Overview of the statistical technique used in Artificial neural network (ANN)

Stefano Truzzi

Teacher: Paolo Francavilla

December 14, 2021

1 Introduction

The **Artificial neural network (ANN)** are a new type of algorithms that implement system that can automatic improve and learn through the use of **big data**. These algorithms are inspired by the systems of connection and neuron that characterize the animal brains. All ANN algorithms are based on a sort of complex sets of **regression** functions. In order to show how the ANN work we start to treat the **Linear regression** with the **Ordinary Least Squares** and then move the discussion to the **Gradient Descent**, a technique used to find the relative minimum of a function widely used in the ANN. In the end we show the Back Propagation (BP) a method to implement the calculation of the minimum of the loss function with a computer algorithm using the GD/SGD.

2 Linear regression and Least Squares method

The **linear regression** is the set of linear estimation techniques used to express the linear relationship between a scalar with one or more variables. In small words if one want to find a function that approximate the trend of linear distributed data sample one can use a linear regression techniques to do that. The **Ordinary Least Squares** method is a linear regression technique used to approximate a solution of a linear distribuited data problem. Following the Figure: 1 we calculate the linear

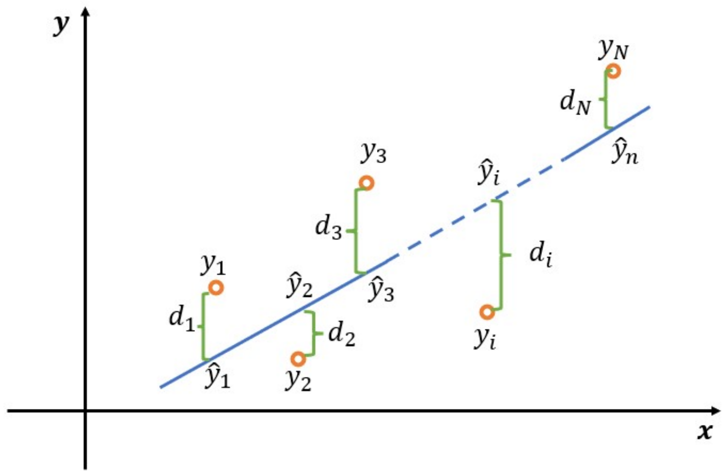


Figure 1: A generic example of OLS where y_i are the empirical data, $\hat{y}_i$ are the estimated data d_i are the distance/differences between the empirical and estimated data $(\hat{y}_i - y_i)$ the blue line is the estimated linear fit calculated with the distances

regression for a generic linear distributed set of data. We are interested in find a linear function that can represent the distance d, intuitively the "best" function should be the one that have smaller combination of d_i. Mathematically, this can be achieved finding the minimum of the function of the sum of d_i; the minimum can be calculated setting the first derivative equal to zero.

$$\sum_{i=1}^{N}(d_i)^2 = \sum_{i=1}^{N}(\hat{y_i} - y_i)^2 \tag{1}$$

Where we use the square d because a distance is always positive. We write $\hat{y_i}$ in the form of a linear function:

$$\hat{y_i} = ax_i + b \tag{2}$$

This function is called **linear regression function** and, in this case, is represented by a straight line with a angular coefficient and b for the intercept.

$$\sum_{i=1}^{N}(ax_i + b - y_i)^2 = f(a,b) \tag{3}$$

We want to minimize the function for the coefficient a and b. To do that we solve the linear system of the partial first derivatives.

$$\begin{cases} \dfrac{\partial f(a,b)}{a} = 0 \\ \dfrac{\partial f(a,b)}{b} = 0 \end{cases}$$

We exchange the order of sum and derivative operator and then we calculate the first derivative

$$\begin{cases} \sum_{i=1}^{N} 2(ax_i + b - y_i)(x_i) = 0 \\ \sum_{i=1}^{N} 2(ax_i + b - y_i) = 0 \end{cases}$$

$$\begin{cases} a\sum_{i=1}^{N} x_i^2 + b\sum_{i=1}^{N} x_i - \sum_{i=1}^{N} x_i y_i = 0 \\ a\sum_{i=1}^{N} x_i + bN - \sum_{i=1}^{N} y_i = 0 \end{cases}$$

Finally, we obtain the value for a and b that minimize the function.

$$\begin{cases} a = \dfrac{N\sum x_i y_i - \sum x_i \sum y_i}{N\sum x_i^2 - (\sum x_i)^2} \\ \\ b = \dfrac{\sum y_i \sum x_i^2 - \sum x_i \sum x_i y_i}{N\sum x_i^2 - (\sum x_i)^2} \end{cases} \tag{4}$$

This method is one of the simplest method to find a linear regression function that can estimate an independent from a dependent variable, to solve that we calculate only two parameters based on the angular coefficient and on the intercept.
After this short demonstration it's easy to understand that the complexity of the regression problems raise a lot with the increment of the number of parameters. The natural extension of the linear regression is the general regression, that change the type regression function using logarithms, exponential, parabolic (ex. $\hat{y_i}(x_i) = a(x_i)^2 + bx_i + c$), cubic and many others function's types.
In the following pages we show how this technique is the starting point for the modern **ANN** algorithms.

3 Artificial Neural Network (ANN)

The Artificial neural network (ANN) are algorithms modelled like the animal brain, in biology and medicine the brain can be think like a set of neurons (nodes) and synapses (connections). Each of these objects can be mathematically represented with a function like that which can see in figure 2 and 3.

The purpose of the ANN is to solve a complex regression or classification problem, following the fig 2 and fig 3 example we describes how a simple ANN algorithm works.

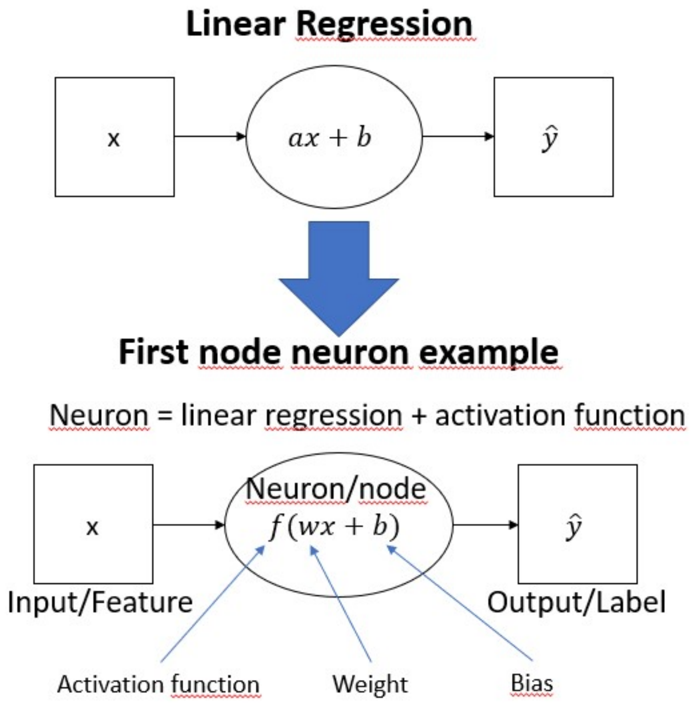


Figure 2: example of simple neuron/node and synapses/connection.

We start from linear regression to solve a problem described by points:

$$x_i, y_i \in R, i = 1, ..., N_{data}$$

where x_i are the N independent input variables and y_i are the N dependent output variables. We want to find the "best" function for estimate the y_i values:

$$\hat{y_i}(x_i) = ax_i + b$$

to do this one should use a function like that in eq: 1:

$$\sum_{i=0}^{N} (\hat{y}(x_i) - y_i)^2$$

these type of functions in ANN are called **Loss (or Cost) function**[1] representing the difference between the empirical/target output value and the estimated/calculated output value for y. The objective is to minimize the loss function, in this way the function $\hat{y}(x_i)$ will be very similar to the points y_i distribution. We listed some type of loss function with theirs common use in tab: 1. These are the same steps and consideration made in 2.
The first issue of linear regression is that it's not enough because lack of **non-linearity**. Following the fig 2 the ANN basic idea is to add a **non-linearity** component by inserting a function called **activation function** (f in 2). These functions are a sort of **switches** function that turns ON or OFF the node depending on the input. We discuss in detail these type of functions in section 4. Other important things in the figure are: the input quantities that are called **features**: the characteristic properties of the data[2], the output quantities in general are called the **labels**, while for that regard the regression function eq 2 the angular coefficients are in general called **weights** and the intercept terms are called **bias**. In general the algorithm search the best set of weights that minimize the loss function

[1]In literature sometimes there are small difference between Loss and Cost function. In our case we use the two terms as synonyms

[2]the name features derives from Machine Learning where the features are human chosen, in ANN the name remain the same but the features are extracted by the AI network

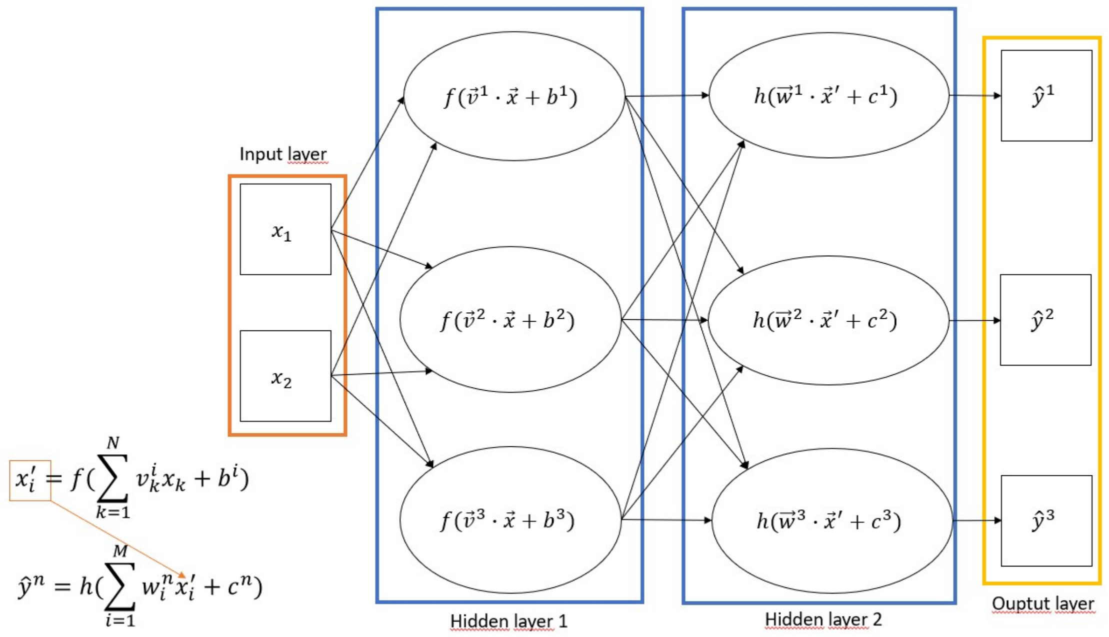

Input layer
x_1
x_2
$f(\vec{v}^1 \cdot \vec{x} + b^1)$
$f(\vec{v}^2 \cdot \vec{x} + b^2)$
$f(\vec{v}^3 \cdot \vec{x} + b^3)$
$h(\vec{w}^1 \cdot \vec{x}' + c^1)$
$h(\vec{w}^2 \cdot \vec{x}' + c^2)$
$h(\vec{w}^3 \cdot \vec{x}' + c^3)$
$\hat{y}^1$
$\hat{y}^2$
$\hat{y}^3$
$x_i' = f(\sum_{k=1}^{N} v_k^i x_k + b^i)$
$\hat{y}^n = h(\sum_{i=1}^{M} w_i^n x_i' + c^n)$
Hidden layer 1
Hidden layer 2
Ouptut layer

Figure 3: An example of NN with 2 features input layer, 2 hidden layers with 3 node and an output layer with 3 labels

while the bias is a set of constant values that are used for increasing the network generalization. All the previous consideration are valid in general for every node/neuron connection. The second "improvement" respect linear regression is that the nodes can be combined in a complex **Networks** and consequentially the scalar values used for the weights, biases, inputs and ouputs become vector values. A last consideration about these systems is that the process of training is iterative, for each iteration which is called **epoch**, the network weights are re-calculated randomizing the input dataset.

In figure: 3 we show how a simple network can appear. In general is very difficult to write all the network's equation: in our example, with 2 hidden layers and 3 nodes, we have arrived at the function by combining 2 sum terms with 6 vectors. In a common **deep learning** algorithm the network can have even 50 layers, for that reason is the ANN algorithm which generates the model. In small words the mathematical analytical function is so complicated that cannot be calculated by a human, instead one have a black box algorithm on which it can intervene on the inputs data and hyperparameters to estimate the outputs.
To **tune** the network [3], the designer, can operates only on the input data and on the **hyperparameters** of the network. The hyperparameters are all the parameters that one can use to build the network, for example the two main ANN parameters are the number of layers and number of nodes.

3.1 LOSS function type

There are many types of loss functions, each of them is used for different kind of problem; we listed some of the most used loss functions with theirs typical use in table: 1.

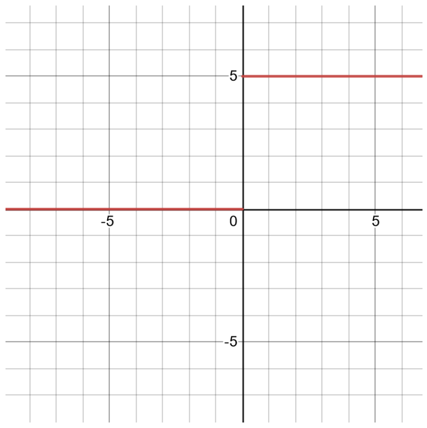
4 Activation functions

The activation functions are used to **activate/deactivate** the nodes and to introduce the **non-linearity**. The most simplest function that one can imagine is the binary step function (fig: 4 left) but, this function is not very useful because generates a linear response, the output is multiplied for 2 fixed values level and the derivative is 0. This function can be used in the last layer of the binary classification network but in general the sigmoid function is preferred to this one for the same purpose.

[3]in ANN field, tune means: search the best value for the hyperparameters

loss function	equation	typical use
Mean squared error	$L(y_i, \hat{y}_i) = \frac{1}{N} \sum_{i=0}^{N} (y_i - \hat{y}_i)^2$	Regression
Mean squared logarithmic error	$L(y_i, \hat{y}_i) = \frac{1}{N} (\log(y_i + 1) - \log(\hat{y}_i + 1))^2$	Regression
Mean absolute error	$L(y_i, \hat{y}_i) = \frac{1}{N} \sum_{i=0}^{N} \lvert y_i - \hat{y}_i \rvert$	Regression
Binary cross-entropy	$L(y_i, \hat{y}_i) = -\frac{1}{N} \sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)$	Binary classification
Squared Hinge	$L(y_i, \hat{y}_i) = \sum_{i=0}^{N} (max(0, 1 - y_i \hat{y}_i)^2)$	Binary classification
MultiClass Cross Entropy	$L(y_i, \hat{y}_i) = -\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i)$	Multiclass classification

Table 1: Loss functions example and general uses



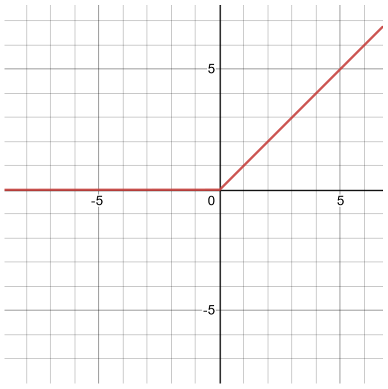


Figure 4: Left: Binary step function. Right: ReLU function

The most used activation function (fig: 4 right) is the **rectified linear unit (ReLU)**, it's easy to see that the response is not linear because in the active part the y is proportional to the x coordinate. The activation functions are many each one with different benefits and drawback, each is used for different situation. Generally the function type depending on the task and on the type of network; the used function are the same for all the nodes of the same layer, for the output layer the function is chosen according to the type of problem (regression, binary or multiclass classification), for all the hidden layers, it should be used the same activation function, the most used is the ReLU. Another consideration for the activation functions is that they must be **derivable at first order** due to the use of **GD** technique that we discuss in 5. During the application of the GD the activation function derivative is calculated for each node (and weight) in order to update the weights; it can happen that the chain derivative can **exploding** (tends to infinity) or **vanishing** (tends to 0), both of these cases are problematic for the ANN model generation and there are various solution to avoid that which we will not deal with in this text. We listed the most used activation functions and their common use in table 2. Each of that has a common use, in theory is not forbidden use the function in different situation.

These are only the most common activation functions, the ANN field is in constantly evolving, moreover the mathematics formal demonstration are very complicated for the complexity of the problem and many times the functions are tested with a more experimental method. An example of a more experimental approach can be seen in the ReLU: we say that the activation functions must be differentiable and this implies the **continuity**, the ReLU is not continues in 0 but it's the most used function in DL. The reason is that when the algorithm calculate the ReLU in the NN systems it's extremely rare have a "perfect" 0 number, most likely the value will be a very small tiny quantity close to 0.

Activ.function	Equation	Derivative	Common use
Linear	x	1	Regression Output
Binary step	$\begin{cases} 0\ for\ x < 0 \\ 1\ for\ x \geq 0 \end{cases}$	$\begin{cases} 0\ for\ x \neq 0 \\ undef.\ for\ x = 0 \end{cases}$	Binary classification
Softmax	$s_i = \frac{e^{z_i}}{\sum_{l=1}^{K} e^{z_l}}\ for\ \vec{z}$	$\frac{\partial z_i}{\partial z_j} = \begin{cases} 1\ for\ i = j \\ 0\ other \end{cases}$	Multiclass classification output
Sigmoid function	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma(x)(1-\sigma(x))$	Binary/Multilabel out e RNN hidden
Hyperbolic Tangent	$tanh(x)$	$1 - tanh^2(x)$	Recurrent NN Hidden
ReLU	$y = \begin{cases} 0\ for\ x \leq 0 \\ x\ for\ x > 0 \end{cases}$	$\begin{cases} 0\ for\ x < 0 \\ 1\ for\ x > 0 \\ undef.\ for\ x = 0 \end{cases}$	CNN e Multilayer Perceptron Hidden

Table 2: Activation functions and theirs common use

5 Gradient Descent (GD)

The **gradient descent** is a techniques to calculate a **local minimum** (global if the function is convex) of a **differentiable** function introduced by Cauchy in 1847 [1], in ANN GD is used to search a minimum for the loss function.

Cauchy idea was to moving step-by-step through the function following the negative towards of the gradient to reach the local minimum.

Term k+1 can be written:

$$x_{k+1} = x_k - n_k \nabla_{f_k}(x_k) \tag{5}$$

for ANN problem $f(x)$ is a loss and can be written with a sum of function:

$$f(x) = \sum_{i=1}^{n} f_i(x)$$

The gradient linear operator can be exchanged with sum:

$$x_{k+1} = x_k - n_k \sum_{i=1}^{n} \nabla f_i(x_k) \tag{6}$$

where n_k is called the **learning rate (lr)**.

It can be notice that in 6.1 appear a **summation operator**, in ANN science the GD that use all terms of the summation is called **batch GD**.
For deduce how many operations we need for upgrading one GD we can write the equation for the update of a single parameter k (par_k) during the GD:

$$par_{k+1} = \frac{d}{d(par_k)}[Loss(y_k - \hat{y_k})] \tag{7}$$

For a simple approximation we replace the loss with the **mean squared error** function and solve the derivative operation for a **linear regression system**:

$$par_{k+1} = \sum_{i=1}^{N} (y_i - \hat{y_i}) \frac{d\hat{y_i}}{dpar_k} \tag{8}$$

where $\hat{y_i} = ax_i + b$ with a and b parameters. For parameter a:

$$par_{a_{k+1}} = \sum_{i=1}^{N} (a_k x + b_k - y_k) * (a_k)$$

and for b:

$$par_{b_{k+1}} = \sum_{i=1}^{N}(a_k x + b_k - y_k)$$

In this very easy example we can see that the sum is iterated on data and parameters so we have number of operations ($n_o p$):

$$n_{op} = n_{par} * n_{data} \tag{9}$$

Looking the simple example with the linear regression function it easy conclude that for a more complex Loss function and increasing the number of data the number of operations can easy arrive at very large quantity for each gradient updating, this can be prohibitive for the computational power even for the modern architecture. This kind of problems in literature are called **finite sum problem**. Many solution have been proposed to reduce the computational requirement of the GD, we will examine some of that in sec: 6.

6 GD variants

Considering that the GD eq: 6.1 can be computationally expensive as the number of parameters and data increases many solution are proposed and other are under investigation. The GD variants are many and under study, the problem regard the compromise between the number of computation (time) and the efficiency for the minimum estimation, in general more one wants be precise more he approaches to the **batch GD** case (the case with all the summation terms). Following the overview in article: [2] we present some gd variant used to reach a compromise between the precision and the computation time.

6.1 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is a simple technique used to simplify reduce the number of operations in the GD, the solution consist in fix randomly (stochastic) f_i to simplify the summation in each iteration. Restarting from the GD equation 6.1:

$$x_{k+1} = x_k - n_k \sum_{i=1}^{n} \nabla f_i(x_k)$$

and using a fixed i in the sum we obtain:

$$i(k) \in 1, 2, ...n \, x_{k+1} = x_k - n_k \nabla f_{i(k)}(x_k) \tag{10}$$

This simplification that counterintuitively works good, reduce the number of computation of a factor $n_d ata$.

The differences between the GD and SGD are represented in fig: 5.

Like show in figure the SGD can be used to approximate good the minimum of the function, the major problem is the step choice, in simple one should find a compromise about the step dimension (**learning rate(lr)**. There are two considerations about that: the first one is that SGD right fig: 5 is unstable, this instability depend directly on the lr dimension. More the lr is smaller more the SGD is stable and reach better the minimum but, smaller lr means more iterations to reach the minimum and this means more elaboration time. The second consideration is due to the fact that the GD/SGD allow to find a **global minimum** for a **convex** function. For a not convex function the descent can remain trapped in a local minimum point, an example is showed in fig: 6. Moreover observing the figure it's easy understand that if a big lr step is selected the SGD can jump out of the local minimum. In general this technique is prefered than GD when the dataset sample is big.

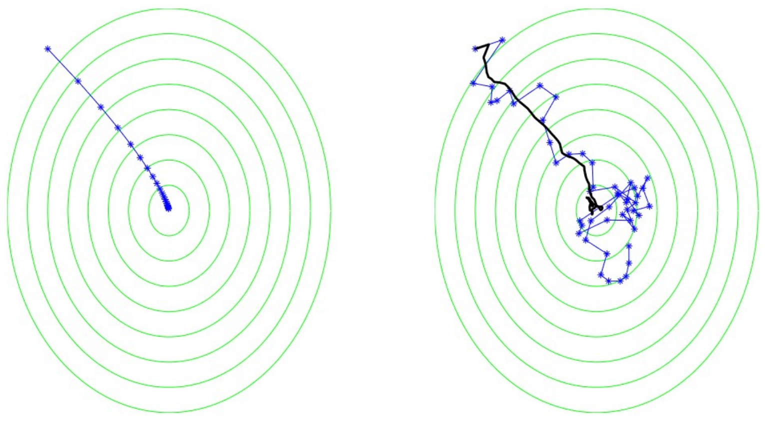


Figure 5: On left a gradient descent example on right sgd example from [3].

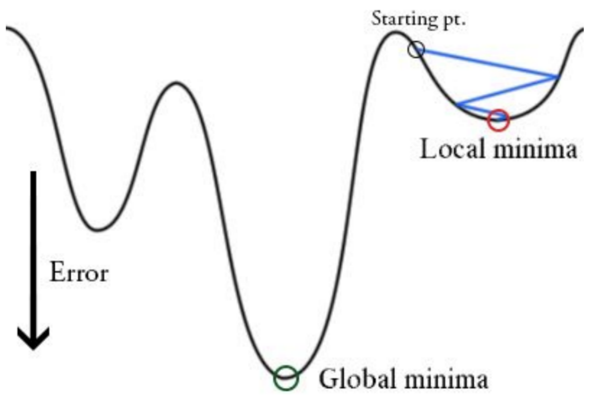


Figure 6: example of not convex SGD works

6.2 Mini-batch Gradient Descent

Another solution is the **mini-batch Gradient Descent** , that is the middle compromise between the GD efficiency and SGD robustness. In this technique, for each epoch, a different small set (mini-batch) of data are taken to calculate the gradient updating. An issue of this technique is that introduce an hyperparameter for the mini-batch size that should be tuned during the network design. This technique, actually, is the most used technique for the ANN training[2]. [4]

7 Back Propagation (BP)

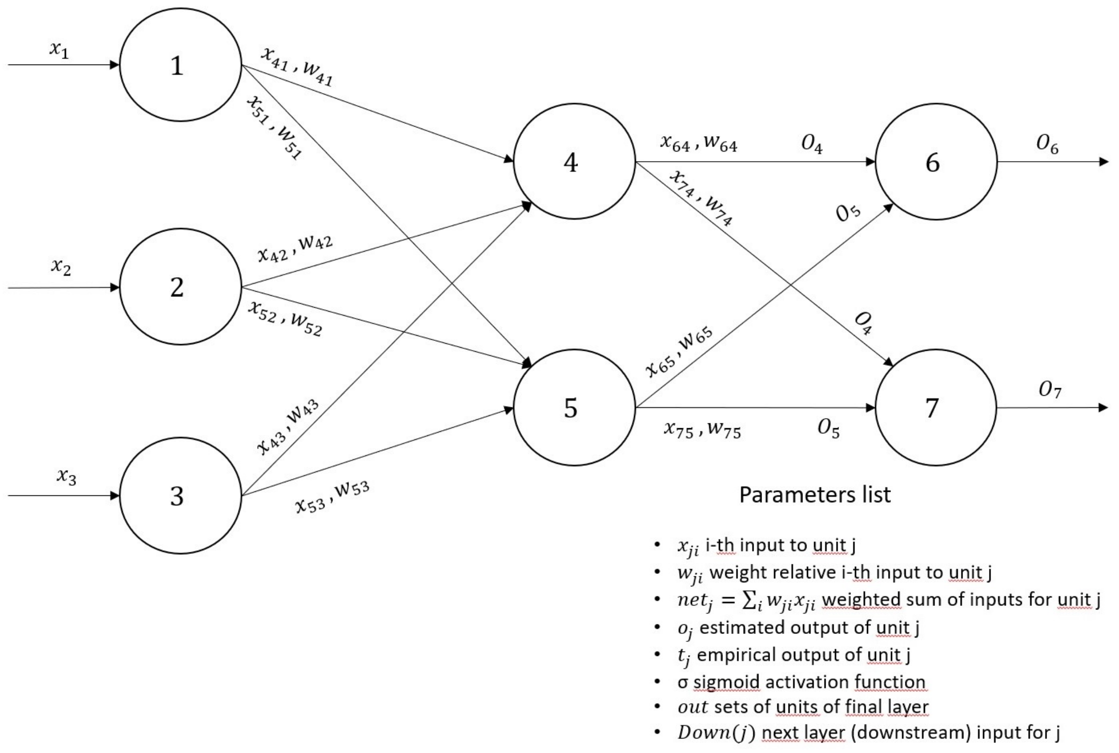
Back Propagation (BP) is a technique to find how the total loss propagate (back) through the neural network in order to calculate how much is the loss in each node of the network. In that way one can know how the weights should be updated to minimizing the loss.

Back Propagation (BP) is a diffuse and efficient algorithm used for the training and optimize the model weights of **supervised** and **feedforward** NN. The BP using **Stochastic Gradient Descent (SGD)** technique to calculates/approximate the gradient of loss function respect to weights in this way it is possible to minimize the loss function. The algorithm can be generalized for all the not-feedforward NN [4]. The "back" in the name is due to the fact that calculation of the gradient takes place backwards through the network, starting the calculation of the gradient of the last layer of weights up to the gradient of the first layer of weights.

[4]the term SGD is usually used also when mini-batches are used[2]

7.1 Example of Back Propagation (BP)

We show a simple demonstration of back propagation algorithm calculation to better understand how it works. We use the figure 7 to extrapolate an example of back propagation algorithm.



Parameters list

- x_{ji} i-th input to unit j
- w_{ji} weight relative i-th input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ weighted sum of inputs for unit j
- o_j estimated output of unit j
- t_j empirical output of unit j
- σ sigmoid activation function
- out sets of units of final layer
- $Down(j)$ next layer (downstream) input for j

Figure 7: ANN example used to calculate the back propagation

The objective is to find how and how much the algorithm should update the weight in each cycle:

$$w_{ji} \leftarrow w_{ji} + \Delta wji \tag{11}$$

the quantity we want to find is Δwji where the indexes i are the input in the node and j the node where the input arrive. The Δw_{ji} can be written with the partial derivative of the total loss function E_d of a training example d:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \tag{12}$$

Using the mean squared loss function:

$$E_d(\vec{w}) = \frac{1}{2} \sum_{k \in output} (t_k - o_k)^2 \tag{13}$$

where $\boldsymbol{\eta}$ is the **learning rate**: the value that is related to the steps of **GD**/**SGD**, t_k are the target/empirical data, o_k are the estimated/output data.

For each node one can write the quantity

$$net_j = \sum_i w_{ji} x_{ji} + bj \tag{14}$$

where w_{ji} are the weight, x_{ji} are the input i for the node j and b_j are the node biases. Now using the chain rule we can write:

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} * \frac{\partial net_j}{\partial w_{ji}} \tag{15}$$

we solve the second term of eq:15, we can delete the sum term because the derivative for all terms except w_{ji} is 0:

$$\frac{\partial net_j}{\partial w_{ji}} = \frac{\partial(\sum_i w_{ji}x_{ji} + b_j)}{\partial w_{ji}} = x_{ji} = o_i \tag{16}$$

Were in the last passage we rewrite the input i for node j as the output of the i (previous) node.

For the first term of the 16 we should distinguish between two cases: first when j is an output unit for the network and second when j is an internal unit. Starting with j as an output unit:

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} * \frac{\partial o_j}{\partial net_j} \tag{17}$$

Solving the first terms of eq 17:

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial(\frac{1}{2}\sum_k (t_k - o_k)^2)}{\partial o_j} \xrightarrow[j \neq k = 0]{\text{j=k}} -(t_j - o_j)$$

(18)

.for second term of eq 17 ($\frac{\partial o_j}{\partial net_j}$) we use a **sigmoid** activation function (σ_{act}):

$$o_j = \sigma_{act}(net_j) \tag{19}$$

Remembering the sigmoid property:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) * (1 - \sigma(x)) \tag{20}$$

combining the second term with 20:

$$\frac{\partial o_j}{\partial net_j} = o_j * (1 - o_j) \tag{21}$$

and finally for the first case (j output unit):

$$\Delta w_{ji} = \eta(t_j - o_j)(oj(1 - o_j))o_i \tag{22}$$

For that regarding the second case (j internal node) instead we have:

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Down(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \tag{23}$$

Were we introduce new index k that represent the node of the next layer respect j and the notation $k \in Down(j)$ that is the summation of the input of the subsequent layer (backpropagation). We found the first term before in eq: 17 with j instead k.

$$\frac{\partial E_d}{\partial net_k} = -(t_k - o_k) \tag{24}$$

We solve the second part of eq 23 ($\frac{\partial net_k}{\partial net_j}$):

$$\frac{\partial net_k}{\partial net_j} = \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \tag{25}$$

we can not consider the summation for the second part of eq 25 because does not contain k, furthermore we can observe that we can find this before in eq 21:

$$\frac{\partial o_j}{\partial net_j} = o_j(1 - o_j) \tag{26}$$

and for the first part of eq 25:

$$\sum_{k\in Down} \frac{\partial net_k}{\partial o_j} = \sum_{k\in Down} \frac{\partial (w_{kj}x_{kj})}{\partial o_j} \tag{27}$$

Now it's easy observe that x_{kj} is o_j (for example from fig:7 x_{64} is o_4). Making the substitution:

$$\sum_{k\in Down} \frac{\partial (w_{kj}x_{kj})}{\partial o_j} = \sum_{k\in Down} \frac{\partial (w_{kj}o_j)}{\partial o_j} = \sum_{k\in Down} w_{kj} \tag{28}$$

and in the end for an hidden layer node we have:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{net_j} x_{ji} = \eta o_j(1-o_j) \sum_{k\in Down(j)} (t_k - o_k) w_{kj} x_{ji} \tag{29}$$

And for last we rewrite the eq 29 for an hidden layer:

$$\Delta w_{ji} = \eta o_j(1-o_j) \sum_{k\in Down(j)} (t_k - o_k) w_{kj} x_{ji} \tag{30}$$

and reporting the eq: 31 for the output layer to have the two equations close:

$$\Delta w_{ji} = \eta(t_j - o_j)(oj(1-o_j))x_{ji} \tag{31}$$

We show how basically work the back-propagation of the error in ANN. From eq: 30 it's easy to see the back-propagation from the output towards the input layer ($k \in Down$). We use many times the derivative operator and in particular for the equation 7.1 one can understand the reason because the activation function must be derivable to the first order.

8 Conclusion

Starting from a linear regression we showed some of the statistical techniques used in ANN to find (iterate) the minimum of a loss function. These techniques are the basis of the modern ANN algorithms which are spreading over the world in parallel with the improvement of the hardware technologies. These algorithms are capable to deal with a big quantity of data without the human direct intervention, for this reason the ANN are becoming very important and are studied in various research fields

References

References

[1] C. Lemaréchal. Cauchy and the gradient method. pages 251–254.

[2] Sebastian Ruder. An overview of gradient descent optimization algorithms.

[3] S. Shalev-Shwartz and S. Ben-David. Understanding machine learning from theory to algorithms.

[4] Igor Aleksander. *Backpropagation in non-feedforward networks.* 2003.