is implemented in C++.

5.2 OpenMP

OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for different platforms. It is formed by three complementary components: a set of compiler directives, a runtime library, and environmental parameters.

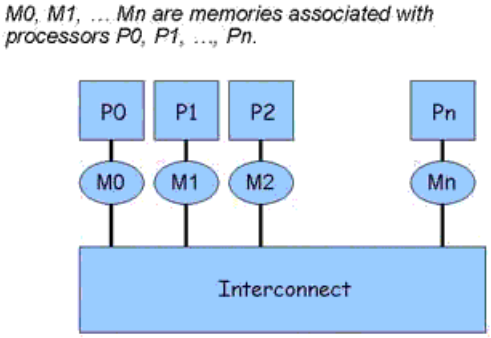


Figure 5.2: (a) Distributed memory

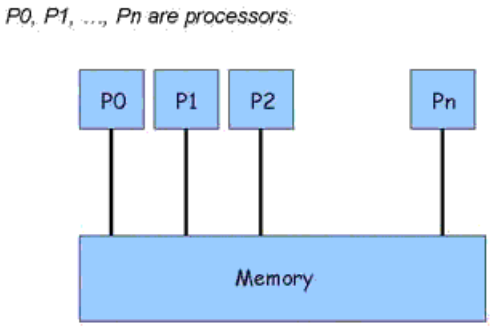


Figure 5.3: (a) Shared memory

Environmental parameters are used to define runtime system parallel parameters (i.e. number of threads). The compiler directives are used by the programmers in order to tell the compiler how the parallelism should be done. There are different types of directives and clauses availables. Some of them are:

- `parallel for`:
  Indicates that the following loop is executed in parallel. The loop is distributed among the number of threads.

- `private`:
  Prevent to share the defined variables among the different threads. Private variables must be initialized within the loop. By default, all variables (except the index) are shared.

- `shared`:
  Clause to share the defined variables among the different threads. Usually, it's used at the same time with the clause `default private`, which defines the variables private by default.

- `last private`:
  Last private retains the value of a private variable for use after the loop. Similarly, first private is used to initialize a variable.

- `reduction`:
  It's used when a specified reduction operation is performed on the individual values of the variable from each thread.

Also, synchronization clauses (i.e `critical`, `nowait`) and scheduling clauses (i.e. `static`, `dynamic`) are available, and can be found in the documentation [9]

Some small examples using OpenMP will be the following ones[1]:

Simple for loop:

```
#pragma omp parallel for
   for(i=1; i<=n; i++)
      a[:i] = b[i] + c[i]
```

Shared and private variables:

```
 #pragma omp parallel for default(private) shared(n,a,b,c)
{
   for(i=1; i<=n; i++){
      temp = 2.0*a[i];
      a[i] = temp;
      b[i] = c[i]/temp;
   }
}
```

`Lastprivate` usage:

```
 #pragma omp parallel for lastprivate(x)
{
   for(i=1; i<=n; i++){
      x = sin( pi * dx * (float)i );
      a[i] = exp(x);
   }
}
lastx = x;
```

[1]Extracted from the National Center for Supercomputer Applications, University of Illinois. `http://www.ncsa.illinois.edu/`

Reduction example:

```
 #pragma omp parallel for reduction(+:sum)
{
   for(i=1; i<=n; i++){
      sum = sum + a[i];
   }
}
```

Although its advantages as programming model, OpenMP also has some disadvantages:

- Codes parallelized with OpenMP can only be run in multiprocessor mode on shared-memory environments.
- Low parallel efficiency: OpenMP codes tend to rely more on parallelizable loops, which could leave a relatively high percentage of code in serial processing mode.
- A compiler that supports OpenMP is needed.

As this section presents, OpenMP appears like a suitable alternative to implement and realize the reparallelization of the neural network source code studied in this thesis.

5.3 Profiling

First of all, the sections of the source code that can be improved have to be detected. Then, the optimization can be implemented and evaluated.

Theoretically, two aspects of the code are the most time consuming ones. Therefore, the focus will be on them in order to optimize the code. They are:

- Forward pass:
  The propagation of the signal from the input to the output implies that the activation function has to be calculated for every propagation. Also, the error made has to be measured, which implies that this is a time consuming part of the execution.

- Backward pass:
  The error signal is propagated from the output layer to the input layer. Local gradients have to be calculated recursively for each neuron. Weights of the output units are altered, error of the hidden nodes calculated, and their weights are modified using these values. This is the most time consuming part of the whole learning process.

In order to check this, a profiling tool (gprof [22]) has been used to profile the code, obtaining the following results:

Name	% Time	Self seconds	Cumulative seconds
boost:numeric::ublas:: indexing_matrix_assign	25.53	0.12	0.12
MultiLayerPerceptron:: getOutput	25.53	0.12	0.24
boost::numeric::ublas:: indexing_vector_assign	14.89	0.07	0.31
boost::numeric::ublas:: indexing_matrix_assign	12.77	0.06	0.37
neuralNetworks:: TangentSigmoidFunction	8.51	0.04	0.41
BatchTrainingSystem:: train	6.38	0.03	0.44
boost::numeric::ublas:: indexing_vector_assign	4.26	0.02	0.46
boost::numeric::ublas:: same_impl_ex	2.13	0.01	0.47

Tests have been done for a sinus pattern of medium size. Results show that the more time consuming part regarding the neural network are located in the backward pass (train), the forward pass (getOutput), and in the activation function (TangentSigmoid-Function)

5.4 Implementation

The idea behind is to use the OpenMP library to improve the performance of the neural network.

The main aspects which determine the behaviour of the parallelized program are the number of threads, and how these threads are scheduled. Therefore, when evaluating the new implementation, different number of threads and different schedule types have to be tested.

5.4.1 Code optimizations

- **SinusTest:** The way in which the pattern is distributed (input and output) can be optimized. The code, without OpenMP optimizations is the following:

```
1  for(PatternSet::const_iterator p = allPattern.begin(); p !=
       allPattern.end(); ++p, ++index)
2  {
3    switch(index % 10)
4    {
5      case 0:
6      case 1:
7      case 2:
8      case 3:
9      case 4:
10     case 5:
11     case 6:
12     case 7:
13       result[0].addPattern(*p);
14       break;
15     case 8:
16       result[1].addPattern(*p);
17       break;
18     case 9:
19       default:
20       result[2].addPattern(*p);
21       break;
22   }
23 }
```

The changes needed to use OpenMP will be explained in detail:

- For condition initialization: p has to be declared before the loop.
- For condition checking: the not equal comparison can not be used with OpenMP, and has to be changed.
- For condition parameter update: only one parameter can be modified in the formed clause, therefore, *index* has to be changed inside the for loop.
- omp clauses: *result* is a shared variable, p and *index* are private.
- scheduling: the scheduling will be defined during the runtime. This is useful for checking different test cases.
- critical sections: modifying *result* is a critical part of the code, and has to be taken into account.

All the following optimizations are done analogue to this one. The optimized code is the following:

```
1  PatternSet::const_iterator p_end = allPattern.end();
2  PatternSet::const_iterator p_start = allPattern.begin();
3  PatternSet::const_iterator p;
4
5  #pragma omp parallel for firstprivate(index) private(p)
       shared(result) schedule(dynamic,1)
6  for(p=allPattern.begin();p<allPattern.end();++p)
7  {
8          switch(++index % 10)
9          {
10         case 0:
11         case 1:
12         case 2:
13         case 3:
14         case 4:
15         case 5:
16         case 6:
17         case 7:
18         #pragma omp critical
19         {
20                 result[0].addPattern(*p);
21         }
22         break;
23         case 8:
24         #pragma omp critical
25         {
26                 result[1].addPattern(*p);
27         }
28         break;
29         case 9:
30         default:
31         #pragma omp critical
32         {
33                 result[2].addPattern(*p);
34         }
35         break;
36
37         }
38 }
```

- **MultiLayerPerceptron:** The process of loading a MLP network from a file, and its initialization can be optimized when doing these tasks in parallel. Some other constructor aspects can also be optimized.
  However, the most important aspect to optimize in this class is the forward pass, contained in the *simulate()* function:

```
void MultiLayerPerceptron::simulate(Iter from, Iter to) {
        #pragma omp parallel for schedule(runtime)
        for (Iter pattern = from; pattern < to; ++pattern) {
                pattern->getOutput() = getOutput(pattern->
                   getInput());
        }
}
```

- **BatchTrainingSystem:** The training system itself implements the backward pass. Not only some aspects of the initialization can be done in parallel, but also the process of recalculating different values needed for the forward pass.

  The training process:

```
void BatchTrainingSystem::train(..)
{
        ...
        MultiLayerPerceptron::BiasList::iterator i =
           sumBiasChanges.begin();
        #pragma omp parallel
        { // parallel region begins

        #pragma omp for schedule(runtime)
        for (int j=0;j<sumBiasChanges.size();++j)
        {
                (*(i+j))->clear();
        }
        ...
        MultiLayerPerceptron::WeightList::iterator k =
           sumWeightChanges.begin();
        #pragma omp for schedule(runtime)
        for(int j=0;j<sumWeightChanges.size();++j)
        {
                (*(k+j))->clear();
        }
        ...
        #pragma omp for schedule(runtime)
        for(int i=0;i<sumWeightChanges.size();++i)
        {
```

```
24                  **(owc+i) *= m_learningParameter.getMomentum
                       ();
25                  **(owc+i) += (m_learningParameter.
                       getLearningRate() * **(w+i)) / count;
26          }
27          ...
28          #pragma omp for schedule(runtime)
29          for(int i=0;i<sumBiasChanges.size();++i)
30          {
31                  **(obc+i) *= m_learningParameter.getMomentum
                       ();
32                  **(obc+i) += (m_learningParameter.
                       getLearningRate() * **(b+i)) / count;
33          }
34          ...
35          } // parallel region ends
36  }
```

The recalculation process: (backward pass)

```
1   void BatchTrainingSystem::calcChanges(...) {
2           ...
3           #pragma omp parallel
4           { // parallel region begins
5
6           #pragma omp for schedule(runtime)
7           for(int i=0;i < l_s;++i)
8           {
9                   Matrix &weights = **(weightIterator+i);
10                  Vector &previousDelta = **(
                       previousDeltaIterator +i);
11                  Vector &biasDelta = **(biasDeltaIterator+i);
12                  Matrix &weightDelta = **(weightDeltaIterator+
                       i);
13                  Vector &toVoltage = **(toVoltageIterator+i);
14                  Vector &fromVoltage = **(fromVoltageIterator+
                       i);
15
16                  scalar_vector<double> flatSpotVector(
                       biasDelta.size(), m_learningParameter.
                       getFlatSpotElimination());
17
18                  Vector error(weights.size1());
19                  noalias(error) = prod(previousDelta, trans(
                       weights));
```

```
20                  std::transform(toVoltage.begin(), toVoltage.
                       end(), biasDelta.begin(), &
                       TangensSigmoidFunction::
                       getVoltageDerivation);
21
22                  noalias(biasDelta) = element_prod(biasDelta +
                        flatSpotVector, error);
23                  noalias(weightDelta) = outer_prod(fromVoltage
                       , biasDelta);
24
25          }
26          ...
27          #pragma omp for schedule(runtime)
28          for(int i=0;i < sumWeightChanges.size();i++)
29          {
30                  **(w+i) += **(wd+i);
31          }
32          ...
33          #pragma omp for schedule(runtime)
34          for(int i=0;i < sumBiasChanges.size();++i)
35          {
36                  **(b +i) += **(bd+i);
37          }
38          ...
39          } // parallel region ends
40  }
```

- **OnlineTrainingSystem:** The optimizations that can be done are analogue to the ones done in the batch training system:

```
1   void OnlineTrainingSystem::train(...) {
2           ...
3           #pragma omp parallel
4           { //parallel region begins
5
6           #pragma omp for schedule(runtime)
7           for(int i=0;i < dis;++i)
8           {
9                   Matrix &weights = **(weightIterator+i);
10                  Vector &previousDelta = **(
                       previousDeltaIterator+i);
11                  Vector &biasDelta = **(biasDeltaIterator+i);
12                  Matrix &weightDelta = **(weightDeltaIterator+
                       i);
```

```
13                  Vector &toVoltage = **(toVoltageIterator+i);
14                  Vector &fromVoltage = **(fromVoltageIterator+
                       i);
15
16                  scalar_vector<double> flatSpotVector(
                       biasDelta.size(), m_learningParameter.
                       getFlatSpotElimination());
17                  Vector error(weights.size1());
18                  noalias(error) = prod(previousDelta, trans(
                       weights));
19                  std::transform(toVoltage.begin(), toVoltage.
                       end(), biasDelta.begin(), &
                       TangensSigmoidFunction::
                       getVoltageDerivation);
20                  noalias(biasDelta) = element_prod(biasDelta +
                        flatSpotVector, error);
21                  noalias(weightDelta) = outer_prod(fromVoltage
                       , biasDelta);
22
23          }
24          ...
25          #pragma omp for schedule(runtime)
26          for(int i=0; i < weightDeltas.size();++i)
27          {
28                  **(owc+i) *= m_learningParameter.getMomentum
                       ();
29                  **(owc+i) += m_learningParameter.
                       getLearningRate() * **(w+i);
30          }
31
32          BiasList::const_iterator obc = oldBiasDeltas.begin();
33          BiasList::iterator b = biasDeltas.begin();
34
35          #pragma omp for schedule(runtime)
36          for (int i = 0; i < biasDeltas.size(); ++i) {
37                  **(obc + i) *= m_learningParameter.
                       getMomentum();
38                  **(obc + i) += m_learningParameter.
                       getLearningRate()
39                                  * **(b + i);
40          }
41          ...
42          } // parallel region ends
43  }
```

5.4.2 Evaluation

All the simulations done behave in a similar way: when working with a low number of neurons, the time needed for the learning process is much lower when using only one thread. As soon as the neuron count is increased, the time needed for the simulation is lower when using multiple cpus. This is because creating threads is a complex and a time consuming process. If the number of neurons is low, the simulator would not take the advantage of using more than one thread, but it will need more time to create the threads and then simulate the learning process.

For the evaluation, the simulator has been tested from 1 to 8 threads. The time needed for each simulation has been measured. Also, different scheduling algorithms have been tested.

The results obtained for one hidden layer are the followings:

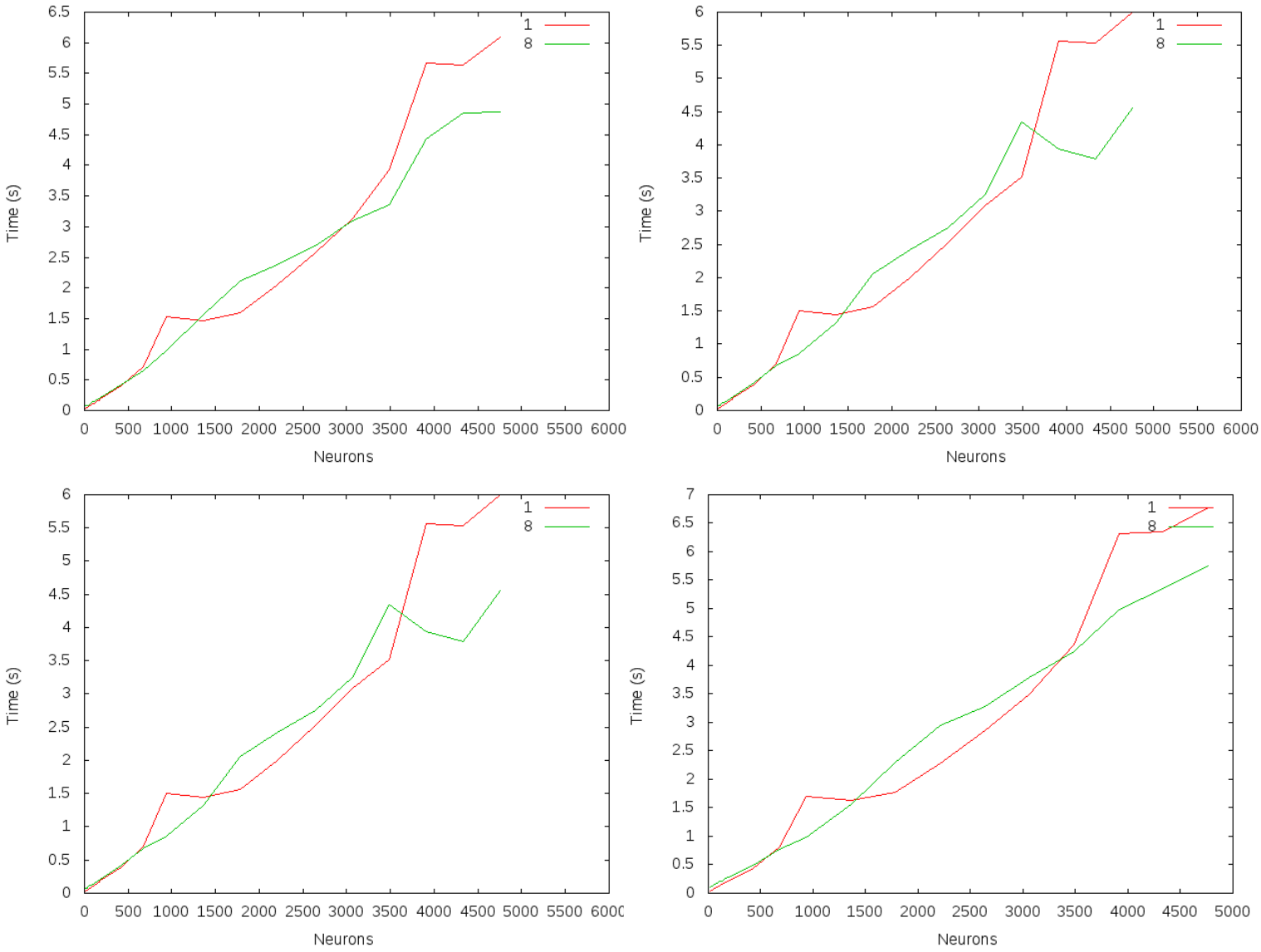


Figure 5.4: One layer: training results using different schedules: auto(upper left), guided(upper right) static (lower left) and dynamic (lower right)

The results obtained for two hidden layers are the followings:

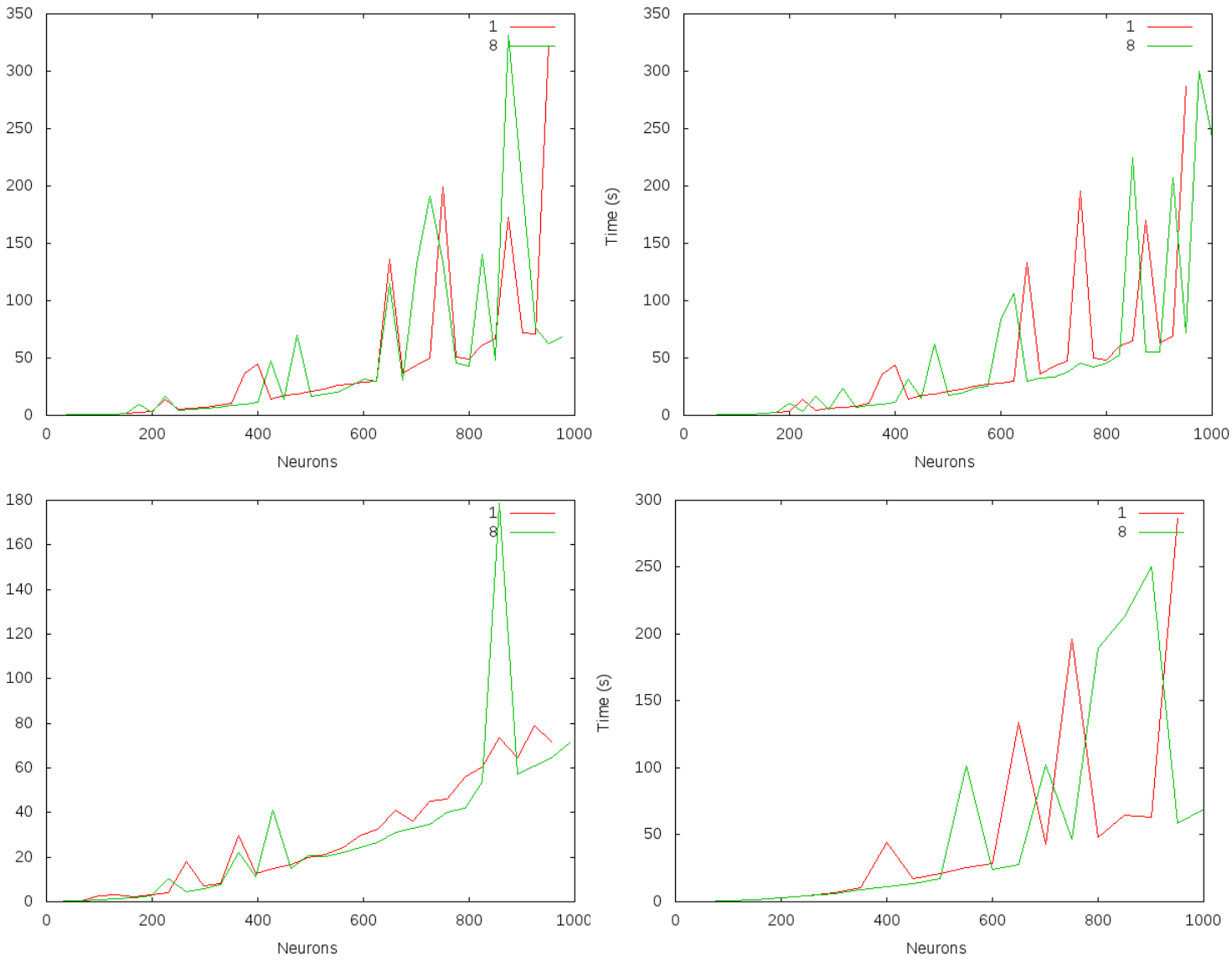


Figure 5.5: Multiple layers: training results using different schedules: auto(upper left), guided(upper right) static (lower left) and dynamic (lower right)

As we can see, as the number of neurons increase, the graph converges, giving slightly faster results when working with more than one thread. When working with a very grained test set (i.e low number of neurons) the results are better when the training is done only with one thread.
This is because creating the different threads, and synchronizing them, is a complex process. Although static scheduling (iterations are divided into predefined chunks) and dynamic scheduling (each thread execute a chunk of iterations, and then request more) seems to give good results for one hidden layer and multilayer neural networks respectively, guided scheduling (the chunk size starts large and shrink) is recommended, and also provides optimal and accurate results. The auto scheduling should not be taken into account, because it depends on the compiler.

However, results show that OpenMP is not working as expected. This is due to the fact that most of the time and tasks ($\approx 60\%$) are spent in arithmetic calculations, done by the boos::numeric::ublas library. This library is not parallelized using OpenMP, and therefore, it can't take advantage of its usage.

Amdahl's law [12] defines the maximum expected improvement to an overall system when only part of the system is improved. This can be applied to parallel computing, also taking into account that the improvement not only depends on the parallelized part, but also on the nature of the algorithm. Due to this, and if C is the number of available cores and P is the ratio of parallelized portion of an algorithm then according to Amdahl's law the speed-up would be

$$Speedup = \frac{1}{(1-P) + \frac{P}{C}} \tag{5.1}$$

According to Amdahl's law (equation 5.1) a speed-up of 1.33x using 8 cores implies that only around 25% of the code has been parallelized. This low parallelization rate is explained because of the numeric::ublas library.

Chapter 6

Conclusion and future work

6.1 Conclusion

Optimize a neural network can have many different approaches.

On one side, a proper configuration of the learning process can be done: the topology of the network can be optimized. To do so, a different number of hidden layers could be set. Also, the neuron count in each layer can be changed. Different topologies produce different results, depending on the problem we're working with. Results presented in this work show that a higher number of hidden layers an neurons usually produce more accurate neural networks, due to its flexibility, until the over-fitting point is reached. However, more time is needed for the training process.

Also the parameters in the back propagation algorithm can be configured. While the Momentum and the Flat Spot Elimination rate slightly affects the performance of the network, variations on the learning rate produce very different trained networks. For the tested problems (sinus, mail classification and two spirals) low learning rates (i.e. around 0.01) produce more accurate networks. This could change depending on which problem the network is working with.

Before working with an ANN, different tests should be realized, in order to know which configuration is better for each problem, not only in terms of topology, but also in terms of the learning process.

On the other side, optimizations regarding the algorithm itself can be done. A possible approach is to make use of parallelization paradigms (i.e OpenMP) to execute it in CPUs. This work shows how better performance can be achieved using OpenMP. However, the

speed-up that can be achieved is limited not only by the number of cores, but for the parallelizing process itself.

Depending on the specific problem, the focus will be different. If we're interessted in a very precise network, like a math function might require, the network should be trained carefully, doing different tests to check which parameters produce more accurate results, and doing a higher number of iterations. This means that the proccess of training the ANN will be slow. The learning rate and the sample data should be studied carefully.

When we want faster results, but precission is not the main topic, the effort should be put in the parallelization problem, in order to produce the output as soon as possible. CUDA libraries and OpenMP might be specially helpfull for this task, taking in mind that the most time consuming parts of the algorithm are the forward and the backward pass.

6.2 Future work

Some pending topics that could be researched are the following ones:

- Different learning algorithms:
  Back propagation algorithm has shown that, although it works and produces optimal networks, the training process might be a bit slow, specially when working with more complex problems and input patterns. A study and evaluation about different training methods and could be realized, such as the Conjugate Gradient or Resilient Back-propagation.

- Different input patterns:
  Complex problems might lead to different network behaviours. An study about how sampling should be done, or how the network configuration is affected by different patterns might give useful information about artificial neural networks and its learning process.

- OpenMP parallelization:
  Parallelizing the boost::numeric::ublas library might produce good results when obtaining faster trained networks. Although there are some functions already parallelized, this might be a complex process. Also, using MKL [23] or any other library that can be parallelized instead of ublas will result in faster ANN training processes..

- Implementation in hardware accelerators:
  Although there's already a version of the ANN simulator using CUDA, its behaviour might not be optimum when working with small patterns. Implementing the simulator for GPU with a different technology, such as HMPP [21], might produce different results depending on the test case scenario.

Bibliography

[1] Danilo P. Mandic, Jonathon A. Chambers, *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability. Wiley, Jun. 2002*

[2] Geoffrey Hinton, Terrence J. Sejnowski, *Unsupervised Learning: Foundations of Neural Computation. MIT Press, 1999*

[3] Richard S. Sutton, Andrew G. Barto, *Reinforcement Learning: An Introduction. MIT Press, 1998*

[4] GNU, *GNUplot documentation* `http://www.gnuplot.info/docs_4.6/gnuplot.pdf/`

[5] Anne Auger, Nikolaus Hansen, *CMA-ES: Evolution Strategies and Covariance Matrix Adaptation. GECCO, July 2011* `http://www.lri.fr/ hansen/gecco2011-CMA-ES-tutorial.pdf`

[6] Zaher Ameed, *Optimization of Neural Network Simulator on GPGPU. HLRS, August 2011*

[7] SJ Thorpe, *Spike arrival times: A highly efficient coding scheme for neural networks. Parallel processing in neural systems, 1990* `http://pop.cerco.ups-tlse.fr/fr_vers/documents/thorpe_sj_90_91.pdf`

[8] OpenACC, *The OpenACCTM Application Programming Interface. Version 1.0, November 2011* `http://www.openacc-standard.org/sites/default/files/OpenACC.1.0_0.pdf`

[9] GChandra et al., *Parallel Programming in OpenMP. Morgan Kaufmann, 1999*

[10] Patrick Viry, *The Ateji PX 1.2 Manual. Revision 22, September 2011* `http://www.ateji.com/px/1.0/manual/`

[11] Jason Sanders, Edward Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, 2011*

[12] Mark D. Hill, Michael R. Marty, *Amdahl's law in the multicore era. IEEE Computer Society, 2008*

[13] OpenHMPP Consortium, *OpenHMPP: Concepts and Directives. Version 1.0, June 2011* `http://www.openhmpp.org`

[14] Hock-Chuan Chua, *Solving two-spiral problems through input data representation. IEEE International Conference, December 1995* `http://www.openhmpp.org`

[15] Ben Margolis, *Two spirals problem solved, 2009* `http://www.benmargolis.com/compsci/ai/two_spirals_problem.htm`

[16] Claudio Moraga, *Design of Neural Networks. European Center for Soft Computing, 2009*

[17] Chin-Chi Teng et al., *An automated design system for finding the minimal configuration of a feed forward neural network. University of Illinois, 1994*

[18] Subhash C. Kalk, *On generalization by neural networs. Louisiana University, 1997*

[19] Xavier Sierra-Canto, Francisco Madera-Ramrez and Victor Uc-Cetina, *Parallel Training of a Back-Propagation Neural Network Using CUDA, Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on, pp.307–312, 12–14 Dec. 2010*

[20] S. Narain, A. Jain, *Evaluation of the sensitivity of learning rate on the training of neural network hydrologic models. NASA Astrophysics, 2005*

[21] Romain Dolbeau et al., *HMPP: A Hybrid Multi-core Parallel Programming Environment. CAPS Enterprise, 2007*

[22] Jeffrey Osier, *Gprof Manual. Free Software Foundation, 1993.* `http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`

[23] Intel Corporation, *Using Intel Math Kernel Library for Matrix Multiplication. 2012*

Acknowledgment

I am heartily thankful to my supervisors Dr. Colin W. Glass, Mhd. Amer Wafai from HLRS and Oliver Zweigle and Paul Levi from Universität Stuttgart whose encouragement, guidance and support from the initial to the final stage of this work. I am grateful to my coworkers at HLRS for advising me useful tips.

Lastly, I offer my regards and gratitude to Dr. Ing Onay Urfalioglu from HMI-Tec, who provide me good advices during the Master thesis.

Special thanks to my family, friends and girlfriend for their moral support.

Daniel del Hoyo

Erklärung / Declaration

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Daniel del Hoyo)