UNIVERSITATEA „ALEXANDRU IOAN CUZA”

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Coordonator ştiinţific

PhD Henri Luchian

Absolvent

Jean Arnaud Pierre Micheli

IAŞI
2018

UNIVERSITATEA ALEXANDRU IOAN CUZA IAŞI
FACULTATEA DE INFORMATICĂ

Optimization of neural networks structure by genetic algorithm

Jean Arnaud Pierre Micheli

Sesiunea: Iulie, 2018

Coordonator ştiinţific

PhD Henri Luchian

DECLARAȚIE PRIVIND ORIGINALITATE ŞI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licență cu titlul “Optimization of neural networks structure by genetic algorithm” este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituție de învățământ superior din țară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi dețin referința precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alți autori deține referința precisă;
- codul sursă, imagini etc. preluate din proiecte open-source sau alte surse sunt utilizate cu respectarea drepturilor de autor şi dețin referințe precise;
- rezumarea ideilor altor autori precizează referința precisă la textul original.

Iaşi, 25.06.2018

Absolvent Micheli Jean Arnaud Pierre

_____

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul "Titlul complet al lucrării", codul sursă al programelor şi celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea Alexandru Ioan Cuza Iaşi să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iaşi, 25.06.2018

Absolvent Jean Arnaud Pierre Micheli

_____

Introduction

In last past years, the interest of artificial intelligence rose exponentially and is now present in every imaginable field. Since we discovered that machine learning methods could reproduce and even surpass human analyzing tasks as image or speech recognition, medical diagnosis, etc., it became necessary for public infrastructures to follow the standard and for businesses to implement such an intelligence to survive in the competitive industries.

Artificial neural networks are capable to find the correct analyze solution without knowing exactly the rapport between the sources and the result. They implement the feeling of a solution. However to be well performing in a particular domain of classification, a neural network needs to be well parametrized. No generalized method exists to always find the optimal parameters of a neural network. Genetic algorithms have the particularity to find good parameters in wide searching spaces. The interest of this work is to see how those two principles can be combined to achieve better results. The parameters of the genetic algorithm are easier to predefine than the neural network ones. This allows the possibility to automatise the creation of the neural network for every classification or regression problem.

We will first understand what makes neural networks the perfect candidates for classification tasks. Then why genetic algorithms are finding best parameters in wide search spaces. Finally, how both methods can be adapted to complete themselves.

Contributions

The main contribution of this work has been the elaboration process of a genetic algorithm which could provide good initial neural network parameters. This includes an original encoding of neural networks presented in the third chapter. The implementation of such an algorithm has been done using python programming language, in order to retrieve some results. The implementation of a neural network was also necessary, but the contribution with the neural network lays only in the adaptation of existing open source code.

We can report three important run of the program of 1, 8 and 18 hours. Only the last one can partially atest of the genetic algorithm success. Analysis of these results are made, helping to understand the effects of the designed neural network encoding.

I. Artificial Neural networks

1. Overview

An artificial neural network (ANN) is a function based on the biologic neural networks, permitting to give some output values given some inputs. The network is formed by neurons and synapses connections (weights). Even if the variety of neural network types is important, we can at least identify three types of neurons.

First, are the input neurons, which represent the inputs of the function. These neurons only have exiting synapses. Then the calculs go through hidden neurons, allowing the complexity of the function (intermediary calculs). They can be organised on one or more layers. Finally, the outputs neurons are giving the results of the function. They only have entering synapses.

We generally use ANNs for classification or regression problems. For classification, each output is linked to a class or action decision. Given some inputs, the output having the highest value represents the chosen class. When an error of decision is encountered, the biologic neural network, decreases the amount of neurotransmitters in synapses, to hinder the same comportment, next time a similar situation is encountered. Analogically, the artificial one can adjust its parameters (modifying the importance of the weights to get the desired output). To adequately adjust the weights, three general learning methods can be used.

In supervised learning, we have a set of input data labeled with the correct expected output. The network is trained many times on this same set to finally get the expected output. Then it is ready to classify new data that hasn't been encountered yet. Our attention will be focused on supervised learning, but neural networks can be applied to other types of learning methods like unsupervised and reinforcement learning.

When unsupervised, the data is furnished to the program without label. The purpose is just to find common features between the data and create an arbitrary classification taking in consideration these similarities.

In reinforcement learning, the data is taken from an environment after a choice from the neural network. The network is trained based on rewards or

punishments (augment or lower the importance of the weights which conducted to this choice).

An interesting aspect of the ANN is the parallelism furnished by the weights which are all perfectionated in the same time.

ANNs may also approximate a function, referred as regression problems. Thus this possibility will not be developed in this work. Different architectures can be defined as neural networks, our focus will be set on feed forward multilayer perceptron.

2. Perceptron, the single neuron network

Let's explain first how a single artificial neuron (perceptron) is working. The Perceptron was invented by Frank Rosenblatt in 1957 for image recognition.

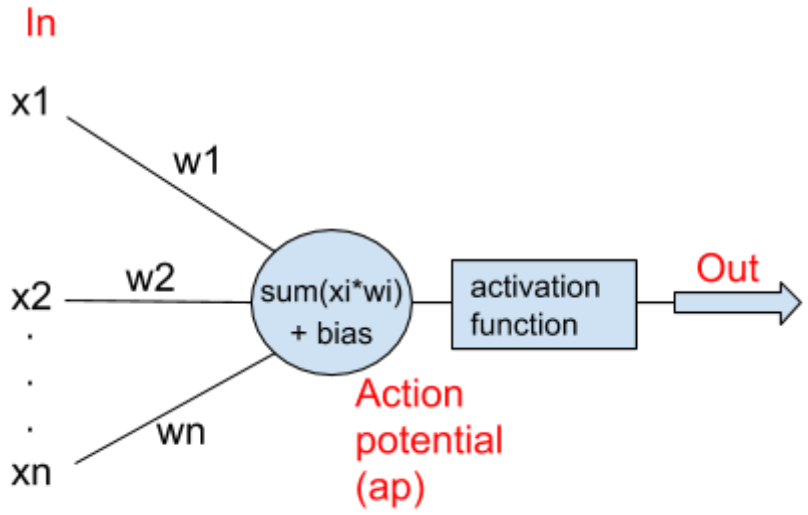

Figure 1 : the perceptron and its components

In the perceptron (figure 1), each inputs (xi) is multiplied by a weight (wi) and then summed with the other inputs. A bias specific to the neuron is also added. The result pass by an activation function which is going to determinate the final output. The activation function was originally a threshold (return 1 if the sum is higher than 0, else 0). The perceptron is able to give answers "yes" or "no". It is

a binary classifier. The perceptron is written mathematically as the following:

$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^{m} w_i x_i + b > 0 \\ 0 & \text{otherwise} \end{cases}$$, with m the number of inputs.

We can observe that the function represented by the perceptron is a linear combination.

The linear classifiers are working perfectly when the data can be separated by a hyper-plan, but when it's not, the function will never success to classify correctly all the data. This principle will be explained more in details in part 3 about neural networks. A well known example is the Xor function with two Boolean inputs. The perceptron output can be defined as the boolean expression:

(w1B1 + w2B2 + b) > 0

B1	B2	B1 XOR B2	Possible deduction from perceptron output calculation
0	0	0	1. b <= 0
0	1	1	2. w2 + b > 0
1	0	1	3. w1 + b > 0
1	1	0	4. w1 + w2 + b <= 0

Table 1 : perceptron parameters requirements for XOR approximation

As we can see in the table 1, from points 2 and 3, w1 > -b and w2 > -b. In 4 we have w1+w2 <= -b. Replacing with w1 = w2 = -b, we get:

$$\begin{cases} -2b \leq -b \\ 2b \geq b \end{cases}$$

Since b <=0, 2b should be lower than b. This is not the case, we can conclude that for any value of b, w1 and w2 the perceptron will never success to reproduce the Xor function.

If other possible answers (classes) are wished, other output nodes will be added. Thus, with the threshold activation function, if we have other outputs crossing the threshold value, we cannot know which one is more activated. For this reason other activation function are used. A simple one often seen until recently is the sigmoid σ(x). This function smoothly rescale the input between 0 and 1. Before explorating activation functions, we need to understand how the network learns.

3. Learning with backpropagation by stochastic gradient descent

We explained the way a perceptron computes an output. In this section, we'll answer the question: how supervised learning can modify the weights to predict the expected result of the inputs data?

At the beginning, some predefined weights and biases are affected to each connections in the network. The first outputs may be significantly different from the expected ones linked to the train data. This difference or error is expressed by a cost function.

Generally a cost function permits to estimate a difference between two values. Each cost function should be strictly positive and proportional to the difference (close to 0 if the values are similar and far from 0 if not). In classification, the cost function quantifies the error of prediction made by the model on the current labeled data.

One commonly seen in mathematics is the quadratic cost function, or mean square error:

$C = (out\text{-}target)^2/2$

Our goal is to minimize this cost function. A general method is to find the partial derivative of the function and modify the parameter In order to get a lower result. The cost function of the perceptron depends on many parameters (weights, input, bias, target). The key to know how to correctly adapts the weights is to use the cost function. If the error is high, we can deduce that the weights are far from their ideal value. We would desire to find the relationship between each weight and the cost function. If the cost function is differentiable, we can use partial derivatives. A gradient is a derivative of a function having more than one

parameter. We can express the gradient vector of the perceptron cost function as follow:

$$\nabla C = \left(\frac{\partial C}{\partial w1}, \frac{\partial C}{\partial w2}, \cdots \frac{\partial C}{\partial w_n}, \frac{\partial C}{\partial b} \right)$$

The cost function depends also on inputs and the target output but those are constant for a training unit, therefore they don't appear in the gradient. The partial derivative ∂C/∂w1 represents one dimension of the gradient vector pointing towards a local minimum of the cost function. In other words, this is the relationship we were searching. Let be *delta w* an adjustment of the weight w1. We can now pre-calculate the impact (noted *delta C)* it will have on the cost function:

$$\frac{\partial C}{\partial w1} * \Delta w1 = \Delta C$$

We want to be sure to decrease the cost function, requiring ΔC to be negative. We could take $\Delta w1 = -\frac{\partial C}{\partial w1}$, but we may cross the local minima. The partial derivative indicate only a direction, so we should make only a small step per training unit. To archive this, ∂C/∂w is multiplied by a small constant named learning rate. The final adaptation of the weight becomes:

$$\Delta w1 = -\alpha * \frac{\partial C}{\partial w1}.$$

To develop this form we need to know how ∂C/∂w could be calculated.

Before depending on the weight w, C depends on the result of the output neuron of the network. Using the chain rule we obtain:

$$\frac{\partial C}{\partial w1} = \frac{\partial C}{\partial out} * \frac{\partial out}{\partial w1}.$$

Analogically, the output of the neuron depends on its input. After a second application of the chain rule, the result is:

$$\frac{\partial C}{\partial w1} = \frac{\partial C}{\partial out} * \frac{\partial out}{\partial in} * \frac{\partial in}{\partial w1}.$$

$\frac{\partial C}{\partial out} * \frac{\partial out}{\partial in}$ *can be written* δ *and correspond to the error of the output neuron. This is also the value with which we adapt the bias (b = b +* δ)

Derivating the sum of weighted inputs with respect to a particular weight results in the corresponding input:

$\frac{\partial C}{\partial w1} = C'(out, target) * a'(ap) * x1$, with ap the sum of weighted inputs

This also can be written as $\frac{\partial C}{\partial w1} = \delta * x1$

Replacing C by the mean square error and a by the sigmoid function we get:

$$\frac{\partial C}{\partial w1} = (\sigma(ap) - target) * \sigma(ap) * (1 - \sigma(ap)) * x1$$

$$w1 = w1 + \alpha(\sigma(ap) - target) * \sigma(ap) * (1 - \sigma(ap)) * x1$$

The efficiency of a cost function may be influenced by the activation function. When using sigmoid and mean square error, the adaptation of the weight is not always proportional to the error. For example, in the case of a really high error with target = 0, sig(ap) almost 1 and x1 = 1, the adaptation of the weight becomes

$$w1 = w1 + \alpha * 1 * 1 * (\approx 0) * 1$$

In those conditions, even with high error, the weight will adapt significantly slow. For this reason, when using sigmoid activation function, we prefer the cross-entropy cost function:

$$C = target * ln(ap) + (1 - target) * ln(1 - ap)$$

$$= \frac{target}{\sigma(ap)} - \frac{(1 - target)}{(1 - \sigma(ap))} * \sigma'(ap) * x1$$

$$= \frac{target(1 - \sigma(ap)) - \sigma(ap)(1 - target))}{\sigma(ap)(1 - \sigma(ap))} * \sigma'(ap) * x1$$

$$= target(1 - \sigma(ap)) - \sigma(ap)(1 - target)) * x1,$$

since σ'(x) = σ(x)(1-σ(x))

$$= (target - \sigma(ap)) * x1$$

Finally in the weights adapting formula:

$$w1 = w1 + \alpha * (target - \sigma(ap)) * x1$$

We can observe that only the difference between the output and the target (σ(ap)-target) influences the weight adaptation.

This method is so called back-propagation using stochastic gradient. Naturally, as for minimizing any function, other possibilities exist to adapt the weights, but this one is the most popular and relatively fast with matrix operations.

The presented formulas explained how the perceptron learns. However, the real interest of neural networks begin with more hidden neurons. Stochastic gradient descent can be applied to a fully connected multi layer perceptron and calculation can be made under a vectorized forms. We will use the following notation:

w(L,Nin,Nout) refers to the value of the weight entering in the neuron Nin on layer L and leaving the neuron Nout on layer L-1. δ(L,N) represents the back-propagated error of the neuron N in the layer L.

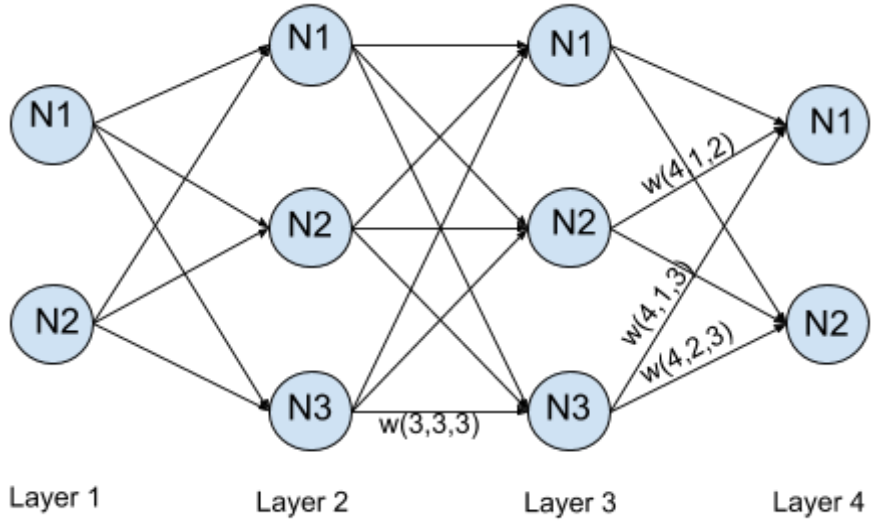

Figure 2 : feed forward multilayer perceptron

Firstly, we may generalize the error formula of the weights for any weight in the network. We simply can use the previously presented form and imagine the weight is linked to an output neuron:

$$\frac{\partial C}{\partial w(L, Nin, Nout)} = \delta(L, Nin) \times a(L-1, Nout)$$

We also need to express the error of any neuron in the network δ(L,N). If the neuron is an output, the formula is the same as for the perceptron. Otherwise, the error of a neuron depends on every weight leaving this neuron. The error is accumulated on each weight, before being transmitted to the neuron:

$$\delta(L,N) = \left(\sum_{i=1}^{x} \Delta w(L+1,i,N) \right) \times a'(L,N)$$

We could make a vectorization to calculate in a single operation all neurons error on a layer δ(L):

$$\delta(L) = \left(\sum_{i=1}^{x} \Delta w(L+1,i) \right) \cdot A'(L)$$

We identify A'(L) the vector containing the derivative of the activation function of every neuron on layer L.

Analogically, the error of all weights leaving the neuron N on layer L can be calculated as:

$$\Delta W(L, Nout) = \delta(L) \times a(L-1, Nout)$$

Stochastic refers to the fact that the weights are adapted at each training unit. This conduce to an imprecise or noisy convergence toward global minima. For each training unit, a slightly different direction would be indicated, this helps to prevent an eventual local minima. By opposition, if all training units error were averaged before adapting the weights, as the case of batch gradient descent, the convergence would be lean until a local minima is reached. Both methods are reaching near optimal solutions. Stochastic gradient descent is faster, having its weights updated more often, but on the long run, batch gradient descent will have better results (providing it did not get trapped on local minima). In general, a intermediate solution called mini batch gradient descent is used. Training units are organized randomly in small groups and the gradient is calculated on their error average. Weights are updated after each group.

Another important point to ardess is whether the learning process can be trapped in a local mina or not. Most of the time the cost function takes fouthands parameters, conducing to a high dimension number. This makes local minima unlikely compared to saddle points (local minima on some dimensions and maxima in others).

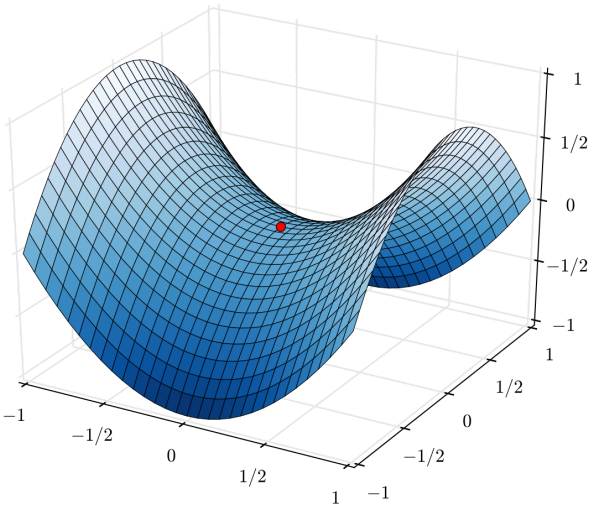

Figure 3 : Saddle point example (source: https://en.wikipedia.org/wiki/File:Saddle_point.svg)

In those points, the gradient almost equals 0. Most of the time the gradient finally falls down, but may cost much processing time.

Several techniques may help to avoid this problem. A simple one proposes to remember the previous update of the weight in order to reduce oscillations when converging. The name of this method, momentum, can be seen of an analogy of the acceleration of a ball. The new formula to adapt the weights can be defined as follow:

$$w(t) = \beta \Delta w(t-1) + \alpha \frac{\partial C}{\partial w1}$$

Where t-1 refer to the previous adaptation step of the weight and Beta is in [0,1]. If Delta w(t-1) has the same direction as the current gradient value, the update will be higher, giving a faster convergence (losing less time on saddle points). On the other hand if they are not having the same direction, the update is reduced, which will reduce oscillation (figure 4). A usual value for Beta is 0.9.

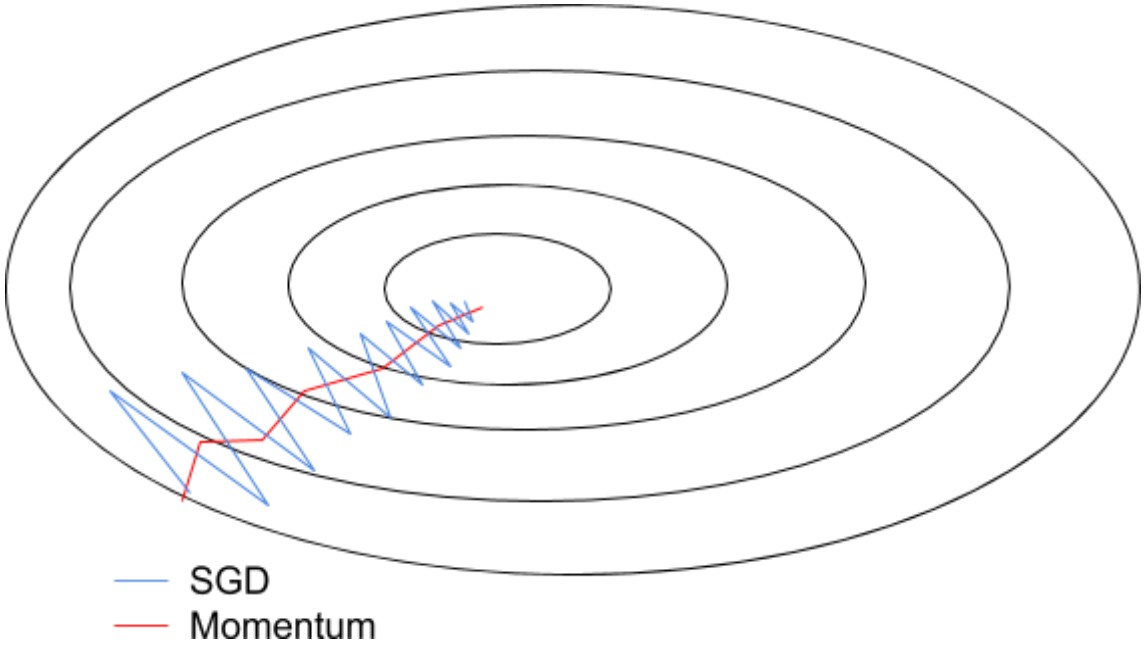

Figure 4 : momentum compared to stochastic gradient descent

Another way would be to adapt the learning rate to each gradient coordonne (each weight). This can be done by RMSprop (root mean square propagation). The learning is actually divided by the recent adaptation average of the corresponding weights.

$$adaptAvg(w,t) = \beta * adaptAvg(w,t-1) + (1-\beta)\frac{\partial C}{\partial w1}^2$$

$$w(t) = w(t) - \frac{\alpha}{(\sqrt{(adaptAvg(w,t))}} * \frac{\partial C}{\partial w1}$$

More recent methods combine momentum and RMSprop such as ADAM or AMSGrad. Not being used in the last chapter those extension won't be described. We analysed the process of learning and how could, diverse ways of adapting the weights, be converging. Other specificity about neural networks are clarified in the next section.

4. Influence of activation functions and weights initialisation

4.1. Linearity and activation functions

First we should understand why an activation function is required. The need of an activation function is to add complexity to the overall function represented by the ANN. Without it, a neural network would be a simple linear function. Effectively even with many hidden neurons, if the neuron output is not modified by a nonlinear function, the final output will still be a weighted sum of inputs.

Activation functions permit to artificially augment the dimension of the features (inputs). By choosing appropriate value of the added dimensions, it will be always possible to find a hyper-plan (of higher dimension) which separates the original features. For example, if data is defined by one input (one dimension):

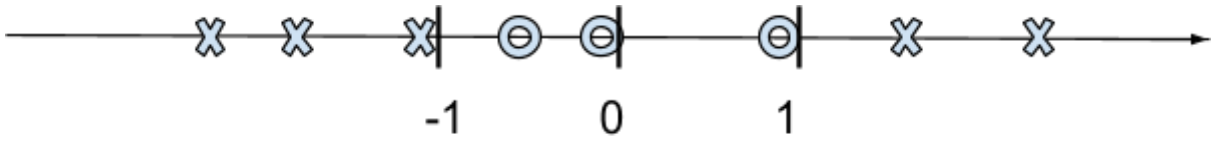

Figure 5 : data figured in one dimension

We can see in the figure 5 that no line can be traced to separate the data. Now adding the ordonee dimension and trace the modified features $x = x^2$, we obtain (figure 6) :

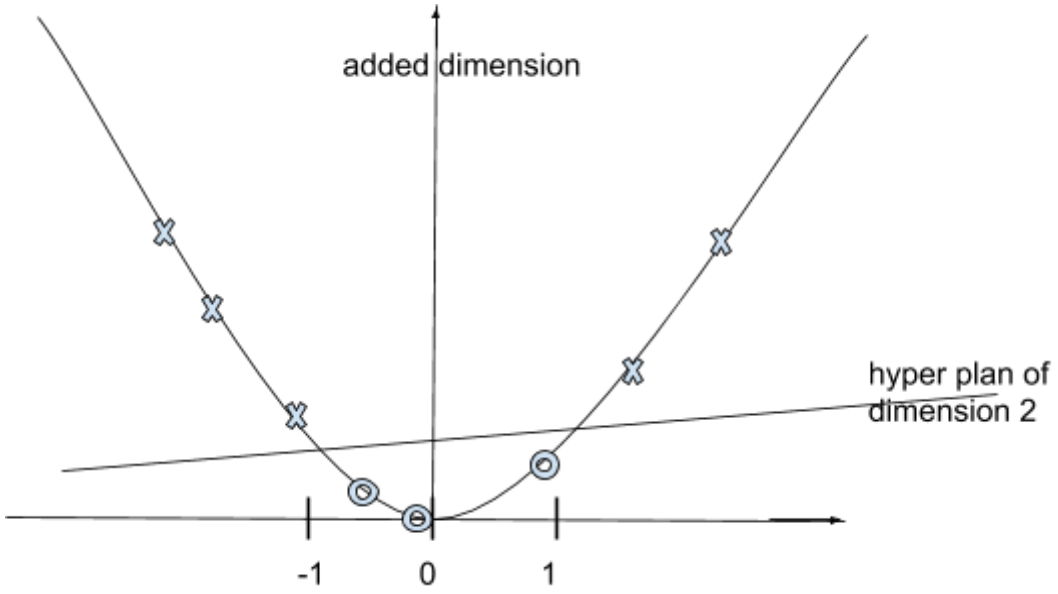

Figure 6 : modified data representation (2 dimensions)

A line can be traced to separate the data. Naturally, the activation function needs to be differentiable in order to compute ∂in/∂out in backpropagation algorithm. As rapidly seen previously, a commonly activation function used in the past few years was the sigmoid (figure 7):

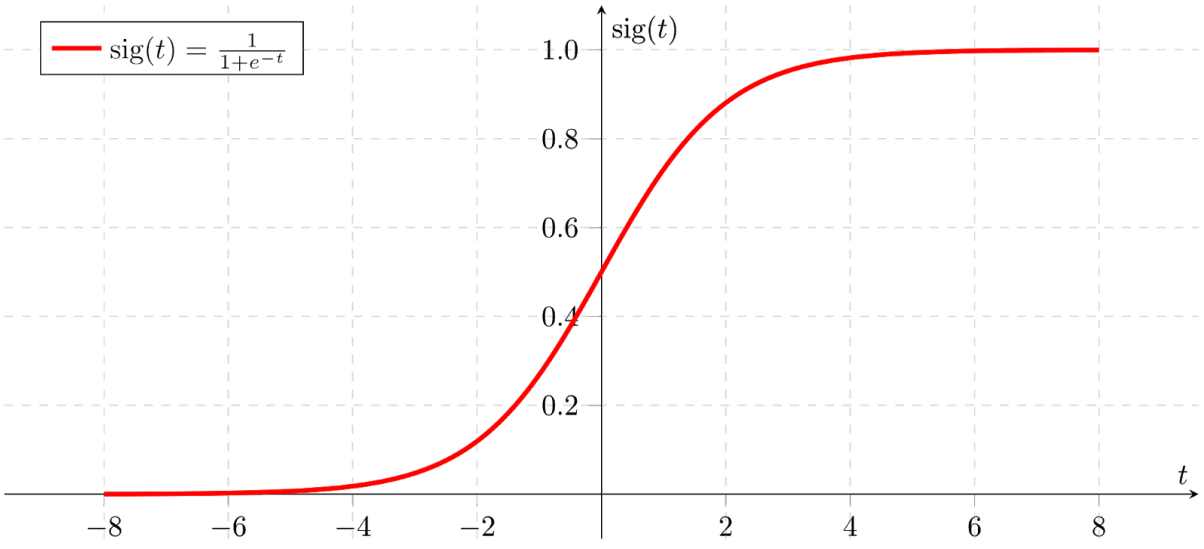

Figure 7 : sigmoid function
(source: https://commons.wikimedia.org/wiki/File:Sigmoid-function-2.svg)

This function rescales the input in the interval (0,1). Note that the derivative can be written as sig'(x) = sig(x)(1-sig(x)) which is always positive.

The output of sigmoid is not zero centered. In the case of a multi-layered neural network, a hidden neuron always outputs a value superior to 0. In the adapting weights formula:

$$\frac{\partial C}{\partial w1} = C'(out, target) * a'(ap) * x1$$

$$sign\left(\frac{\partial C}{\partial w1}\right) = sign(C'(out, target)) * 1 * 1$$

We can conclude that all gradients of the weights entering in the same neuron will have the same sign. This can slow down the learning process because in one step of backpropagation, it is not possible to increase some weights and lower others.

The problem can be solved by the hyperbolic tangent function tanh which is a rescaling of the sigmoid function (figure 8).

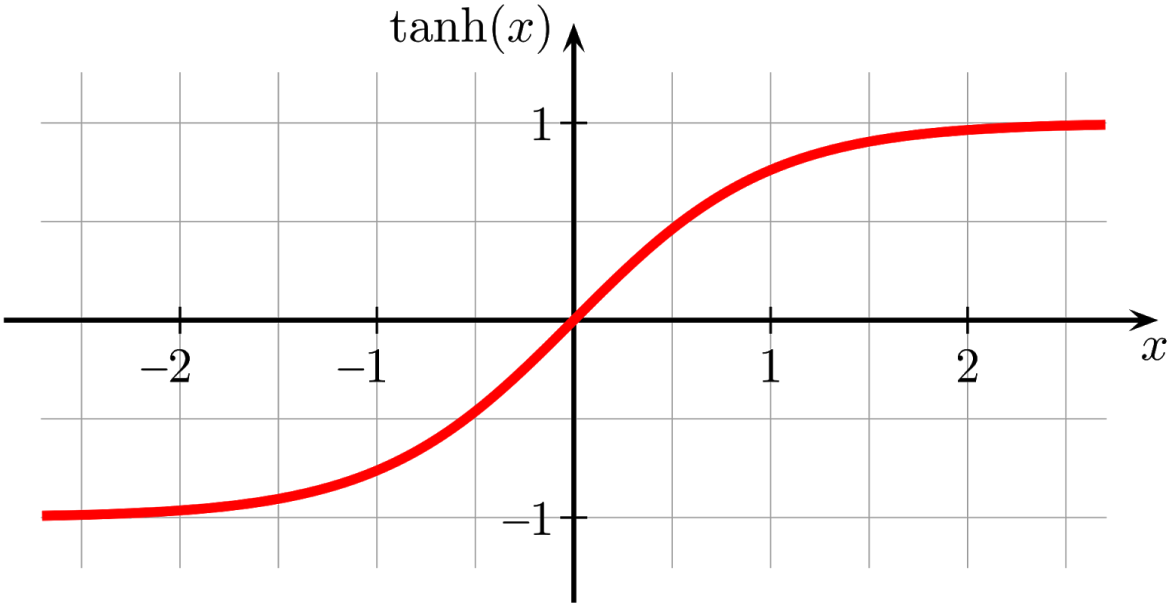

Figure 8: Tanh function (source: https://commons.wikimedia.org/wiki/File:Hyperbolic_Tangent.svg)

The range is [-1, 1] so the output is zero-centered, the weights are now able to be adapted as wished.

However, both sigmoid and tanh are saturating the outputs making the gradient to be multiplied for each layer by an absolute value between 0 and 1 making it lower and lower, after each layer. This can even cause no adaptation of

weights at all in first layers. This is called the vanishing gradient problem. Hopefully, another function is introduced: the Rectified linear unit (ReLU). It is defined as ReLU(x) = max(0, x). It is not differentiable in 0, but we can consider the value 0 or 1 if the precise value 0 is entered. The derivative equals 0 or 1, therefore, the gradient values are not decreasing. This avoid the vanishing gradient problem. Thus, in some cases, ReLU neurons can "die". Effectively, if the bias of a ReLU neuron is too negative compared to its weighted sum input, 0 will be returned. Since its derivative is also 0, there is no possibility to modify the input weights, resulting to never change the output. This will also affect previous neurons.

A simple variant of ReLU is proposed, the Leaky ReLU: Leaky ReLu(x) = max(cx, x). c is a small constant, usually 0.01. The derivative cannot equal 0 anymore, so the "dying neurons problem" is solved (figure 9).

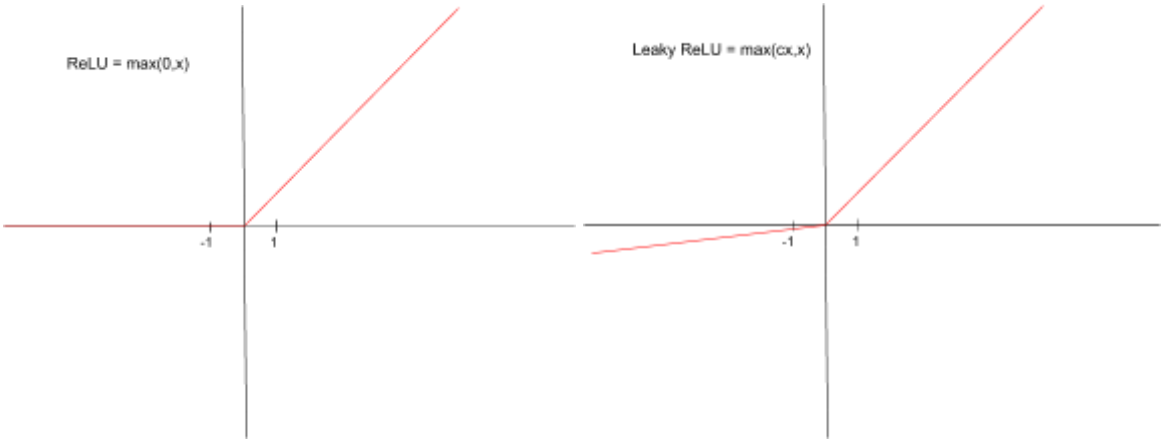

Figure 9 : ReLU and Leaky ReLU

Nowadays, the ReLu and variants are the mostly seen activation functions.

4.2. Weight initialization

A first default idea would be to initialise the weights to 0. In this case only biases are used for the computation of the outputs. This conduce the derivative of the cost function with respect to the each weights in a layer to be the same. A usually seen approach consist in taking a normal distribution centered at 0. The squared variance differ with the activation function. The purpose would be to have a similar variance of neuron outputs in different layers. A recent work [3] proposes proofs of the approximative good values for the variance. For sigmoid activation function the author recommend a variance of $v^2 \approx 1.8/N$ with N the number of weights entering in the same neuron. For the hyperbolic tangent function, this

value becomes 1/N. This last one is also known as *"Xavier initialization"*. Finally, with ReLU we have $v^2 \approx 2/N$, also referred as *"He initialisation"*. A important remark is that deeper is the network, more precise and adapted needs to be the variance.

5. Overfitting

Many neurons permit a high complexity of the model that can even classify correctly all training data. We could think this is a nice assumption, but it's actually rarely a good sign. Effectively if the training data contains noises or is not representative of the actual classification problem, the model may explain (fit) too well those wrong or incomplete data and not generalize enough. This will result in an incorrect classification of data never seen yet by the model. Such a situation is named overfitting.

By opposition underfitting corresponds to a model which did not success to learn from training data. Underfitting can only be caused by the model itself and the way it learns (too low complexity or prematurely stopped learning). We usually identify underfitting by a poor accuracy at the end of training.

For an example of overfitting, let be a model which classifies image of dogs and cats. If the train set contains only images of large dogs, the model might attributes the characteristic of being large to dogs and small to cats. Size is then becoming an important factor for the model decision. At testing, if an image of small dog is seen, the network has a chance to recognize it as a cat. In this case, overfitting happens because the training data was not representative of the real phenomena.

Overfitting caused by noise can be frequently seen with a regression task. Let consider, on the picture below (figure 10), the points to be the training data and the straight line the real function to be approximated:

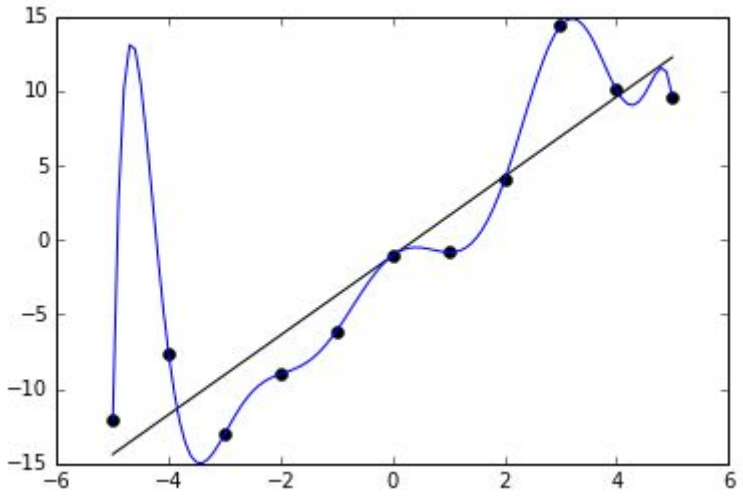

Figure 10 : regression overfitting
(source : https://en.wikipedia.org/wiki/File:Overfitted_Data.png)

Even with small noises, the model might create a complex nonlinear function which pass by all observable points, but is totally different from the original function. If the value -4.5 is tested, the error will much higher than if a standard interpolation would have been used.

Some methods exist to avoid overfitting and regularize the model. A simple way is to add more data to make noises less significant and have higher chances to have a more generalized data. Because labeled data may be expensive, we can artificially add data. For images, this can be made by adding small noises or rotations. Thus, this cannot be made automatically. Effectively the purpose is to make the data more general, which would mean that the generalized idea of the class is already known by the program.

A popular method is dropout. If the training is using dropout, some neurons are randomly inactivated and does not affect the output. The input weights are also deactivated. This produces a gain of robustness of the neurons on next layer, requiring to get the same output without one of their input. Also after backpropagation, all weights would be affected. The goal behind dropout is to combine different architectures of neural networks in a single one and dynamically while learning.

Another possibility is the use of a regularization parameter producing a L1 or L2 regularisation on the cost function. The effect of this parameter can be observed when adapting the weights. For L2, the weight is first lowered by the expression:

$$w = \left(1 - \frac{\gamma * \alpha}{n}\right) * w.$$

Gamma is the regulation parameter and n is the number of training units involved in the error calculation (mini-batch size). Then the weight is adapted normally with its gradient value. This avoids to have weights with a too high value, lowering the complexity of the model.

It is also possible to directly verify the generalization of the model by evaluating the total cost on a validation set. A validation set is generally smaller than the training set and should contain new features not encountered in the training set. During training, after a predefined number of epochs, the generalization can be evaluated by calculating the total error on the validation set. The generalization can be monitored until the error on validation set stops decreasing. This occurs when the model is beginning to calibrate its parameters to overfit the training data. Training is then stopped, giving the name of this method early stopping (figure 11). Early stopping is one of the most important method, because every complex model after a long training time will begin to overfit the data even if the training accuracy does not improve.

Figure 11 : Early stopping (modified from https://commons.wikimedia.org/wiki/File:2d-epochs-overfitting.svg)

II. Genetic algorithms

1. Definitions

Genetic algorithms (GA) proposed by J. Holland in the 1970s have interesting properties when applied to optimisation of complex problems. The theoretical foundations of GA are described in the Holland's schema theorem popularized by Goldberg. It is not surprising that GA are used to solve optimisation problems, since some natural problems as vascularisation in human body has been shown optimized. GAs attempt to simulate the genetical base and surviving law pronounced by Darwin in the 19th century, in other words, the evolution process of species in their natural environment. In Genetics an individual is fully represented by a **code** (**DNA**). This code is composed of **chromosomes**, themselves composed of **genes**. The reproduction is a random mix of chromosomes from two individuals (**cross-over**) giving birth to a new individual. The genetic **mutation** is characterised by the modification of a single gene resulting a new chromosome not existing in the parents DNA. The mutation is rare, but it explains the apparition of a different appearance in species permitting to adapt better in their environment. The mutation can also induce a less good adaptation of the individual but it will have more chance to die before to generate descendants. The **natural selection** conduce from generation to generation, a **population** composed of individuals better and better adapted.

A GA is designed in an analogue way (phases presented in figure 12). In the ensemble of solutions from an optimisation problem, a population of a fixed size N solutions (individuals) is generated. Each individual is fully represented by a code. A group of individual from the population is selected in function of their force (fitness function). Then those individual can reproduce (application of genetic operators : crossover + mutation) to complete the new population.

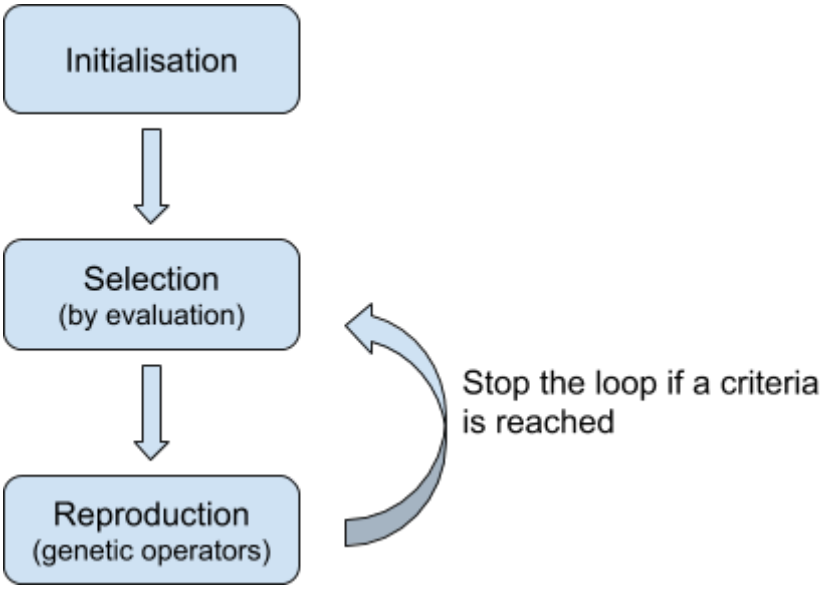

Figure 12 : Phases of a GA

2. Overview

GA are generally used to find the inverse image of the maximal or minimal value for a function which has a large solution space. In a GA, the function is abstracted in a "**fitness function**", which its domain is also abstracted in a "representation code" or **individual**. Abstraction permits first to reduce the space of solutions by doing some approximations of the real function domain (**discretization**). Secondly, through abstract representation, it is possible to set some **operations** to search in the solution space. In classical GA, we use the **binary base** as a representation. The operations altering individuals are **mutation** (bit negation) and **crossover** (re-assembling two parts of the binary code). An operation permits to jump from a solution to another by modifying a part of its representation. The new solution can be "far" from the old one. This is how global optima can be reached (figure 13).

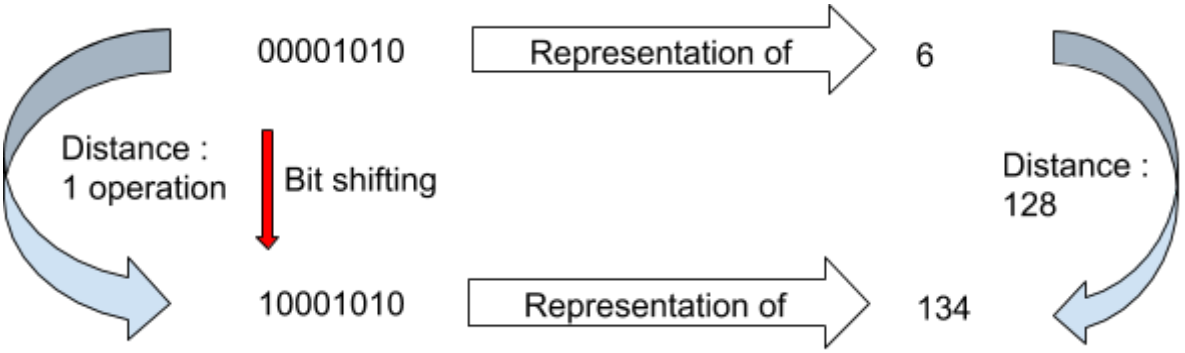

Figure 13 : fitness "jump" by a single mutation

Because an operation is only affecting a part of the representation, it allows to conserve some parts of it. Indeed, we can **choose** the "good" solutions to undergo less operations than "bad" solutions, which can even be totally replaced by new ones. We need the fitness function to identify good solutions from bad ones. This choice is also an operation named **selection**. The selection doesn't affect an individual particularly, but the whole **population**. In facts, GAs simulate the **evolution** of an ensemble of solutions permitting operation between them guided by the fitness function. As demonstrated in the **schemas theorem**, this process permits to make the representation to converge to the best solution.

3. Genetic operators

3.1. Generation of the population

First, a good representation for the solutions of the problem has to be found, a complicated task that will be touched on later. Let's admit that we chose the binary representation for a numerical problem. We now have to choose the size of the binary string representing a solution. This can be done by setting a minimal and maximal borne in the domain (bMin, respectively bMax) between which, we will search the optimal solution. If we search real numbers (infinite domain), we will need to use a precision indice to discretise our infinite domain. The bit string length is given by the formula :

$$bLenth = upperInteger\left(log_2\left(\frac{bMax - bMin}{precision}\right)\right)$$

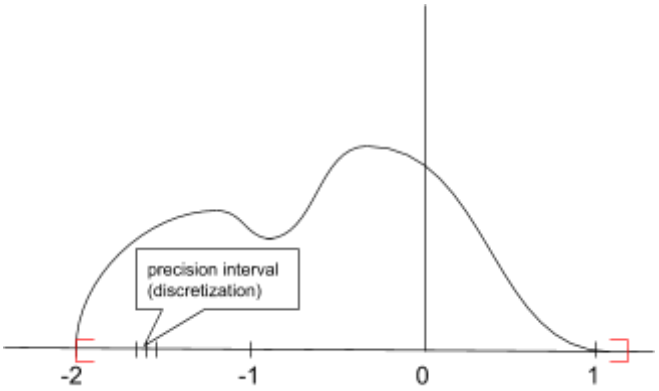

Figure 14 : Discretized function domain

We can remark (figure 14) that some of the last codifications are not allocated to a number. In facts, the real precision interval will be decreased for the bit length to become an integer without the *upperInteger* function. If the function is not too irregular, we can safely choose a precision indice of 10^{-3}.

Once the bit string length is determined, the size of the population (number of individual per generation) has to be set. No precise rule exists, the importance is to be representative of the domain function. Finally, each bit of every individual is initialized randomly or using a greedy approximation, in order to reduce the initial search space.

3.2. Fitness function

The fitness function has the purpose to evaluate how well an individual is adapted in its environment. In a GA, it is a function that attributes a score to each solution. In order to avoid blocking the search on a plateau, the fitness value above it should be reachable in few operations (not as Royal Roads functions). Another important criteria for the fitness function is to return only positive values in order to use it as probability. For the case of a numerical problem, the fitness function can be seen as a composition of two functions g and h :

$$f = h \circ g \qquad (f(x) = h(g(x))$$

with :

g : function transforming the representation of an individual to its real value. For binary representation, this is :

$$g(individual) = bMin + \frac{binToDec(x)}{2bitLength}$$, with binToDec(x) the decimal value of the bit string x.

h : original mathematical function which has to be optimized.

3.3. Selection operator

The selection tries to simulate the "natural selection" of Darwin, deciding which individuals in the population will die or reproduce and form the next population. There are many way to implement this selection, but some common properties are remaining :

- A valid selection always use the fitness value of individuals
- The size of the chosen group is fixed at N (the population size) for every generation (supposing that the population size is fixed)
- Every individual has a nonzero chance to be selected. This permits to avoid blocking at local optima.
- A pressure of selection which privilegiate more exploration (the factor of fitness value is less important, which permit a better diversification) or exploitation (the factor of fitness value is determinant for the survival of the individual, which convergence to local optima).

3.3.1. Roulette wheel selection

In Roulette wheel method, introduced by J. Holland, the chance for an individual to be chosen is proportional to its fitness value. The probability of each individual is accumulated on an axis from 0 to 1. Then we choose randomly a number r between 0 and 1. The individual corresponding to cumulative probability directly superior to the random number is selected (figure 15).

$$p(individual\ i) = \frac{f(individual\ i)}{\sum_{j=1}^{N} f(individual\ j)}$$

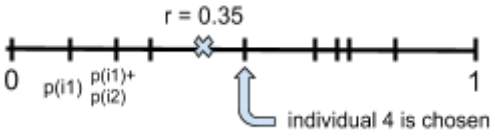

Figure 15 : Roulette wheel selection

This process is repeated until the chosen group size reaches the population size. An individual with a significantly high fitness in comparison to the others would be selected many times which induce a loss of diversity. To avoid this

problem, we usually reduce the importance of the fitness value, as in the tournament selection.

3.3.2. Tournament selection

With the tournament selection, a parameter of luck k between 0 and 1 is set (usually 0.9). We randomly choose two individual in the population. A random number r is also chosen between 0 and 1. If r<k then the individual with the better fitness value is selected, else, it is the one having the less good fitness value which is selected. The process is repeated N times. Because we only use a simple fitness comparison between the individuals, this method is the most efficient in time complexity. With this selection, the pressure of selection is not proportional to fitness, but the variance of the fitness values in the population. This time, if an individual has a significantly high fitness in comparison to the others, it will not appear sensibly more times than the second best individual, even if their fitness difference is big.

3.3.3. Elitism selection

Elitism keeps some (usually 10% of the population size) of the best individuals unmodified through the next generation. This method is used together with another selection. With this option, we ensure to never lose the best solution present in the population. Elitism has a disadvantage because it increases significantly the exploitation and may conduce all individuals to become similar to the elite, destroying diversity.

3.4. Crossover

The crossover operation reproduces the comportment of gametes when they recombine to form a new individual. In nature, there are many way the reproduction of species can be made. For this reason, the crossover in GA is not having a standard implementation. Here the single point crossover method will be presented. Nevertheless, the candidates for reproduction are all the time chosen using a similar mechanism.

Each selected individual (from the selection operator), has a chance to reproduce (set as parameter in GA, the **crossover rate c**, usually **0.8**). A random number is r is chosen and if r<=c, then the individual is added to this list of solutions undergoing the crossover. The individuals in the list are forming random pairs. The number of elements in the list may be odd. In this case different methods can be applied (like deleting the element having the lowest fitness, realizing a crossover with three parents, etc.). In the single point crossover, an

index of the code is randomly chosen for each pair. Two new individuals are created by recombination of the parents binary strings as follow (figure 16) :

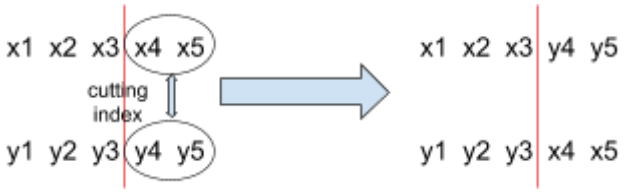

Figure 16 : crossover (example)

The crossover permits to obtain the best combination of genes present in the population, but it doesn't add a new characteristic in the population. To avoid this restriction, the mutation operator has been introduced.

3.5. Mutation

In a usual GA, the mutation is a single gene modification. Because mutation has the possibility to modify drastically he fitness function, in order not to destroy the population adaptation, it has to be rare. A **mutation rate m** of **0.01** is usually used. The mutation is applied on every individual of the new generation from crossover. A random number r [0,1] is set. If r<m, then the individual will be mutated. The gene (index) to be shifted is randomly chosen by a random integer. In bit string representation, a simple negation is made on this gene (figure 17).

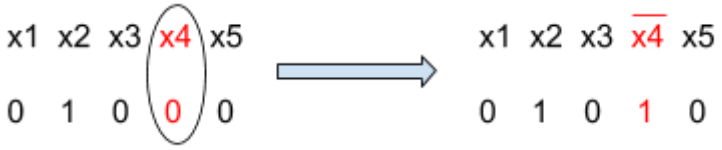

Figure 17 : Mutation (example)

When an individual is chosen for mutation, another variant would say to generate a random number r2 [0,1] for each bit. If we have r2> 1/nr bits, the corresponding bit is negated. This way, many bits or none can be modified, letting more randomness. This is given in exchange of computation cost for all the random generated numbers.

4. Schema theorem and convergence

The schemata theorem created by J. Holland provides a way to understand why genetic algorithms are converging when they do so. The schema theorem is

considered for a classical GA with binary representation, roulette wheel selection, single point recombination and simple mutation at random index. All possible individual can be written on the form {0,1}chromosome length. A schema can contain the character "*" to represent whether a 1 or a 0. This can help understanding the parallelism in genetic algorithms. For example the schema *010*1 represents 4 possible individuals :

001001, 001011, 101001, 101011.

In a more general way, a schema containing n "*" can correspond to 2n individuals.

When an individual can be represented by a schema, it is said to belong to this schema.

We define the order of a schema by the number fixed bits (not "*") in the schema:

o(*10*0*1)=4

Another important concept is the definition length of a schema which is the distance between the first and last fixed gene (bit):

δ(*1*0**1) = 7-2 = 5

The schema theorem help measuring the convergence of the solution. This is made by the calculation of the probability for an individual to be represented by a schema H. In a population at a generation t, we have n chromosomes which belong to a schema H, noted n(H,t). The expected number of selected individuals represented by the schema H can be described as:

$$n(H,t)_{selection} = \frac{n(H,t) * fit(H,t)}{fit_{avg}}$$

Where fit(H,t) is the average fitness of individuals belonging to H and fit_{avg} is the average fitness of all individuals in population at generation t.

We can observe that the number of individuals which belong to a schema will increase if its average fitness is above the total average fitness and decrease otherwise. This provide a proof of the convergence of a fixed population if no genetic operation would be made.

We will now investigate the effect of crossover on a schema. A schema disappears if its fixed genes are not conserved after the recombination. This can happen if the crossing point is between two fixed genes. For example:

i1 = 100111 which belongs to H1 = 1**1*1 and H2= **01**

i2 = 011000 is the opposite of i1.

After recombining i1 and i2, at any crossing point 2, we get

i3 = 101000 and i4 = 010111 (H1 did not survive, H2 is represented by i4)

H1 might disappear for any crossing point, while H2 could disappear only in the crossing point 3.

The crossing point can be chosen between 1 and l-1 with l the length of the individual wich conduce to a probability of a schema to disappear pd at a crossover operation:

$$pd(H)cros \leq \frac{\delta(H)}{l-1}$$

The probability of a schema to survive ps at the crossover of probability pc is:

$$ps(H)cros \geq 1 - pc\frac{\delta(H)}{l-1}$$

We can estimate the number of individuals which belong to the schema H after selection and recombination:

$$ps(H,t)_{selection+crossover} \geq n(H,t)_{selection} * \left(1 - pc\frac{\delta(H)}{l-1}\right)$$

We may notice that a short length schema will increase the number of its representative members.

Finally we need to consider the mutation operator. A schema survive if none of its imposed gene in o(H) is mutated. Since mutations are independent event from each other, the probability of a schema to survive after mutation is:

$$ps(H)_{mut} = (1 - pm)^{o(h)}$$

A schema is more likely to survive with a low order. Putting all together, we can express the number of individuals represented by a schema by the following formula:

$$ps(H,t)_{selection+crossover} \geq \frac{n(H,t) * fit(H,t)}{fit_{avg}} * \left(1 - pc\frac{\delta(H)}{l-1}\right) * (1-pm)^{o(h)}$$

The factors of recombination and mutation are inferior to 1, so the possibilities for the representative members of a schema to increase is to have a really good fitness compared to others or having low order and definition length, in which case it will rise exponentially. Those results are named the “building blocks hypothesis”. John Holland have shown that the number of schemas present in each population of size m equals m^3 or even more. He called this property of genetic algorithm the “implicit parallelism”.

It is important to keep a balance between exploration and exploitation in order to avoid the convergence in a single local optima. Schemas with high order and definition length are more precise and correspond to exploitation while schemas with more “*” permit a higher diversity in the genome.

III. Encoding Neural network parameters for Genetic algorithm

1. Neuro-evolution

We saw how both algorithms operate separately. In this part, we will investigate the possibilities to combine them. Also, a description of the experiment program with some results is presented.

In litterature, the general name for combining of ANN and GA is referred as neuroevolution. Neuroevolution ideas started in the late 80s. The principle is to employ a genetic algorithm to optimize the parameters of a neural network. Two main branches can be identified. In one case, the GA seeks only for the best initial network parameters and then optimizes them with gradient descent. The other option is to directly optimize the network parameters and not use anymore backpropagation methods. Because the representation choice of the solutions is crucial for a genetic algorithm, neuroevolution application usually differ by the way of encoding neural network parameters.

Methods can be divided in two types of encoding, direct and indirect encoding. Direct encoding stands for a direct mapping of the neural network parameters into the genotype. If a binary representation is used, one bit may correspond to the existence or not of a connection between two neurons. With indirect encoding, the genotype represents an intermediary rule which will generate the network.

As direct encoding, in early stages of neuro evolution (1992) we may cite Dasgupta and McGregor [9] for the implementation of a structured genetic algorithm (sGA). The presence of every connection is encoded in a binary matrix, with the source node as line index and destination node as column index. Up right triangle only is considered for feed forward networks. Each weights and bias values are represented by binary strings. The structure term comes from the fact that a weight depends first on its presence encoding, then on its value encoding. No backpropagation were used for adapting the weights value. We can remark that the single parameter predefined is the number of nodes. Although, the weights values are defined and optimized on a fixed length bit string. If high precision on weights is necessary for fine tuning, the length of the bit string will augment quadratically with the number of nodes.

2. Encoding choice and implementation specifications

The type of the network is restricted to a feedforward multilayer perceptron, which particularities have been explained in the first chapter. For the genetic algorithm to search efficiently the best parameters of the ANN, the main difficulty is to find an appropriate encoding. First, an enumeration of the neural network parameters can be made to further choose which one is worth encoding.

Obviously, the structure of the network has an important impact on the final error rate. By structure we can understand the number of hidden layer, the number of hidden neurons per layer, but also the connections between the layers. Parametrizing connections bring an advantage of generalization compared to fully connected networks. This would permit to automatically tune the complexity of the network. A motivation to encode the structure of the ANN comes from the wish to have a model which generalize well, to adapt the complexity and reduce overfitting.

The value of initial weights is also important. Effectively, even with a statistically robust initialisation method (as seen in section describing neural network), a potentially better structure may get a higher error on training than another, because of an unlucky initialisation and not because of the network topology. On the other hand, if each weight is encoded in a large network, the chromosome will be way too long (depending on precision of the weight value). A solution is to make many run of the same individual (network) with different weight initialization and take an average of the final errors. A set of seeds for random number generation could be given to the genetic algorithm. Few bits in the chromosome would be sufficient to associate a random seed to an individual. The choice of seed will then be optimized for each individual.

Hyper parameters such as learning rate, regularization parameter, or even the rate of dropout are interesting to optimize too, since different sizes of networks may have different learning behaviors. Dropout, having a nondeterministic application, may create wrong interpretation of an individual. Two individuals with the same parameters but different dropout rate could end up with the same efficiency (same problem as with initial weights).

The number of epochs could have the merit to be encoded if the speed of training is an important criteria. However, considering the recent advancement

made about parallelised gradient descent computation on GPU, epochs number will not be encoded in the experiment program.

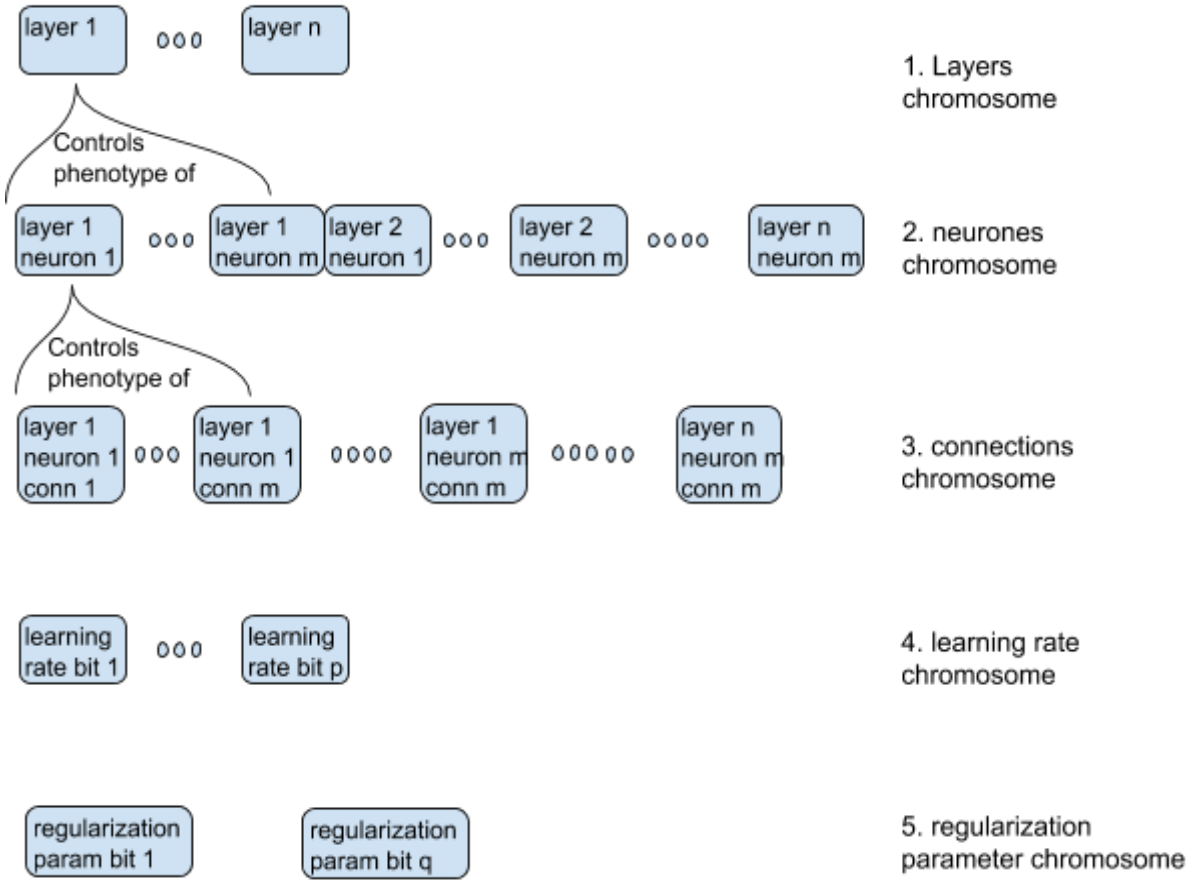

Figure 18 : hierarchical encoding

A hierarchical encoding is used (figure 18) : if the bit encoding a layer is set to 0, the neurons which could be present on this layer will be automatically disabled. This conduce also to the deactivation of the input weights of these neurons. However, the genetic material is conserved in the genotype to be used in the further genetic operators. This way, if a layer is lost in a generation for exploration, the approximative original configuration can easily be restored. Also, this permits to keep a fixed length chromosome. This empirical choice is inspired by recessive and dominant genes. This representation is similar as in the sGA of Dasgupta and McGregor, thus a noticeable difference lays in the fact that the weights value is not encoded.

Some constants are still unexplained : p and q are the bit lengths determining the precision of the learning rate and the regularization parameter. n and m are the maximal number of layers and neurons per layer respectively. n and m are the only factors which can reduce the searching space for the best neural network. Some rules of thumbs exist for majoring and minoring the complexity of a model:

“there seems to be no advantage to using more hidden units than you have training cases, since bad local minima do not occur with so many hidden units.” [4]

"A rule of thumb is for the size of this [hidden] layer to be somewhere between the input layer size ... and the output layer size ..." [4]

From the second assumptions we can set the maximal number of hidden neurons per layer m to equals the number of inputs. Combined to the first one we can set the maximal number of layer by n = N/m, with N is the number of training data. However, this can lead to a really high number of layer which is not usually recommended. In the experiment, the value is majorated by 10.

An individual is constituted of all 5 chromosomes. this division allows to set different initialisations for each chromosome. For example we could wish to have at first, a small number of layers but numerous connections. This would permit to have networks similar to fully connected ones. Here is the code of such an initialisation:

```
    for individual in range(pop_size):

            chromosome[0] = np.random.choice([True, False],
self.nr_max_hidden,0.35)

            chromosome[1] = np.random.choice([True, False],
nr_max_hidden*nr_max_neurons,0.5)

            chromosome[2] = np.random.choice([True, False],
nr_max_hidden * nr_max_neurons * nr_max_neurons +
nr_max_neurons * nr_outputs, 0.65)

            chromosome[3] = np.random.choice([True, False],
self.nr_bits_learning_rate,0.5)

            chromosome[4] = np.random.choice([True, False],
self.nr_bits_regularization_param, 0.5)

        self.population.append(chromosome)
```

Since the chosen encoding permits a direct construction of the network, the genetic operator of crossover and mutation are implemented as in the classical genetic algorithms. Nevertheless 10% of best individuals (elits) are automatically kept for forward generations. This ensure to save some good topologies.

An arbitrary choice have been to separately treat each chromosome of an individual independently. When two individuals are recombined, each chromosome is recombined with the corresponding one of the other individual. For mutation, only a single chromosome chosen randomly is modified. The detail of implementation can be found in the annexe 2.

The fitness function has the purpose to evaluate the quality of the neural network structure and hyper parameters. If the structure is wide enough, 100% of accuracy can be archived but this would probably produce overfitting. In order to avoid this problem, the accuracy will be calculated on a validation set and the implementation of early stopping is necessary. Only the rate of correctly classified objects is taken into account for the fitness function.

If the generated network is not viable (no hidden layer or connections with output), the fitness returns 0. This statement is implemented by a try: … except: in the code when calling the neural network for learning. Because of matrix multiplication, an error will rise in the feedforward pass when trying to multiply a layer activation with an empty next layer. The figure below (figure 19) describes all steps of the genetic algorithm.

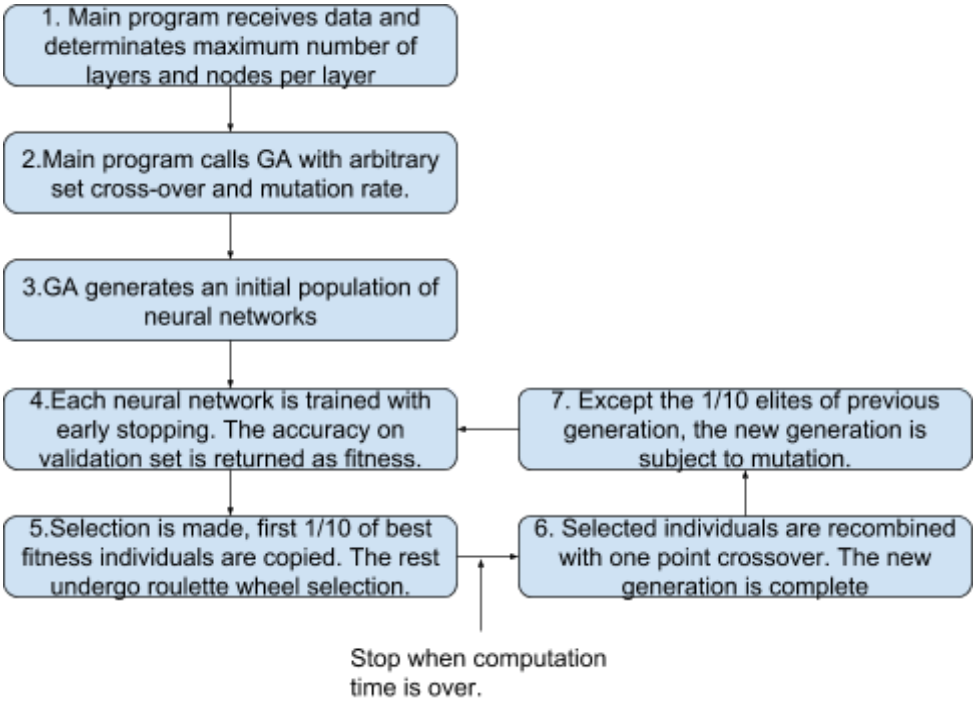

Figure 19 : steps of the experiment program

The code corresponding to the neural network has been reused from [1]. This implementation uses sigmoid activations and cross-entropy cost functions. The learning method is the standard stochastic gradient descent. Some modifications were required because the original version only supports fully connected networks. The matricial operations executed in the gradient calculations cannot be applied with a sparsely connected network. A first try has been made using for loops to replace those operations. The backpropagation was taking a more than second for only one training example ! The solution proposed was to set all non existing weights to 0 and force the delta gradient of those weights to 0 as well. Only one loop is necessary at the initialisation of the weights. A mask matrix will contain 1 for present weights and 0 otherwise.

Initialisation:

```
        self.biases = np.array([np.random.randn(y, 1) for y in self.sizes[1:]])

        self.mask_matrix = [0]*(len(self.connections)-1)

        self.weights = [0]*(len(self.connections)-1)

        for layer in range(1,len(self.connections)):  # for each layer (except input layer)

            self.weights[layer-1] = [0]*len(self.connections[layer])
```

```
                                self.mask_matrix[layer-1]          =
[0]*len(self.connections[layer])

        for neuron in range(len(self.connections[layer])):   # for
each neuron of current layer

            nr_connections = len(self.connections[layer][neuron])

            # initialise all weights as it would be fully connected

                        self.weights[layer-1][neuron]   =   [0]   *
len(self.connections[layer-1])

                    self.mask_matrix[layer  -  1][neuron]  =  [0]  *
len(self.connections[layer-1])

            for active_con in self.connections[layer][neuron]:

                    # set randnom value only for weights actually
entering in the neuron (following Xavier initialisation)

                        self.weights[layer-1][neuron][active_con]  =
np.random.randn() * np.sqrt(nr_connections)

                self.mask_matrix[layer-1][neuron][active_con] = 1
```

Adaptation of weights:

```
    nabla_w = np.multiply(nabla_w,self.mask_matrix)

    self.weights                                                    =
[(1-eta*(lmbda/n))*w-(eta/len(mini_batch))*nw

                    for w, nw in zip(self.weights, nabla_w)]
```

The dataset used for the experiments is MNIST digits database because of its reasonable complexity and its numerous existent supervised learning experiments. The MNIST dataset is composed of image (28x28 grey levels pixels) of digits written by hand. Each image is labeled with the digit represented (from 0 to 9). The set is cut in three parts: a training set containing 50 000 images, a validation set of 10 000 images and a test set with another 10 000 images. Every image is unique.

3. Results

All results are obtained through the console of the pycharm community edition IDE. The hardware specifications are the following: processeur Intel Core i5-3230M CPU 2.60GHz, 4Go RAM.

The primary results of this experiment were not concluent for computation time reasons. First, the size of data and the network required a training time many hours. The objective to see benefits of the genetic algorithm was impossible. Effectively, even the calculation of the first generation fitness would need a full week.

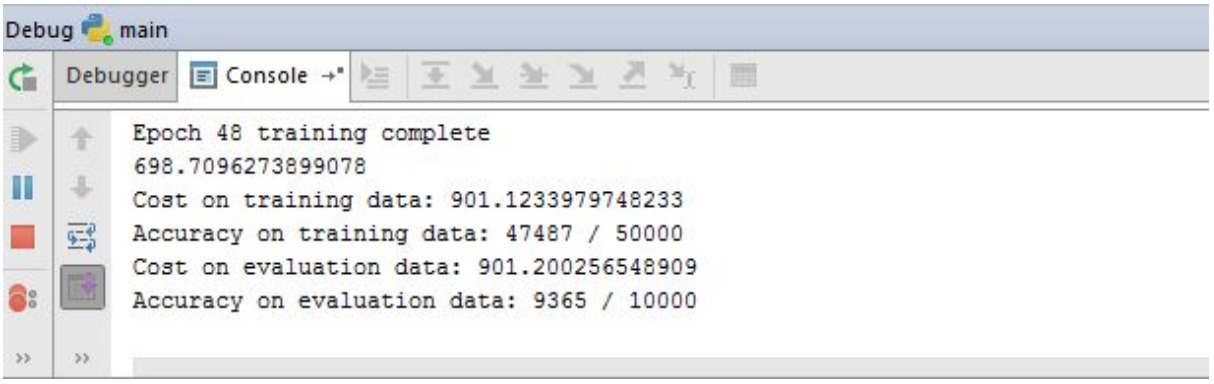

Figure 20 : Experiment result after 8 running hours. 689 is the number of seconds needed for one epoch (each training data seen once).

We could observe (figure 20) that 48 epochs is not really necessary to identify if the network generalize well or not. After 20 epochs, the accuracy only get few improvements. An idea could be to prematurely stop training if the accuracy did not improve significantly from previous epoch. A slope limitation is added in the early stopping condition:

```
if accuracy > best_accuracy * minimal_accuracy_gain_slope:
```

This would divide by 2 the computation time needed to evaluate the fitness of an individual. However, final accuracy is important to distinguish the best of close topologies. For this reason this limitation should exist only in early generations to reduce faster the searching space and ameliorating structures which are already good. *minimal_accuracy_gain_slope* equals 1.5 at start and is lowered by 0.1 after each generation until it reaches 1. A second experiment has been run reducing the training data set to 5000 examples (the evaluation set is not modified). The number of nodes per layer is also restricted to ¼ the number of inputs (196).

```
Epoch 3 training complete
9.502543449401855
Accuracy on evaluation data: 3795 / 10000
```

Figure 21 : Best fitness of first generation. 9.5 seconds for the epoch indicates a small architecture (20 for majority of others).

```
Epoch 15 training complete
7.254995731024695
Accuracy on evaluation data: 4876 / 10000
```

Figure 22 : Best fitness of third generation (18 running hours). 7.2 seconds indicates an even smaller architecture, but this time the average was 16 seconds.

The most problematic implementation part, is probably the vanishing gradient problem, which cancel the ability of stacking layers and obtaining large architecture. We observe that after few generations, that all individuals have converged towards a smaller topology. From another point of view, we may interpret this result by concluding that a small neural network is ideal for this data set. Further work suggests the use of the ReLU activation function

Eventually, as shown in [7], a committee of neural networks, all having good results but ones succeeding where other fails (with a high variance), can success to avoid some overfitting. When a test data is presented to the committee, each network calculates its own output. The output having the most votes is the one returned by the program. The use of a population in the genetic algorithm could potentially contain such a committee due to the parallelism in the solution search space. This would ensure to have a high variance in the committee, sub reserve that the GA didn’t converge in a single local minima.

Conclusion

The presented methods have a strong potential of generalization in supervised learning. Effectively 1: neural network learning, through gradient descent and its ameliorated versions, ensures an optimization of its initial parameters (weights and bias), succeeding to classify training data. 2: Genetic algorithms and early stopping find one or many organizations (structure and/or value) of neural networks initial parameters, which produce a good balance between under and overfitting. This way, both generalisation and optimization are obtained, involving better results on test data.

The genetic algorithm not only fine-tunes the hyper parameters, but also does this automatically. It aims to generate the perfect networks with only the training data as reference. This particularity means that the same implementation could be applied to every classification problem, requiring only the data to be gathered and formatted.

Nevertheless, two major inconvenients are appearing in practical applications. First, in order to easily find adapted topologies of neural networks, the genetic algorithm needs to keep some diversity in the population. This imply that the encoding of the neural network parameters and genetic operations demands careful choices. Secondly, experiments have shown that a powerful and parallelized computing architecture is necessary to obtain well performing networks in a reasonable time. However this last assumption is nuanced if final accuracy on validation test can be approximated, saving time from training.

Another direction which merits to be investigated is reinforcement learning. Since genetic algorithms are able to optimize the hidden part of the network, a suggestion would be to explore optimization of input and output layers too. Captors and action mechanisms might be adapted to serve a higher level fitness function.

Bibliography

[1] Michael A. Nielsen, "*Neural Networks and Deep Learning*", Determination Press, 2015

[2] Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Springer.

[3] Philipp Koehn, Combining Genetic Algorithms and Neural Networks: The Encoding Problem, The University of Tennessee, Knoxville, (1994)

[4] Warren S. Sarle, http://www.faqs.org/faqs/ai-faq/neural-nets/part3, 2001

[5] Siddharth Krishna Kumar, *On weight initialization in deep neural networks*, CoRR, 2017

[6] Jozef Fekiac, Ivan Zelinka and Juan C. Burguillo, *A Review Of Methods For Encoding Neural Network Topologies In Evolutionary Computation*, ECMS, 2011

[7] Rahim Barzegar, Asghar Asghari Moghaddam, *Combining the advantages of neural networks using the concept of committee machine in the groundwater salinity prediction*, Springer International Publishing Switzerland 2016

[8] Łukasz Pater, *Application of artificial neural networks and genetic algorithms for crude fractional distillation process modeling*, PKN ORLEN S.A. Płock, Poland, Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Toruń, Poland

[9] Dipankar Dasgupta and Douglas R. Mcgregor. *Designing application-specific neural networks using the structured genetic algorithm*. In Proceedings of the International Conference on Combinations of Genetic Algorithms and Neural Networks, pages 87–96. IEEE Computer Society Press, 1992.

Annexes

Annexes contain the python code of the experiment. Every used ressources are disponnible online: https://github.com/JeanMicheli/NN-GA-superisedLearning

Annexe 1 : Main program

Annexe 2 : Genetic Algorithm

Annexe 3 : Neural network

Annexe 1 : Main program

```
import numpy as np
import gzip
import pickle as cPickle
import GA
import mnistLoader as data
import time

#load data
f = gzip.open('data/mnist.pkl.gz', 'rb')
fTemp = cPickle._Unpickler(f)
fTemp.encoding = 'latin1'
train_set, validation_set, test_set = fTemp.load()
f.close()

classes = list(set(train_set[1]))
nr_inputs = len(train_set[0][0])
nr_training_units = len(train_set[0][:5000])
nr_output = len(classes)

#rules of thumb applications
nr_max_neurons = int(nr_inputs/4)
```

```
    nr_max_hidden =
min(int(nr_training_units/(nr_max_neurons)),10)
    nr_max_hidden = max(nr_max_hidden,1)

    train_set, validation_set, test_set =
data.load_data_wrapper()

    pop_size = 40
    crossover_rate = 0.5
    mutation_rate = 0.05
    nr_bits_learning_rate = 10
    nr_bits_regularization_param = 10
    start = time.time()
    myGA = GA.genetic_algorithm(train_set, validation_set,
nr_max_hidden, nr_max_neurons, nr_output, nr_inputs,
                                nr_bits_learning_rate,
nr_bits_regularization_param,
                                pop_size, crossover_rate,
mutation_rate)
    myGA.minimal_accuracy_gain_slope = 1.5
    precision = 0.1
    while time.time() - start < 3600 * 17: # 17 hours of run
        myGA.nn_population = []
        # train and select neural networks, also best 10% are
saved in best_nn and their genotype is
        # reused for next population
        selected_pop = myGA.selection()
        myGA.crossover_operation(selected_pop) # will complete
the new population
        myGA.mutation_operation() # randomly mutate some
individuals
        myGA.population = myGA.new_population

myGA.minimal_accuracy_gain_slope=max(myGA.minimal_accuracy_gai
n_slope-precision,1)
    file_name_inc = 1
    for bestNN in myGA.best_nn:

bestNN.save('Trained_network'+str(file_name_inc)+'.txt')
        file_name_inc+=1
```

Annexe 2 : Genetic algorithm

```
import numpy as np
import NNmodified as NN
import random
import copy


class genetic_algorithm(object):
   def __init__(self,train_set, validation_set,
nr_max_hidden, nr_max_neurons, nr_outputs,
                     nr_inputs, nr_bits_learning_rate,
nr_bits_regularization_param, pop_size,
                     crossover_rate, mutation_rate,
elitism_rate = 0.1):

            # global values to calculate only once :
            self.index_begin_connections = nr_max_hidden +
nr_max_hidden * nr_max_neurons
            self.nr_bits_learning_rate = nr_bits_learning_rate
            self.denominator_learning_rate = pow(2,
nr_bits_learning_rate) * 2  # range learning rate = [0.000001,
0.5000001]
            self.nr_bits_regularization_param =
nr_bits_regularization_param
            self.denominator_regularization_param = pow(2,
nr_bits_regularization_param)  # range regularization param =
[0.01, 0.51]


            self.nr_max_hidden = nr_max_hidden
            self.nr_max_neurons = nr_max_neurons
            self.nr_outputs = nr_outputs
            self.nr_inputs = nr_inputs
            self.population = []
            chromosome = [0] * 5
            for individual in range(pop_size):
                chromosome = [0] * 5
                chromosome[0] = np.random.choice([True,
False], self.nr_max_hidden,0.35)
```

```
                chromosome[1] = np.random.choice([True,
False], nr_max_hidden*nr_max_neurons,0.5)
                chromosome[2] = np.random.choice([True,
False], nr_max_hidden * nr_max_neurons * nr_max_neurons +
nr_max_neurons * nr_outputs, 0.65)
                chromosome[3] = np.random.choice([True,
False], self.nr_bits_learning_rate,0.5)
                chromosome[4] = np.random.choice([True,
False], self.nr_bits_regularization_param, 0.5)
                self.population.append(chromosome)

            self.crossover_rate = crossover_rate
            self.mutation_rate = mutation_rate
            self.train_set = train_set
            self.validation_set = validation_set
            self.elitism_rate = elitism_rate
            self.nn_population = []

        def get_fitness(self, chromosome):
            connections = [[0]*self.nr_inputs]  # set
connections of input layer empty (no connection before input
layer)
            index_neurons = 0
            index_connections = 0
            first_hidden_layer_activated = False
            neurons_numeral_previous_layer = []
            neurons_numeral_cur_layer = []
            for cur_layer_index in range(len(chromosome[0])):
# for each bit corresponding to hidden layer presence
                if chromosome[0][cur_layer_index]:

                    connections.append([])  # create layers
list
                    neuron_number = 0
                    for cur_neuron_index in
range(index_neurons, index_neurons+self.nr_max_neurons):  #
for each bit corresponding to neuron presence in current
hidden layer
                        index_end_connections =
index_connections + self.nr_max_neurons
                        if chromosome[1][cur_neuron_index]:
```

```
neurons_numeral_cur_layer.append(neuron_number)
                            connections[-1].append([])  #
create neurons list
                            connection_number=0
                            for connection in
range(index_connections, index_end_connections):  # for each
bit corresponding to connection presence toward current neuron
                                # verify existence of
connection and neuron sender from previous layer
                                if chromosome[2][connection]:
                                    if not
first_hidden_layer_activated:
                                # connect neuron n° 'neuron'
from current layer to neuron n° 'connection_number' from
previous activatedlayer

connections[-1][-1].append(connection_number)  # every neurons
on input layer are existing
                                    else:
                                        try:
                                            # connect neuron
n° 'neuron' from current layer to neuron n°
'connection_number' from previous activatedlayer

connections[-1][-1].append(neurons_numeral_previous_layer.inde
x(connection_number))
                                        except:
                                            pass
                                connection_number += 1
                        index_connections =
index_end_connections
                        neuron_number += 1
                    neurons_numeral_previous_layer =
list(neurons_numeral_cur_layer)
                    neurons_numeral_cur_layer = []
                    first_hidden_layer_activated = True

                index_neurons += self.nr_max_neurons

            # output layer connections
            if not first_hidden_layer_activated:
```

```
                return 0  # no hidden layer activated, not
viable chromosome
            connections.append([])  # create layers list
            for output_neuron in range(self.nr_outputs):  #
for each bit corresponding to neuron presence in current
hidden layer
                index_end_connections = index_connections +
self.nr_max_neurons
                connections[-1].append([])  # create neurons
list
                connection_number = 0
                # for each bit corresponding to connection
presence toward current neuron
                for connection in range(index_connections,
index_end_connections):
                    # verify existence of connection and
previous neuron
                    if chromosome[2][connection]:
                        try:
            # connect neuron n° 'neuron' from current layer to
neuron n° 'connection_number' from previous activatedlayer

connections[-1][-1].append(neurons_numeral_previous_layer.inde
x(connection_number))
                        except:
                            pass
                    connection_number += 1
                index_connections = index_end_connections

            learning_rate = 0
            power_of_two = self.denominator_learning_rate/4
            for bit_learning_rate in chromosome[3]:  # for
last nr_bits_learning_rate bits of chr
                if bit_learning_rate:
                    learning_rate += power_of_two
                power_of_two /= 2
            learning_rate /= self.denominator_learning_rate
            learning_rate+=0.000001

            regularization_param = 0
            power_of_two =
self.denominator_regularization_param/4
```

```
            for bit_regularization_param in chromosome[4]:  #
for last nr_bits_regularization_param bits of chr
                if bit_regularization_param:
                    regularization_param += power_of_two
                power_of_two /= 2
            regularization_param /=
self.denominator_regularization_param
            regularization_param += 0.01
            try:
                nn = NN.Network(connections)  # ,
self.train_set, learning_rate, regularization_param
                fitness = nn.SGD(self.train_set,1000,20,
learning_rate, regularization_param, self.validation_set,
True, 3,

self.minimal_accuracy_gain_slope)
                self.nn_population.append(nn)
                return fitness
            except:
                return 0

        def selection(self):
            all_fit = [self.get_fitness(individual) for
individual in self.population]
            ordered_fit = sorted(all_fit)
            # remember indexes of corresponding individuals
            index_population=sorted(range(len(all_fit)),
key=lambda k: all_fit[k])

            # save elits for next generation
            self.new_population =
[copy.deepcopy(self.population[index_population[i]])
                                    for i in
range(1,int(len(self.population)*self.elitism_rate)+1)]
            self.best_nn =
[self.nn_population[all_fit.index(ordered_fit[-i])]
                                    for i in
range(1,int(len(self.population)*self.elitism_rate)+1)]
            ordered_fit = np.divide(ordered_fit, sum(all_fit))
# make fitness on a probability form
            cumulativ_fit=0
            selected_pop = []
```

```
            for fit_index in range(len(ordered_fit)):
                if ordered_fit[fit_index]+cumulativ_fit >
random.random():

selected_pop.append(self.population[index_population[fit_index
]])
                cumulativ_fit += ordered_fit[fit_index]
            return selected_pop

        def crossover_operation(self, selected_pop):
            while len(self.new_population) <
len(self.population):
                index_parent1 =
random.randint(0,len(selected_pop))
                index_parent2 =
random.randint(0,len(selected_pop))
                while index_parent1 == index_parent2:
                    index_parent1 = random.randint(0,
len(selected_pop)-1)
                    index_parent2 = random.randint(0,
len(selected_pop)-1)
                if self.crossover_rate > random.random():
                    child1 = []
                    child2 = []
                    for chr_index in
range(len(self.population[index_parent1])):
                        child1.append([])
                        child2.append([])
                        cutting_point =
random.randint(1,len(self.population[index_parent1][chr_index]
-1))
                        child1[-1] =
np.append(self.population[index_parent1][chr_index][:cutting_p
oint],

self.population[index_parent2][chr_index][cutting_point:])
                        child2[-1] =
np.append(self.population[index_parent2][chr_index][:cutting_p
oint],

self.population[index_parent1][chr_index][cutting_point:])
                    self.new_population.append(child1)
```

```
                    self.new_population.append(child2)
                else:

self.new_population.append(copy.deepcopy(self.population[index
_parent1]))

self.new_population.append(copy.deepcopy(self.population[index
_parent2]))

            if len(self.new_population) >
len(self.population):
                self.new_population.pop(-1)
            return

        def mutation_operation(self):
            for individual_index in
range(int(len(self.new_population)*self.elitism_rate),
len(self.new_population)):
                if random.random() < self.mutation_rate:
#determine if the current individual will undergo mutation
                    chromosome_index =
random.randint(0,len(self.population[0])-1)
                    gene_index  =
random.randint(0,len(self.population[0][chromosome_index])-1)

self.new_population[individual_index][chromosome_index][gene_i
ndex] = \
                        not
self.new_population[individual_index][chromosome_index][gene_i
ndex]
            return
```

Annexe 3: Neural network

Source: https://github.com/mnielsen/neural-networks-and-deep-learning

```
#### Libraries
     Michael A. Nielsen, "Neural Networks and Deep Learning",
Determination Press, 2015
     # Standard library
     import json
     import random
     import sys
     import time
     import copy

     # Third-party libraries
     import numpy as np


     #### Define the quadratic and cross-entropy cost
functions

     class QuadraticCost(object):

        @staticmethod
        def fn(a, y):
            """Return the cost associated with an output ``a``
and desired output
            ``y``.
            """
            return 0.5*np.linalg.norm(a-y)**2

        @staticmethod
        def delta(z, a, y):
            """Return the error delta from the output
layer."""
            return (a-y) * sigmoid_prime(z)


     class CrossEntropyCost(object):

        @staticmethod
        def fn(a, y):
```

```
            """Return the cost associated with an output ``a``
and desired output
            ``y``.  Note that np.nan_to_num is used to ensure
numerical
            stability.  In particular, if both ``a`` and ``y``
have a 1.0
            in the same slot, then the expression
(1-y)*np.log(1-a)
            returns nan.  The np.nan_to_num ensures that that
is converted
            to the correct value (0.0).
            """
            temp =
np.sum(np.nan_to_num(-y*np.nan_to_num(np.log(a))-(1-y)*np.nan_
to_num(np.log(1-a))))
            if  np.isnan(temp):
                temp = np.nan_to_num(temp)
            return temp

        @staticmethod
        def delta(z, a, y):
            """Return the error delta from the output layer.
Note that the
            parameter ``z`` is not used by the method.  It is
included in
            the method's parameters in order to make the
interface
            consistent with the delta method for other cost
classes.
            """
            return (a-y)


    #### Main Network class
    class Network(object):

        def __init__(self, connections,
cost=CrossEntropyCost):
            """The list ``sizes`` contains the number of
neurons in the respective
            layers of the network.  For example, if the list
was [2, 3, 1]
```

```
            then it would be a three-layer network, with the
first layer
            containing 2 neurons, the second layer 3 neurons,
and the
            third layer 1 neuron.  The biases and weights for
the network
            are initialized randomly, using
            ``self.default_weight_initializer`` (see docstring
for that
            method).
            """
            self.num_layers = len(connections)
            self.sizes = [len(connections[i]) for i in
range(self.num_layers)]
            self.connections = connections
            self.cost = cost
            self.default_weight_initializer()

        def default_weight_initializer(self):
            """Initialize each weight using a Gaussian
distribution with mean 0
            and standard deviation 1 over the square root of
the number of
            weights connecting to the same neuron.  Initialize
the biases
            using a Gaussian distribution with mean 0 and
standard
            deviation 1.
            Note that the first layer is assumed to be an
input layer, and
            by convention we won't set any biases for those
neurons, since
            biases are only ever used in computing the outputs
from later
            layers.
            """
            self.biases = np.array([np.random.randn(y, 1) for
y in self.sizes[1:]])
            self.mask_matrix = [0]*(len(self.connections)-1)
            self.weights = [0]*(len(self.connections)-1)
            for layer in range(1,len(self.connections)):  #
for each layer (except input layer)
```

```
                self.weights[layer-1] =
[0]*len(self.connections[layer])
                self.mask_matrix[layer-1] =
[0]*len(self.connections[layer])
                for neuron in
range(len(self.connections[layer])):  # for each neuron of
current layer
                    nr_connections =
len(self.connections[layer][neuron])
                    # initialise all weights as it would be
fully connected
                    self.weights[layer-1][neuron] = [0] *
len(self.connections[layer-1])
                    self.mask_matrix[layer - 1][neuron] = [0]
* len(self.connections[layer-1])
                    for active_con in
self.connections[layer][neuron]:
                        # set radnom value only for weights
actually entering in the neuron

self.weights[layer-1][neuron][active_con] = np.random.randn()
* np.sqrt(nr_connections)

self.mask_matrix[layer-1][neuron][active_con] = 1
                self.weights[layer - 1] =
np.array(self.weights[layer-1])
        def feedforward(self, a):
            """Return the output of the network if ``a`` is
input."""
            for b, w in zip(self.biases, self.weights):
                a = sigmoid(np.dot(w, a)+b)
            return a

        def SGD(self, training_data, epochs, mini_batch_size,
eta,
                lmbda = 0.0,
                evaluation_data=None,
                monitor_evaluation_accuracy=False,
                early_stopping_n = 0,
                minimal_accuracy_gain_slope = 1):
            """Train the neural network using mini-batch
stochastic gradient
```

```
            descent.  The ``training_data`` is a list of
tuples ``(x, y)``
            representing the training inputs and the desired
outputs.  The
            other non-optional parameters are
self-explanatory, as is the
            regularization parameter ``lmbda``.  The method
also accepts
            ``evaluation_data``, usually either the validation
or test
            data.  We can monitor the cost and accuracy on
either the
            evaluation data or the training data, by setting
the
            appropriate flags.  The method returns a tuple
containing four
            lists: the (per-epoch) costs on the evaluation
data, the
            accuracies on the evaluation data, the costs on
the training
            data, and the accuracies on the training data.
All values are
            evaluated at the end of each training epoch.  So,
for example,
            if we train for 30 epochs, then the first element
of the tuple
            will be a 30-element list containing the cost on
the
            evaluation data at the end of each epoch. Note
that the lists
            are empty if the corresponding flag is not set.
            """

            # early stopping functionality:
            best_accuracy=1

            training_data = list(copy.deepcopy(training_data))
            n = len(training_data)

            if evaluation_data:
                evaluation_data =
list(copy.deepcopy(evaluation_data))
```

```
            n_data = len(evaluation_data)

        # early stopping functionality:
        best_accuracy=0
        no_accuracy_change=0

        evaluation_cost, evaluation_accuracy = [], []
        training_cost, training_accuracy = [], []
        for j in range(epochs):
            start = time.time()
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(
                    mini_batch, eta, lmbda,
len(training_data))

            print("Epoch %s training complete" % j)
            stop = time.time()
            print(stop - start)
            if monitor_evaluation_accuracy:
                accuracy = self.accuracy(evaluation_data)
                evaluation_accuracy.append(accuracy)
                print("Accuracy on evaluation data: {} /
{}".format(self.accuracy(evaluation_data), n_data))

            # Early stopping:
            if early_stopping_n > 0:
                if accuracy > best_accuracy *
minimal_accuracy_gain_slope:
                    best_accuracy = accuracy
                    no_accuracy_change = 0
                    #print("Early-stopping: Best so far
{}".format(best_accuracy))
                else:
                    no_accuracy_change += 1

                if (no_accuracy_change ==
early_stopping_n):
```

```
                        #print("Early-stopping: No accuracy
change in last epochs: {}".format(early_stopping_n))
                        return best_accuracy

            return best_accuracy

        def update_mini_batch(self, mini_batch, eta, lmbda,
n):
            """Update the network's weights and biases by
applying gradient
            descent using backpropagation to a single mini
batch.  The
            ``mini_batch`` is a list of tuples ``(x, y)``,
``eta`` is the
            learning rate, ``lmbda`` is the regularization
parameter, and
            ``n`` is the total size of the training data set.
            """
            nabla_b = [np.zeros(b.shape) for b in self.biases]
            nabla_w = [np.zeros(w.shape) for w in
self.weights]
            for x, y in mini_batch:
                delta_nabla_b, delta_nabla_w =
self.backprop(x, y)
                nabla_b = [nb+dnb for nb, dnb in zip(nabla_b,
delta_nabla_b)]
                nabla_w = [nw+dnw for nw, dnw in zip(nabla_w,
delta_nabla_w)]
            nabla_w = np.multiply(nabla_w,self.mask_matrix)
            self.weights =
[(1-eta*(lmbda/n))*w-(eta/len(mini_batch))*nw
                            for w, nw in zip(self.weights,
nabla_w)]
            # for layer in range(1,len(self.connections)):  #
for each layer (except input layer)
            #     for neuron in
range(len(self.connections[layer])):  # for each neuron of
current layer
            #         for active_con in
self.connections[layer][neuron]:
            #
self.weights[layer-1][neuron][active_con] = \
```

```
            #
(1-eta*(lmbda/n))*self.weights[layer-1][neuron][active_con]-\
            #
(eta/len(mini_batch))*nabla_w[layer-1][neuron][active_con]

            self.biases = [b-(eta/len(mini_batch))*nb
                           for b, nb in zip(self.biases,
nabla_b)]

        def backprop(self, x, y):
            """Return a tuple ``(nabla_b, nabla_w)``
representing the
            gradient for the cost function C_x.  ``nabla_b``
and
            ``nabla_w`` are layer-by-layer lists of numpy
arrays, similar
            to ``self.biases`` and ``self.weights``."""
            nabla_b = [np.zeros(b.shape) for b in self.biases]
            nabla_w = [np.zeros(w.shape) for w in
self.weights]
            # feedforward
            activation = x
            activations = [x] # list to store all the
activations, layer by layer
            zs = [] # list to store all the z vectors, layer
by layer
            for b, w in zip(self.biases, self.weights):
                z = np.dot(w, activation)+b
                zs.append(z)
                activation = sigmoid(z)
                activations.append(activation)
            # backward pass
            delta = (self.cost).delta(zs[-1], activations[-1],
y)
            nabla_b[-1] = delta
            nabla_w[-1] = np.dot(delta,
activations[-2].transpose())
            # Note that the variable l in the loop below is
used a little
            # differently to the notation in Chapter 2 of the
book.  Here,
```

```
            # l = 1 means the last layer of neurons, l = 2 is
the
            # second-last layer, and so on.  It's a
renumbering of the
            # scheme in the book, used here to take advantage
of the fact
            # that Python can use negative indices in lists.
            for l in range(2, self.num_layers):
                z = zs[-l]
                sp = sigmoid_prime(z)
                delta = np.dot(self.weights[-l+1].transpose(),
delta) * sp
                nabla_b[-l] = delta
                nabla_w[-l] = np.dot(delta,
activations[-l-1].transpose())
            return (nabla_b, nabla_w)

        def accuracy(self, data, convert=False):
            """Return the number of inputs in ``data`` for
which the neural
            network outputs the correct result. The neural
network's
            output is assumed to be the index of whichever
neuron in the
            final layer has the highest activation.
            The flag ``convert`` should be set to False if the
data set is
            validation or test data (the usual case), and to
True if the
            data set is the training data. The need for this
flag arises
            due to differences in the way the results ``y``
are
            represented in the different data sets.  In
particular, it
            flags whether we need to convert between the
different
            representations.  It may seem strange to use
different
            representations for the different data sets.  Why
not use the
```

```
            same representation for all three data sets?  It's
done for
            efficiency reasons -- the program usually
evaluates the cost
            on the training data and the accuracy on other
data sets.
            These are different types of computations, and
using different
            representations speeds things up.  More details on
the
            representations can be found in
            mnist_loader.load_data_wrapper.
            """
            if convert:
                results = [(np.argmax(self.feedforward(x)),
np.argmax(y))
                           for (x, y) in data]
            else:
                results = [(np.argmax(self.feedforward(x)), y)
                           for (x, y) in data]

            result_accuracy = sum(int(x == y) for (x, y) in
results)
            return result_accuracy

        def total_cost(self, data, lmbda, convert=False):
            """Return the total cost for the data set
``data``.  The flag
            ``convert`` should be set to False if the data set
is the
            training data (the usual case), and to True if the
data set is
            the validation or test data.  See comments on the
similar (but
            reversed) convention for the ``accuracy`` method,
above.
            """
            cost = 0.0
            for x, y in data:
                a = self.feedforward(x)
                if convert: y = vectorized_result(y)
                cost += self.cost.fn(a, y)/len(data)
```

```
                cost +=
0.5*(lmbda/len(data))*sum(np.linalg.norm(w)**2 for w in
self.weights) # '**' - to the power of.
            return cost

        def save(self, filename):
            """Save the neural network to the file
``filename``."""
            data = {"sizes": self.sizes,
                    "weights": [w.tolist() for w in
self.weights],
                    "biases": [b.tolist() for b in
self.biases],
                    "cost": str(self.cost.__name__)}
            f = open(filename, "w")
            json.dump(data, f)
            f.close()

    #### Loading a Network
    def load(filename):
        """Load a neural network from the file ``filename``.
Returns an
        instance of Network.
        """
        f = open(filename, "r")
        data = json.load(f)
        f.close()
        cost = getattr(sys.modules[__name__], data["cost"])
        net = Network(data["sizes"], cost=cost)
        net.weights = [np.array(w) for w in data["weights"]]
        net.biases = [np.array(b) for b in data["biases"]]
        return net

    #### Miscellaneous functions
    def vectorized_result(j):
        """Return a 10-dimensional unit vector with a 1.0 in
the j'th position
        and zeroes elsewhere.  This is used to convert a digit
(0...9)
        into a corresponding desired output from the neural
network.
        """
```

```
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e

def sigmoid(z):
    """The sigmoid function."""
    return np.nan_to_num(1.0/(1.0+np.exp(-z)))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```