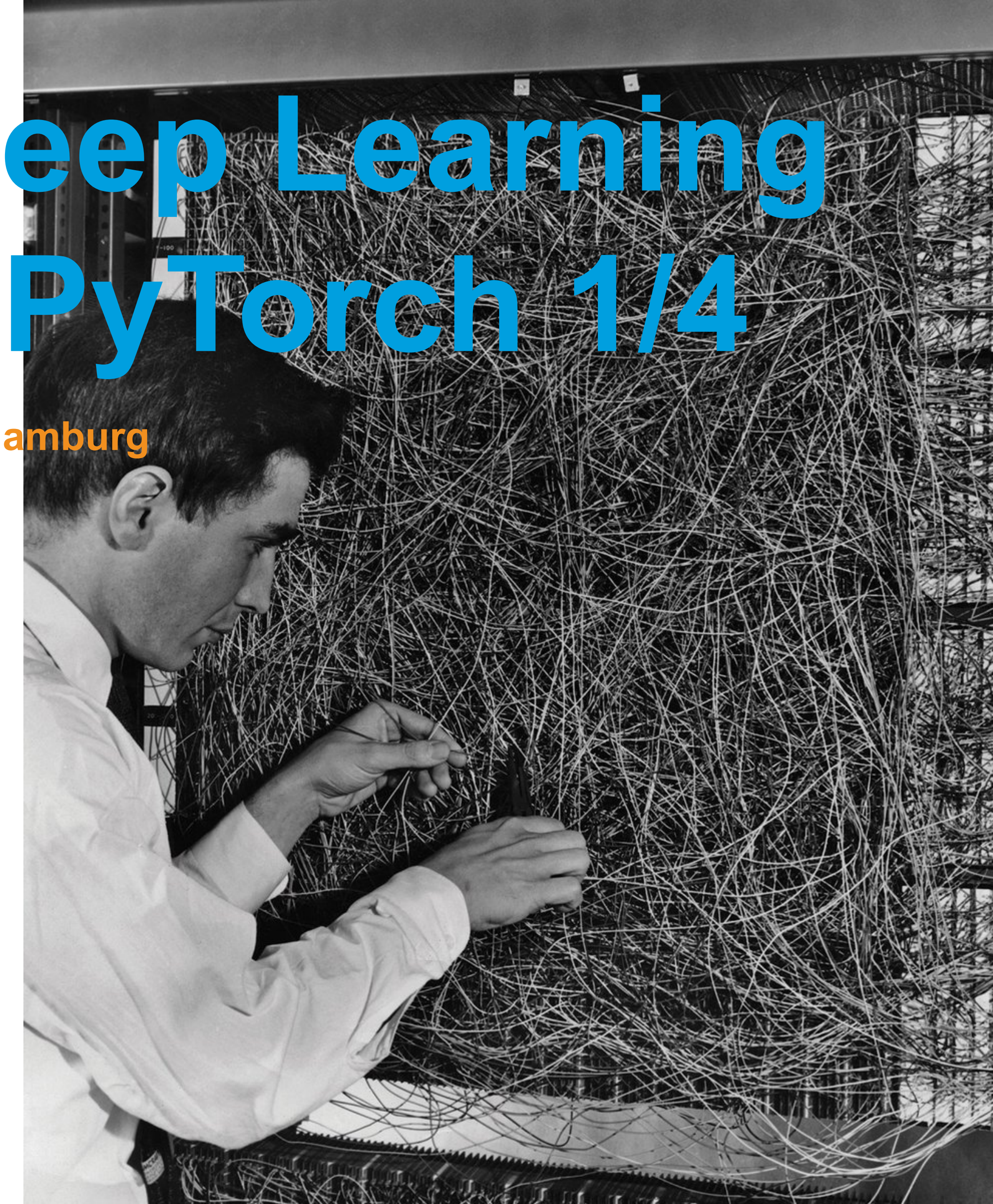
Introduction to Deep Learning with NumPy and PyTorch 1/4

1st Terascale Alliance Machine Learning School, DESY, Hamburg



Dirk Krücker
DESY - Hamburg, 23.10.2018

HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

General Overview

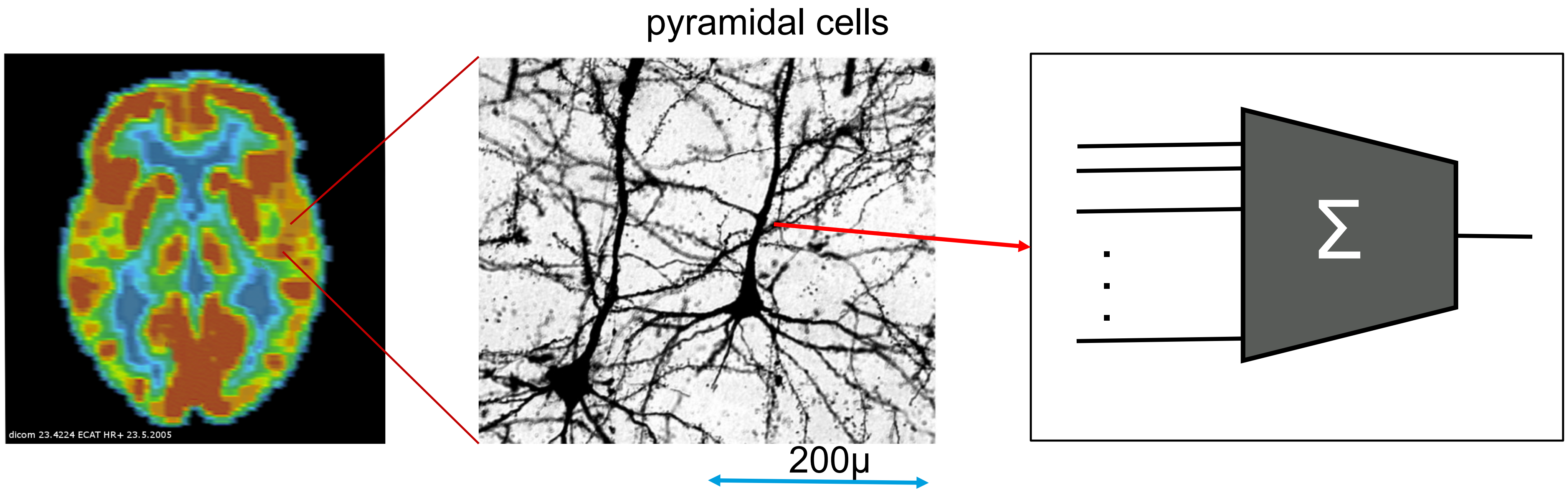
- I will try to cover basics, details you may not see so often. I believe that you must have seen some concepts at least once before you can forget this and ‘just’ use Deep Learning software in plug’n play way.
- There are underlined links in this talk where you can find additional information.

Content 1

- Introduction into Neural Networks
  - Neurons - A geometrical view
  - Activation functions
  - Basic operations
  - Universal approximation
  - Networks as Matrix multiplication
  - Loss functions I
  - Backpropagation and gradient descent
- Tutorial 1
  - Getting connected
  - Running a remote Jupyter notebook etc.

Me

I worked on brain imaging a while ago (PET)

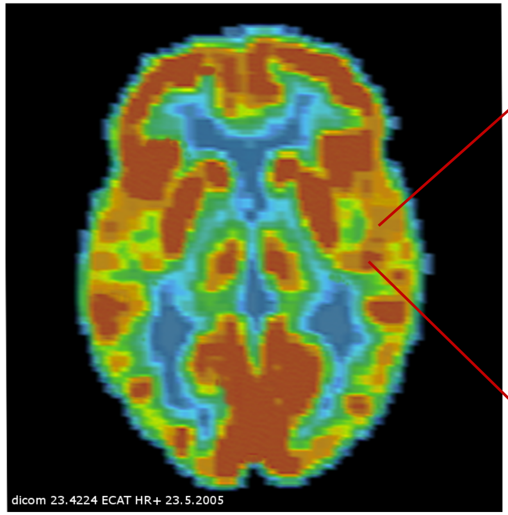


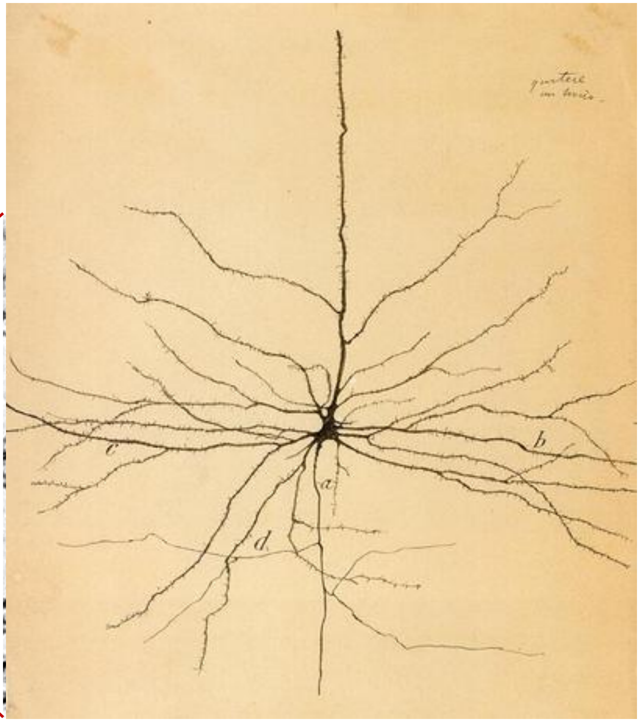
Neurons as computational units are an old idea.

- W. McCulloch and W. Pitts 1943 (sums and thresholds)
- D.O. Hebb 1940 (**learning by modified synaptic strength**)

Me

Some history





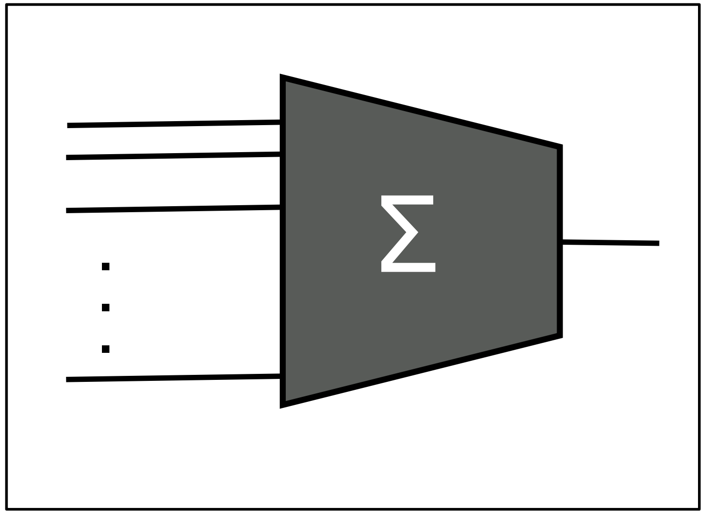
One of Cajal's drawing
from what he saw under the microscope
Golgi's method (1873): silver staining of neural tissue

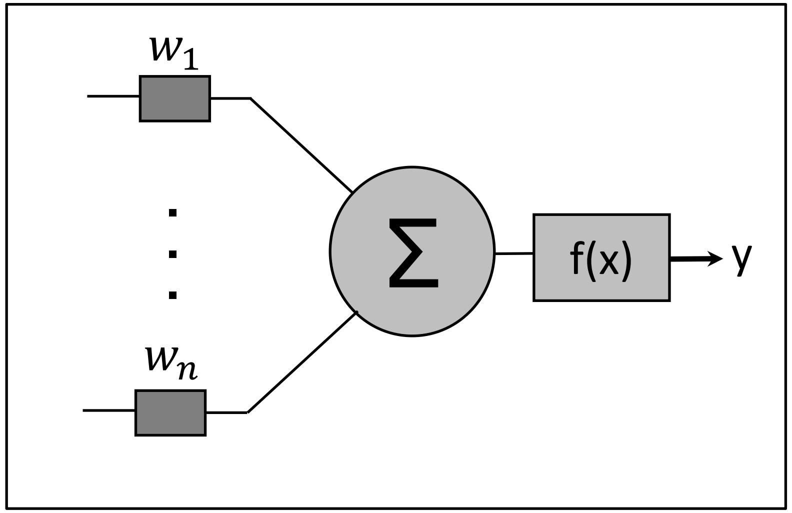
The idea that the neuron wiring is connected to learning is even older at least 120 years

- Cajal, the father of modern neuro science, had this idea in 1894 (learning by new connections).
  This is not yet the idea of computation.

Computation

Spelling out the mathematical model





$$y=f(\sum_{i=1}^{n} w_i\, xi)$$

Simple mathematical model

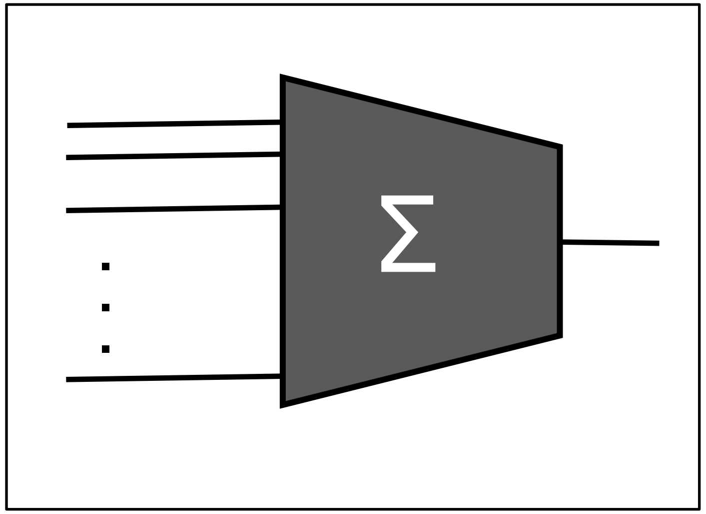
- Each input gets multiplied by a **weight**,
- then they are summed up
- and some function is applied.

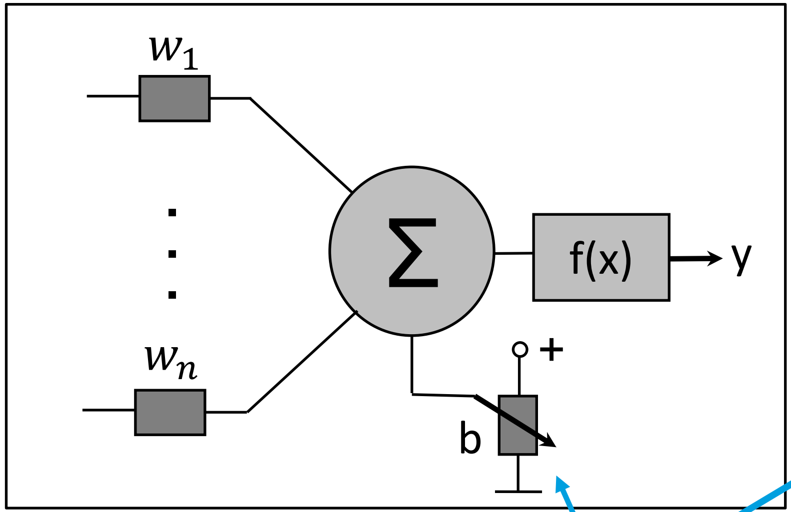
It emerged later that a **network** of such nodes has a rich structure

BTW, to simple as a biological model e.g. spikes, transmitters etc.

Computation

Bias term





$$y=\mathbf{f}\left(b + \sum_{i=1}^{n} w_i \, xi \right)$$

It will be useful to add a **bias** term i.e.
a constant term added as input

We have n inputs (real numbers) x_i , $i = 1 \ldots n$.
We assume that they form a vector space $\mathbb{R}^n$

Geometric interpretation ->

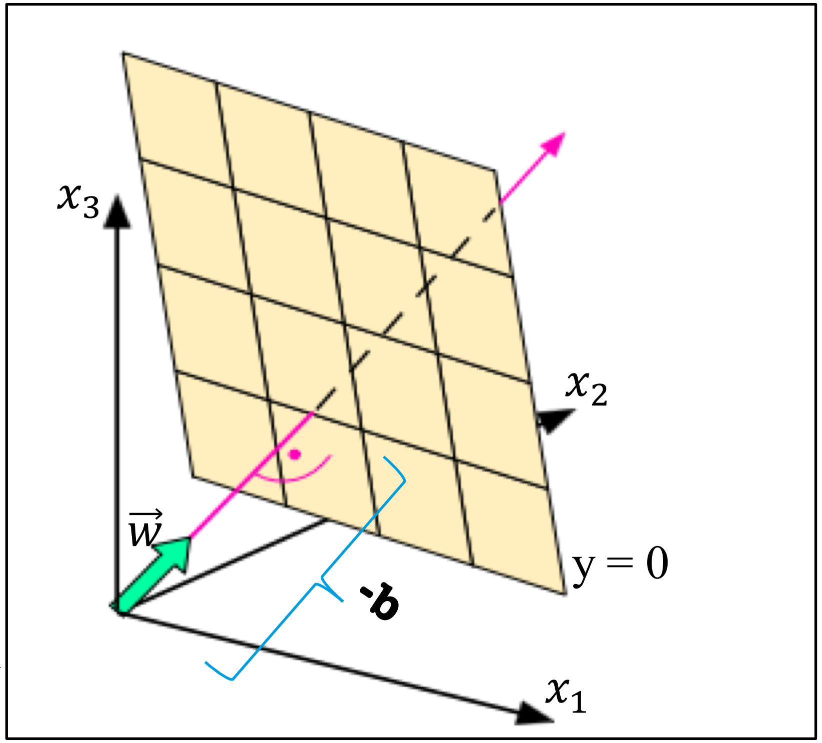
Geometry

Separating plane

$$y = b + \sum_{i=1}^{n} w_i \, xi$$

The same equation with vectors:

$$y = b + \vec{w}^{\mathrm{T}} \vec{x}$$



There is **a geometrical picture for this calculation**

- The inputs form a vector $\vec{x}$ and the weights a vector $\vec{w}$ (NB not normalized)
- y = 0 : defines a plane
- The neuron = the node represents a separating hyperplane cutting the space $\mathbb{R}^n$ into two halves

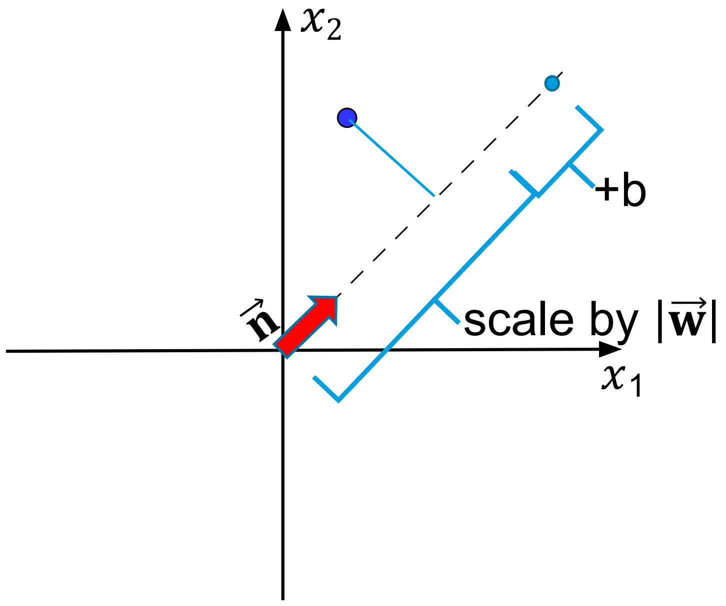
We assume that the input $\vec{x}$ forms an Euclidian Space (**distance**!).
This is neither trivial nor correct in general.

Geometry

1d projection

$$y = b + \vec{\mathbf{w}}^{\mathrm{T}}\vec{\boldsymbol{x}}$$
$$y = |\vec{\mathbf{w}}|(b/|\vec{\mathbf{w}}| + \vec{\mathbf{n}}^{\mathrm{T}}\vec{\mathbf{x}})$$
$$\text{with } \vec{\mathbf{n}} := \vec{\mathbf{w}}/|\vec{\mathbf{w}}|$$

- Direction is the normalized weight vector
- $|\vec{\mathbf{w}}|$ and b define scaling and a shift



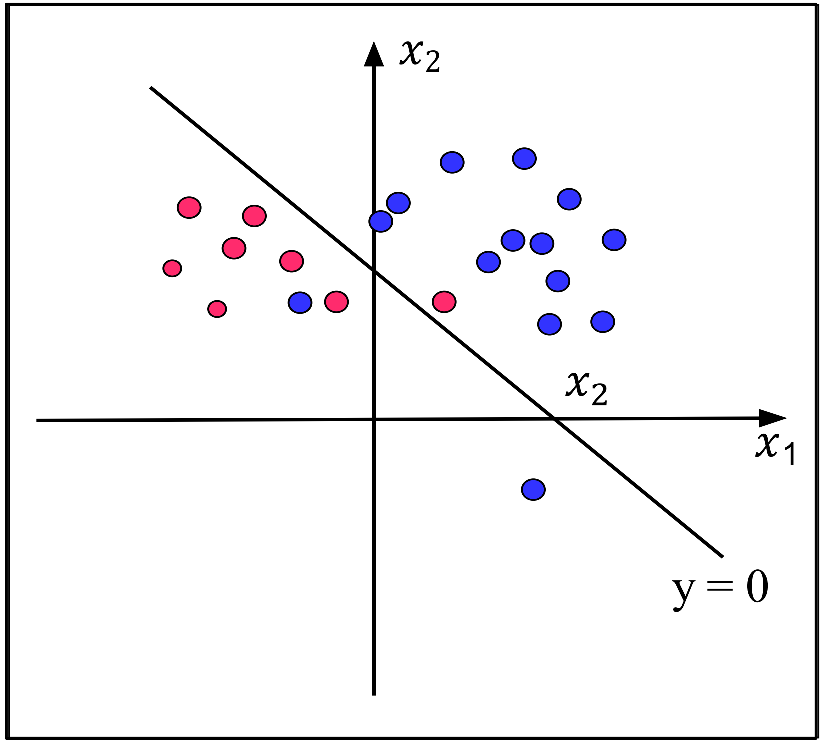
Perceptron

The first learning machine - Frank Rosenblatt 1957

Threshold:

$$\theta(x) := \begin{cases} 1 & if \quad x \geq 0 \\ 0 & else \quad x < 0 \end{cases}$$

$$\theta(b + \vec{w}^{\mathrm{T}}\vec{x})$$



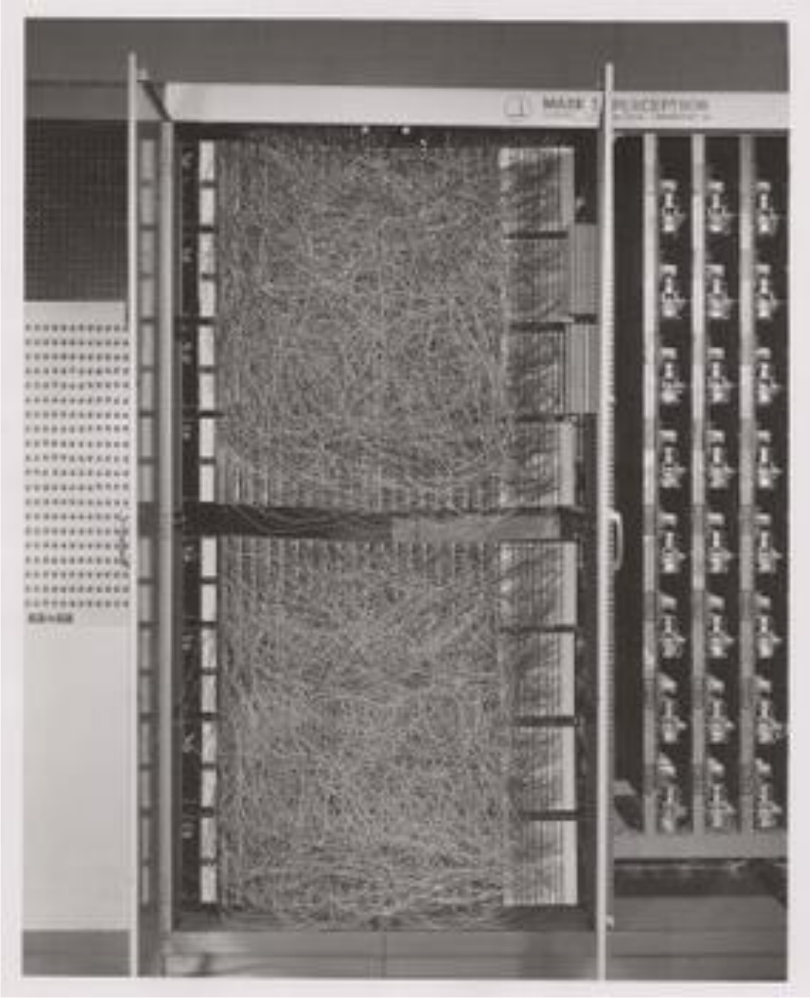
The Perceptron is one of the oldest ideas for machine learning and Artificial Intelligence. Cutting the input space into two halves. Learning means finding the best values for b and $\vec{w}$

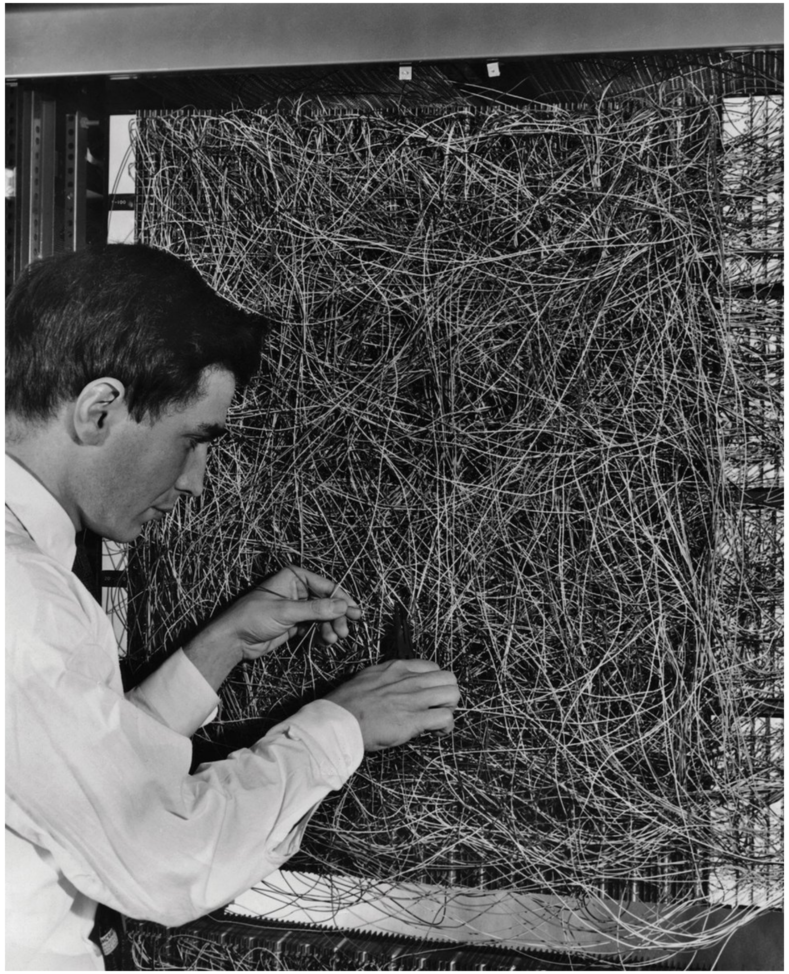
Mark I Perceptron

Some history

- This was really a machine with potentiometers steered by electric motors and a lot of wires
- The algorithm has been before implemented on an early IBM 704

An IBM 704 computer in 1957

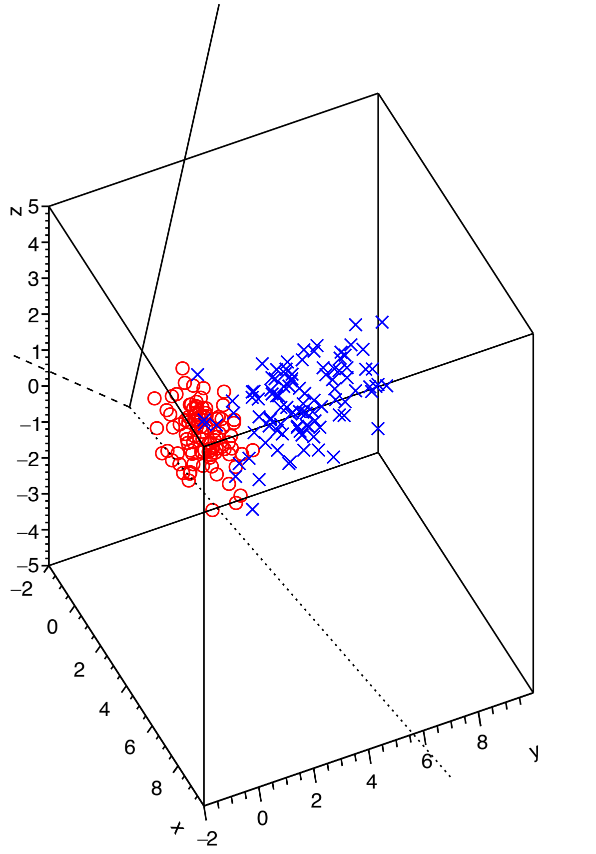


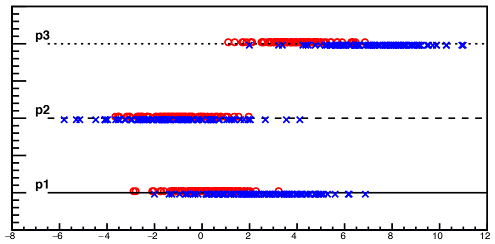


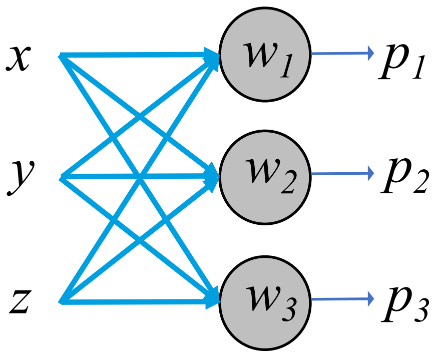
Already at that time we have the idea to use specialized hardware for NN

Aside: Random projection

Going to multiple nodes in a layer





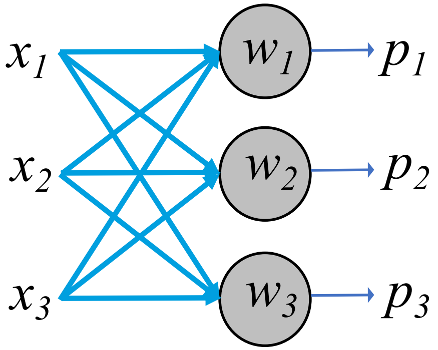


- Each node represents a projection N ->1
$$p_i = \overrightarrow{w_i}^{\mathrm{T}} \vec{x}$$
with its own weight vector and WOLG[1] $b_i = 0$
- **Multiple nodes form an N -> M mapping**
- (A high dimensional N to a low dimensional M may even work with a randomly chosen set of M axes if the data 'lives' on a low dimensional subspace (Johnson–Lindenstrauss lemma[1]))

[1] Without Loss Of Generality

Feedforward network

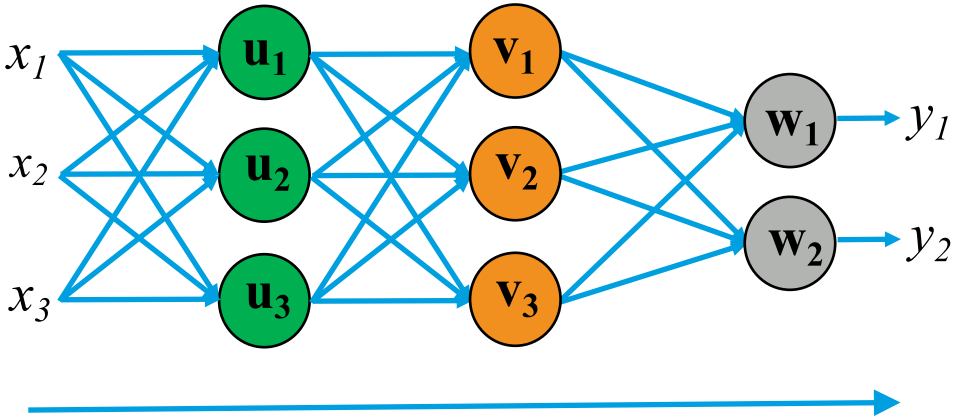
$$\begin{bmatrix} \vec{\mathbf{w}}_1^\intercal \\ \vec{\mathbf{w}}_2^\intercal \\ \vec{\mathbf{w}}_3^\intercal \end{bmatrix} \vec{\mathbf{x}} = \begin{bmatrix} (w_{11}, w_{12}, w_{13}) \\ (w_{21}, w_{22}, w_{23}) \\ (w_{31}, w_{32}, w_{33}) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \mathbf{W}\vec{\mathbf{x}} = \vec{\mathbf{p}}$$



- Each node represents a projection N ->1
  $$p_i = \overrightarrow{w_i}^{\mathrm{T}} \vec{x}$$
  with its own weight vector and WOLG $b_i = 0$
- **Multiple nodes form an N -> M mapping**
- The weight vectors of one layer can be considered as a Matrix **W**

Feedforward network

- Multiple nodes form an N -> M mapping
- The weight vectors of one layer can be considered as a Matrix **W**
- **Multiple layers form a chain of mappings**

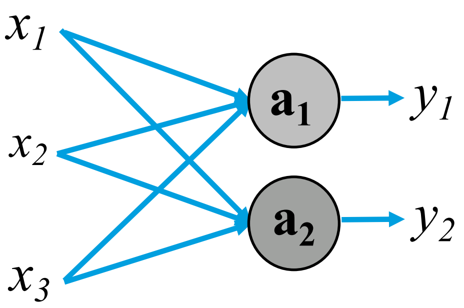


$$\underbrace{\vec{\mathbf{y}}}_{2\times1} = \overbrace{\mathbf{W}}^{2\times3}\underbrace{\mathbf{V}}_{3\times3}\overbrace{\mathbf{U}}^{3\times3}\underbrace{\vec{\mathbf{x}}}_{3\times1}$$

But this is all trivial since ...

Feedforward network

- Multiple nodes form an N -> M mapping
- The weight vectors of one layer can be considered as a Matrix **W**
- **Multiple layers form a chain of mappings**



$$\underbrace{\vec{\mathbf{y}}}_{2\times 1} = \overbrace{\mathbf{A}}^{2\times 3}\underbrace{\vec{\mathbf{x}}}_{3\times 1}$$

... due to **linearity** this all collapses into one mapping. The fun starts when we include some non-linearity

Nonlinearity

Wrapping a function around

$$y=\mathbf{f}\left(b + \sum_{i=1}^{n} w_i \, xi \right)$$

All nonlinearity comes from the mapping function

In general the activation function **f** can be chosen arbitrarily but it has a **strong** influence on the behavior of the resulting NN

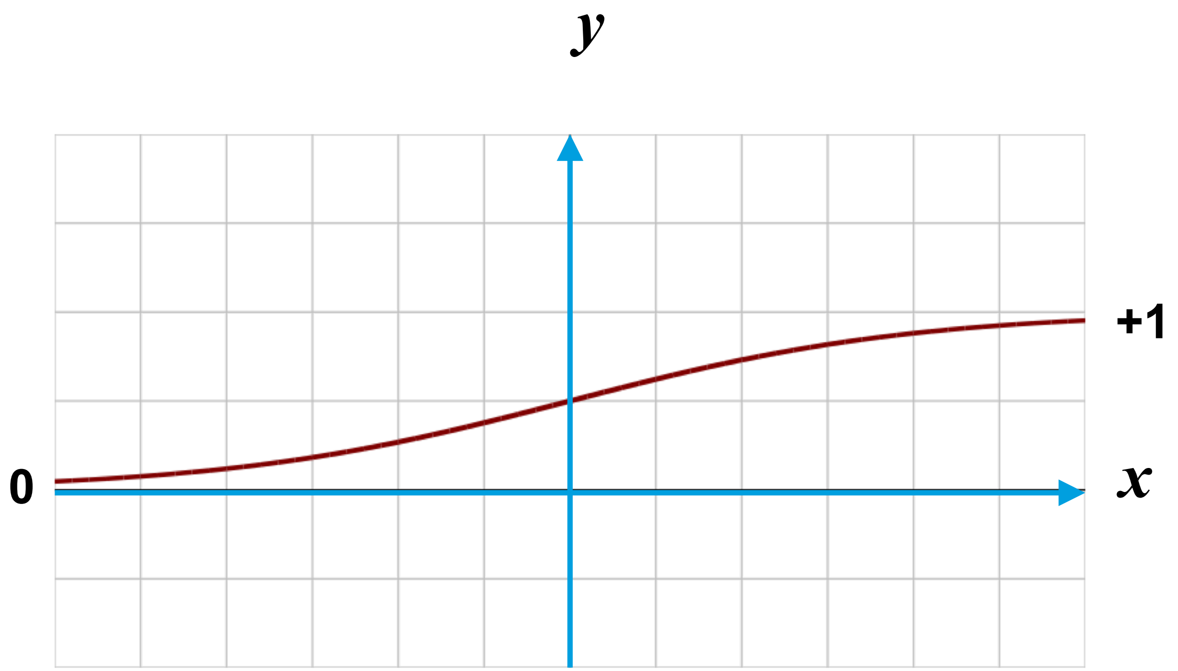
Nonlinearity – activation functions

- Identity $f(x) = x$
- Simple threshold (classic NN) $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
- **Sigmoid** (logistic function)
- **Tanh**
- **ReLU** (REctified Linear Unit)
- Leaky ReLU
- RBF (see SVM)
- Softmax
- ... https://en.wikipedia.org/wiki/Activation_function

Nonlinearity – activation functions

- Identity
- Simple threshold (classic NN)
- **Sigmoid** (logistic function)
- **Tanh**
- **ReLU** (REctified Linear Unit)
- Leaky ReLU
- RBF (see SVM)
- Softmax
- ...

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

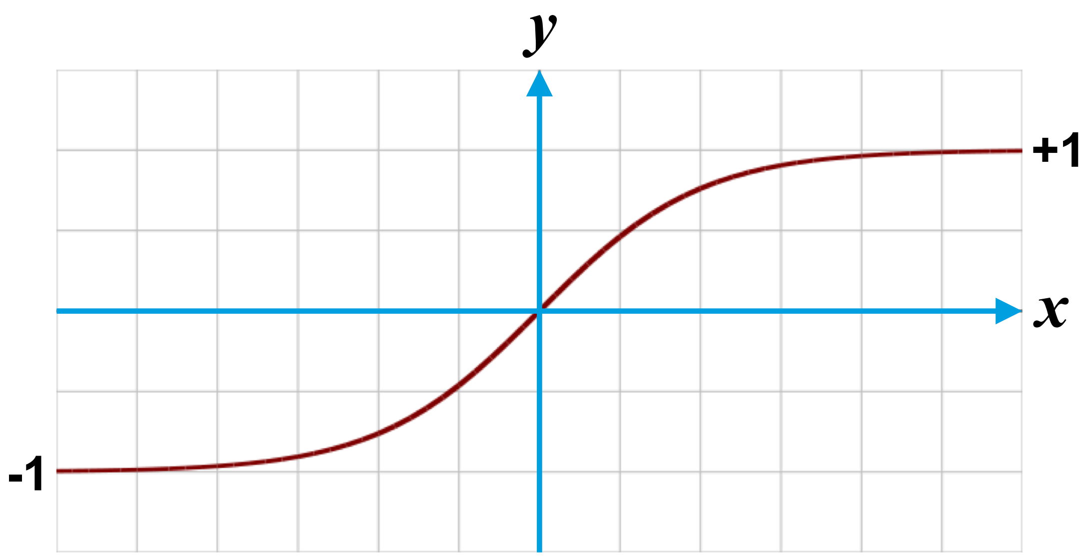


- Maps between 0 and 1
- Useful for probability model e.g output node
- Not linear at origin

https://en.wikipedia.org/wiki/Activation_function

Nonlinearity – activation functions

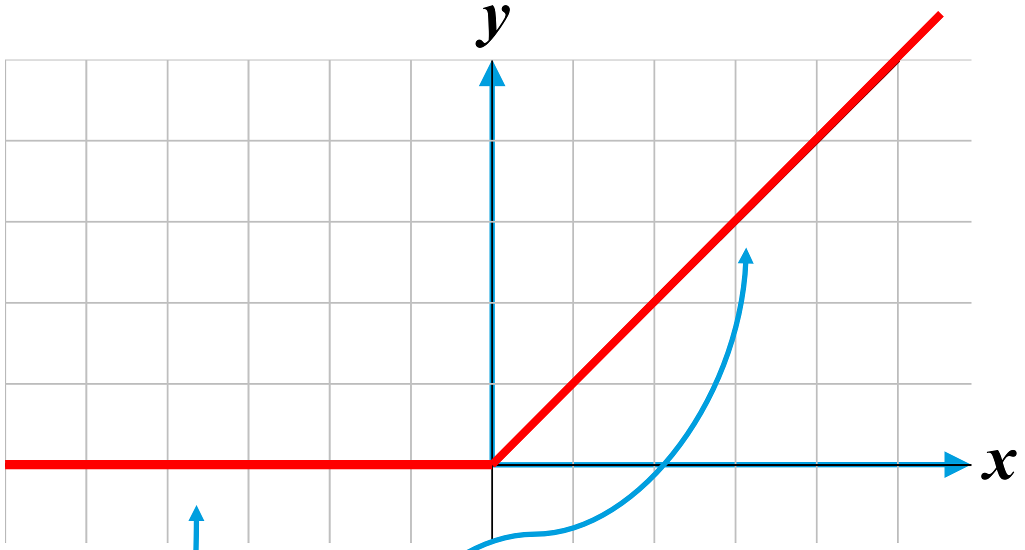
- Identity
- Simple threshold (classic NN)
- **Sigmoid**
- **Tanh** $$f(x) = \tanh(x)$$
- **ReLU** (REctified Linear Unit)
- Leaky ReLU
- RBF (see SVM)
- Softmax
- … https://en.wikipedia.org/wiki/Activation_function



- Maps between -1 and 1 (or 0 and 1)
- Useful for classification e.g output node
- Use to be popular in 1st generation NN
- local (see later)
- Linear at origin

Nonlinearity – activation functions

- Identity $f(x) = x$
- Simple threshold (classic NN) $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
- **Sigmoid**
- **Tanh**
- **ReLU** (REctified Linear Unit) $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
- Leaky ReLU
- RBF (see SVM)
- Softmax
- … https://en.wikipedia.org/wiki/Activation_function

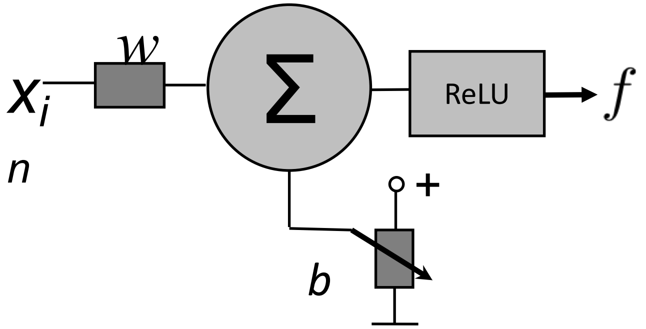


- Threshold behaviour
  - 0 below threshold
  - Identity above threshold
  - Unrestricted growth
- Popular choice now

ReLU as a cut in 1D

- The condition
  defines a cut
  on the input variable x

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

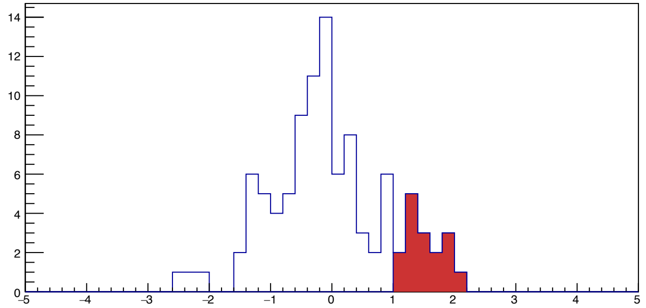


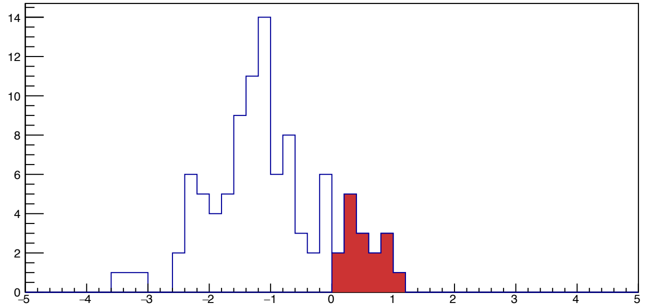
- The cut on x is defined by the weight w and the bias b

- Instead of cutting on $x_{in} > x_{cut}$
  we scale x_{in} by w and shift the input distribution by b:

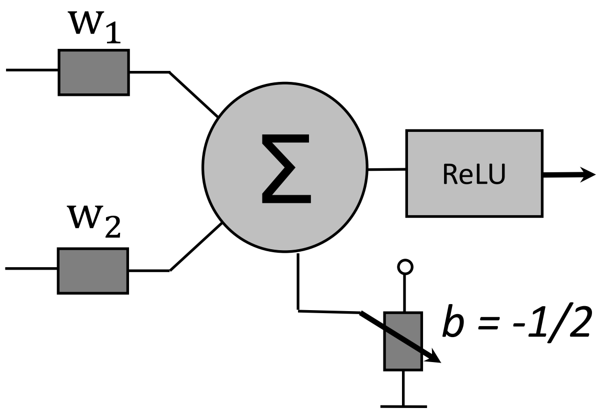
$$f(w\, x_{in} + b)$$

- *"we move the distribution to the cut"*





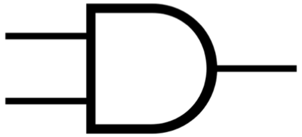
ReLU as a logic gate



All HEP cutflows can be realize by ReLU networks

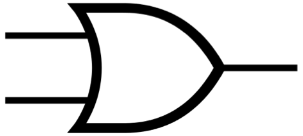
negative weight -> logic not

$w_1 = w_2 = 1/2$



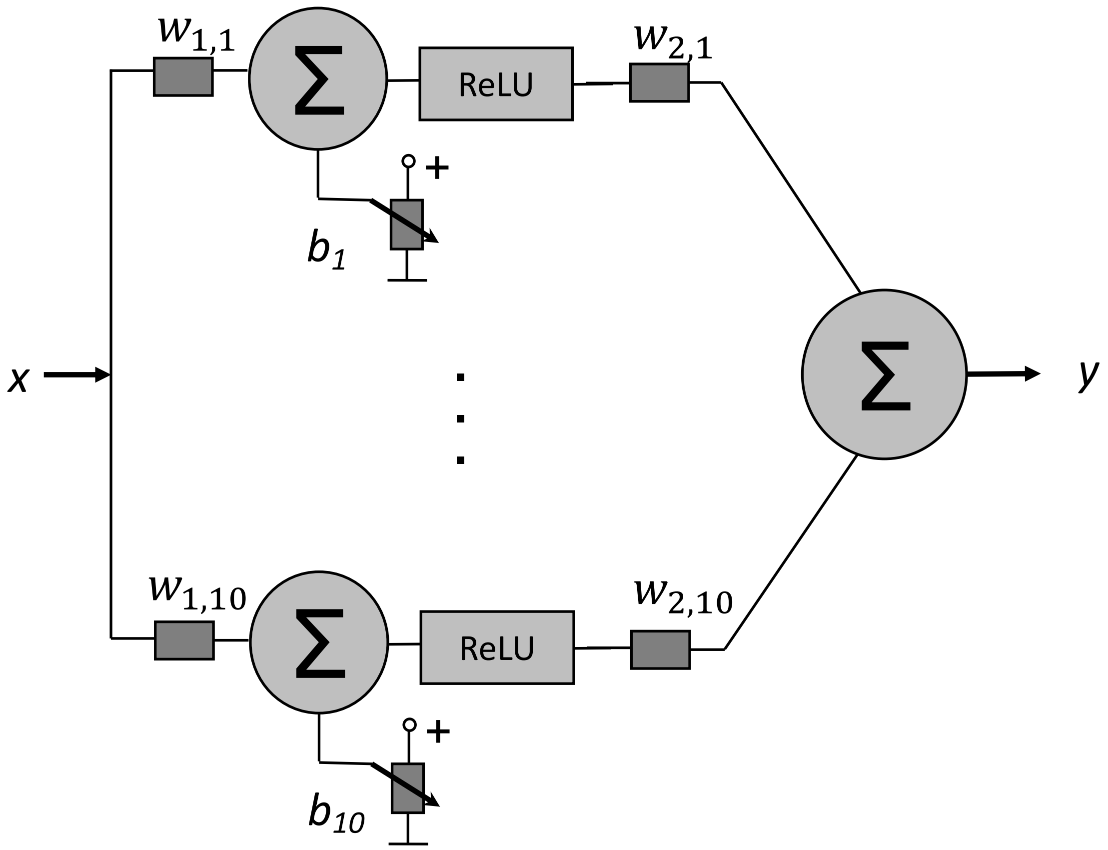
x	y	Σ	y
0	0	0	0
1	0	1/2	0
0	1	1/2	0
1	1	1	1

$w_1 = w_2 = 1$



x	y	Σ	y
0	0	0	0
1	0	1	1
0	1	1	1
1	1	2	1

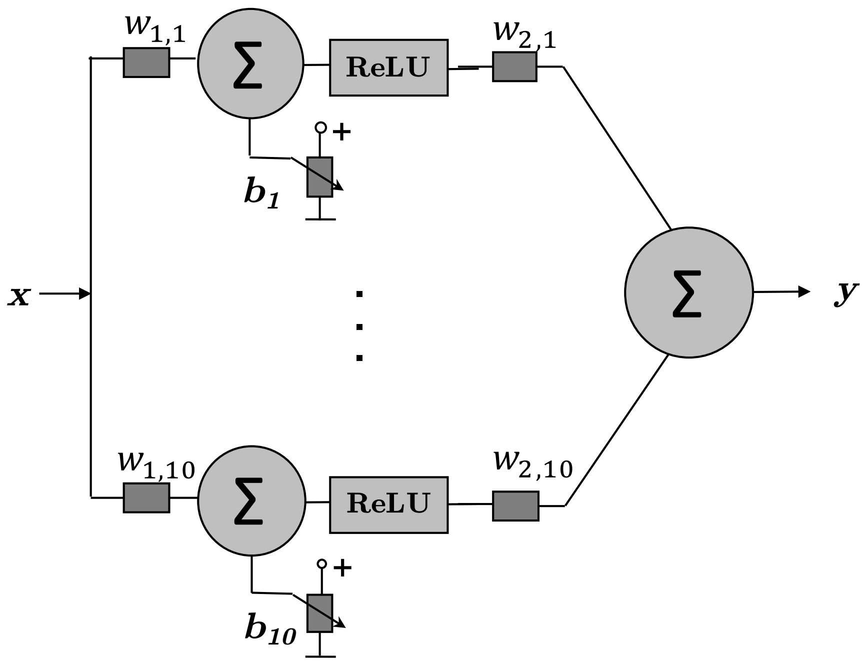
Computation II – universal approximation

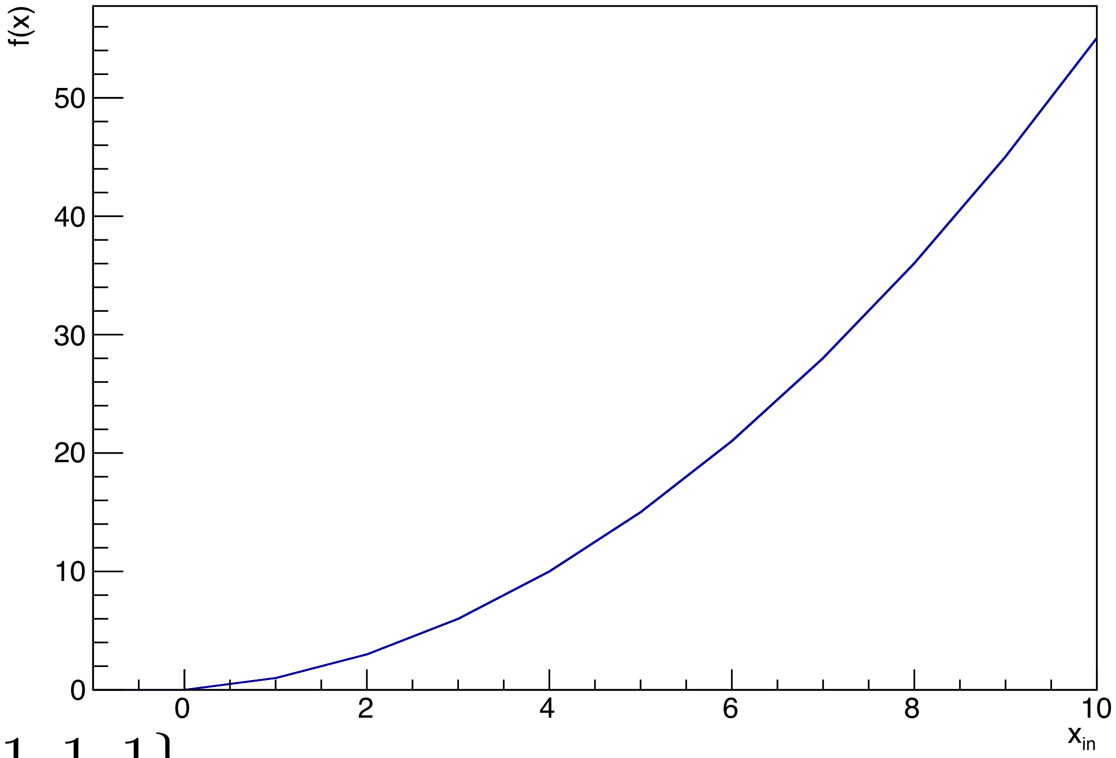


- Nodes with one input x and one output y
- We get a piecewise linear function that can model an arbitrary function.
- Intuitively clear[1] that **one hidden layer with sufficient many nodes can approximate any smooth function** to a given accuracy ε

[1] https://en.wikipedia.org/wiki/Universal_approximation_theorem

Computation II – universal approximation





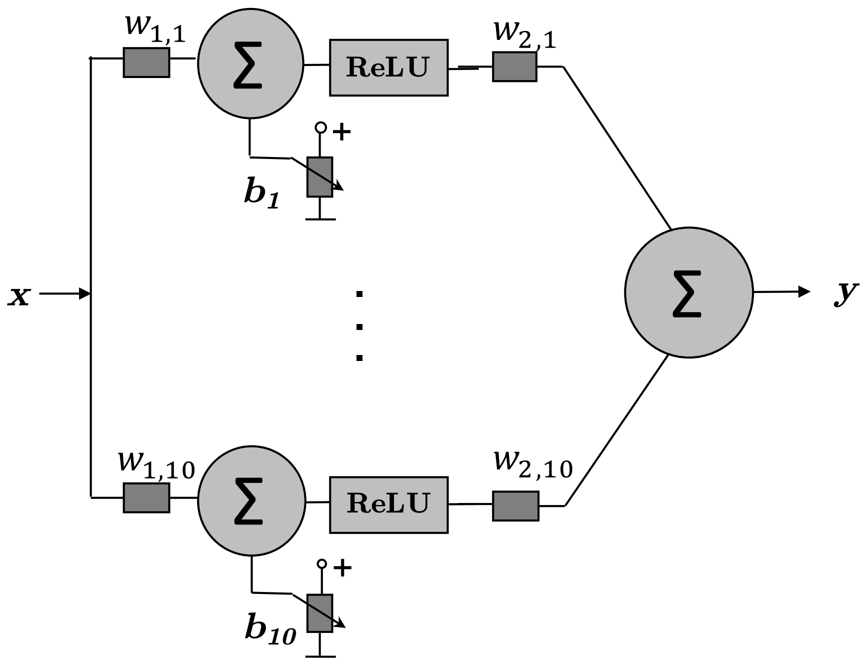
$$\{w_{1,1},\ldots,w_{1,10}\} = \{1,1,1,1,1,1,1,1,1,1\}$$
$$\{w_{2,1},\ldots,w_{2,10}\} = \{1,1,1,1,1,1,1,1,1,1\}$$
$$\{b_1,\ldots,b_{10}\} = \{0,-1,-2,-3,-4,-5,-6,-7,-8,-9\}$$

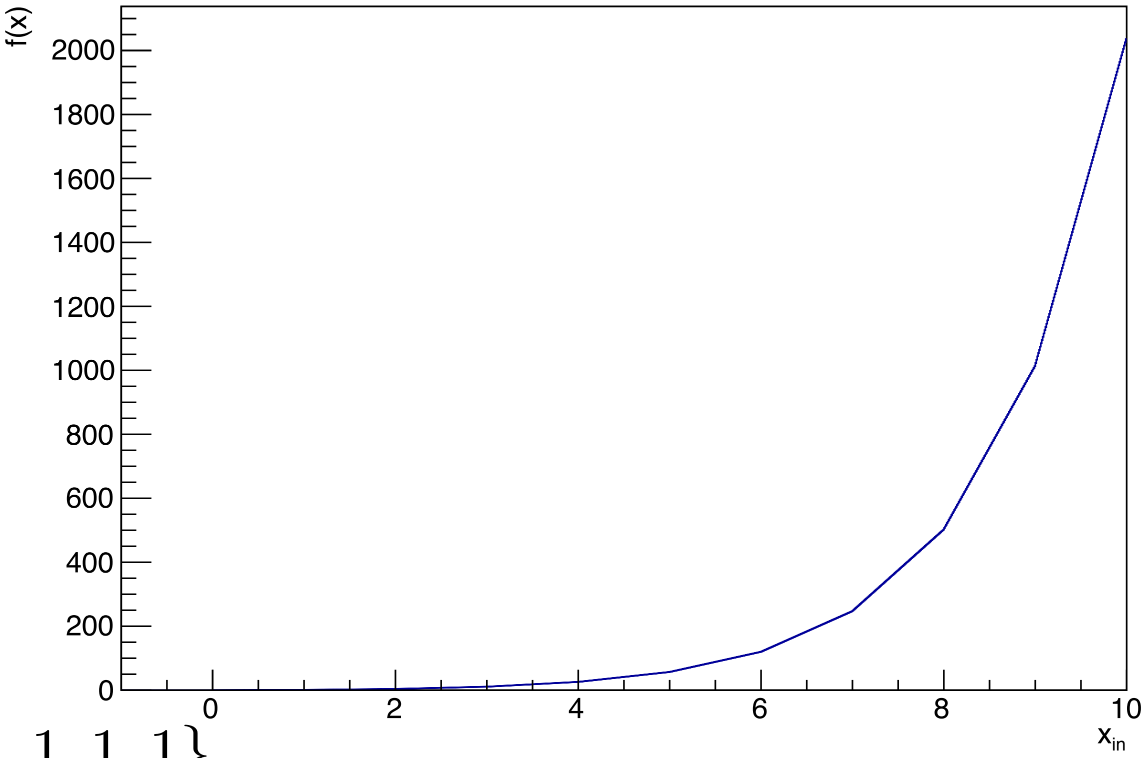
$$f(x_{in}) \approx x^2$$

1 layer of ReLU presents a piecewise linear approximation

[1] https://en.wikipedia.org/wiki/Universal_approximation_theorem

Computation II – universal approximation





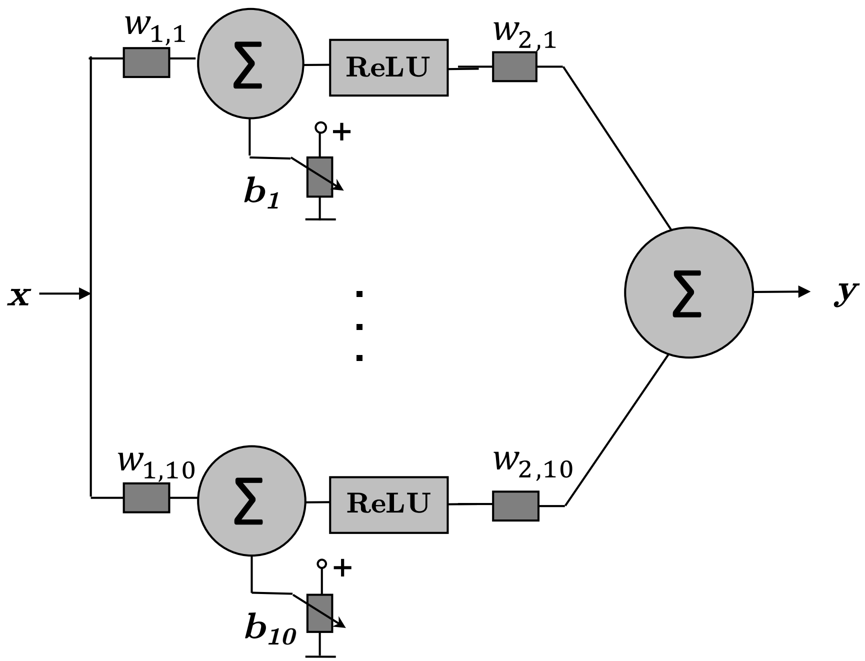
$$\{w_{1,1},\ldots,w_{1,10}\} = \{1,1,1,1,1,1,1,1,1,1\}$$
$$\{w_{2,1},\ldots,w_{2,10}\} = \{1,2,4,8,16,32,64,128,256,512\}$$
$$\{b_1,\ldots,b_{10}\} = \{0,-1,-2,-3,-4,-5,-6,-7,-8,-9\}$$

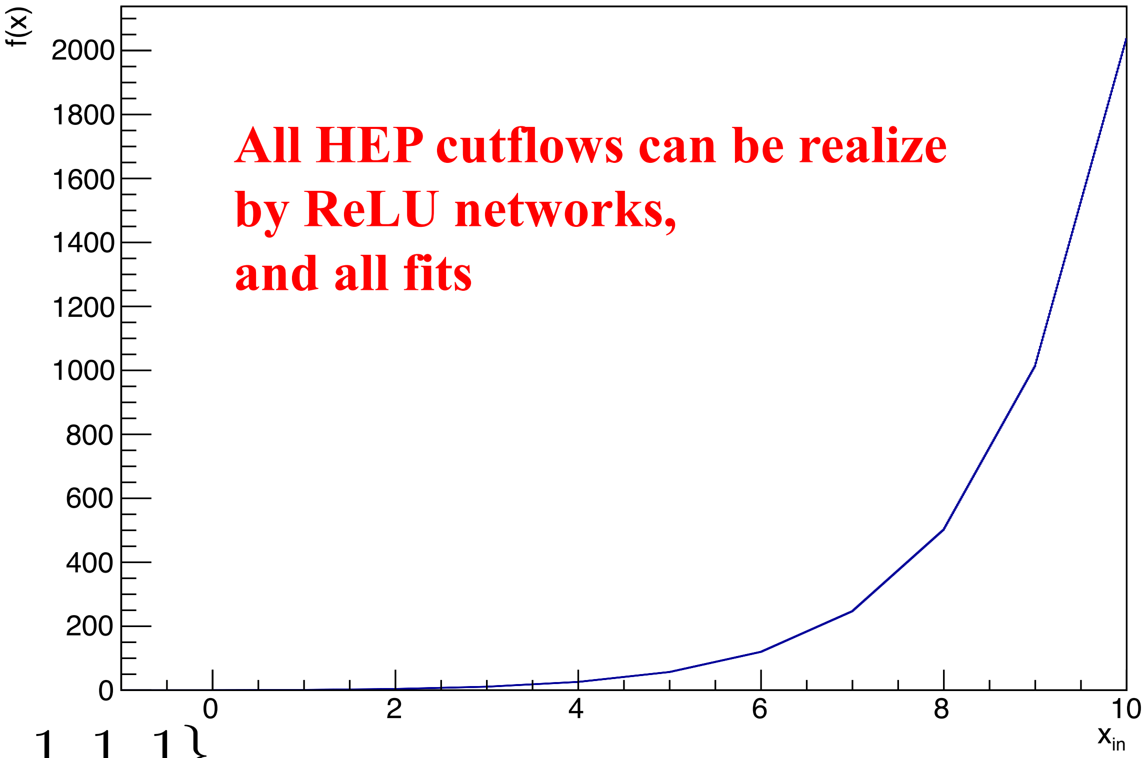
$$f(x_{in}) \approx 2^x$$

1 layer of ReLU presents a piecewise linear approximation

[1] https://en.wikipedia.org/wiki/Universal_approximation_theorem

Computation II – universal approximation





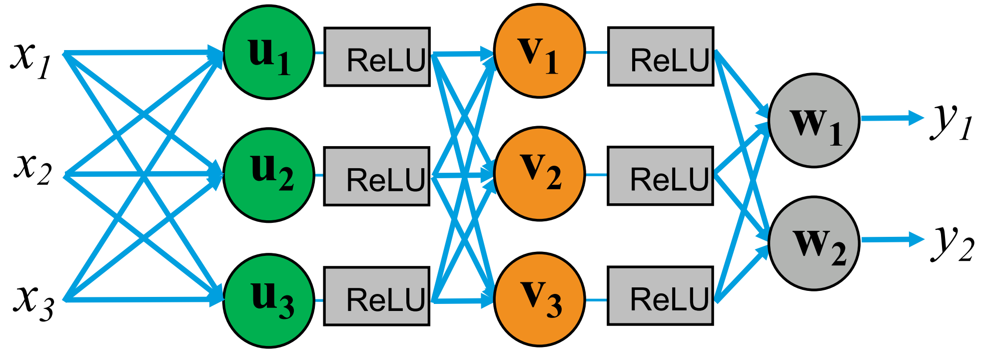
$$\{w_{1,1}, \ldots, w_{1,10}\} = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$$
$$\{w_{2,1}, \ldots, w_{2,10}\} = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$$
$$\{b_1, \ldots, b_{10}\} = \{0, -1, -2, -3, -4, -5, -6, -7, -8, -9\}$$

$$f(x_{in}) \approx x^2$$

1 layer of ReLU presents a piecewise linear approximation

[1] https://en.wikipedia.org/wiki/Universal_approximation_theorem

ReLU Feedforward network



The parentheses become nested

$$\underbrace{\vec{\mathbf{y}}}_{2\times 1} = \overbrace{\mathbf{W}}^{2\times 3} F(\underbrace{\mathbf{V}}_{3\times 3} \underbrace{F(\mathbf{U}\vec{\mathbf{x}})}_{3\times 1})$$

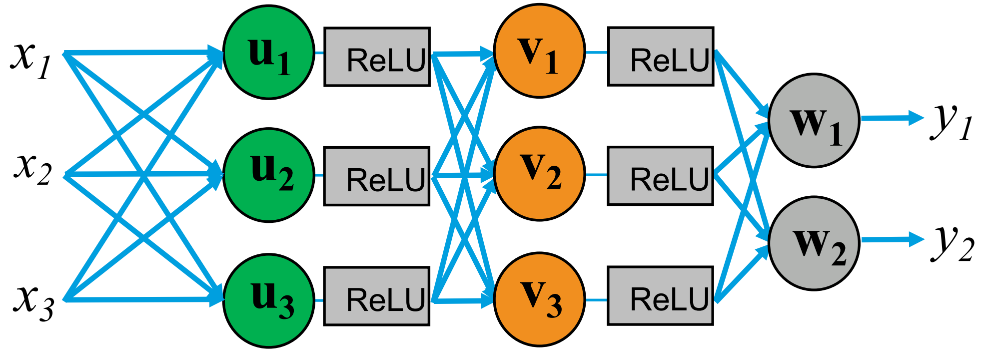
with $F(\vec{\mathbf{x}}) := \left[F(x_i)\right]$

i.e. F applied to each component of $\vec{\mathbf{x}}$

and F=ReLU

Now we can go **deep!** but
we better write this with tensors

ReLU Feedforward network



The parentheses become nested

$$y_i = w_i{}^j F(v_j{}^k F(u_k{}^l x_l))$$

with $i = 1, 2$ and $j, k, l = 1, 2, 3$
sum convention assumed

That's why it is called TensorFlow™
In principal Graph, Matrix and Tensor notation are equal but conventional Tensor notation is most general!

What kind of structure can be approximated with a multilayer ReLU network?
This is related to the question how many linear areas can be described.
(As before in the universal approximation example we have linear areas)

Cutting hyperplanes II

- How many independent areas do we get with n hyperplanes?
- Dimension d and the number of different hyperplanes s
- This is also the maximum number of linear units with s ReLU units within one layer

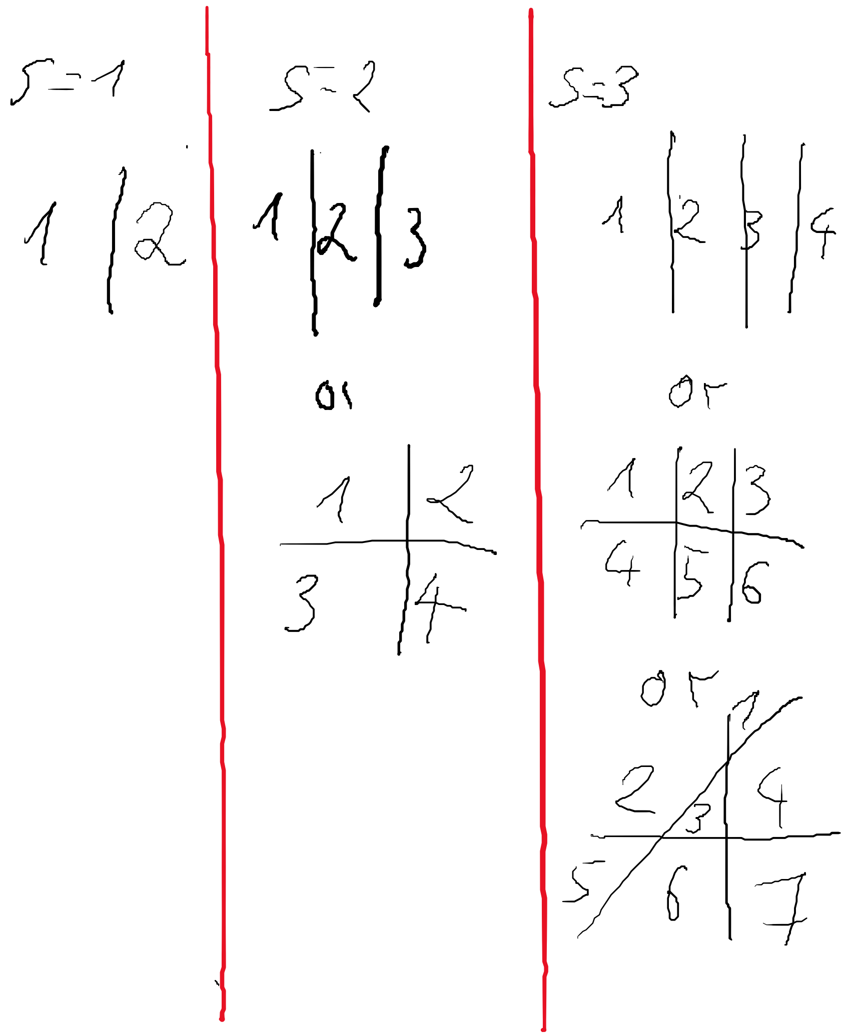
d = 2 (paper plane)

s = # hyperplanes (lines)

At least $s+1$

At most

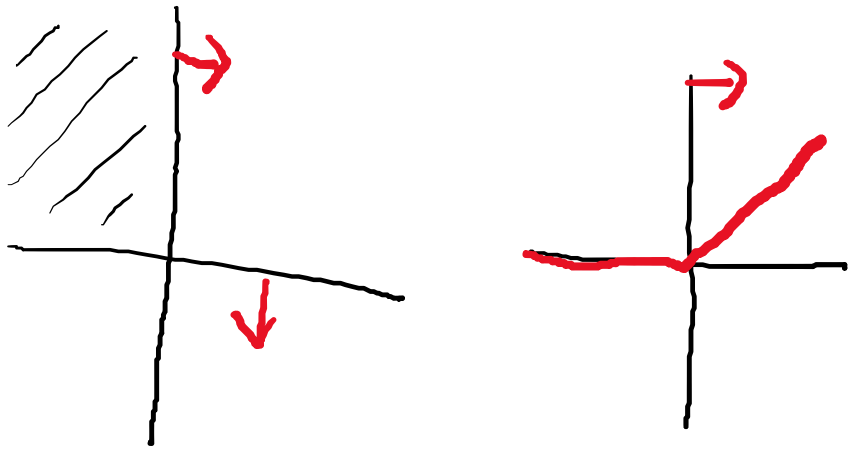
$$\sum_{n=0}^{d} \binom{s}{n}$$



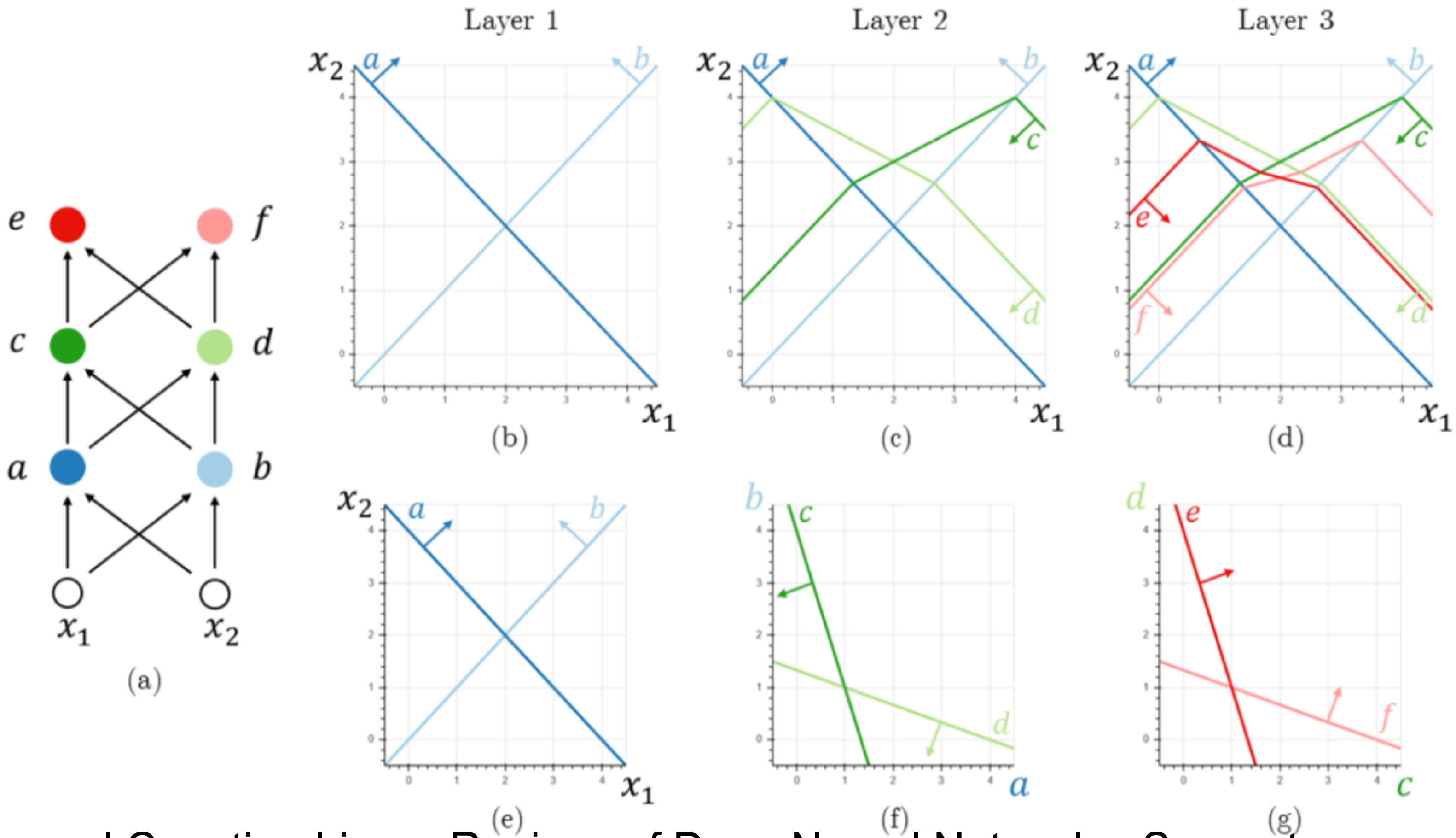
10 cuts in 2d => 56 pieces
40 cuts in 2d => 821
10 cuts in 10d => 1024
40 cuts in 10d => 1221246132

Cutting hyperplanes II

- How many independent areas do we get with n hyperplanes?
- Dimension d and the number of different hyperplanes s
- This is also the maximum number of linear units with s ReLU units within one layer



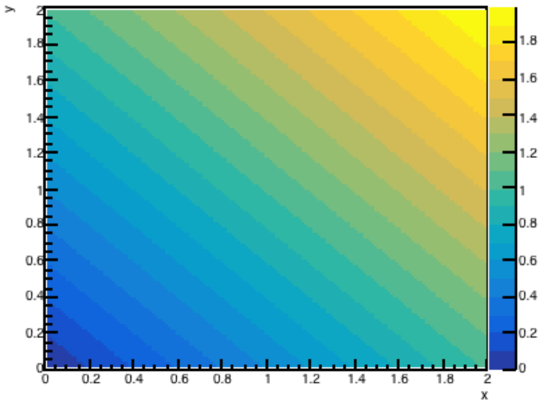
Cutting hyperplanes II - DL



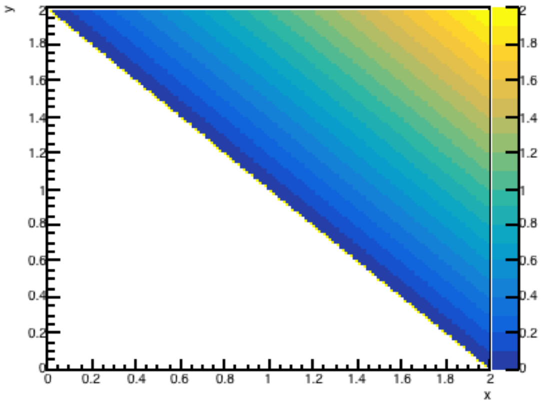
Bounding and Counting Linear Regions of Deep Neural Networks, Serra et al., https://arxiv.org/abs/1711.02114

x, y <-[0,2]
z=0.5(x+y)

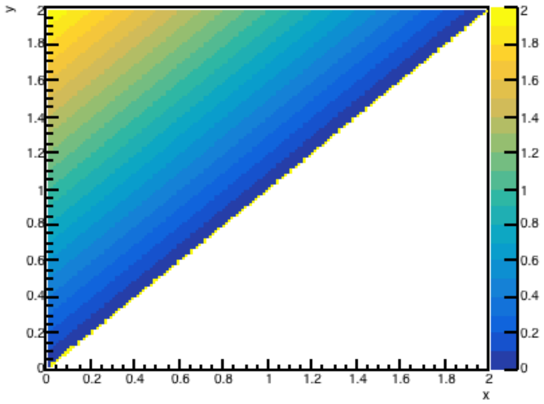
z = 0.5*(x + y) in x,y



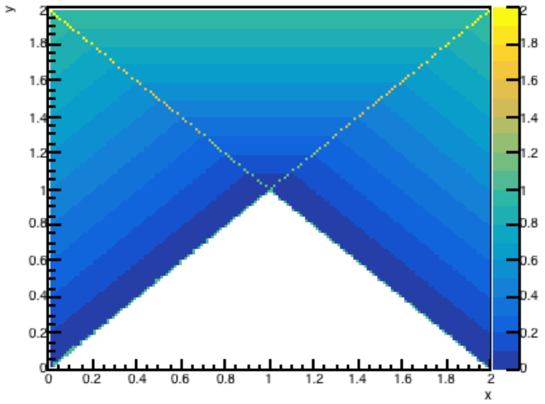
a=ReLU(x+y-2)



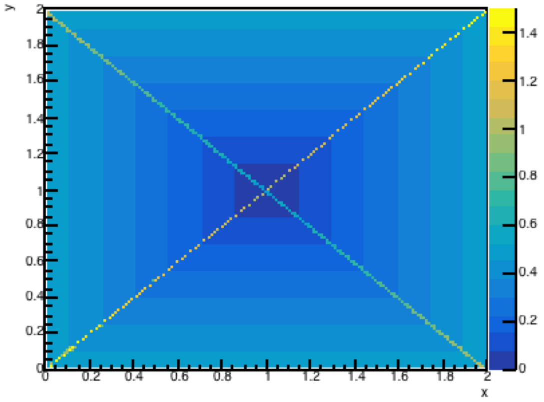
a=ReLU(-x+y)



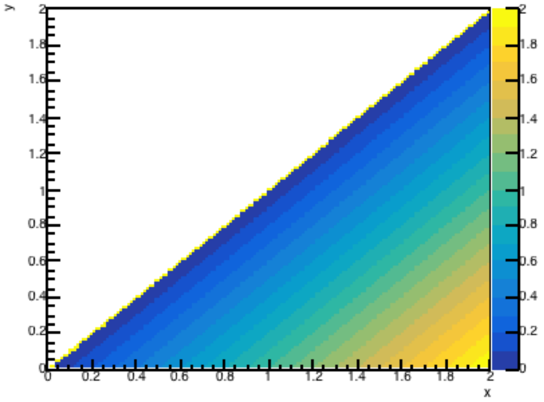
z = 0.5*(a + b) in x,y



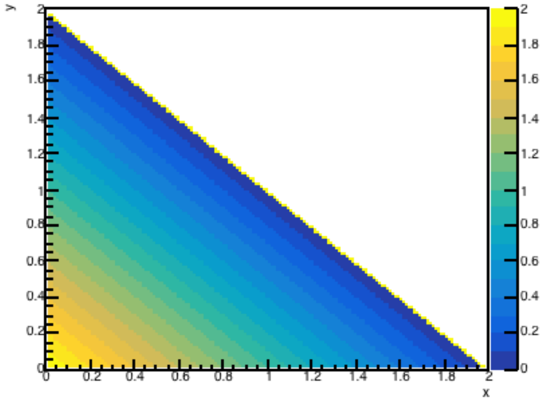
z = 0.25*(a + b + c + d) in x,y



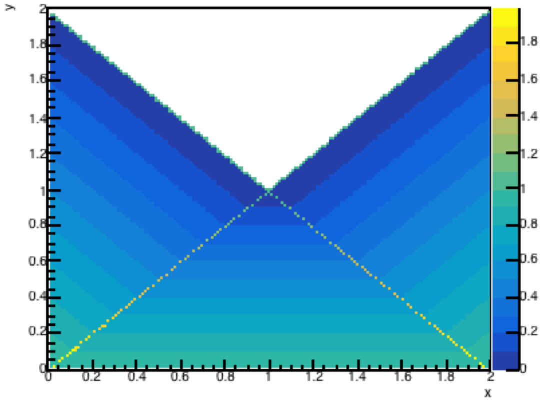
a=ReLU(x-y)

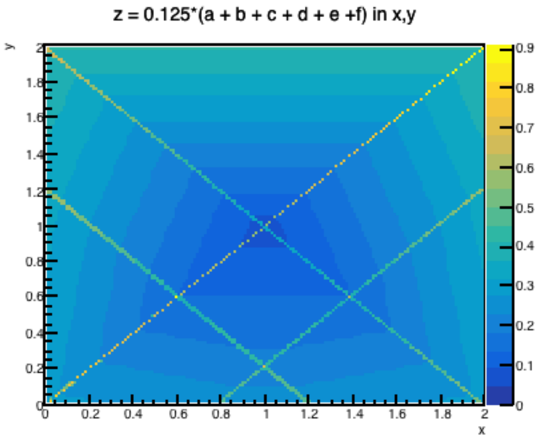


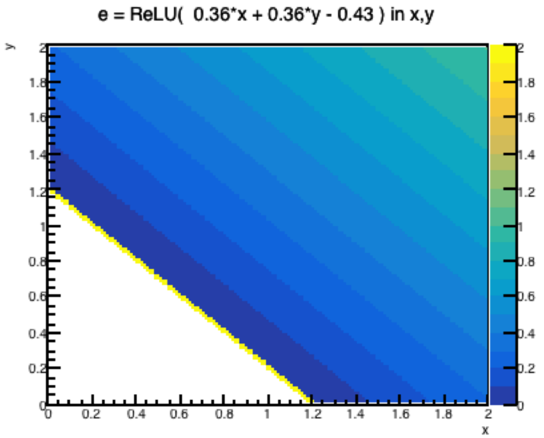
a=ReLU(-x-y+2)

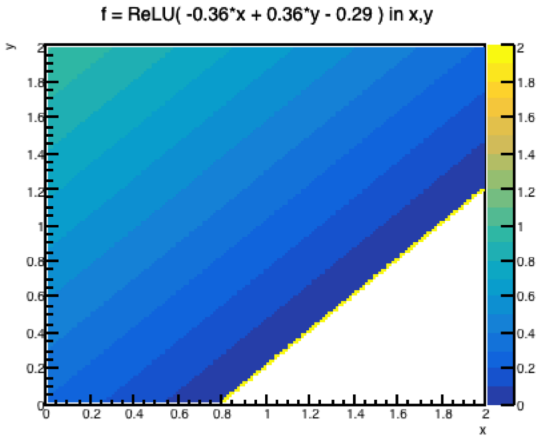


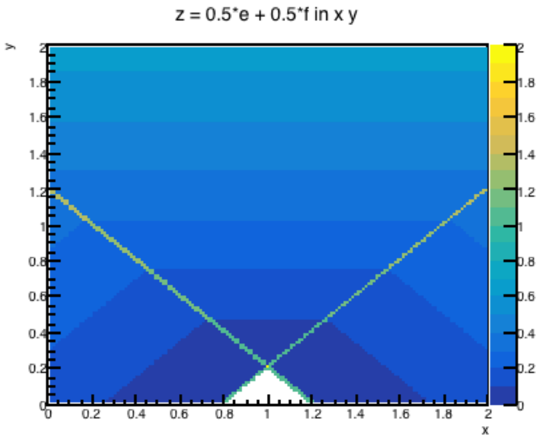
z = 0.5*c + 0.5*d in x y

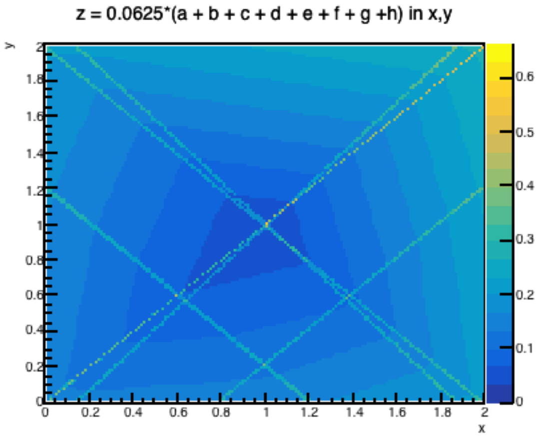


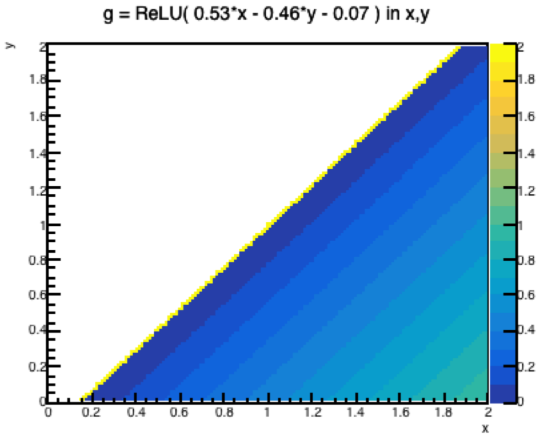

z = 0.125*(a + b + c + d + e +f) in x,y

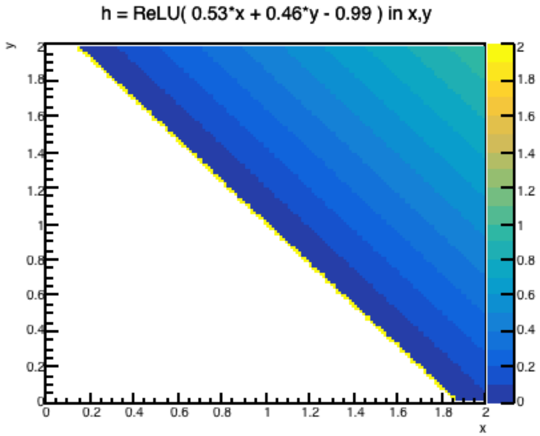

e = ReLU(0.36*x + 0.36*y - 0.43) in x,y

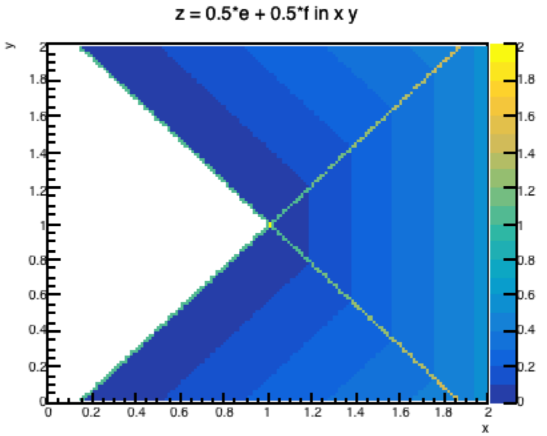

f = ReLU(-0.36*x + 0.36*y - 0.29) in x,y

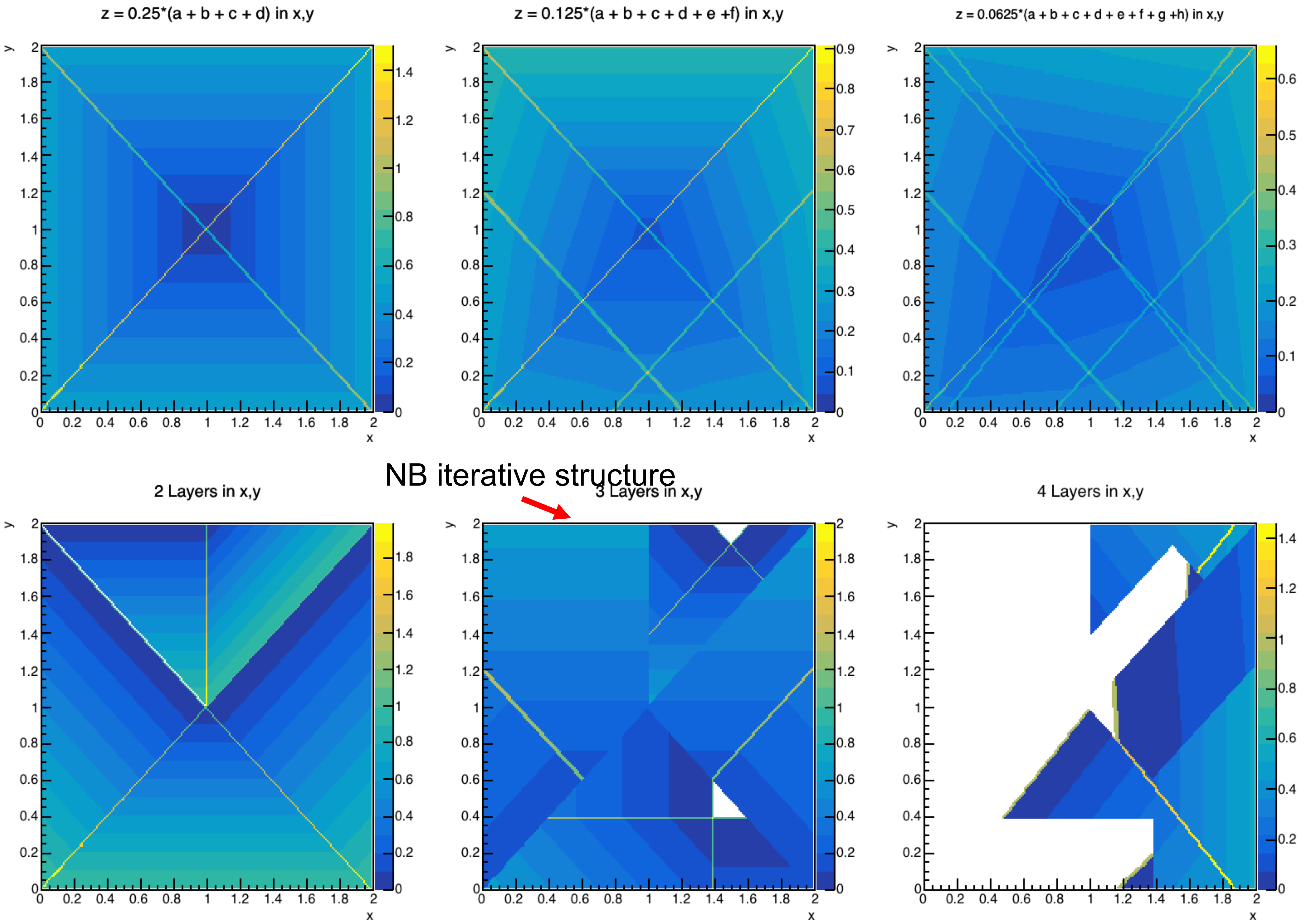

z = 0.5*e + 0.5*f in x y


z = 0.0625*(a + b + c + d + e + f + g +h) in x,y


g = ReLU(0.53*x - 0.46*y - 0.07) in x,y


h = ReLU(0.53*x + 0.46*y - 0.99) in x,y


z = 0.5*e + 0.5*f in x y



NB iterative structure

Cutting hyperplanes II - DL

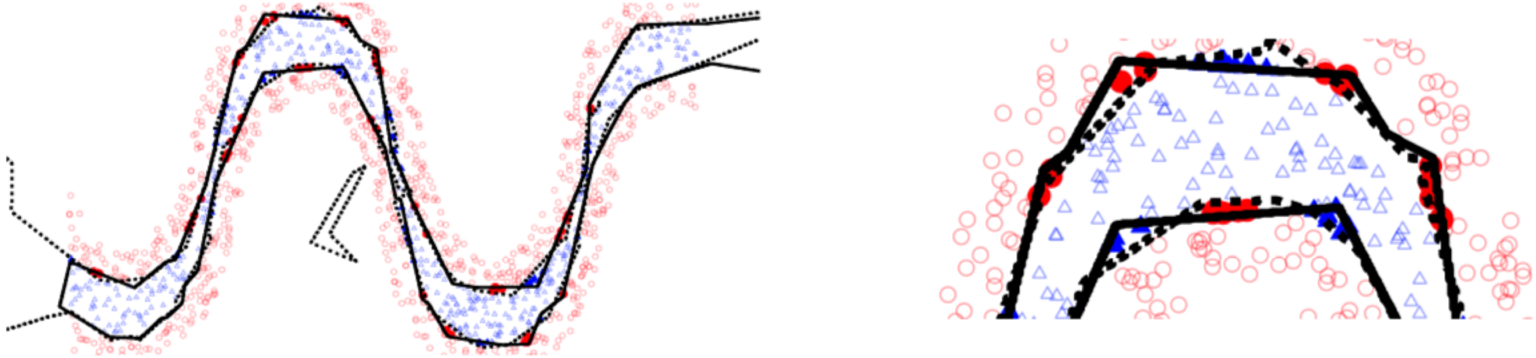


Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

On the Number of Linear Regions of Deep Neural Networks, Montúfar et al.,
https://arxiv.org/abs/1402.1869
https://arxiv.org/abs/1312.6098

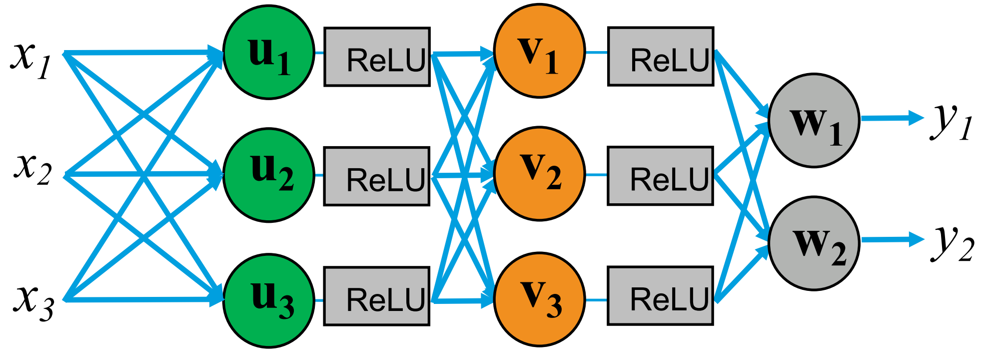
Cutting hyperplanes II - DL

- Multilayer networks are often more expressive (but not always for large input dimension, see paper)
- More complicated boundaries can be described with the same number of nodes
- But these boundaries are not independent (kind of fractal structure)
- For the ReLU networks one can show quantitative bounds on the number of linear units, see paper

Deep is better than large

Bounding and Counting Linear Regions of Deep Neural Networks, Serra et al., https://arxiv.org/abs/1711.02114

ReLU Feedforward network



The parentheses become nested

$$\underbrace{\vec{\mathbf{y}}}_{2\times 1} = \overbrace{\mathbf{W}}^{2\times 3}\, F(\underbrace{\mathbf{V}}_{3\times 3}\, \underbrace{F(\mathbf{U}\vec{\mathbf{x}})}_{3\times 1})$$

with $F(\vec{\mathbf{x}}) := \left[F(x_i)\right]$

i.e. F applied to each component of $\vec{\mathbf{x}}$

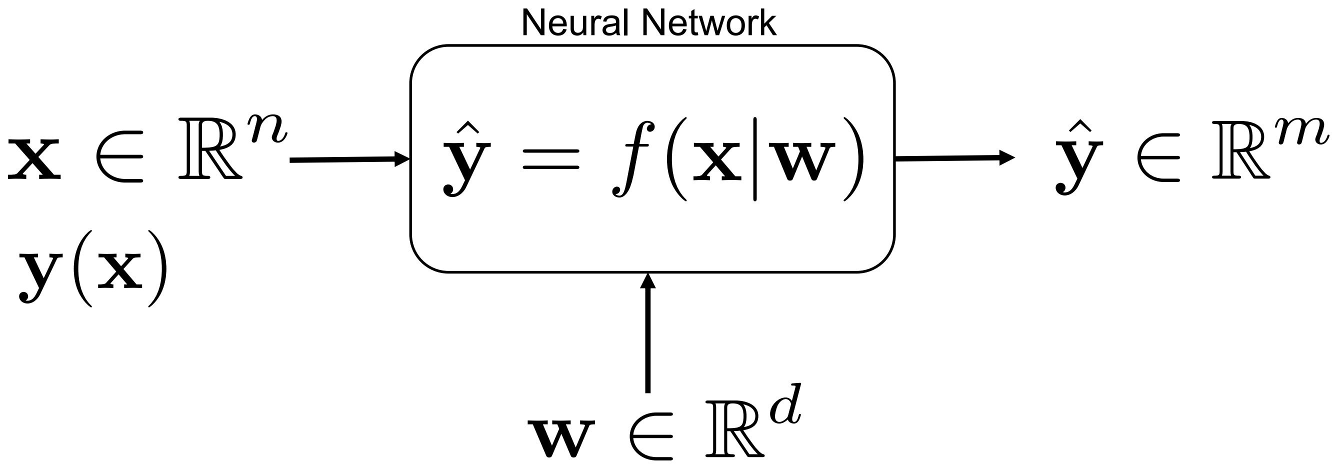
and F=ReLU

Now we can go **deep!** but
we better write this with tensors

Loss Function I

What can we do with a Neural Network?

- We can consider the NN as a complicated model that translates some input **x** into some output $\hat{y}$
  Depending on the weights and bias terms.
  (In the following: *weight* as generic term for weight and bias)



- This can be used for regression, i.e. fitting some data: $\mathbf{y}(\mathbf{x})$

- As we do it in curve fitting, we can optimize the model parameters such that they describe the data best with respect to some measure - the **Loss Function**

$$\min_{\mathbf{w}} \; loss(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{w}))$$

  - E.g. Least Squares or MSE (Mean Squared Error)

$$loss_{MSE}(\mathbf{y}, \hat{\mathbf{y}}) = (\hat{\mathbf{y}} - \mathbf{y})^2 = \sum_i^m (\hat{y}_i - y_i)^2$$

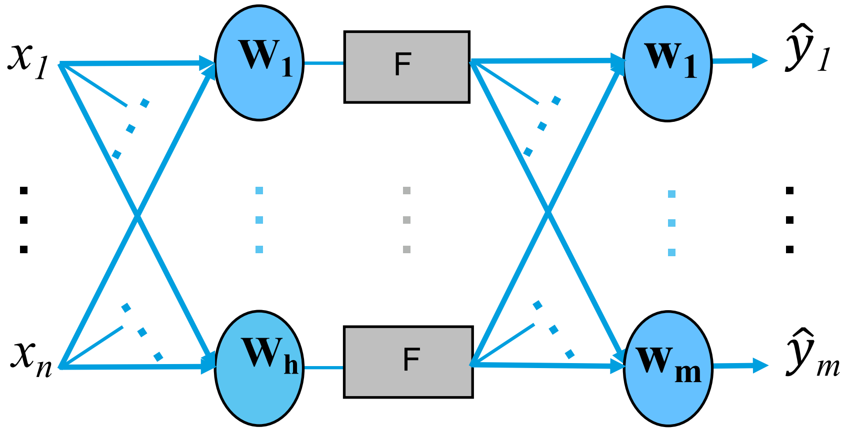
- **Minimized loss** wrt. to weights w

Loss minimization

Let's consider a fully connected 2-layer network with squared error as loss

$$loss(\mathbf{y}, \hat{\mathbf{y}}) = (\hat{\mathbf{y}} - \mathbf{y})^2 \quad \text{with}$$

$$\underset{m \times 1}{\hat{\mathbf{y}}} = \underset{m \times h}{\mathbf{W}_2} F(\underset{h \times n}{\mathbf{W}_1} \underset{n \times 1}{\mathbf{x}}) \Longleftrightarrow$$



y true values

- To minimize the loss we take the derivative wrt. to $\boldsymbol{w}$ and set it to 0
- **Chain rule**

Similar for W_2

Loss minimization

Let's consider a fully connected 2-layer network with squared error as loss

$$loss(\mathbf{y}, \hat{\mathbf{y}}) = (\hat{\mathbf{y}} - \mathbf{y})^2 \quad \text{with}$$

$$\underset{m\times 1}{\hat{\mathbf{y}}} = \underset{m\times h}{\mathbf{W}_2} F(\underset{h\times n}{\mathbf{W}_1} \underset{n\times 1}{\mathbf{x}})$$

- To minimize the loss we take the derivative wrt. to $\boldsymbol{w}$

$$\frac{\partial loss}{\partial w_i} = 0 \Rightarrow \frac{\partial (\hat{\mathbf{y}} - \mathbf{y})^2}{\partial \mathbf{W}_i} = 2(\hat{\mathbf{y}} - \mathbf{y}) \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}_i} = 0$$

E.g. for W_2 -
Similar for W_2

$$\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}_1} = \mathbf{W}_2 \frac{\partial F(z)}{\partial z} \mathbf{x}$$

- Equation to be solved for $W_{1,2}$
- **Chain rule**

Backpropagation

From loss to weights

- The previous chain rule calculation is known as **Backpropagation**

- The deviation between true and estimated $\boldsymbol{y}$,
i.e. the loss, is back-propagated to a linear change of the weights.

$$\Delta loss(\hat{\mathbf{y}}) \approx \frac{\partial loss}{\partial \mathbf{W}} \Delta \mathbf{W}$$

  - Invented by different people in the 1960/70s

- **NB: The chain rule creates a chain of factors (see later)**

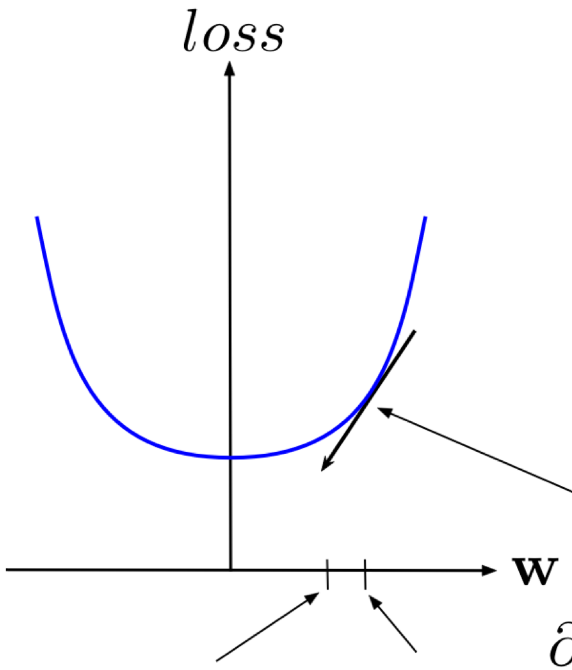
$$\frac{\partial loss(\hat{\mathbf{y}}, \mathbf{y})}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial F} \frac{\partial F(z)}{\partial z} \frac{\partial z}{\partial \mathbf{W}_i}$$

- These are vector and matrix multiplication (sums!). If you need a Matrix calculus primer or work it out with tensor indices.

Optimizer I - Gradient Descent

Iterative minimizing

- The previous chain rule calculation is known as **Backpropagation**
- In general we have due to the activation functions a non-linear behavior and in real applications the minimization of the loss is done **numerically and iteratively** to solve for the optimal weights
- To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient of the function at the current point
- The step size is chosen according to some **learning rate**

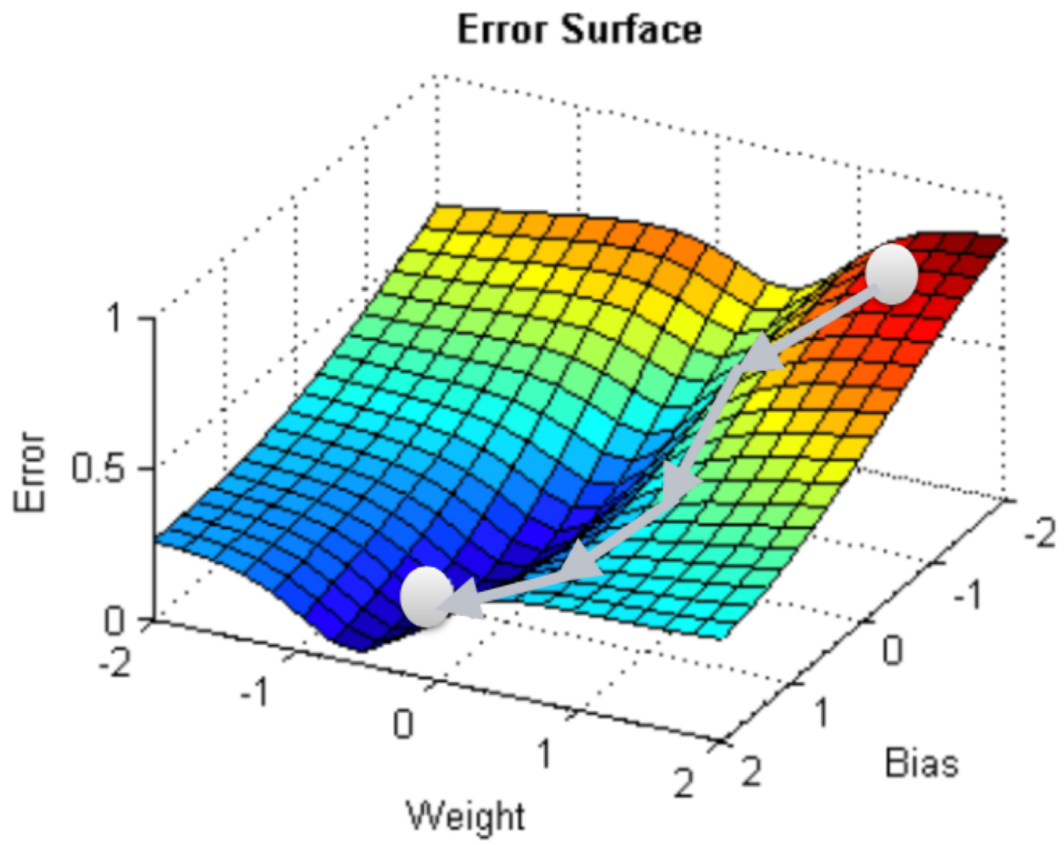


$$\mathbf{w_{i+1}} = \mathbf{w_i} - \left.\frac{\partial loss}{\partial \mathbf{w}}\right|_{\mathbf{w_i}} \times learning\ rate$$

Gradient Descent

Iterative minimizing

- The previous chain rule calculation is known as **Backpropagation**
- In general we have due to the activation functions a non-linear behavior and in real applications the minimization of the loss is done **numerically and iteratively**
- At each step the weights are updated according to some **learning rate**
- In general a complicated high dimensional surface



Summary

A neural network ...

A neural network

- is a model defined by a set of parameters (weights and bias),
- is a model that can practically learn all kind of data manipulation (decisions, cuts, fits)
- is a chain of linear transformations and non-linear activation functions,
- learns by modifying the weights,
- is trained by backpropagation and gradient descent
- using the gradient with respect to the weights.
- A deep neural networks had many layers and seems to do jobs better with the same number of parameters - no fundamental understanding yet

AWS setup
and jupyter notbooks
- IPython_and_Jupyter.ipynb

How to connect

We use AWS

- You will run a **jupyter notebook** server which you can access locally with your browser

1. Connect from your **laptop shell**!:

2.
```
mkdir tmp
ssh -i A_DESY.pem \
        -S./tmp \
        -L 1080:127.0.0.1:8888 \
```

   ubuntu@ec2-18-202-237-151.eu-west-1.compute.amazonaws.com

   - This is a machine in the amazon cloud.
     Please replace the ubuntu@"name" of the machine with that name in the Google doc which is reserved for you
   - We connect by ssh
     - The pem key you have got is used instead of a password
     - Your username is ubuntu
     - -L creates a tunnel, that means you can access later the jupyter notebook on http://localhost:1080/
     - The -S ./tmp you sometimes need on a Mac

3. When you are on the AWS machine:
   - The tutorials for today can be clone with
     - `git clone\ https://github.com/dkgithub/DESY_ML_PyTorch.git`
     - README.md for more details!!
   - `jupyter notebook or jupyter lab`
   - Copy the token

4. Connect your browser http://localhost:1080/
   - Use the token

- If everything is running try the: DLpytorch/tutorials/IPython_and_Jupyter.ipynb to get familiar with jupyter
  - You should be able to navigate to that file from within your browser

Connect to Local Computers

naf-school01.desy.de ..06

- These computers are open from outside the DESY network
- We have all necessary software installed here
- `module load anaconda/2`
- `python`
- `ipython`
- `jupyter notebook`
- `jupyter lab`
- If you run a notebook here the standard port may be in use. Check the output jupyter is telling you which port is used. See README on github

Thank you

Contact

DESY. Deutsches
Elektronen-Synchrotron

www.desy.de

Dirk Krücker
CMS
dirk.kruecker@desy.de