ENGR110 T2 2018

Engineering Modeling and Design

Arthur Roberts

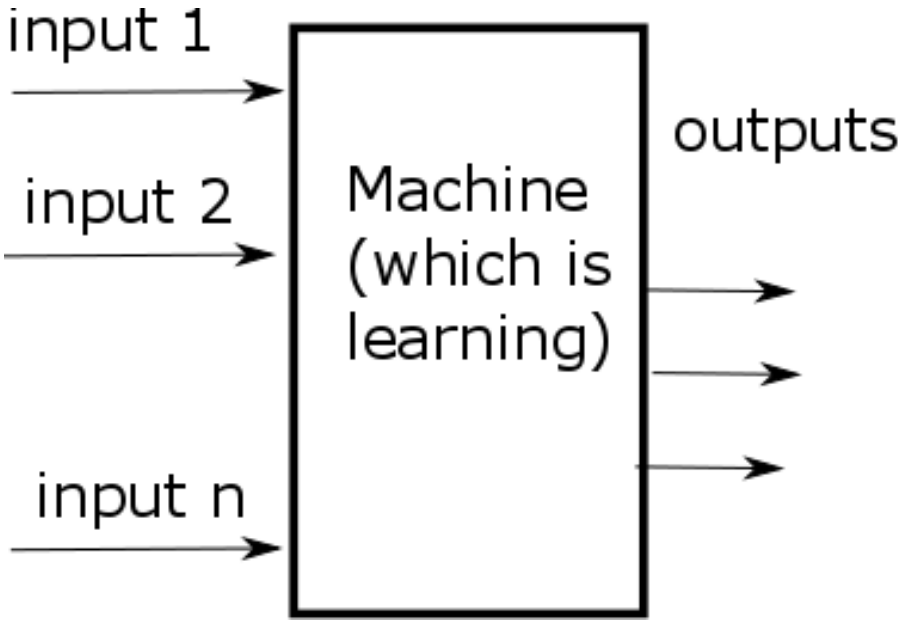
School of Engineering and Computer Science
Victoria University of Wellington

There are several approaches to Artificial Intelligence (AI).

- Logic based - AI based on rules of logic equations. Several programming languages have been developed to make the task easier.
- Machine Learning - create model of the brain and throw more and more data into it. Somehow intelligence will spring up.

We will go with Machine Learning , specifically **Neural Networks** approach, which now is under active development. Turing test. Self-awareness, strong AI.

Machine Learning

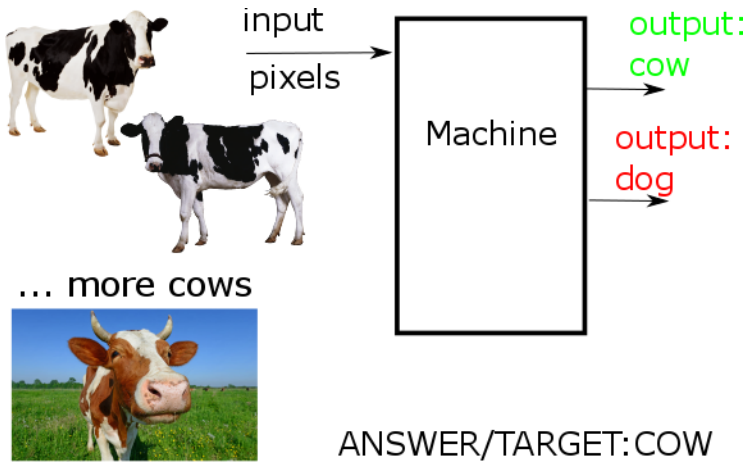


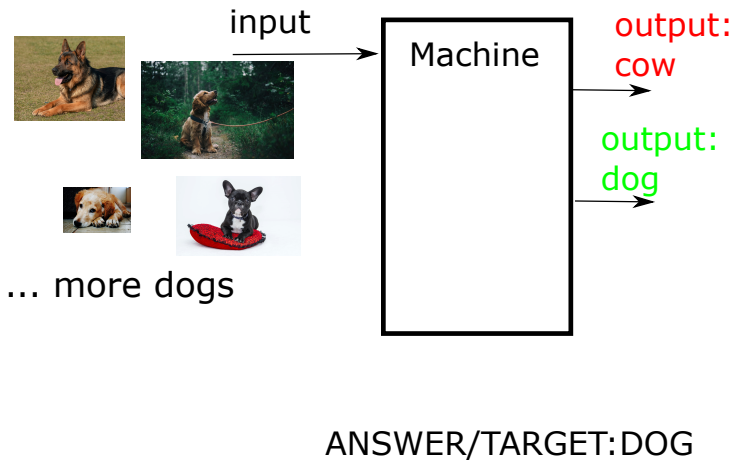
Not that different from the school.

- Training stade:
  You provide a lot of data to the system. You also provide right answers. System tunes itself until for each set of data it calculates right answer (passes the test).
- Working stage:
  You give system another data and hope that it gives right answer.

ML for image processing

Neural Networks are quite popular for image processing (and that is what we will be programming).





Input: arrays of pixels, Output - classification of the images. Some output goes high(fires, shown in green) to indicate that certain thing is present in the image.

Does it do it, really?

Lets have a look..

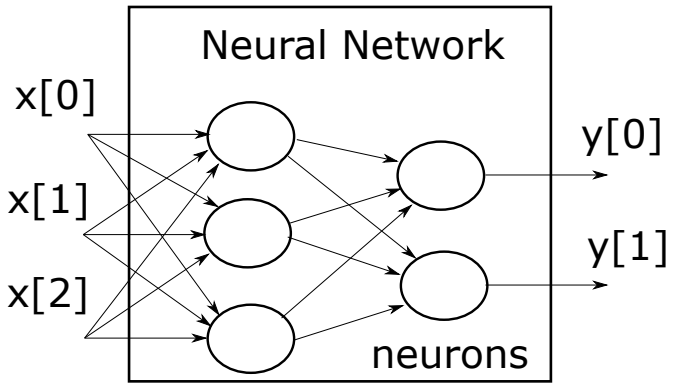
`http://nautil.us/blog/this-neural-net-hallucinates-sheep`

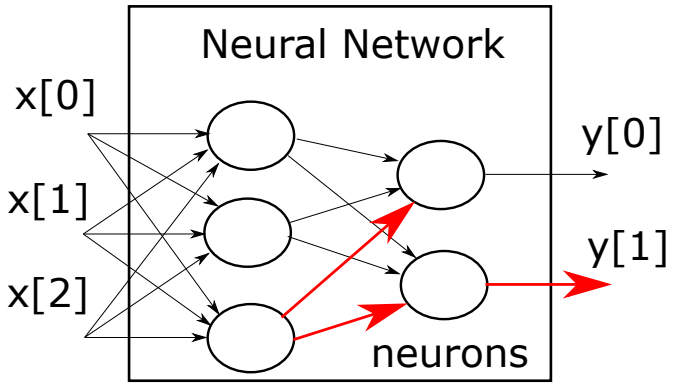
History of Neural Networks

- 1943 McCulloch and Pitts proposed neuron model (building block of the brain)
- 1958 Rosenblatt introduced the simple single layer networks now called Perceptrons. Minsky and Paperts book Perceptrons demonstrated the limitation of single layer perceptrons, and almost the whole field went into hibernation.
- 1986 The Back-Propagation learning algorithm (more on it later) for Multi-Layer Perceptrons was discovered. AI become active again.
- 2000s The power of Neural Network Ensembles, Support Vector Machines, Bayesian Techniques, Simulated Evolution, and Deep Learning becomes apparent. Development of customized hardware.

How Neural Network calculates the outputs?

1. It is called **Neural** because it is made out of **neurons**.
2. And there are many of them (network) and they are interconnected.
3. Some neurons can **fire** and its output output goes high (1).
4. It can make other neurons fire as well

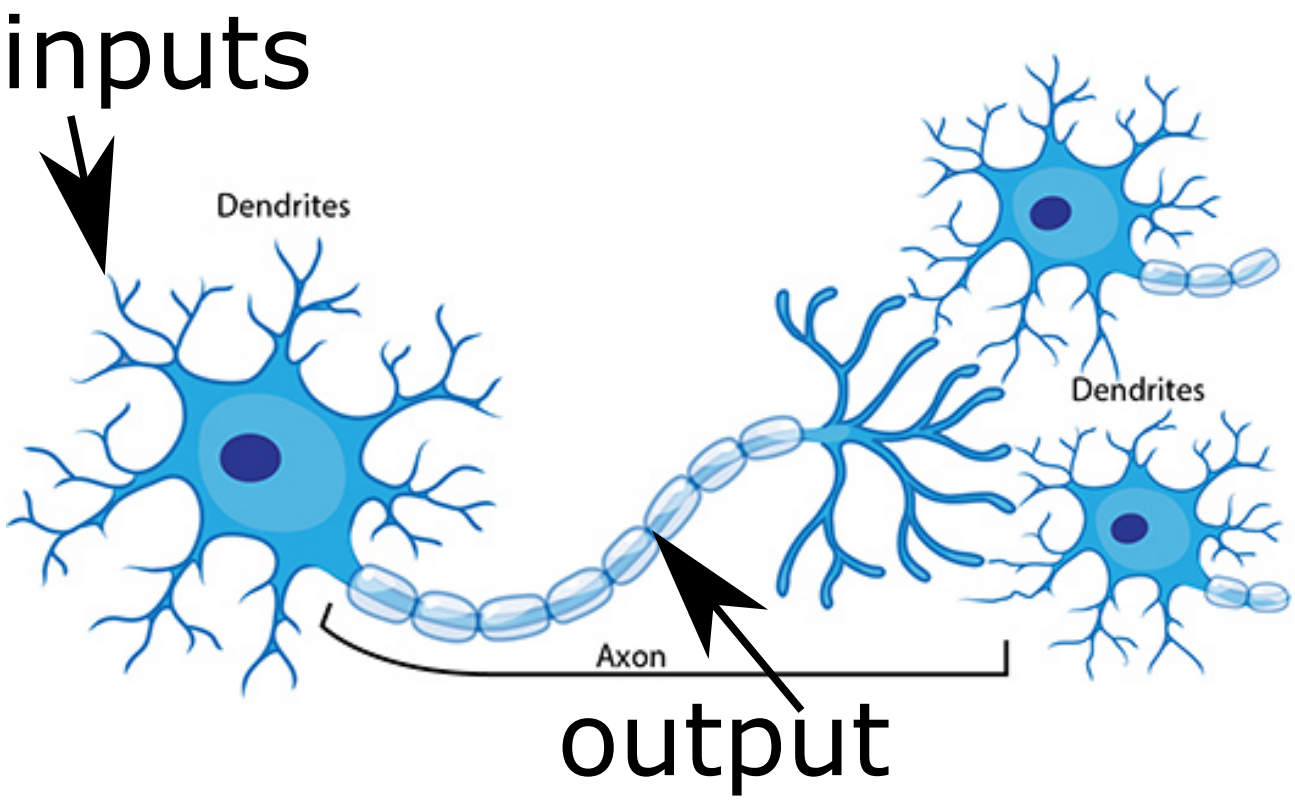




What is Neuron?

Structure of brain neuron

How was it invented? It was not...They copied it. Biologically inspired.



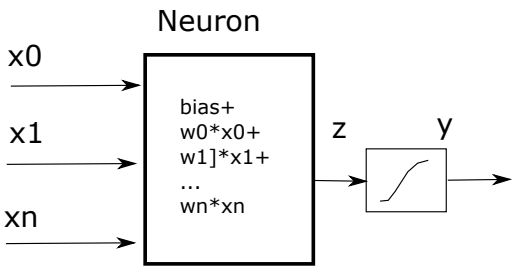
https://askabiologist.asu.edu/plosable/speed-human-brain Output of the neuron ia an input for another one.

What is the neuron model?

Basically neuron is a formula which can be converted to the hardware.
As a formula:

$$y = \sigma(bias + x_0 \cdot w_0 + x_1 0 \cdot w_1 + ...x_n \cdot w_n) \quad (1)$$

Not as scary as it looks...

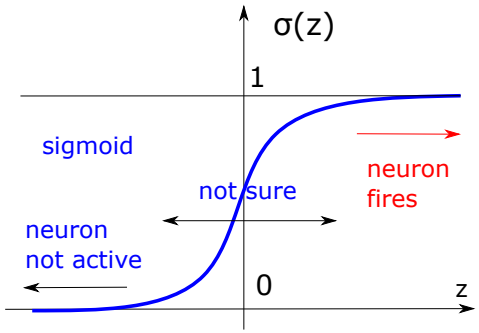


- Take input signal x_0 and multiply it with w_0
- Repeat for all remaining inputs ,**weight**s w are different for the inputs
- Add it all together and add bias to the sum:

$$z = bias + x_0 \cdot w_0 + ... + x_n \cdot w_n \quad (2)$$

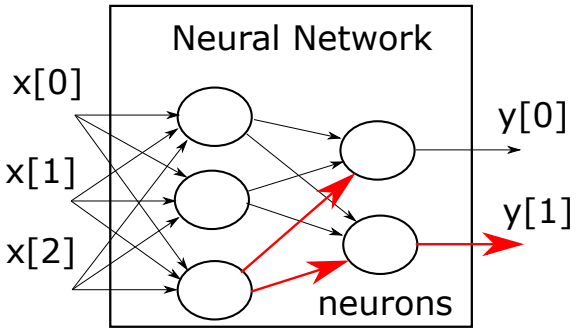
- Apply function $\sigma(z)$ to the sum.
- Output y goes high if $z > 0$.

Activation function $\sigma()$



- Function determines how neuron fires (activated)
- Common form is:

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



- This particular one is called **sigmoid**
- Three regions: Yeah, Nah, I don't know

It is simplest function but not only one used.

Neuron with one input

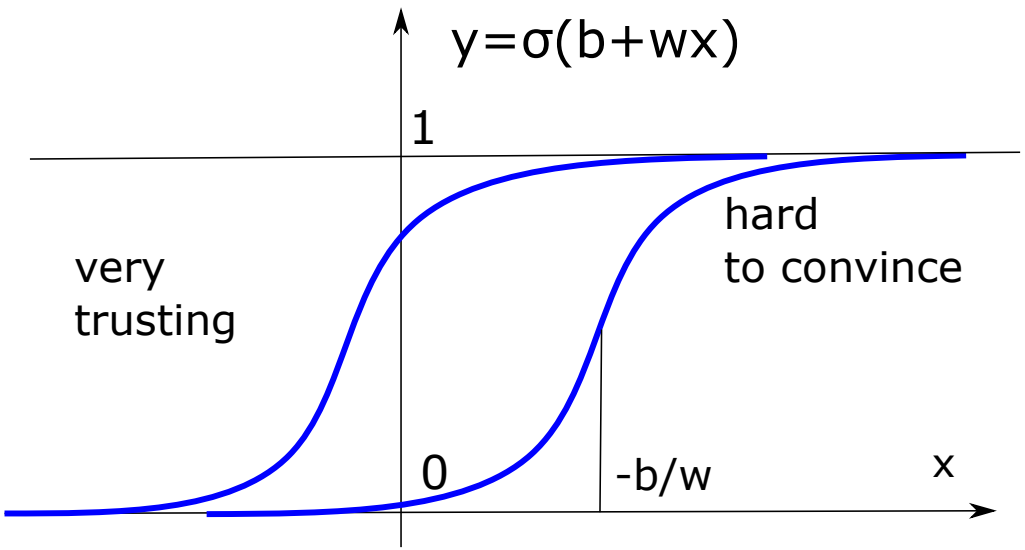


Figure: Two different neurons

- Equation:

$$y = \sigma(b + w \cdot x) \quad (4)$$

- Depending upon values of b and w:
  - neuron can fire for small values of x
  - .. or is hard to convince
- Transition happens when argument of $\sigma(z)$ is about 0, i.e. $b + w \cdot x = 0$

Neuron with 2 inputs

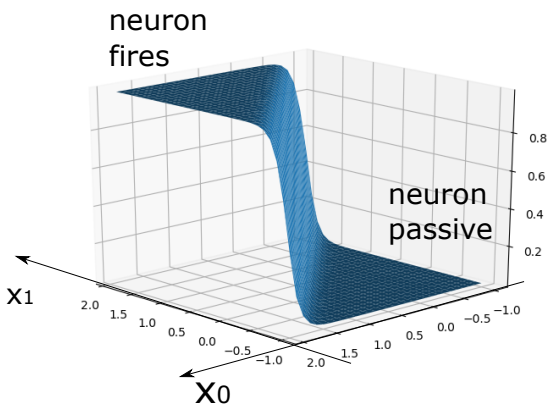


Figure: Output of neuron with two inputs

- Equation of neuron with two inputs:

$$y = \sigma(b + w_0 \cdot x_0 + w_1 \cdot x_1) \quad (5)$$

- If $b, w_0, w1$ change - shape of output $y(x_0, x_1)$ changes (demo)
- Space (x_0, x_1) is divided into two regions
- Boundary between regions is located where

$$y = \sigma(b + w_0 \cdot x_0 + w_1 \cdot x_1) \quad (6)$$

goes from 0 to 1.

- Transition happens when

$$b + w_0 \cdot x_0 + w_1 \cdot x_1 = 0 \quad (7)$$

Required neuron output

As you seen we can move/turn/scale neuron output as a function of its inputs. What we want it to be?

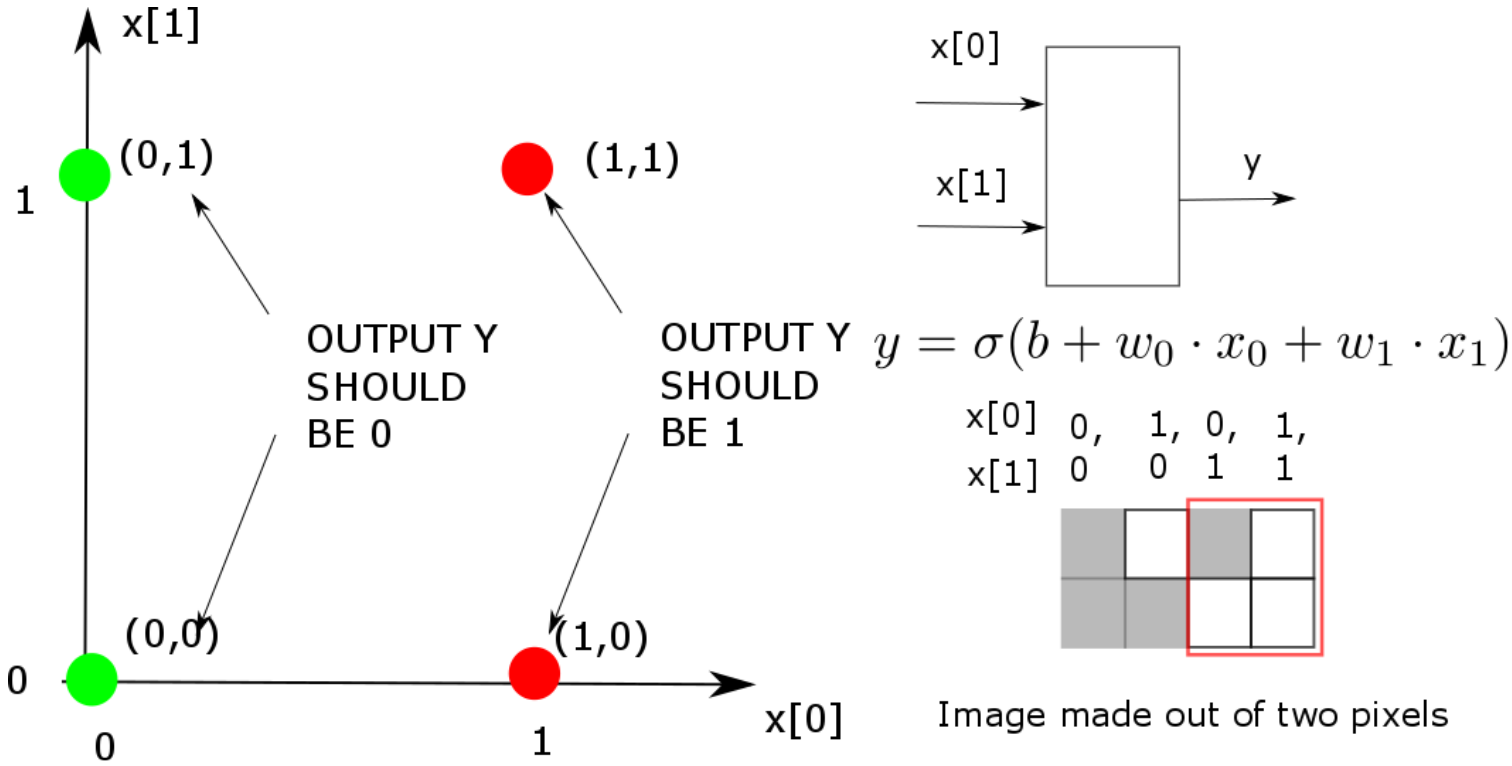


Figure:

Boundary ?

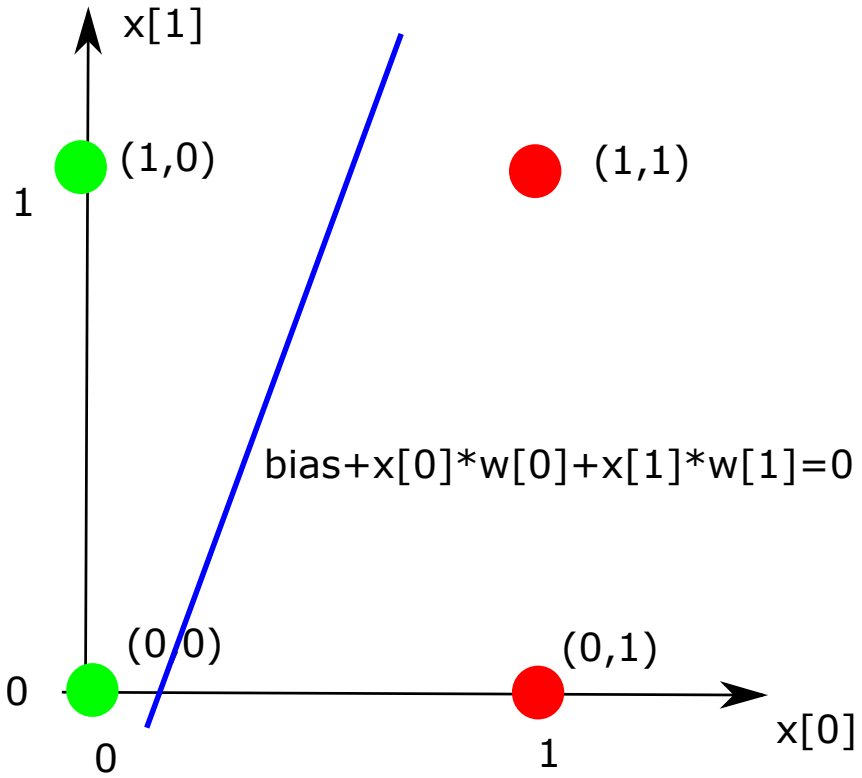


Figure:

What is the shape of the boundary?

$$b + w_0 \cdot x_0 + w_1 \cdot x_1 = 0 \quad (8)$$

We can write it as:

$$w_1 \cdot x_1 = -b - w_0 \cdot x_0 \quad (9)$$

and

$$x_1 = -b/w_1 - (w_0/w_1) \cdot x_0 \quad (10)$$

Which is equation of a straight line. x_1 as y, x_0 as an x and here it is:

$$y = -b/w_1 - (w_0/w_1) \cdot x \quad (11)$$

Boundary

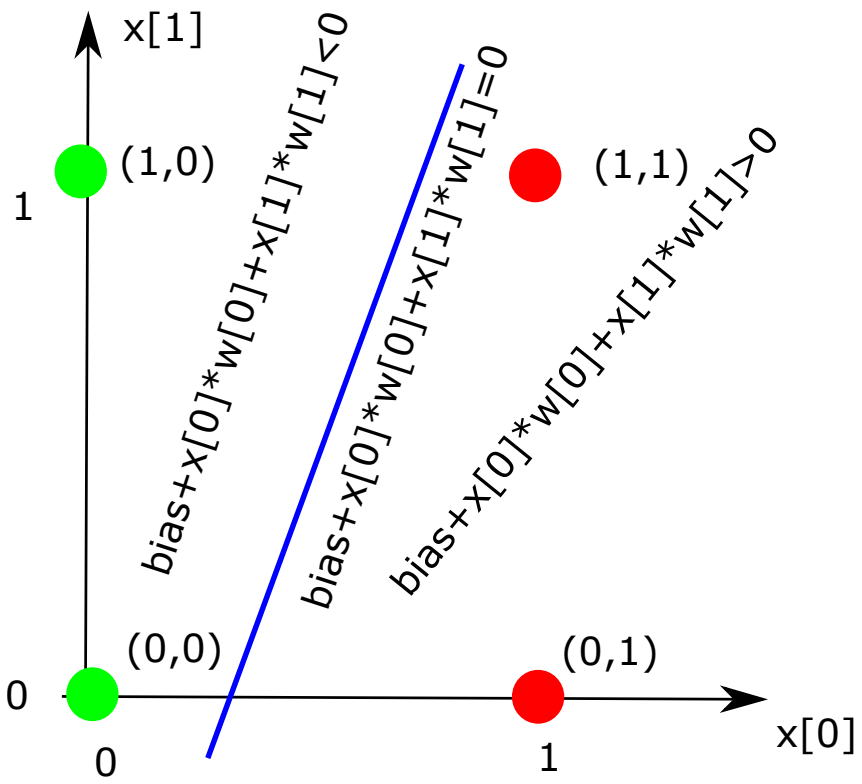


Figure:

With boundary

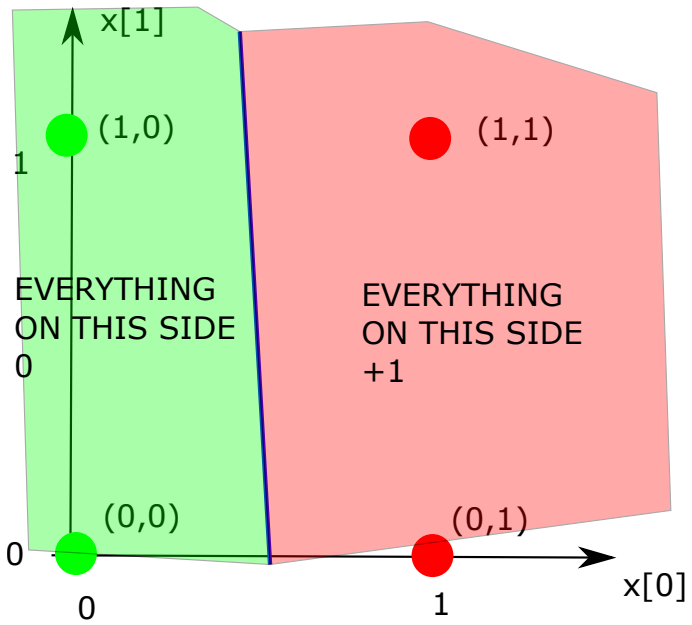
$$bias + w_0 \cdot x_0 + w_1 \cdot x_1 = 0 \quad (12)$$

we break all our possible data values into regions.
In one region

$$bias + w_0 \cdot x_0 + w_1 \cdot x_1 > 0 \quad (13)$$

and neuron fires.
In another region its output stays low.

Boundary



If we manage to find such a line the our equation

$$y = \sigma(bias + w_0 \cdot x_0 + w_1 \cdot x_1) \qquad (14)$$

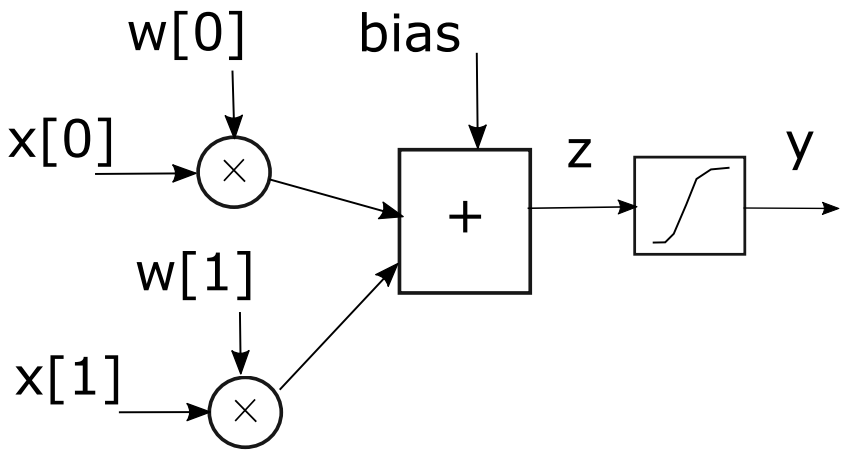
will give right answers.

Figure:

Summary: When neuron is trained properly all possible data values combinations are broken into regions. To decide that data belongs to certain category we check into which region data falls in.
All possible data values combinations are called data **space**.

Tuning/training

OK, hopefully we understand now how neuron classifies the data - by dividing data space into regions.
One neuron can produce one line.



It is all very nice, but it does not answer the question: **How to tune the Neural Net?** (How to find this magic line?)

Training dataset

To find the **decision boundary** we provide the Neural Net with training data.

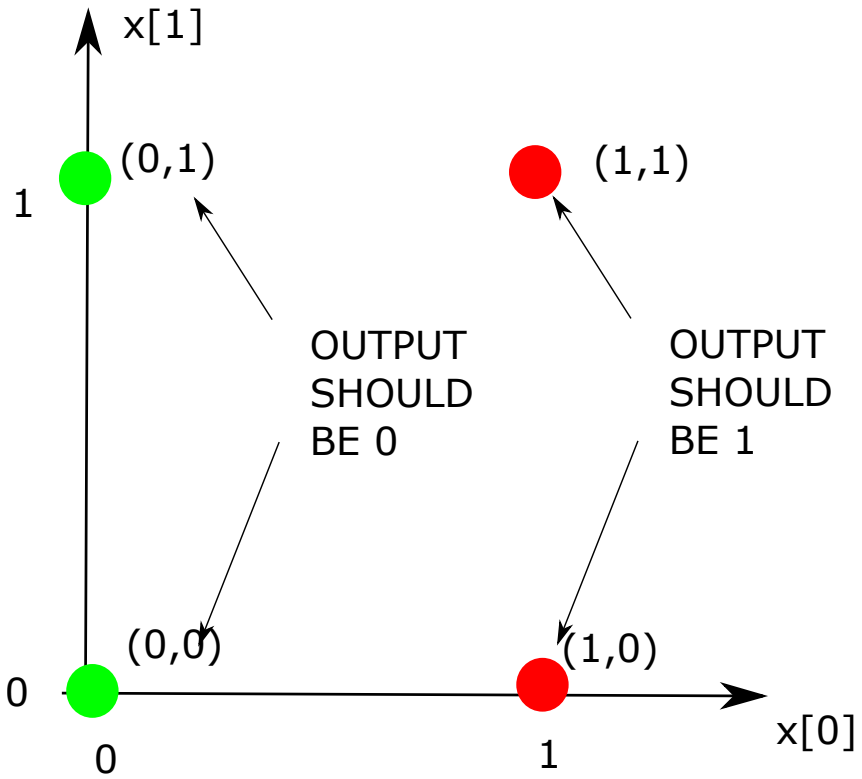


Figure:

Truth Table:

x0	x1	t
0	0	0
0	1	0
1	0	1
1	1	1

What it is when training finished?

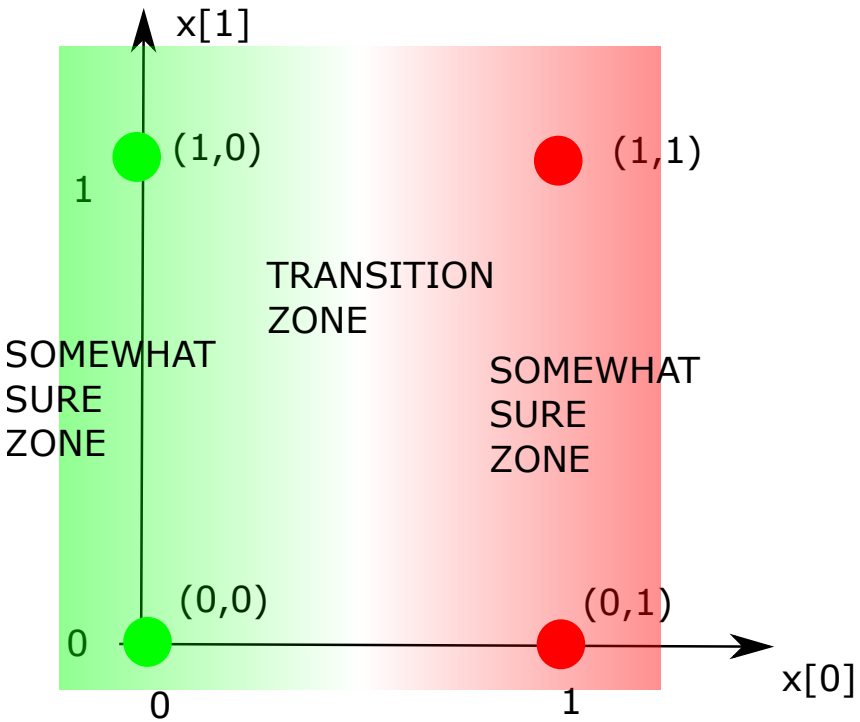


Figure:

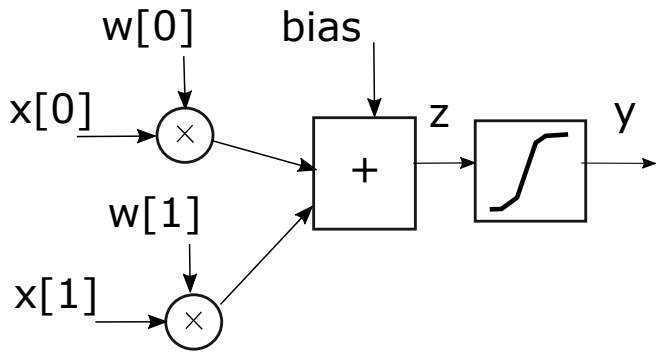


Figure:

Boundary

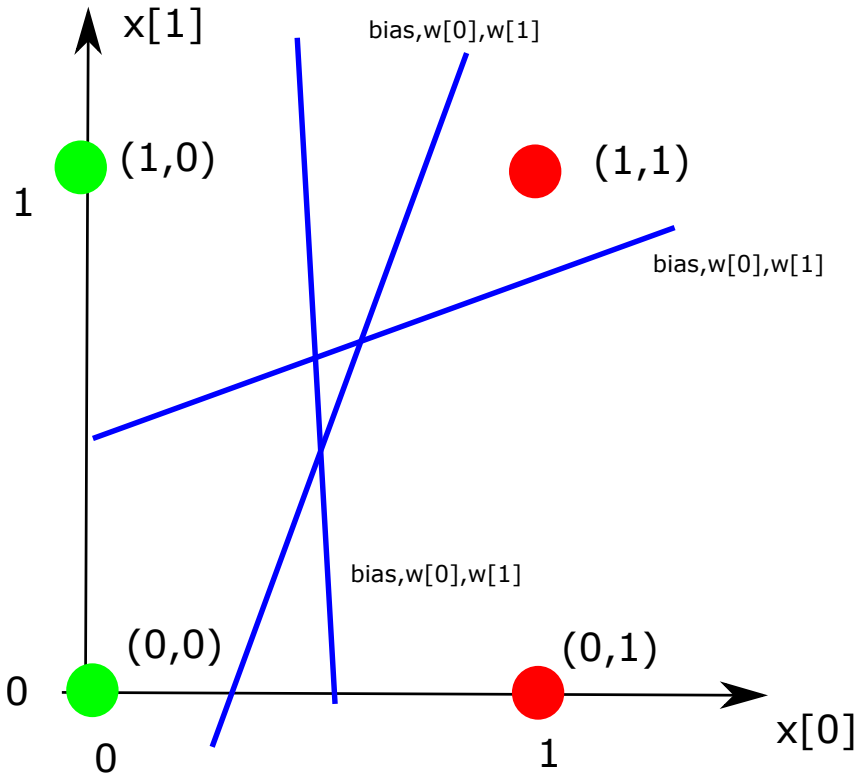


Figure:

As we change *bias*, w_0 and w_1 we draw different lines.
Our goal is to draw such a line that all 0s are on one side of it ans all 1s are on another side.

What is happening during training?

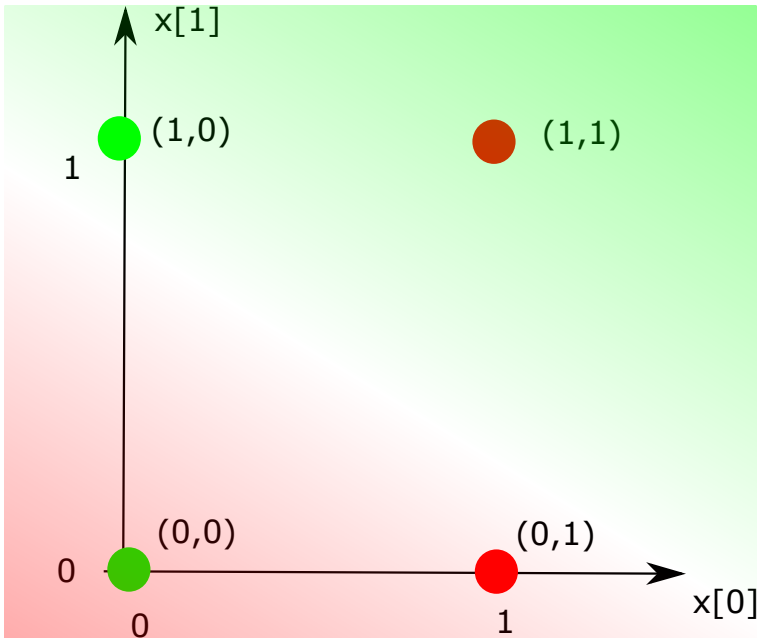


Figure:

- We start by choosing some random values for b, w_0 and w_1.
- That gives us somewhat random division into regions
- We calculate NN outputs and compare outputs with the target
- In this case:
  - point (0,0) is very wrong. So is point (1,1)
  - points (0,1) and (1,0) are somewhat right
- If classification error is big we change b, w_0 and w_1.

What is happening during training?

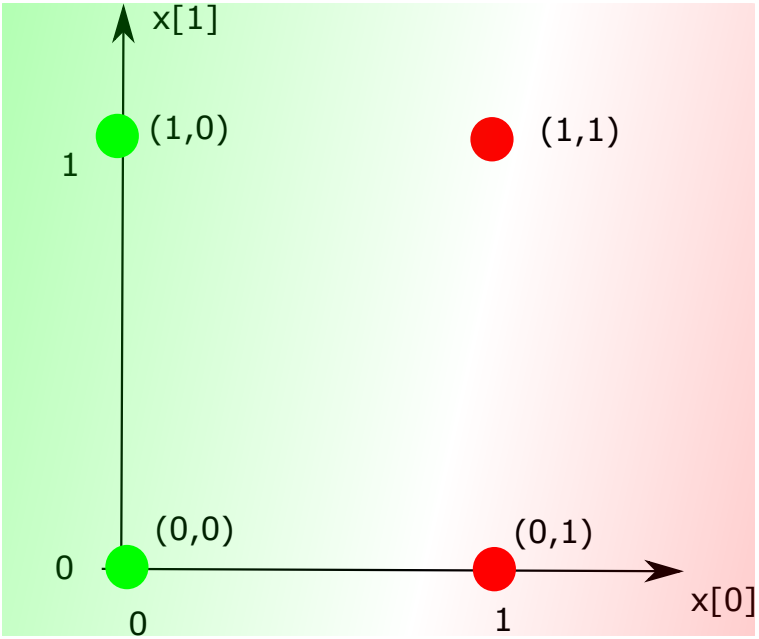


Figure: Bad classification

- We changed b, w_0 and w_1.
- It gives us new division
- Error value changes
- Keep going until error is small

How to calculate the error? 3D view of neuron output.

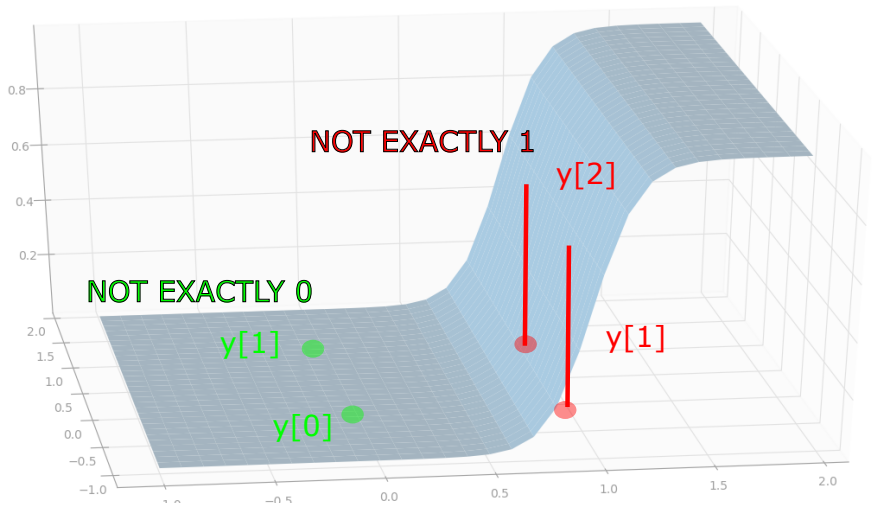


Figure: Neuron output

- As we tune the Neural Network blue surface shifts, rotates and scales
- Perfect fit happens when outputs of the neuron are same as training data:
  $y[0] = t[0] = 0$,
  $y[1] = t[1] = 0$,
  $y[2] = t[2] = 1$,
  $y[3] = t[3] = 1$

How to calculate the error? 3D view of neuron output.

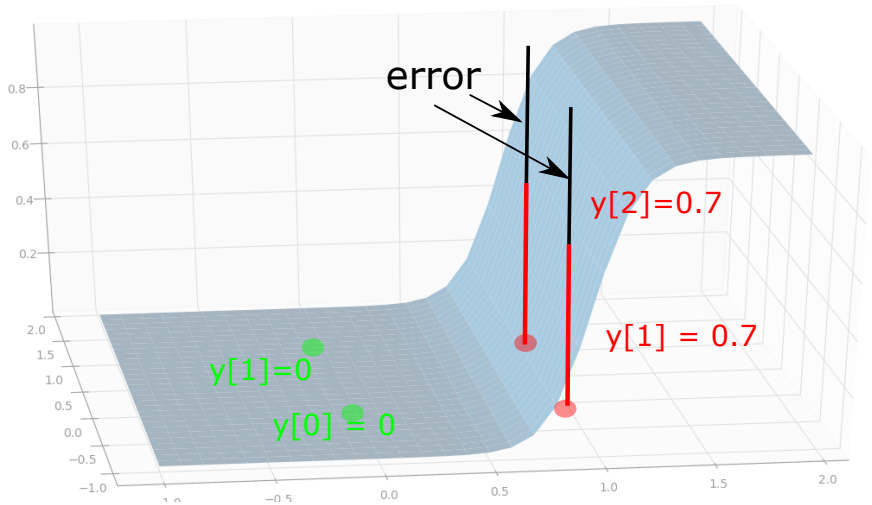


Figure: Can be better shifted to the left

- If tuning is not perfect:
  y[0] != t[0], y[1] != t[1], y[2] != t[2], y[3] != t[3]
- We have errors for each of the outputs:
  error[0] = y[0] - t[0],
  error[1] = y[1] - t[1],
  error[2] = y[2] - t[2],
  error[3] = y[3] - t[3],

Error for whole training dataset.

Lets try to get **error signal** - just like ENGR101. Error signal will tell us how far away from optimum point we are. And then we will try to drive error to 0.
We have Truth Table and neuron output (such as it is)

x0	x1	t(target)	$y = \sigma(bias + w_0 \cdot x_0 + w_1 \cdot x_1)$	y-t
0	0	0	y0=0.1	0.1
0	1	0	y1=0.2	0.2
1	0	1	y2=0.8	0.2
1	1	1	y3=0.3	0.7

with coefficients $w_0, w_1, bias$ to tune (train) until y is same as t. If tuning is not perfect then for each row of Truth table there is an error:
$error_i = y_i - t_i$.

Error of whole dataset

Error for whole of training set is the sum of the squares of the errors for individual entries:

$$err = (y_0 - t_0)^2 + (y_1 - t_1)^2 + (y_1 - t_1)^2 + (y_2 - t_1)^2 \tag{15}$$

- Why squares?

We can calculate an error. How we can calculate combination of *bias*, w_0 and w_1 so that

$$\begin{aligned} err = (y_0(b, w_0, w_1) - t_0)^2 + (y_1(b, w_0, w_1) - t_1)^2 \\ +(y_1(b, w_0, w_1) - t_1)^2 + (y_2(b, w_0, w_1) - t_1)^2 \end{aligned} \tag{16}$$

is minimum?
Global search? It will guarantee perfect result but would too slow to be useful.

How to minimize this error

If we have error which changes smoothly we can use this fact to accelerate the search.

Here is an example of the error changing as w_1 changes (*bias* and w_0 are fixed).

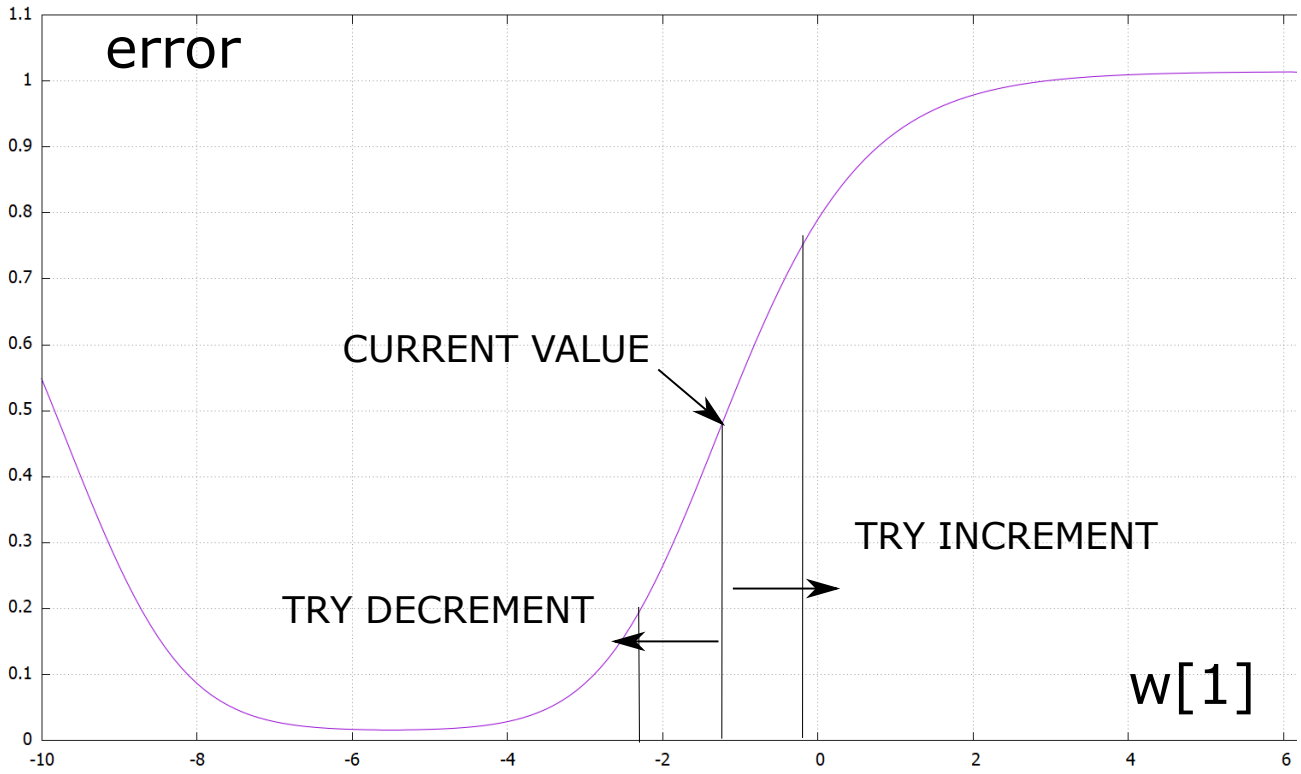


Figure:

Is error smooth?

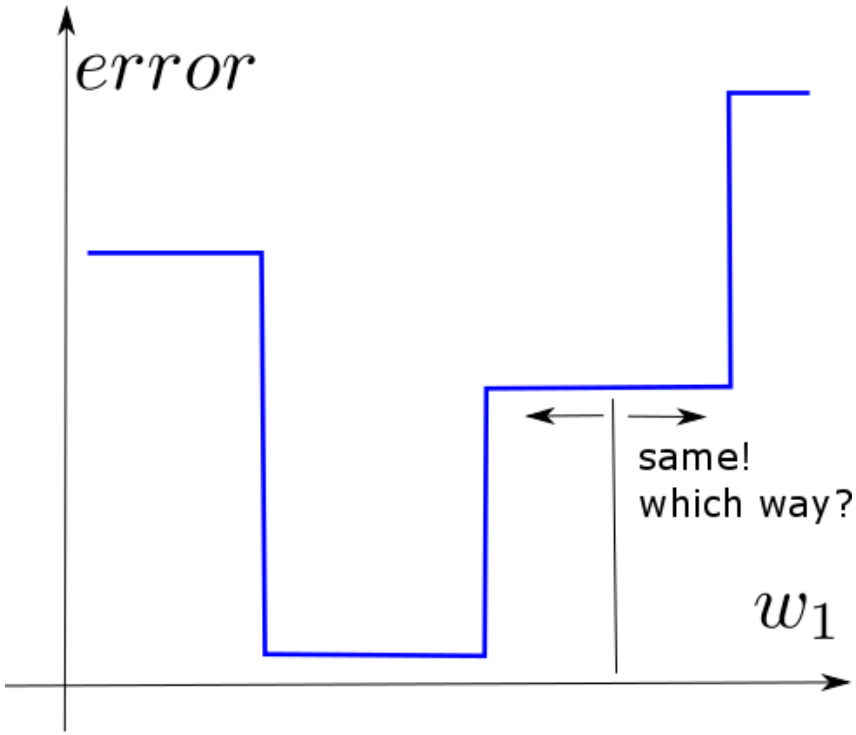


Figure:

- If there is no change in either direction...
- How do we know which way to go?
- How to avoid this?

Activation function is the answer.

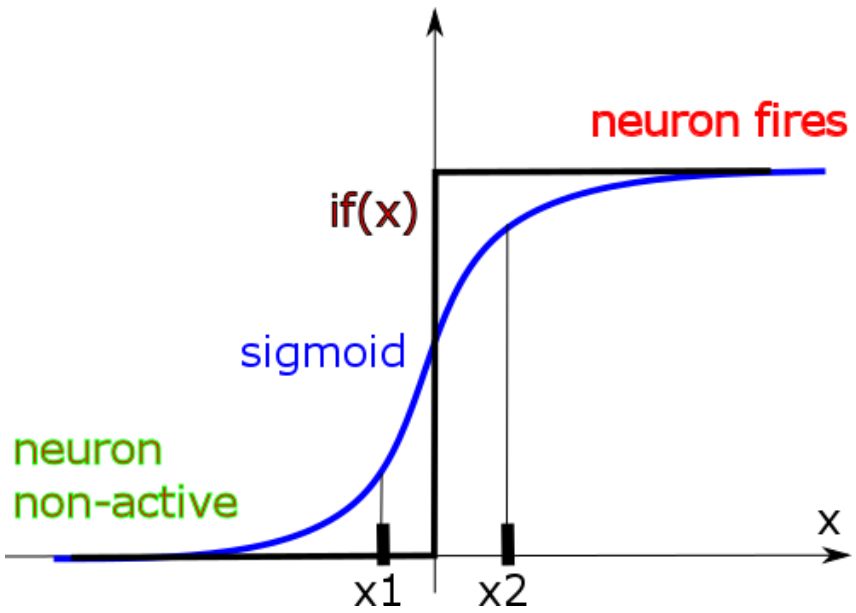


Figure:

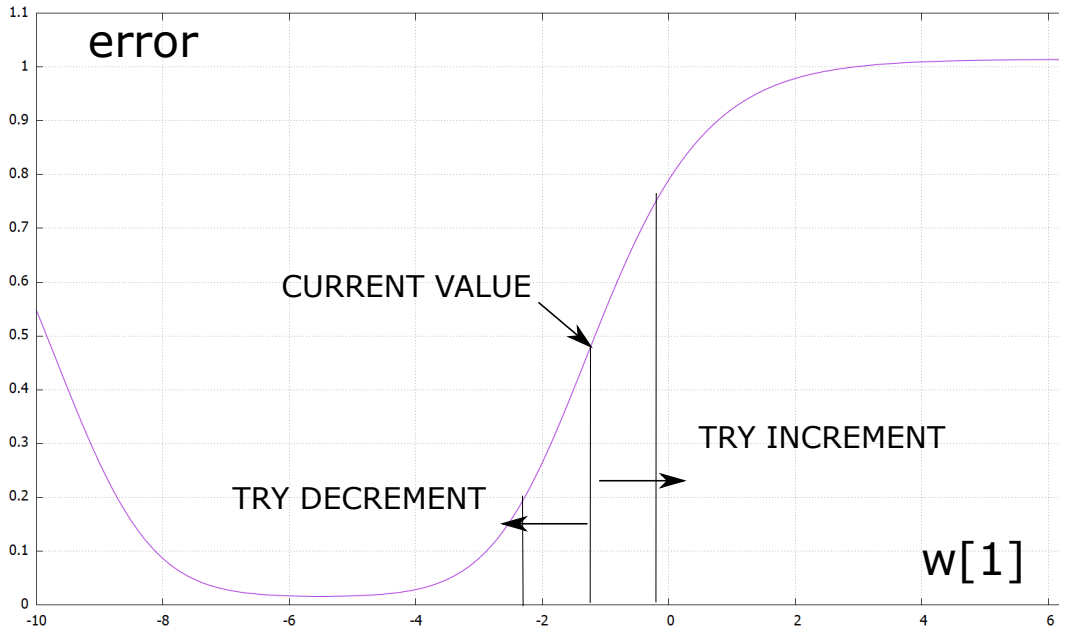
- Sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{17}$$

  is smooth
- First Neural Nets had been using step **if()** function instead
- Neuron with with step activation function is called **perceptron**
- If perceptorns are used then **error** have abrupt changes
- Only global search can be used then

OK, error is smooth. How to minimize the error?

Instead of trying all possible values we can do something better.



This way we will go down-hill with fixed step.

- Calculate error
  $e0 = error$
- Try increment:
  $w_i = w_i + dw$
- Calculate error
  $e1 = error$
- If $e1 > e0$ - increment is wrong thing to do.
  Step back.

How it looks in 2D?

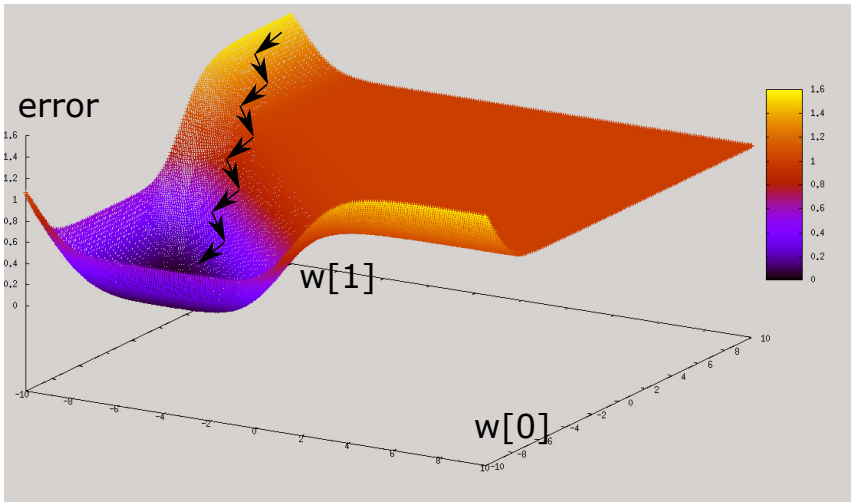


Figure: Searching for minimum error

- Algorithm will go down-hill turning 90 degrees all the time
- Eventually it will find the point with minimum error
- It moves in fixed steps.
- Can stop if error becomes small
- Goes faster if step is big - but misses minimum by bigger margin
- Slower with small steps - but can closer to the minimum

What we can do to accelerate the search?

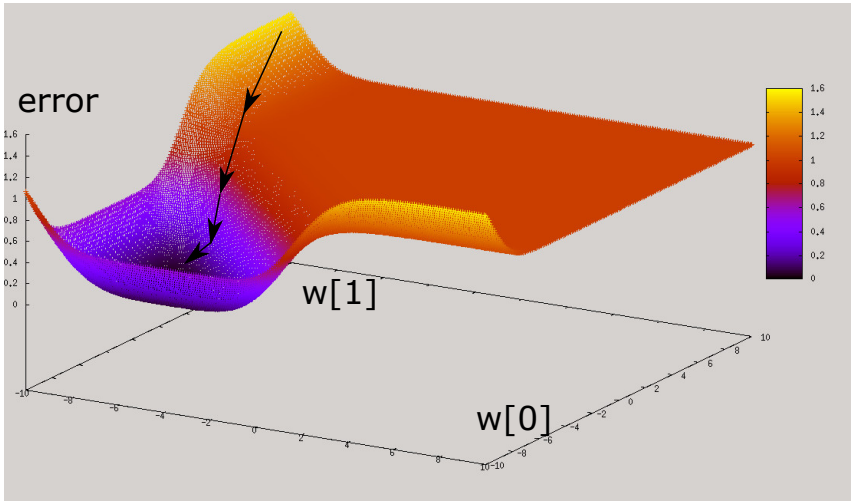
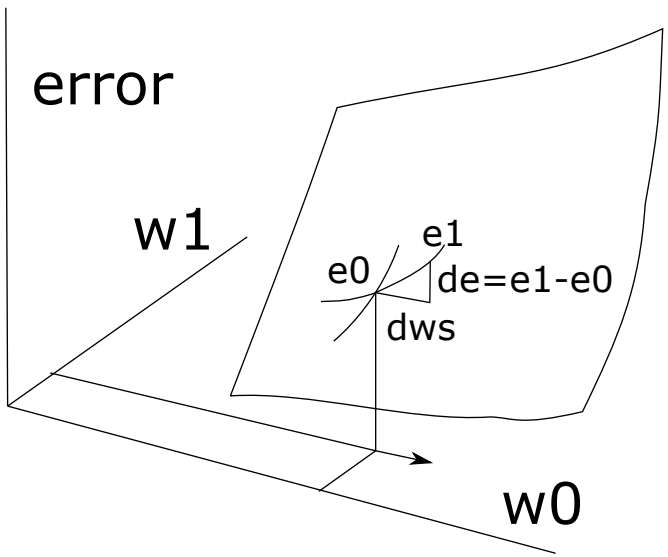


Figure: Faster search

- We look around
- Find steepest descent direction
- Step in this direction.

Look for direction of steepest descent.

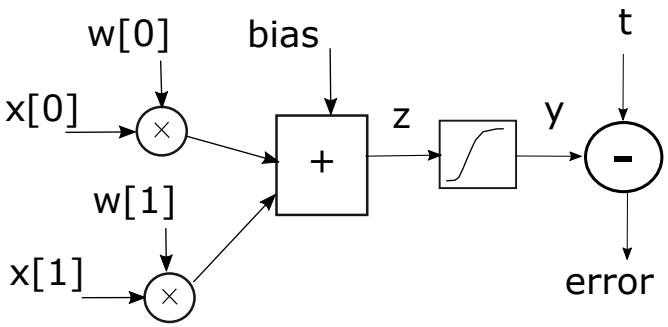
How to find the gradient?

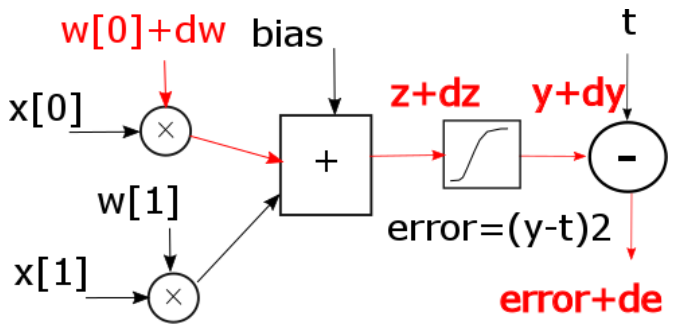


For all w and bias:

- Calculate error value. Remember it.
- Change w_i: $w_i = w_i + dws$
- Calculate error again
- Calculate slope in direction of $\frac{derror}{dw_i} = \frac{(e1-e0)}{dws}$
- Remember it.
- Change w_i back: $w_i = w_i - dws$
- Go for next i

Calculating gradient: Numerical example





x[][0]	x[][1]	t[]	y[]	err[]=t[]-y[]
0	0	1	0.5	0.5
0	1	1	0.4	0.6
1	0	1	0.4	0.6
1	1	0	0.7	-0.7

Total error:

$e0 = 0.5^2 + 0.6^2 + 0.6^2 + (-0.7)^2 = 1.46$

x[][0]	x[][1]	t[]	y[]	err[]=t[]-y[]
0	0	1	0.7	0.3
0	1	1	0.6	0.4
1	0	1	0.4	0.6
1	1	0	0.2	-0.2

Total error:

$e1 = 0.3^2 + 0.4^2 + 0.6^2 + (-0.2)^2 = 0.65$

Total error goes down. Change is: $de = e1 - e0 = 0.65 - 1.46 = -0.81$
Gradient component is $0.81/dw$.

Step in this direction

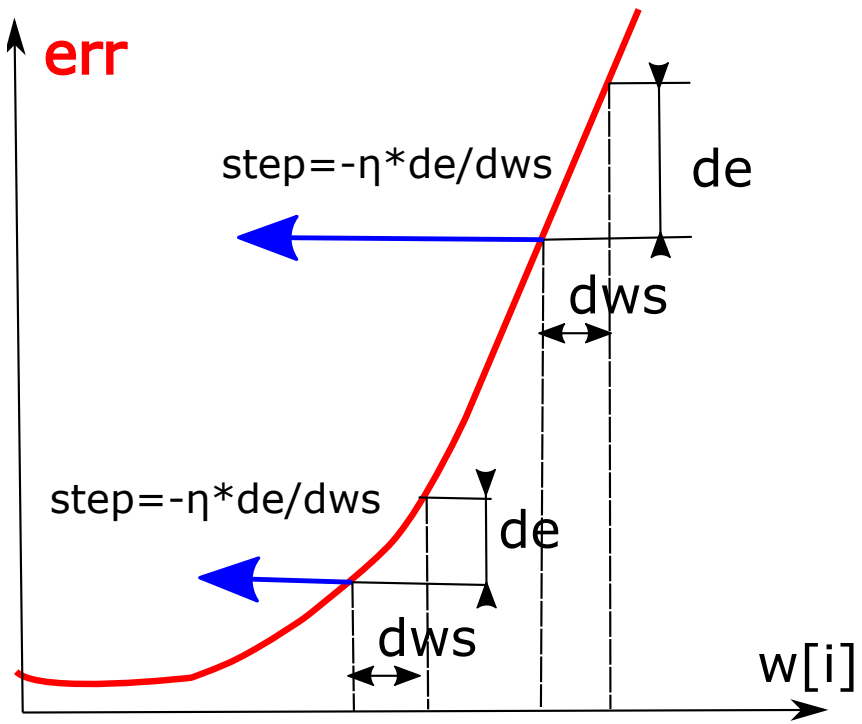


Figure:

- We have vector(1D array) of slopes:
  dw_i.
- It contains slope information: bigger the dw_i - steeper error changes with w_i
- To make a step:
  For all is:

$$w_i = w_i - \eta \cdot dw_i \qquad (18)$$

- It works almost automatically: Bigger the dw_i (slope) - bigger is the step
- *eta* - learning rate: how big is the step

Step carefully

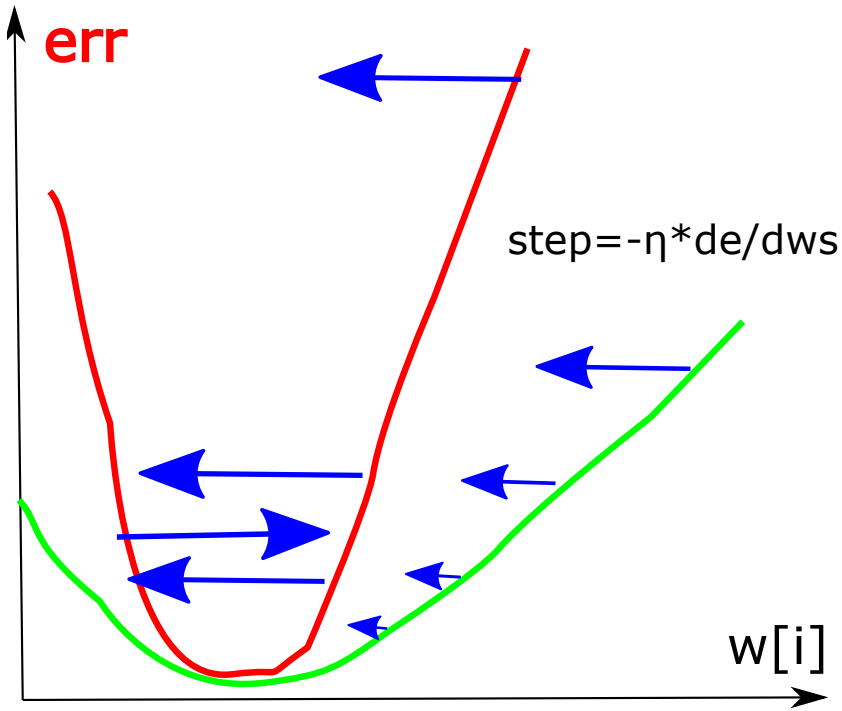


Figure:

- η converts slope into step size - how far algorithm will go for given slope
- If η - algorithm moves in small steps
- With bigger η algorithm moves in bigger steps, what is faster
- If algorithm moves in big steps it can miss narrow **valley** and become unstable

How to choose learning rate η ?

Figure: Which is good learning rate?

- η is called **learning rate**
- It determines how fast Neural Net learns (goes downhill trying to reduce the error)
- Faster is not necessarily better - if there are fine details Neural Net is going to miss them (is here a lesson for us?)
- It is set by trying
- Like tuning PID(E101) really - keep trying.

Two ways to train: Batch and on-line.

- So far we used what is called **batch** training. Error is calculated for the whole dataset and Network parameters are changed to minimize this error.

x[0]	x[1]	t	y	err=t-y
0	0	1	0.5	0.5
0	1	1	0.7	0.3
1	0	1	0.4	0.6
1	1	0	0.1	0.1

Total error: $0.5^2 + 0.3^2 + 0.6^2 + 0.1^2 = 0.71$. Minimize this value.

- If dataset is big calculating error for all entries(rows) in dataset takes long time. And then we have to repeat it many times.
- There is another way which works faster for big dataset.

On-line training. 1st step

Pick one entry(row) (i) from dataset. Calculate error for this entry: $(t_i - y_i)^2$. Make one search step. Error for this entry improves.

Before search step:

x[0]	x[1]	t	y	err=t-y
0	0	1	0.5	0.5
0	1	1		
1	0	1		
1	1	0		

After the step:

x[0]	x[1]	t	y	err=t-y
0	0	1	0.7	0.3
0	1	1		
1	0	1		
1	1	0		

On-line training, 2nd step

Pick another entry(row) from training set. Calculate error for this entry and make one step trying to improve it.

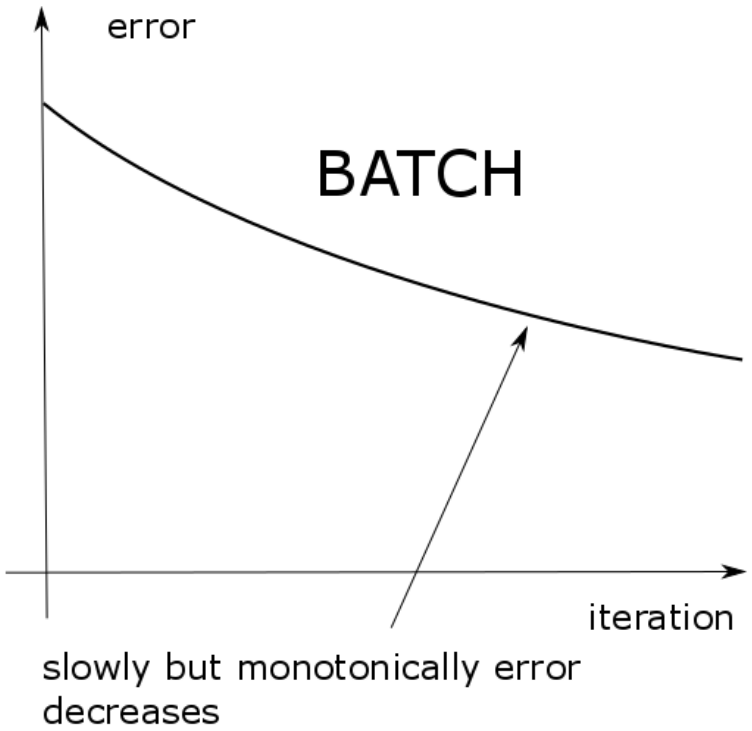
x[0]	x[1]	t	y	err=t-y
0	0	1	0.7	0.3
0	1	1		
1	0	1	0.4	0.6
1	1	0		

x[0]	x[1]	t	y	err=t-y
0	0	1	0.7	0.3
0	1	1		
1	0	1	0.8	0.2
1	1	0		

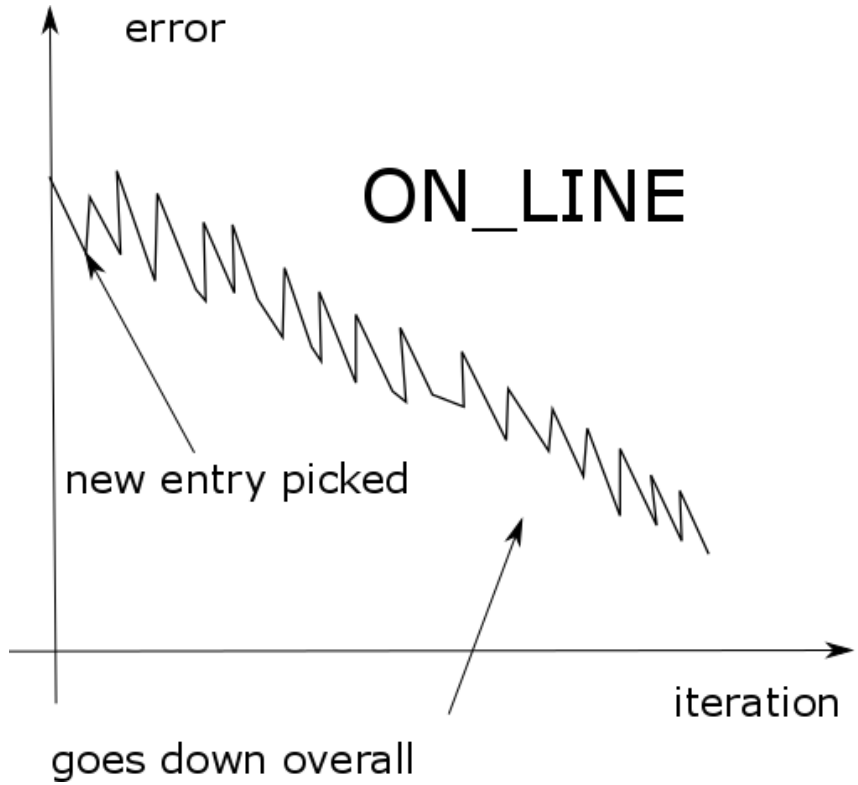
Continue to do that.
Rows usually are picked randomly. It guarantees that every row has a chance to contribute to the training. Result training is a compromise between providing smallest possible error for all and each row.

On-line versus batch convergence

Before second step



After second step



Error can go up if on-line training is used: new training dataset entry was picked, NN never seen it before, not tuned for this at all.

Multilayer networks

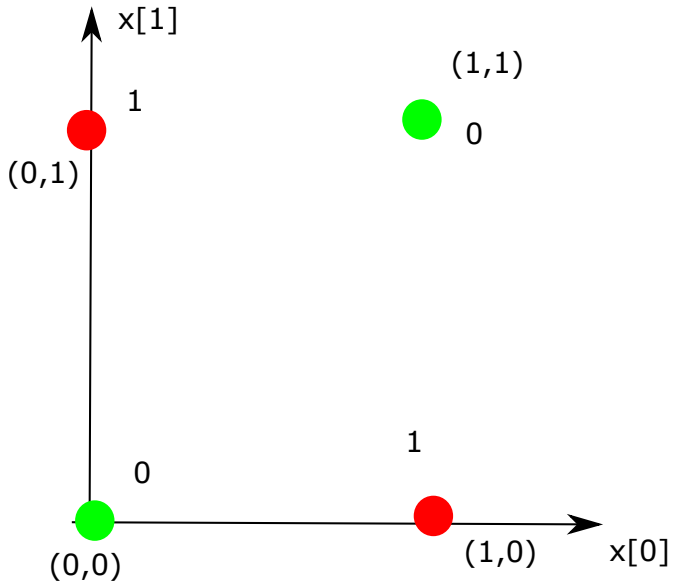


Figure: Linearly non-separable dataset-can not be divided into categories using simple single line.

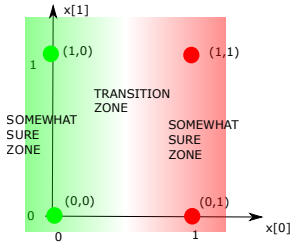


Figure: Single neuron can do that

- Some data are such that no single straight line can provide division into classification regions
- More complex shapes are needed

Multilayer networks

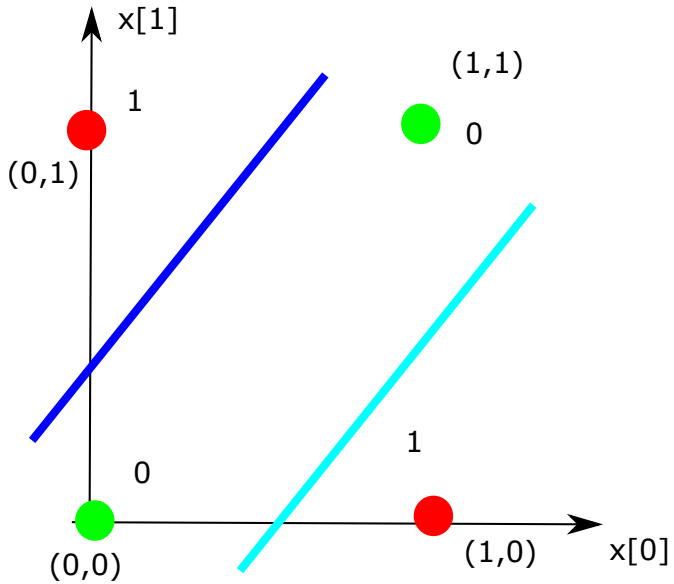
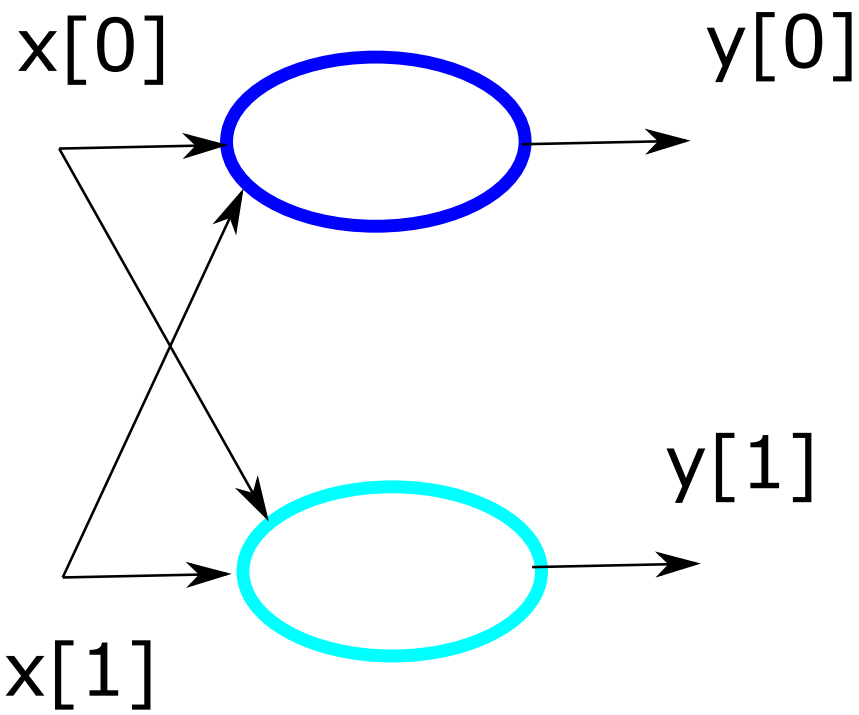


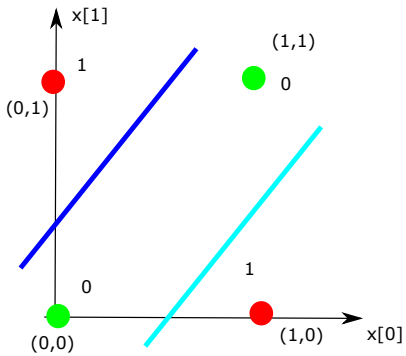
Figure: Two lines would be sufficient for this dataset

We can have two neurons producing two different decision boundaries

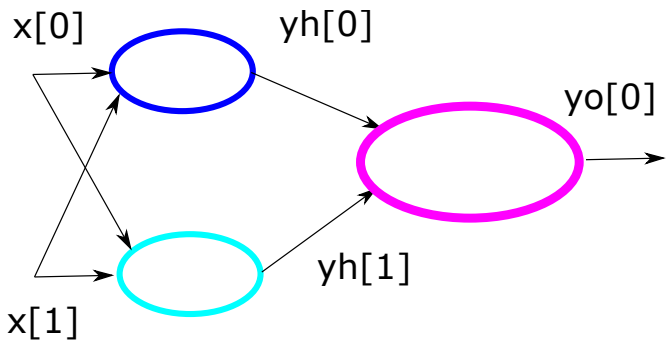


How to combine their outputs ?

Multilayer networks



Is it possible to unite the outputs of using another neuron:



$$yo = 0 + yh[0] * (1) + (-1) * yh[0] \tag{19}$$

Higher level neuron parameters can be:

$$\begin{aligned} bias &= 0 \\ w[0] &= 1 \\ w[1] &= -1 \end{aligned} \tag{20}$$

Multilayer Networks

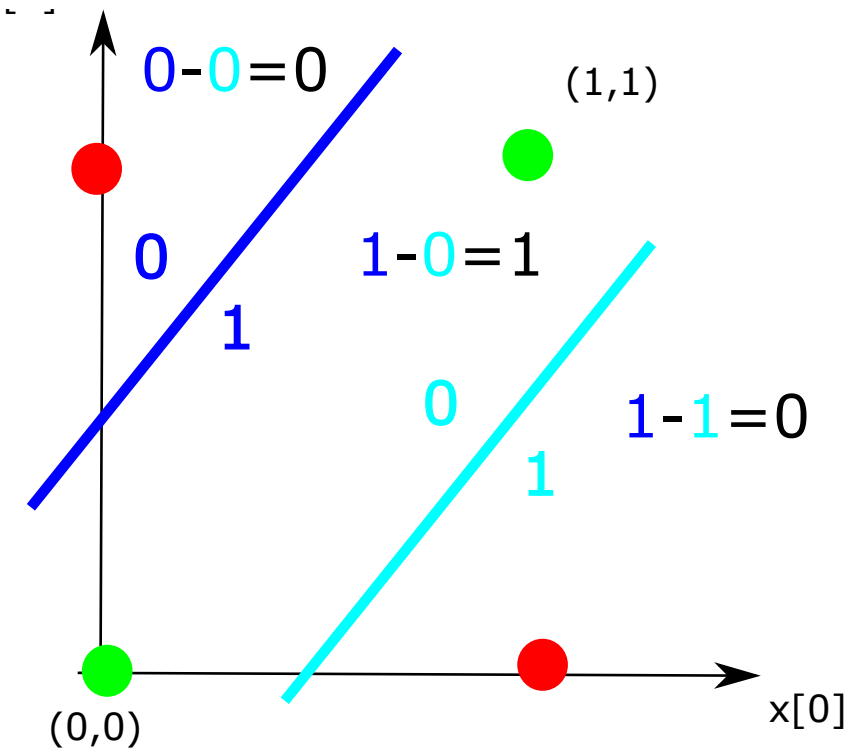


Figure: Combining outputs

- Neuron output equals 1 on one side of decision boundary and 0 on another side
- What happens if we subtract output of one neuron from another?
- Activation function can clamp neuron output so it is not greater than 1 or negative
- More than one classification boundaries
- Combining outputs complicated classification regions can be created

Demo - multi2d.py

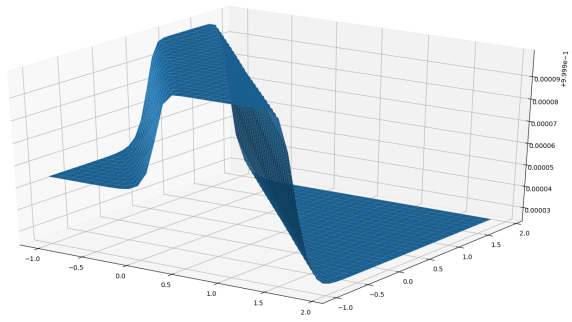


Figure: 2 layers network

Listing 1: Caption

```
b0 = -5.0
w00 = 10.5
w10 = 10.0
z0 = b0+w00*x0+w10*x1
y0 = 1.0/(1.0+np.exp(-z0))
b1 = 5.0
w10 = 12.5
w11 =10.0
z1 = b1+w10*x0+w11*x1
y1 = 1.0/(1.0+np.exp(-z1))
b2 = 10.0
w20 = -10.5
w21 = 10.0
z = b2+w20*y0+w21*y1
y = 1.0/(1.0+np.exp(-z))
```

Which is equivalent to...

Demo - multi2d.py

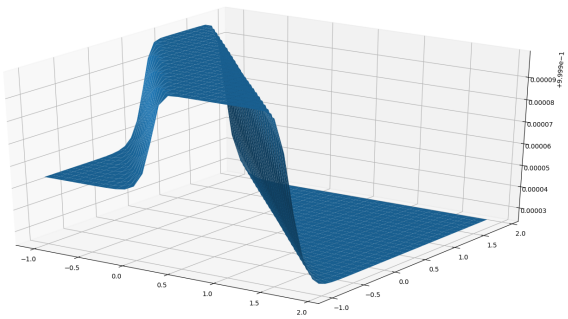


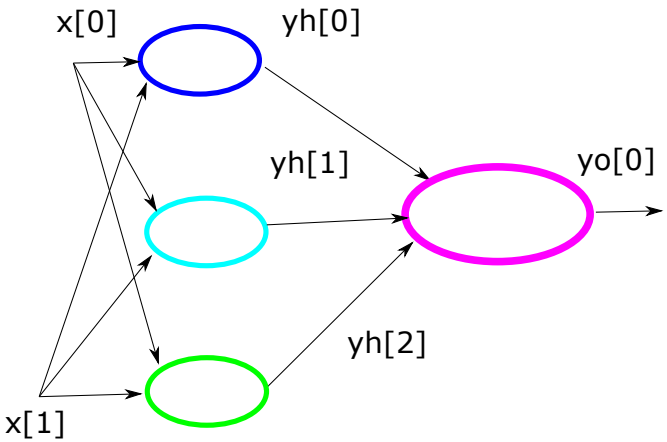
Figure: 2 layers network

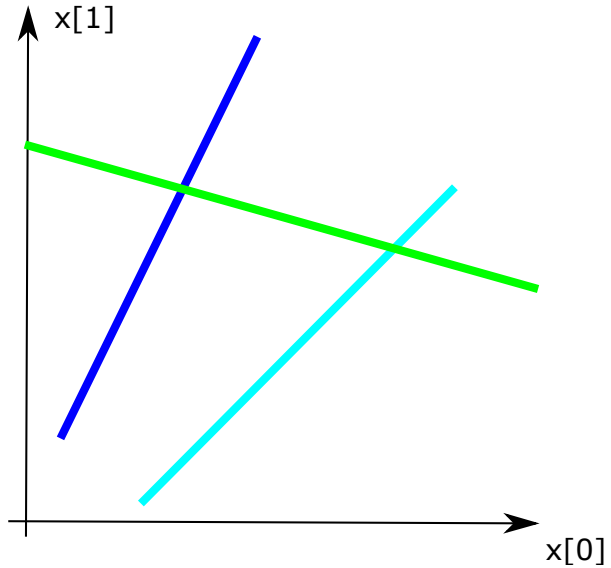
$$
\begin{aligned}
z_0 &= b_0 + w_{00} * x_0 + w_{10} * x_1 \\
y_0 &= \frac{1}{1 + e^{-z_0}} \\
z_1 &= b_1 + w_{10} * x_0 + w_{11} * x_1 \\
y_1 &= \frac{1}{1 + e^{-z_1}} \\
z &= b_2 + w_{20} * y_0 + w_{21} * y_1 \\
y &= \frac{1}{1 + e^{-z}}
\end{aligned}
\tag{21}
$$

Outputs of two neurons are inputs for third one.

Complicated shapes

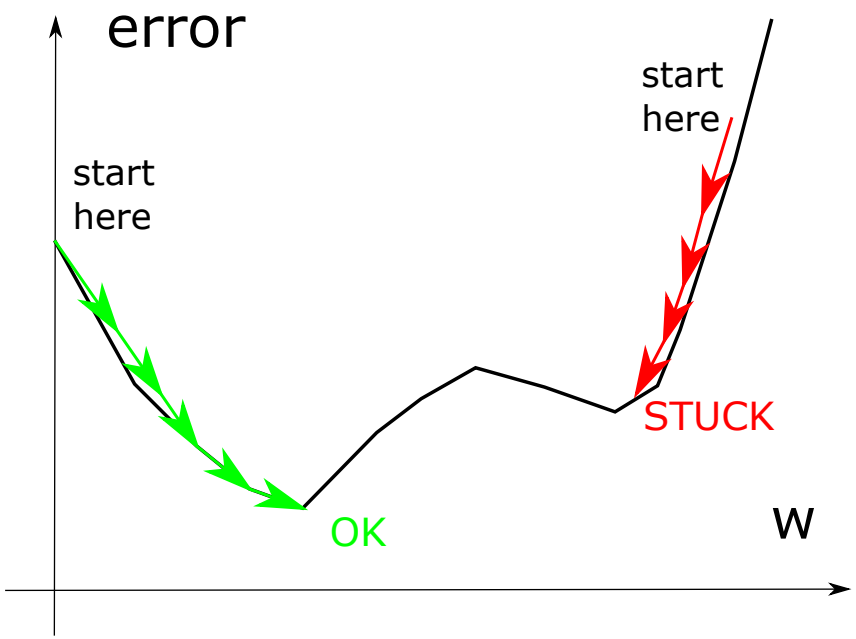
Adding more neurons in first layer (called **hidden** layer) we can create more lines. Second layer (called **output**) neurons combine them into regions made out of line segments.





As shapes are getting more complicated it is possible that we will have several solutions. And some of these solutions will be better than others. What it means?

Several minimums are possible



Error can have several minimums, but only one of them is really good one.
What can we do if we found not so good minimum first?
Out search algorithm always try to go downhill and we can never get out of local minimum.

Try to start the search from different point. It becomes game of chance, but the chances are that we can miss local minimum and go straight to true one.

Make starting point random.

How random?

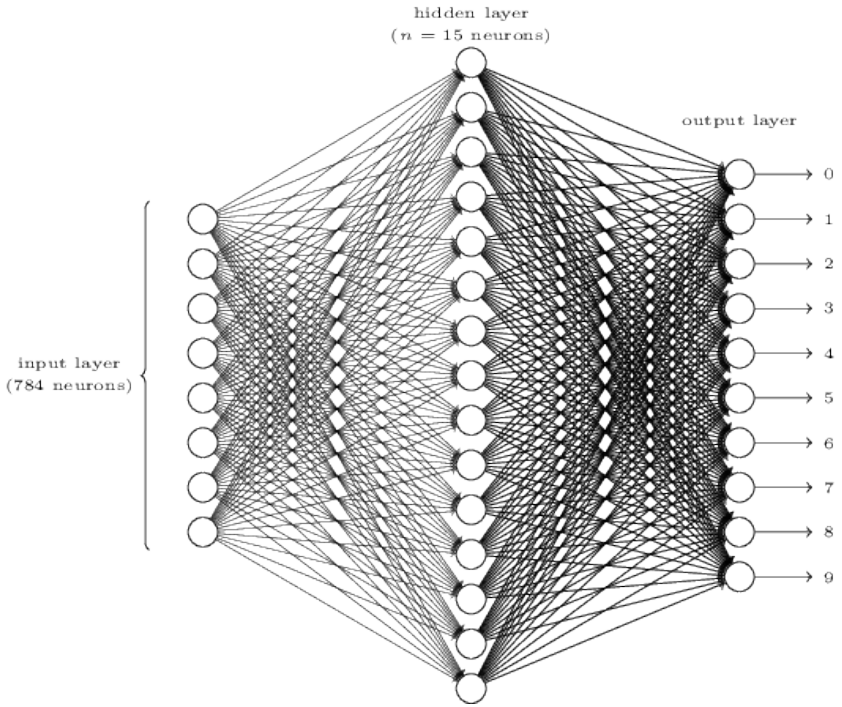
Everything exactly equal to 0 is bad idea. Remember neuron equation?

$$y = \sigma(bias + w[0] * x[0] + w[1] * x[1]) \tag{22}$$

If *bias* and all *ws* are equal to 0 then line actually does not exist at all. It would be hard to find optimum position of something which is not there at all.
So, initializing biases and weights with too small values can be not such a great idea.

Any useful Neural Network is BIG

‘

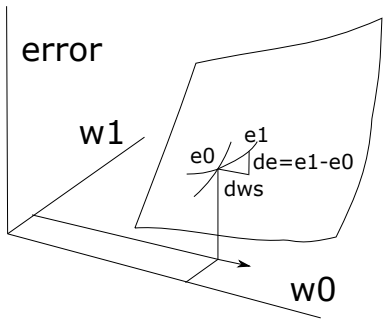


Let’s recall something...

- In the project you will use network with 64 inputs, 8 outputs and 100-200 neurons in hidden layer
- It contains 64*150+150*8= 1200 weights and 158 biases
- It is just a toy network
- ”Stanford researchers have created an even bigger network, with 11.2 billion parameters” [https://www.popsci.com/science/06/stanfords-artificial-neural-network-biggest-ever/]

Back to gradient: Copy of previous slide.

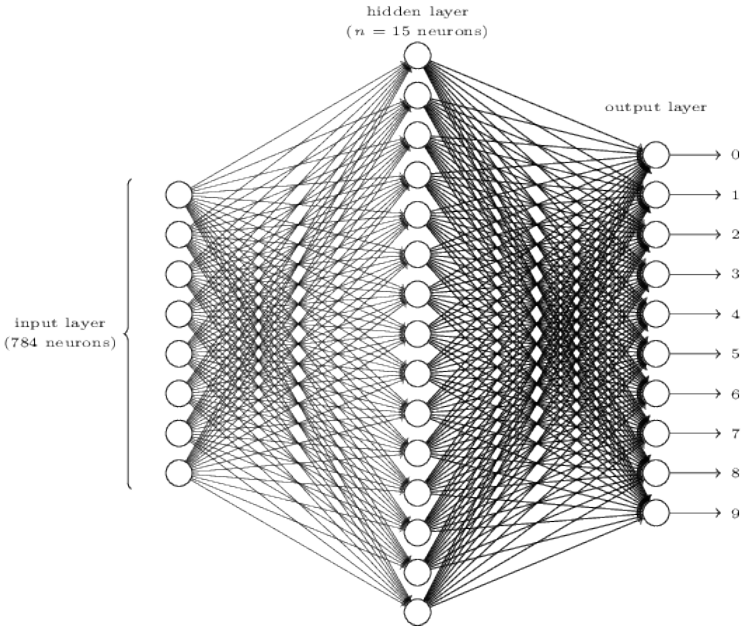
How to find the gradient?



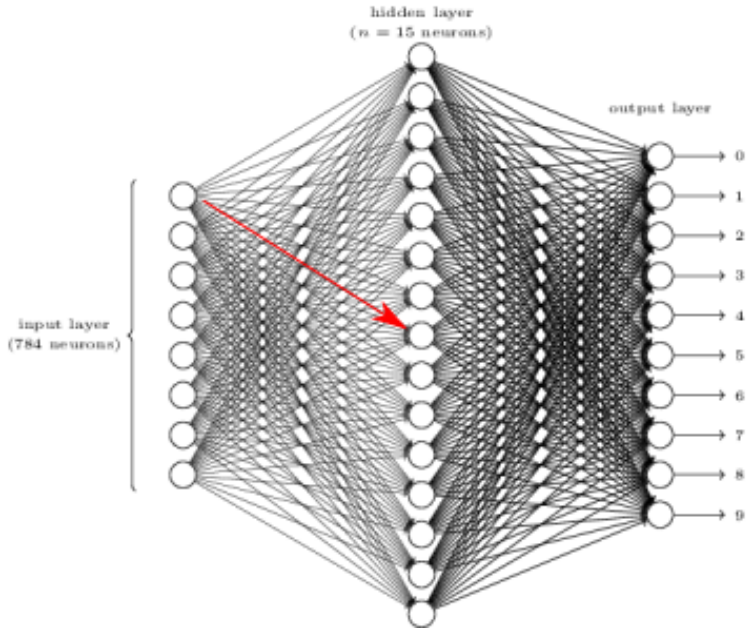
For all w and bias:

- **Calculate error value**. Remember it.
- Change w_i: $w_i = w_i + dws$
- **Calculate error again**
- Calculate slope in direction of $\frac{derror}{dw_i} = \frac{(e1-e0)}{dws}$
- Remember it.
- Change w_i back: $w_i = w_i - dws$
- Go for next i

And it has to be done for **EACH** weight and **EACH** bias



Calculate net output



Change one weight. Calculate output again

And do it for every connection(i.e. weight)
Now we are somewhat in 1985. Neural Networks were non-practical then because they were slow. Until team of psychologists and computer scientist proposed **backpropagation** algorithms. Backpropagation is way to estimate $\frac{de}{dw}$ very fast.

Backpropagation - for our toy example

We change one weight. All numbers downstream from this weight change too (shown in red).

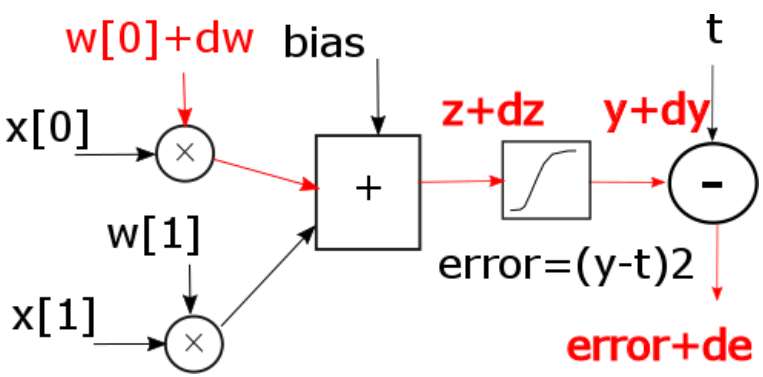


Figure: Neuron being tuned for $w[0]$

- We want to figure out how error is changing when w_0 is changing, i.e. we want to know $\frac{de}{dw_0}$
- It goes in **chain**: if w_0 changes the z changes. In turn, when z changes, y changes too. And, at last, if y changed then *error* changed too.
- ENGR121, chain rule:
  If $y = y(z)$ and $z = z(x)$ then
  $\frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx}$

Backpropagation - for our toy example

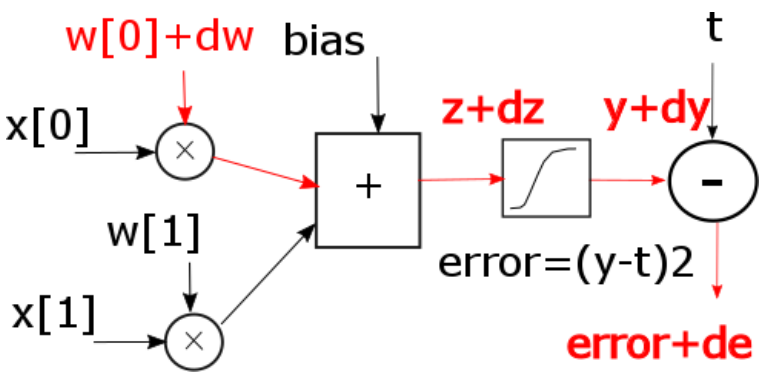


Figure:

- How fast *erroe* is changing when w_0 changes?

$$\frac{derror}{dw_0} = ? \tag{23}$$

- Chain rule:

$$\frac{derror}{dw_0} = \frac{derror}{dy} \cdot \frac{dy}{dz} \cdot \frac{dz}{dw_0} \tag{24}$$

First term: $\frac{de}{dy}$?

Simple one:
$error = (y - t^2)$

$$\frac{de}{dy} = \frac{d((y-t)^2)}{dy} = 2 \cdot (y-t) \cdot \frac{d(y-t)}{dy} = \quad 2 \cdot (y-t) \qquad (25)$$

Second term $\frac{dy}{dz}$

It is derivative of sigmoid function. We have to derive only once.

$$\frac{d}{dx}\sigma(x) = \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \tag{26}$$

$$= \frac{d}{dx}\left(1+\mathrm{e}^{-x}\right)^{-1} \tag{27}$$

$$= -(1+e^{-x})^{-2}(-e^{-x}) \tag{28}$$

$$= \frac{e^{-x}}{(1+e^{-x})^2} \tag{29}$$

$$= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \tag{30}$$

$$= \frac{1}{1+e^{-x}} \cdot \frac{(1+e^{-x})-1}{1+e^{-x}} \tag{31}$$

$$= \frac{1}{1+e^{-x}} \cdot \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) \tag{32}$$

$$= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \tag{33}$$

$$= \sigma(x) \cdot (1-\sigma(x)) \tag{34}$$

Third term $\frac{dz}{dw_0}$

Recall that:

$$z = b + x_0 \cdot w_0 + x_1 \cdot w_1 \tag{35}$$

Then

$$\frac{dz}{dw_0} = \frac{d(b + x_0 \cdot w_0 + x_1 \cdot w_1)}{dw_0} = x_0 \tag{36}$$

Put all terms together

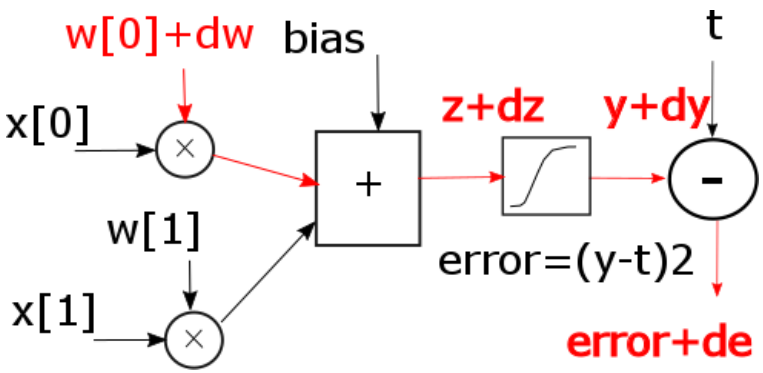
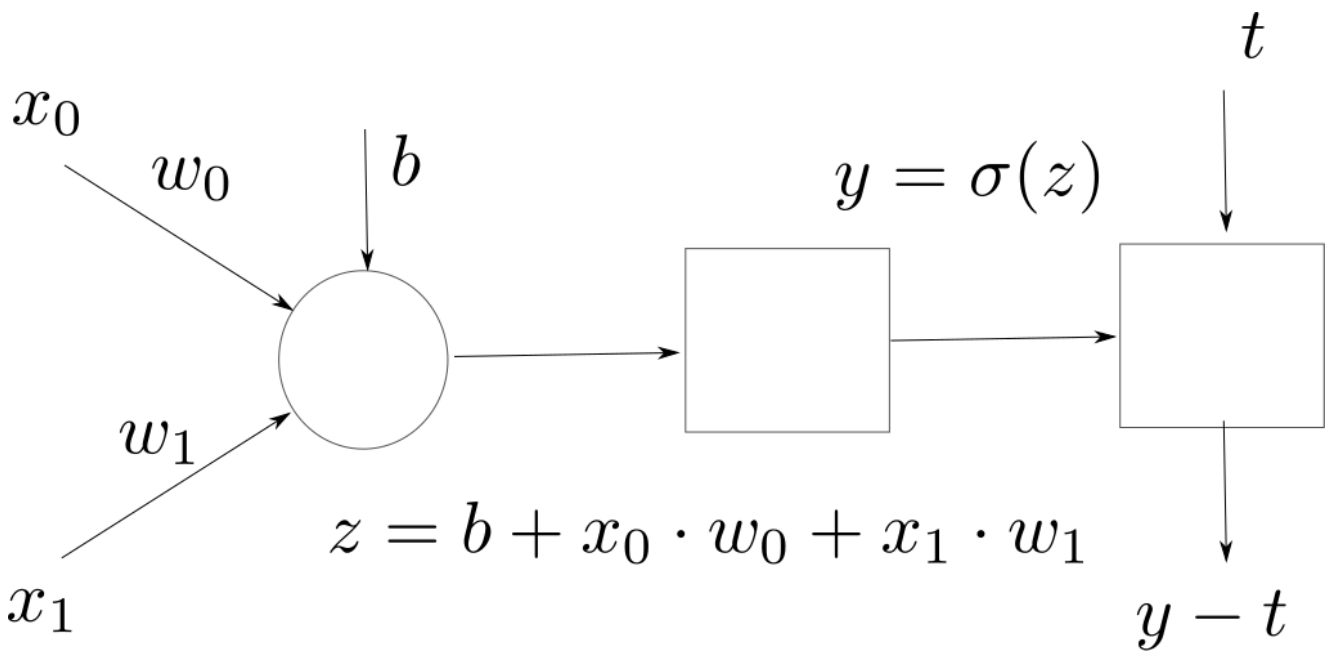


Figure:

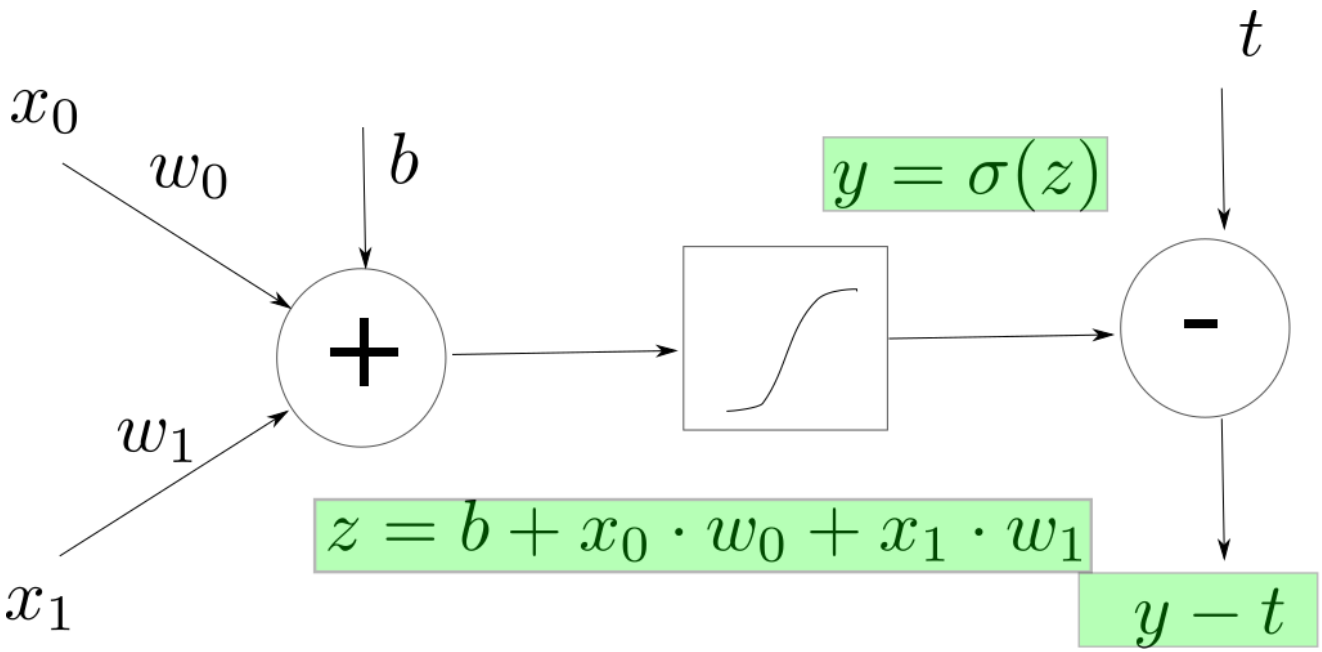
$$\frac{de}{dw_0} = x_0 \cdot \sigma(z) \cdot (1 - \sigma(z)) \cdot (y - t) \quad (37)$$

Which is much faster compared with calculating *error* twice.

What is the procedure?

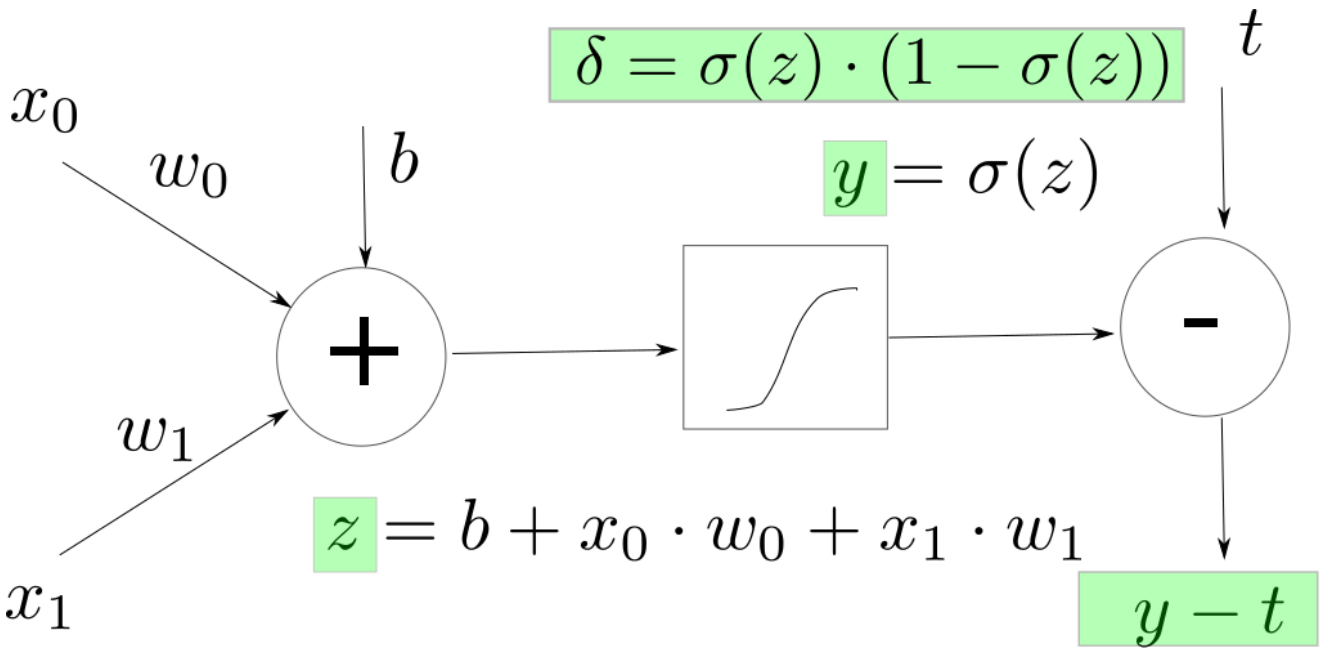


Propagate forward



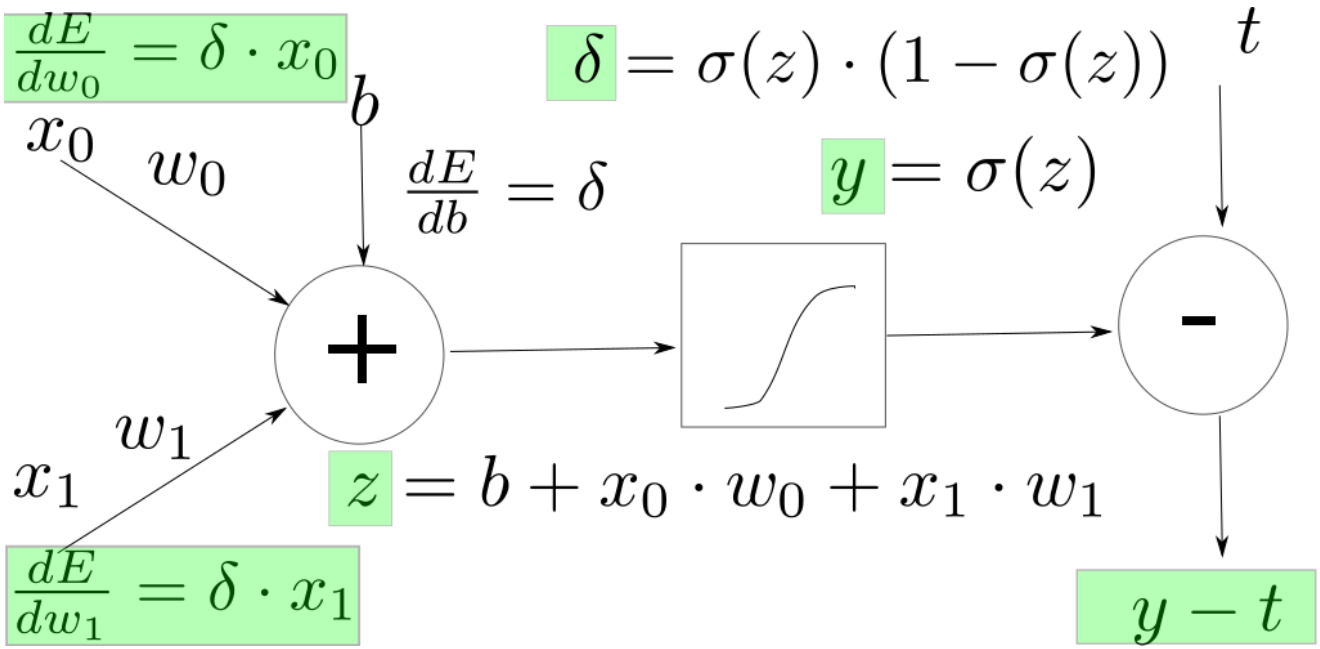
Calculate y,z and $y - t$. Remember values.

Calculate δ



Calculate $\delta = \sigma(z) \cdot (1 - \sigma(z))$.

Calculate δ



Calculate gradient: $\frac{dE}{dw_0} = \delta \cdot x_0$, $\frac{dE}{dw_1} = \delta \cdot x_1$. For bias it is $\frac{dE}{db} = \delta$ (third derivative term is 1). Now can make the step.

Delta - more important than it looks

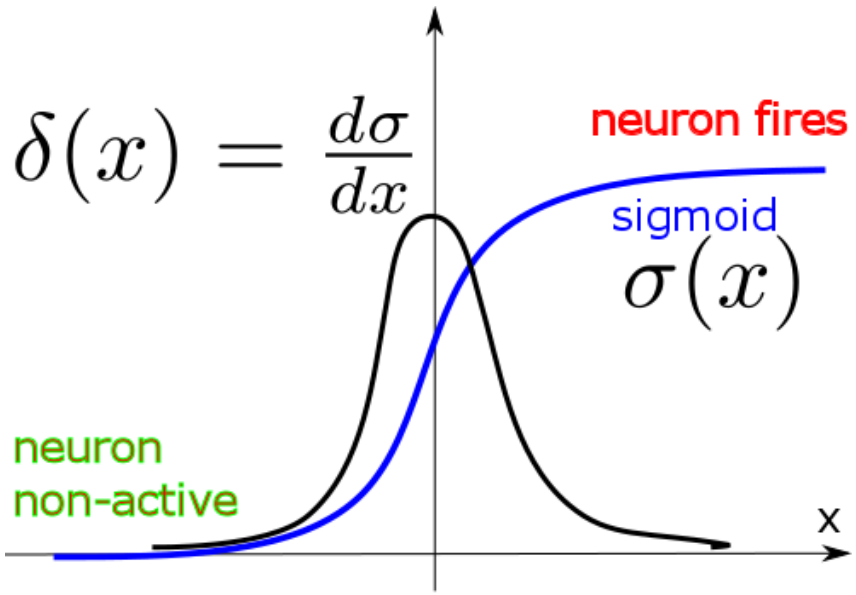


Figure:

- While ago it was mentioned that when NN settles neurons either fire or are dormant
- Middle region - neuron is un-certain
- $\delta(z)$ is **error** - only it is not output error,

Backpropagation in multi-layer networks

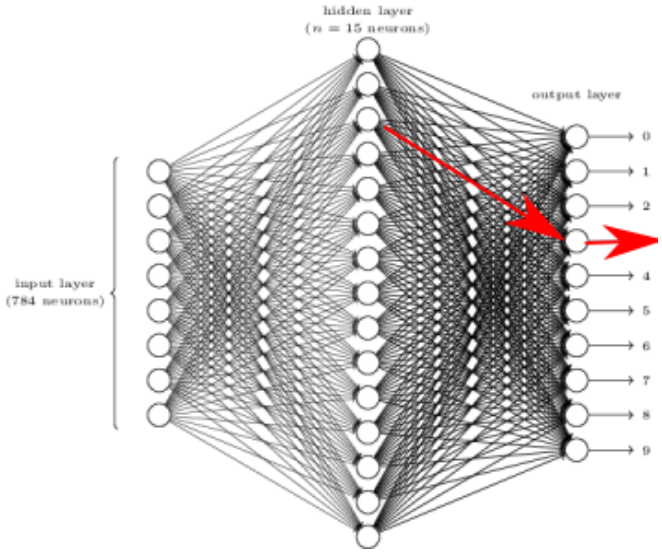


Figure: Changing one weight in output layer - not much changes in the network

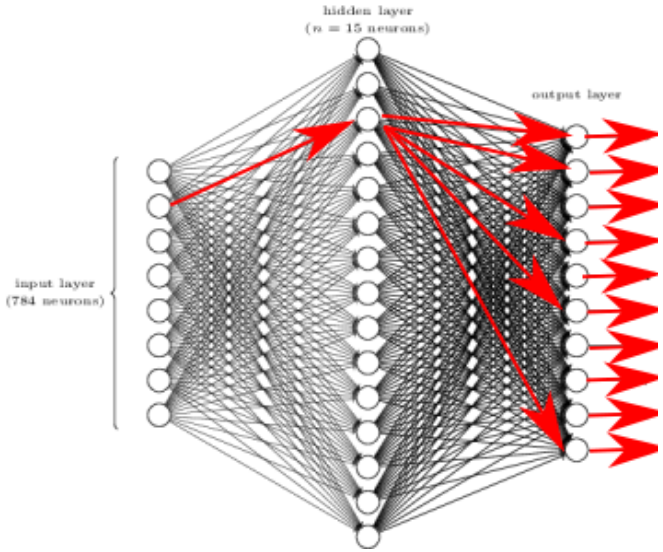


Figure: Changing one weight in hidden layer - a lot of changes

Backpropagation in multi-layer networks

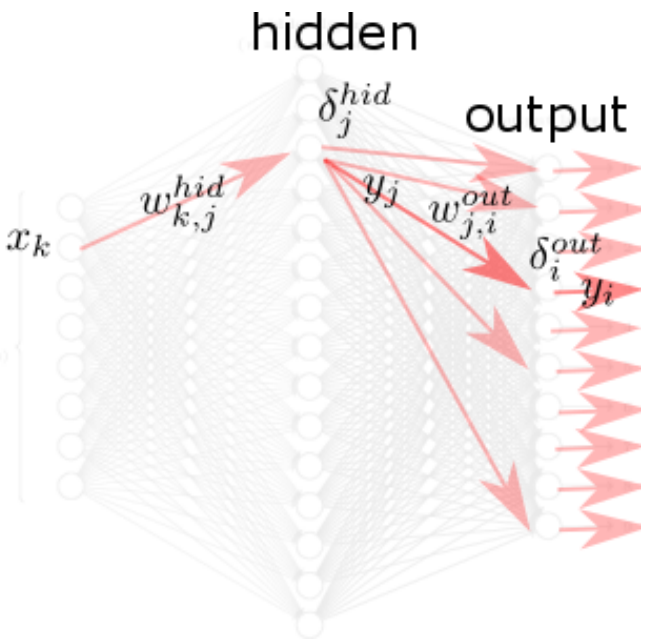


Figure:

Doing same as we did for hidden layer:

$$\frac{derroe}{dw_{k,j}^{hid}} = \frac{derror}{dy_j} \cdot \frac{dy_j}{dz_j} \cdot \frac{dz_j}{dw_{k,j}^{hid}}$$

After some derivations:

$$\delta_j = (\sum_i \delta_i \cdot w_{i,j}^{out}) \sigma(z_j) \cdot (1 - \sigma(z_j))$$

And so

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

$$\frac{derroe}{db_j^{hid}} = \delta_j$$

How state of the art are we?

We covered most of Neural Networks. Difference with top-notch networks is scale. Industrial networks have millions of neurons and many layers. AlphaGo, for example, is comprised of 3 NNs, 13 layers each. Uses 1202 processors.

Project

TBA by end of the week