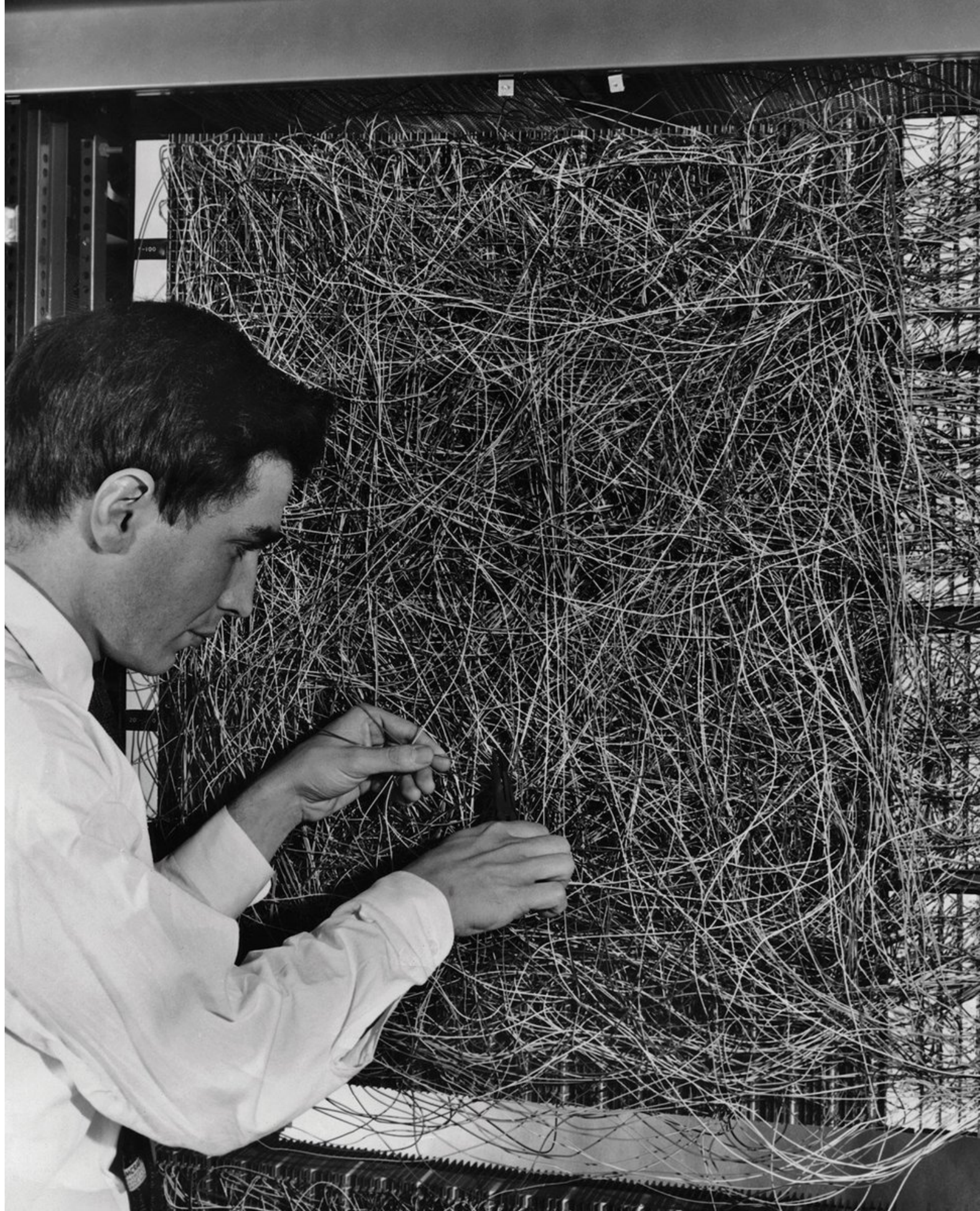
A Gentle Introduction to Neural Networks and Deep Learning

Deep Learning and GPU Computing at DESY

Dirk Krücker
DESY - Hamburg, 25.02.2018

HELMHOLTZ RESEARCH FOR GRAND CHALLENGES



What is a Neural Network?

Some answers

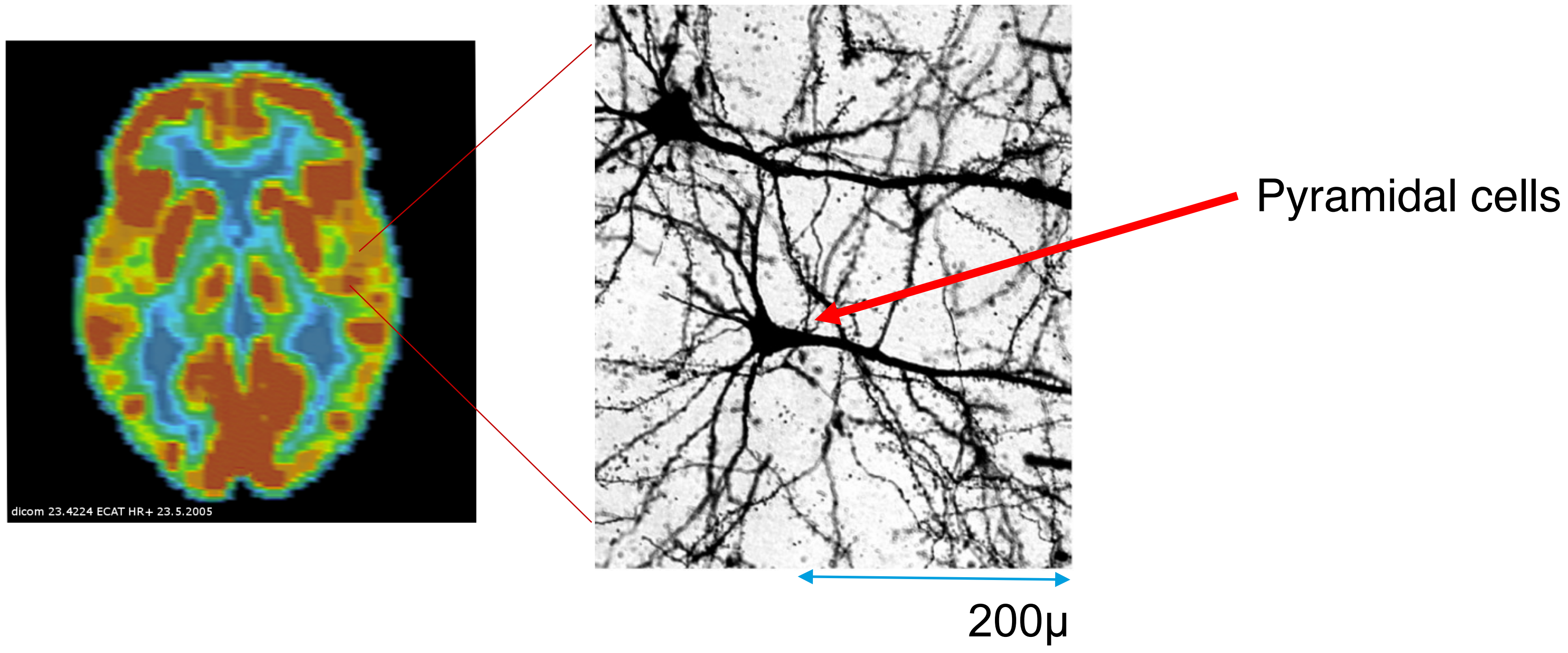
- A primitive model for a piece of brain matter
- A series of geometric projections
- A few lines of linear algebra + some functions
- A type of function approximation
- An optimization problem
- A few lines of python code

- I‘m a particle physicist.
  I tend to call data points: events
- To be honest gentle means crash course

- We will go quickly through all of these views
- Follow the links later

Me - Neurons

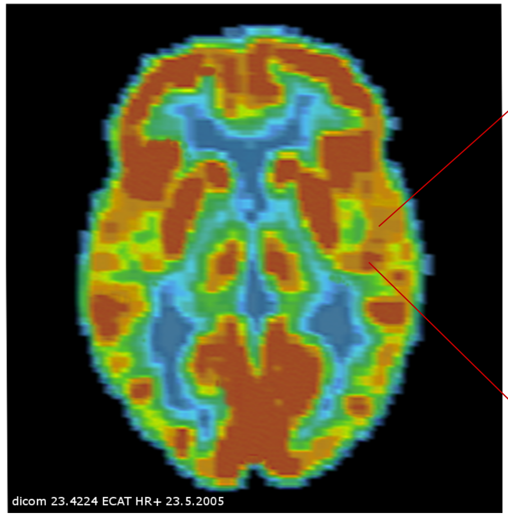
I worked on brain imaging a while ago (PET)

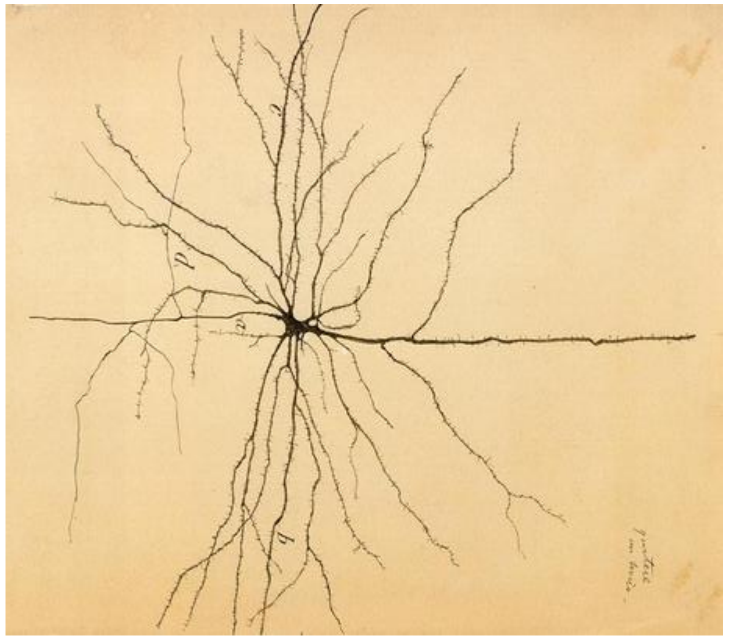


Neurons as computational units are an old idea.

Me

Some history





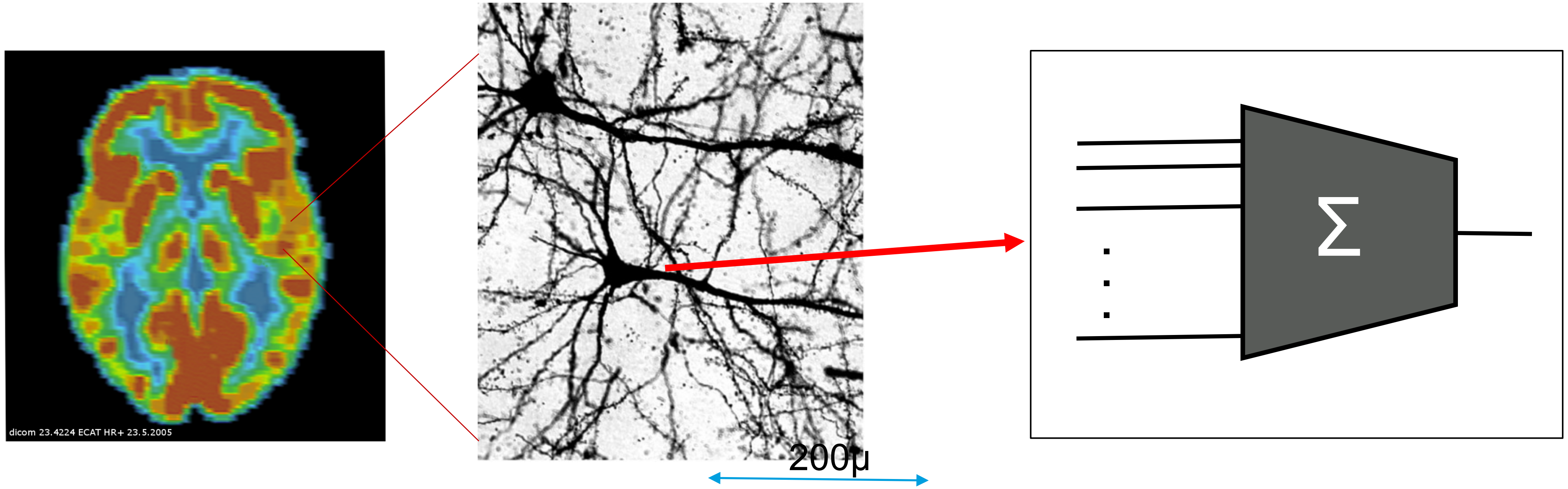
One of Cajal’s drawing from what he saw under the microscope
Golgi's method (1873): silver staining of neural tissue

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

- Santiago Ramón y Cajal, the father of modern neuroscience, had this idea in 1894 (**learning through new connections**).
  This is not yet the idea of computation.

Me - Neurons

I worked on brain imaging a while ago (PET)

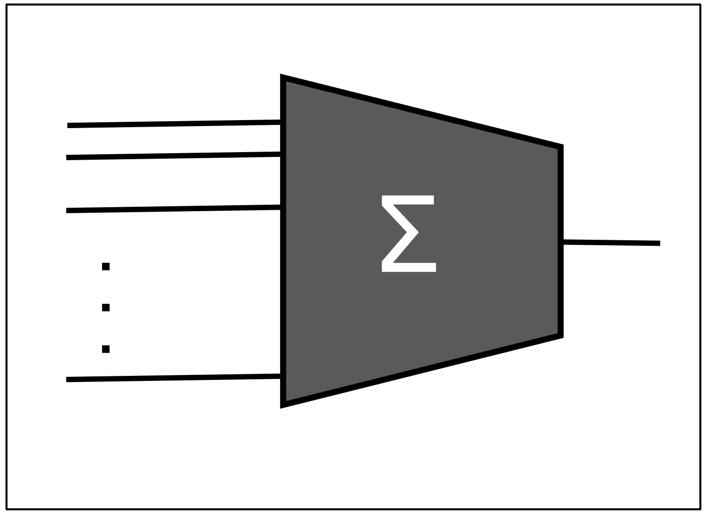


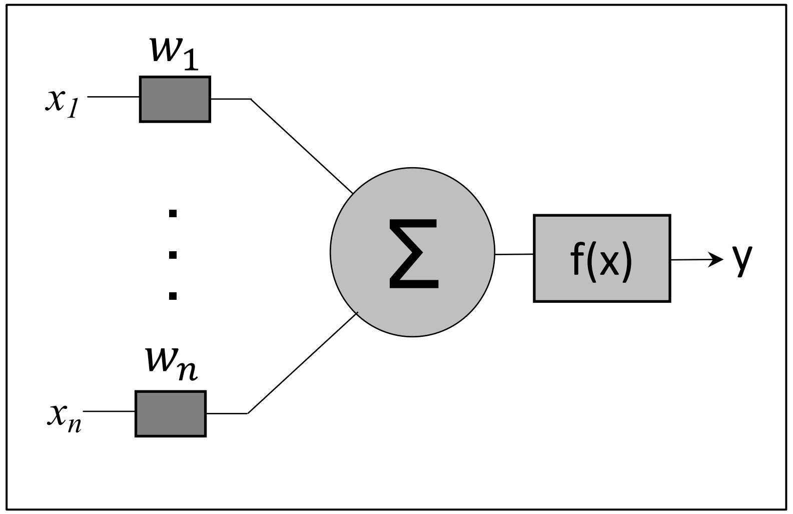
Neurons as computational units are an old idea.

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

Computation

Spelling out the mathematical model – learning by modifying the strength of connections





$$y = f\left(\sum_{i=1}^{n} w_i x_i \right)$$

Simple mathematical model

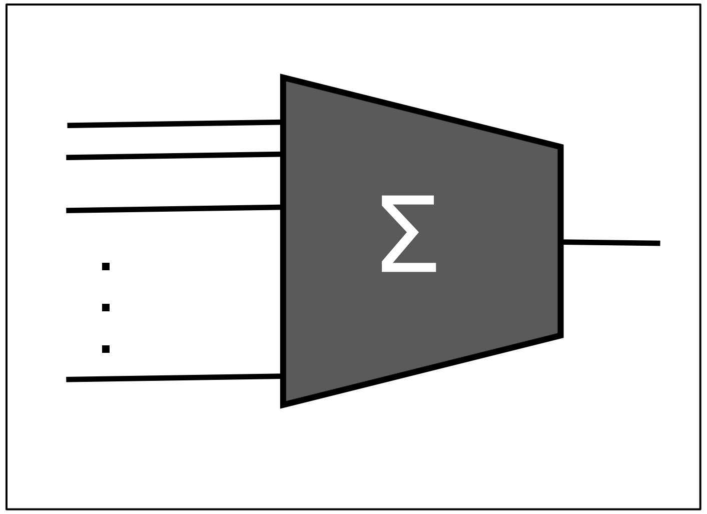
- Each neuron input gets multiplied by a **weight**,
- they are summed up
- and some **function** is applied.

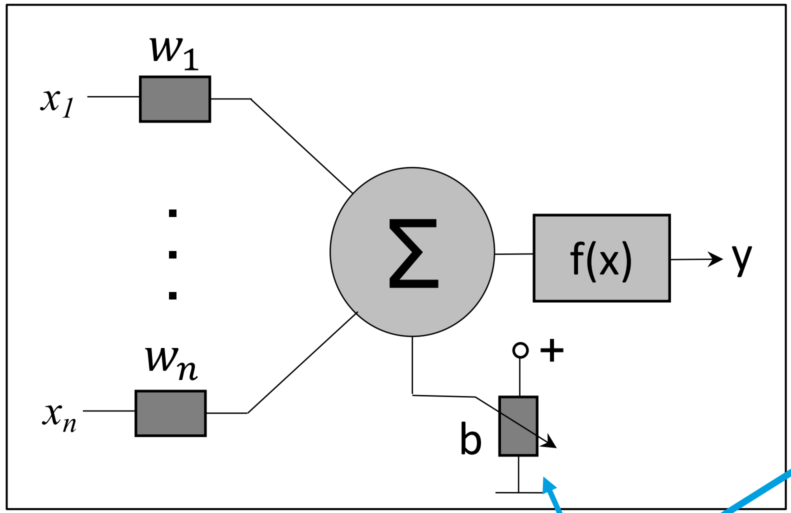
It emerged later that **<u>networks</u>** of such nodes provide rich structures

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

Computation

Bias term





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

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 $\mathbb{R}^n$.
In Machine Learning the input variables are called **features**

Geometric interpretation -->

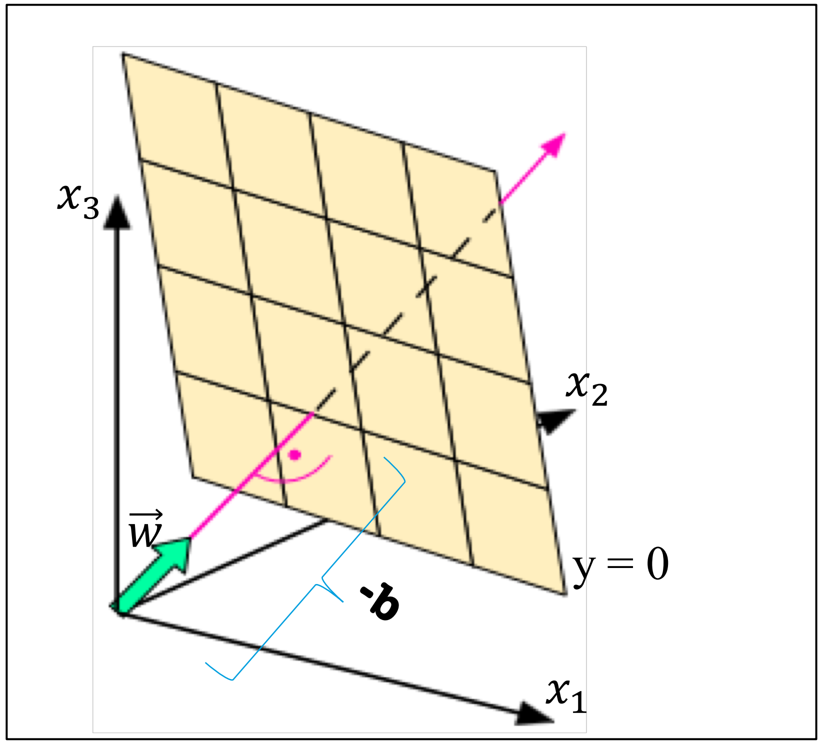
Geometry

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.

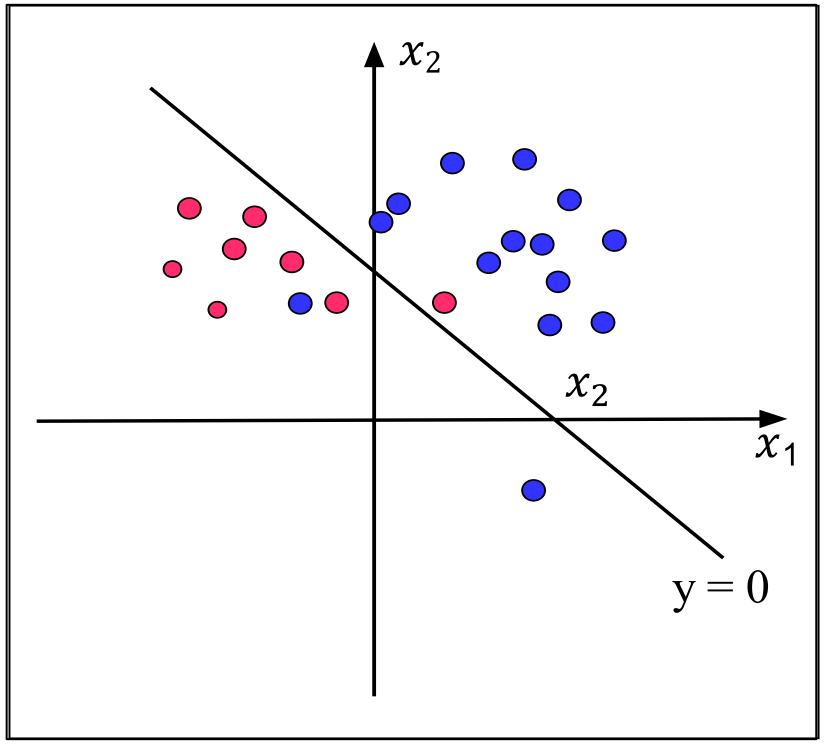
Perceptron

The first learning machine - Frank Rosenblatt 1957

Threshold:

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

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



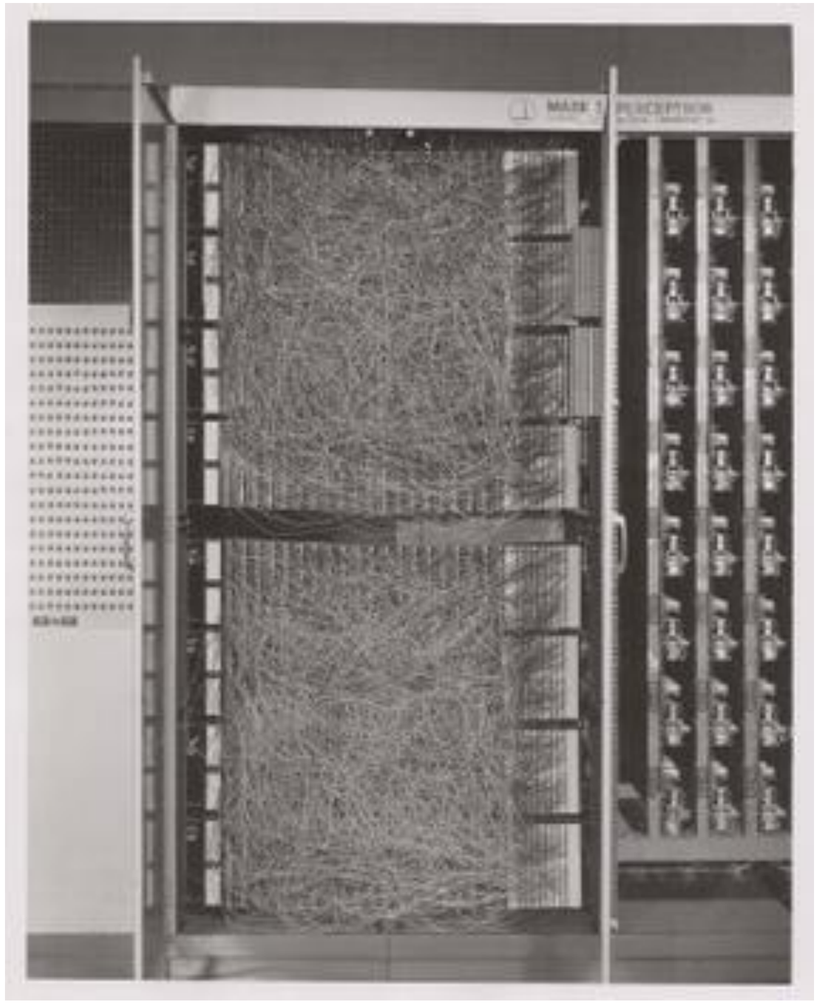
The Perceptron is one of the oldest ideas for machine learning and Artificial Intelligence. Cutting the input space into two halves. Learning means finding the best values for b and $\vec{w}$

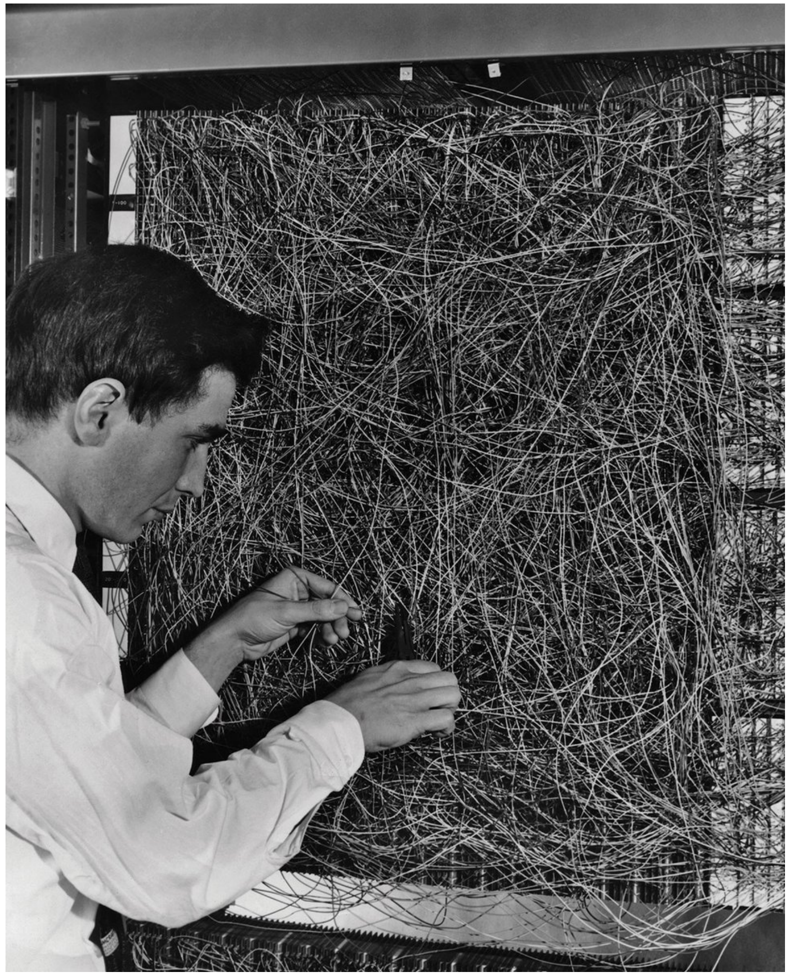
Mark I Perceptron

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

Some Vocabulary

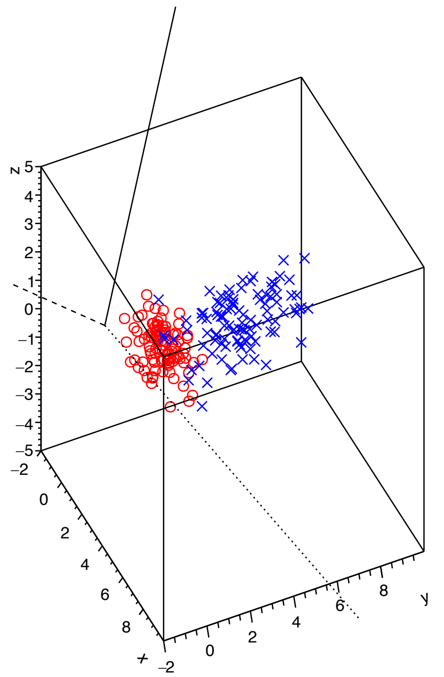
Machine Learning

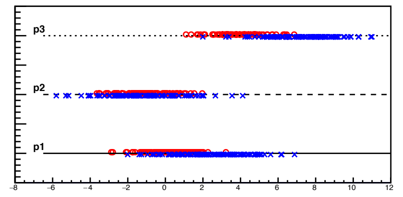
- We are interested in **Supervised Learning,** that means we have a **Training Dataset** where we know the true **Target Values**
  - **Classification → Label**
    - **Binary**
      e.g. Background/Signal
    - **Multi-class**
      e.g. the picture shows a: cat, dog or horse
  - **Regression → Numerical target value**
    - Some numerical fit value as function of some high-dimensional input vector
- We want to learn from the trainings dataset and apply this to some other new data where we do not know the label/target value

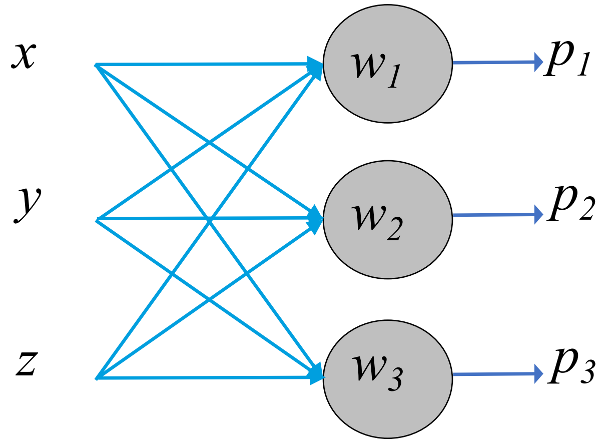
- The input variables are called **features**
- We assume that they form an **vector space** with a **distance** although we compare apples with orange
  -
  - E.g. Temperature in C, mm rainfall and average hotel prices in Euro and predict the number of hotel guest
  - If/since there is no natural common unit we gain by
    $\Longrightarrow$ **Normalize the input features**

From 1 node to 1 layer

Aside: Random projection





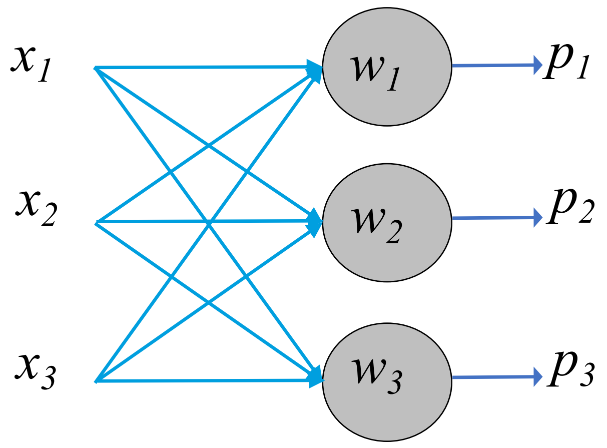


- Each node represents a projection **N ⟶ 1**:
$$p_i = \vec{w_i}^{\mathrm{T}} \vec{x} + b$$
with its own weight vector and bias
- **A layer with multiple nodes form an N ⟶ M mapping**
- (A high dimensional N to a low dimensional M may even work with a randomly chosen set of M axes if the data 'lives' on a low dimensional subspace (Johnson–Lindenstrauss lemma[1]))
- **Even if we just start with random *w*, we will keep some separation power that can be optimized by optimizing the weights**
  - But see appendix on weight initialization

Feedforward Network

1 Layer as matrix

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

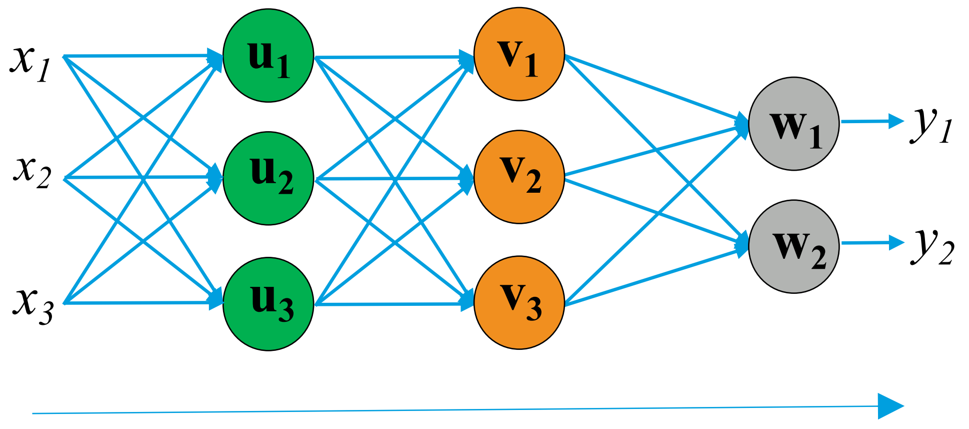


- Each node represents a projection **N ⟶ 1**:
  $$p_i = \overrightarrow{w_i}^{\mathrm{T}} \vec{x}$$
  with its own weight vector
  - We drop the bias for a while for simplicity
- **A layer with multiple nodes form an N ⟶ M mapping,** i.e. it a layer can be represented by some Matrix ***W*** of weights
- A **N⟶N** is called a **dense network**

Feedforward Network

Multiple layers

- 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**



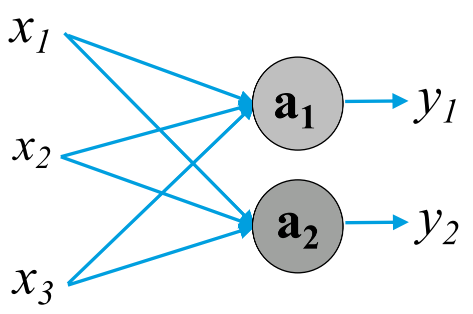
$$\underbrace{\vec{\mathbf{y}}}_{2\times 1} = \overbrace{\mathbf{W}}^{2\times 3}\ \underbrace{\mathbf{V}}_{3\times 3}\ \overbrace{\mathbf{U}}^{3\times 3}\ \underbrace{\vec{\mathbf{x}}}_{3\times 1}$$

$$\longleftarrow$$

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

$$\boxed{y = \underset{\uparrow}{f}\left(b + \sum_{i=1}^{n} w_i x_i\right)}$$

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

Activation Functions

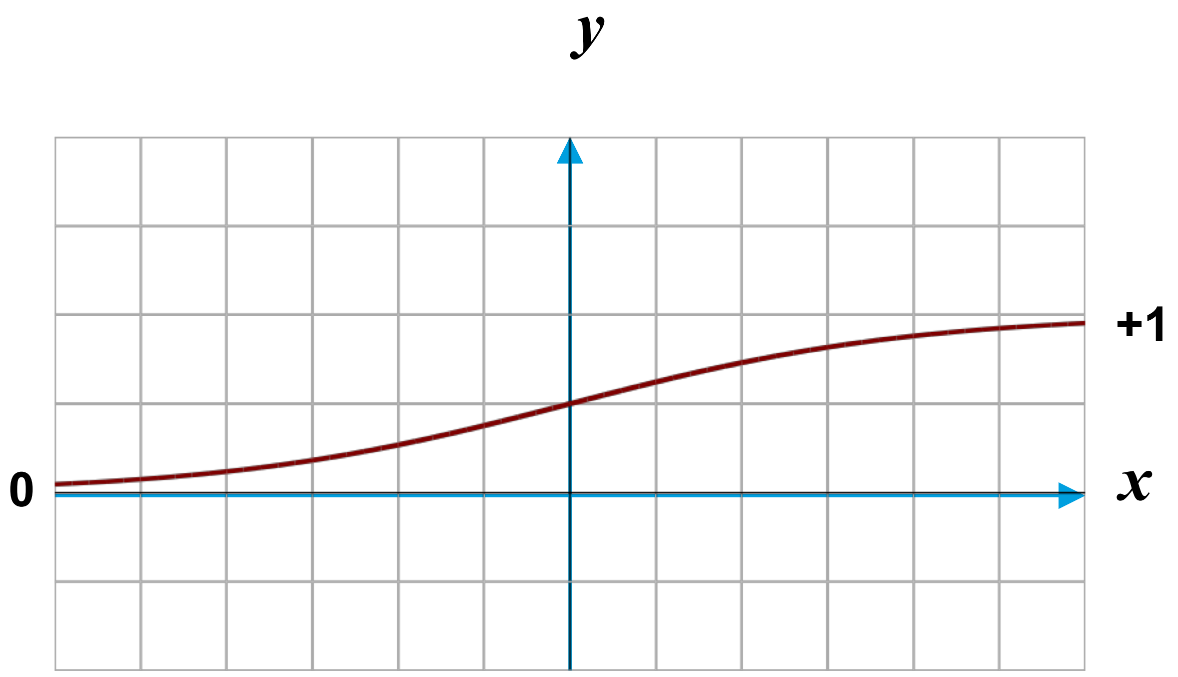
- Identity $f(x) = x$
- Simple threshold (classic NN) $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
- **Sigmoid** (logistic function)
- **Tanh**
- **ReLU** (REctified Linear Unit)
- Leaky ReLU
- RBF (see SVM)
- **Softmax**
- ... https://en.wikipedia.org/wiki/Activation_function

Activation Functions

- Identity
- Simple threshold (classic NN)
- **Sigmoid** (logistic function)

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

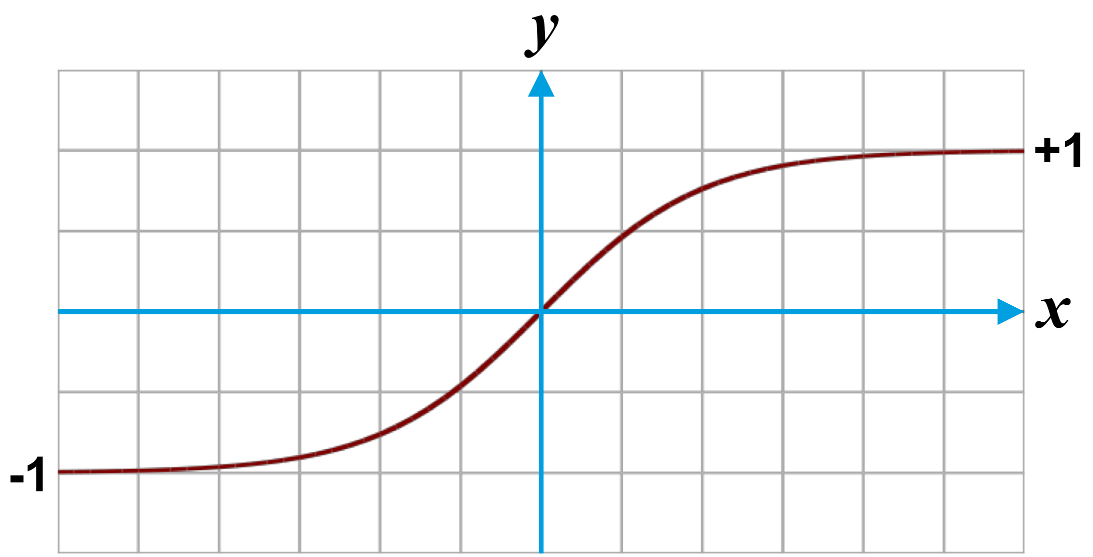
- **Tanh**
- **ReLU** (REctified Linear Unit)
- Leaky ReLU
- RBF (see SVM)
- **Softmax**
- ... https://en.wikipedia.org/wiki/Activation_function



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

Activation Functions

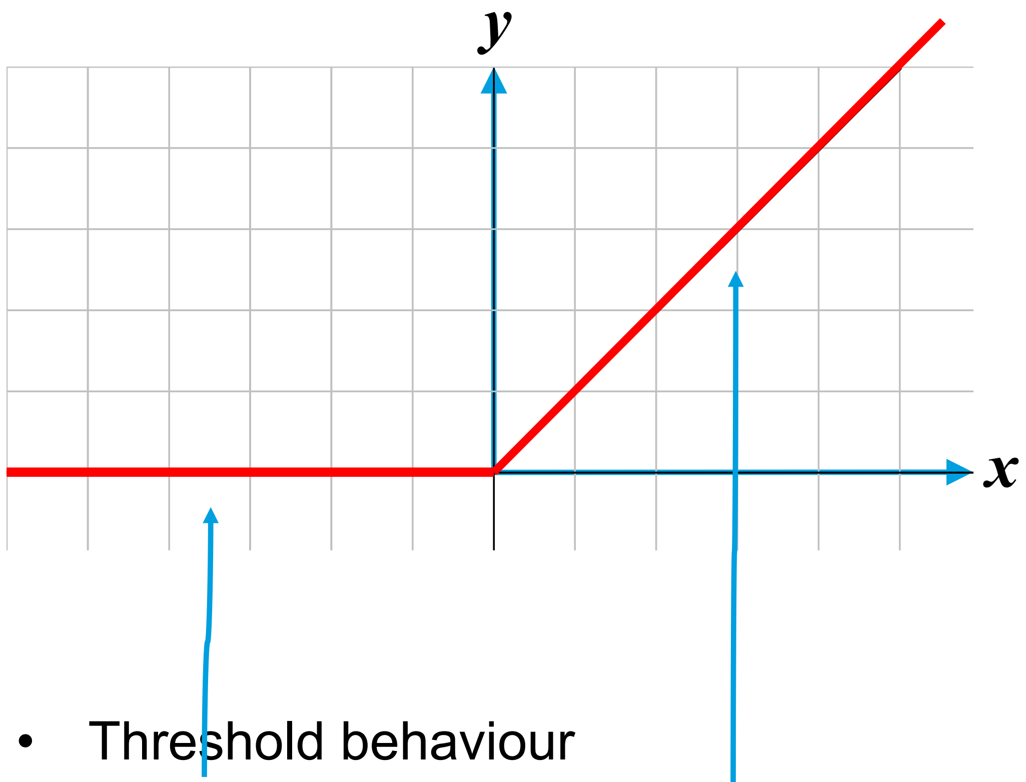
- Identity
- Simple threshold (classic NN)
- **Sigmoid**
- **Tanh** $f(x) = \tanh(x)$
- **ReLU** (REctified Linear Unit)
- Leaky ReLU
- RBF (see SVM)
- **Softmax**
- … https://en.wikipedia.org/wiki/Activation_function



- Maps between -1 and 1
- Use to be popular in 1st generation NN
- Linear at origin $\longrightarrow$ loss of non-linearity
- Saturates easily

Activation Functions

- Identity $f(x) = x$
- Simple threshold (classic NN)
- **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
  - Scale invariant
- **Most popular choice now- "Game changer"**

ReLU as a cut in 1D

Let's play with these ReLU

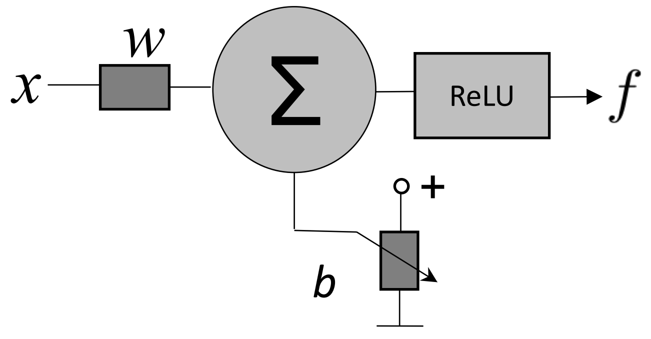
- The condition defines a cut on the input variable x

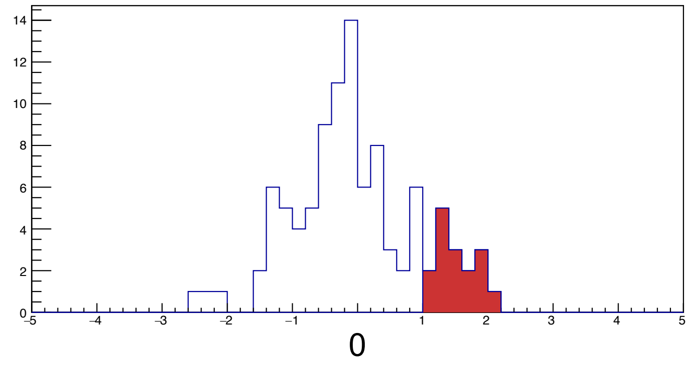
$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

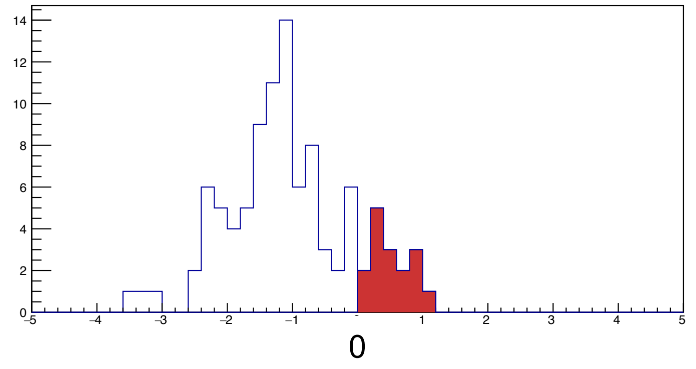
- The cut on *x* is defined by the weight *w* and the bias *b*
- Instead of cutting on $x_{in} > x_{cut}$ we scale x_{in} by *w* and shift the input distribution by *b*:

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

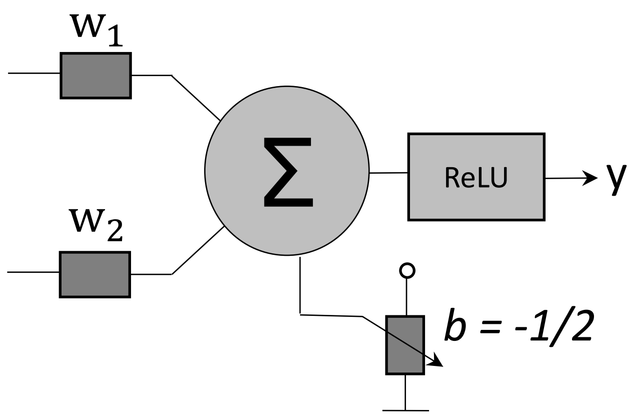
- *"we move the distribution to the cut"*
- In a ReLU layer the **bias defines the scale**



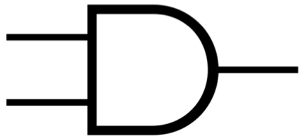




ReLU as a logic gate

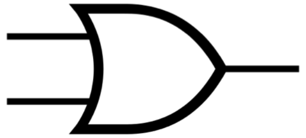


$w_1 = w_2 = 1/2$



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

$w_1 = w_2 = 1$



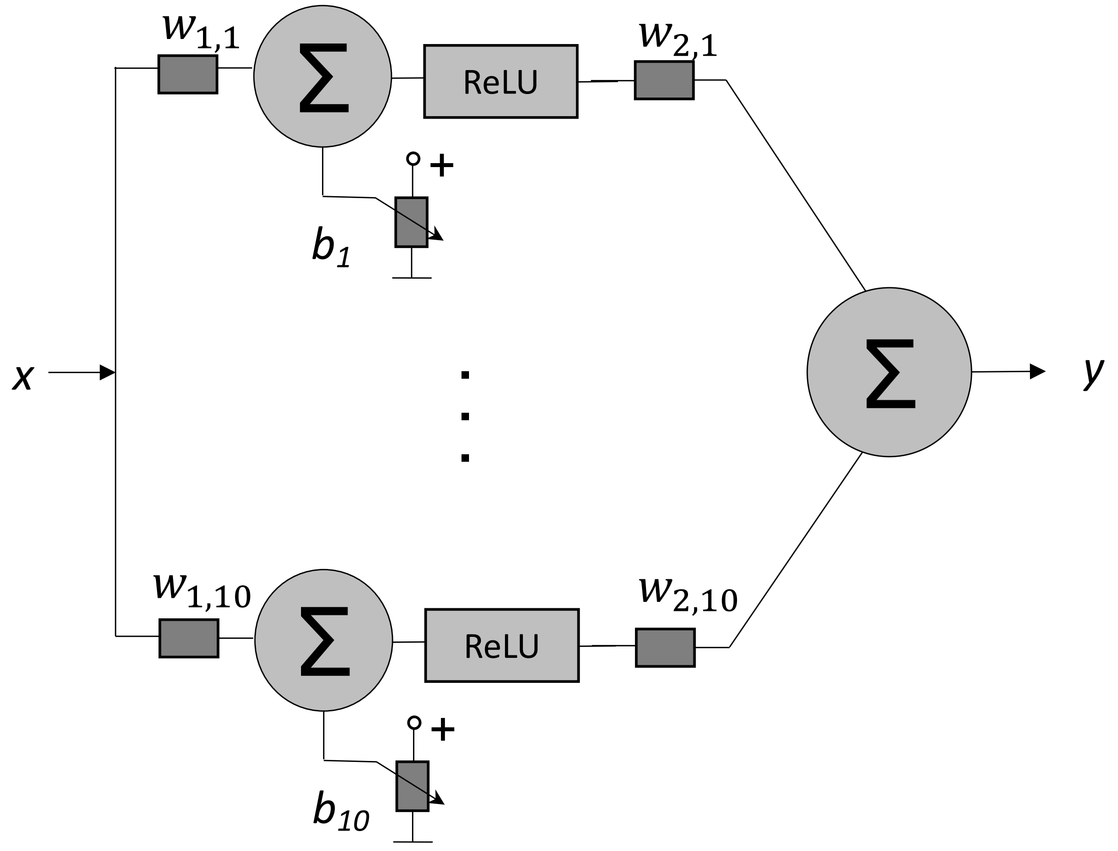
x	y	Σ	y
0	0	0	0
1	0	1	1
0	1	1	1
1	1	2	1

All kinds of can data manipulations can be realize by ReLU networks

negative weight ⟶ logic not

Universal Approximation

Function approximation

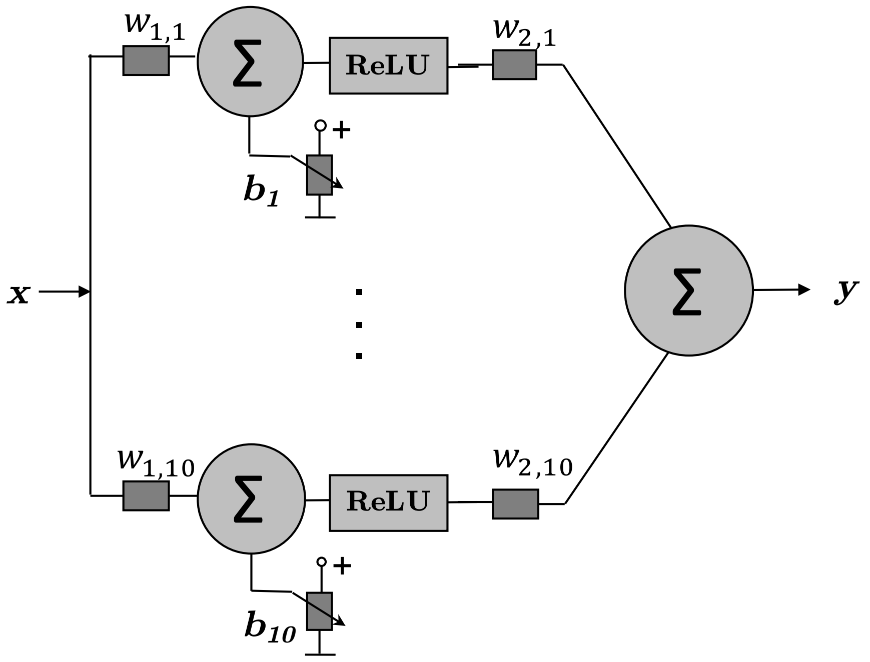


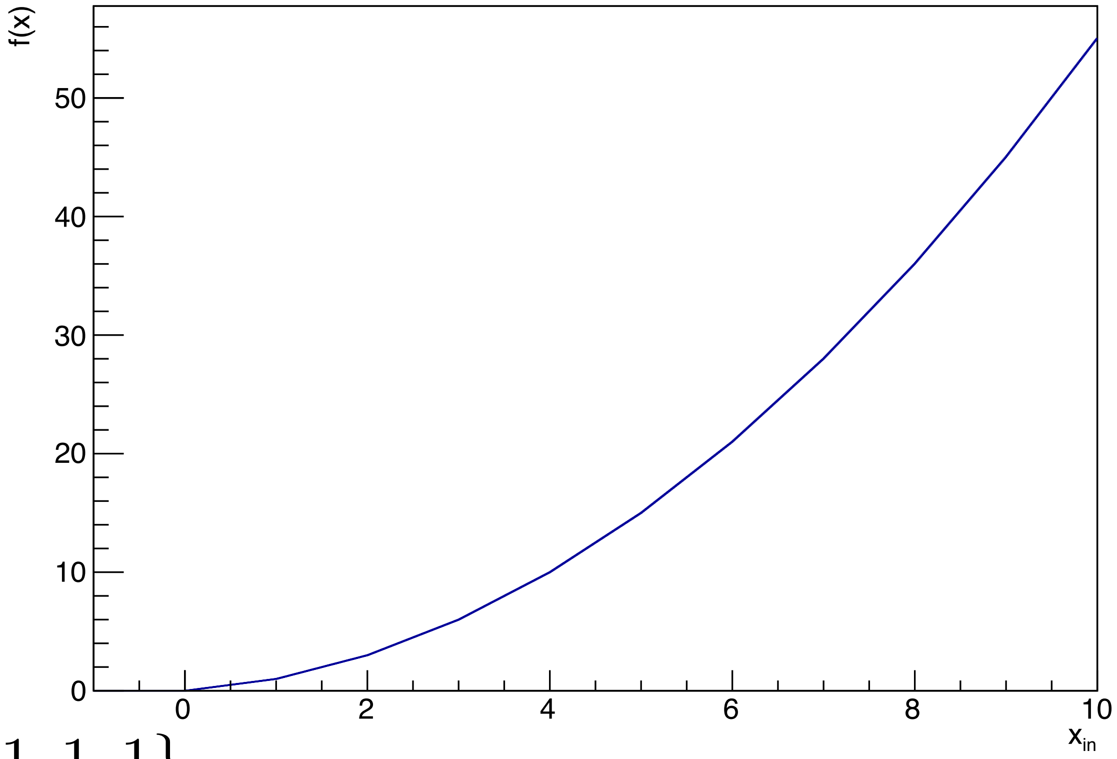
- Nodes with one input *x* and one output y
- We get a piecewise linear function that can model an arbitrary function.
- Intuitively clear[1] that **one hidden layer with sufficient many nodes can approximate any smooth function** to a given accuracy ε

$$\int_{\mathbb{R}^n} \left|f(x) - F_{\mathcal{A}}(x)\right| \mathrm{d}x < \epsilon$$

[1] https://en.wikipedia.org/wiki/Universal_approximation_theorem, G.Cybenko(1989)

Universal Approximation





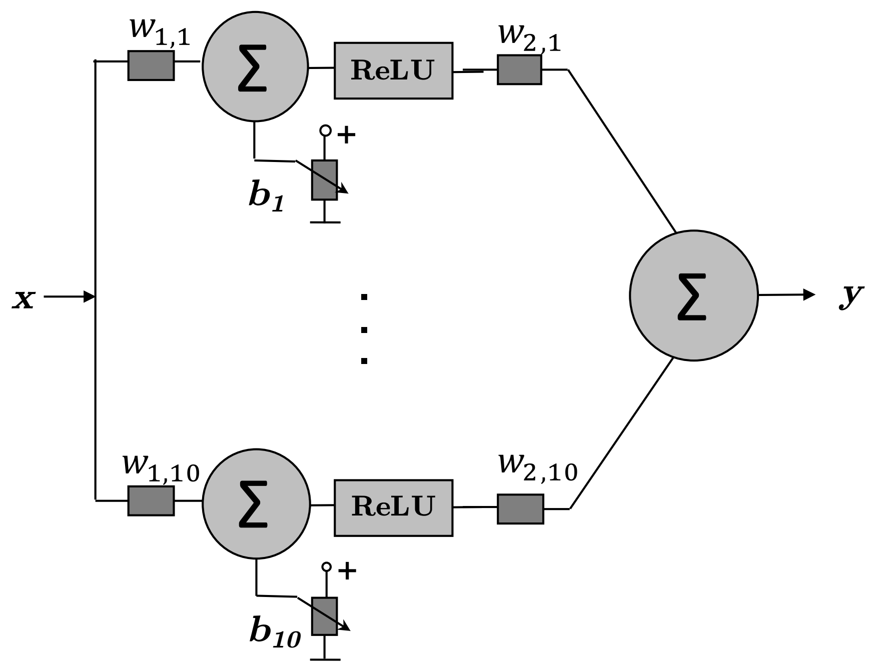
$$\{w_{1,1}, \ldots, w_{1,10}\} = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$$
$$\{w_{2,1}, \ldots, w_{2,10}\} = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$$
$$\{b_1, \ldots, b_{10}\} = \{0, -1, -2, -3, -4, -5, -6, -7, -8, -9\}$$

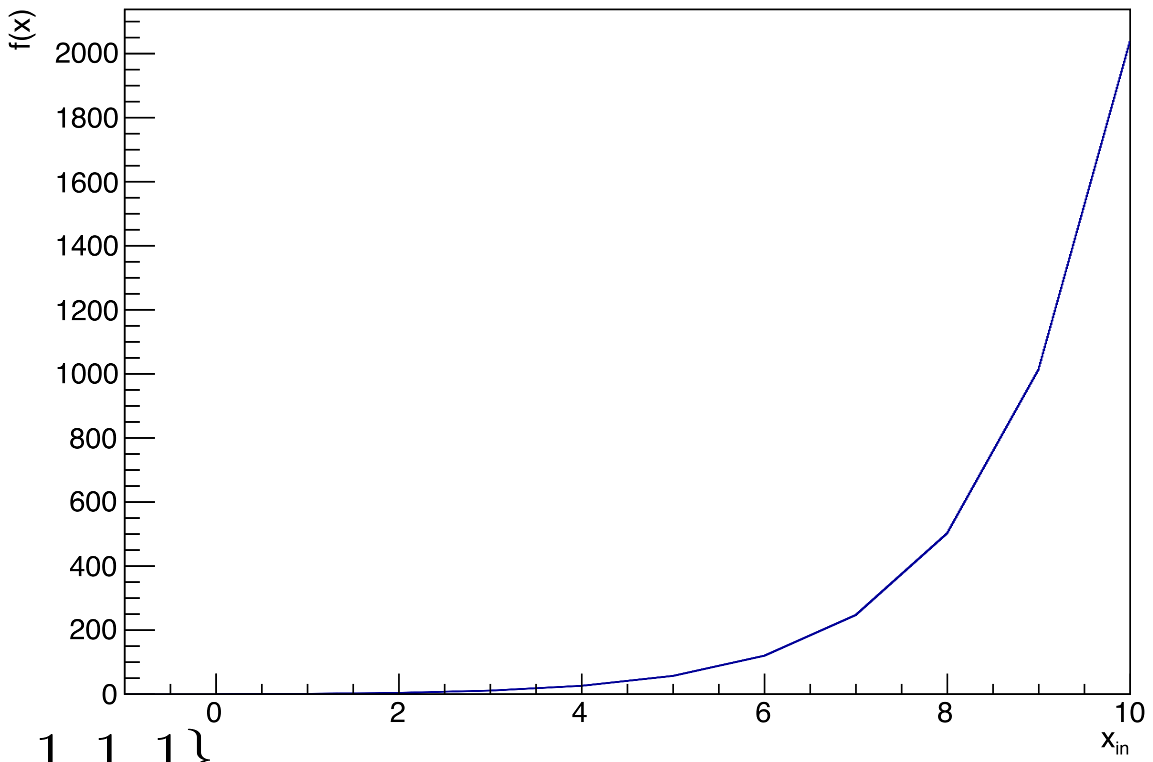
$$f(x_{in}) \approx x^2$$

1 layer of ReLU presents a piecewise linear approximation

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

Universal Approximation





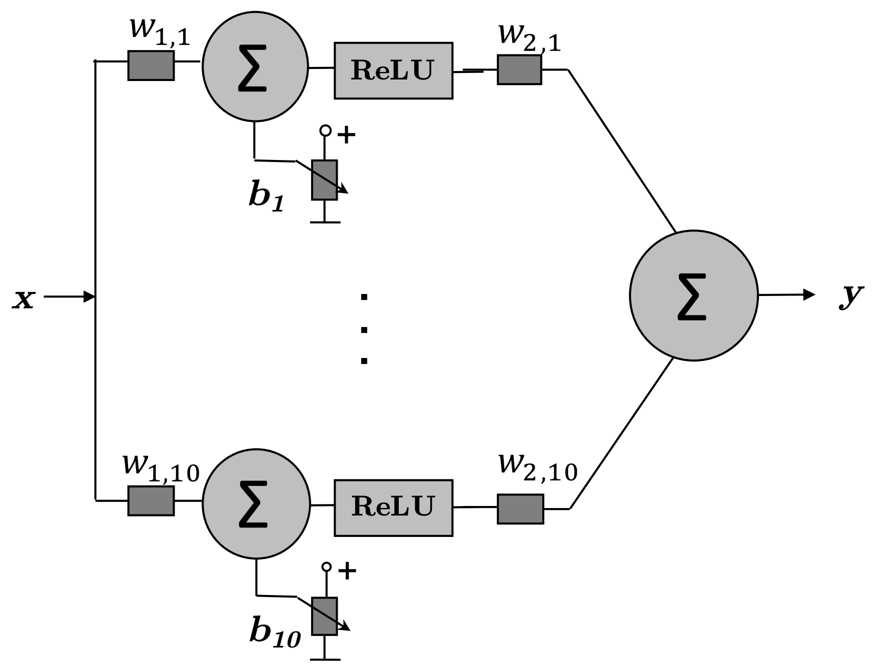
$$\{w_{1,1},\ldots,w_{1,10}\} = \{1,1,1,1,1,1,1,1,1,1\}$$
$$\{w_{2,1},\ldots,w_{2,10}\} = \{1,2,4,8,16,32,64,128,256,512\}$$
$$\{b_1,\ldots,b_{10}\} = \{0,-1,-2,-3,-4,-5,-6,-7,-8,-9\}$$

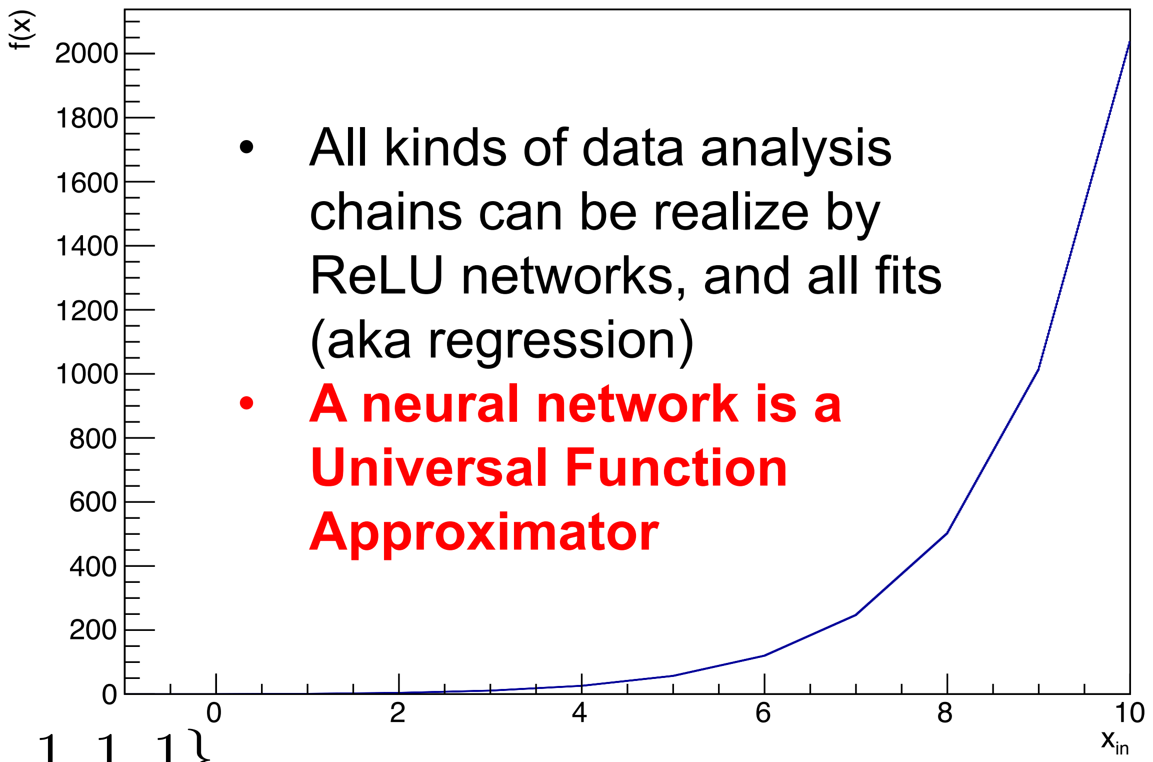
$$f(x_{in}) \approx 2^x$$

1 layer of ReLU presents a piecewise linear approximation

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

Universal Approximation





- All kinds of data analysis chains can be realize by ReLU networks, and all fits (aka regression)
- **A neural network is a Universal Function Approximator**

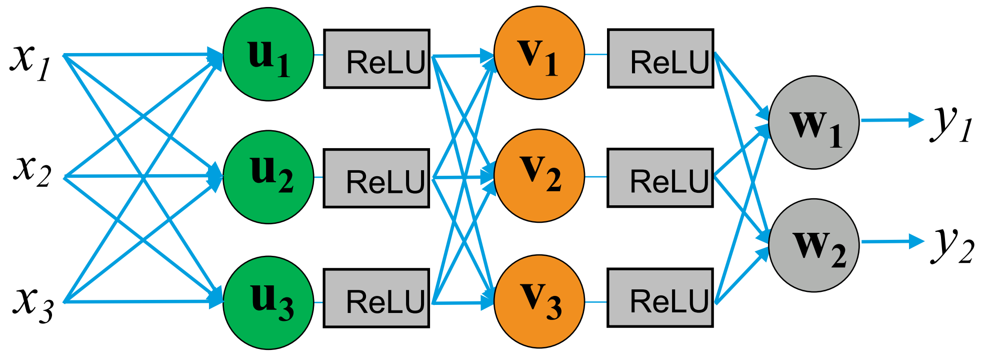
$$\{w_{1,1}, \ldots, w_{1,10}\} = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$$
$$\{w_{2,1}, \ldots, w_{2,10}\} = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$$
$$\{b_1, \ldots, b_{10}\} = \{0, -1, -2, -3, -4, -5, -6, -7, -8, -9\}$$

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

1 layer of ReLU represents 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 $F(\vec{\mathbf{x}}) := \left[F(x_i)\right]$

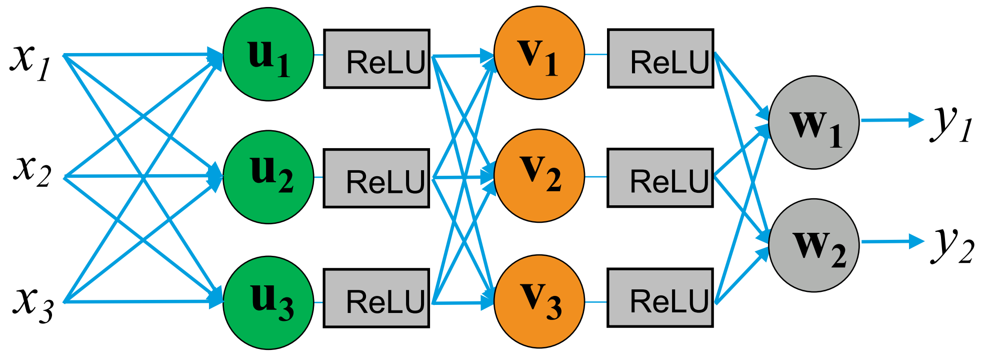
i.e. F applied to each component of $\vec{\mathbf{x}}$

and F=ReLU

Now we can go **deep!**

we also can 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 graphical, 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

Complexity explosion and dimensionality – number of compartments for ReLU networks

- 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
- For example: 10 dimensional input with 40 nodes may cut your phase space into up to max **1.2 billion pieces**
  - more then parameters (~10x40+40)
  - more then data
  - but there may be an appropriate low dim configuration

d = 2 (paper plane)

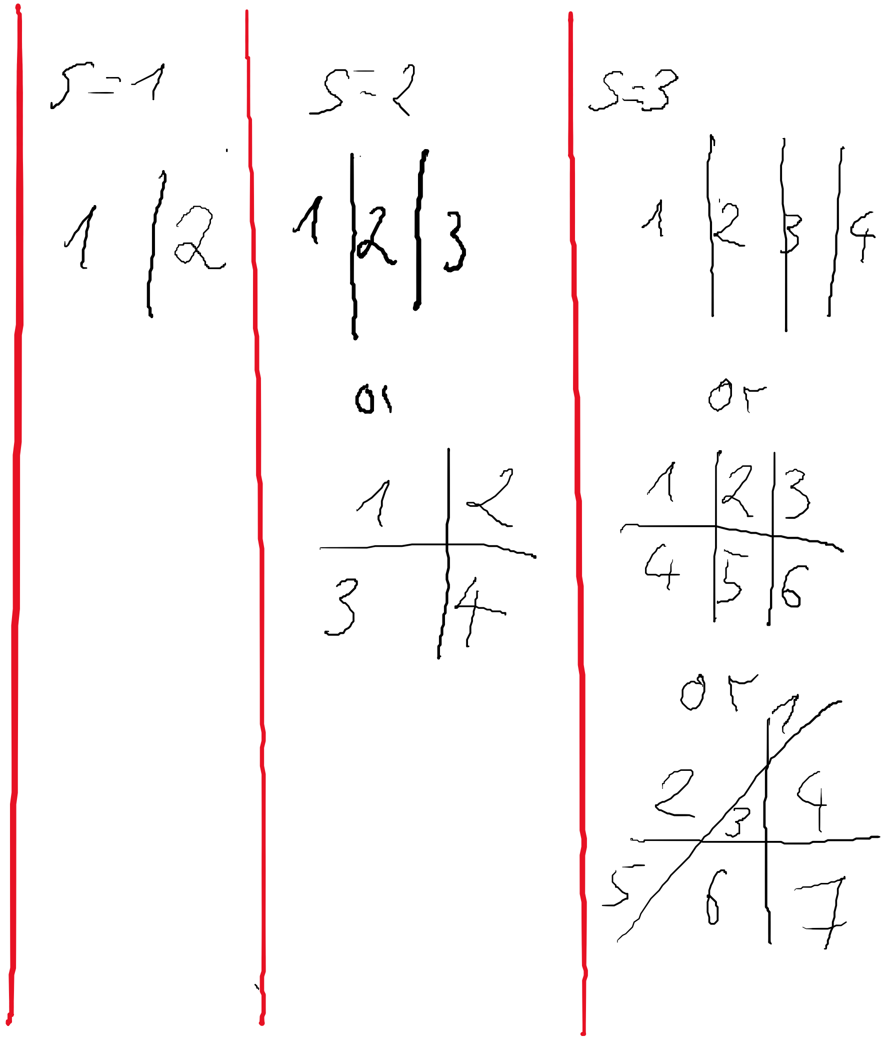
s = # hyperplanes (lines)

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



What is the advantage of being deep?

Not well understood theoretically but extremely successful

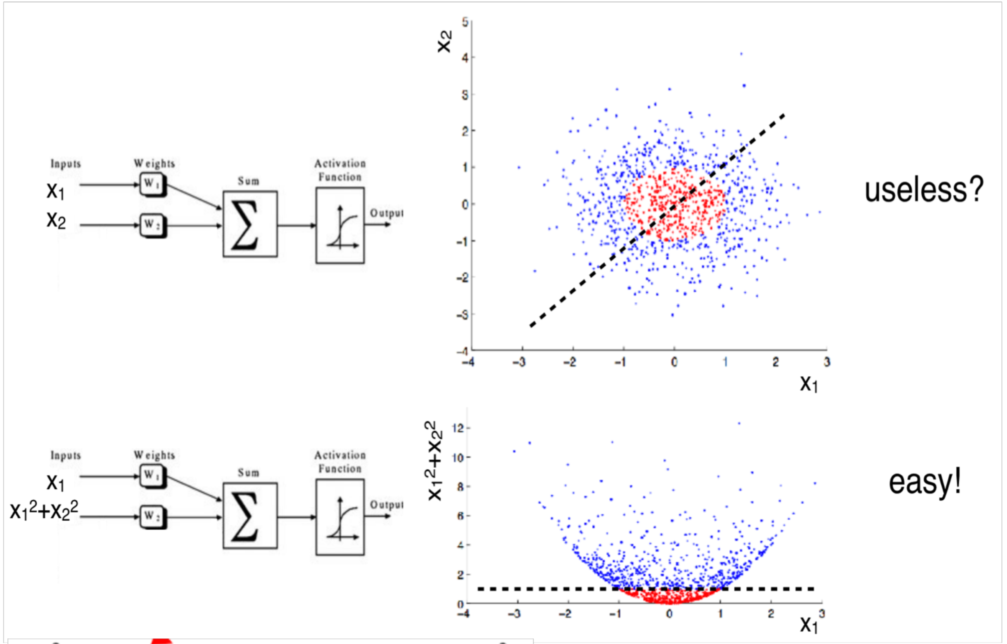
A common answer

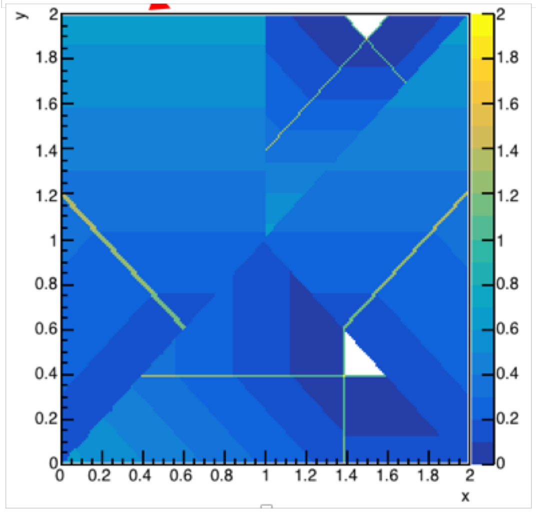
- In classical Machine Learning we had been interested in **feature** engineering, finding the best variable to solve the problem easily
- In a deep NN the first layer(s) can construct such high level features themselves from low level features

Another aspect, they can create **iterative structures** and smoother structures with the same number of parameters

What ever it is, a NN is a very expressive function approximator that uses a large number of parameter to model complex problems

- Astonishing that they can be trained with sufficient data
  - How do we train an NN?



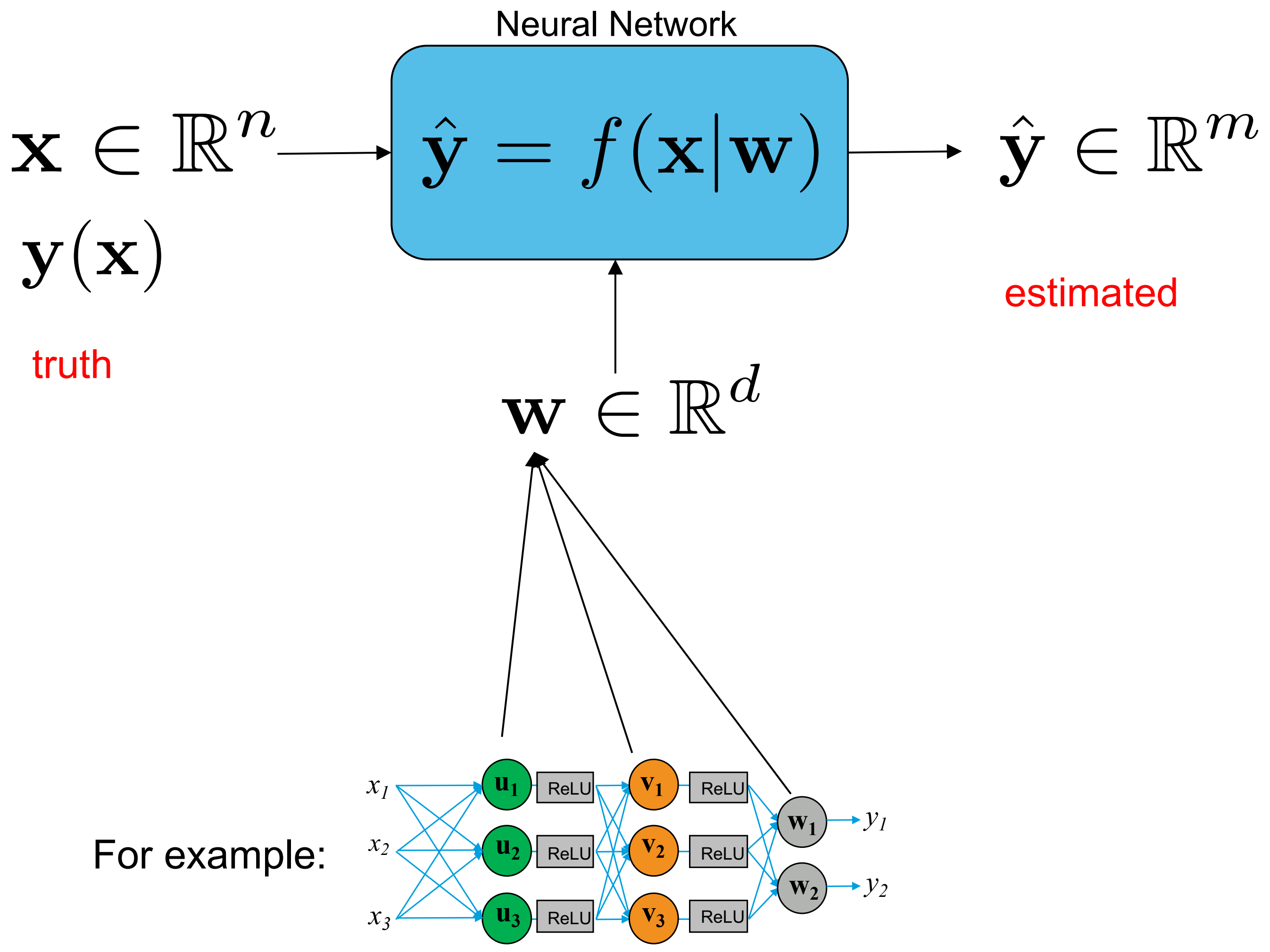


2d – 3 layers

Loss Function - Regression

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: $\mathbf{y}(\mathbf{x})$
  - We know the true values since we do supervised learning



Loss Function - Regression

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 (or Cost)**
  - E.g. Least Squares or MSE (Mean Squared Error)
- **Minimize loss** wrt. to weights w

Neural Network

$$\begin{tikzcd}
\mathbf{x} \in \mathbb{R}^n \arrow[r] & \boxed{\hat{\mathbf{y}} = f(\mathbf{x}|\mathbf{w})} \arrow[r] & \hat{\mathbf{y}} \in \mathbb{R}^m \\
\mathbf{y}(\mathbf{x}) & \mathbf{w} \in \mathbb{R}^d \arrow[u] &
\end{tikzcd}$$

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

minimizing w $\Leftrightarrow$ training

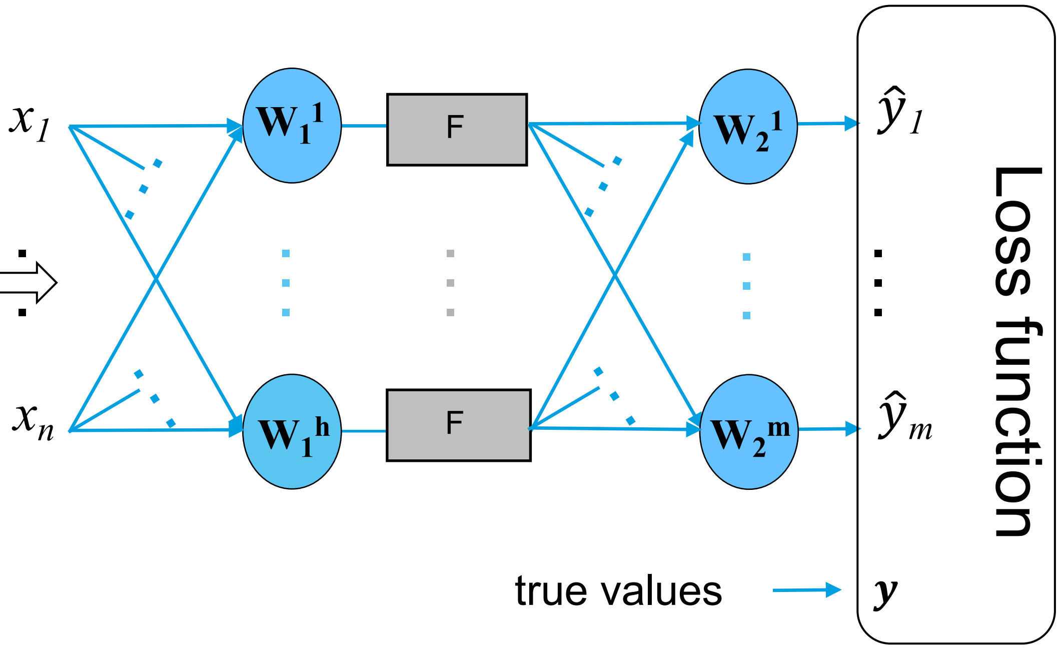
$$loss_{MSE}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{m}\sum_{i}^{m}(\hat{y}_i - y_i)^2$$

Loss minimization

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

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

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



- To minimize the loss we take the derivative wrt. to ***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 – from the end to the beginning**

Backpropagation

From loss to weights

- The previous chain rule calculation is known as **Backpropagation**
- The deviation between true and estimated ***y***, i.e. the loss, is back-propagated to a linear change of the weights.
  - Invented by different people in the 1960/70s
- **NB: The chain rule creates a chain of factors that can be evaluated numerically**

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

We calculate a **gradient** with respect to the weights/biases

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

- These are vector and matrix multiplication. If you need a Matrix calculus primer or work it out with tensor indices but in reality …

How to calculate all these derivatives?

Automatic Differentiation

- Deep Learning libraries are able to calculate the derivative of a piece of code
  - This is **not** a numerical approximation (no small epsilon)
  - This **not** symbolic differentiation (not as in e.g. Mathematica)
  - Think about it as **a derivative of your python code**

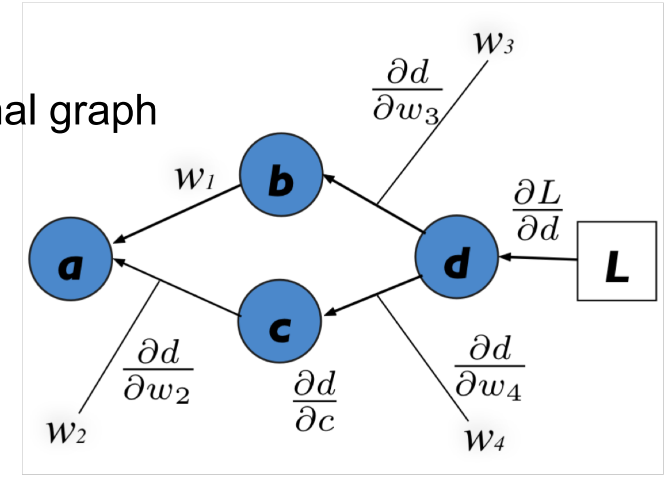
- It records your calculations (Forward step)
- It then calculates by applying the chain rule (Backward step) the gradients at the same numerical value
- The DL library keeps track of **hundreds of thousands of weights and more**
- There are different approaches to automatic differentiation

The different libraries e.g. Tensorflow, Pytorch etc. follow slightly different concepts

Important to understand if you want to define your own special layers and loss function

computational graph

```
b = w1 * a
c = w2 * a
d = (w3 * b) + (w4 * c)
L = f(d)
```



w_3
$\frac{\partial d}{\partial w_3}$
w_1
b
$\frac{\partial L}{\partial d}$
a
d
L
c
$\frac{\partial d}{\partial w_2}$
$\frac{\partial d}{\partial c}$
$\frac{\partial d}{\partial w_4}$
w_2
w_4

How do we do all this Numerical Linear Algebra?

Vectorization and GPU

- Efficient matrix manipulation is needed to make things fast
  - All python Deep Learning libraries provide **NumPy-** style operations
    - NumPy is the most common tool for scientific, numerical calculation
  - We need to go as close to the “metal” as possible
    - **BLAS etc.**

BTW NumPy, pandas and all that

- Vanderbilt University's Department of Biostatistics
- Short Terascale Python and NumPy course
  (skip the docker installation part)

What is BLAS?

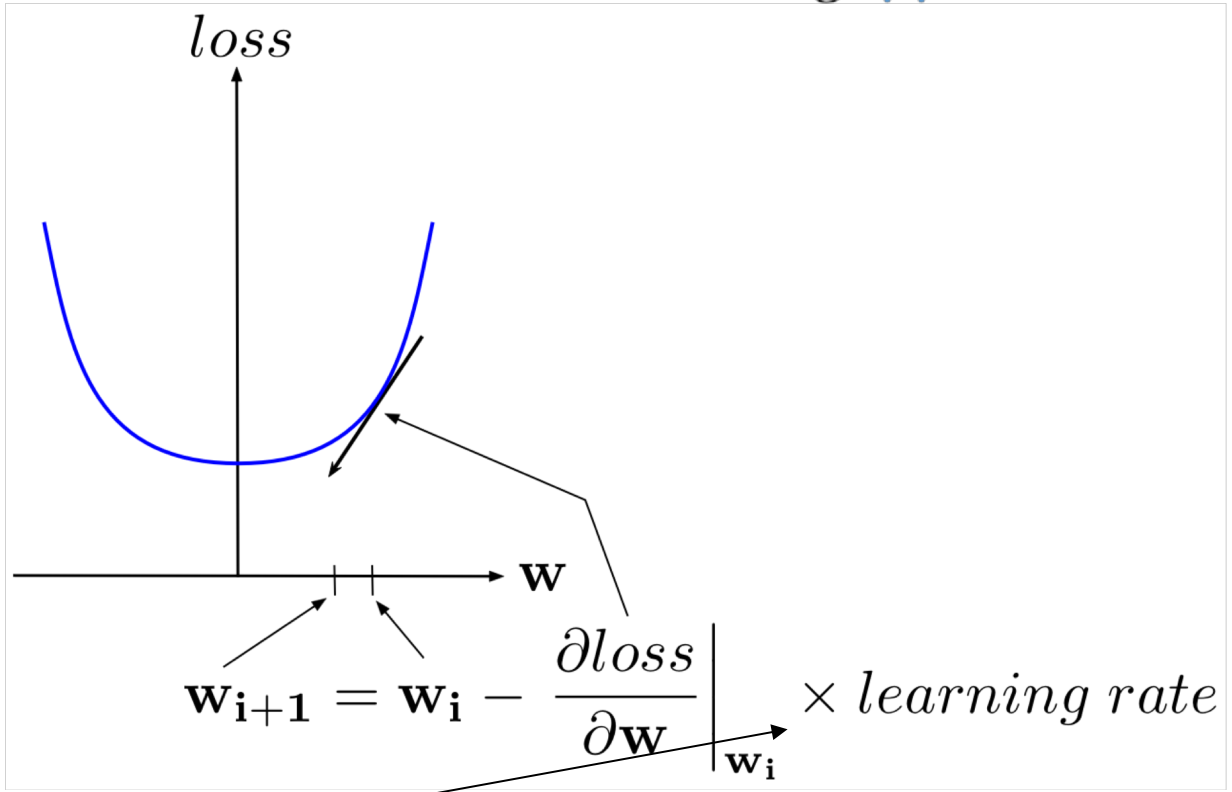
- **Basic Linear Algebra Subprograms** (BLAS)
  - a **set of low-level routines** for common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication
- Defines an interface to make libraries interchangeable
- BLAS implementations take advantage of special floating point hardware e.g. **SIMD** instructions.
  - SIMD (**Single instruction, multiple data**)
    vector register
  - All modern **CPUs** come with these abilities
    - SSE, AVX, AVX512, ...
  - And **GPUs**
    - Highly parallel - Latest NVidia 2080 Ti (Turing) 4352 cores
  - Rarely used directly in programming but by optimized libraries
    - openBLAS or **MKL** (Intel) or Apple's Accelerate framework
      - Multithreading!
    - **cuBLAS** (NVIDIA)
  - And other libraries e.g. cuDNN

Gradient Descent

Iterative minimizing

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

- The previous chain rule calculation is known as **Backpropagation**
- We can not solve this directly due to the size of the problem and the non-linear activation functions
- But we solve **numerically and iteratively** 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**



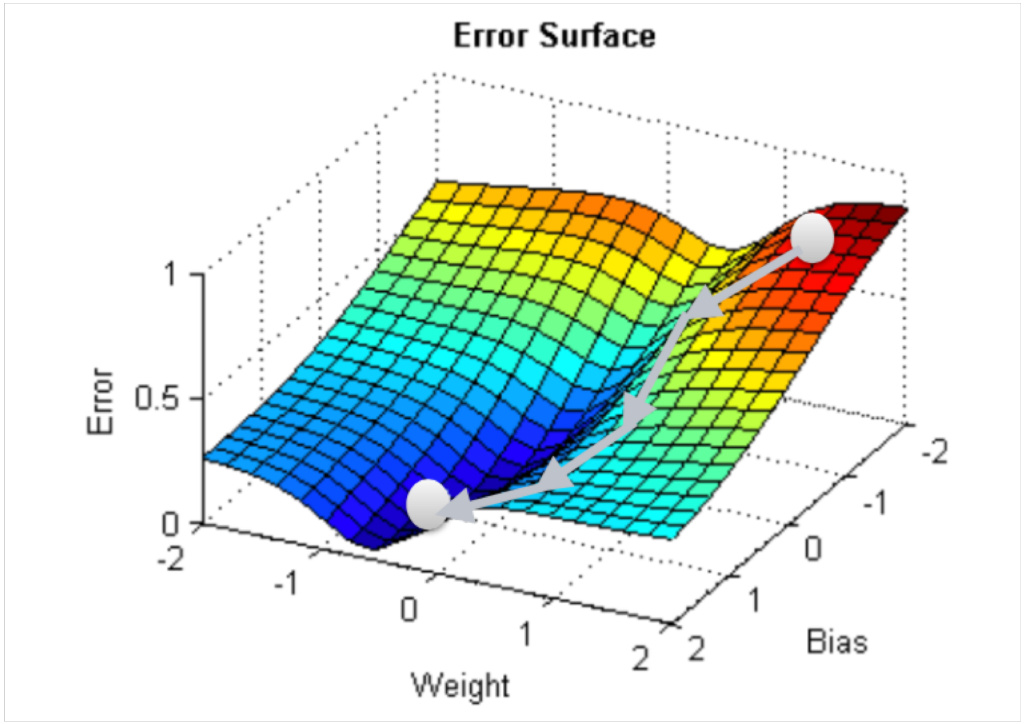
$$\mathbf{w_{i+1}} = \mathbf{w_i} - \left.\frac{\partial loss}{\partial \mathbf{w}}\right|_{\mathbf{w_i}} \times learning\ rate$$

Gradient Descent

Iterative minimizing

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

- The previous chain rule calculation is known as **Backpropagation**
- We can not solve this directly due to the size of the problem and the non-linear activation functions
- But we solve **numerically and iteratively** 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**

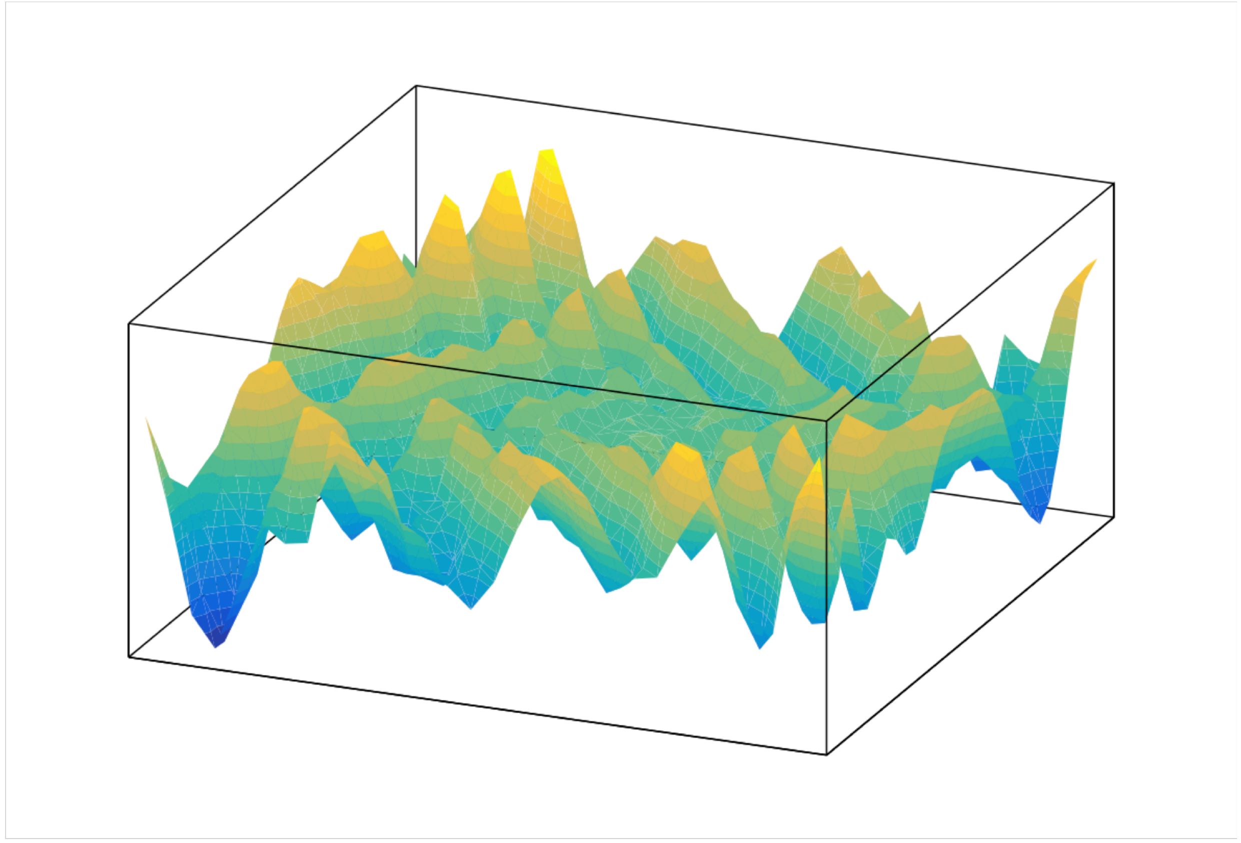


Gradient Descent

Iterative minimizing

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

- The previous chain rule calculation is known as **Backpropagation**
- We can not solve this directly due to the size of the problem and the non-linear activation functions
- But we solve **numerically and iteratively** 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**

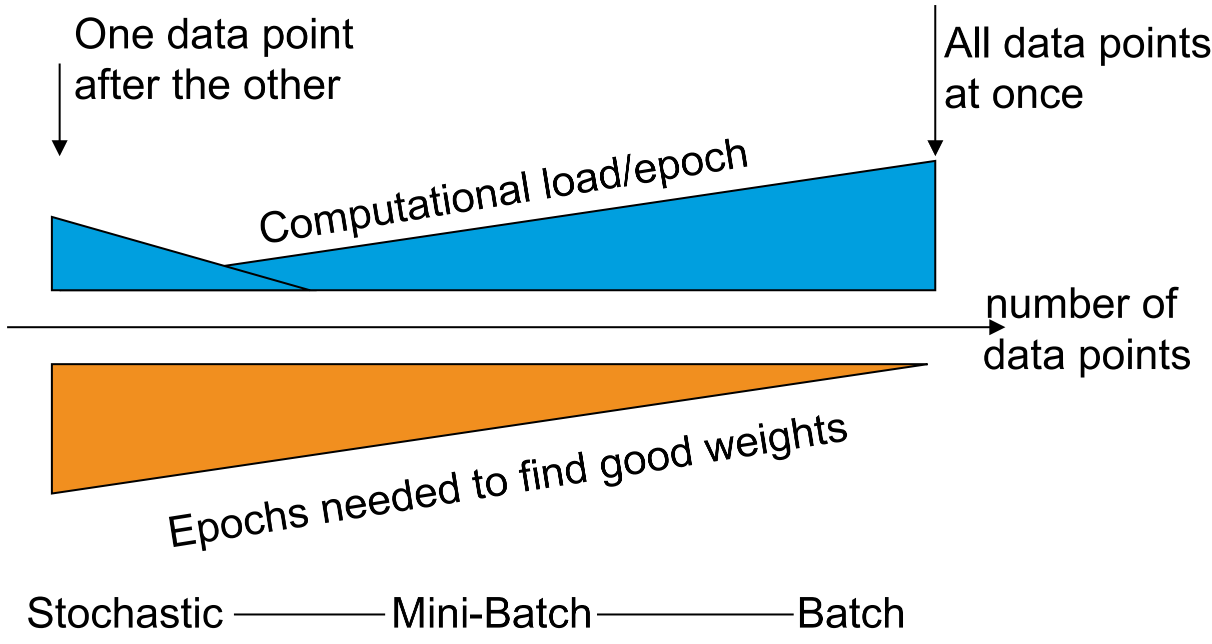


- In general a complicated high dimensional surface with **many local minima**
  Shaking helps i.e. add some noise ⟶ **Stochastic** Gradient Descent

Stochastic Gradient Descent - SGD

That's were the magic comes from

- If we do gradient descent we have the choice between 2 alternatives
  - **Batch gradient descent**
    Take *all* data and calculate then the loss and update the weights
  - **Stochastic gradient descent (SGD)**
    Take just *one* data point and calculate the loss and update the weights
  - When all data points have been used we call it an **Epoch**
  - **Batch size** typically means mini-batch
  - Talking about loss on the slides before I had been a bit unprecise we optimize the **average loss over a** (mini-) **batch**



- Medium batch size can be processed efficiently (BLAS, cuBLAs)
- Stochastic gradient descent introduces some noisiness which can be an advantage to avoid local minima
- SGD converges faster - frequent updates
- The exact size is a hyperparameter that can be tuned - typical value 32-256 or more
  (2^n to fit into memory)

Optimizer

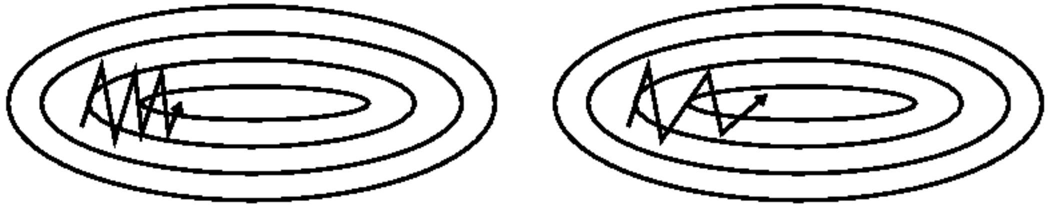
More on Stochastic Gradient Descent Algorithms

- There are many refinements available for the optimizer, often heuristic
  - **Momentum:** Gradient descent is often very slow near the optimum ⟶ Remember the last step
    - Creates an average oscillating part (changing gradient sign) is damped
- Different kind of adaptive behavior (lecture Hinton)

$$\Delta^i := \eta \frac{dL}{d\mathbf{W}}$$

η learning rate

$$\mathbf{W}^{i+1} = \mathbf{W}^i - \Delta^i$$



Gradient descent in a 'canyon', i.e. gradient different in the 2 dimensions without and with **momentum**

$$\Delta^i := \gamma\Delta^{i-1} + \eta \frac{dL}{d\mathbf{W}}$$

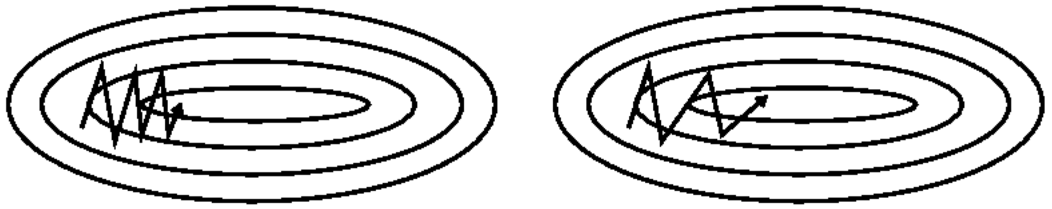
Optimizer

More on Stochastic Gradient Descent Algorithms

$$\Delta^i := \eta \frac{dL}{d\mathbf{W}}$$ η learning rate

$$\mathbf{W}^{i+1} = \mathbf{W}^i - \Delta^i$$

- There are many refinements available for the optimizer, often heuristic
  - **Momentum:** Gradient descent is often very slow near the optimum ⟶ Remember the last step
    - Creates an average oscillating part (changing gradient sign) is damped
- Different kind of adaptive behavior (lecture Hinton)



Gradient descent in a 'canyon', i.e. gradient different in the 2 dimensions without and with **momentum**

$$\Delta^i := \gamma\Delta^{i-1} + \eta\frac{dL}{d\mathbf{W}}$$

BTW remember the gradient calculation:

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

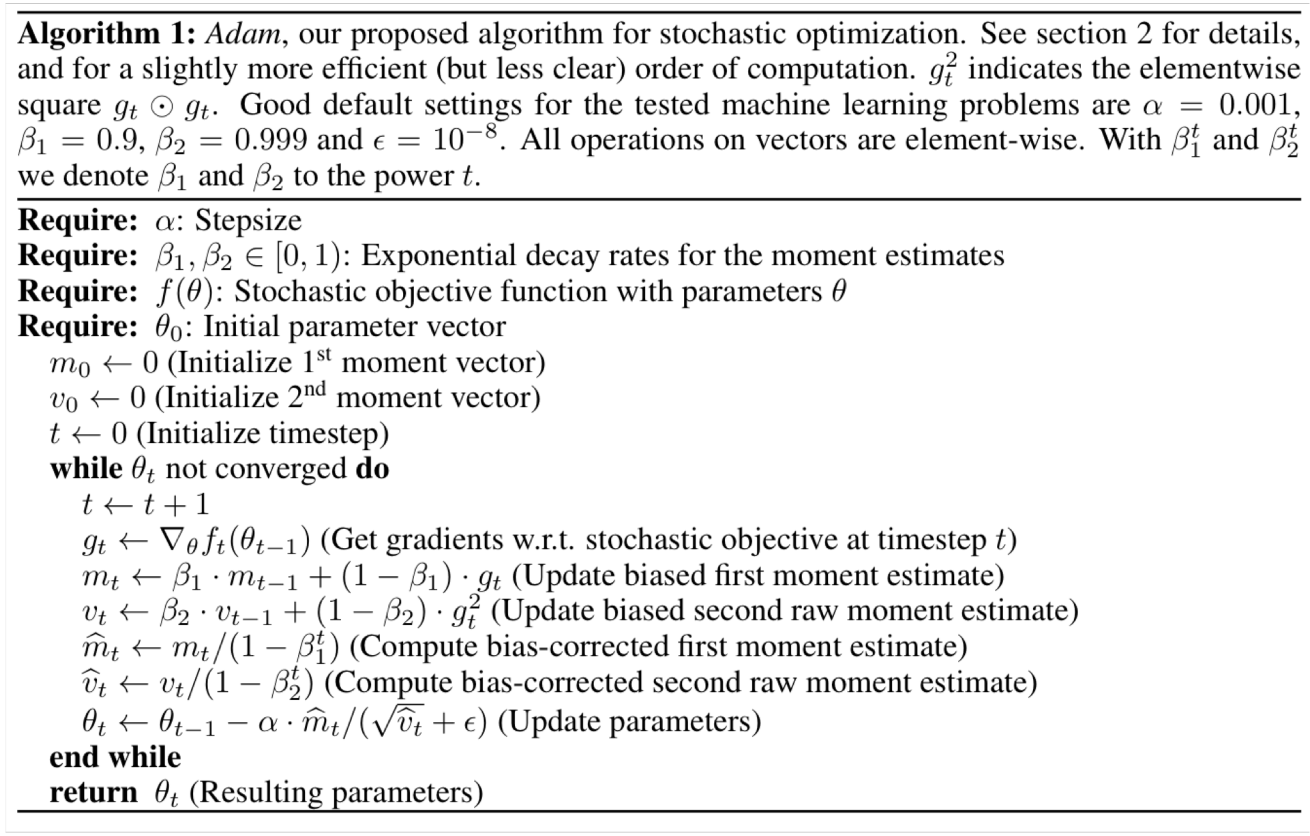
Normalizing input data to zero mean helps to learn

Optimizer

More on Stochastic Gradient Descent Algorithms

- There are many refinements available for the optimizer, often heuristic
  - **Momentum:** Gradient descent is often very slow near the optimum ⟶ Remember the last step
    - Creates an average oscillating part is damped
- Different kind of adaptive behavior (lecture Hinton)
  - Would like to have an adaptive learning rate for each weight etc. – **developed over the last 20-30y**
  - Popular and efficient choice at present: **ADAM** (Adaptive Moment Estimation, 2014)
  - The general **SGD** of your DL Library
  - Defaults will do the job in most cases after **playing a bit to find a working learning rate**
  - **Batch size and learning rate a correlated**

ADAM:



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

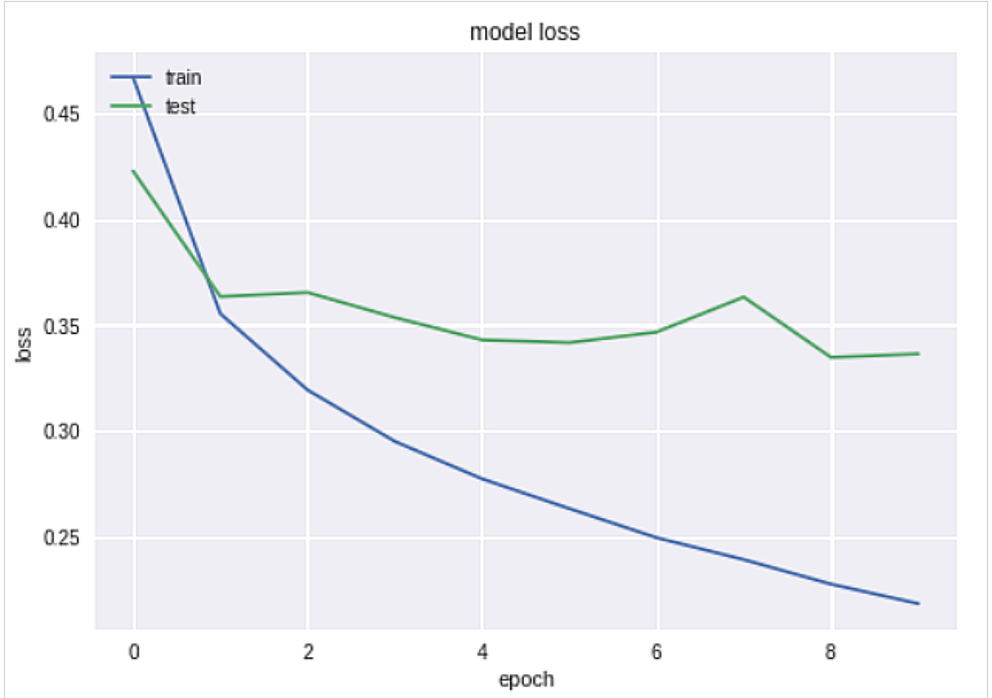
```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

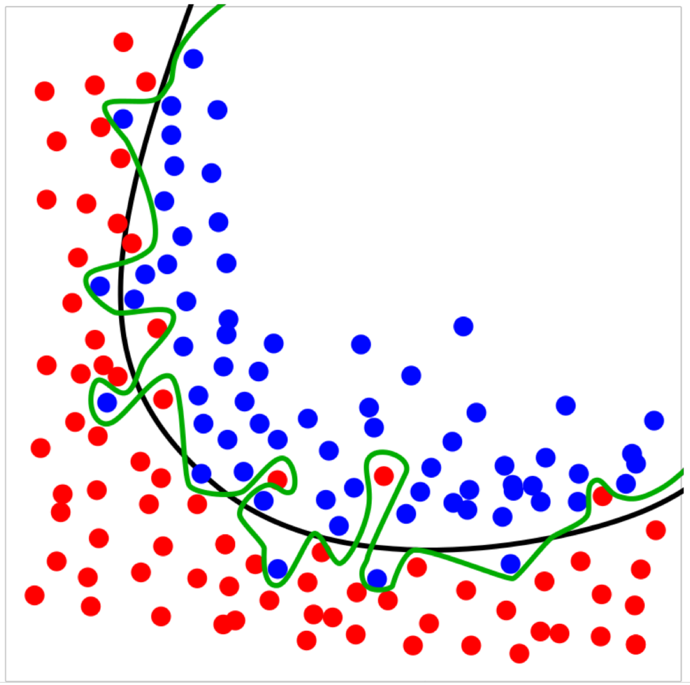
Overview: https://arxiv.org/abs/1609.04747
Seminal paper by LeCun (1998)

Learning curve

Keep an eye on the iterative learning!

- It is useful to observe the learning process over several epochs
  - **Epoch** == when all data has been used
    - For the iterative learning we use all data randomly in mini-batches until all data has been used and then we start from the beginning with the same data
- A too large learning rate will result in jumpy or even non converging behavior (not shown in this example)
- A too small will not learn at all/fast enough
- You need an independent data sample: **test data !**
  - Sufficiently large not to be fooled by statistical fluctuations
- At some point the loss/metric on test data will become worse than on the trainings data ⟶ **biased on trainings data**
  - Your net learns accidental patterns in your trainings data that does not **generalize**
- At some point the loss/metric may become worse on the test data ⟶ **overtraining**



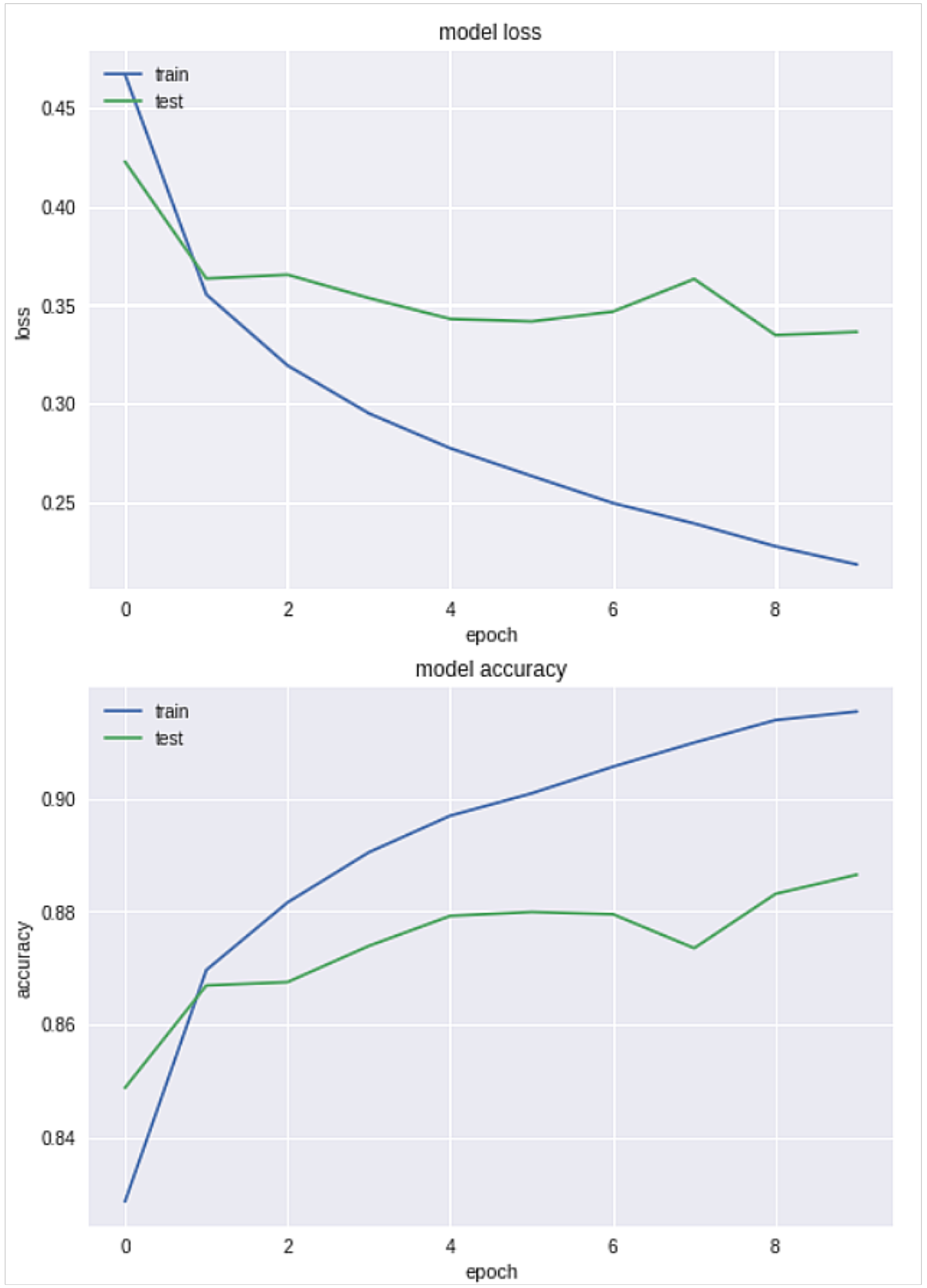


Learning curve

Keep an eye on the iterative learning!

- It is useful to observe the learning process over several epochs
  - **Epoch** == when all data has been used
    - For the iterative learning we use all data randomly in mini-batches until all data has been used and then we start from the beginning with the same data
- A too large learning rate will result in jumpy or even non converging behavior (not shown in this example)
- A too small will not learn at all/fast enough
- You need an independent data sample: **test data !**
  - Sufficiently large not to be fooled by statistical fluctuations
- At some point the loss/metric on test data will become worse than on the trainings data ⟶ **biased on trainings data**
    - Your net learns accidental patterns in your trainings data that does not **generalize**

• Extra metric, for example in classification
Accuracy=(correct labels)/(all labels)
as additional performance measure or **metric**



Short recap

A NN is

- A set of layers + Activation function
- We train by stochastic gradient descent
- The DL library handles all the complicated derivatives by automatic differentiation
- For speed-up vector units or GPU together with highly optimized numerical libraries
  - presenting our data in (mini-)batches
  - Mini-batches are anyhow good for the extra noise
- Generalization capability of the trained network must be checked on a test sample

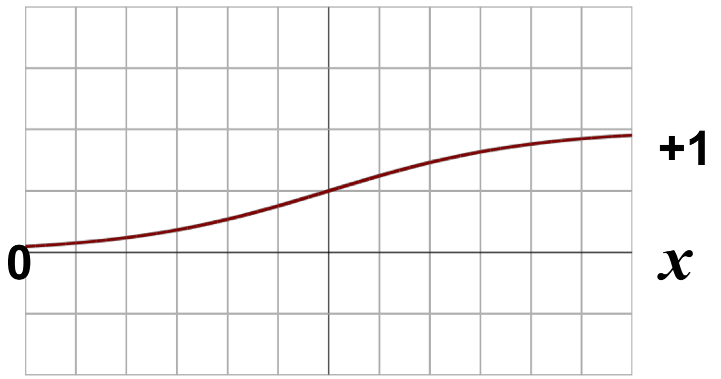
- Dense Network
- MSE
- Stochastic Gradient Descent
  - Learning rate
  - Momentum
  - SGD
  - Adam
- Learning curve
- Train-test splitting
- Cross entropy
- Softmax

Binary Classification – Cross Entropy Loss

More on loss functions - Classification

- For classification we need a different loss functions
- **Binary Cross Entropy**
- For the last node we take a **Sigmoid** or **Logistic function**

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



0 1

Binary classification

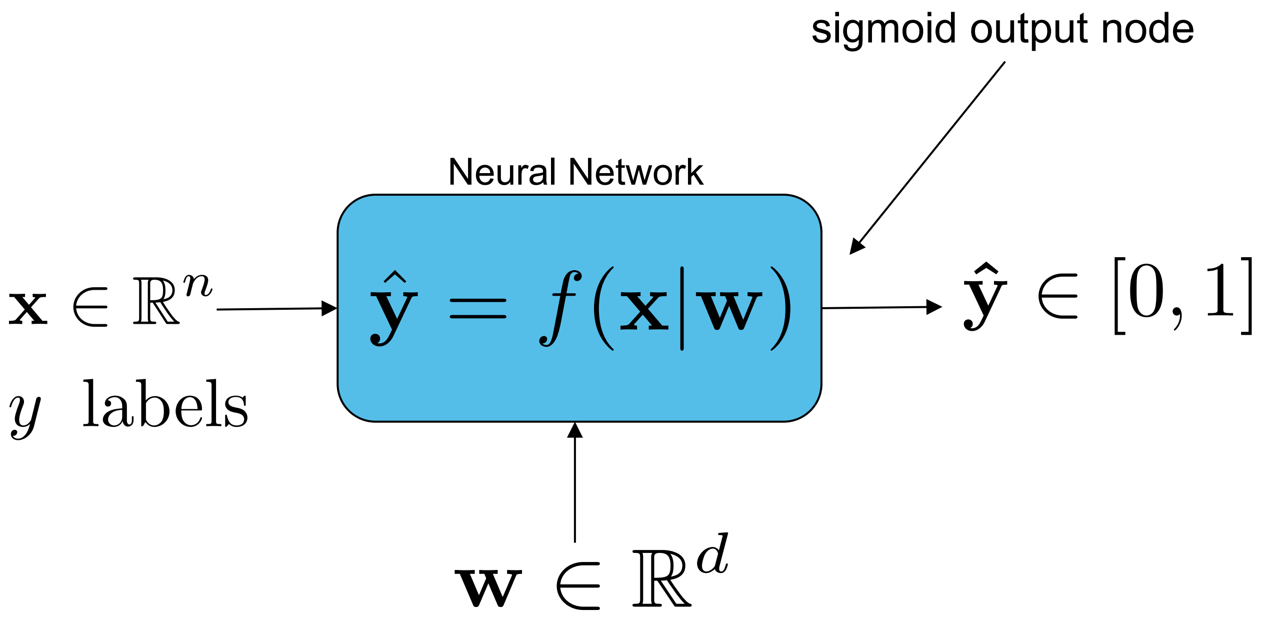
signal or background

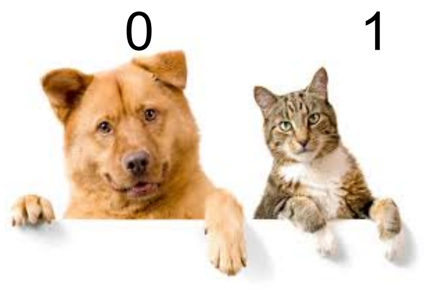
What to do if our true values are not real numbers but just integer labels 0,1 ?

Binary Cross Entropy

Binary what?

- We want to do **binary classification**:
  - We have **labeled** trainings data: Background vs signal
  - Some input variables x_i label $\boldsymbol{y_i} \in \{\mathbf{0}, \mathbf{1}\}$
    Note: y is **discrete** now
  - We want to get a **probability** $\hat{y} \in [\mathbf{0}, \mathbf{1}]$ that we have a signal
  - Cat: $$P(1|\mathbf{x}_i, \mathbf{w}) = \hat{y}_i$$
    Dog: $$P(0|\mathbf{x}_i, \mathbf{w}) = 1 - \hat{y}_i$$
  - That's a binomial model for which we can get the optimal weights $\mathbf{w}$ by a **Maximum Likelihood fit**
  - The likelihood for one batch of n events with $n = n_s + n_b$ and the $-lnL$


sigmoid output node
Neural Network
$\hat{\mathbf{y}} = f(\mathbf{x}|\mathbf{w})$
$\mathbf{x} \in \mathbb{R}^n$
y labels
$\hat{\mathbf{y}} \in [0, 1]$
$\mathbf{w} \in \mathbb{R}^d$



Binary Cross Entropy

Binary what?

- We want to do **binary classification**:
  - We have **labeled** trainings data: Background vs signal
  - Some input variables x_i label $y_i \in \{0,1\}$ Note: y is **discrete** now
  - We want to get a **probability** $\hat{y} \in [0,1]$ that we have a signal
  - signal (cat): $P(1|\mathbf{x}_i, \mathbf{w}) = \hat{y}_i$

    bckgrnd (dog): $P(0|\mathbf{x}_i, \mathbf{w}) = 1 - \hat{y}_i$
  - That's a binomial model for which we can get the optimal weights $\mathbf{w}$ by a **Maximum Likelihood fit**
  - The likelihood for one batch of n events with $\boldsymbol{n=n_s+n_b}$ and the *-lnL*

$$L(\mathbf{w}|batch) = \prod_{i=1}^{n_s} \hat{y}_i \prod_{j=1}^{n_b} (1 - \hat{y}_j)$$

$$-\ln L(\mathbf{w}|batch) = -\sum_{i=1}^{n_s} \ln \hat{y}_i - \sum_{j=1}^{n_b} \ln(1 - \hat{y}_j)$$

using the label

$$= -\sum_{i=1}^{n} \left[y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_j)\right]$$

Binary Cross Entropy

Binary what?

- We want to do **binary classification**:
  - We have **labeled** trainings data: Background vs signal
  - Some input variables x_i label $y_i \in \{0,1\}$
    Note: y is **discrete** now
  - We want to get a **probability** $\hat{y} \in [0,1]$ that we have a signal
  - signal (cat): $P(1|\mathbf{x}_i, \mathbf{w}) = \hat{y}_i$

    bckgrnd (dog): $P(0|\mathbf{x}_i, \mathbf{w}) = 1 - \hat{y}_i$

  - That's a binomial model for which we can get the optimal weights $\mathbf{w}$ by a **Maximum Likelihood fit**
  - The likelihood for one batch of n events with $n{=}n_s{+}n_b$ and the $-lnL$

$$L(\mathbf{w}|batch) = \prod_{i=1}^{n_s} \hat{y}_i \prod_{j=1}^{n_b} (1 - \hat{y}_j)$$

$$-\ln L(\mathbf{w}|batch) = -\sum_{i=1}^{n_s} \ln \hat{y}_i - \sum_{j=1}^{n_b} \ln(1 - \hat{y}_j)$$

using the label

$$= -\sum_{i=1}^{n} [y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_j)]$$

- This expression is called **cross entropy** (up to 1/n)
- Minimizing this expression provides the best optimal weight

Binary Cross Entropy

Why is it called cross entropy?

$$\text{Entropy} \equiv \int ln(p(x))\, p(x)\; dx$$

$$\text{Cross Entropy} \equiv \int ln(p(x))\, q(x)\; dx$$

- Reminder: Entropy is the averaged <$ln(p)$>
  - Shannon Entropy for some thing distributed with p
- Cross entropy is averaged over a different distribution q
- Gibb's inequality

$$\int ln(p)\, p\; dx \leq \int ln(p)\, q\; dx$$

- **Cross Entropy Loss**
  is a measure how similar two distributions are

$$-\frac{1}{n}\sum_{i=1}^{n}\left[\underset{\text{data}}{\underset{\uparrow}{y_i}} \log(\underset{\text{network}}{\underset{\uparrow}{\hat{y}_i}}) + (1-y_i)\log(1-\hat{y}_i)\right]$$

- The minimum is $p(x)=q(x)$. Minimizing makes similar

- Can be easily generalized to more then 2 classes

And if we have more than 2 classes?

Softmax output node

- For classification we need different loss
- **Multi-class** classification - {1,..,K} labels
  - Similar logic, replace the binomial model by a multinomial model
  - Softmax (smooth version of max/multidim version of logistic function

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$

For example 10 classes: typical output is a vector of probabilities for the different classes that add up to 1

(one hot encoding) true: [0,0,0,0,0,0,1,0,0,0]
predicted: [0.005, 0.007, 0.011, 0.018, 0.032, 0.019, 0.830, 0.005, 0.042, 0.031]

used together with **sparse cross entropy**

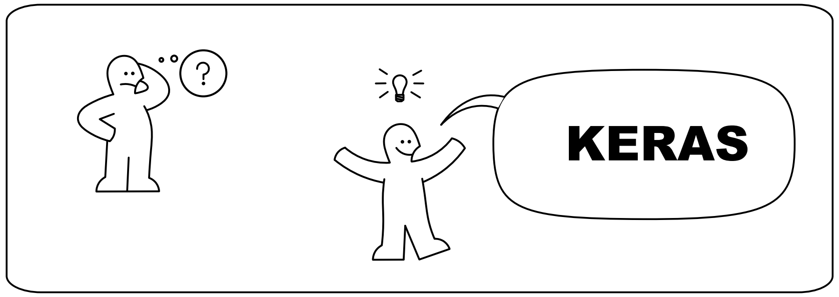
Summary

A neural network ...

- is a universal function approximator
- is a chain of linear transformations and non-linear activation functions,
- is able to learn everything,
- learns by modifying the **weights/biases**,
- is trained by **backpropagation** and **stochastic gradient descent** using the gradient with respect to the weights/biases
  - there are several heuristic algorithms for a adaptive SGD, eg **Adam**
- uses **cross entropy** for discrete labels, i.e. classification
  - with a **logistic function** for binary labels or **softmax** as multiclass version
- **ReLU** is the most important (game changing) activation function
- At the end of the day, the only thing you have to do is to plug together prebuild pieces that implements the knowledge of decades
  - Standing on the shoulders of giants
  - Or more like putting together a Billy bookcase

NEURALA NÄTVERK

Assembly Instruction

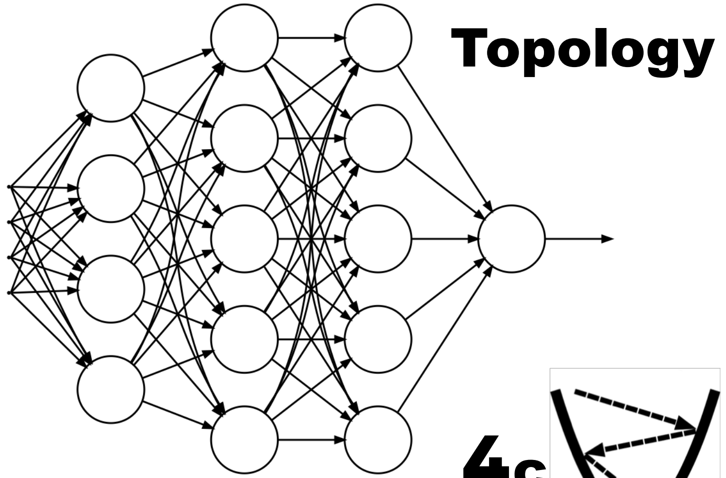


1 Features

2a Train Test

2b Normalize

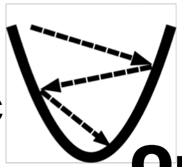
3 Metric

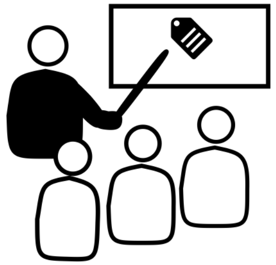


4a

4b Loss

4c Optimizer





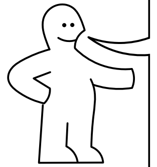
5 Fit

6 Overtraining

7 Apply!

NEURALA NÄTVERK

Assembly Instruction



1x python™

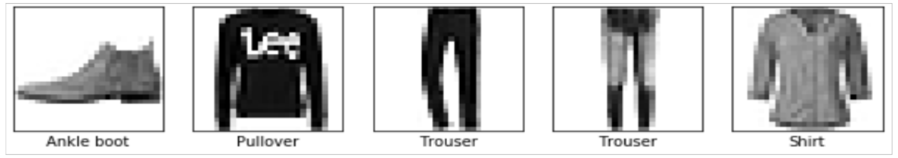
1x K + TensorFlow

1x

2x	**ReLU**
1x	**Flatten**
1x	**Softmax**
1x	**X-entropy**
1x	**Adam**

```
import tensorflow as tf
mnist = tf.keras.datasets.fashion_mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0 - 0.5, x_test / 255.0 - 0.5

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
=model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(x_train, y_train,
           validation_data=(x_test,y_test),
           batch_size=32, epochs=5 )
model.evaluate(x_test, y_test)
```

```
import numpy as np
fiveImages = x_test[0:5]
predictions = model.predict(fiveImages)
predictions = np.argmax(predictions,axis=1)
import matplotlib.pyplot as plt
class_names = ['T-shirt/top', 'Trouser', 'Pullover',
               'Dress', 'Coat', 'Sandal', 'Shirt',
                'Sneaker', 'Bag', 'Ankle boot']
plt.figure(figsize=(10,10))
for i in range(5):
    plt.subplot(1,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(fiveImages[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[predictions[i]])
plt.show()
```

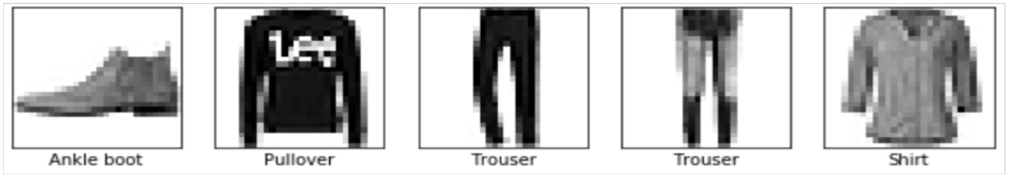


A Simple Keras Implementation

On the Fashion MNIST dataset - thanks to Zalando

- Link to this example on Google’s free collab
- 1. File Menu: Save a copy to your Google drive
- 2. click the: [ ]

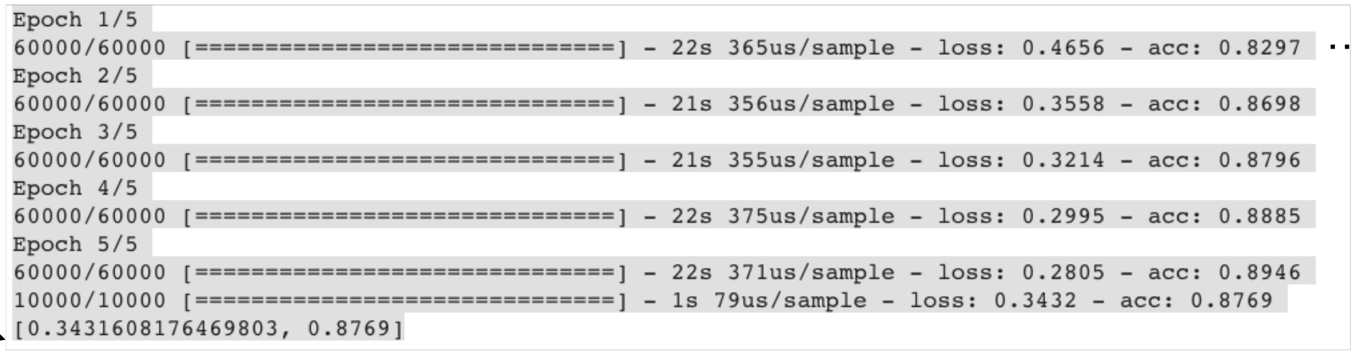
```
import tensorflow as tf
mnist = tf.keras.datasets.fashion_mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0 - 0.5, x_test / 255.0 - 0.5

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
model,summary()
model.compile(optimizer='adam’,
              loss='sparse_categorical_crossentropy’,
              metrics=['accuracy'])

model.fit(x_train, y_train,
    validation_data=(x_test,y_test),
    batch_size=32, epochs=5
)

model.evaluate(x_test, y_test)
```



Input dimension: 28x28=784
10 classes
60k trainings data, 10k test data
~70 millions trainings pixel

1818 Nodes*
Trainable params: **669706**

Run times: batch size of 32 on a CPU



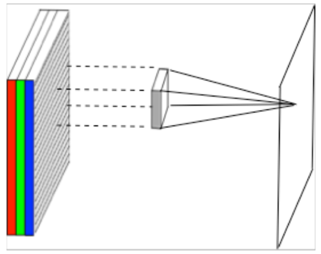
```
Epoch 1/5
60000/60000 [==============================] - 22s 365us/sample - loss: 0.4656 - acc: 0.8297 ...
Epoch 2/5
60000/60000 [==============================] - 21s 356us/sample - loss: 0.3558 - acc: 0.8698
Epoch 3/5
60000/60000 [==============================] - 21s 355us/sample - loss: 0.3214 - acc: 0.8796
Epoch 4/5
60000/60000 [==============================] - 22s 375us/sample - loss: 0.2995 - acc: 0.8885
Epoch 5/5
60000/60000 [==============================] - 22s 371us/sample - loss: 0.2805 - acc: 0.8946
10000/10000 [==============================] - 1s 79us/sample - loss: 0.3432 - acc: 0.8769
[0.3431608176469803, 0.8769]
```

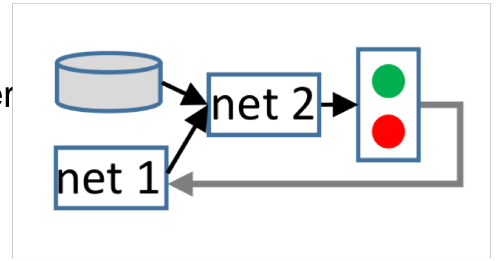
*Drosophila 100000 neurons - Human 100000000 neuron/7000 connections

Where to go from here?

Resources and further explorations

- PLAY with the code!
  - The example from the previous slide is complete, working code
  - **TF tutorials**
- **The fun starts when we look into complicated architectures,** there are
  - CNN (Convolutional NN) ⟹ Philipp Heuser
  - Autoencoder
- Generative
  - VAE
  - GAN (Generative) ⟹ Torben Ferber
- Recurrent (memory)
  - RNN
  - LSTM
- …





We only had a small DNN example

- Deep learning Resources
  just google, or click on my subjective list
  - https://www.deeplearningbook.org/
  - Wikipedia – but a wide range of quality
  - https://stats.stackexchange.com/
  - https://medium.com/topic/machine-learning
  - https://machinelearningmastery.com/start-here/
  - https://dataelixir.com/
  - https://towardsdatascience.com/
  - and many more including a lot of rubbish you have to judge yourself
  - Ceat sheets: https://becominghuman.ai/cheat-sheets-for-ai-neural-networks-machine-learning-deep-learning-big-data-678c51b4b463

Thank you

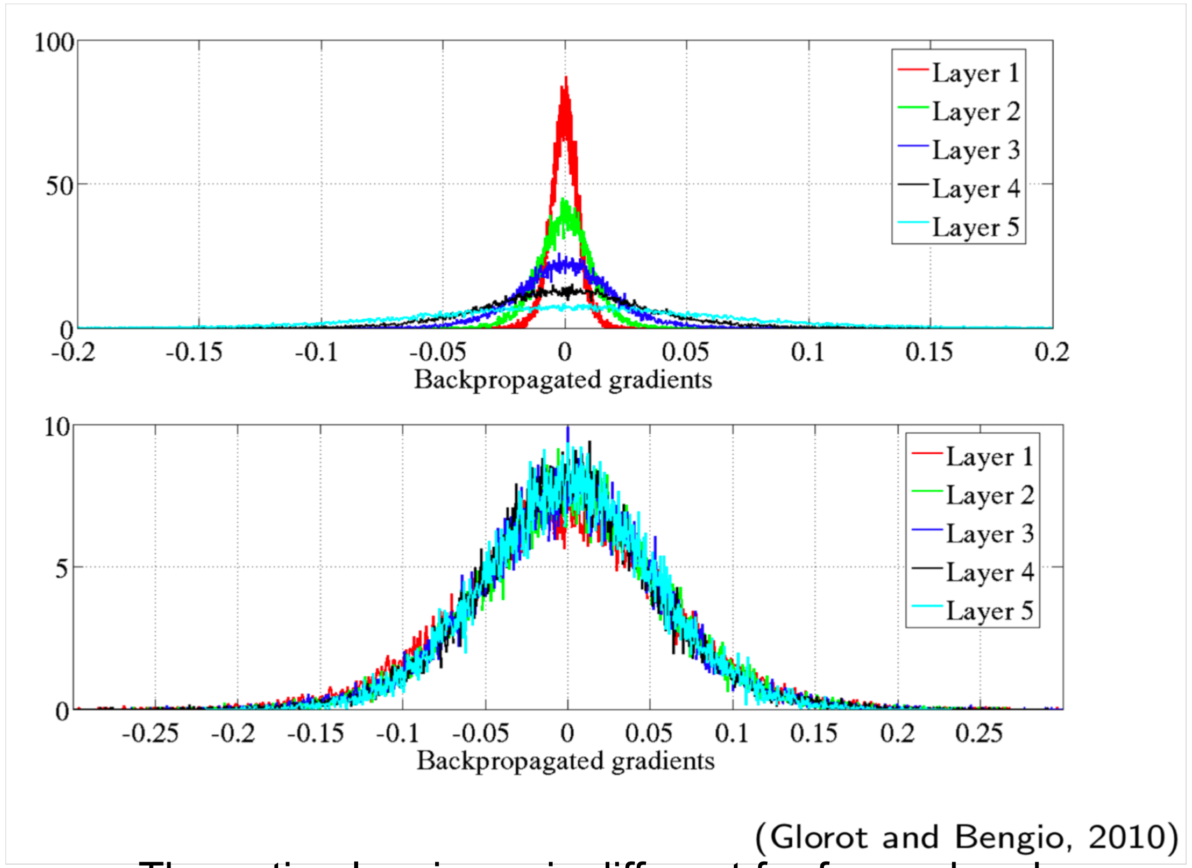
IDEA style thanks to https://idea-instructions.com/ (*CC by-nc-sa 4.0*)
Rosenblatt and overfitting image from Wikipedia

Xavier Initialization

Appendix I

Why do we care about the size of the weights?

- Forward view
  - Large weights will increase the signal after passing each layer. If we use a activation function as *tanh,* it will go into saturation.
  - Small weights will result in a signal that mainly stays in the linear part of the *tanh* around 0. Then our network starts to loose its non-linearity.
- Backward view
  - Deep neural networks become difficult to train because of the *vanishing gradient problem.* If we follow the gradient through the different layers, it tends to become smaller. The effect strongly depends on the initial size, i.e. variance of the weights
- This had been studied by Xavier Glorot and Yoshua Bengio (2010) and later by He at al. (2015)
  The problem can be mitigated by choosing the proper variance when randomly initializing the weights.



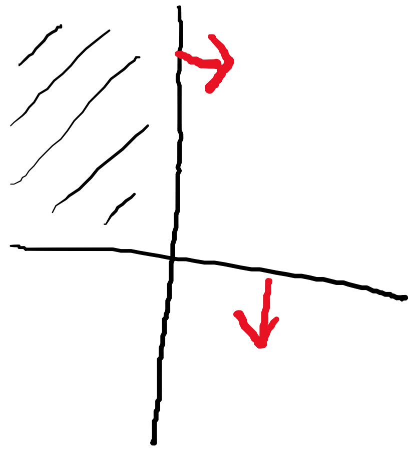
(Glorot and Bengio, 2010)

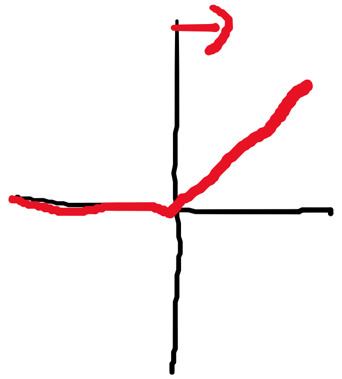
- The optimal variance is different for forward and backward step but they propose to initialize with $Var(w) = \frac{2}{N_{in}+N_{out}}$ for example by sampling uniformly from $\pm\sqrt{\frac{6}{N_{in}+N_{out}}}$ The details depend on the activation functions. ReLU had been studied by He et al.

Cutting hyperplanes

Appendix 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

Appendix 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
- For example: 10 dimensional input with 40 nodes may cut your phase space into up to 1.2 billion pieces

d = 2 (paper plane)

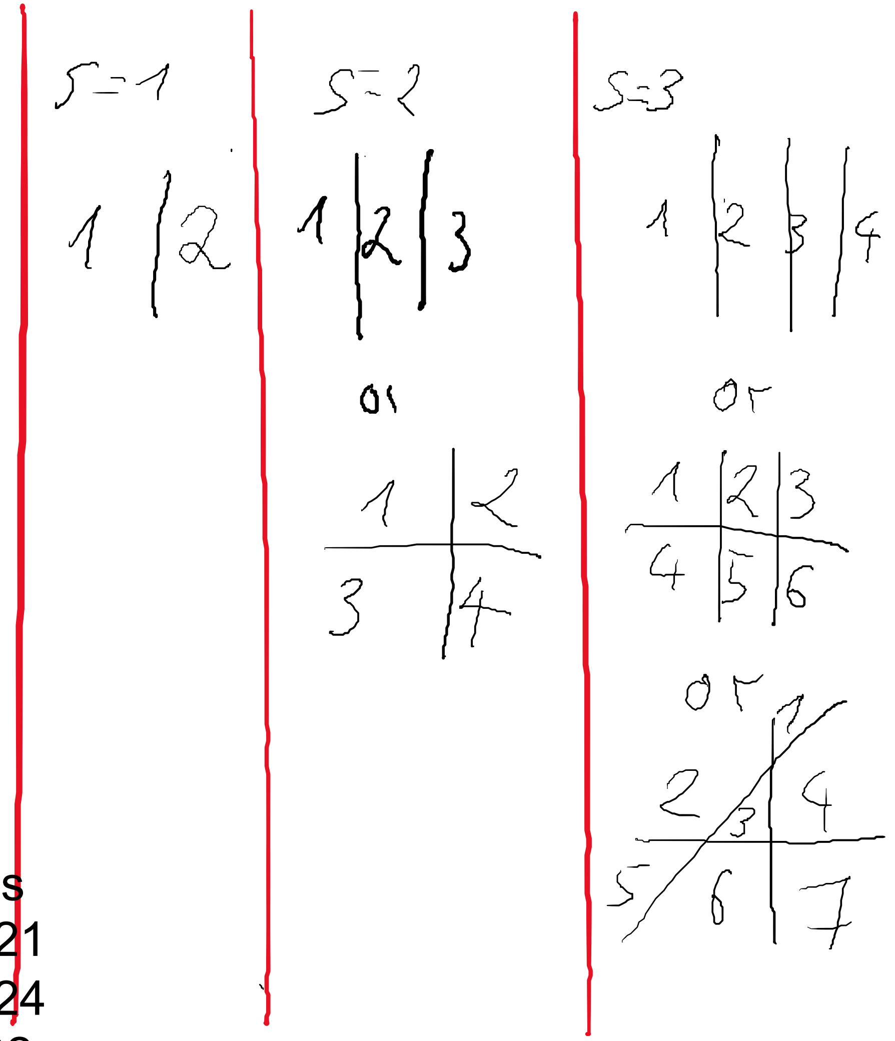
s = # hyperplanes (lines)

At least s+1

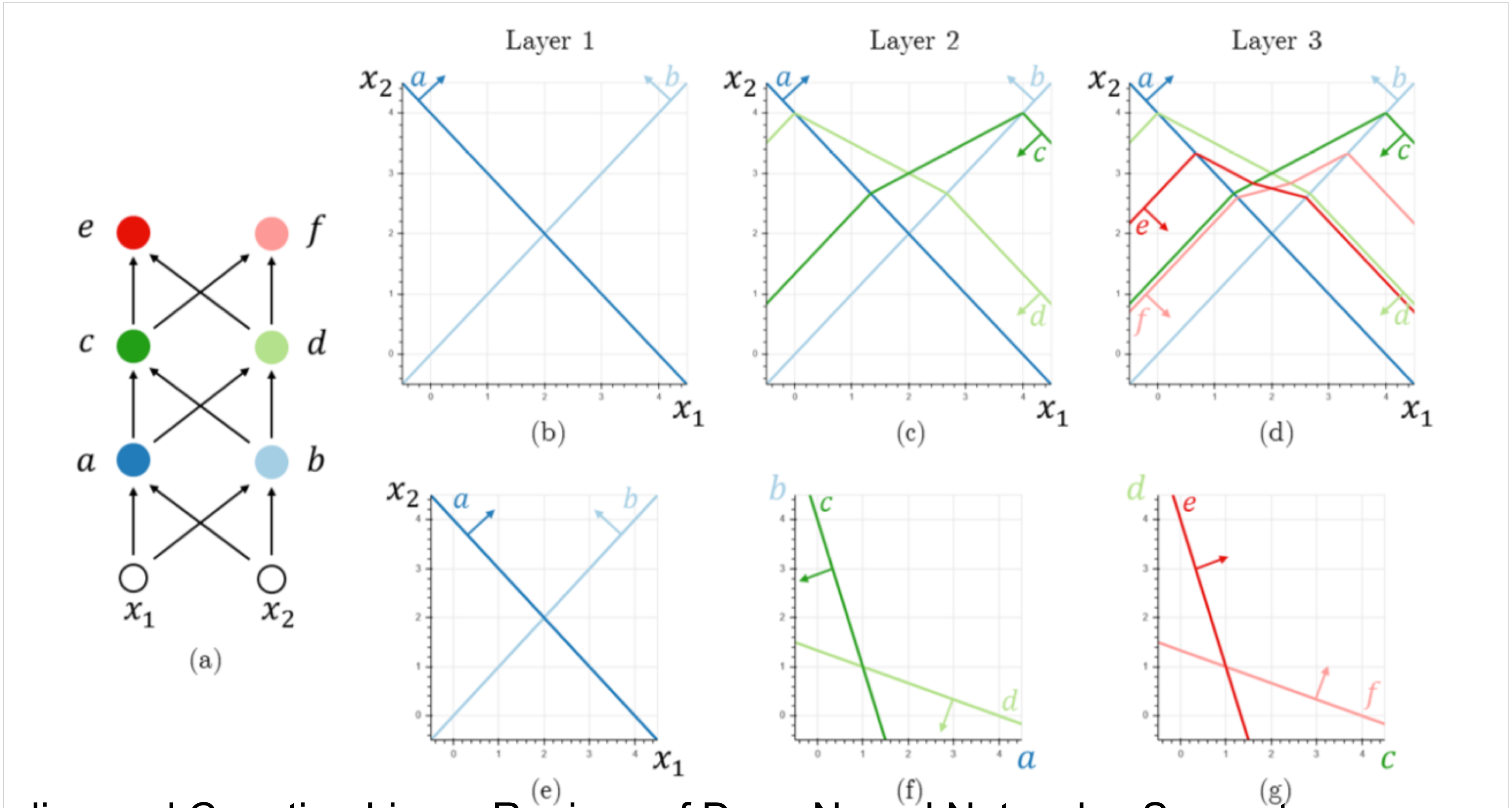
At most

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

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



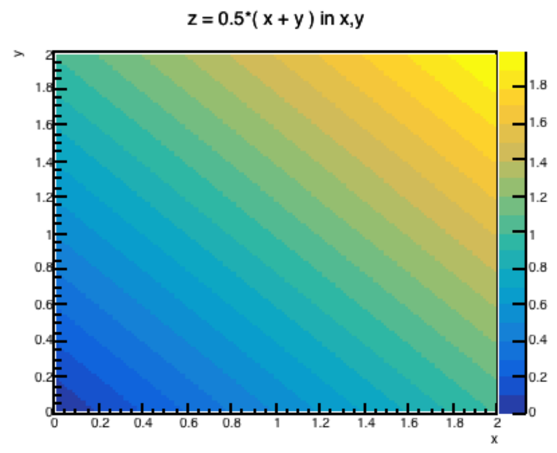
Cutting hyperplanes - DL



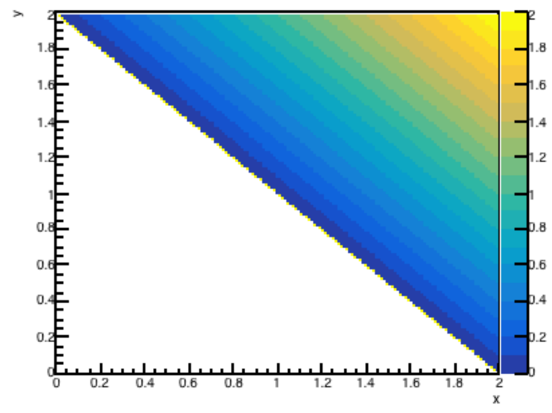
Bounding and Counting Linear Regions of Deep Neural Networks, Serra et al., https://arxiv.org/abs/1711.02114

1 layer

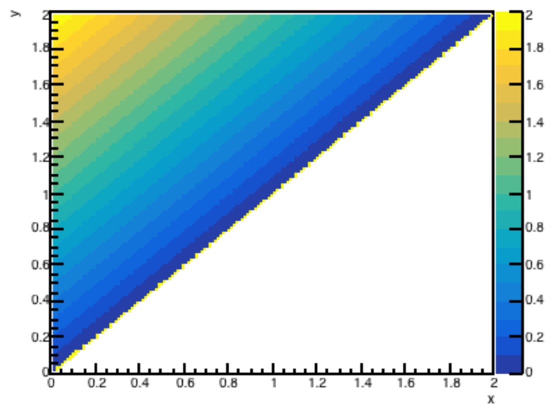
x, y <-[0,2]
z=0.5(x+y)

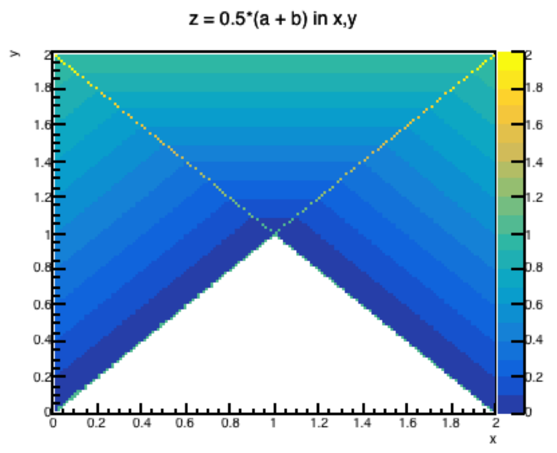


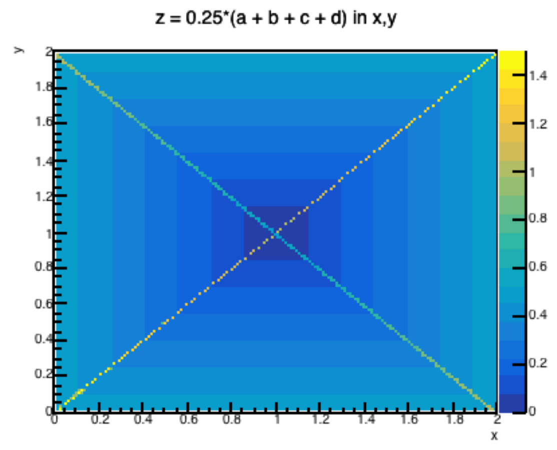
a=ReLU(x+y-2)



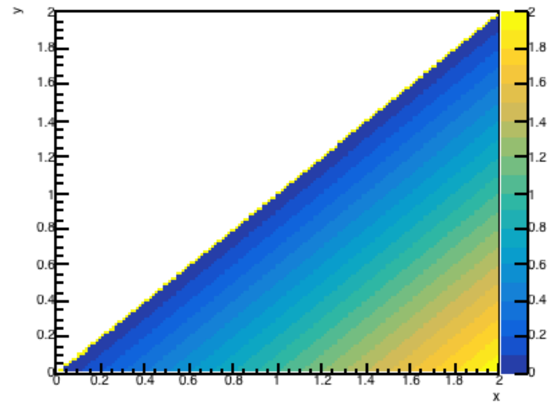
a=ReLU(-x+y)



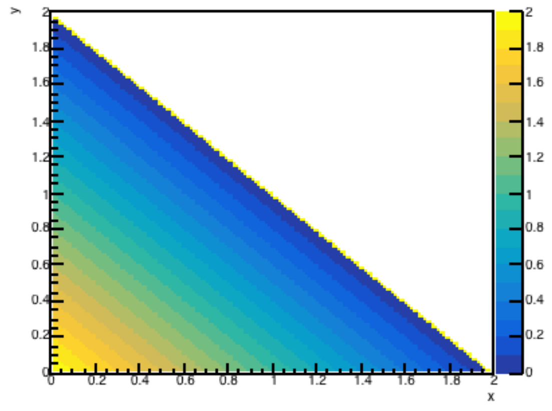


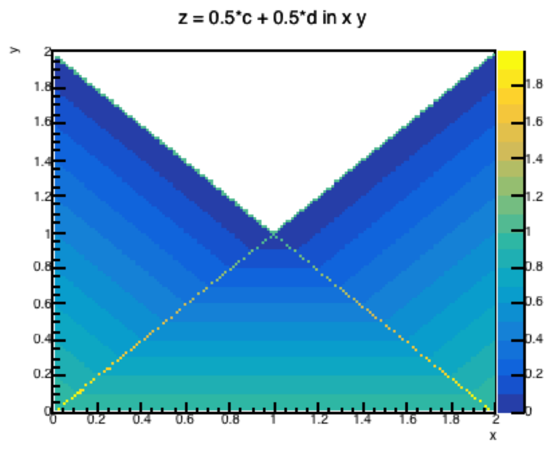


a=ReLU(x-y)

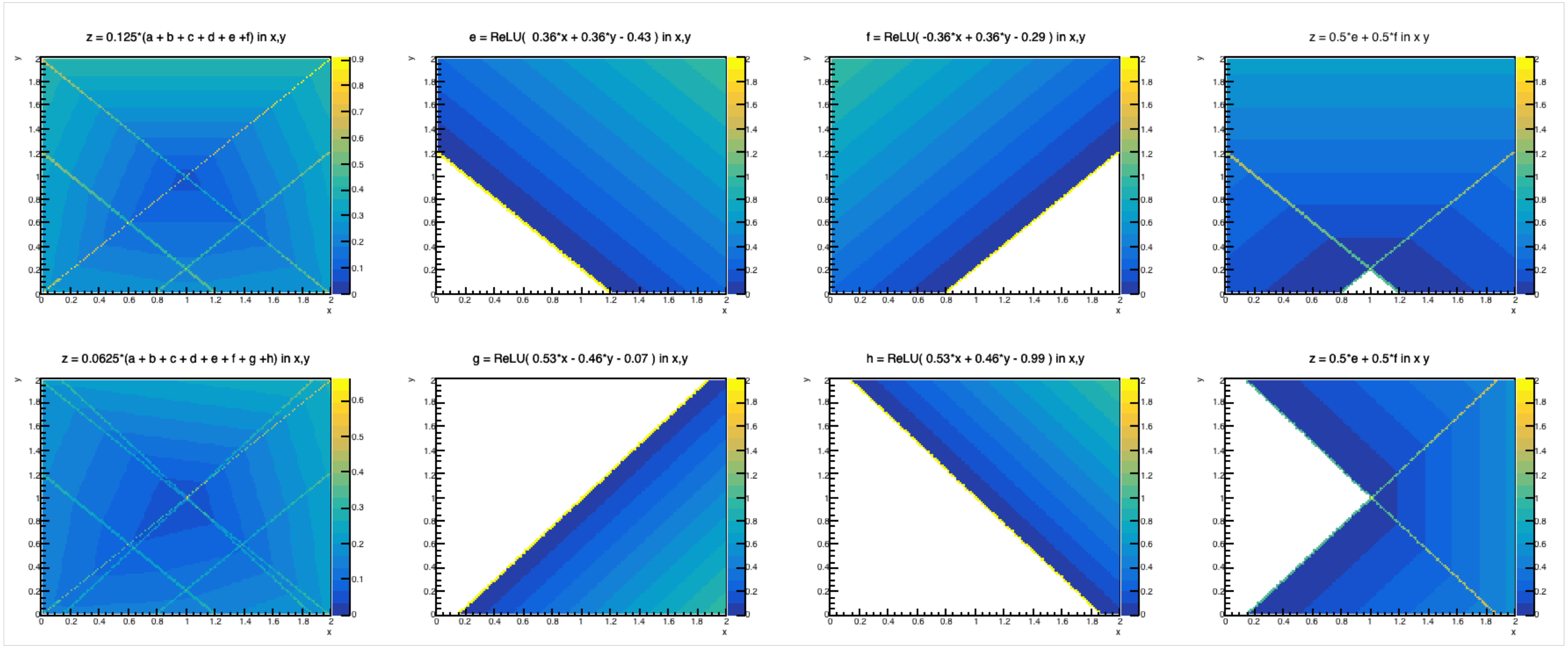


a=ReLU(-x-y+2)



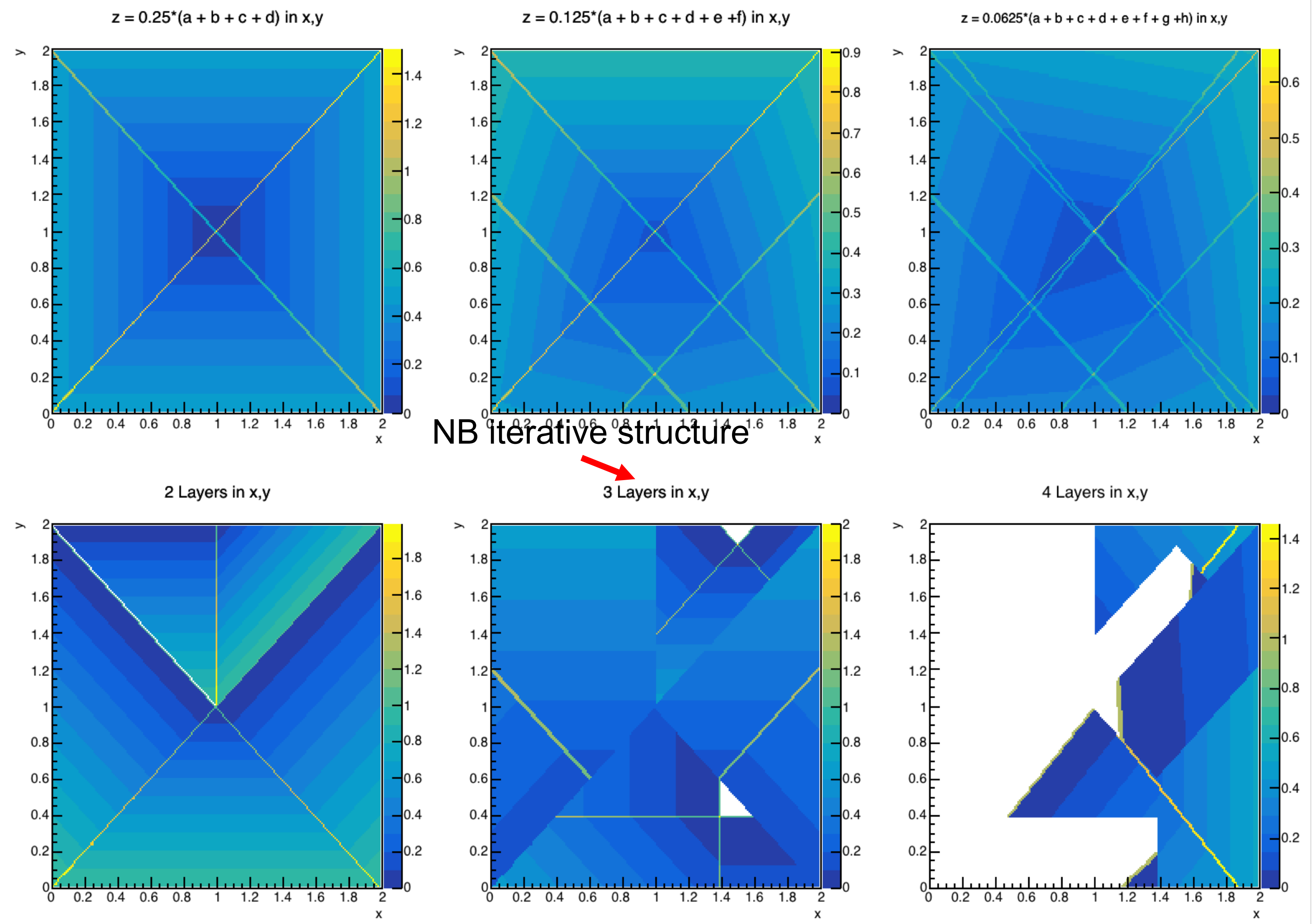


2 layers



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
x
y

3 layers



NB iterative structure

Cutting hyperplanes - DL

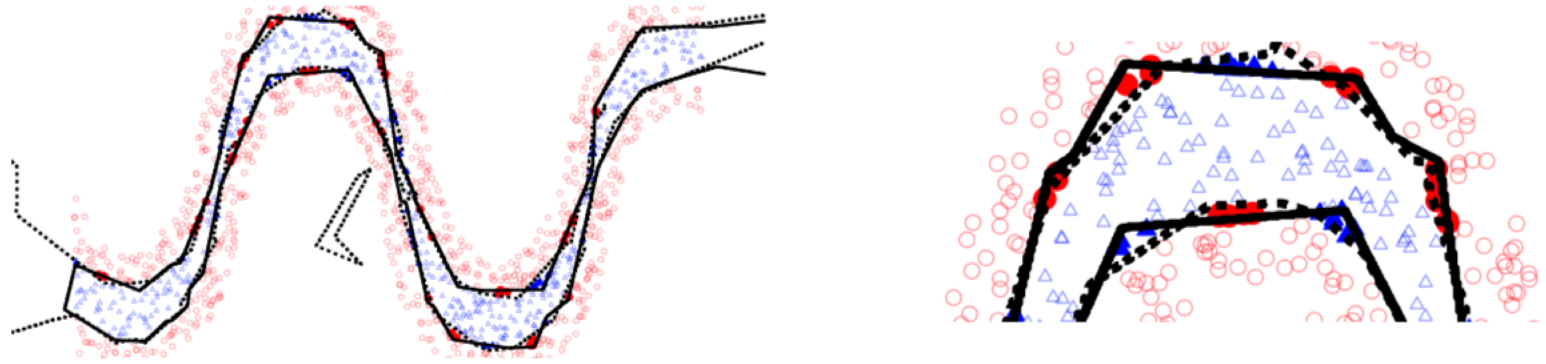


Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

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

Cutting hyperplanes - 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

Contact

DESY. Deutsches Elektronen-Synchrotron

www.desy.de

Dirk Krücker
CMS
E-mail dirk.kruecker@desy.de
Phone 3749