

# Neural Networks and Deep Learning

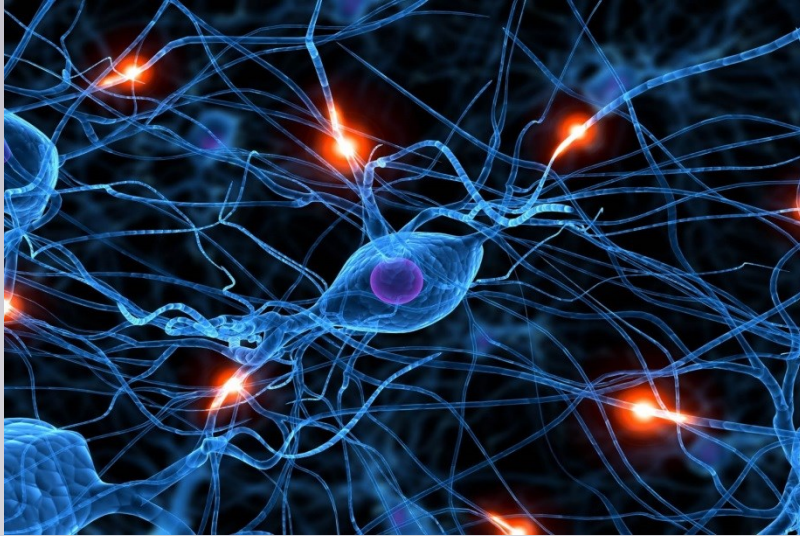
IN2P3 School of Statistics 2016



## Helge Voss

MAX-PLANCK-INSTITUT FÜR KERNPHYSIK IN HEIDELBERG

# Neural Networks



- **Powerful and very flexible machine learning algorithms**
  - **originally inspired by modelling the brain functions**
- 
- **huge revival with success of ‘deep networks/learning’**
  - **... still far from ‘intelligent’ though... ☹**

# Outline

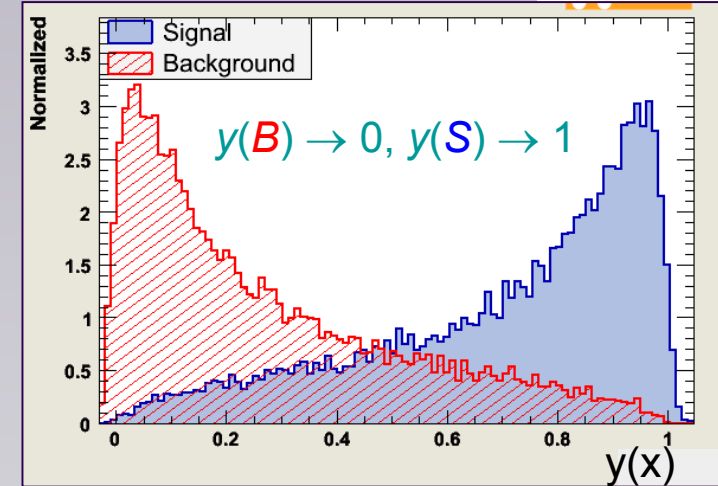
- Small recap of yesterdays “ $y(x)$ ”
- **What are neural networks** (simple vanilla feed forward nets)
  - Loss function
  - Backpropagation
- **Deep Learning – advances that made it possible**
  - Weight initialisation
  - SGD  $\rightarrow$  momentum  $\rightarrow$  auto tuned learning rates
  - Regularisation  $\rightarrow$  Dropout
- **Other network types:**
  - Auto encoder
  - Convolutional Neural Networks
- **Examples of their usage in (astro-) particle physics**

# Classification $\leftrightarrow y(x)$

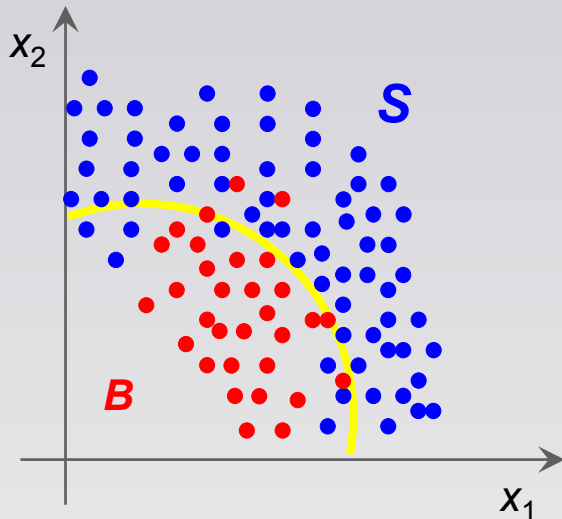
## Classification:

- $y(x): \mathbb{R}^D \rightarrow \mathbb{R}$ : “test statistic” in  $D$ -dimensional space of input variables
- $y(x)=\text{const}$ : surface defining the decision boundary.

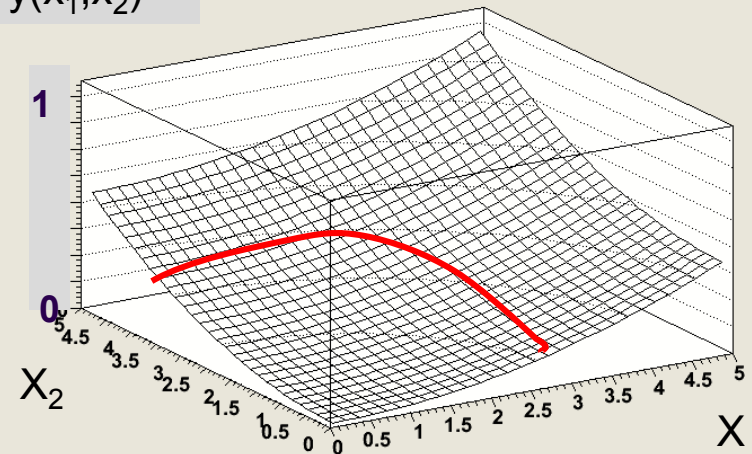
$y(x): \mathbb{R}^D \rightarrow \mathbb{R}$ :



$y(x)$ : function whose ‘contour lines’ define reasonable (good) decision boundaries



$y(x_1, x_2)$



# $y(x)$ – the MVA output

Assume you have found the  $y(x)$  which gives ‘perfect’ decision boundaries for any desired ‘signal efficiency’

perfect == one cannot do any better

$$\rightarrow y(x) == \frac{pdf(x|S)}{pdf(x|B)} \quad (\text{or a monotonic function thereof})$$

If  $y(x)$  is ‘forced’ to be between 0,1 (e.g. using the logistic/sigmoid function  $y(x) \rightarrow \text{sigm}(y(x))$  like in ‘logistic regression’) AND

$$L = y_i^{train} \log(y(x_i)) + (1 - y_i^{train}) \log(1 - y(x_i)) \quad \text{binomial loss}$$

Which came from:  $y(x)$  should simply parametrize  $P(S|x)$ ;  $P(B|x)=1-P(S|x)$

$$L = - \sum_i^{events} \log(P(y_i^{train}|y(x_i))) = - \sum_i \log(P(S|x_i)^{y_i^{train}} P(B|x_i)^{1-y_i^{train}})$$

**THEN  $y(x)$  parametrizes directly  $P(S|x)$**

“arbitrary” non-linear decision boundaries

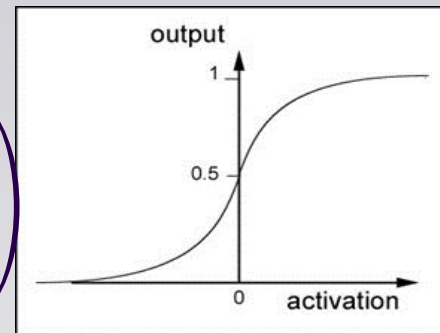
$$y(\vec{x}) = \text{sigmoid} \left( \sum_k^M w_k h_k(\vec{x}) \right)$$

- $y(x)$  built from set of “basis” functions  $h_k(x)$
- $h(x)$  is sufficiently general (i.e. non linear),  
→ Can model any function

there are also mathematical proves for this statement.

Imagine you chose do the following:

$$y(x) = A \left( \sum_k^M w_k \underbrace{A \left( w_{k0} + \sum_{jj=1}^D w_{kj} x_{jj} \right)}_{h_k(x)} \right)$$



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

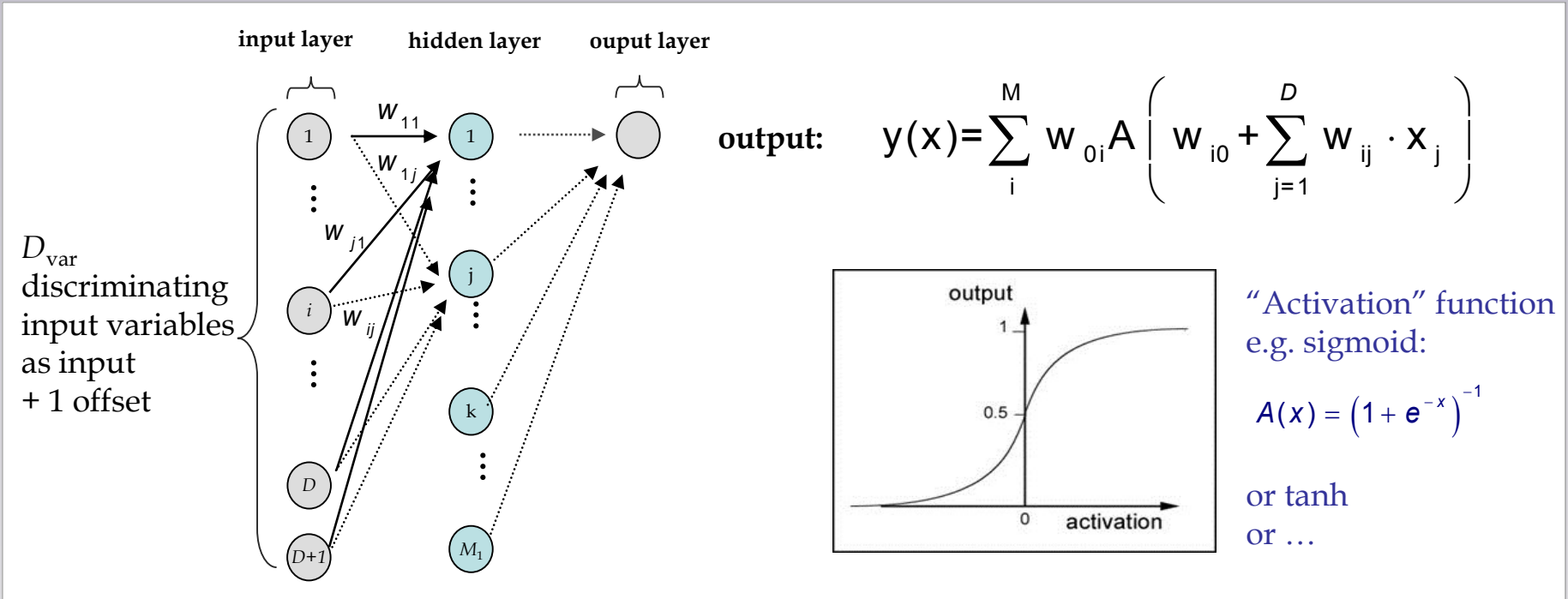
the sigmoid function

A non linear (sigmoid) function of  
a linear combination of  
non linear function(s) of  
linear combination(s) of  
the input data

Ready is the Neural Network  
Now we “only” need to find the appropriate “weights”  $w$

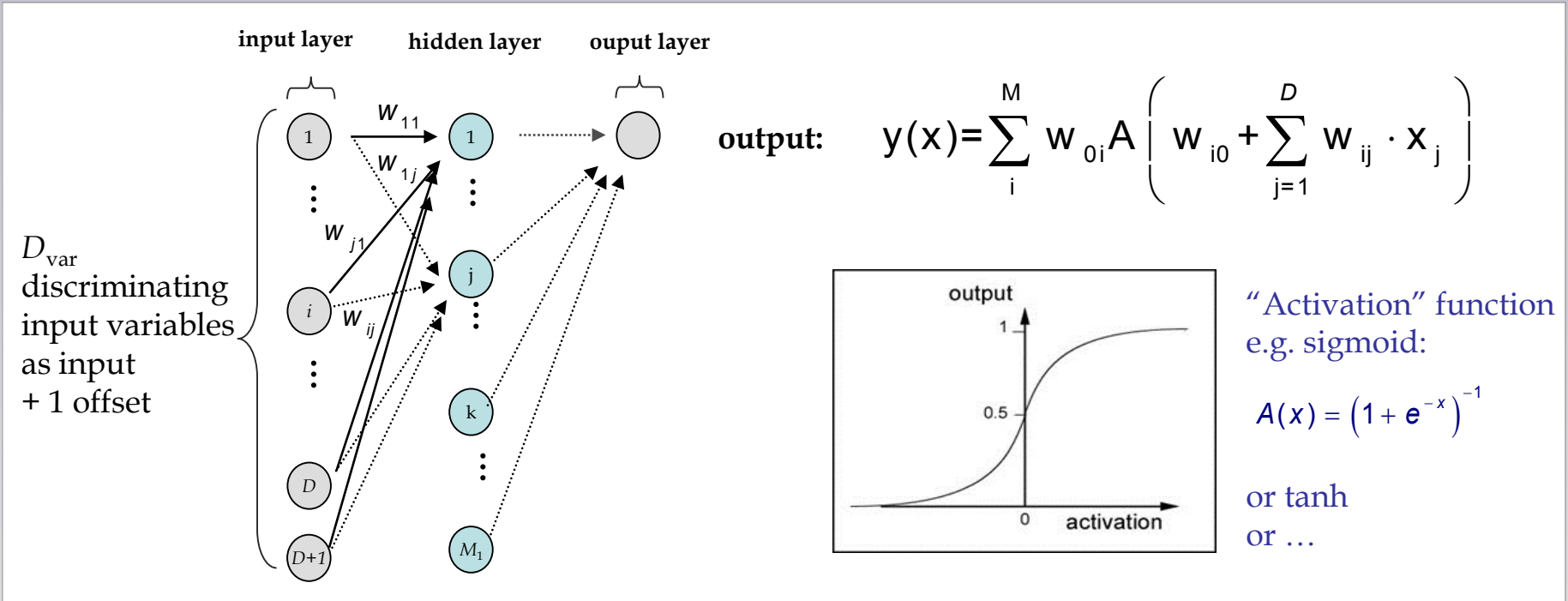
# Neural Networks: Multilayer Perceptron MLP

But before talking about the weights, let's try to “interpret” the formula as a Neural Network:

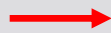


- Nodes in hidden layer represent the “activation functions” whose arguments are linear combinations of input variables → non-linear response to the input
- The output is a linear combination of the output of the activation functions at the internal nodes
- Input to the layers from preceding nodes only → feed forward network (no backward loops)
- It is straightforward to extend this to “several” input layers

# Neural Networks: Multilayer Perceptron MLP



nodes → neurons  
links(weights) → synapses



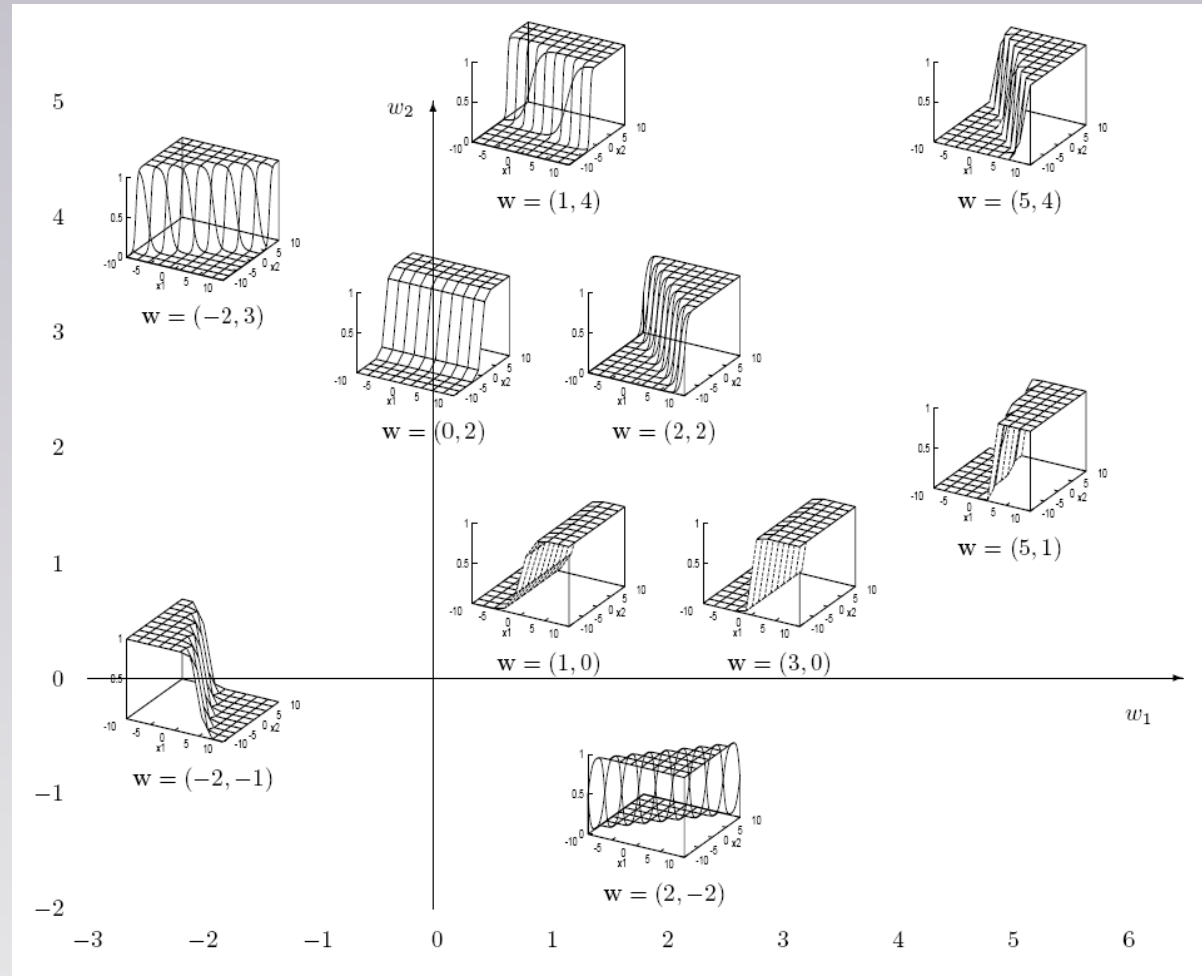
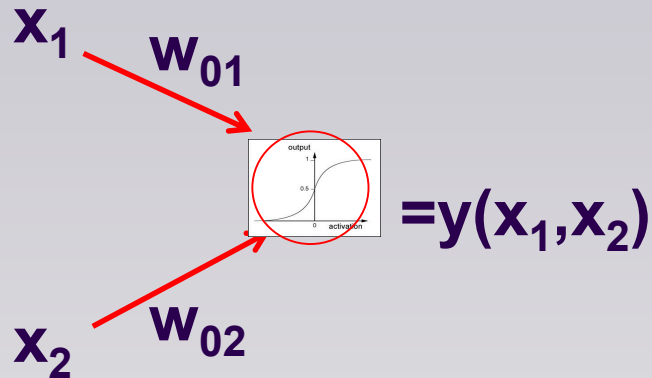
Neural network: try to simulate reactions of  
a brain to certain stimulus (input data)

‘activation’ of output node: linear(→ regression)    sigmoid(→ classification)



# $y(x)$ from Neural Network

“NN” with two input variables and ‘one node’



Choose those weights where contourlines == good decision boundaries

# Training → Minimize Loss Function

regression:

$$L(w) = \frac{1}{2} \sum_i^{\text{events}} \underbrace{\left( y_i^{\text{train}} \right)}_{\text{true}} - \underbrace{y(x_i; w)}_{\text{predicted (the network output)}} \bigg)^2$$

i.e. use usual “sum of squares”

classification: Binomial loss

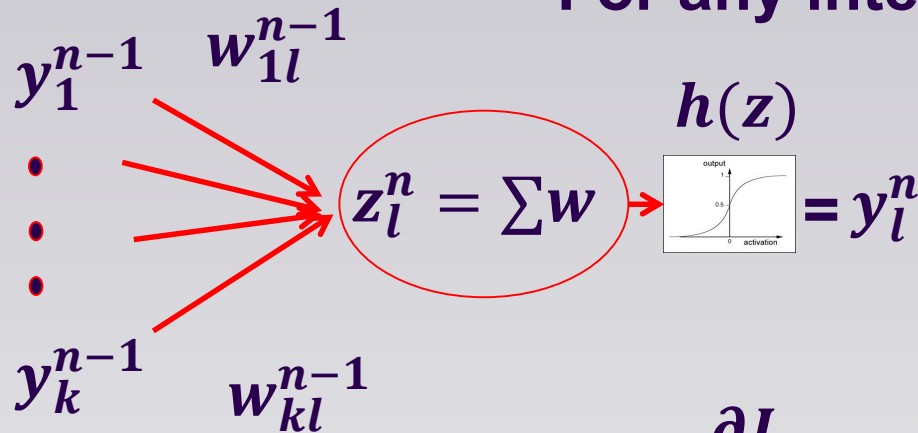
$$L(w) = \sum_i^{\text{events}} \left( y_i^{\text{train}} \log(y(x_i; w)) + (1 - y_i^{\text{train}}) \log(1 - y(x_i; w)) \right)$$

where  $y^{\text{train}} = \begin{cases} 1, & \text{signal} \\ 0, & \text{backgr} \end{cases}$

# Back-propagation

- recursive formulation of the gradient  $\frac{\partial L}{\partial w_{ij}}$  using ‘chain rule’
- ‘adjust’ weights  $w$  to minimize the “loss function”

**For any internal node: i.e. node  $l$  in layer  $k$**



$$\frac{\partial L}{\partial w_{kl}} = \frac{\partial z_l}{\partial w_{kl}} \frac{\partial h}{\partial z_l} \frac{\partial L}{\partial h} = y_k^{n-1} \frac{\partial h}{\partial z_l} \frac{\partial L}{\partial h}$$


**... etc...**

# Back-propagation

- recursive formulation of the gradient  $\frac{\partial L}{\partial w_{ij}}$  using ‘chain rule’

→ For the ‘last layer’ we get:

Output of k-th node in  
previous layer



$L = \frac{1}{2} (y - y(x))^2$  and linear output neuron:  $y(x) = \sum w_{nk} y_k = z$

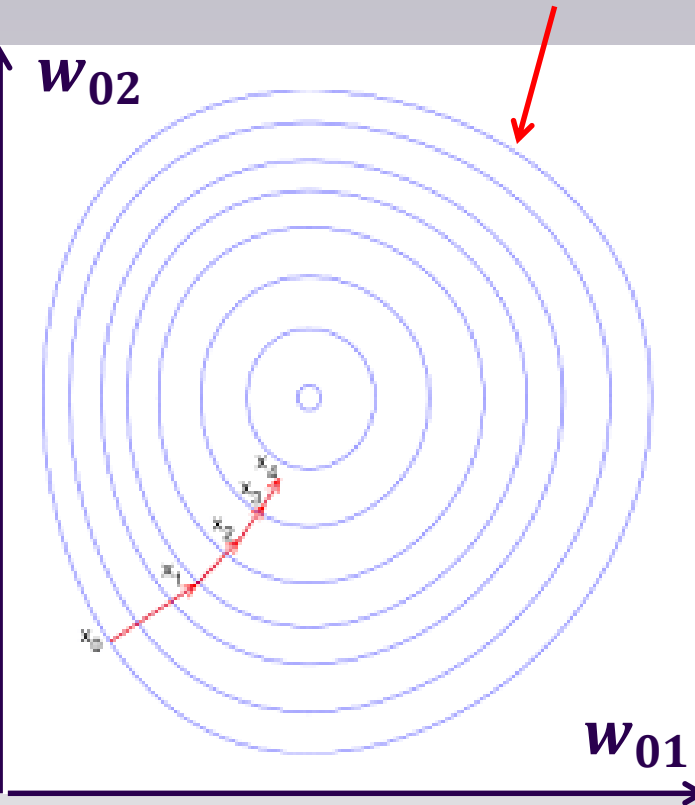
$$\frac{\partial L}{\partial w_{nk}} = \frac{\partial y(x)}{\partial w_{nk}} \frac{\partial L}{\partial y(x)} = y_k (y - y(x))$$

And: for ‘binomial loss function’ and ‘sigmoid output neuron’

→ Same result ☺  $\frac{\partial L}{\partial w_{nk}} = \dots = y_k (y - y(x))$

# (Stochastic) Gradient Descent SDG

Contour plot of  $E(L(w))$  or  $L(w|x_k)$  for event  $k$  **learning rate**



$$w_{ij} \rightarrow w_{ij} - \eta \frac{\partial E(L)}{\partial w_{ij}} : \text{gradient decent}$$

and if you don't want to evaluate the expectation value every time for the whole sample:

stochastic gradient decent: event by event

$$w_{ij} \rightarrow w_{ij} - \eta \frac{\partial L(event_k)}{\partial w_{ij}}.$$

mostly: something in between  $\rightarrow$  mini-batches

$\rightarrow$  Assume 'average' of mini-batch gradients

approximates the 'gradient' of the  $E(L)$  (i.e. full sample)

Sounds simple and if error- surface looks THAT simple..... BUT:

# Stochastic Gradient Decent

- $y(x)$  and  $L(x;w)$  are nasty, heavily non-parabolic functions
  - difficult to minimize
  - Long time people thought to be trapped in local minima:
  - But were more likely walking slowly along narrow valleys

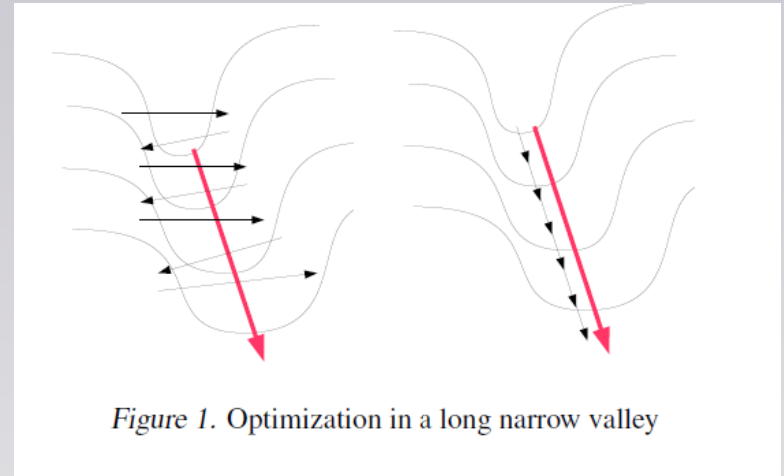


Figure 1. Optimization in a long narrow valley

- **Add “Momentum”**
  - accelerate when gradient direction stays ‘constant’

$$v \rightarrow \mu v - \eta \nabla L \quad ; \quad w_{ij} \rightarrow w_{ij} + v \quad (\mu \text{ called momentum})$$

# Neural Networks and Local Minima

large NNs are difficult to ‘train’!

→ but due trapping in local minima?

... recent research suggests:

*arXiv:1412.0233, LeCun et.al.*

Different in ‘many dimensions’ !

- For large networks: most local minima are equivalent
- Probability for finding a bad (high value) local minimum is non zero for small-size networks but decreases quickly with network size
- Global minimum is not useful → represents overtraining
- Bad critical points (much higher than global minimum) are mostly ‘saddle points’

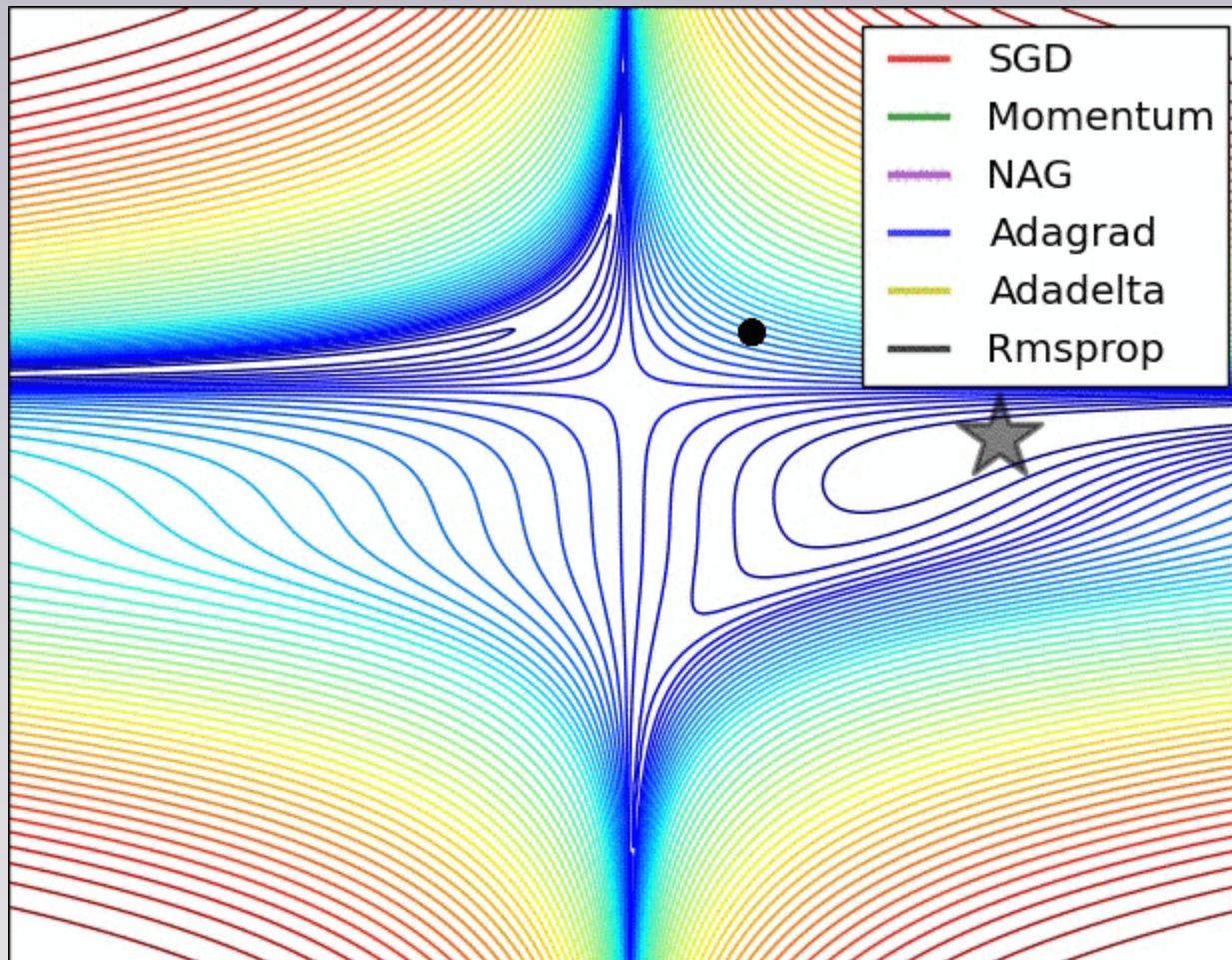
You might have seen analogies  
such as this here.....



**It's very unlikely that all  $\frac{d^2L}{dw^2} > 0$  at the same time ☺**



# Gradient Descent

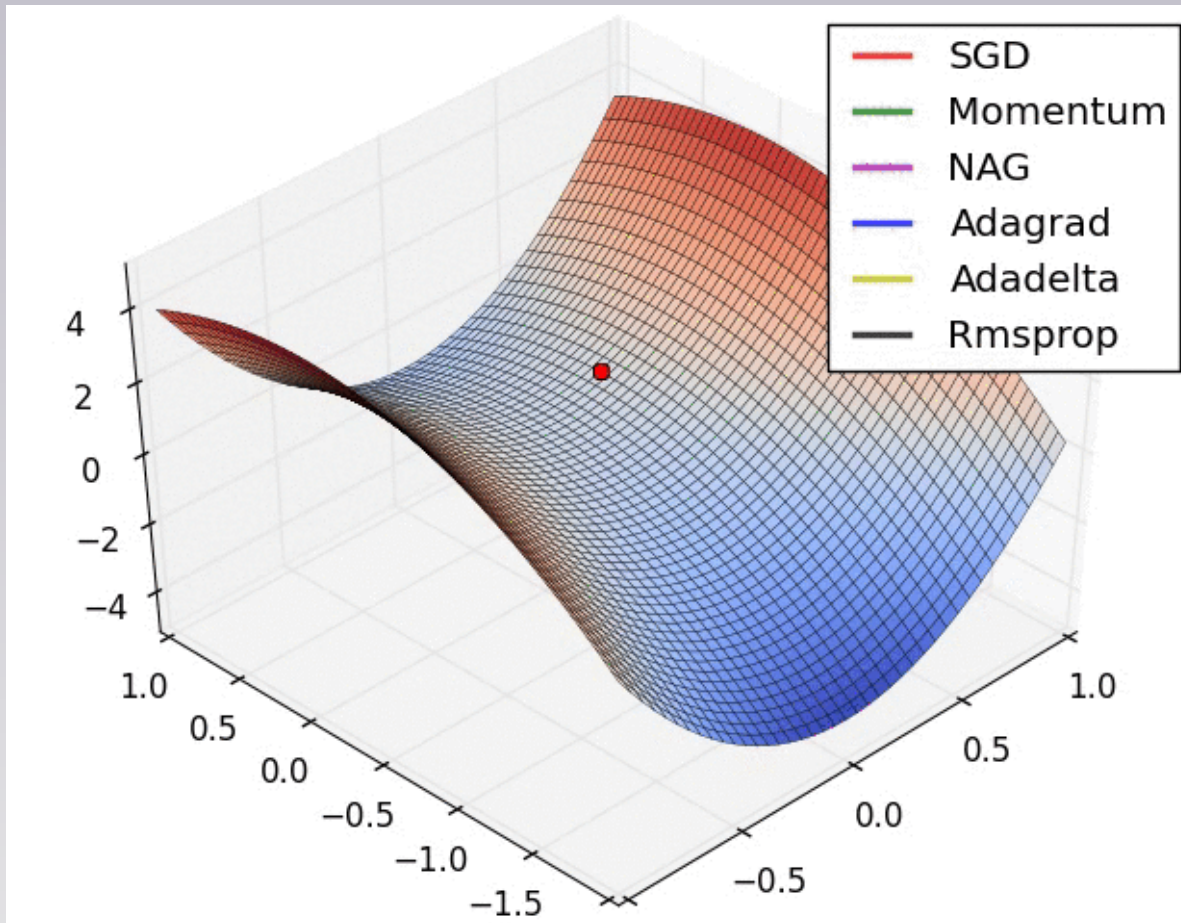


StanfordLectureCS231:Image AlecRadford



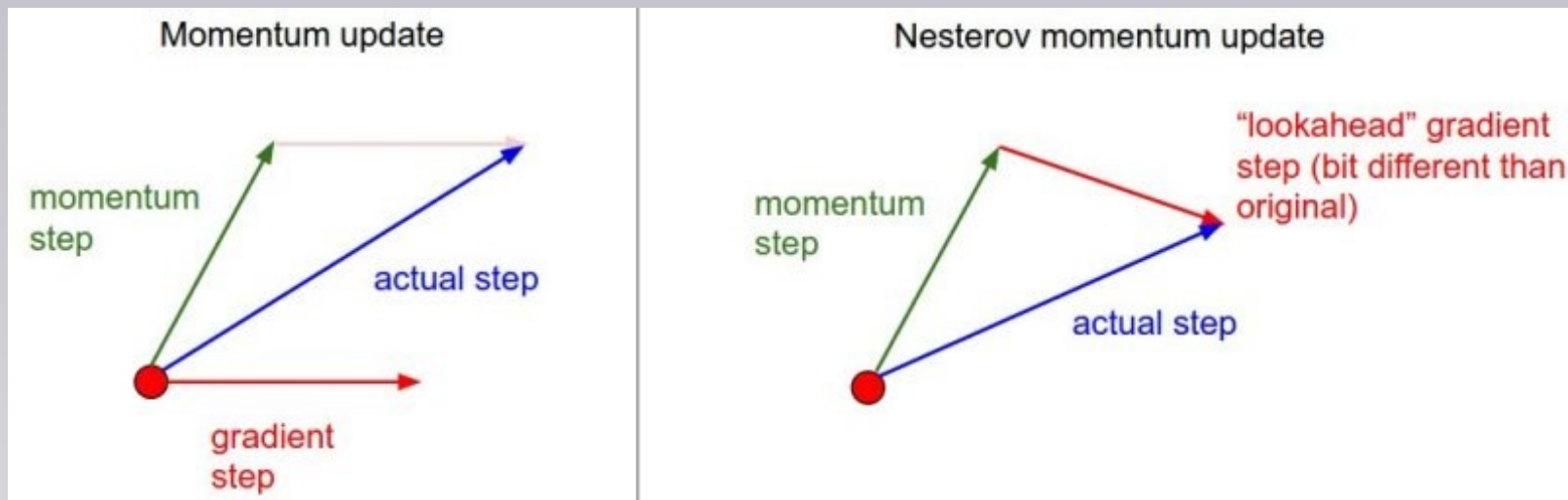
# Gradient Decent

→ escaping the saddle points



StanfordLectureCS231:Image AlecRadford

# Nesterov Accelerated Momentum



StanfordLectureCS231

Idea: Yuri Nesterov (1983) ...

First look where you would 'end up' following your 'momentum' and correct for gradient you would 'see there'

# RMSProp

[http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture\\_slides\\_lec6.pdf](http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf)

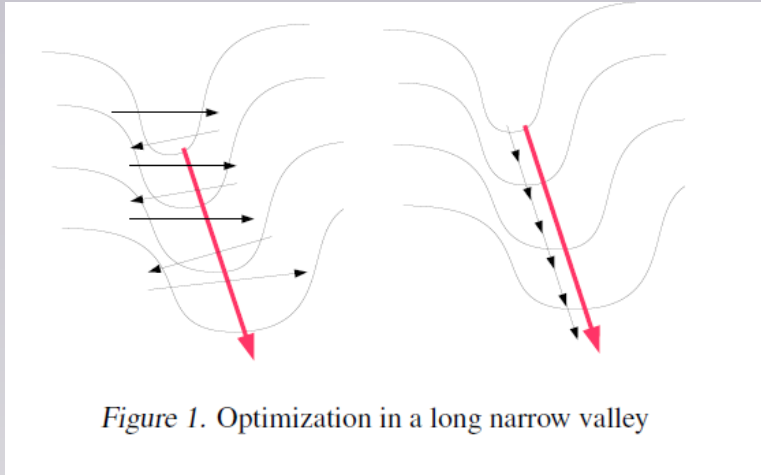


Figure 1. Optimization in a long narrow valley

- “change of sign” → more important than “size” of the gradient
- **Rprop** (resilient backpropagation 1993)

**Rprop: problems with large fluctuations in minibatches**  
→ **RMSprop: scaling weight update by ‘running RMS’ of gradients**

# Neural Network Training

NN with ‘many hidden layers’ → used to be ‘impossible to train’

→ due to vanishing gradient problem:  $\frac{\partial L}{\partial w_{ij}} \approx 0$  for all but the last layer(s)

- Enormous progress in recent years
  - Layerwise pre-training using ‘auto-encoders’ or ‘restricted-Boltzman machines’
  - ‘new’ activation functions whose gradient do not vanish
  - ‘intelligent’ random weight initialisation
  - Stochastic gradient decent with ‘momentum’

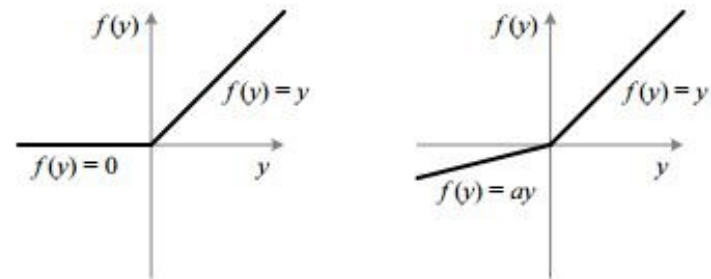
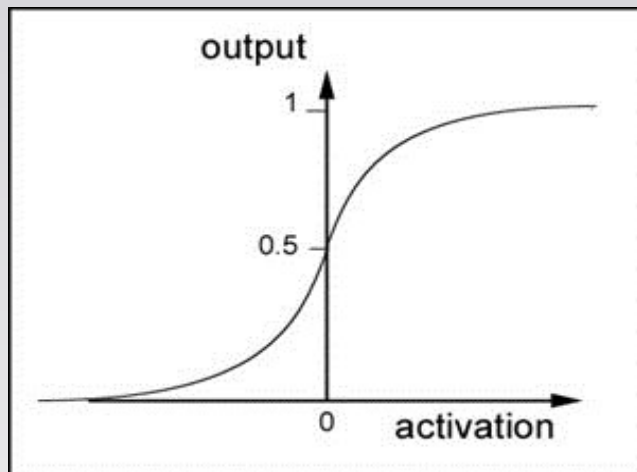


Figure 1. ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

# Weight initialisation

- Used to set all weights, randomly with small value  
→ almost linear classifier
- Set weight via ‘pretraining’ each layer seperatly using auto-encoder
- Set weights randomly but such that in each layer (regardless of #inputs to the nodes) the node activations are normally distributed with ‘same’ variance

# Neural Network Regularisation

**control model complexity:** (deep networks can have O(millions) of weights!)

- #nodes and # layers
- early stopping → very first (old) NN ‘regularizer’
  - Start with small random weights → sigmoid approximately linear → essentially a linear model → stop before it deviates too much from that
- Weight decay:
  - add ‘regularizing’ term to the loss function  $L = L + \frac{1}{2} \sum w^2$ 
    - == ‘Gaussian prior centered at zero’ for the weights
  - Favours small weights → i.e. simpler models

# Regularisation: weight decay

Minimize loss function: e.g. via  $W \rightarrow W - \eta \nabla_W L$ : SGD

Include prior distribution on ‘weights’/‘parameters’  $W$ :

$$L = -\log\left(\prod_i^{\text{events}} P(y_i^{\text{train}} | y(x_i; W)) * p(W)\right)$$

$$= -\left(\sum_i^{\text{events}} \log(P(y_i^{\text{train}} | y(x_i; W))) - \log(p(W))\right)$$

often (e.g if  $y$  = polynomial or  $y$  = neural network)

$W$  “small”  $\rightarrow$  model is less ‘flexible’

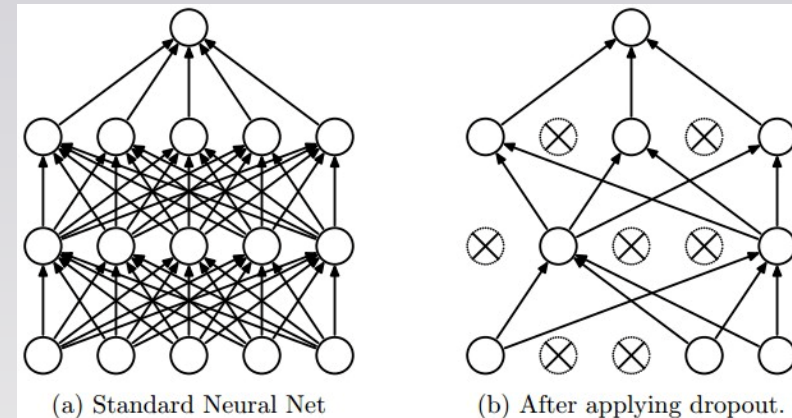
$\rightarrow$  reasonable prior  $p(W)$  would be: Gaussian with mean zero

$\rightarrow L = L + \frac{1}{2} \sum w^2$  **called ‘weight decay’**

# Neural Network Regularisation

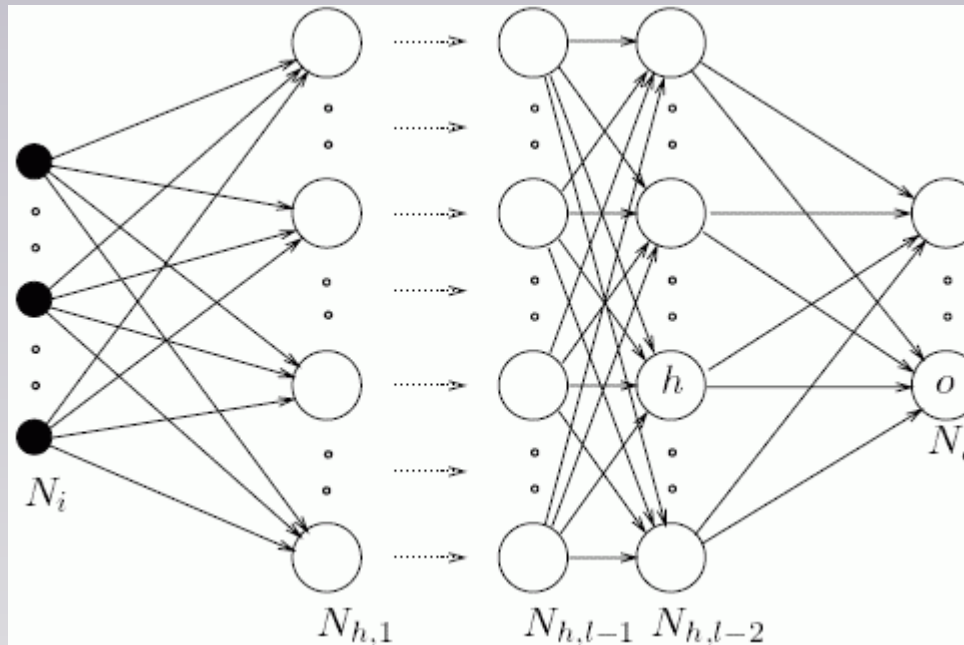
**control model complexity:** (deep networks can have O(millions) of weights!)

- #nodes and # layers
- early stopping → very first (old) NN ‘regularizer’
  - Start with small random weights → sigmoid approximately linear → essentially a linear model → stop before it deviates too much from that
- Weight decay:
  - add ‘regularizing’ term to the loss function  $L = L + \frac{1}{2} \sum w^2$ 
    - == ‘Gaussian prior centered at zero’ for the weights
  - Favours small weights → i.e. simpler models
- Dropout
  - Randomly remove nodes during each training step
  - Avoid co-adaptation of nodes
  - Essentially a large model averaging procedure like ‘bagging’





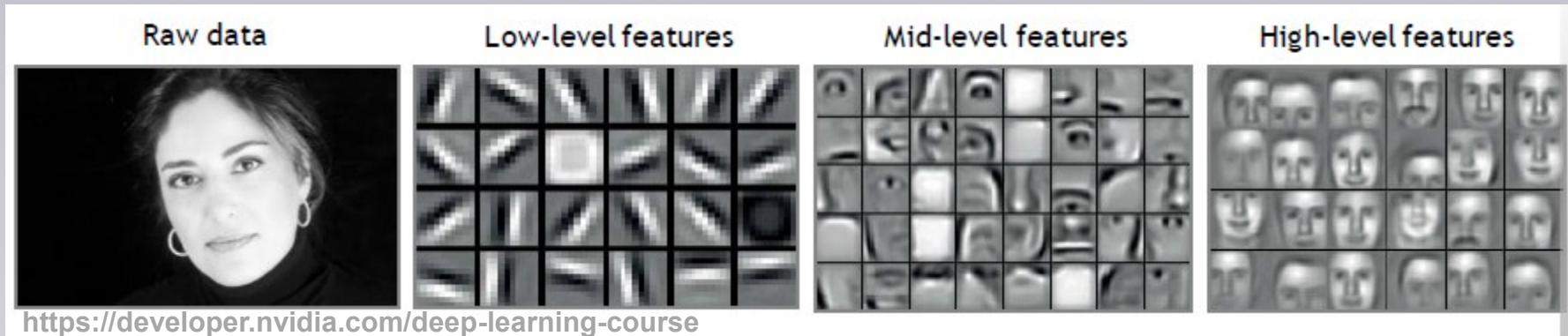
# Deep Networks == Networks with many hidden layers



- That's at first sight “all” it means...

# Deep Learning

NN: 'many hidden layers' → hierarchy of features



- Getting rid of “hand crafted features” → revolutionized:
  - Image recognition
  - Speech recognition, Natural Language Processing
- HEP ?
  - No ‘high’ level features need anymore, just 4-vectors?

# Learning HEP features

Search for exotic particles in high energy physics with deep learning

P.Baldi, P. Sadowski, D. Whiteson, Nature Communications 5, Article: 4308 (2014)

$$gg \rightarrow H^0 \rightarrow W^\mp H^\pm \rightarrow W^\mp W^\pm h^0 \rightarrow W^\mp W^\pm b\bar{b}, \quad (1)$$

**signal**

**$t\bar{t} \rightarrow WbWb$  : background**

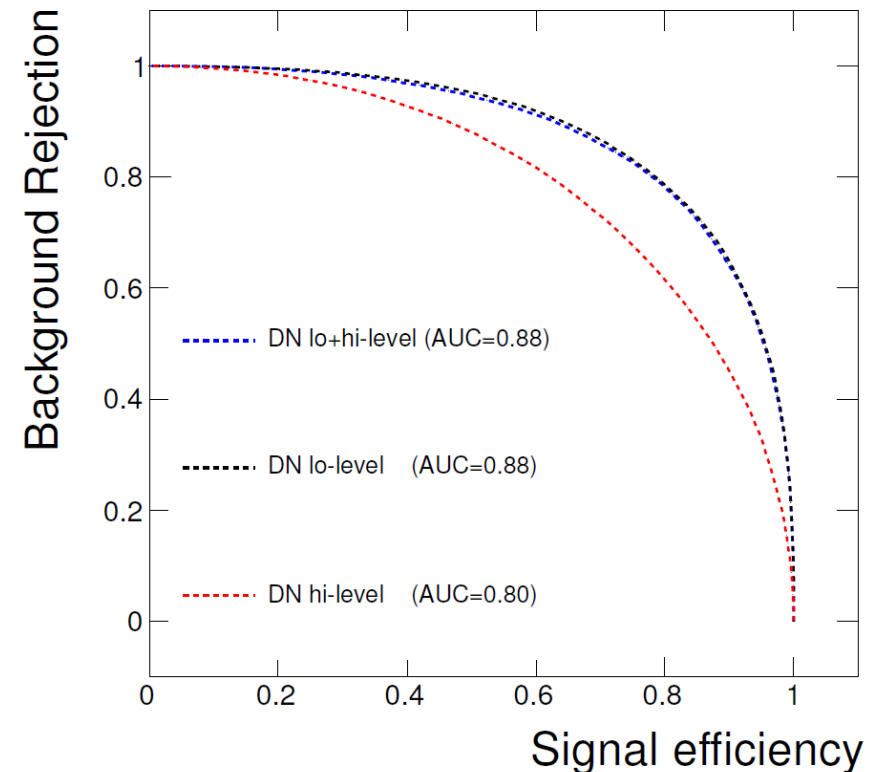
## ■ High level features:

$m_{jj}, m_{lv}, m_{jlv}, m_{jbb}$

AUC			
Technique	Low-level	High-level	Complete
BDT	0.73 (0.01)	0.78 (0.01)	0.81 (0.01)
NN	0.733 (0.007)	0.777 (0.001)	0.816 (0.004)
DN	0.880 (0.001)	0.800 (< 0.001)	0.885 (0.002)

Discovery significance			
Technique	Low-level	High-level	Complete
NN	$2.5\sigma$	$3.1\sigma$	$3.7\sigma$
DN	$4.9\sigma$	$3.6\sigma$	$5.0\sigma$



# Deep Learning for SUSY

P.Baldi, P. Sadowski, D. Whiteson, Nature Communications 5, Article: 4308 (2014)

case). Instead, a great deal of intellectual energy has been spent in attempting to devise features which give additional classification power. These include high-level features such as:

- Axial  $\cancel{E}_T$ : missing transverse energy along the vector defined by the charged leptons,
- stransverse mass  $M_{T2}$ : estimating the mass of particles produced in pairs and decaying semi-invisibly [17, 18],
- $\cancel{E}_T^{Rel}$ :  $\cancel{E}_T$  if  $\Delta\phi \geq \pi/2$ ,  $\cancel{E}_T$  otherwise where  $\Delta\phi$  is the minimum angle between a jet or lepton,
- razor quantities  $\beta_{R,1}$  and  $\beta_{R,2}$
- super-razor quantities  $\beta_{R+1}$ ,  $M_{R,T}^T$ , and  $\sqrt{\hat{s}_R}$  [20].

■ Puh... physicists still did a good job

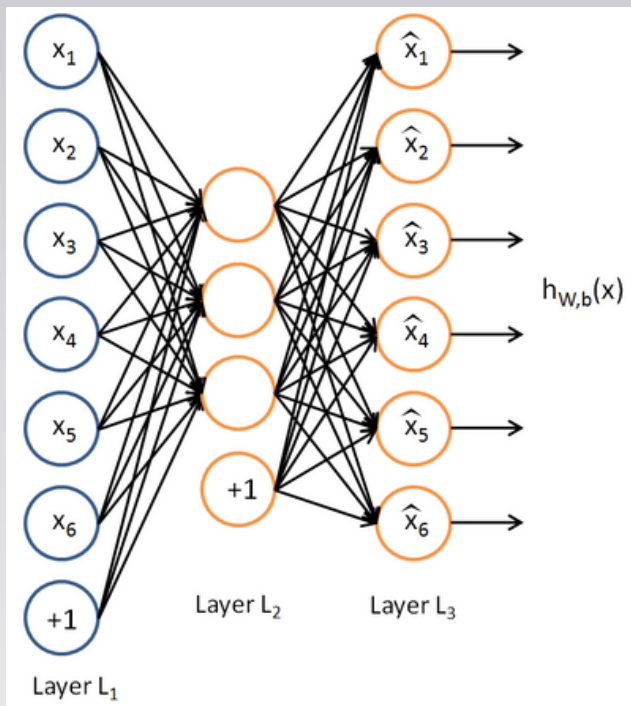
→ Little BUT statistically significant gain using Deep Neural Network

Technique	AUC		
	Low-level	High-level	Complete
BDT	0.850 (0.003)	0.835 (0.003)	0.863 (0.003)
NN	0.867 (0.002)	0.863 (0.001)	0.875 (< 0.001)
NN <sub>dropout</sub>	0.856 (< 0.001)	0.859 (< 0.001)	0.873 (< 0.001)
DN	0.872 (0.001)	0.865 (0.001)	0.876 (< 0.001)
DN <sub>dropout</sub>	0.876 (< 0.001)	0.869 (< 0.001)	0.879 (< 0.001)

■ Note: High level features were hardly ‘needed’ in DNN

# Deep Learning for SUSY

- These Deep learning studies using '4-vectors' used very large fast simulated MC samples
  - Might be 'infeasible' for 'real' analysis
  - Perhaps 'revive' idea of auto-encoder pre-training using 'real data' ??



## Auto-encoder:

- network that 'reproduces' its input
- hidden layer < input layer
  - hidden layer 'dimensionality reduction'
  - needs to 'focus/learn' the important features that make up the input
- Hidden layer > input layer + sparsity enforced
  - interesting features

# Deep Neural Networks and HEP

We (TMVA) used to say:

- Typically in high-energy physics, non-linearities are reasonably simple,
  - 1 layer with a larger number of nodes probably enough
  - still worth trying 2 (3?) layers (and less nodes in each layer)

But Higgs ML Challenge:

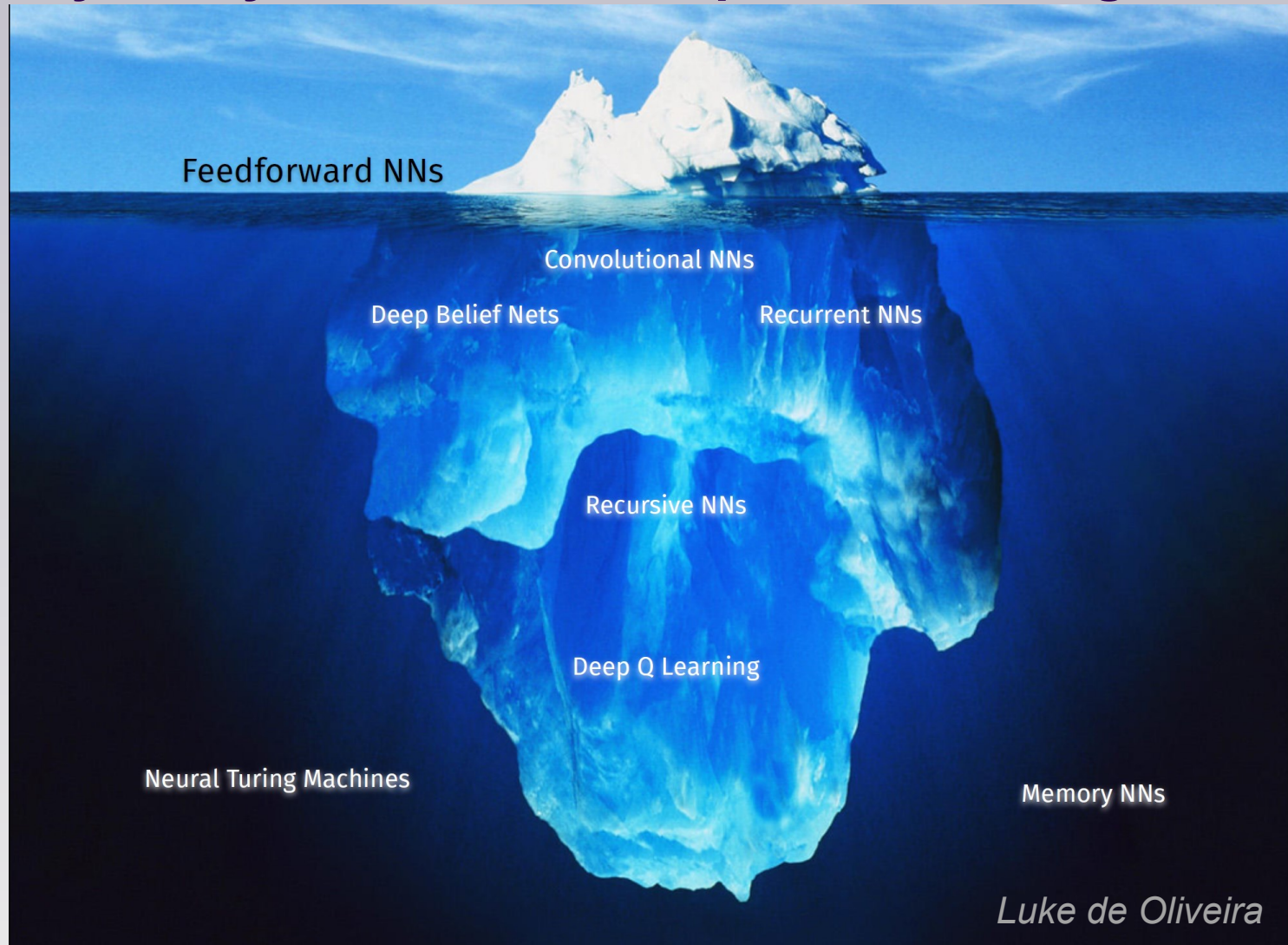
The screenshot shows the Kaggle website interface for the 'Higgs Boson Machine Learning Challenge'. The header includes navigation links like 'Customer Solutions', 'Competitions', 'Community', 'Sign up', and 'Login'. The main title is 'Higgs Boson Machine Learning Challenge' with a subtitle '\$13,000 • 1,602 teams'. It indicates the competition started on 'Mon 12 May 2014' and ends on 'Mon 15 Sep 2014 (12 days to go)'. A sidebar on the left contains links for 'Dashboard', 'Home', 'Data', 'Make a submission', 'Information', 'Forum', 'Leaderboard', and a 'Leaderboard' section listing top performers: 1. Gábor Melis, 2. Tim Salimans, 3. Luboš Motl's team, and 4. nhixShaze. The main content area features a large image of the ATLAS experiment detector with the text 'Use the ATLAS experiment to identify the Higgs boson' and technical details like 'Run: 204153', 'Event: 35369265', and '2012-05-30 20:31:28 UTC'.

- Won by a 'Deep Neural Network'
- well... 3 hidden layers with 600 nodes each



# Deep Learning

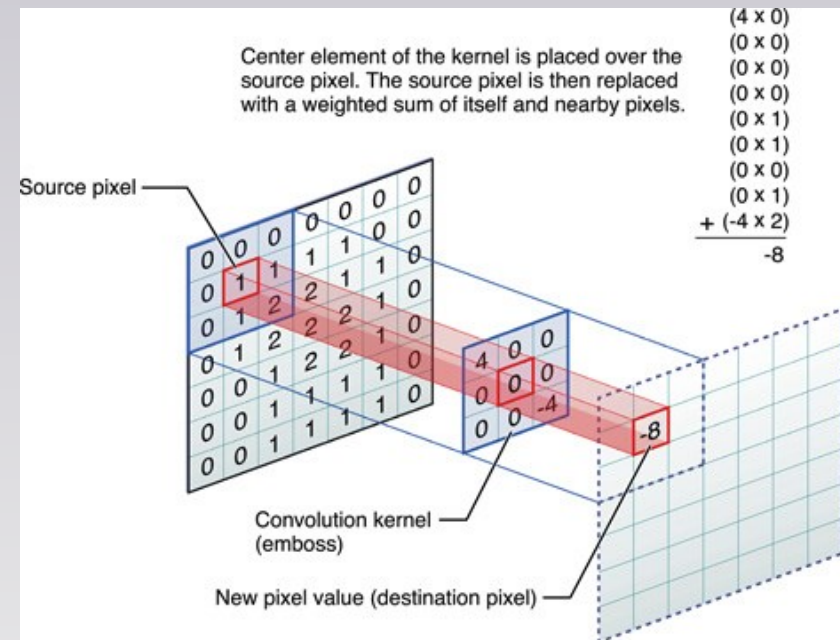
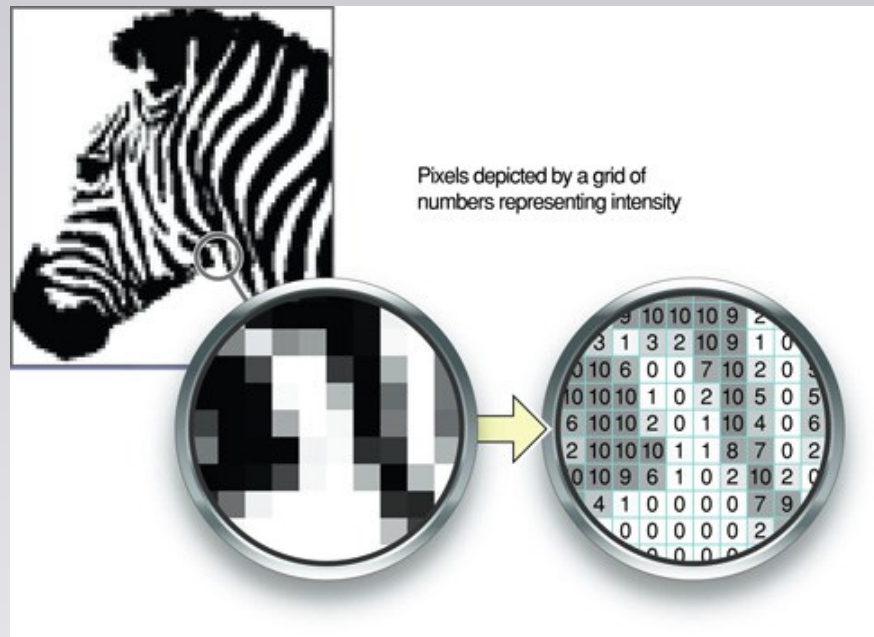
- Obviously, I only scratched the ‘tip of the iceberg’



# Convolutional Neural Networks

Images are 2D arrays of 'numbers'

- neighbouring pixels in images are 'correlated'
- Convolutional Neural Network: same kernel applied over the whole image

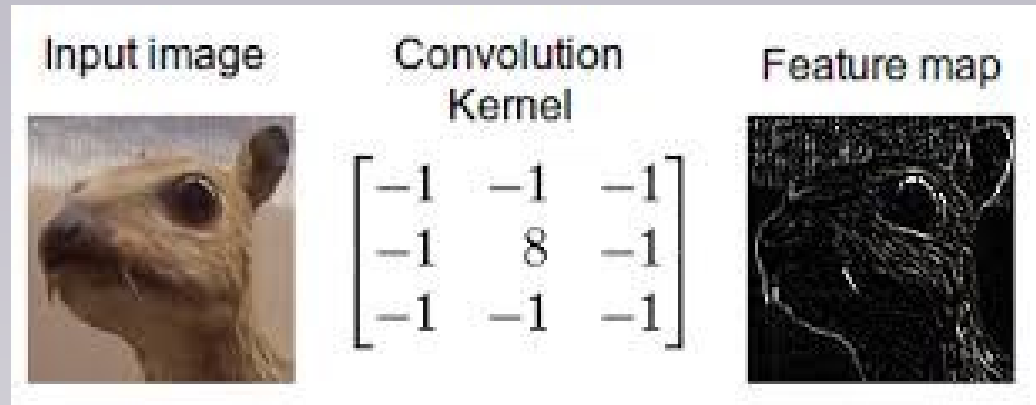


<https://developer.apple.com/library/ios/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html>

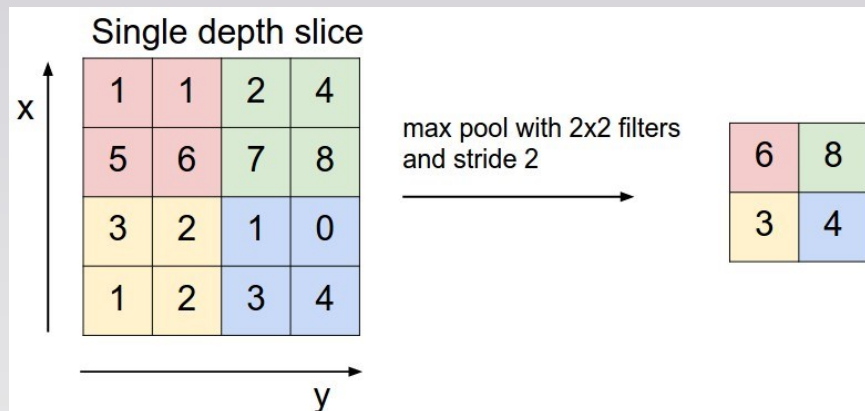
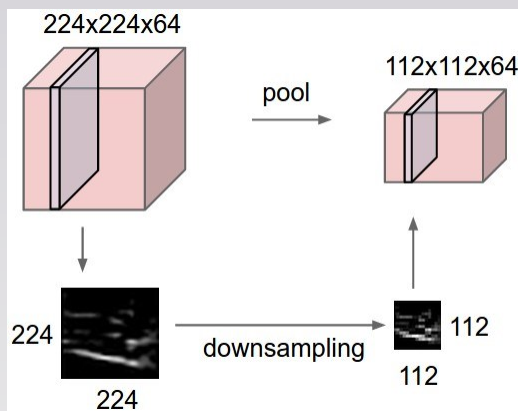


# Convolutional Neural Networks

## Example:

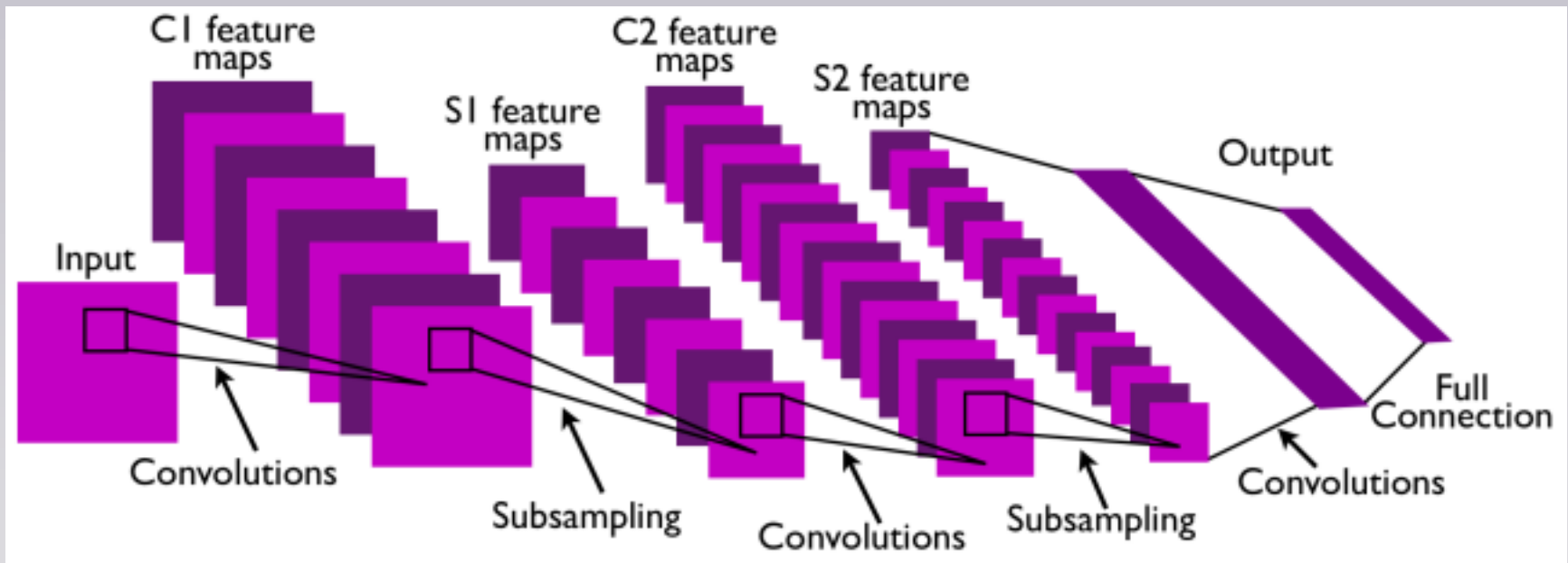


In practice: often followed by ‘down sampling’ – pooling step



<http://cs231n.github.io/convolutional-networks/>

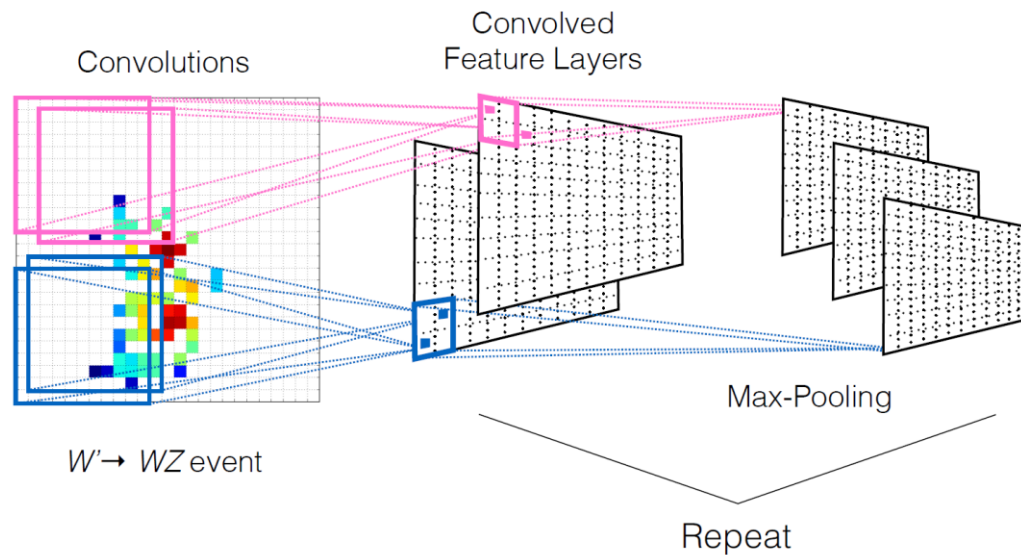
# Convolutional Neural Networks



# Deep Learning

## ■ Jet images in the calorimeter

Even more non-linearity: Going Deep



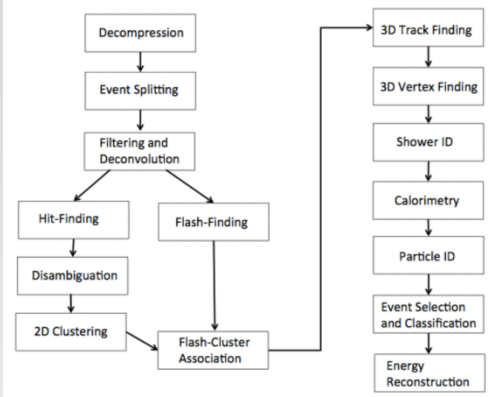
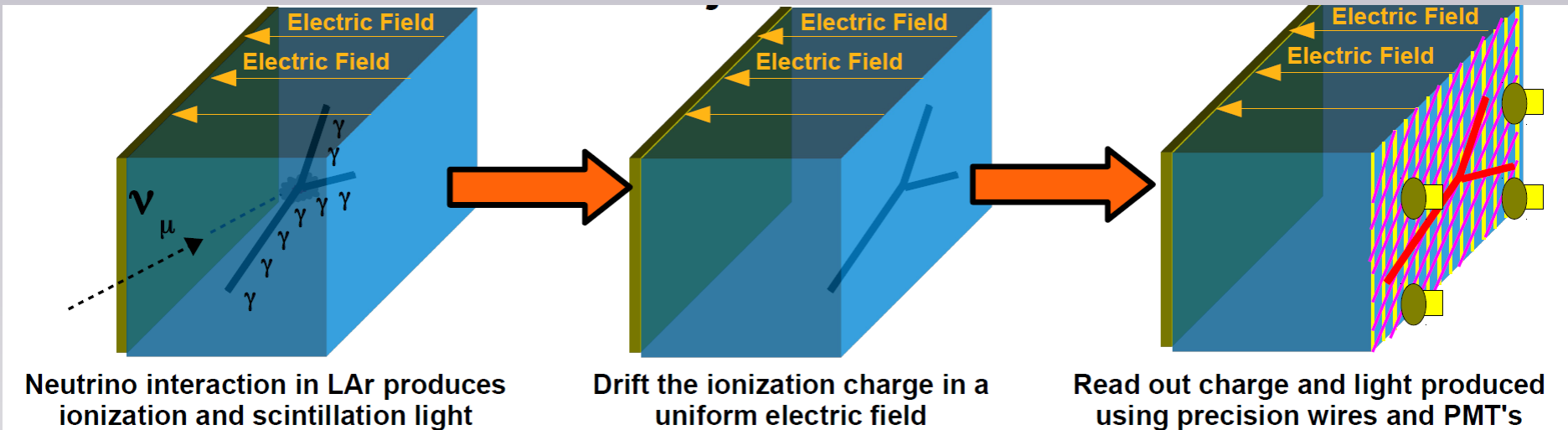
Apply deep learning techniques on jet images! [3]

*convolutional nets are a standard image processing technique; also consider maxout*

(L.Oliveira, M.Kagan... [arXiv:1511.05190](https://arxiv.org/abs/1511.05190))

# 'Images' from LArTPC in DUNE

- Neutrino Experiment: Type + Energy of interacting neutrino
- TPC: Tracker, ParticleID and Calorimeter



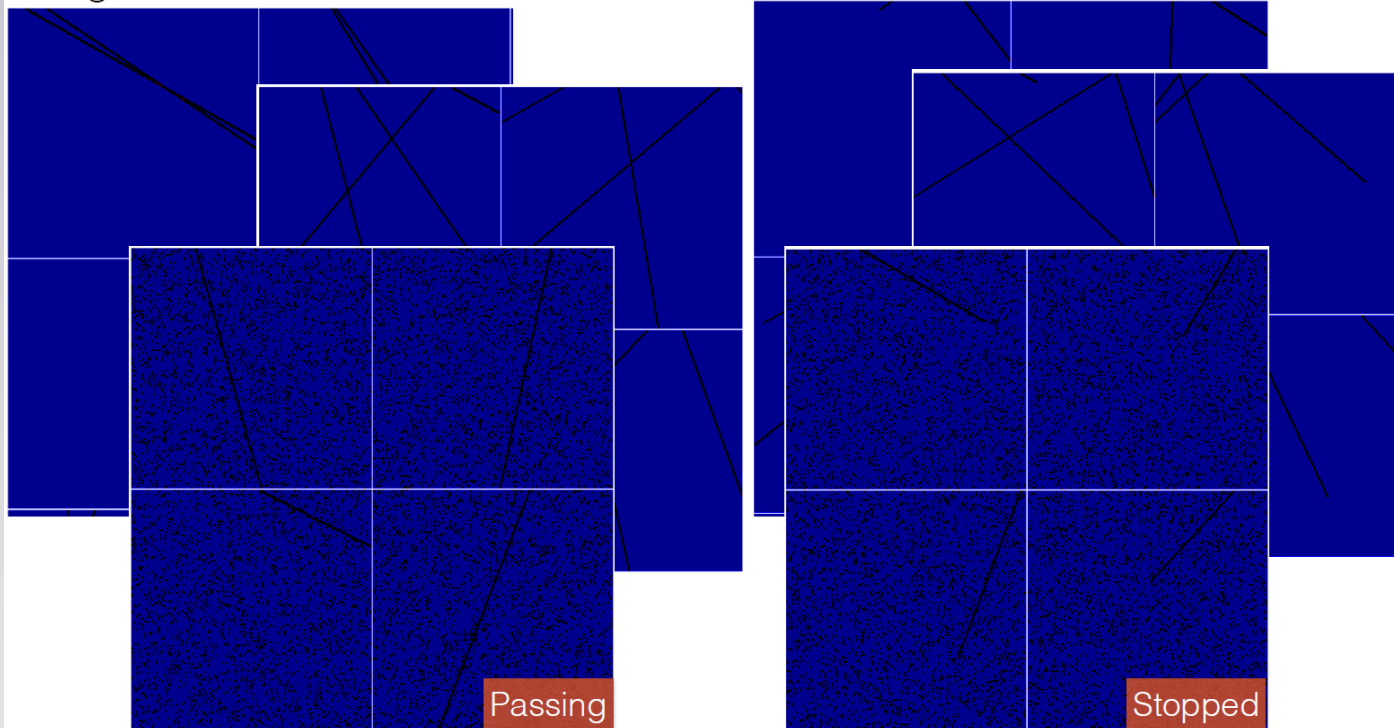
- Fully automated reconstruction using Deep Neural Network?

# 'Images' from LArTPC

## Initial toy study: event (track) reconstruction

### DNN Classification of "Raw" LArTPC Data

GoogleLeNet 256x256

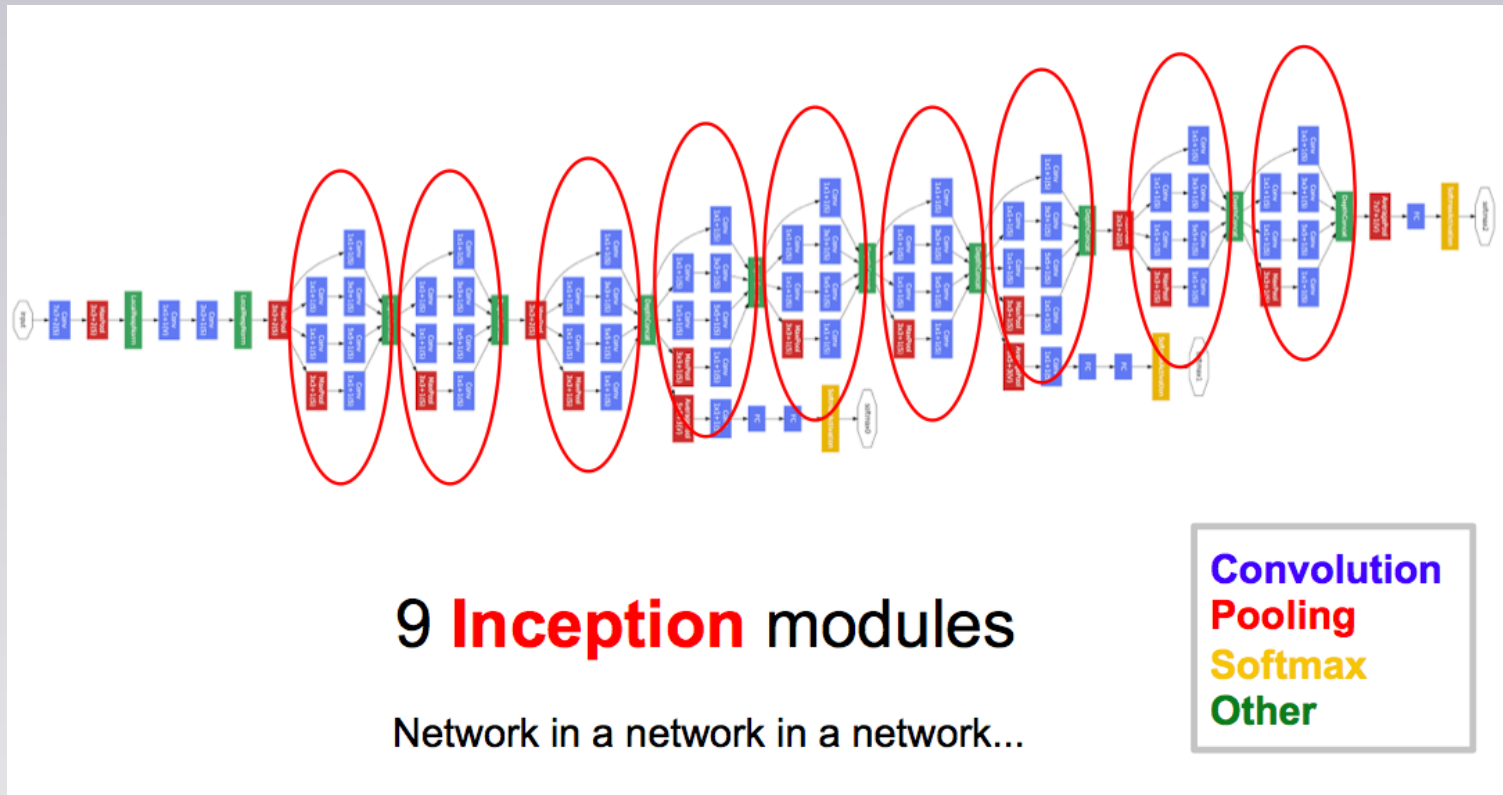


1-4 Tracks With or without noise, DNN correctly classifies ~90-99%

*Amir Farbin, University of Texas*

# More 'Images': LArTPC

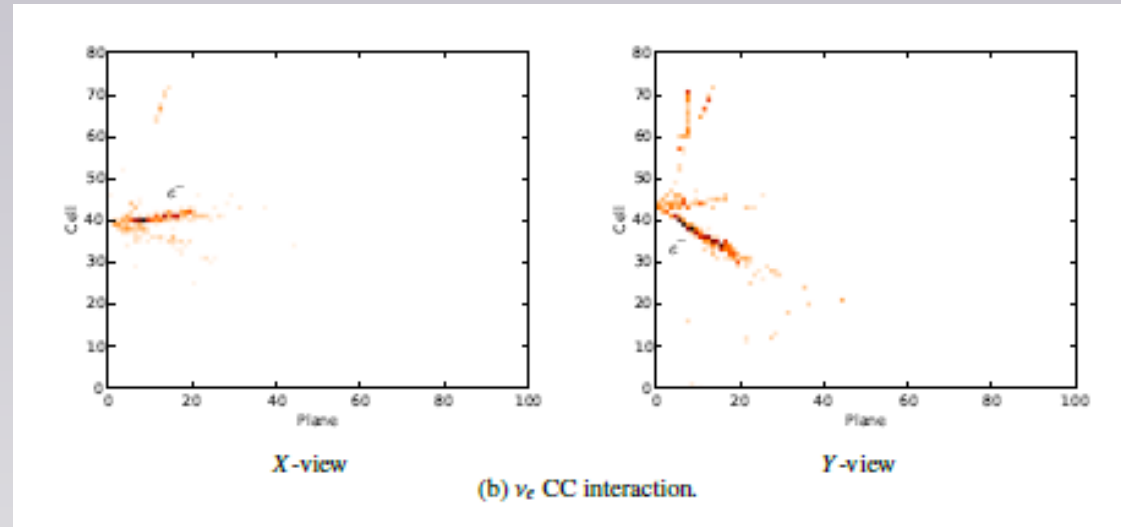
- The model used: Network in a Network instead of simple convolution filters



- Google's winning entry to the ImageNet 2014 competition

# example: NOvA

feed 2 projections of 3D image into 'siamese' GoogLeNet  
type of network

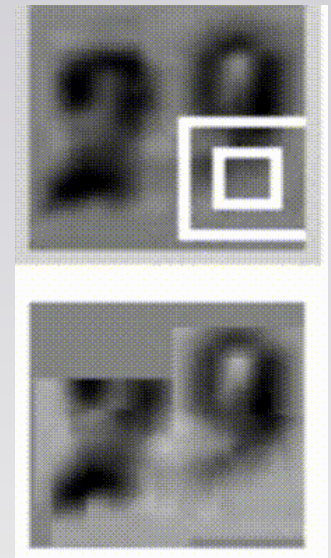
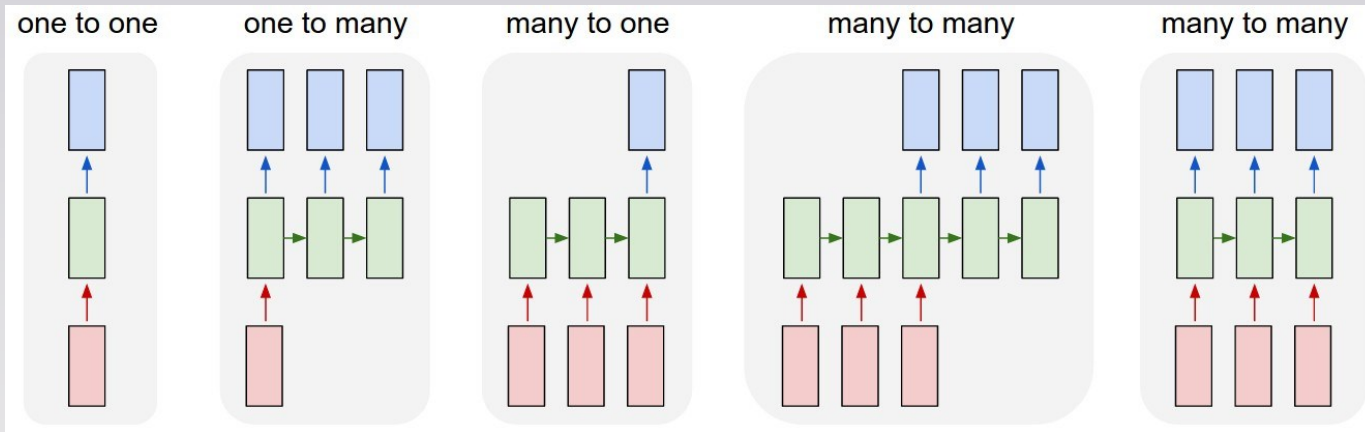


$\nu_e$  -CC  $\text{eff}_{\text{rec}}$ : 35%  $\rightarrow$  49%  
efficiency increase  
(same background)

arXiv:1604.01444

# More Exotic Stuff .. e.g. LSTMs (LongSortTermMemory recurrent networks)

- recurrent networks → loops → input of (time) sequences
  - natural language processing
    - image segmentation (captioning/digits→numbers)
    - distinguish up/down going muons in NOvA



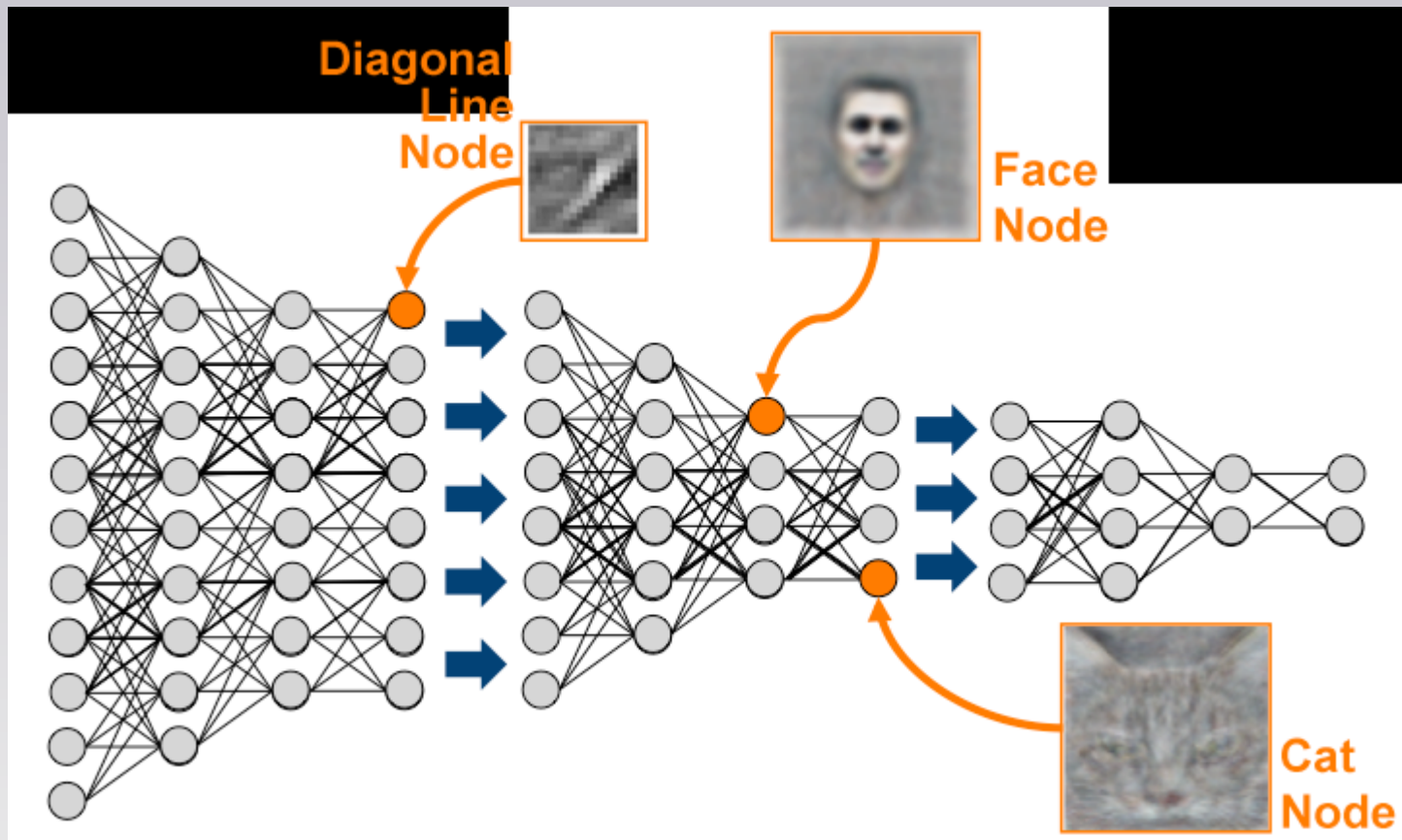


# Summary

- massive improvements Neural Networks done recently
- evaluate and “re-think” how to do
  - Event reconstruction
  - Event classification
  - Computing?  
Optimize data storage according to popularity  
([http://pos.sissa.it/archive/conferences/239/008/ISGC2015\\_008.pdf](http://pos.sissa.it/archive/conferences/239/008/ISGC2015_008.pdf))
  - Trigger using “Light/Fast algorithms with ‘dark knowledge’ ?
- play with neural networks in your browser!
  - <http://playground.tensorflow.org/>

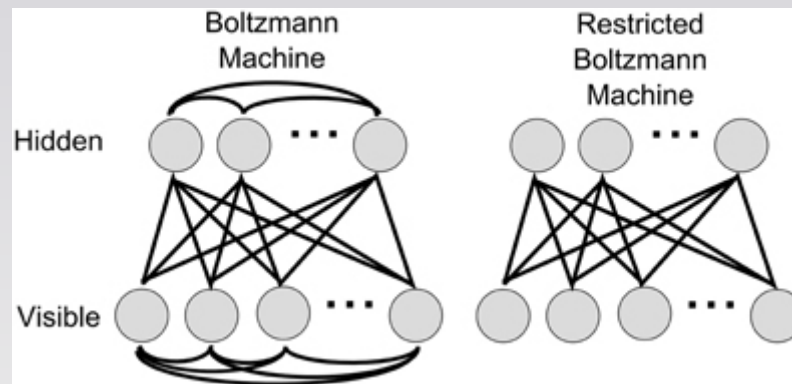
# Finding Cats

- 10 million random Youtube screenshots
- huge Neural Network → reconstruct input (auto-encoder)

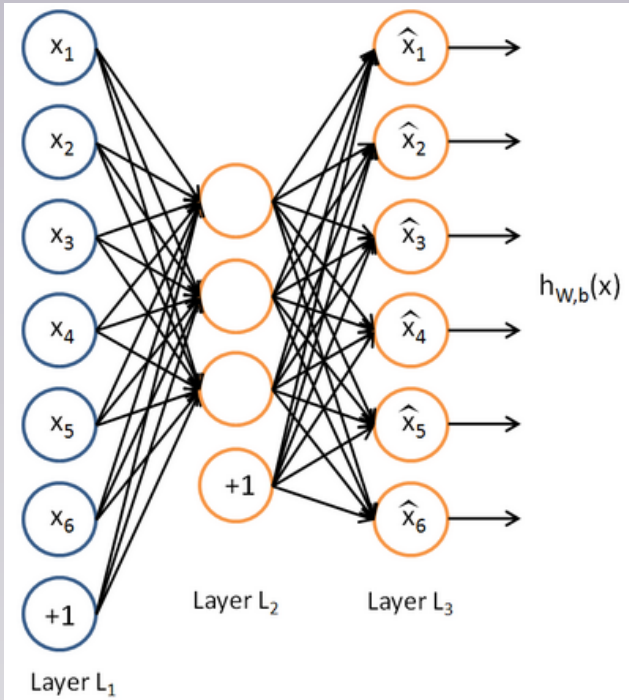


# Training deep networks

- The new trick is: pre-training + final backpropagation to “fine-tune”
  - initialize the weights not ‘random’ but ‘sensibly’ by
  - ‘unsupervised training of’ each individual layer, one at the time, as an:
    - : auto-encoder (definite patterns)
    - : restricted-Boltzmann-machine (probabilistic patterns)



# Auto-Encoder



- network that ‘reproduces’ its input
  - hidden layer  $<$  input layer
  - hidden layer ‘dimensionality reduction’
- needs to ‘focus/learn’ the important features that make up the input