# Machine Learning and Multivariate Techniques in HEP data Analyses

## Artificial Neural Networks

**Extracted from slides by:**
**G. Cowan's lectures at RH London Univ., H. Voss at SOS 2016, K. Reygers lectures at Heilderbeg Univ.**

Prof. dr hab. Elżbieta Richter-Wąs

# Machine Learning - Basic terminology

**The goal of machine learning is to predict results based on incoming data.**

**Features** (also parameters, or variables): these are the factors for a machine to look at. E.g.: carthesian coordinates, pixel colors, a car mileage, user's gender, stock price, word frequency in the text.

- Quantitative (x = {1.02, 0.21, 0.12, 2})
- Qualitative *discrete* (x = {medium, small, large}) or *categorical* (x={red, blue, green})

**Algorithms** (also models): Any problem can be solved in different ways. The method you choose affects the precision, performance, and size of the final model.

- If the data is insufficient/inapproriate (e.g. statistically limited or missing important features), even the best algorithm won't help. Pay attention to the accuracy of your results only when you have a good enough dataset.

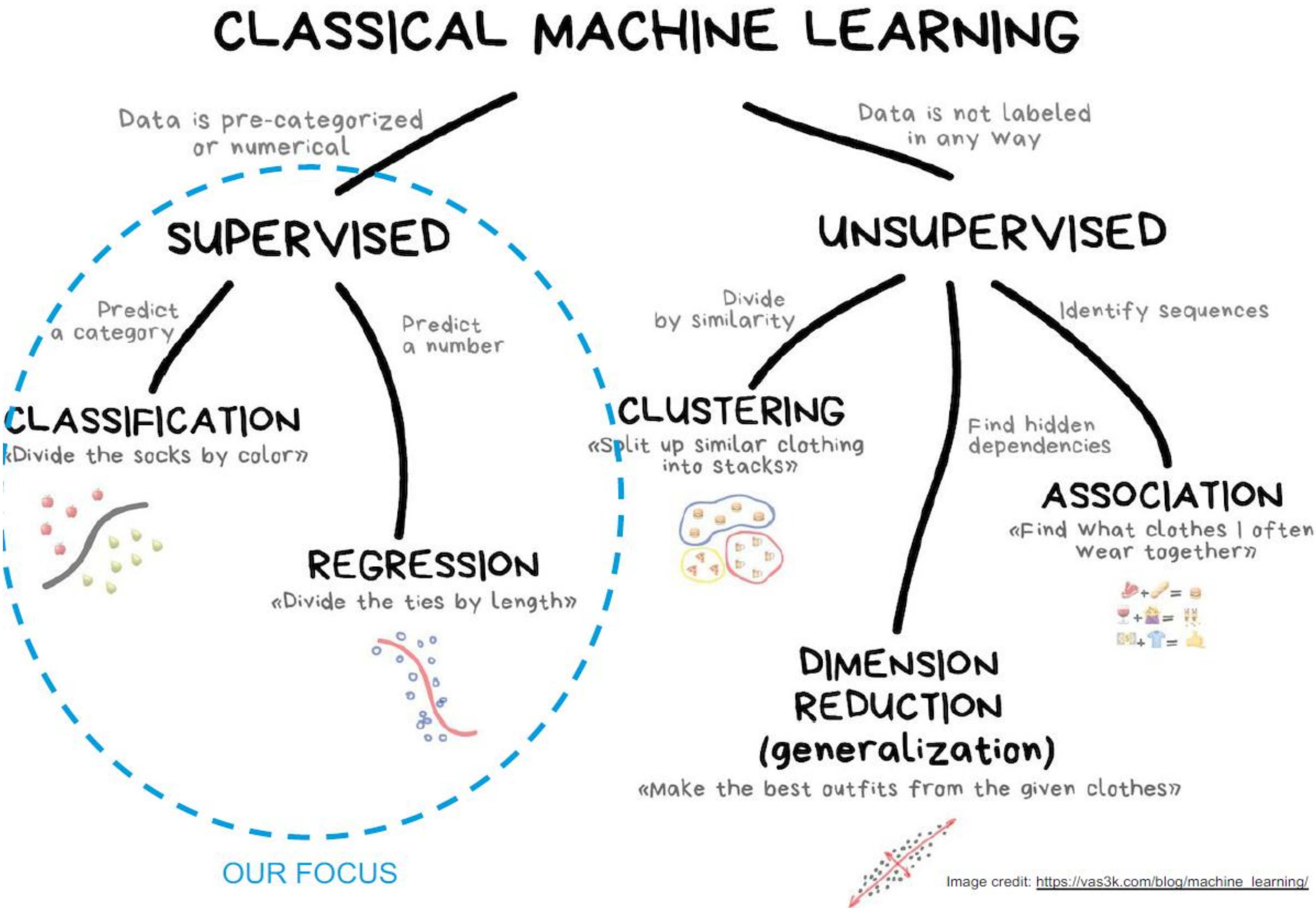



Image credit: https://vas3k.com/blog/machine_learning/

# Where are the Neural Networks?

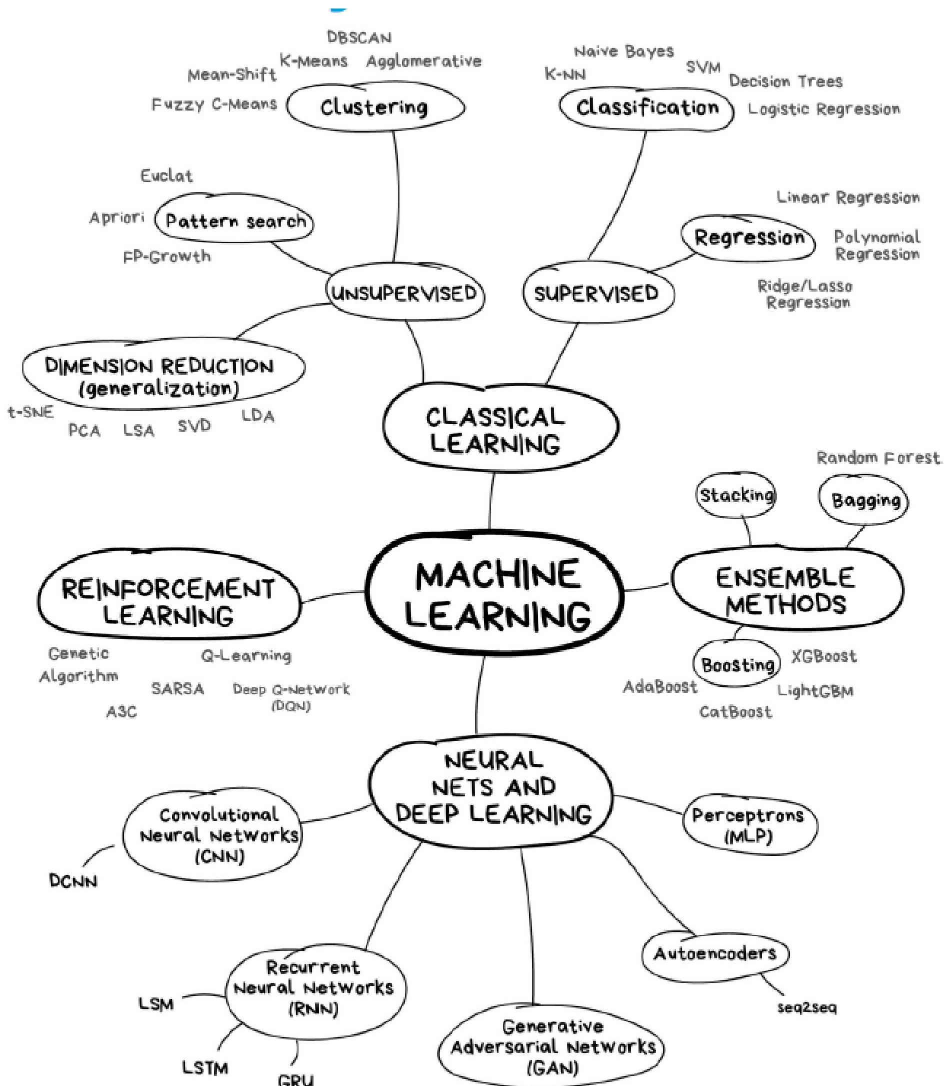



Image credit: https://vas3k.com/blog/machine_learning/

# Neural Networks

Any neural network is a collection of **neurons** and **connections** between them.

**Neuron** is a function with a set of inputs and one output. Its task is to take all numbers from its input, apply a function on them and send the result to the output.

- Example: sum up all numbers from the inputs and if that sum is bigger than N give 1 as a result. Otherwise return zero.

**Connections** are like channels between neurons. They connect outputs of one neuron with the inputs of another so they can send digits to each other. Each connection has only one parameter the *weight*.

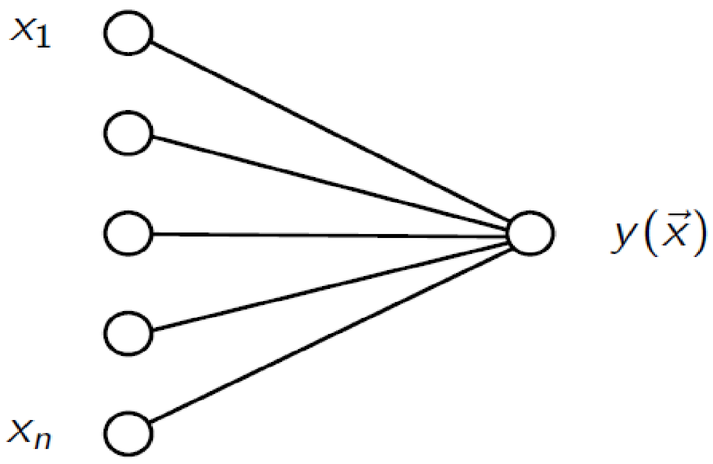
- These weights tell the neuron to respond more to one input and less to another. Weights are adjusted when training — that's how the network learns.

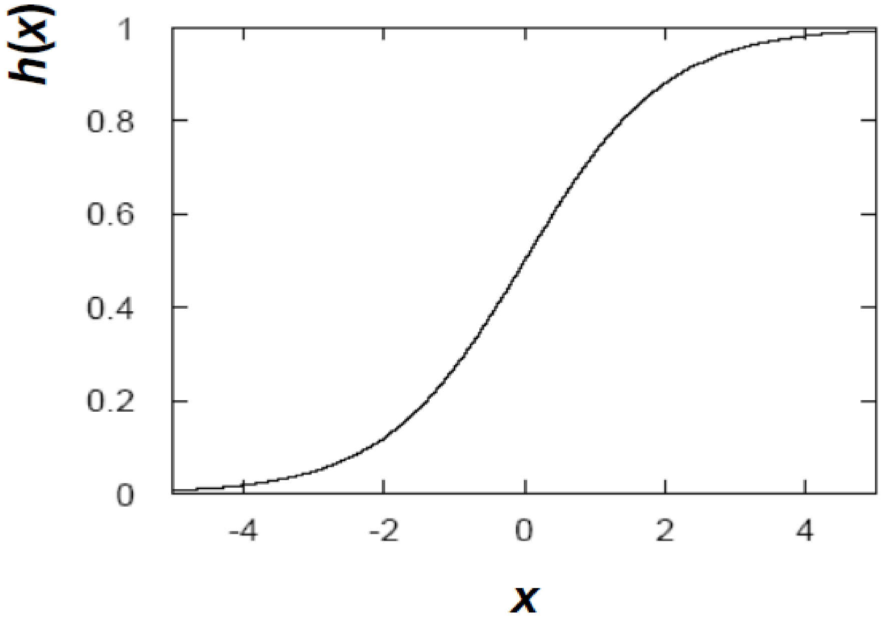
# Perceptron

Discriminant: $$y(\vec{x}) = h\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)$$

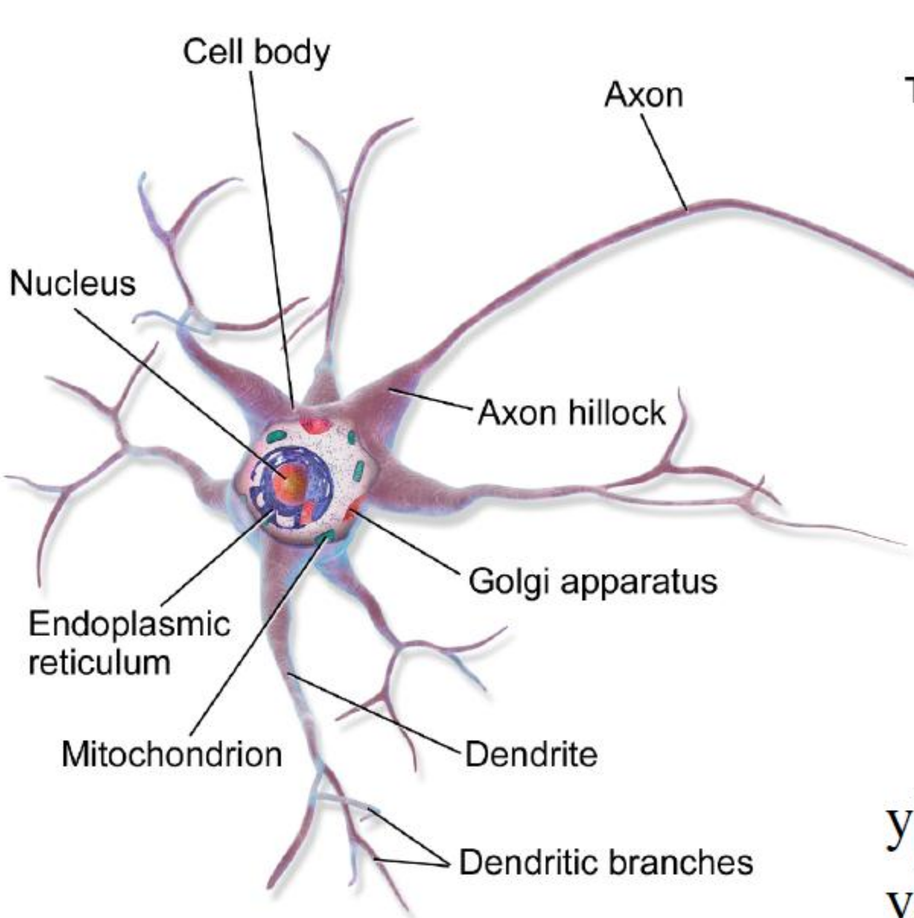
The nonlinear, monotonic function $h$ is called *activation function*.

Typical choices for $h$: $$\frac{1}{1+e^{-x}} \ (\text{"sigmoid"}), \quad \tanh x$$



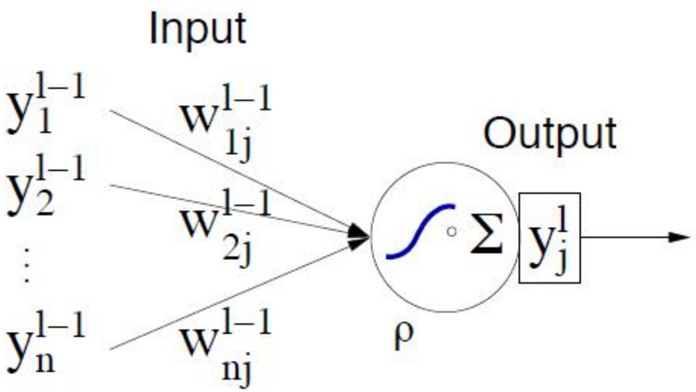



# The Biological Inspiration: the Neuron

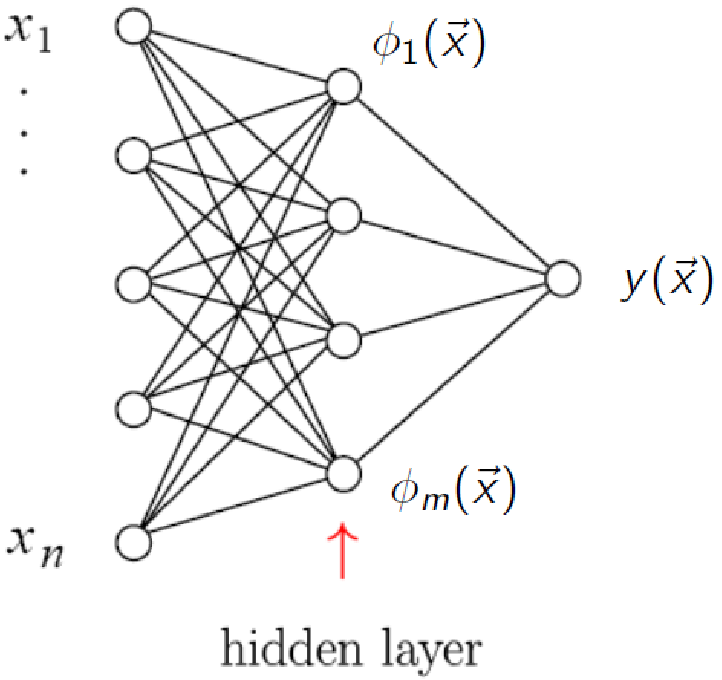



https://en.wikipedia.org/wiki/Neuron

Human brain:
$10^{11}$ neurons, each with on average 7000 synaptic connections



# Feedforward Neural Network with One Hiden Layer




superscripts indicates layer number

$$\phi_i(\vec{x}) = h\left(w_{i0}^{(1)} + \sum_{j=1}^{n} w_{ij}^{(1)} x_j\right)$$

$$y(\vec{x}) = h\left(w_{10}^{(2)} + \sum_{j=1}^{m} w_{1j}^{(2)} \phi_j(\vec{x})\right)$$

Straightforward to generalize to multiple hidden layers

# Network Training

$\vec{x}_a$ : training event, $a = 1, \ldots, N$

$t_a$ : correct label for training event $a$

e.g., $t_a = 1, 0$ for signal and background, respectively

$\vec{w}$ : vector containing all weights

Error function:

$$E(\vec{w}) = \frac{1}{2} \sum_{a=1}^{N} (y(\vec{x}_a, \vec{w}) - t_a)^2 = \sum_{a=1}^{N} E_a(\vec{w})$$

Weights are determined by minimizing the error function.

# Backpropagation

Start with an initial guess $\vec{w}^{(0)}$ for the weights an then update weights after each training event:

$$\vec{w}^{(\tau+1)} = \vec{w}^{(\tau)} - \eta \nabla E_a(\vec{w}^{(\tau)})$$

learning rate

Let's write network output as follows:

$$y(\vec{x}) = h(u(\vec{x})) \;\; \text{with} \;\; u(\vec{x}) = \sum_{j=0}^{m} w_{1j}^{(2)} \phi_j(\vec{x}), \;\; \phi_j(\vec{x}) = h\left(\sum_{k=0}^{n} w_{jk}^{(1)} x_k\right) \equiv h\left(v_j(\vec{x})\right)$$

Here we defined $\phi_0 = x_0 = 1$ and the sums start from 0 to include the offsets.

Weights from hidden layer to output:

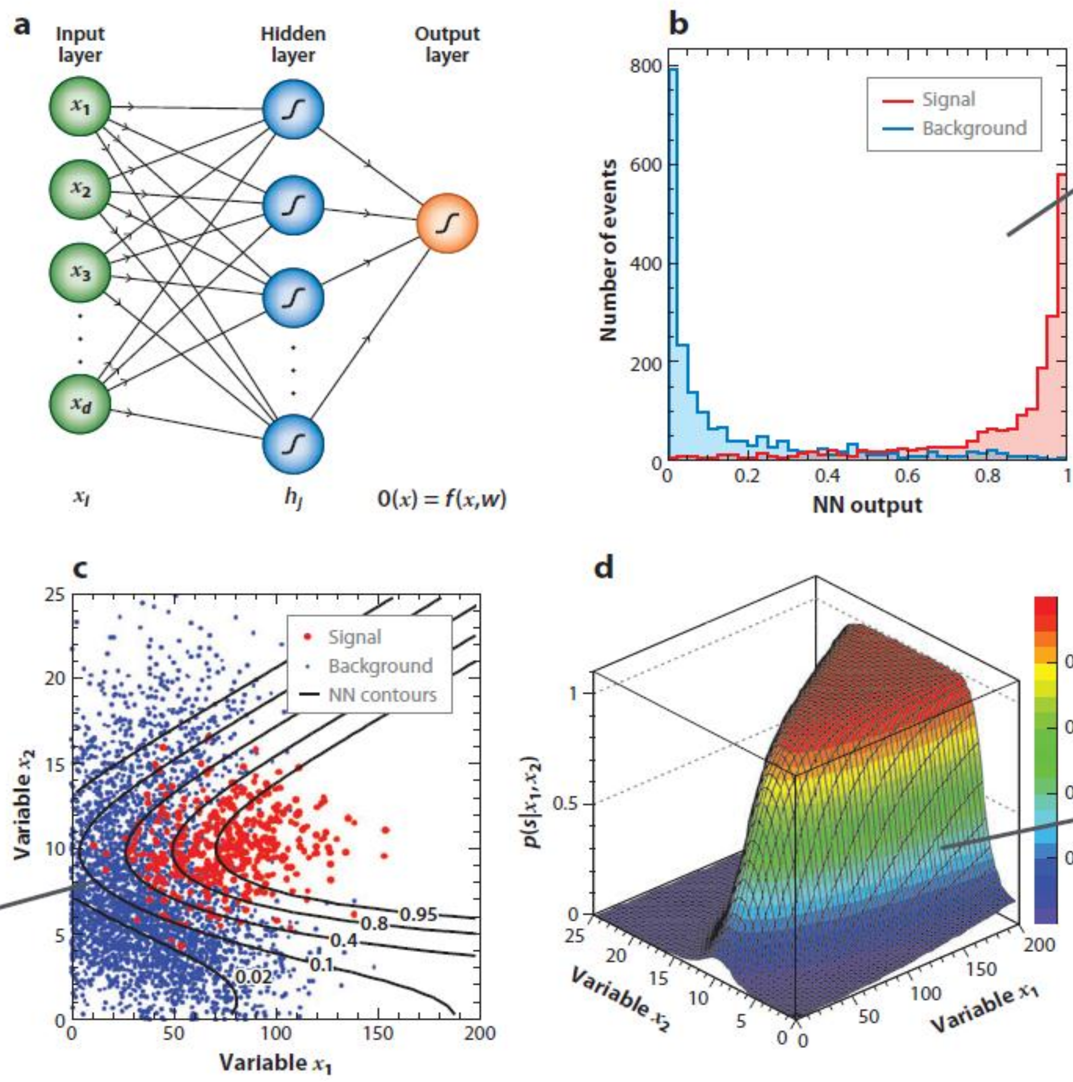
$$E_a = \frac{1}{2}(y_a - t_a)^2 \;\rightarrow\; \frac{\partial E_a}{\partial w_{1j}^{(2)}} = (y_a - t_a) h'(u(\vec{x}_a)) \frac{\partial u}{\partial w_{1j}^{(2)}} = (y_a - t_a) h'(u(\vec{x}_a)) \phi_j(\vec{x}_a)$$

Weights from input layer to hidden layer (→ further application of chain rule):

$$\frac{\partial E_a}{\partial w_{jk}^{(1)}} = (y_a - t_a) h'(u(\vec{x}_a)) w_{1j}^{(2)} h'\left(v_j(\vec{x}_a)\right) x_{a,k}$$

$$\vec{x}_a \equiv (x_{a,1}, \ldots, x_{a,n})$$

# Neural Network Output and Decision Boundaries



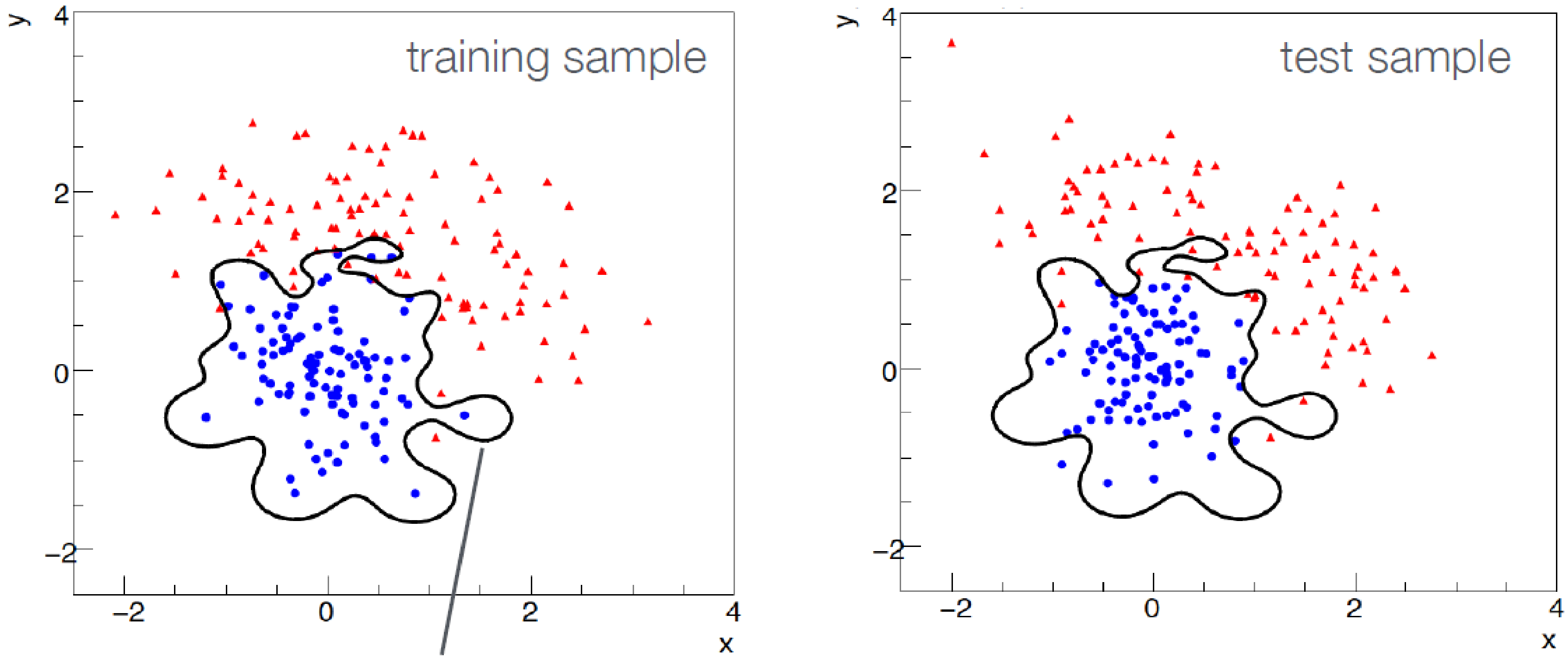

output of neural network

decision boundaries for different cuts on NN output

signal probability $p(s \mid x_1, x_2)$

# Example of Overtraining

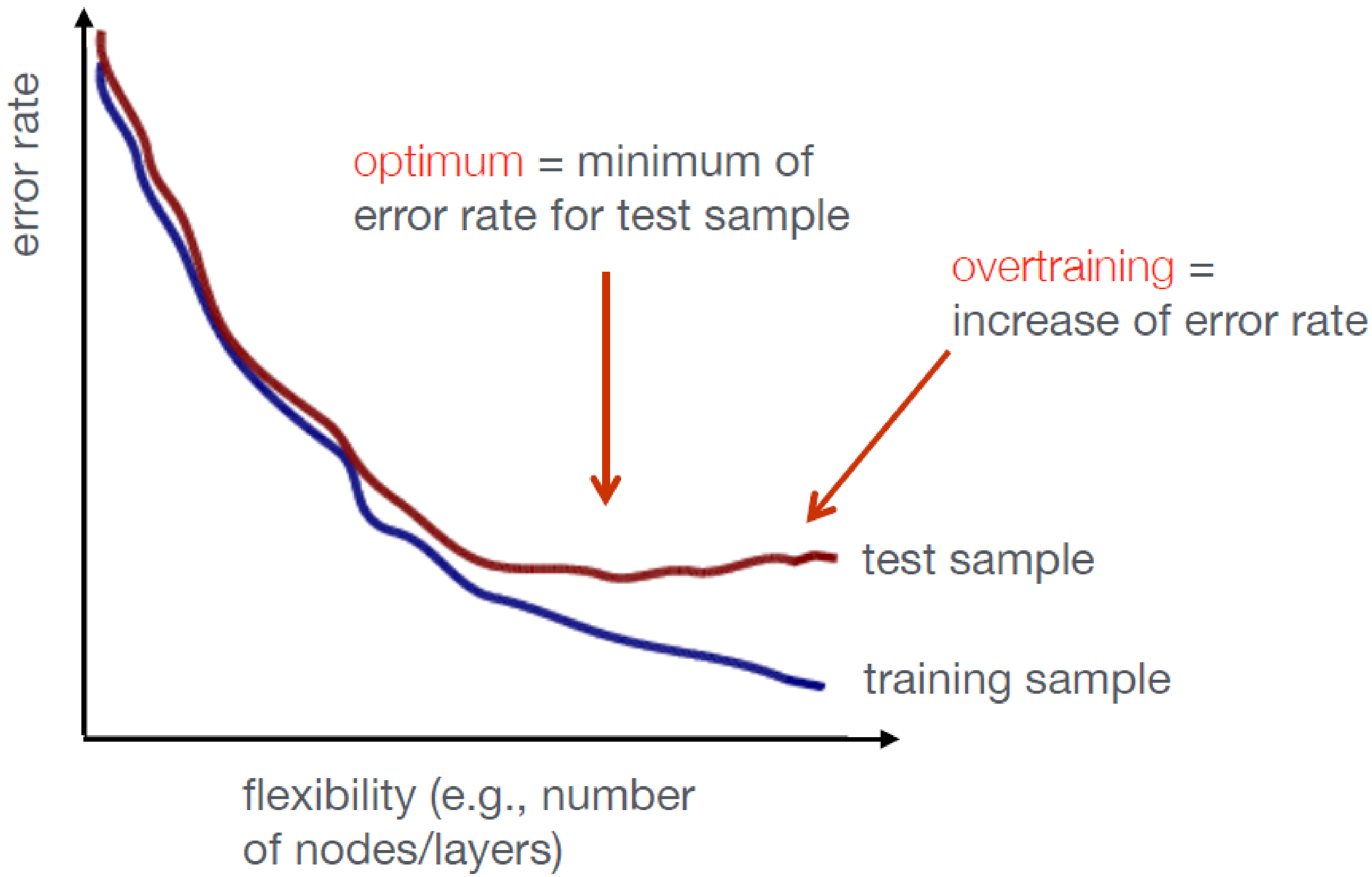
Too many neurons/layers make a neural network too flexible
→ overtraining




Network "learns" features that are merely statistical fluctuations in the training sample

# Monitoring Overtraining

Monitor fraction of misclassified events (or error function:)



# Deep Neural Networks

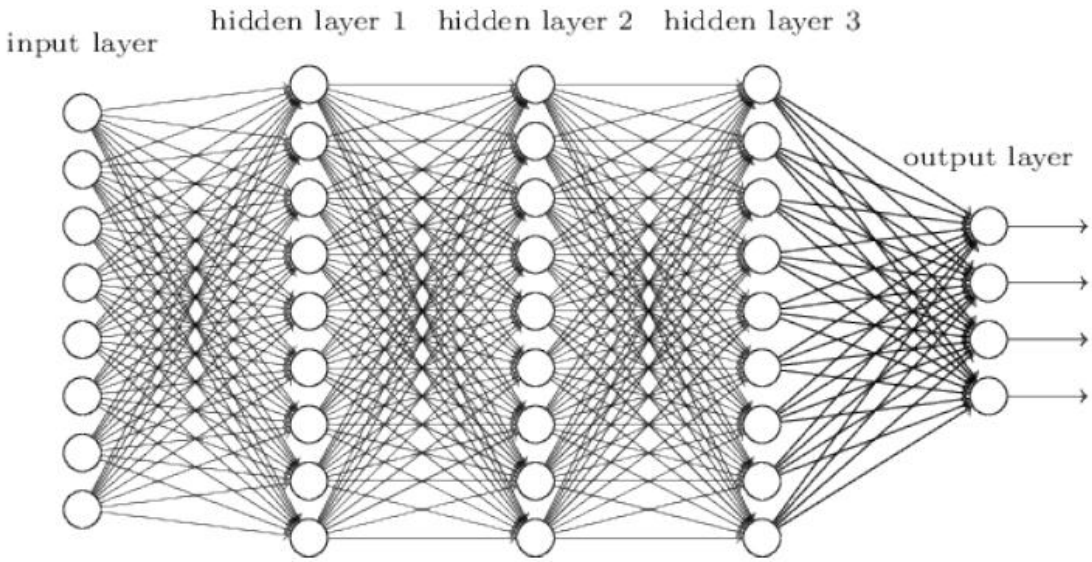
Deep networks: many hidden layers with large number of neurons

Challenges

- Hard too train ("vanishing gradient problem")
- Training slow
- Risk of overtraining

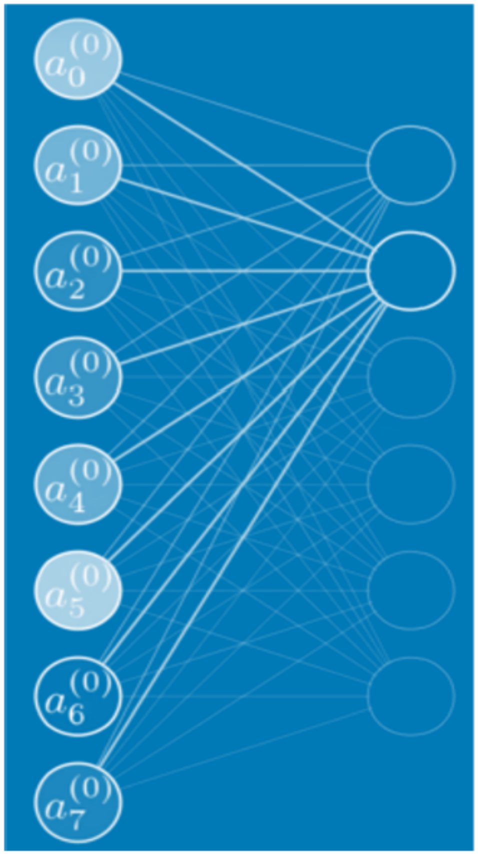
Big progress in recent years

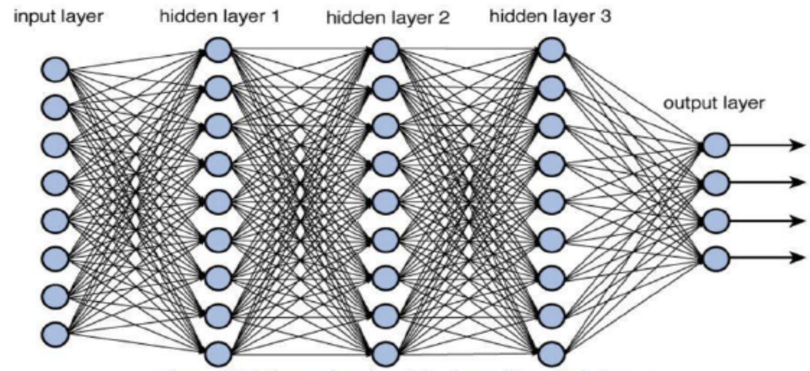
- Interest in NN waned before ca. 2006
- Milestone: paper by G. Hinton (2006): "learning for deep belief nets"
- Image recognition, AlphaGo, …
- Soon: self-driving cars, …



# How do NNs work?

## How do NNs work?




layer

$$a_0^{(1)} = f\left(w_{0,0}\, a_0^{(0)} + w_{0,1}\, a_1^{(0)} + \ldots + w_{0,n}\, a_n^{(0)} + b_0\right)$$

activation function

weights

bias

$$\begin{bmatrix} a_0^{(1)} \\ \ldots \\ a_n^{(1)} \end{bmatrix} = f\left( \begin{bmatrix} w_{0,0} & \cdots & w_{0,n} \\ \vdots & \ddots & \vdots \\ w_{k,0} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ \ldots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ \ldots \\ b_n \end{bmatrix} \right)$$

# How do NNs learn?

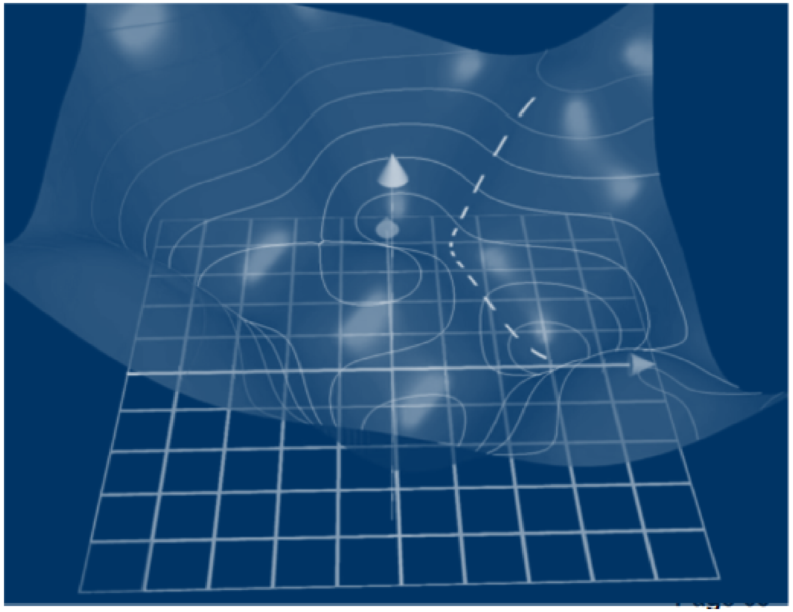
After we constructed a network, our task is to **assign proper weights** so neurons will react correctly to incoming signals.

- define a loss function to measure how far the response is from the truth

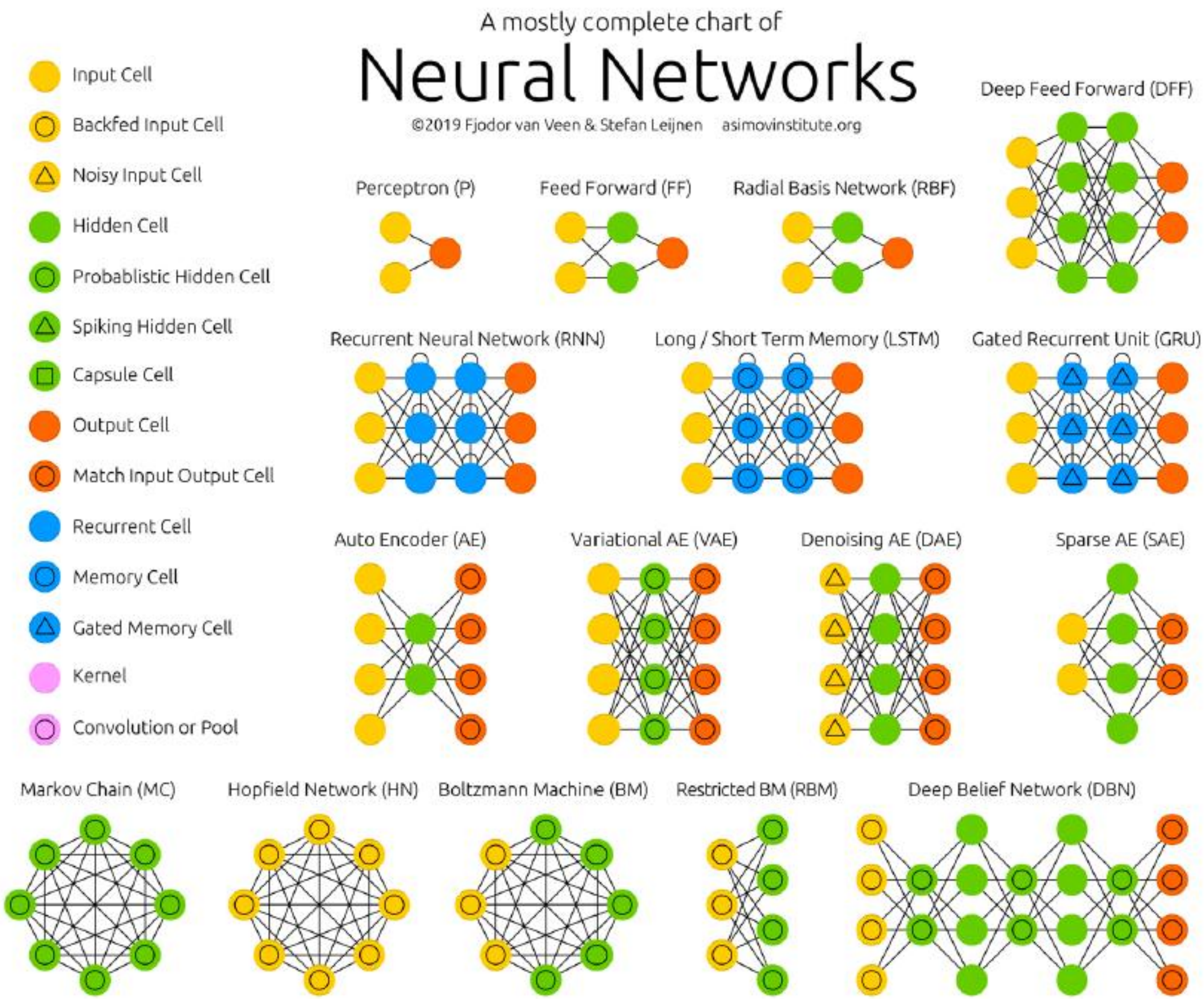
This function is a function of all the weights and biases in the NN (a priori a very large number), and the goal of training is to find its minimum.

- To start with, all weights are assigned randomly.
- After evaluating the NN on the training dataset, we can compute all the per-neuron differences with respect to the correct result.
- Computing the gradient of the loss, gives us a direction in which to tune the weights towards a local minimum

The process of correcting the weights is called backpropagation an error.



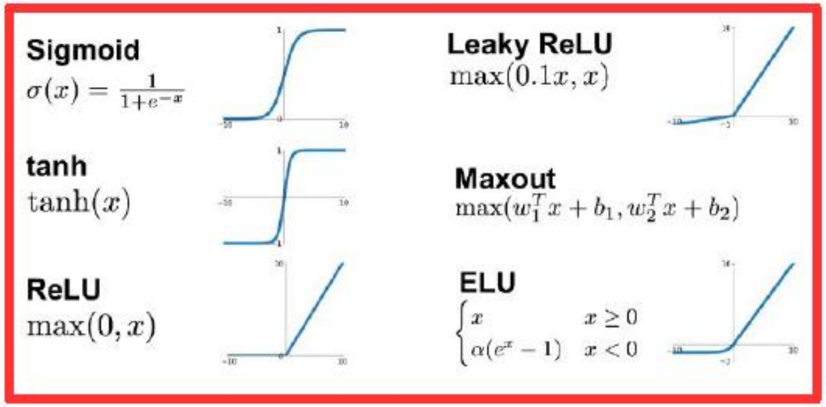
# How do NNs learn?



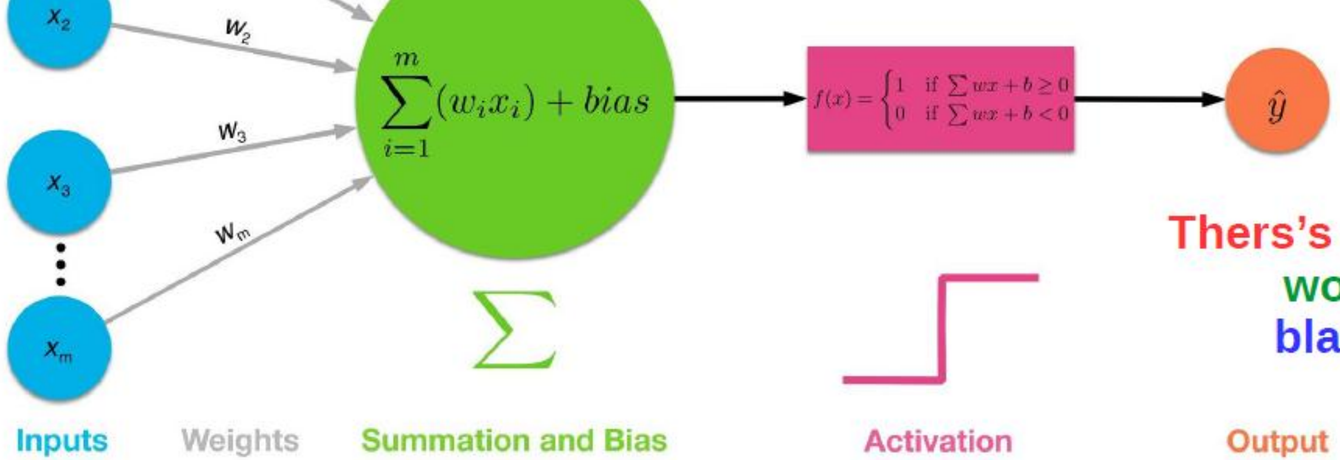

There are many more…

# How do NNs learn?

- Each input is multiplied by a weight.
- The weighted values are summed and a bias is added
- **The result is passed to an activation funciton (non liniarity).**



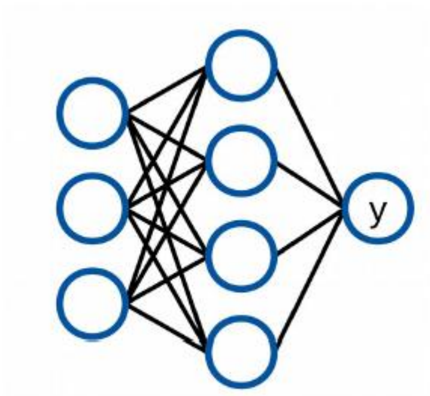

$$\hat{y} = f\left(\sum_{i=1}^{m}(w_i x_i) + bias\right)$$

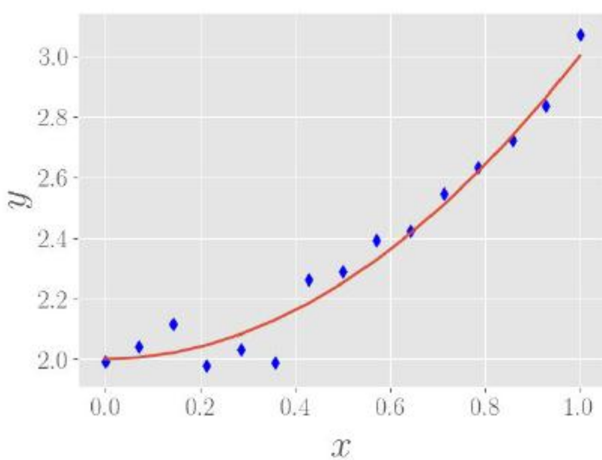



Thers's a colorful world inside black boxes!

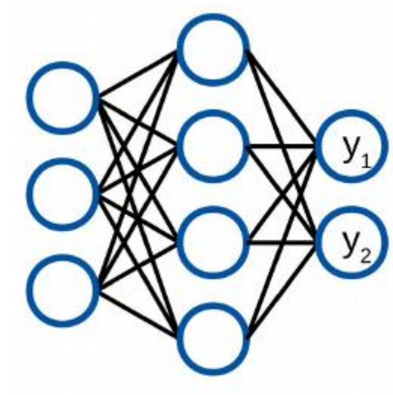
# Typical Applications

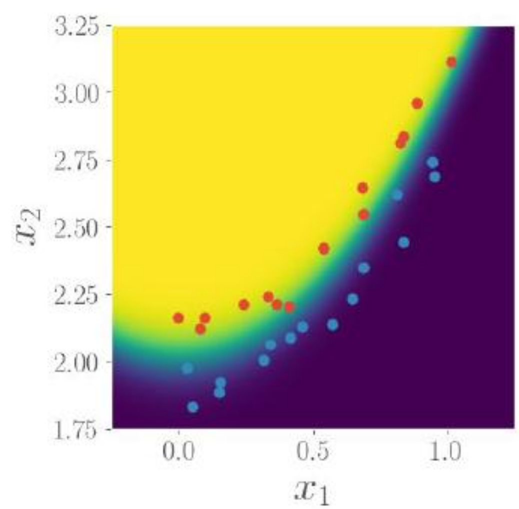
**Regression:**
Predict a continuous label.



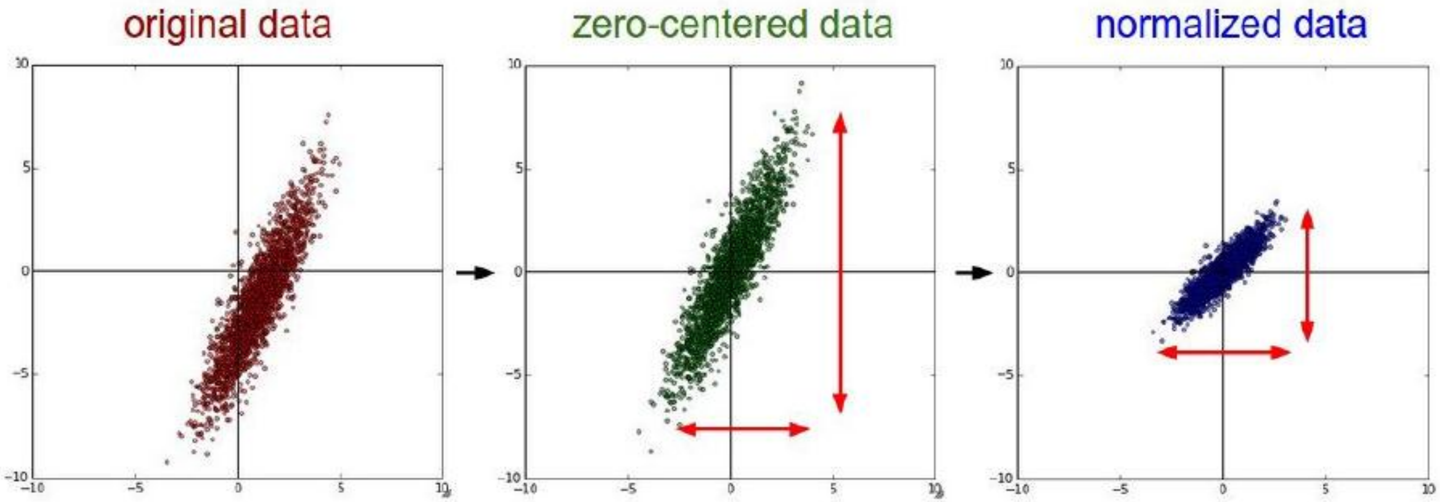

**Classification:**
Separate events into multiple categories.



# Input Preprocesing

- Input features could have vastly different scales (e.g. $p_T$ vs $\eta$).
  - Difficult to find optimal values.
- Basic strategy: Normalize to mean=0 and variance=1.

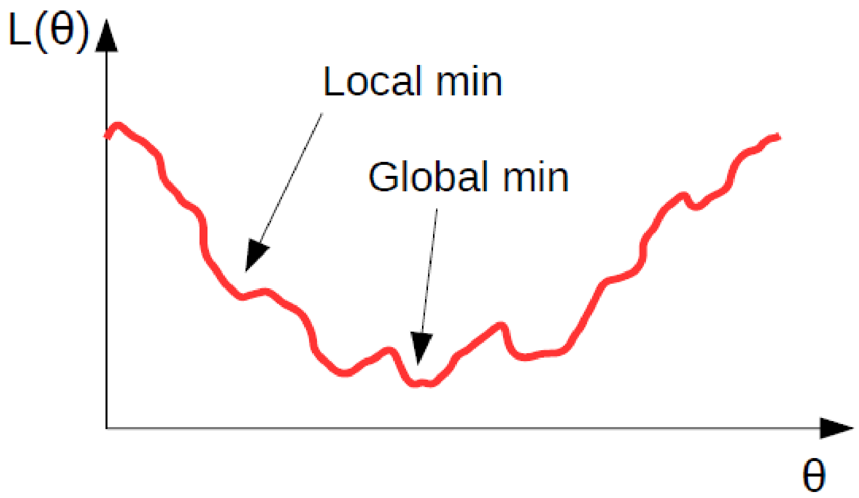



- Other options possible: decorrelation, non-linear transformation, ...

# Training

- Training starts specifying an input and a target dataset.
  - For each input set, the target is what the network should learn for that input.
- A loss function is required L(θ):
  - The loss funtion quantifies the mistakes the NN makes. E.g. *mean squared error* can be used for regression.

Training is the minimization of the loss function w.r.t. the NN parameters.



# Training: (Stochastic) Gradient Descent

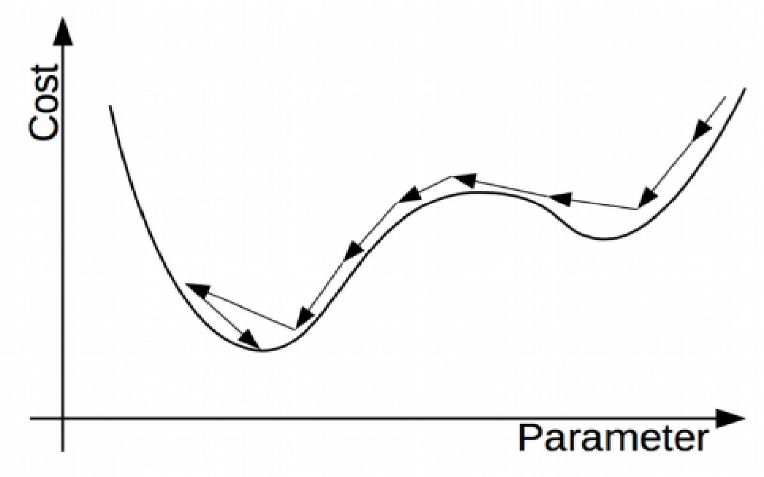
- Given the increasing size of datasets and parameters, it is no more possible to directly minimize the loss function.

Iterative minimization by updating $\theta$ in opposite direction of gradient.

$$\theta_i = \theta_{i-1} - \alpha \frac{\partial L}{\partial \theta}$$ , where α is the so-called « learning rate ».

- Evaluation and derivation of the loss funciton for the full dataset is costly:

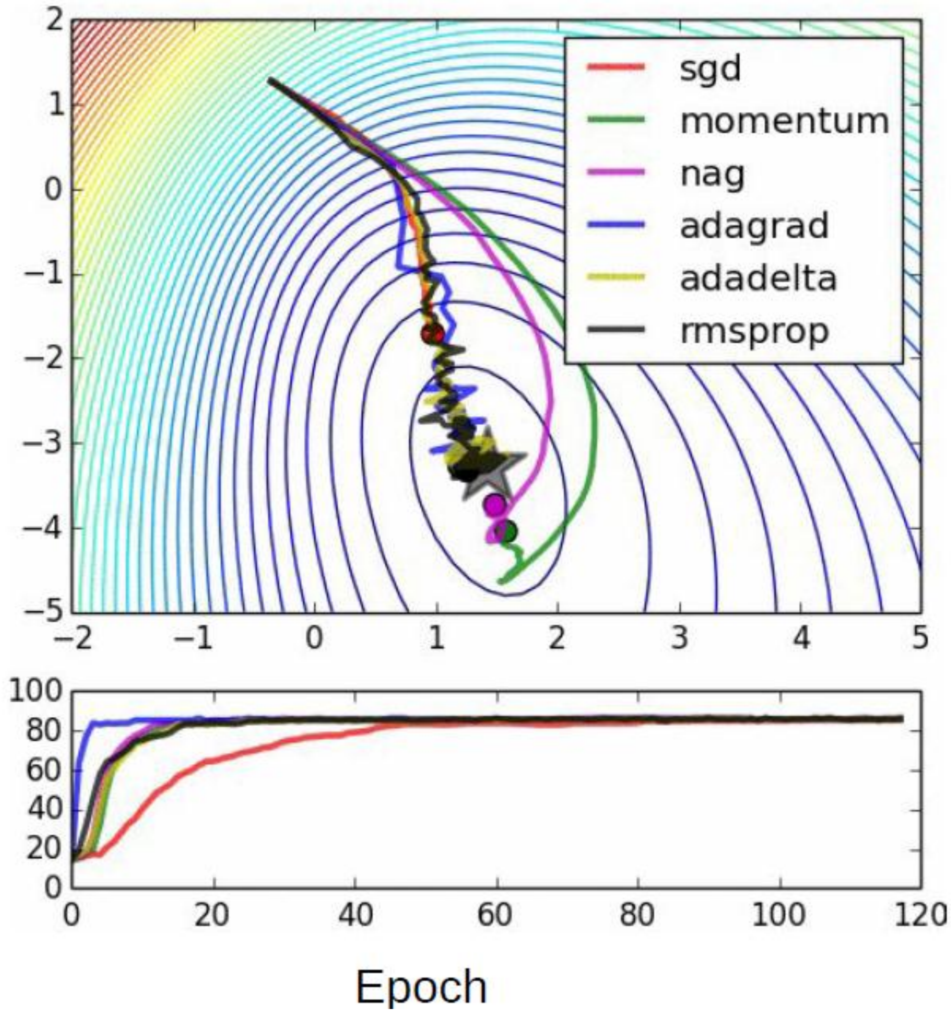
**Stochastic gradient descent:**
Calculate gradient for a small stochastic subset of the training sample (batch). → This also helps to avoid local minima!




One iteration over the full training dataset is called *epoch.*

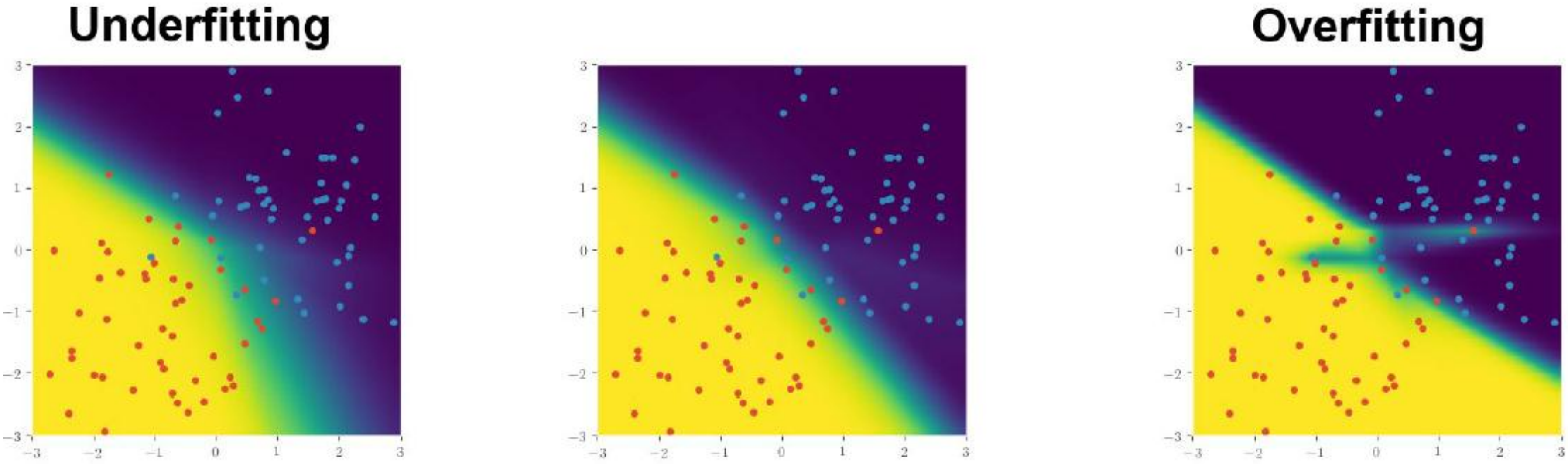
# Training: more optimisers

- More advanced options than fixed learning rate.
  - Momentum: past gradients used as « velocity »
  - Adaptive methods: different learning rates for each parameter and as a function of past gradients.



# Underfitting and overtraining

**Underfitting:** If model capacity is too low or if training is not enough → bad performance.

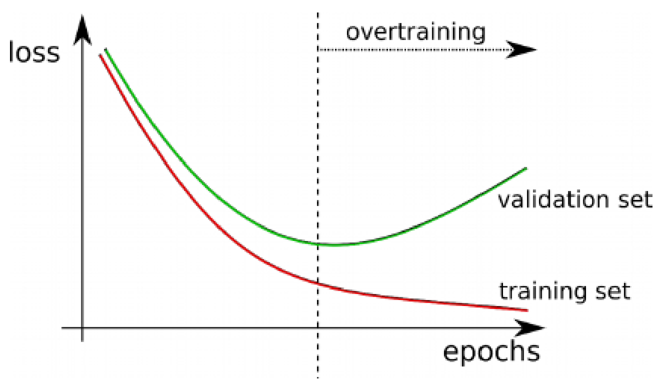


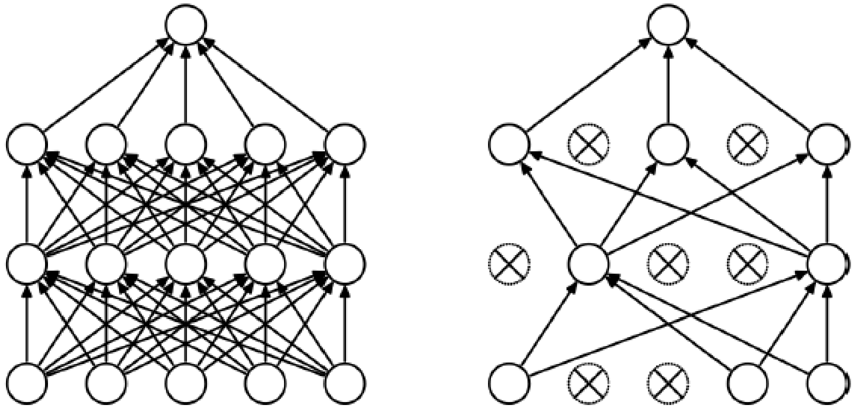

**Overfitting:** If model capacity is too high, network can « memorize » training samples → bad generalization.

# Overtraining solutions

**Early stop:**
Evaluate the performance of the network on a validation dataset.
Stop when performance on validation set decreases

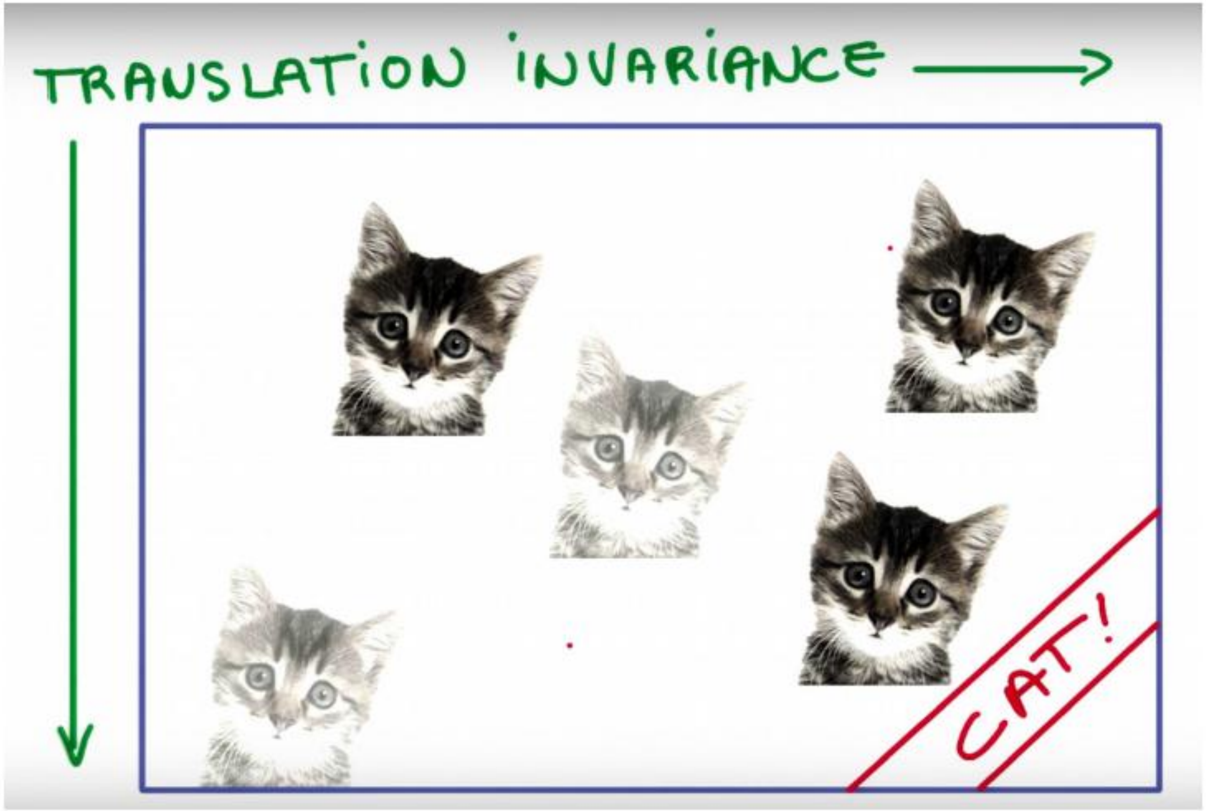
**Dropout:**
Randomly drop a percentage of nodes at each training step. Learn redundant representations, hence giving a more robust model.



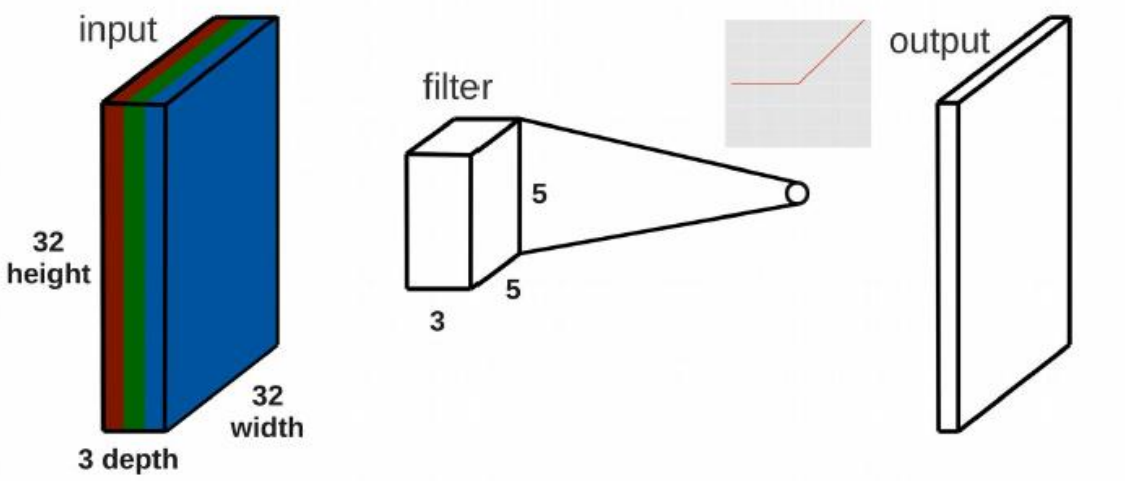
# Convolutional NN

- Convolutional NNs are made to exploit local correlation and translation invariance.
- Typical application are image processing and computer vision.



# Convolution layer

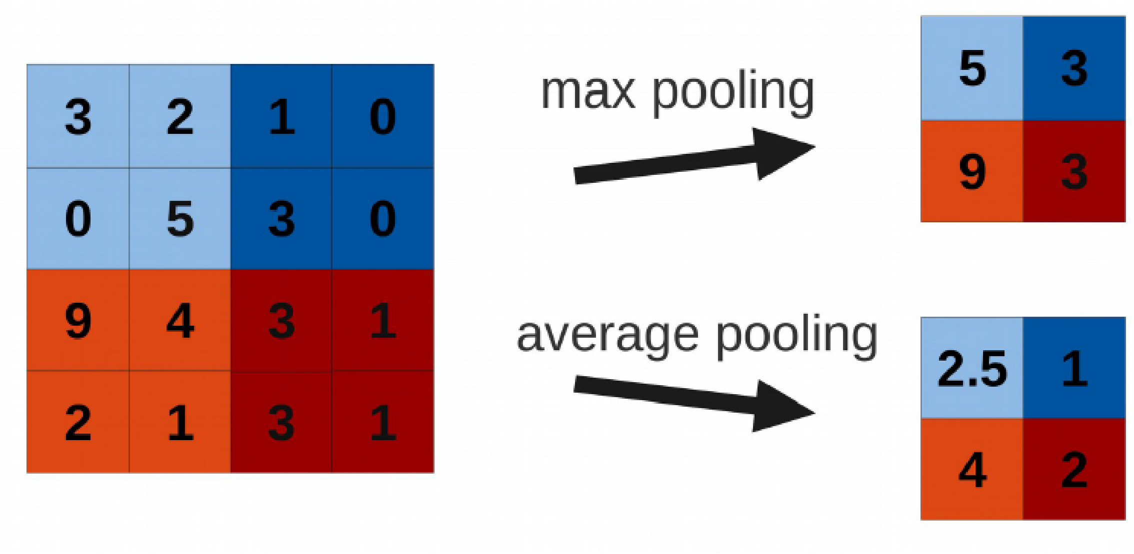
- A small filter (weight tensor) slides over the image to create a feature map.



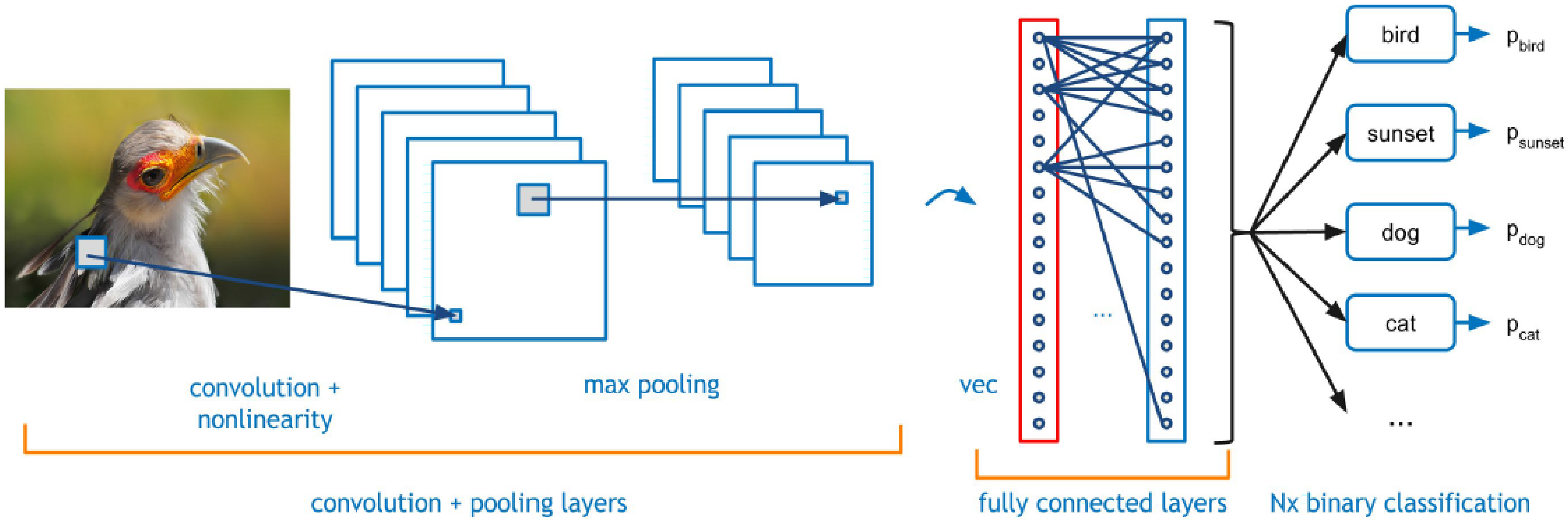

- Several filters could be stacked depth-wise.
- Several convolution can be applied one after the other to extract higher level features.

# Average and Max pooling layers

- When output size reduction is required:
  - Max pooling: takes the maximum of each patch.
  - Average pooling: takes the average of each patch.
- Eg: 2x2 filter with stride=2



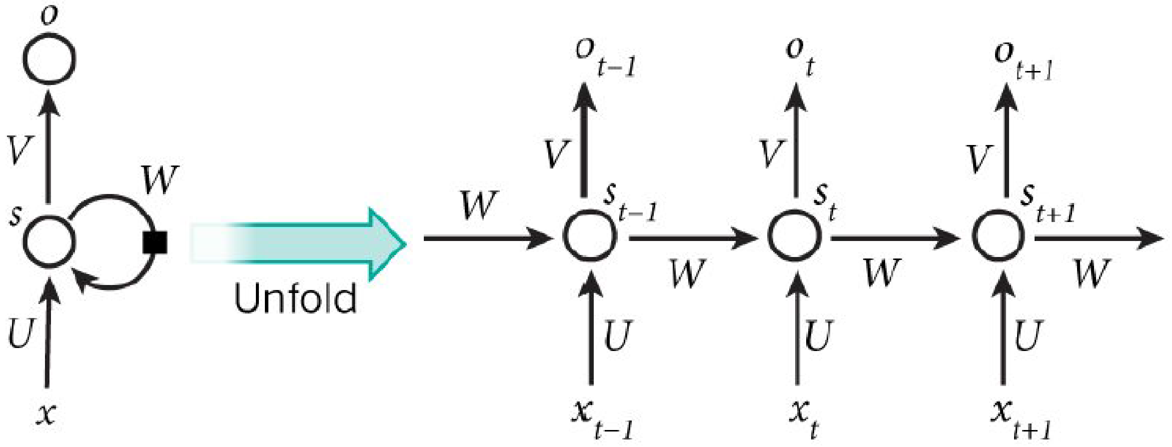
# Convolutional NN architecture




- Convolution and pooling layers to extract features.
- Fully connected layers used at the end to combine features.
- Applications in HEP: PID for neutrino experiments, jet tagging, reduction of seeds for tracking, etc..

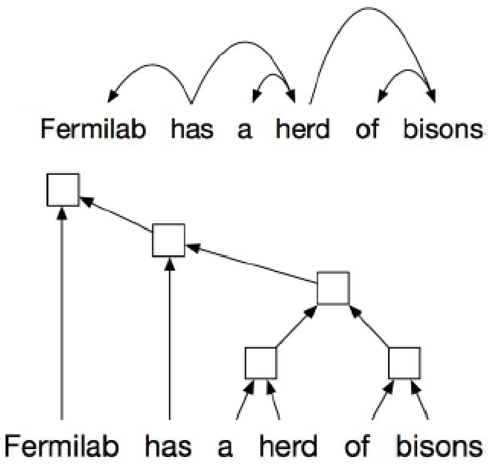
# Recursive NN

- Recursive NNs are deep NN created by applying the same set of weights recursively over a structured input of variable size.
- They are called recurrent because they performe the same task for every element of a sequence, with the output being depended on the previous computations.
- Typical applications in natural language processing: apply the recursive NN to each word in a sentence for text generation (predict the next word in the sequence), translation, etc..

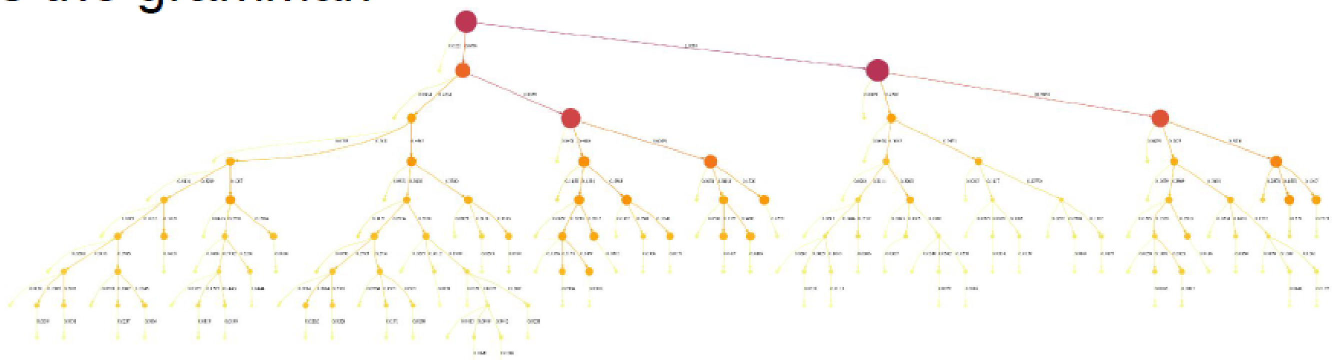


# Recursive NN: possible HEP applications

- With particle-flow, collision raw data is converted in a list of particles.
- Complex objects (e.g. jets) are reconstructed by combining particles from this list.
- Image-processing approaches might not be the best in this case.
- Recursive NNs can be better suited.

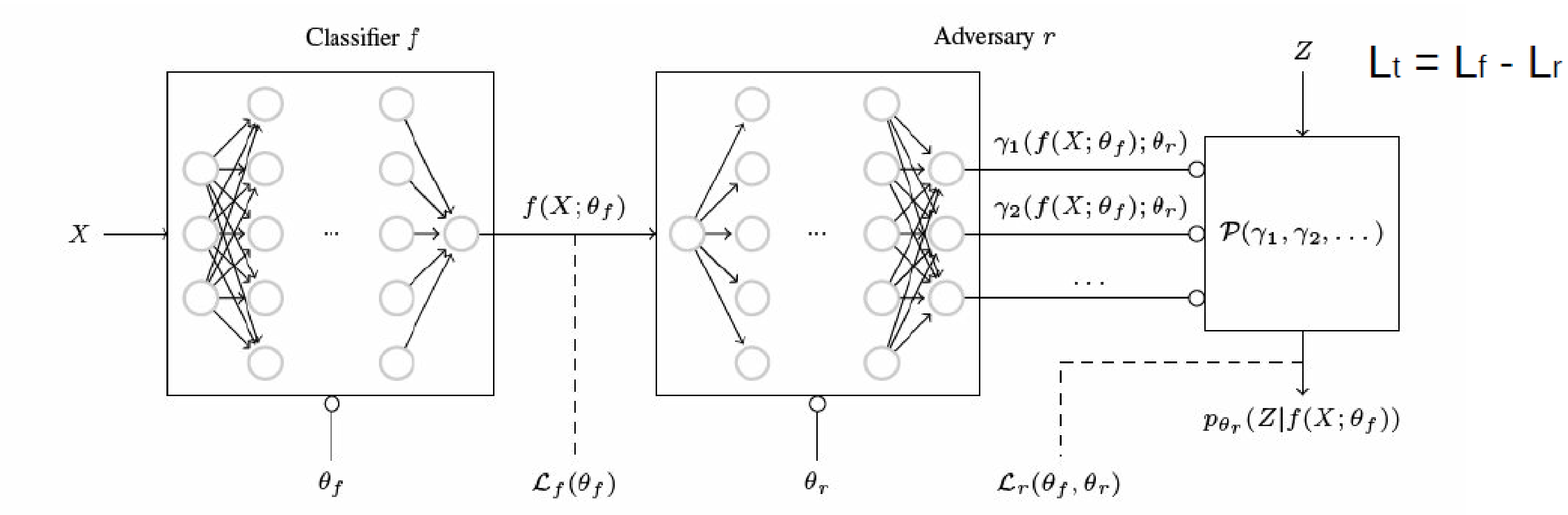



  - Particles are like words in a sentence.
  - QCD is the grammar.



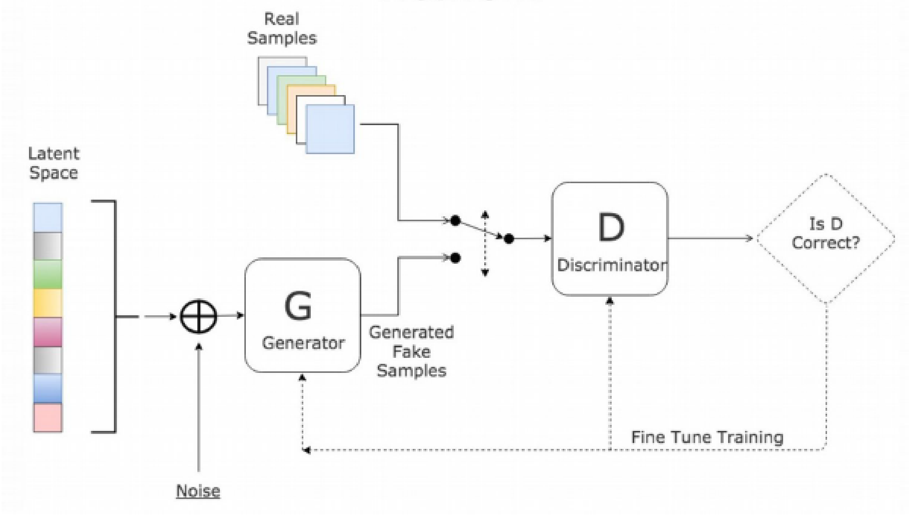
# Adversarial NN

- Two deep NNs in competition with each other.
- The first NN can be used to maximize the classification performance of signal against background events.
- The second NN can be trained to identify dependency on systematic uncertanty of the output of the first NN.
- The minimization of the global loss function guarantees optimal classification performance with reduced systematic dependence.



# Generative Adversarial NN

- Adversarial NNs can also be used for image generation.
- A generator and a discriminator are trained in competition.
  - Generator: creates images starting from noise.
  - Discriminator: tries to distinguish between true and generated images.




- The global loss function is given by Loss(gen) – Loss(discr).
- **The generator learns to make images that it has never seen simply by fooling the discriminator!**
- Application: gen of calo images, jets, and even high-level variables!

# Lorentz boost network: motivation

- Deep learning methods using high-level and low-level variables are outperforming shallow learning methods using high-level variables.
  - There is some information in low-level variables that high-level variables is not using.

- Deep learning methods using low-level variables only are not able to reproduce the same results as deep learning methods using also high-level variables.
  - Need of a new NN architecture to fully exploit low-level information and automatize the design of high-level variables.

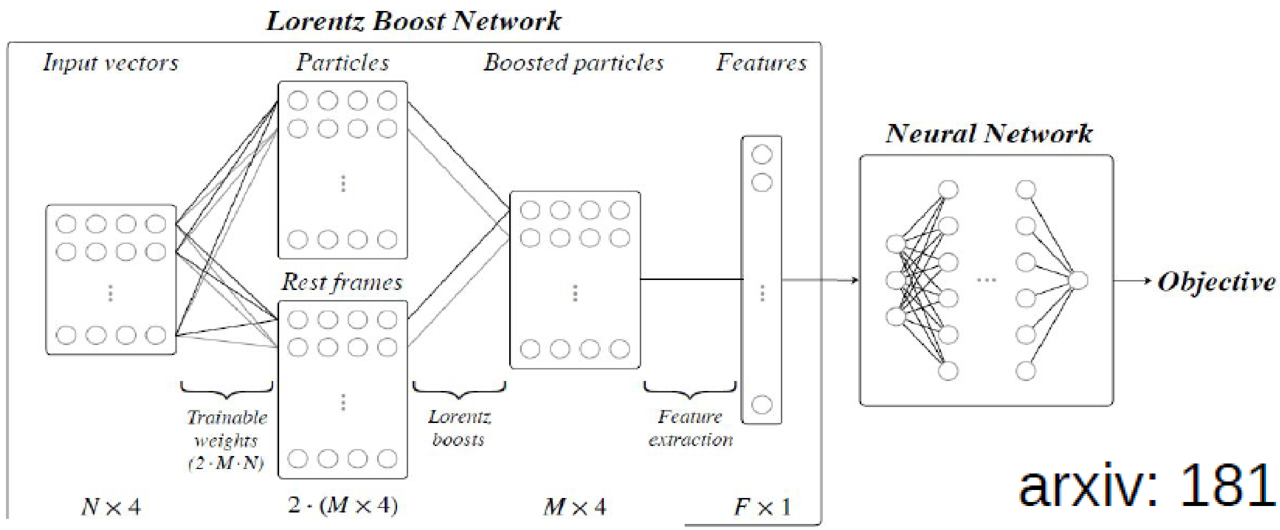
arxiv: 1812.09722

# Lorentz boost network: network architecture

- Two stages approach:

  combines them to form composite particles and rest frames. Composite particles are boosted in the rest frames where features are extracted.

- An application specific NN uses LBN features as input.

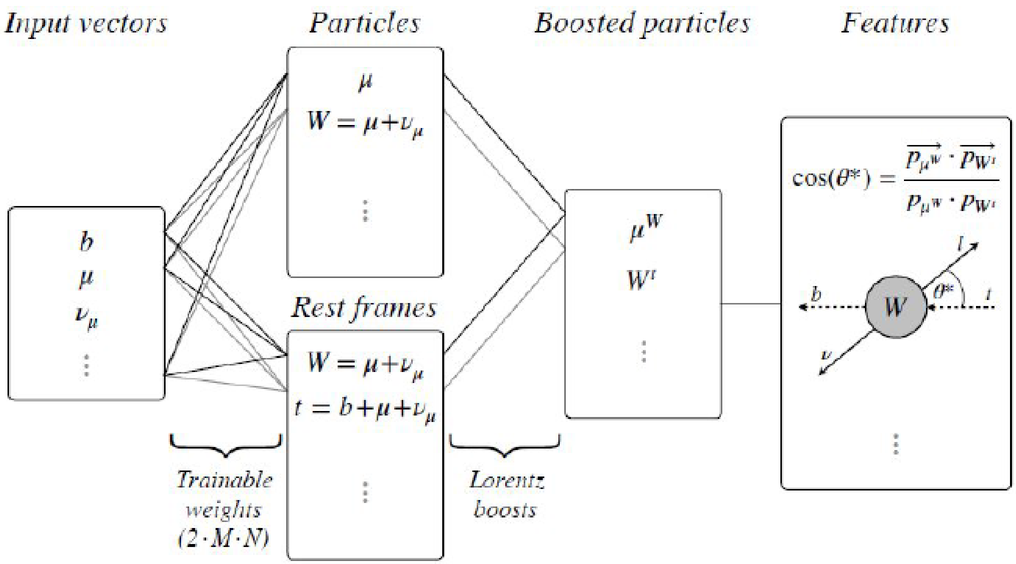



arxiv: 1812.09722

# Lorentz boost network: feauture extraction

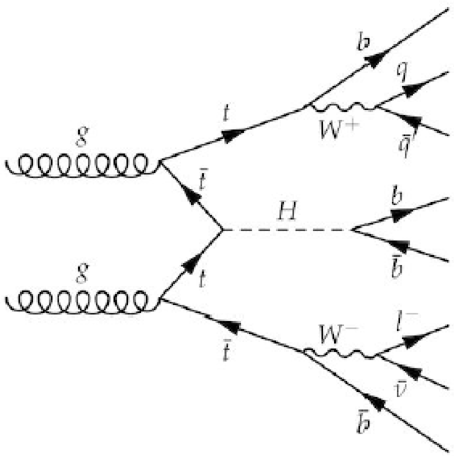
- Extract generic features from boosted particles.
  - Single particle features: $E$, $m$, $pT$, $\eta$, $\phi$.
  - Pairwise features: such as *cos(θ)* between all pairs.

- E.g. *cos(θ*)* in the semi-leptonic decay of the top quark, defined as the angular difference between the direction of the charged lepton in the *W* rest frame and the direction of the *W* in the top rest frame.

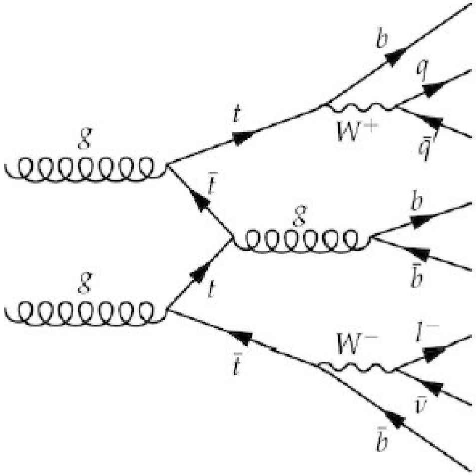


# BN for tth(bb) vs tt+bb: performance

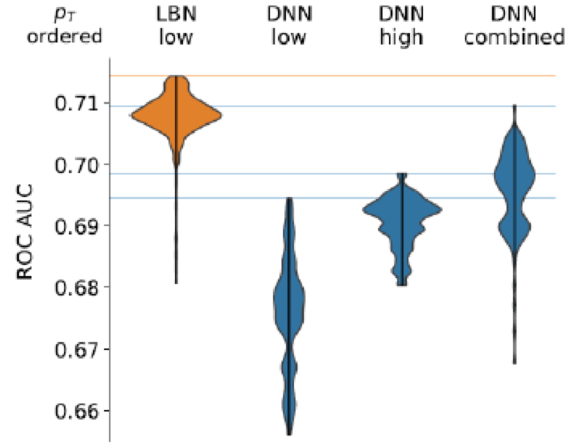
- LBN performance compared to standard DNN with low-, high- and comination of low- and high-level variables.




VS




- LBN shows improved performance in terms of ROC AUC.



# Conclusions

- Neural networks are widely used in HEP and will become more and more important.
- A quick overview of the basic structure of the most used NNs in HEP was given.
- New NNs layers, specifically engineered for HEP, were created.
    - In this case, high performance comes also with a good interpretability of the trained parameters.
- NN is a quickly developing field. Exciting time to work on it and to find new applications for HEP.