Introduction to Neural Networks

ENGR110, Project 3 reading material

25 June 2019

Contents

1 Introduction

There are several possible approaches to creating Artificial Intelligence (if such thing is possible).

- Logic and rule-based: knowledge is represented as rules. Computers use logic reasoning to check if new statements are FALSE/TRUE using these rules. Hope was that if computer is loaded with enough **propositions** some non-obvious **inference** can happen.
- Machine Learning: Takes **from bottom to the top** approach: make a simplified model of the biological brain and throw a lot of data into it. Hope is that eventually it will figure things out and structure itself as a real brain. "Brain" can be implemented either as hardware of software.

We will focus on second approach - it seems to be more successful lately.

Model (greatly simplified, of course) of the brain is **Neural Network**.

Neural Networks are used for classification of data and making predictions. Same can be achieved by writing programs using some clever algorithms, but creating Neural Network is different from writing standard program. You (or more likely somebody else) write the NN code once, and rather simple code it is. As a bare-bone program, Neural Network can not do anything, it will just provide rubbish output.

To make NN useful this code (or hardware) should be **trained**, and it is by far most time consuming and critical part of creating the neural net. What is training?
To train the net you provide input data: it can be a picture (set of pixels, E101 anyone ?), a sound (waveform), or practically any set of data as long as there is some feature (pattern, if you wish) in this data. Key here is that you provide right answer for this input data.

Cutting it to the very basics, Neural Net is huge equation of specific structure. This equation contains many coefficients. It looks like,

$$Output = function(Input_1 \cdot Coefficient_1 + Input_2 \cdot Coefficient_2) \quad (1)$$

, where $function()$ is some non-linear function. Or same calculation can be implemented as a hardware.
Training is the process of finding values of $Coefficient_i$ so that $Output$ is right for certain $Input$. If input, for example, is the picture (i.e. set of pixels) then output can be what is feature of this picture - "look, squirrel!" (or a cow). Or neural net can decide that this sound waveform was letter "a", there is no tumor in MRI scan or that this particular customer's credit limit can be safely increased.

2 Structure of Neural Networks

NNs are made out of neurons - lots and lots of them. Still much less though than any reasonable animal's brain. Latest Intel neural chip contains 130,000 neurons [1], what is more than in brain of the lobster but less then in brain of fruit fly [2].

In original state outputs of all neurons are low (neuron is dormant) as shown in Fig. 1. Some neuron outputs are output of whole Neural Network. Each output of the net is associated

with some feature/pattern present in input data. If all outputs are low (close to 0) then no features/patterns were detected in input data.

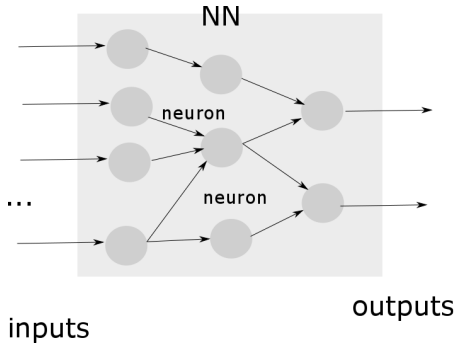


Figure 1: Initial NN state

When Neural Network reaches conclusion (it was detected that input data follow certain pattern) then some of the neurons **fire** - their output goes high, highlighted outputs in Fig. 2. One of the Neural Net

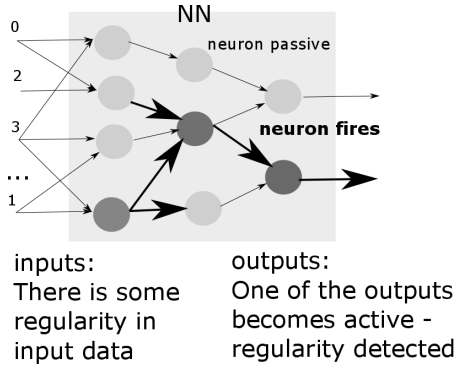


Figure 2: Some neurons are firing - trained NN

Neuron as used in Neural Networks closely resembles biological brain neurons. Of course, it happens that real brain neuron is much more complex than we realized before and more and more things are discovered. One thing is certain though, neuron takes signals from other neurons, analyzes these signals and produces output signal depending upon values of the input signals. You can ask how that fits into current most widely used computer architecture, i.e. von Newman. Short answer is that it does not fit well and there are a lot of efforts underway to create customized hardware, specifically designed to imitate the workings of biological neurons.

2.1 Neuron equation

How can we express what single neuron does in mathematical form so we can actually program it? It is not at all complicated equation:

$$y = f(b + x_0 \cdot w_0 + x_1 \cdot w_1 + ... + x_n \cdot w_n) \tag{2}$$

, where $f()$ is called **activation function**.

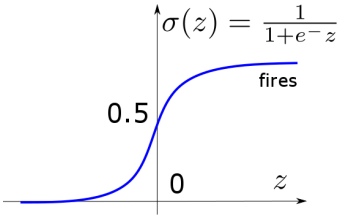


Figure 3: Sigmoid activation function

Non-linear activation function is be used so decision can be made. Output of the activation function is mostly 0 (neuron dormant) or 1 (neuron fires). If output is close to 0.5 - neuron did not come to decision. There are many possible activation functions used , we used simplest one called **sigmoid**, which is described by the equation

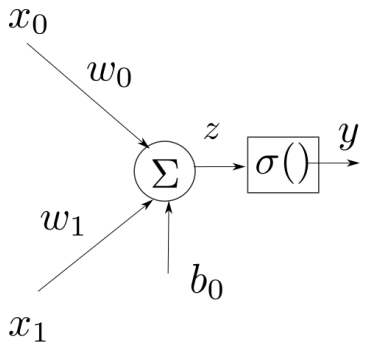
$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

and is shown in Fig. 3.

3 Simple NN example

3.1 Dataset

We are not going to try to simulate human brain, not yet... Let us start with something much simpler - logic gate. It got two inputs, x_0 and x_1, incoming values for either are in range between 0 and 1, and one output, call it y.



$$z = b + w_0 \cdot x_0 + w_1 \cdot x_1 \tag{4}$$

$$y = \sigma(z) \tag{5}$$

$$\sigma(z) = \frac{1}{1 + e^{-x}} \tag{6}$$

Figure 4: Single neuron

As there are 2 binary inputs (lets call them x_0 and x_1) there is 4 possible combinations:

x_0	x_1
0	0
0	1
1	0
1	1

This table exhausts all possible combinations of input data.
Now we want this logic gate to provide a binary output. For no particular reason we decide that output should be 1 when $x_1 = 0$ and $x_2 = 0$ and 0 for all other combinations of the inputs.

x_0	x_1	t
0	0	1
0	1	0
1	0	0
1	1	0

, where t stands for **truth** - known answer.

That is an example of **training dataset** - and it is simple one.
We can draw a picture of this dataset, see Fig.5.

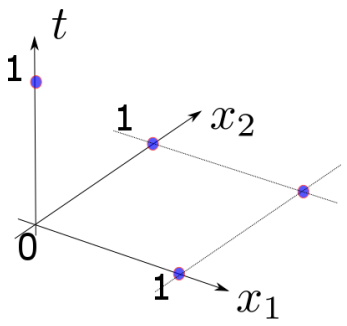


Figure 5: Output of the neuron as it should be

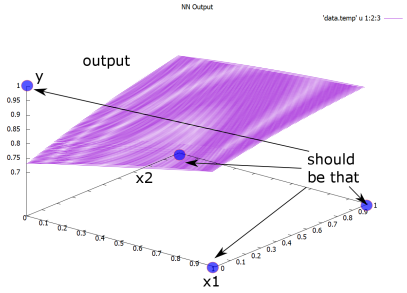


Figure 6: Output for b=1 w1=1 w2=1

3.2 NN output

Neuron provides output for any values of the inputs x_0 and x_1. We know answers only for 4 points. For points in-between neural net has to make the decision.

This short code

```
double NN::GetOutput(double x0, double x1){
  double z = b + w0 * x0 + w1 * x1;
  double y = activation(z);
  return y;
}
```

produces for values $b = 1.0 w0 = 1.1 w1 = 1.0$ output y is shown at Fig.6.

As you can see, output y is not even close to what is should be. At the point ($x1 = 0$,$x2 = 0$) output should be 1 (see first row of training dataset table). It is about 0.73 instead. How can we find values of b, $w1$ and $w2$ so that formula

$$y = \sigma(b + w_1 \cdot x_1 + w_2 \cdot x_2) \tag{7}$$

produces output values we want for given values of the inputs x_1 and x_2, i.e. equal to t?

3.3 NN output errors

Let us have a look at neuron output again. As we can see - there are **errors**, see Fig.8. Error in this case means difference between what output y is and what it should be - t.

Error for point ($x_1 = 0$,$x_2 = 0$) ϵ_{00}, for example, is:

$$\epsilon_{0,0} = y(0,0) - t(0,0) = sig(1.0 + 1.1 \cdot 0 + 1.0 \cdot 0) - 1 = sig(1) - 1 = \tag{8}$$

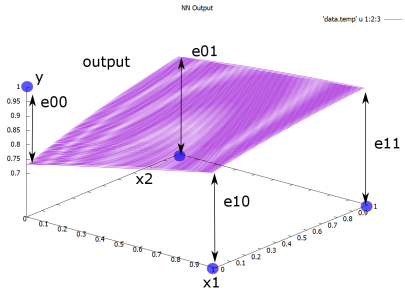


Figure 7: Errors

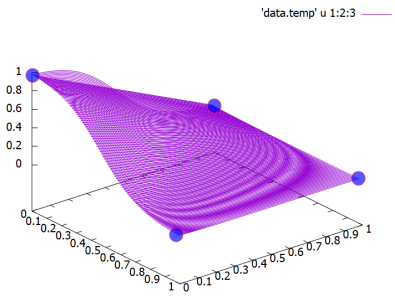


Figure 8: After training - errors for all training entries are small output fits to required data points

If we find values of bias b and weights such that sum of errors for all entries in training dataset is 0, if would mean that neural net produced outputs

$$\epsilon_{tot} = \epsilon_{00}^2 + \epsilon_{01}^2 + \epsilon_{10}^2 + \epsilon_{11}^2 \tag{9}$$

Note that errors are squared. Reason is that values of errors can be negative as well as positive. Then if we used:

$$\epsilon_{tot} = \epsilon_{00} + \epsilon_{01} + \epsilon_{10} + \epsilon_{11} \tag{10}$$

for total error calculation errors can cancel each other, even though each of them is big.

If errors are squared then sum of all eroors will be zero only if each of the errors is zero.

3.4 Training NN, minimizing errors

How can we find values of biases and weights so that error ϵ_{tot} is small?

3.4.1 Global search

Search through all possible combinations - while it is guaranteed that best solution will be found it is very time consuming and usually not used.
There are 3 parameters of single neuron: b, w_0 and w_1. To run three nested cycles

```
for (   b = b_min; b < b_max; b = b +db)
 for (   w0 = w0_min; w0 < w0_max ; w0=w0+dw)
  for (   w0 = w0_min; w0 < w0_max ; w0=w0+dw){
    // calculate output and error for these value
  }
```

will take a long time. It is not too bad for three parameters with only three nested cycles, but any reasonable neural net will contain thousands of neurons and tuning this way will require thousand of nested cycles and will require very long time indeed.

3.4.2 Guided (by gradient) search.

Better (faster) way can be not check every possible value of neural net weights but modify weights trying to improve (reduce) the error all the time.

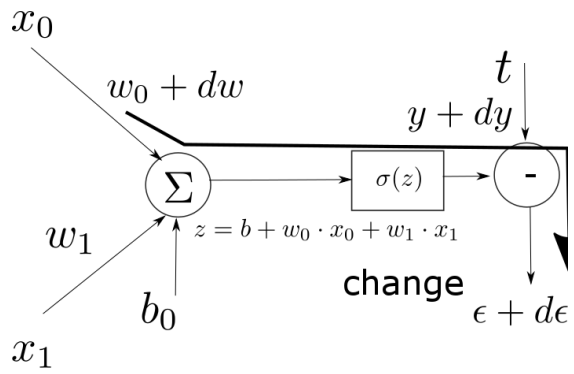


Figure 9: Change in one weight produces changes downstream of neural net

If one of the weights changes (shown in Figure 9 is change in w_0) it affects all downstream levels in neural net and results in change of the output y. For constant answer t (answers do not change) result is change in the error ϵ.
If error ϵ change makes it better (smaller) we better keep the change in w_0 and make it $w_0 + dw$.
If opposite is the case we better change w_0 into opposite direction, i.e. make it $w_0 - 2dw$.
Or, re-phrasing same in calculus terms, if derivative $\frac{d\epsilon}{dw_0}$ is positive - we want to reduce w_0 and another way around.

It is reasonable algorithm but chances of it really settling at value of w_0 providing minimum error are slim. There is a contradiction here: if we want to find minimum error fast the dw should be big, so we get to the minimum faster (in less steps), on another hand, if we want to approach "sweet spot" as close as possible and it requires small value of dw.

A reasonable thing to do here is to adjust dw - keep it big if the algorithm is far from the optimum and reduce it

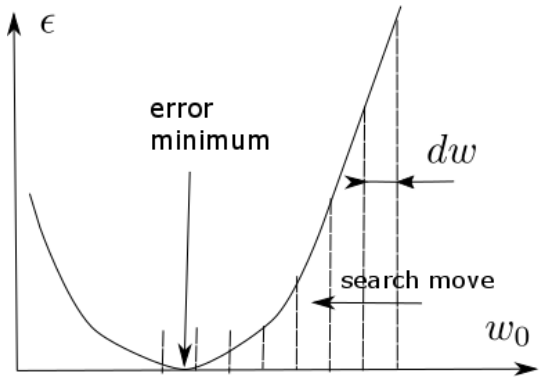


Figure 10: Moving towards minimum error

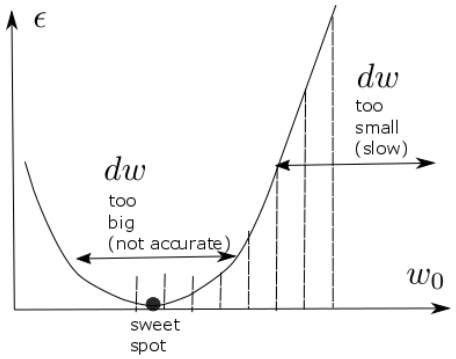


Figure 11:

$$w_0 = w_0 - \mu \frac{d\epsilon}{dw_0} \tag{11}$$

, where μ is **learning rate**.

3.4.3 Calculating gradient (Back-propagation)

Way to estimate derivatives as described in previous section takes a long time, particullary if number of biases and weights in neural net is big. And it usually is. There is a faster way which involves some calculus. Actually, it was discovery of this algorithm (sometime late 1970s) which made training of neural networks feasible.
We can write equation for error (difference between current outputs of the neuron and value from truth table) as:

$$e = (y - t)^2 \tag{12}$$

, where y is calculated as

$$y = \sigma(b + w_1 \cdot x_1 + w_2 \cdot x_2) \tag{13}$$

, i.e. y changes if bias or weights are changed, while t is the right answer from training dataset and would not change if bias or weights are changed.

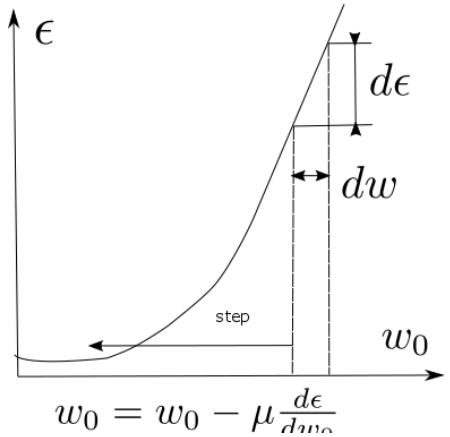


Figure 12:

For a start we want to tune bias b. We want to calculate how error e changes if b is changed, i.e. we want the derivative $\frac{de}{db}$.
Applying chain rule

$$\frac{de}{db} = \frac{(y-t)^2}{db} = 2 \cdot (y-t)\frac{d(y-t)}{db} \tag{14}$$

Now, if we change b then y will change too. However, t is not going to change, as it is constant which is picked from truth table. So we can write

$$\frac{de}{db} = 2 \cdot (y-t)\frac{d(y-t)}{db} = 2 \cdot (y-t)\frac{dy}{db} \tag{15}$$

Now we can pick y from $\sigma(b + w_1 \cdot x_1 + w_2 \cdot x_2)$

$$\frac{dy}{db} = \frac{d(\sigma(b + w_1 \cdot x_1 + w_2 \cdot x_2))}{db} \tag{16}$$

Chain rule again,

$$\frac{dy}{db} = \frac{d(\sigma(z))}{dz} \cdot \frac{d(b + w_1 \cdot x_1 + w_2 \cdot x_2)}{db} = \frac{d(\sigma(z))}{dz} \tag{17}$$

, where z is

$$z = b + w_1 \cdot x_1 + w_2 \cdot x_2 \tag{18}$$

current value at the input of activation function. $\sigma(z)$ is fixed function and so is $\frac{d(sig(z))}{dz}$. It can be derived once.
In very similar fashion we can derive

$$\frac{dy}{dw_1} = \frac{d(\sigma(z))}{dz} \cdot \frac{d(b + w_1 \cdot x_1 + w_2 \cdot x_2)}{dw_1} = \frac{d(\sigma(z))}{dz} \cdot x_1 \tag{19}$$

$$\frac{dy}{dw_2} = \frac{d(\sigma(z))}{dz} \cdot \frac{d(b + w_1 \cdot x_1 + w_2 \cdot x_2)}{dw_2} = \frac{d(\sigma(z))}{dz} \cdot x_2 \tag{20}$$

You can see that equation

$$\delta(z) = \frac{d(\sigma(z))}{dz} \tag{21}$$

Activation function $\sigma(z)$ derivative:

$$\frac{d}{dx}\sigma(x) = \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] = \frac{d}{dx}\left(1+\mathrm{e}^{-x}\right)^{-1} = -(1+e^{-x})^{-2}(-e^{-x}) =$$

$$\frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} = \frac{1}{1+e^{-x}} \cdot \frac{(1+e^{-x})-1}{1+e^{-x}} =$$

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

To summarize back-propagation algorithm for single-layer network:

1. For current values of b,w_0 and w_1 and inputs x_0, x_1 calculate:

   $$z = b + x_0 \cdot w_0 + x_1 \cdot w_1$$

2. Calculate $y = \sigma(z)$
3. Calculate error for the output: $\epsilon = y - t$
4. Calculate δ for this neuron:

   $$\delta = \sigma(z) \cdot (1 - \sigma(z)) \cdot (y - t)$$

   Use value of z calculated in 1, value of y from 2.
5. Calculate derivative of output error by weight

   $$\frac{d}{dw_i} error = \delta \cdot x_i$$

6. Calculate derivative of output error by bias $\frac{d}{db} error = \delta$

4 Limitation of single-layer network

x_0	x_1	t
0	0	1
0	1	0
1	0	0
1	1	0

and output y comes very close to required t for all points (x_0, x_1).
In-between these points neuron output still has some value, but it is not close to 1 (neuron fires) or 0 (neuron dormant). When neuron output y is 1 or 0, neuron is sure that decision was made. When y is somewhere in-between is is neuron's way of saying: "I don't know"

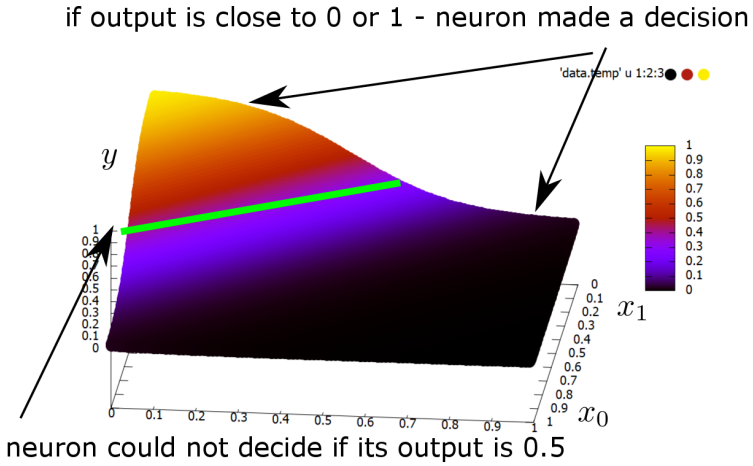


Figure 13: Output of tuned neuron

Neuron output $y = \sigma$ is about 0.5 when value of $z = b + w_0 \cdot x_0 + w_1 \cdot x_1$ is about 0. So neuron "not sure" zone is defined by this equation. If we have a closer look for fixed values of b, w_0 and w_1 it is equation of the line.

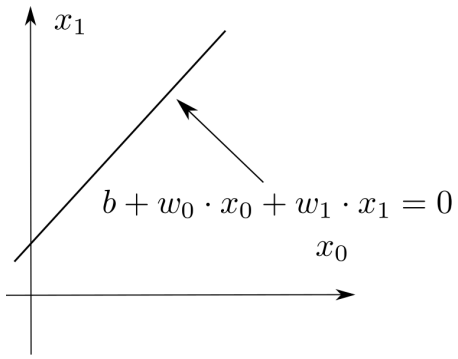


Figure 14: Decision boundary for single neuron

On one side of the line output of the neuron is 0 and it is 1 on another side. Let call it

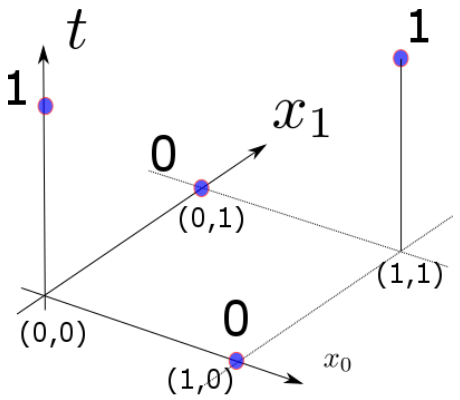


Figure 15: Single neuron can not be trained with this dataset

decision boundary. It means that one neuron can only data points by line.
There is a problem here. What if we want to train the neuron to work with such data set that data of different categories can not be separated by simple line, for example

x_0	x_1	t
0	0	1
0	1	0
1	0	0
1	1	1

There is no way to separate all 1s from all 0s here using only one line. Realization of that brought development of neural networks to dead stop for a while until walk-around was found. It is possible to make decision boundary of more intricate shape (composed out of multitude of straight lines) if we add more neurons to the the so that output of one neuron is an input for another. Lets see how it can be achieved.
If we have two neurons then each of them can be trained to form the decision boundary in shape of the line. Lines can be different for different neurons, like it is shown in Fig.17.

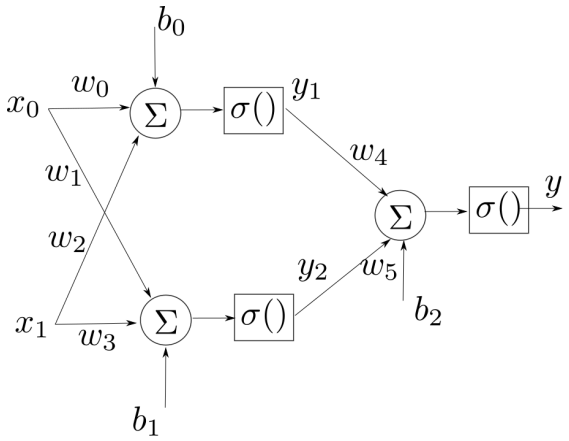


Figure 16: Two layers network

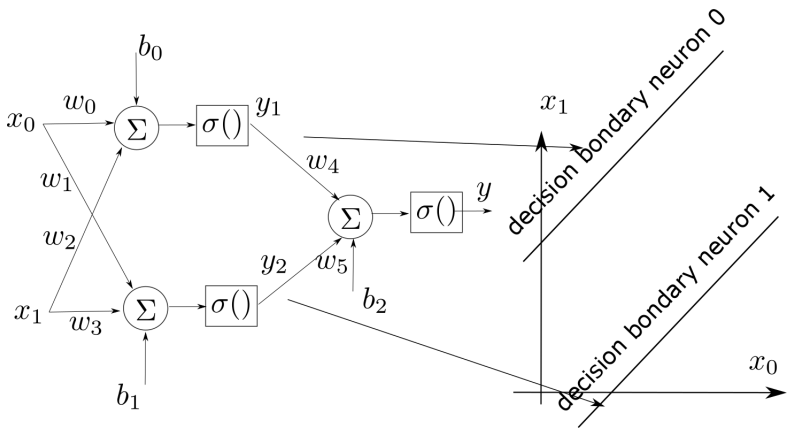


Figure 17: Each neuron can produce boundary line

Two neurons provide two outputs

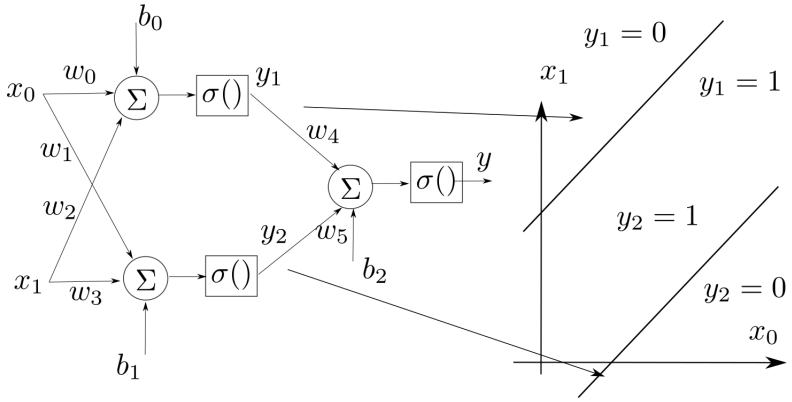


Figure 18: Outputs of two neurons to be combined

And so **deep learning** started. It is not deep in meaning, it is deep because neural net is many layers deep.

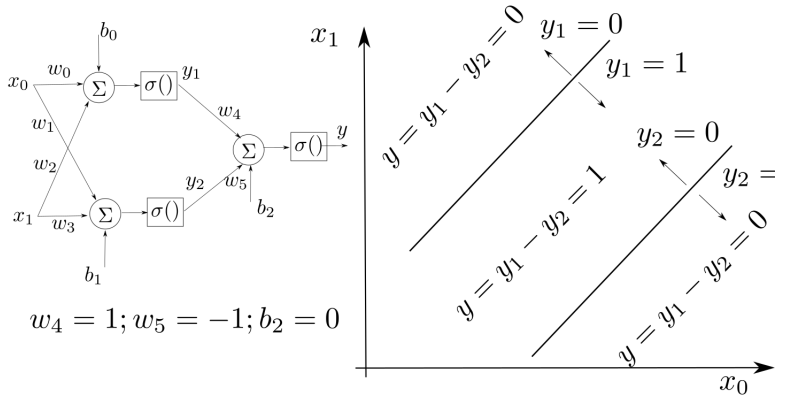


Figure 19: Combining outputs

4.1 Gradient calculation for multi-layer Neural Network

Multi-layered networks still have to be tuned(trained). Back-propagation is very fast method of calculating the gradient and it would be good to use for multi-layer networks, especially because number of the neurons is greater in multi-layers network.

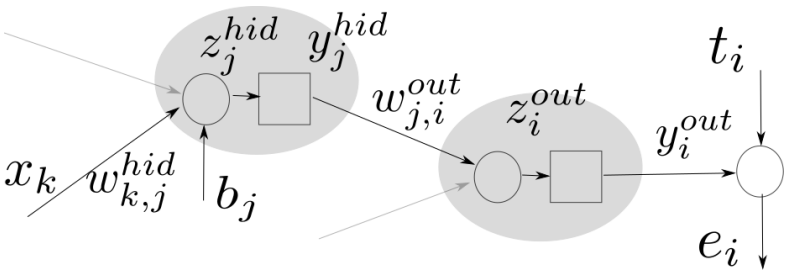


Figure 20: Simplified 2-layers netwok. Left neuron belongs to **hidden** layer. Right neuron - **output** layer

For two-layers network shown at Fig.20 assuming it has N_{out} outputs error is:

$$E = \sum_{i=0}^{N_{out}} (e_i)^2 = \sum_{i=0}^{N_{out}} (y_i - t_i)^2$$

We already have an expression for output layer neurons δ (see before):

$$\delta_i^{out} = \sigma(z_i) \cdot (1 - \sigma(z_i)) \cdot (y_i - t_i)$$

Applying chain rule to hidden layer neurons:

$$\delta_j^{hid} = (- \sum_{i=0}^{N_{out}} \delta_j^{out} \cdot w_{j,i}^{out}) \cdot \sigma(z_j^{hid}) \cdot (1 - \sigma(z_j^{hid}))$$

Derivatives of E by weights and biases of the hidden layer:

$$\frac{d}{dw_{k,j}^{hid}} E = \delta_j^{hid} \cdot x_k$$

and

$$\frac{d}{db_j^{hid}} E = \delta_j^{hid}$$

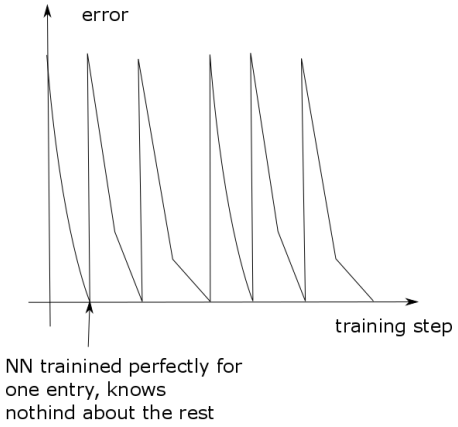


Figure 21:

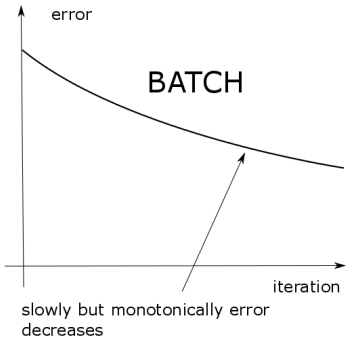
5 Training for big datasets

What NOT to do:
Pick one entry from training dataset and perform full tuning for this particular entry. Then pick next entry...
Your search will NEVER converge. Successful training is compromise between different entries in dataset. What works perfectly for one entry is totally wrong for others. If you use such approach your training starts all over again for each entry. When training we have to consider (somehow) whole training set.

There are two major methods to train and keep NN on track for whole dataset.

5.1 Batch training

When using **batch training** system keeps value of weights w and biases b while computing the error for whole dataset.



Batch training

- For whole dataset
  - Pick first/next row/entry from training dataset
  - Calculate error for this entry
  - Back-propagation
  - Accumulate δ
- From accumulated δ calculate $\frac{de}{dw}$ and $\frac{de}{dw}$
- Make one step in w and b
- Repeat if error is too big

Error (δ) is estimated and minimized for whole dataset so convergence is monotonous but each iteration takes long time.

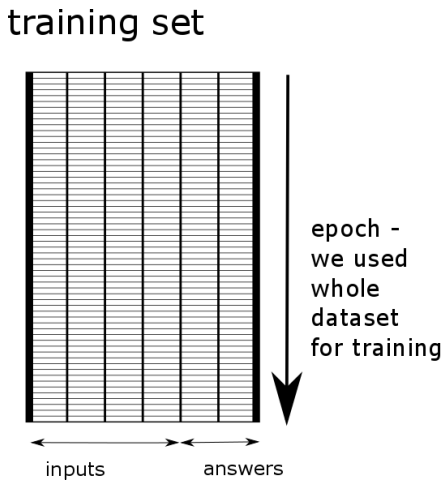
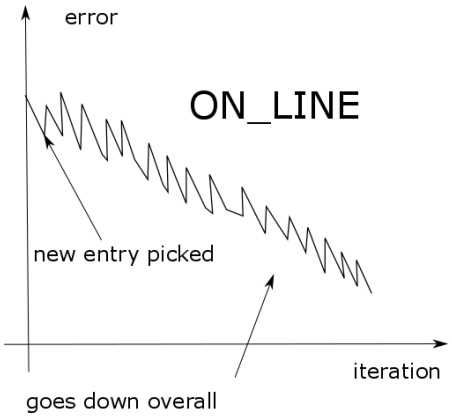


Figure 22: Epoch

5.2 On-line training

New entry from dataset is used for each step. Only one search step is made for the entry.



- Pick random entry from training dataset
- Calculate error for this entry
- Back-propagation
- Make one step in w and b
- Repeat if error is too big or not whole dataset had been used

With one search step completed next entry is picked, usually randomly. Chances are NN never seen this particular entry before and error will go up. Error is not monotonous but on average it decreases.

They both converge to the same minimum. Which is better is still subject for discussion.

When terminate the search/training? Certainly not before all entries in training dataset had been tried.

An epoch refers to one complete pass of the data through the system/algorithm - i.e. we looked through whole dataset.
An iteration refers to one pass of data after which gradients/loss are computed and backpropogation occurs.

An epoch refers to one complete pass of the data through the system/algorithm, whereas an iteration refers to one pass of data after which gradients/loss are computed and backpropogation occurs. When we use batch training, these terms are equivalent, however when we use minibatches we will have k iterations per epoch, where k is the ratio of the dataset size to the minibatch size (or the ceiling of this ratio in most cases but this is a mere technicality).

6 References

1. https://www.intel.com/content/www/us/en/research/neuromorphic-computing.html
2. https://en.wikipedia.org/wiki/List_of_animals_by_number_of_neurons
3. http://neuralnetworksanddeeplearning.com/ - very good introduction. Programming in Python
4. https://theclevermachine.wordpress.com/ - more mathematical
5. https://davidstutz.de/wordpress/wp-content/uploads/2014/03/seminar.pdf - goes beyond what we covered
6. http://page.mi.fu-berlin.de/rojas/neural/ - it is a big book
7. Artificial Intelligence: A Modern Approach (Prentice Hall Series in Artificial Intelligence) by Stuart Russell. "Must" read for anybody interested in AI. Covers much more then Neural Networks. Not free though.