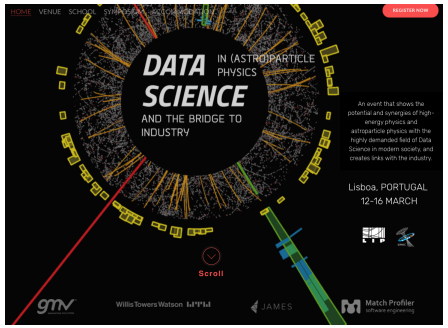
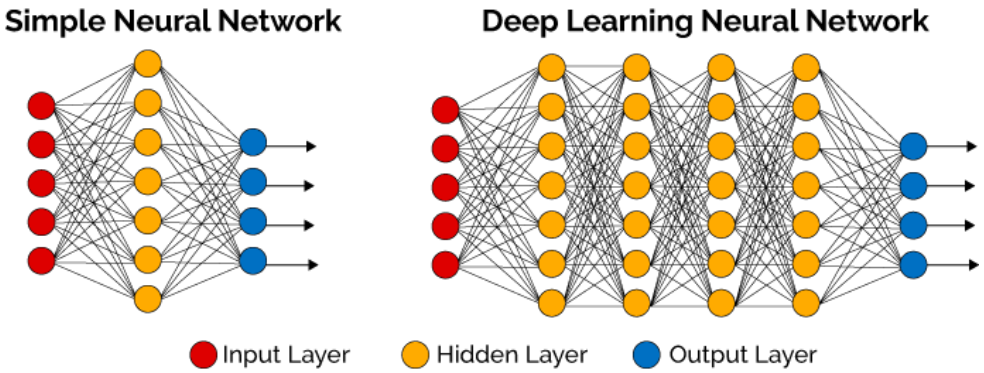



# Machine Learning (Lecture 3)

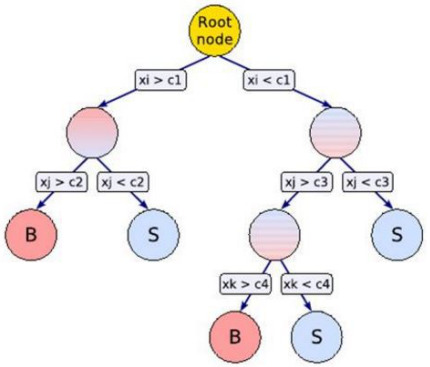



*Lorenzo Moneta*
*CERN - EP-SFT*
Lorenzo.Moneta@cern.ch

LIP Data Science School / 12-14 March 2018

# Outline (yesterday)

- Lecture 1 (Monday)
  - Introduction to Machine Learning
  - Linear Models for Classification and Regression
- Lecture 2 (yesterday)
  - Fischer discriminant
  - Support Vector Machine (SVM)
  - Decision Trees
  - Unsupervised Learning: PCA, clustering

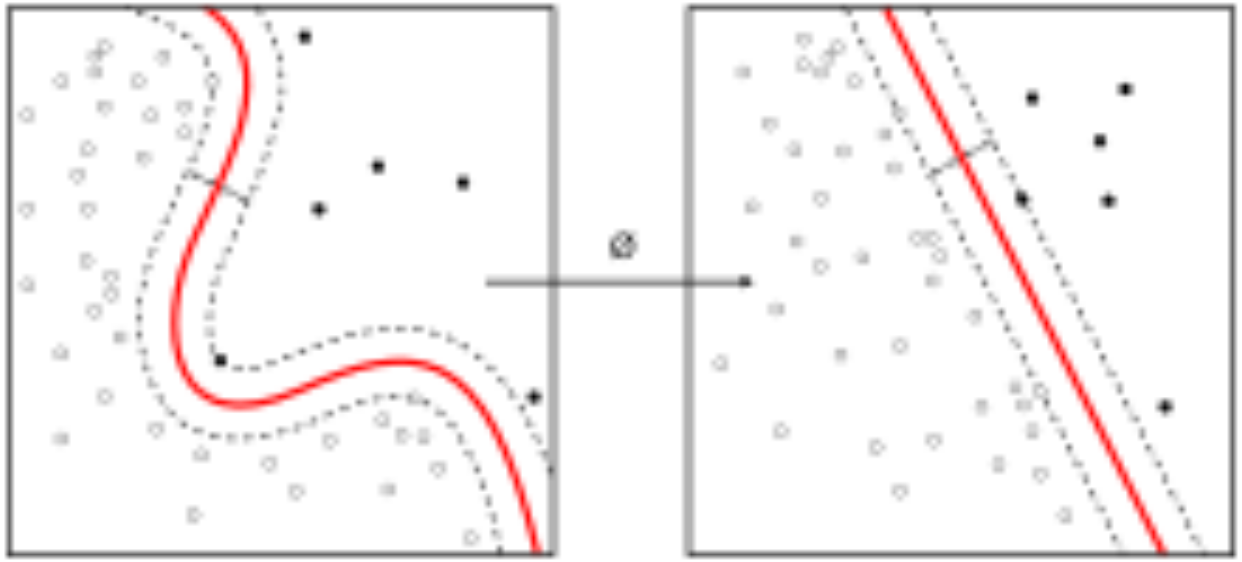


# Support Vector Machine

- Find decision boundary maximising the margins (distance to the closest points)
- Optimization problem

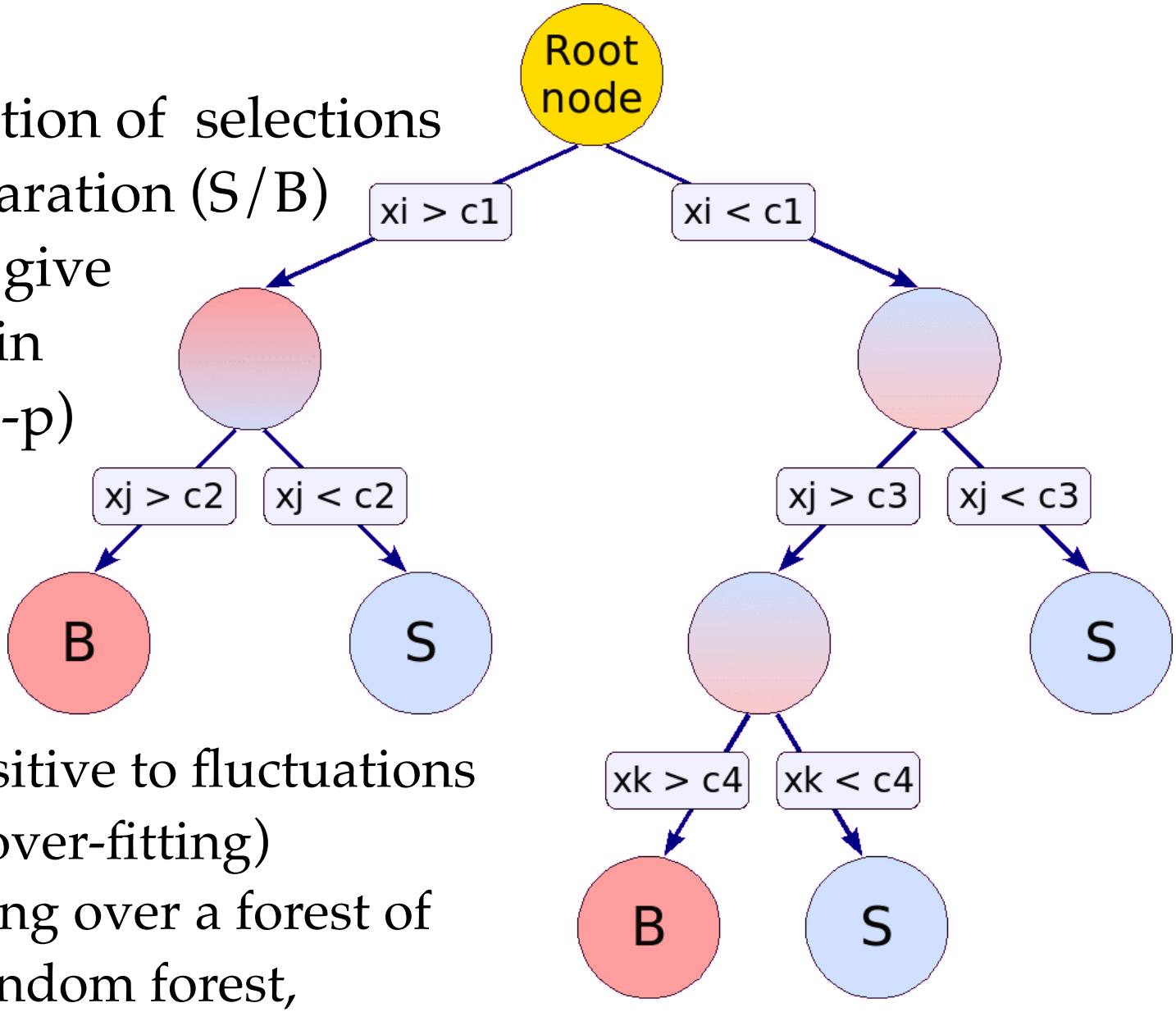
$$L(\mathbf{w}) = C \sum_i \max(0, 1 - y_i(\mathbf{w}^T \mathbf{x}_i + w_0)) + \frac{1}{2} \sum_i w_i^2$$

Using Kernels
to construct non-linear
decision boundaries



# Decision Trees

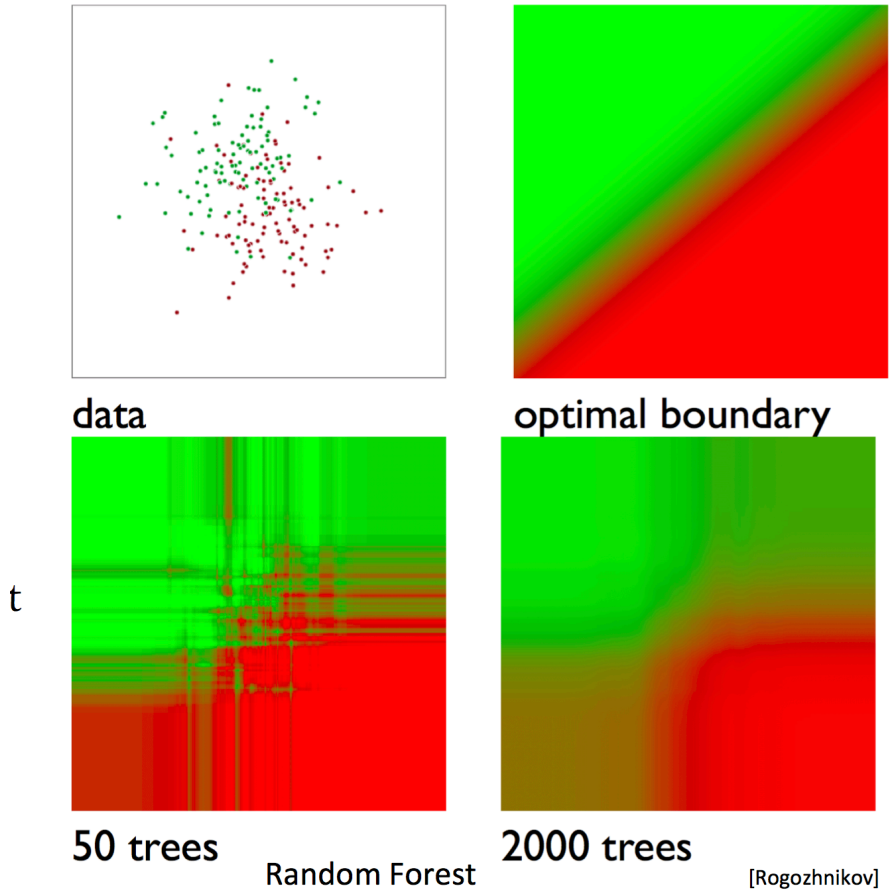
- Sequential application of selections that maximise separation (S/B)
- Use variables that give best separation gain e.g. Gini index p(1-p)



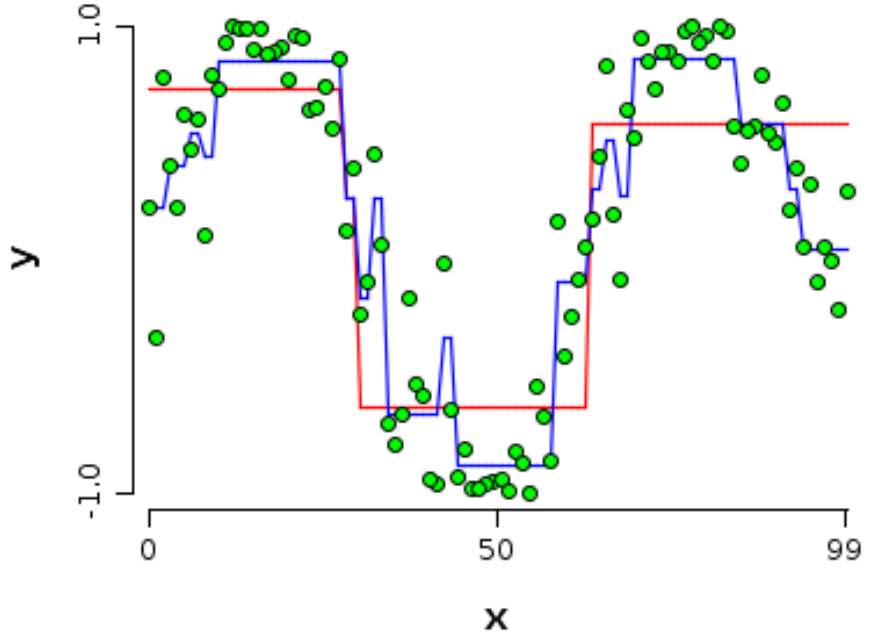

- **Problem**: too sensitive to fluctuations of training data (over-fitting)
- **Solution**: averaging over a forest of trees (bagging, random forest, boosting)

# Decision Trees

Can be used for classifications

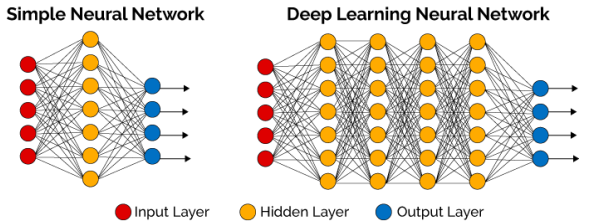



and regression



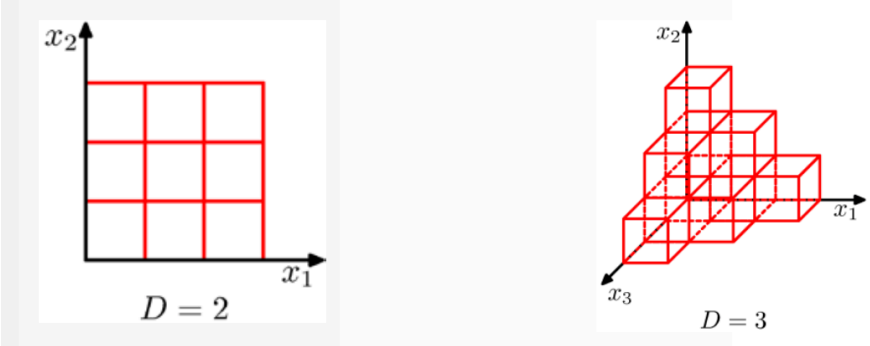
# Outline for today

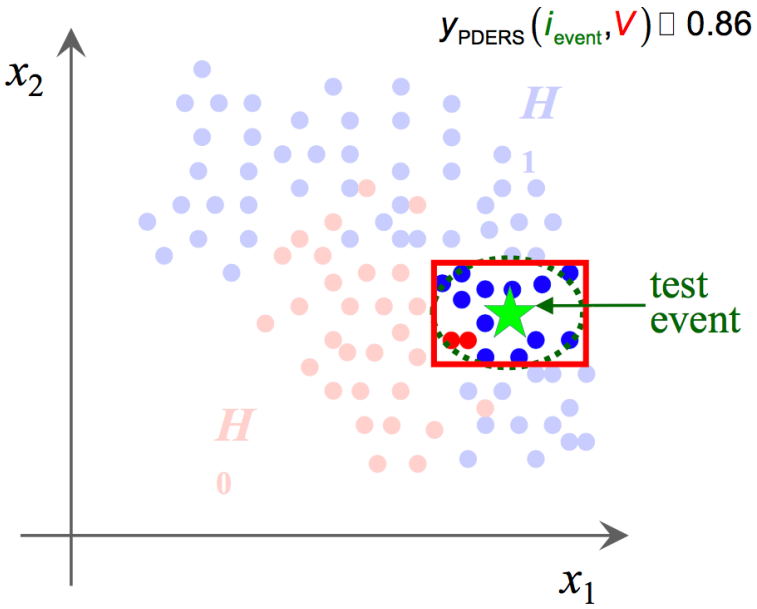
- Lecture 3
  - Introduction to Neural Networks
  - **Deep Learning**
    - optimisation algorithms
    - Convolutional networks (CNN)
    - Recurrent networks (RNN)
    - Generative adversarial networks (GAN)
  - Conclusions
  - Example of using a DNN in TMVA



# Multidimensional PDE

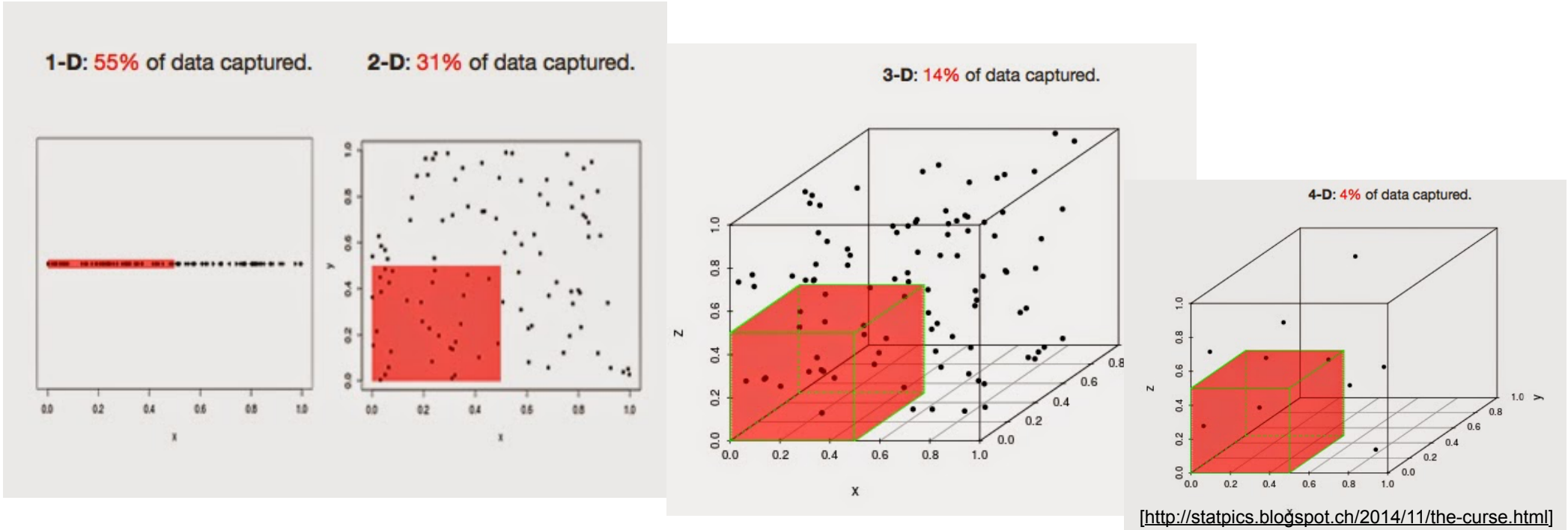
- Estimate the multidimensional probability densities by counting the events of each class in a predefined or adaptive volume
- **Advantages**:
  - All correlation taken into account
- **Disadvantages**:
  - Course of dimensionality



# Curse of Dimensionality

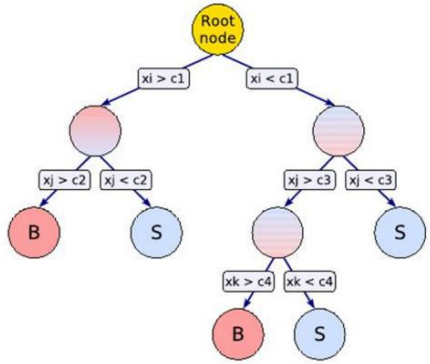
- As dimension of space grows, volume grows exponentially that available data become sparse
  - Example: generate n points $\{x_1,\ldots.x_n\}$ in [0,1]
    - How many have $x_i < 0.5$ ?



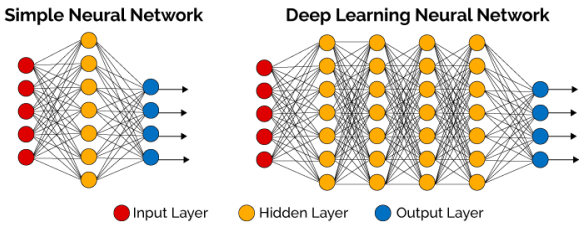

1d : 50% 2d : 25% 3d : 12.5% 4d : 6.25% n-d : $2^{-n}$

# Outline for today

- Lecture 2 (yesterday)
  - Fischer discriminant
  - Support Vector Machine (SVM)
  - Decision Trees
  - Unsupervised Learning: PCA, clustering



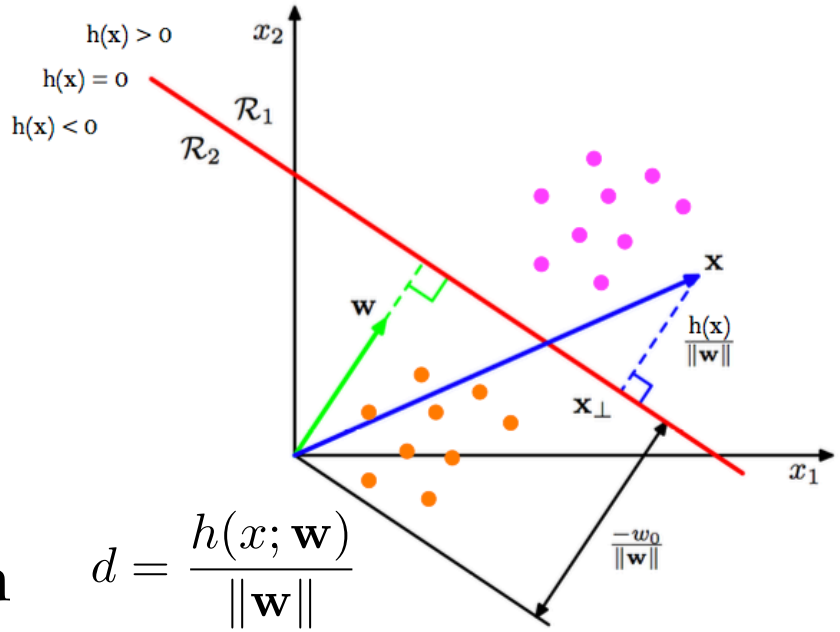

- Lecture 3 (today)
  - Introduction to Neural Networks
  - Deep Learning



# Reminder of Logistic Regression

- Linear classifier

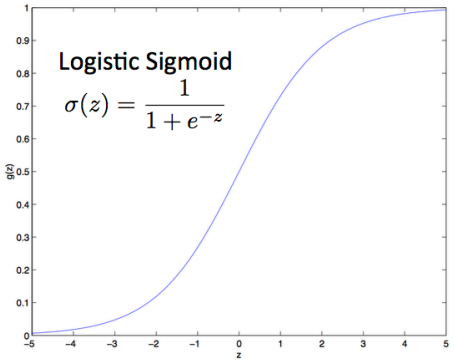
$$h(\mathbf{x};\mathbf{w}) = \mathbf{w}^T\mathbf{x}$$




$$d = \frac{h(x;\mathbf{w})}{\|\mathbf{w}\|}$$

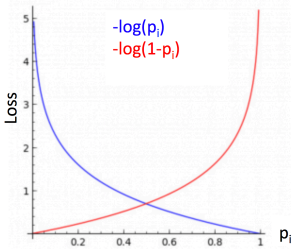
- Sigmoid function
  - convert distance to decision boundary to class probabilities

$$p(y=1|\mathbf{x}) \equiv p_i = \frac{1}{1+e^{-h(\mathbf{x};\mathbf{w})}} = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$$



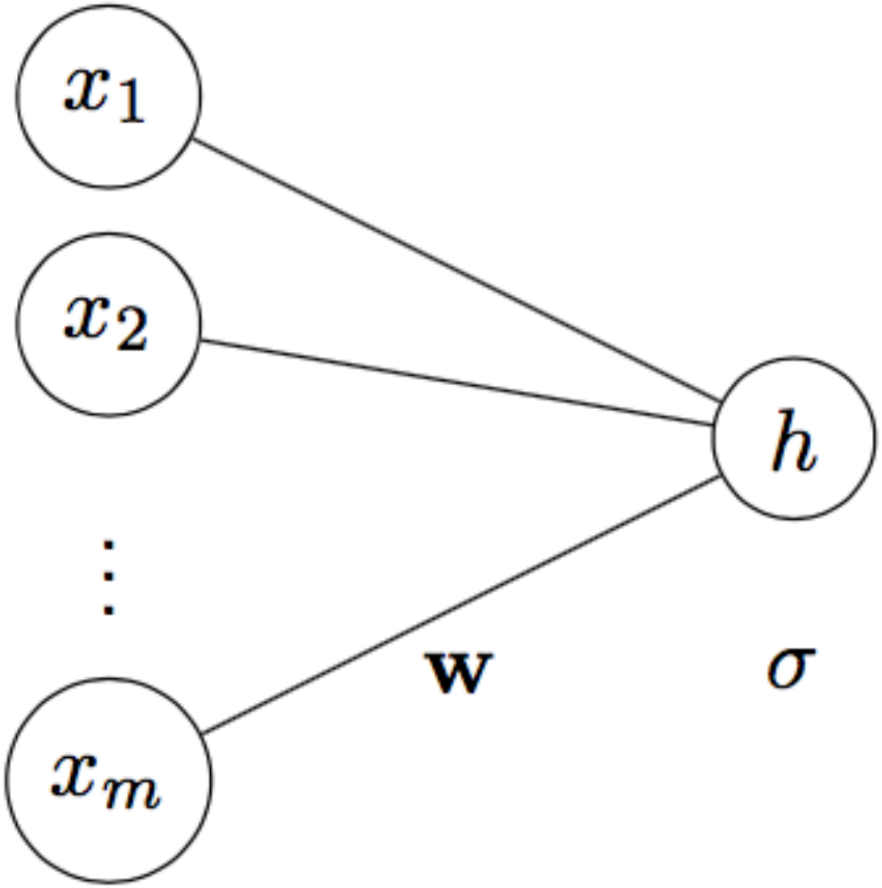

- Cross Entropy Loss Function

$$L(\mathbf{w}) = -\sum_i (y_i \ln p_i + (1-y_i)\ln(1-p_i))$$

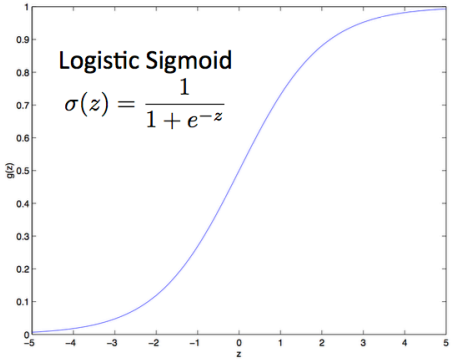


# Logistic Regression : Graphical Representation

input




output




$$p(y=1|\mathbf{x}) = \sigma(h(\mathbf{x},\mathbf{w}))$$
$$= \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$$

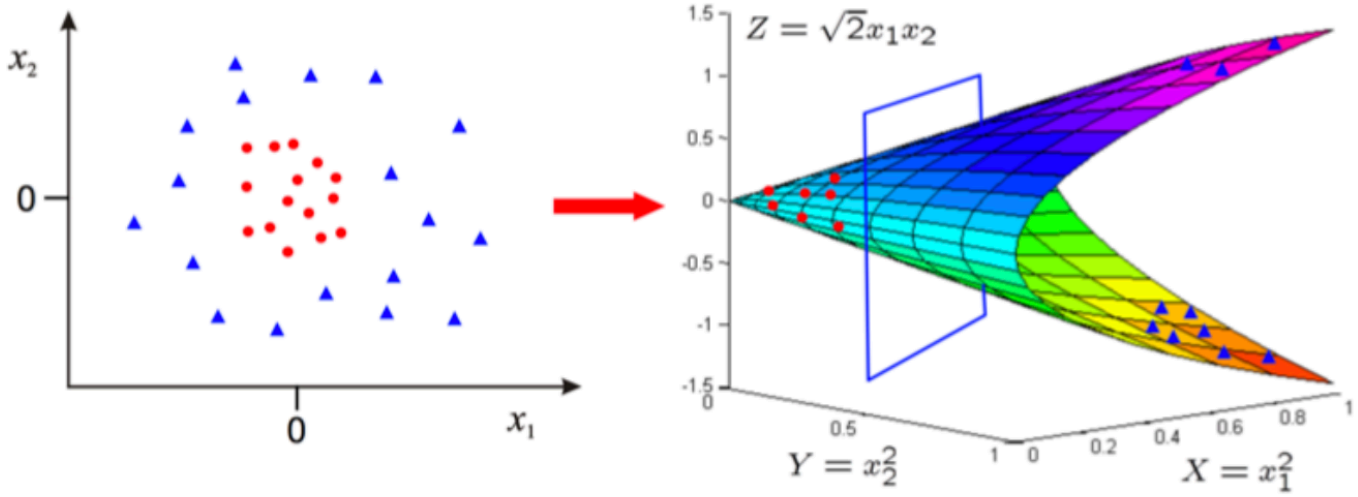
# Adding Non Linearity

- Extend by adding non linearity to decision boundary

$$h(\mathbf{x};\mathbf{w}) = \mathbf{w}^T\mathbf{x} \rightarrow \mathbf{w}^T\Phi(\mathbf{x})$$

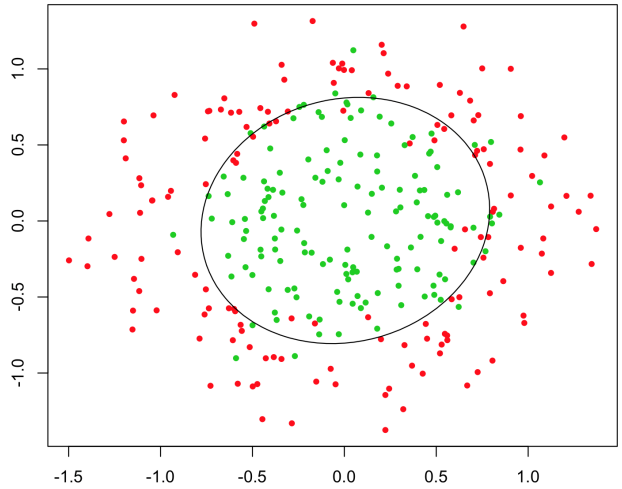
$$\text{where} \quad \Phi(\mathbf{x}) \sim \{\mathbf{x}^2, \sin(\mathbf{x}), \log(\mathbf{x}), ....\}$$

$$\Phi : \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \begin{pmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \end{pmatrix} \quad \mathbb{R}^2 \rightarrow \mathbb{R}^3$$



# Non Linearity

- Problem: how to choose function $\phi(x)$ for the non linear $\mathbf{R}^m \rightarrow \mathbf{R}^d$ transformation ?
- Solution: Learn function directly from the data
  - parametrize $\phi(x) = \phi(x;\mathbf{u})$
  - $\mathbf{u}$ is a set of parameters which will be learned from the data



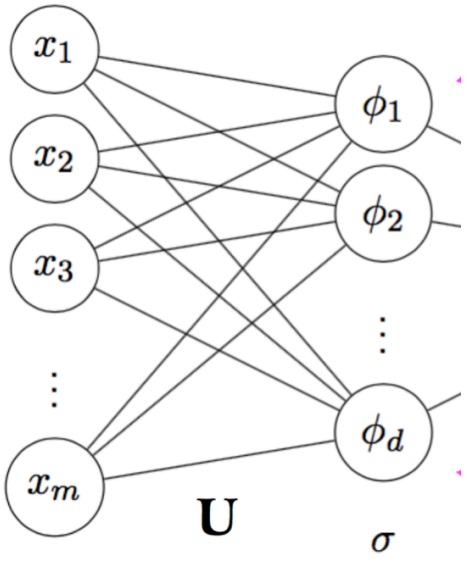
# Neural Networks

- Given input **x** , vector of size *m*
- Define *d* basis functions $\phi_j(x;\mathbf{u})$ with j = {1,…d}for transforming $\mathbf{R}^m \rightarrow \mathbf{R}^d$

$$\phi_j(\mathbf{x};\mathbf{u}) = \sigma(\mathbf{u}_j^T\mathbf{x})$$

- The parameter vectors (ot dim *m)* $\mathbf{u}_j$ can be represented as a matrix **U** with dimension *d* x *m*

$$\phi(\mathbf{x};\mathbf{U}) = \sigma(\mathbf{U}\mathbf{x}) = \begin{bmatrix} \sigma(\mathbf{u}_1^T\mathbf{x}) \\ \sigma(\mathbf{u}_2^T\mathbf{x}) \\ \dots \\ \sigma(\mathbf{u}_d^T\mathbf{x}) \end{bmatrix} \in \mathbb{R}^d$$

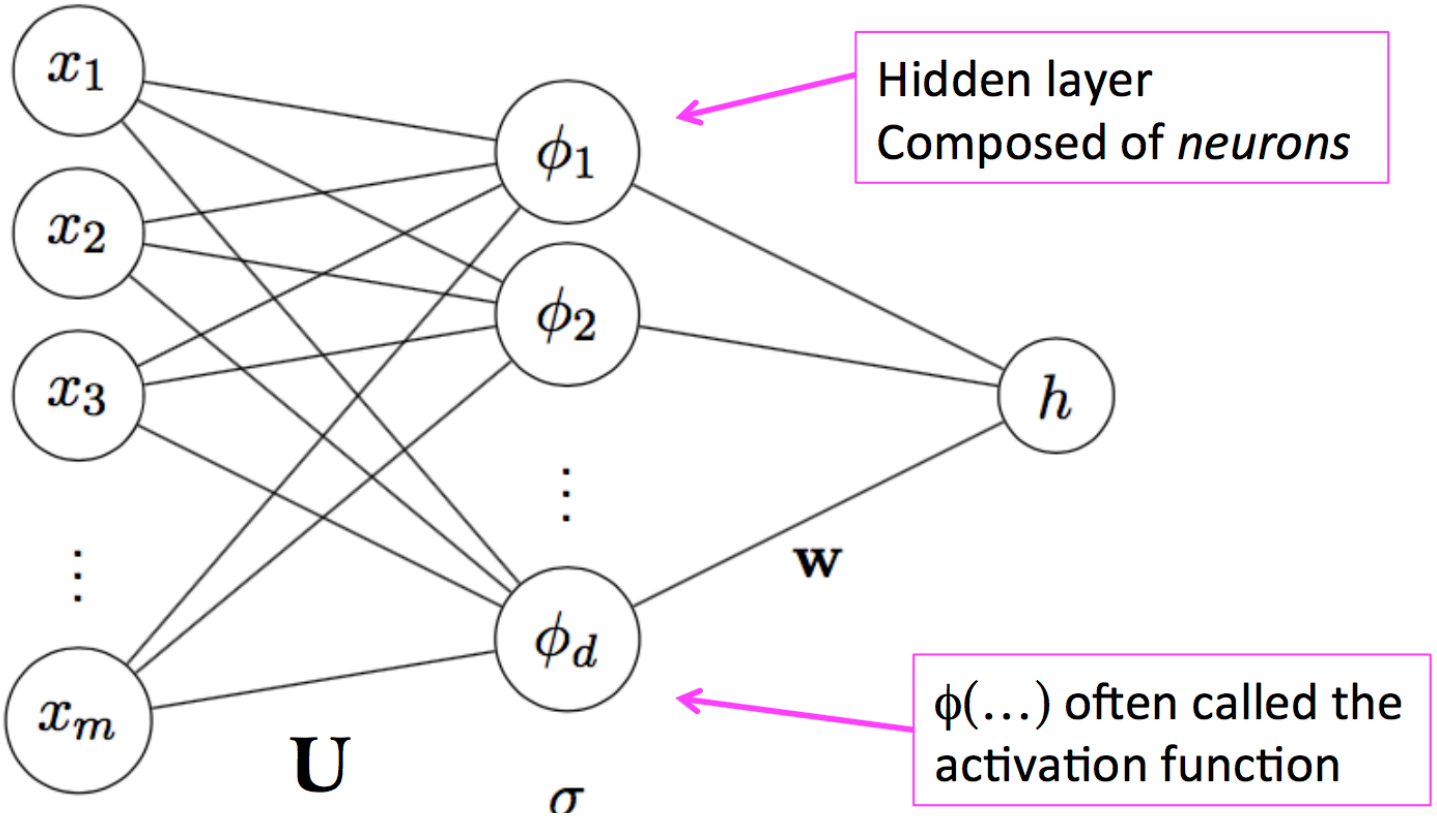



- **σ** is for example the sigmoid activation function
  - one could use also other functions (e.g. tanh, RELU)

# Feed Forward Neural Network

Full model becomes

$$h(\mathbf{x}; \mathbf{w}, \mathbf{U}) = \mathbf{w}^T \phi(\mathbf{x}; \mathbf{U})$$

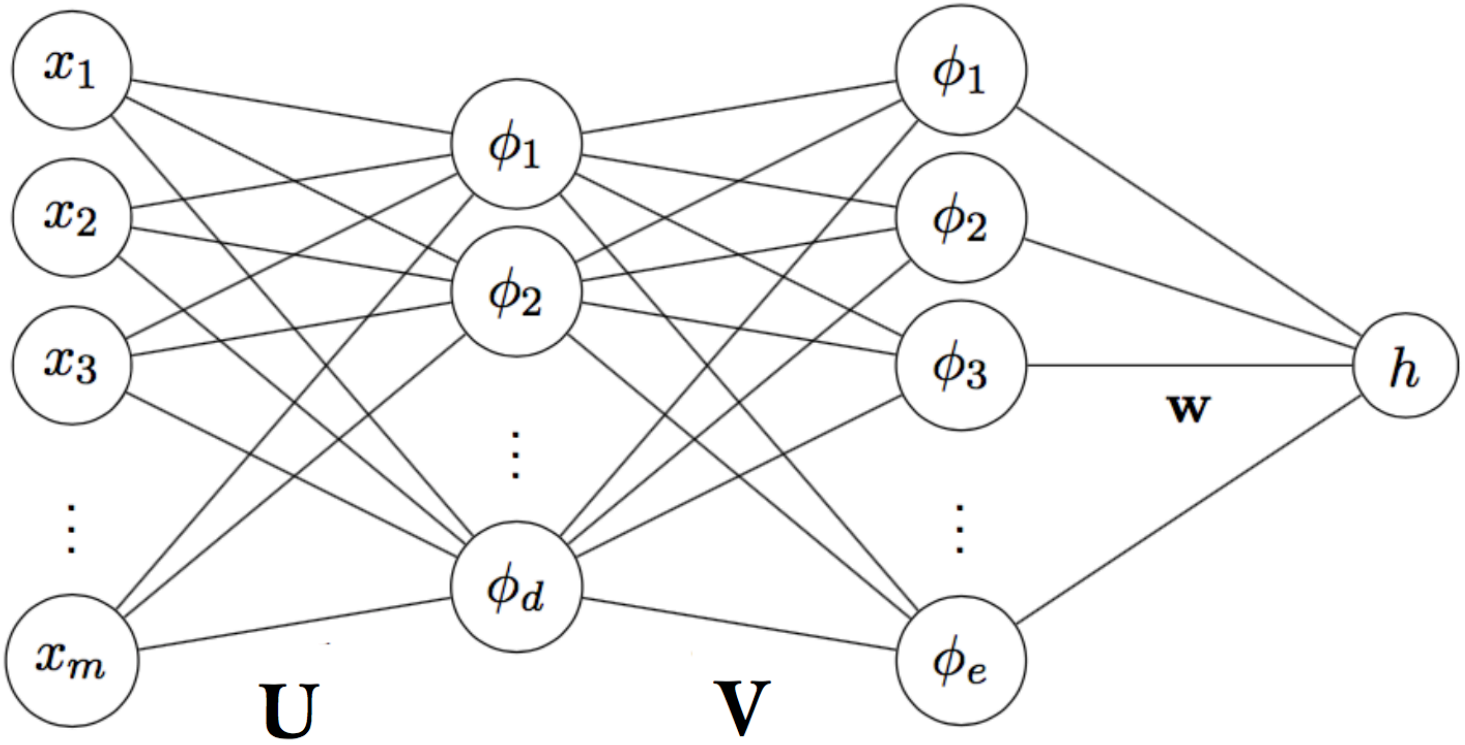



$$\phi(\mathbf{x}; \mathbf{U}) = \sigma(\mathbf{U}\mathbf{x})$$

# Multi Layer Neural Network

Model can be extended to multi-layers

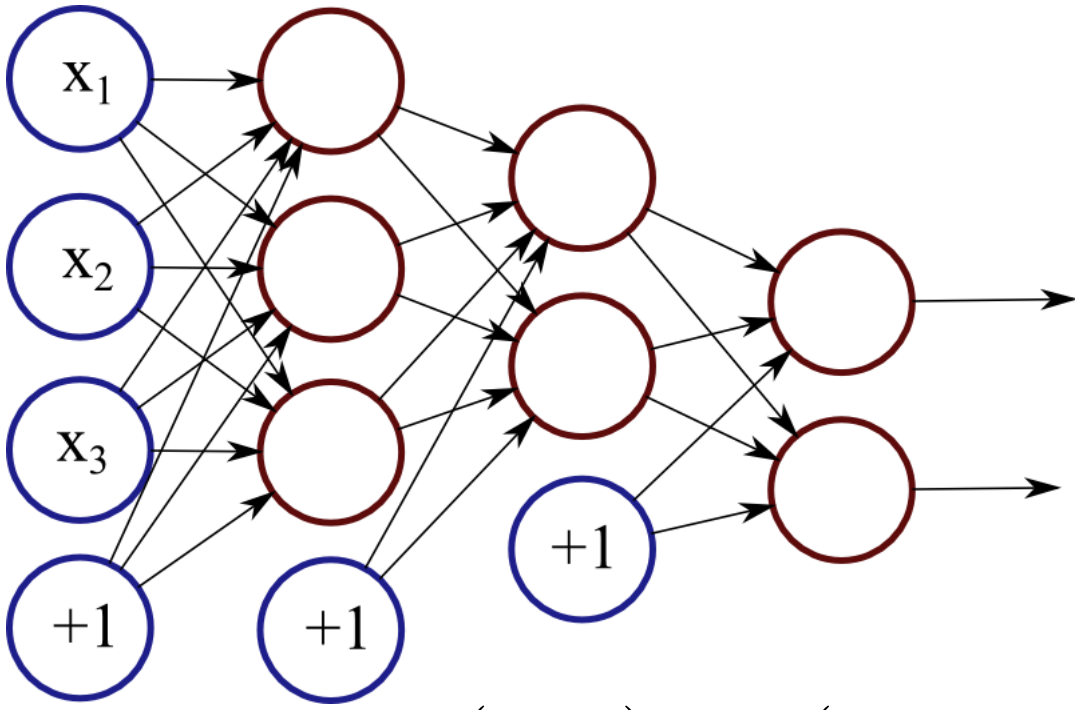
$$h(\mathbf{x}; \mathbf{w}, \mathbf{U}, \mathbf{V}) = \mathbf{w}^T \sigma(\mathbf{V}\sigma(\mathbf{U}\mathbf{x}))$$




Additional parameters matrix **V** with dimension *e* x *d*

# Bias Vector

Also a bias node (a vector) in addition to the weight matrices




$$\Phi(\mathbf{x};\mathbf{u}) = \sigma(\mathbf{U}\mathbf{x} + \mathbf{b})$$

Bias can be absorbed in weight parameters by assuming an additional input variable $x_0$ always equal to 1

# Function Approximation Theorem

- A feed-forward neural network with a single hidden layer with a finite number $N$ of neutrons can approximate any continuous function f($\mathbf{x}$) in $\mathbf{R}^m$
  - only mild assumptions on the non-linear activation function (e.g. works with a sigmoid, but also with others)
- But theorem does not tell anything about the parameters and how many are needed
- How do we find the optimal network parameters ?

# Neural Network Training

- Neural network model

$$h(\mathbf{x}; \mathbf{w}, \mathbf{U_1}, ...\mathbf{U_n}) = \mathbf{w}^T \Phi_n(\mathbf{U_n}\Phi_{n-1}(.....\Phi_1(\mathbf{U_1}\mathbf{x})))$$

- Build loss function
  - Binary Classification: Cross Entropy loss

$$L(\mathbf{w}, \mathbf{U}) = -\sum y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i)$$
$$p_i = p(y_i = 1|\mathbf{x}_i) = \sigma(h(\mathbf{x}_i))$$

  - Regression: Square Loss (using directly output)

$$L(\mathbf{w}, \mathbf{U}) = \frac{1}{2}\sum_i (y_i - h(\mathbf{x}_i))^2$$

- Minimize the loss function with respect to the parameters $\mathbf{w}$ and $\mathbf{U}$

# BackPropagation

- Make use of chain rule of differentiation to compute efficiently the gradient w.r.t to network parameters (weights)
- Loss function is computed from several layers

$$L(\phi^a(...\phi^1(\mathbf{x})))$$

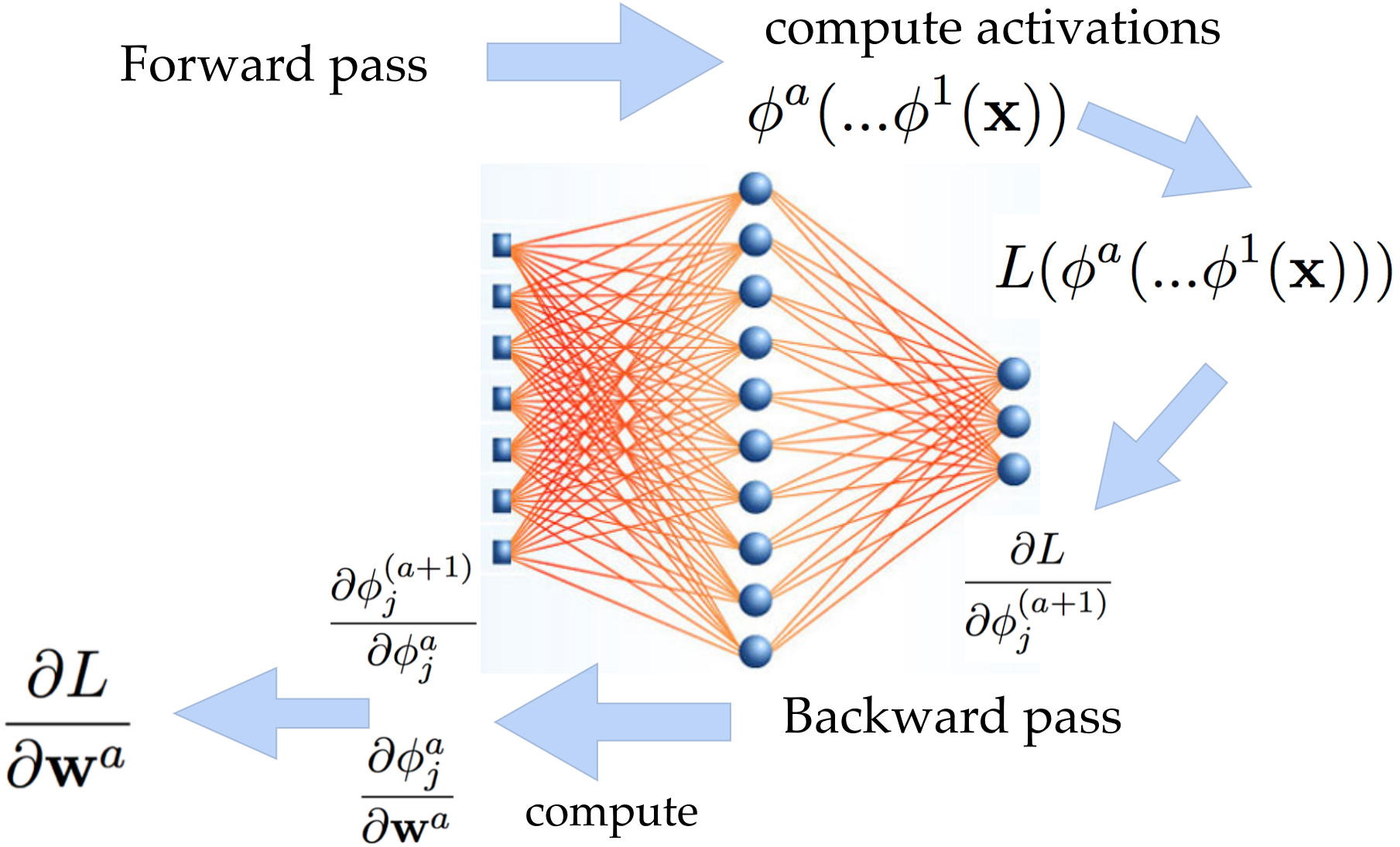
- **Forward step:**
  - compute activations at each layer $\phi^a(...\phi^1(\mathbf{x}))$
- **Backward step:**
  - compute activation gradient
  - compute weight gradient

$$\frac{\partial L}{\partial \phi^a} = \sum_j \frac{\partial \phi_j^{(a+1)}}{\partial \phi_j^a} \frac{\partial L}{\partial \phi_j^{(a+1)}}$$

- **Compute Loss Gradient**

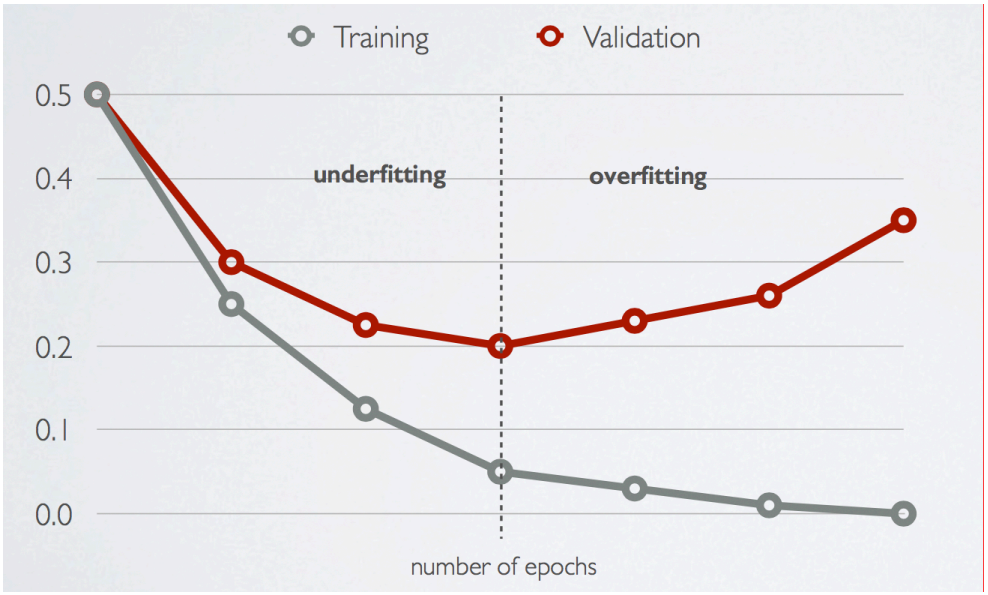
$$\frac{\partial L}{\partial \mathbf{w}^a} = \sum_j \frac{\partial \phi_j^a}{\partial \mathbf{w}^a} \frac{\partial L}{\partial \phi_j^a}$$

# BackProp


Forward pass
compute activations
$\phi^a(...\phi^1(\mathbf{x}))$
$L(\phi^a(...\phi^1(\mathbf{x})))$
$\frac{\partial L}{\partial \phi_j^{(a+1)}}$
Backward pass
$\frac{\partial \phi_j^{(a+1)}}{\partial \phi_j^a}$
$\frac{\partial L}{\partial \mathbf{w}^a}$
$\frac{\partial \phi_j^a}{\partial \mathbf{w}^a}$
compute

# Neural Network Training

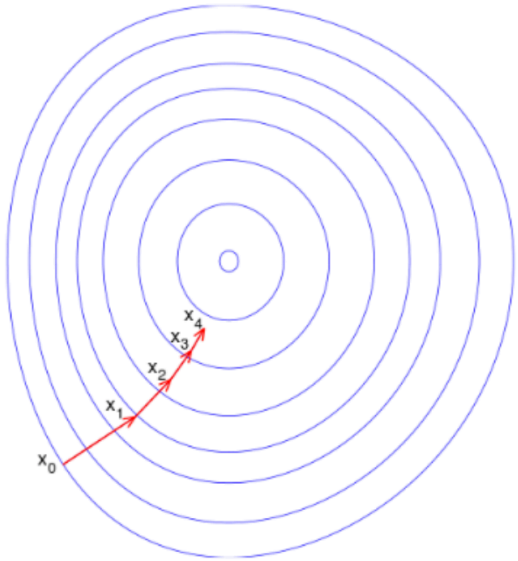
- Repeat for each parameter updates forward-backward pass to compute the loss function gradient
  - each iteration through all dataset is called epoch
- Use a validation set to examine for overtraining and to decide when stopping



# Gradient Descent

Minimize Loss function by repeated gradient steps:

- Compute gradient w.r.t. parameters: $\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}}$
- Update parameters $\mathbf{w}' \leftarrow \mathbf{w} - \eta \frac{\partial L(\mathbf{w})}{\partial \mathbf{w}}$
- $\eta$ is called the learning rate, controls how big of a gradient step to take



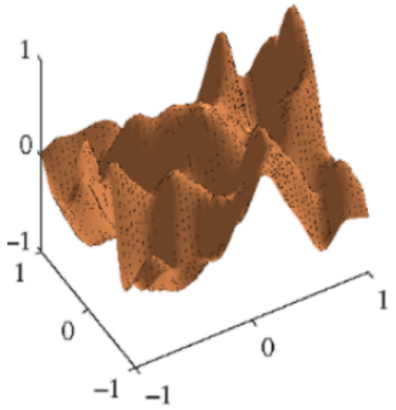

Need to compute gradients of Loss

[M. Kagan]

# Gradient Descent Problems

- Computation of gradient can be costly
  - Require passing on the full data set
  - large memory usage

$$\mathbf{w}' \leftarrow \mathbf{w} - \eta \frac{\partial L(\mathbf{w})}{\partial \mathbf{w}}$$



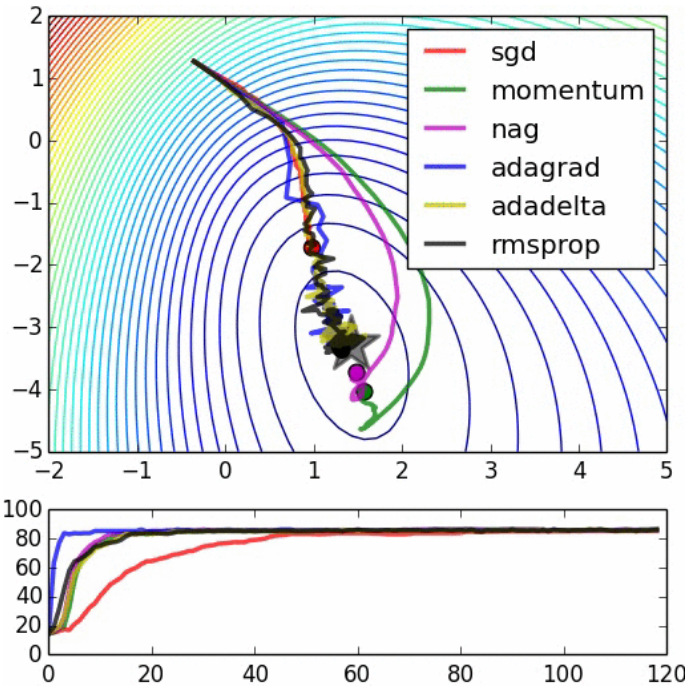

- Loss function is not convex
  - many local minima and saddle points
  - number of parameter can be very large (e.g. in complex deep neural networks)
- Standard gradient descent algorithm will not work
- Quasi-Newton methods using approximate Hessians to accelerate convergence will work even less
- **Solution**:
  - Stochastic Gradient Descent (SGD)

# Stochastic Gradient Descent (SGD)

- Solution:
  - compute the gradient on a random subset of data (mini-batch)

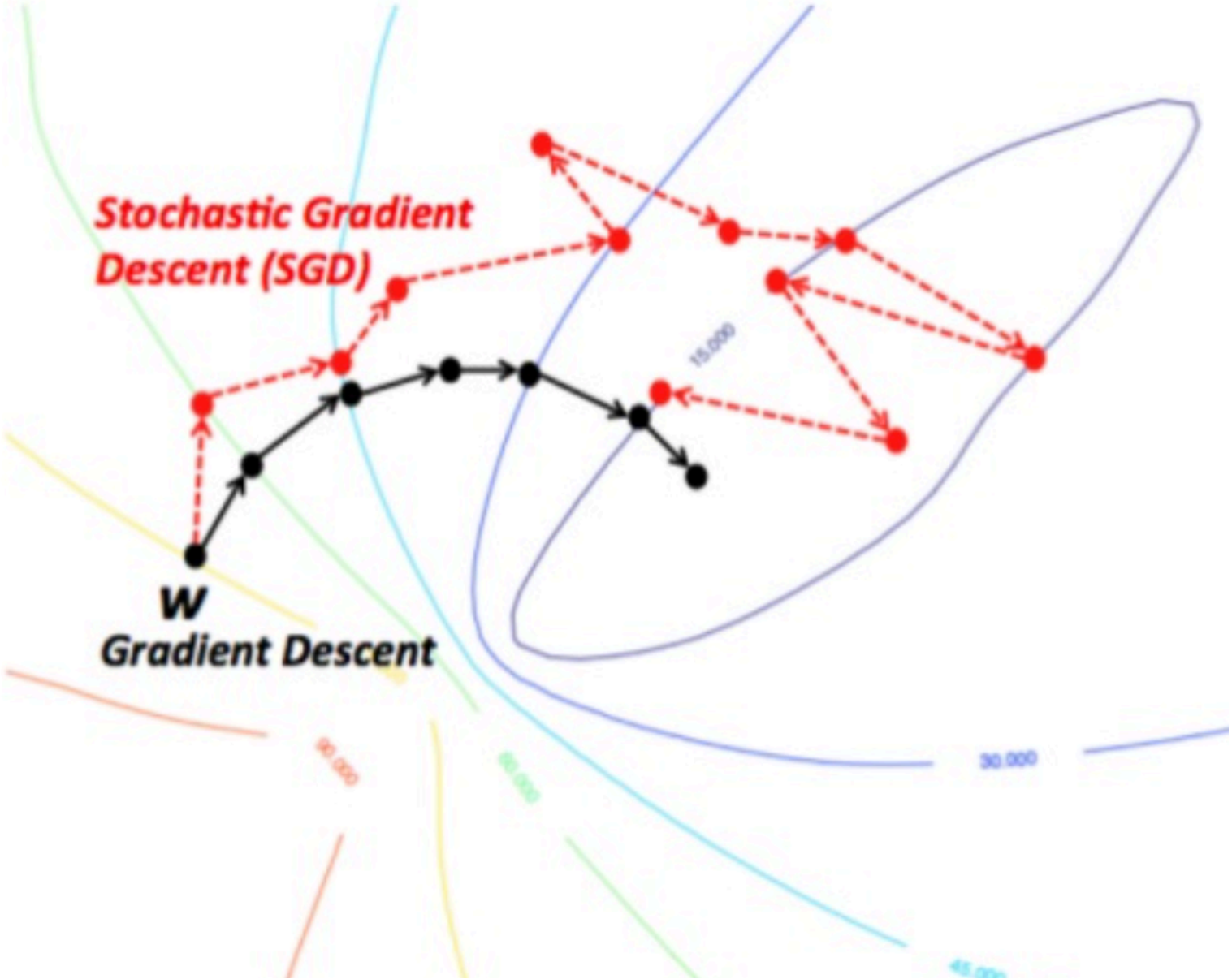
$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

    - update weights after each mini-batch computation
    - noise estimates average out
    - scale well with large data sets
    - can be helpful to jump out of local minima
    - convergence can be difficult
- Several variants exists



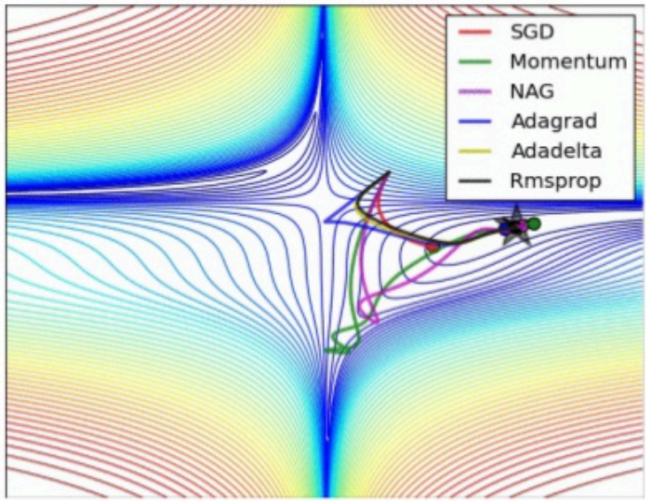

[http://danielnouri.org/notes/category/deep-learning/]

# Gradient Descent



# Variants of Gradient Descent

- Different algorithms have been developed having different rules in computing adaptive update rates
  - Momentum
  - Nesterov accelerated gradient
  - Adagrad
  - Adadelta
  - RMSprop
  - Adam
  - Adam extensions




(a) SGD optimization on loss surface contours

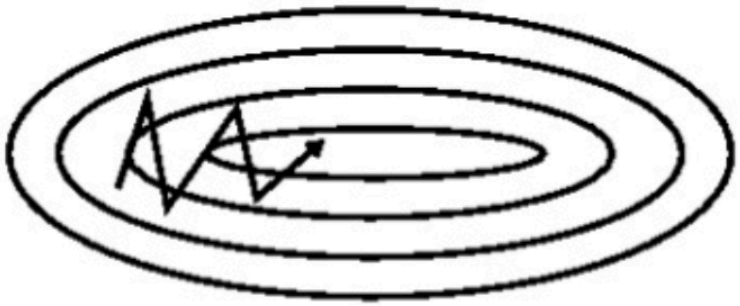
- Different performances in convergence speeds
- Largely dependent on the problem (e.g. sparsity of data)

# Gradient Descent: Momentum

- Increase updates for dimensions whose gradients point in the same directions
- Reduces updates for dimensions whose gradients change directions

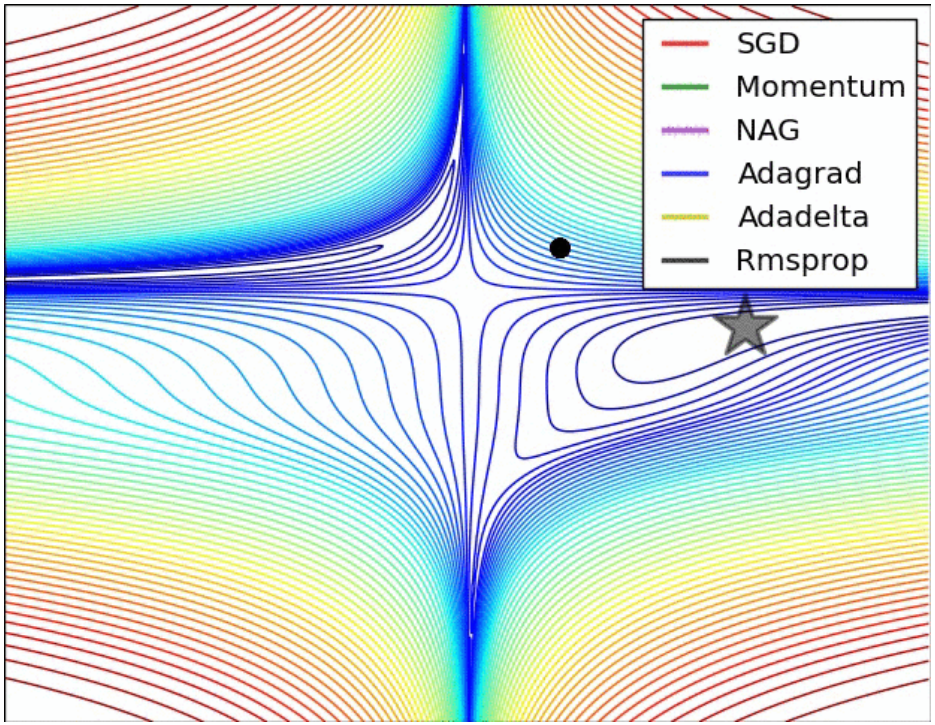
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

(a) SGD without momentum

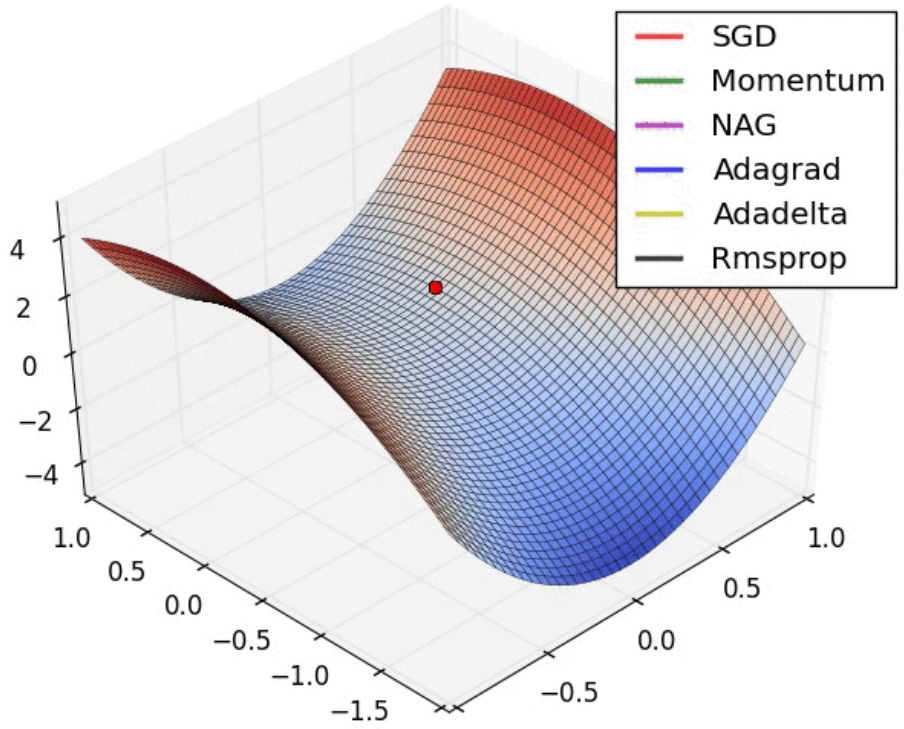


(b) SGD with momentum

# Optimizer Visual Examples




Optimizer steps on loss function contours

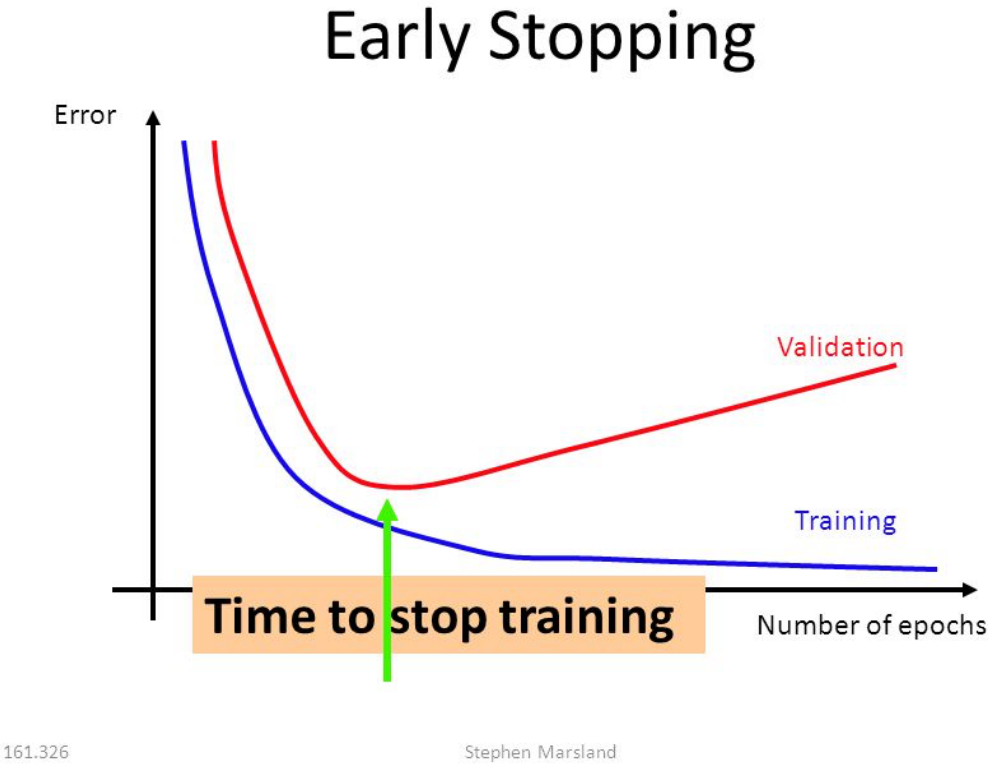



Optimization on loss function saddle points

[S. Ruder]

# Optimizer stopping rule

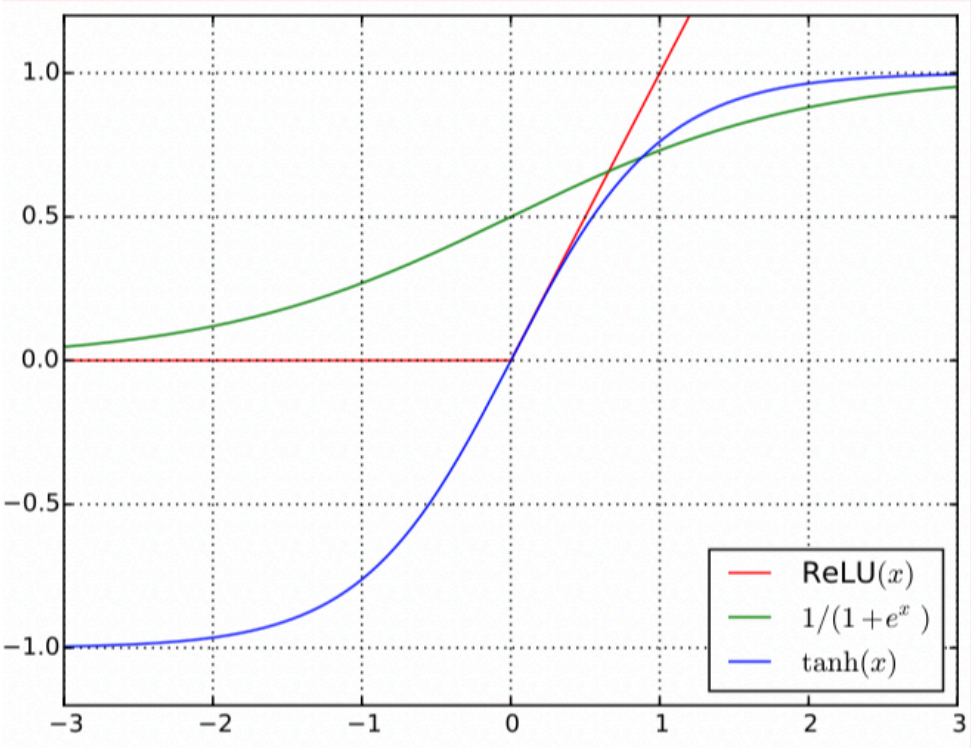
- Early stopping by monitoring validation error during training




- If validation error does not decrease after some iterations stop training

# Activation Functions

- **sigmoid**: $\sigma(x) = \frac{1}{1+e^x}$
  - derivative
    $$\sigma'(x) = \sigma(x)(1-\sigma(x))$$
  - nearly 0 when x far from 0 !
    **Vanishing gradient problem**




- **tanh**: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - similar problem with gradient
    $$\tanh'(x) = 1 - \tanh^2(x)$$

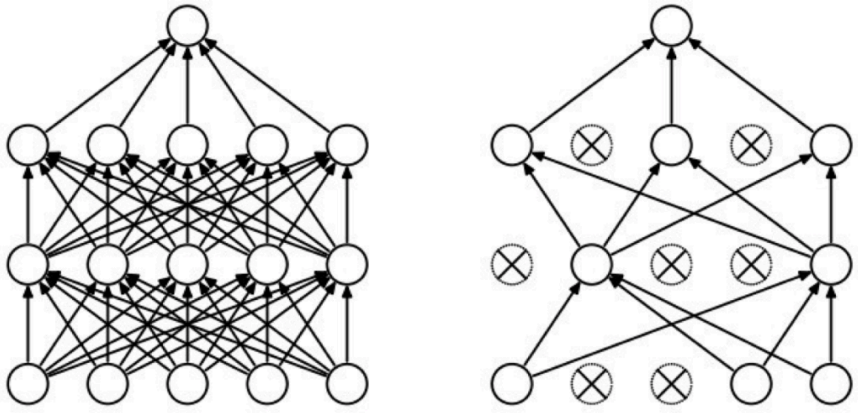
- **ReLU** : **Rectified Linear Unit**
  - ReLU(x) = max{0,x}
  - Derivative constant and not vanishing

$$\frac{\partial \mathrm{Re}LU(x)}{\partial x} = \begin{cases} 1 & when\ x > 0 \\ 0 & otherwise \end{cases}$$

# Regularization

Two possible techniques to avoid overfitting

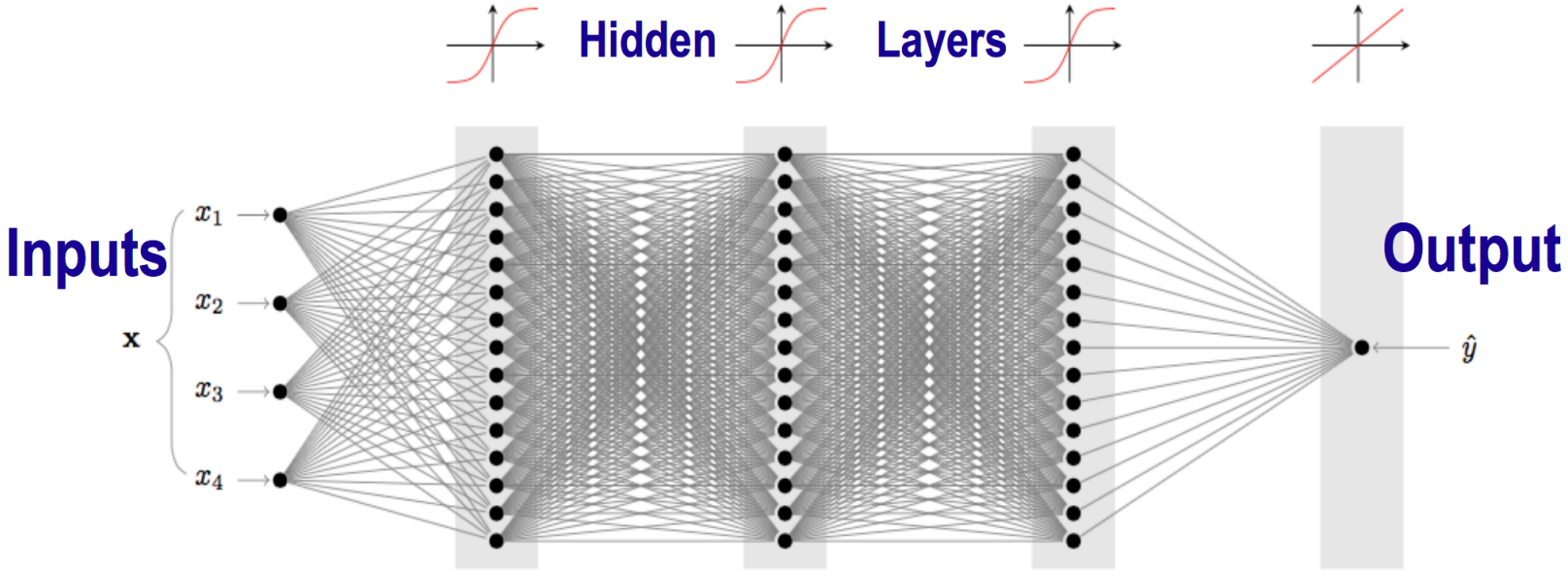
- **L2 regularisation**
  - Add to Loss function $\Omega(\mathbf{w}) = \|\mathbf{w}\|^2$
  - Avoid weights to be too large and saturate activation function. (Gaussian prior on weights)
- **Drop-Out**
  - Randomly remove nodes during training
  - Basically like an averaging model procedure
  - Found to be very effective to reduce overfitting



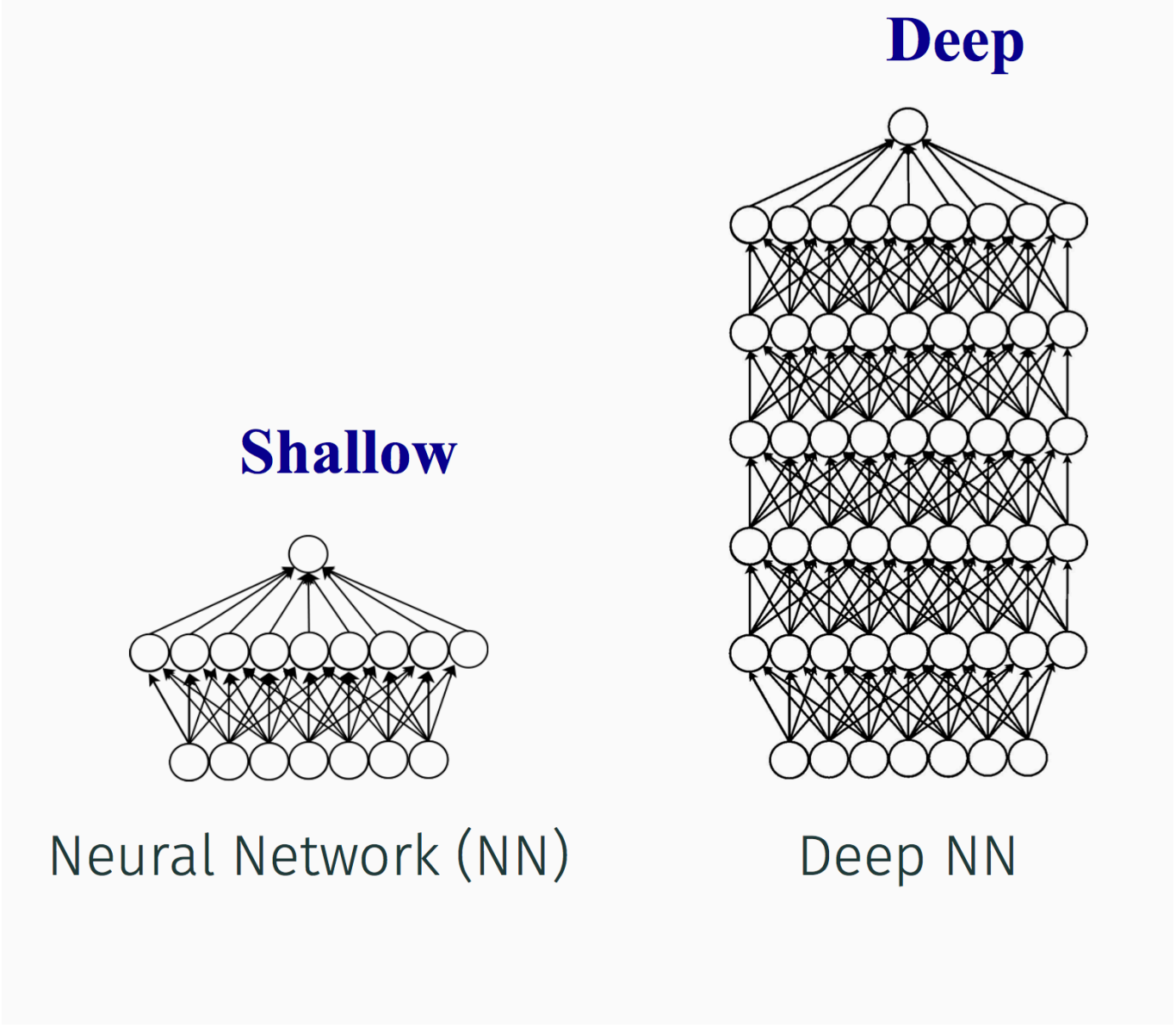
(a) Standard Neural Net (b) After applying dropout.

# Deep Learning

- Deep Neural Networks (DNN)
  - An artificial neural networks using several hidden layers
- DNN can provide significant performance improvements
- Very popular in recent years thanks to improvement in computing performances (e.g. GPU usage)

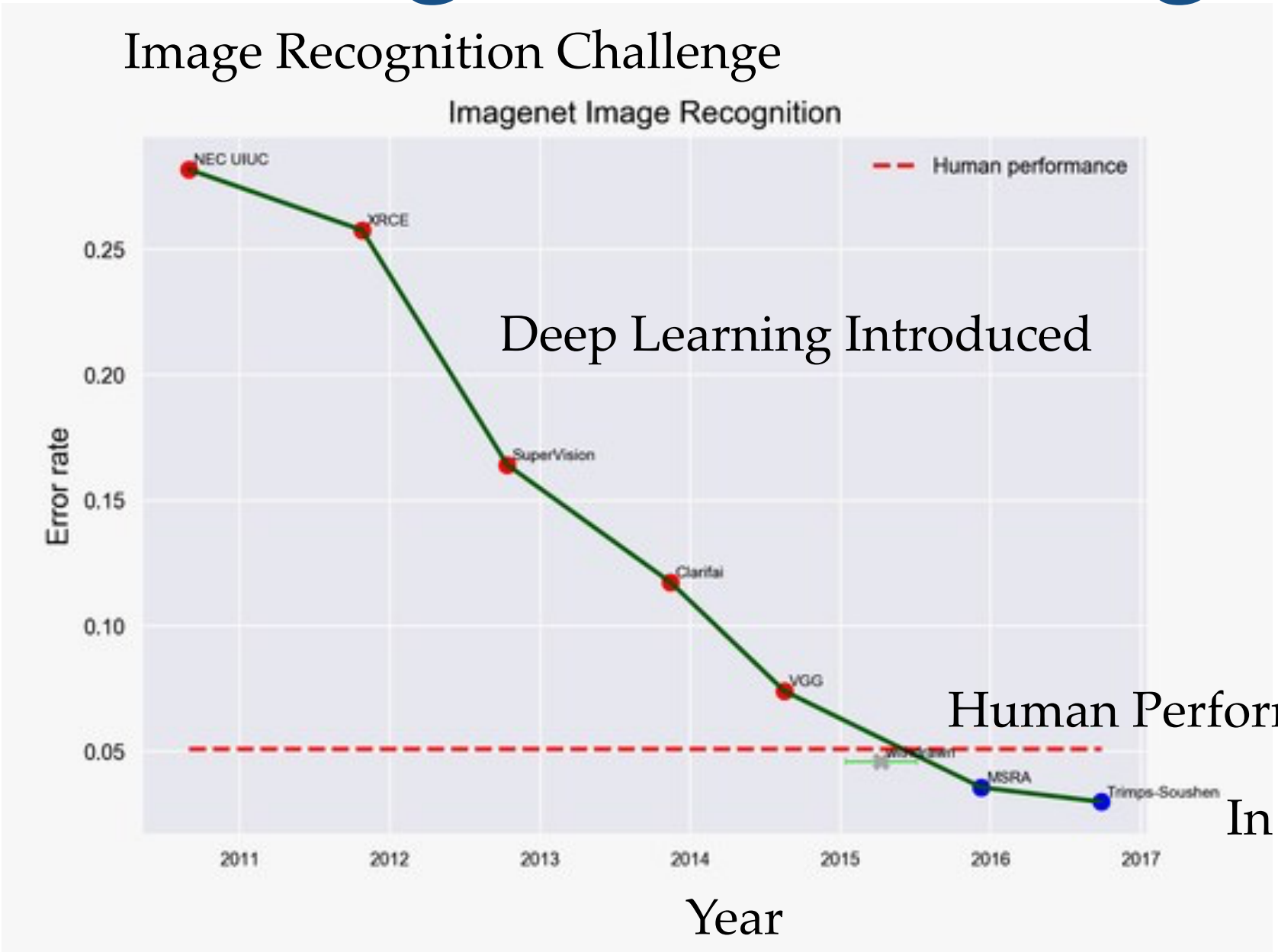


# Deep Learning




[S. Gleyzer]

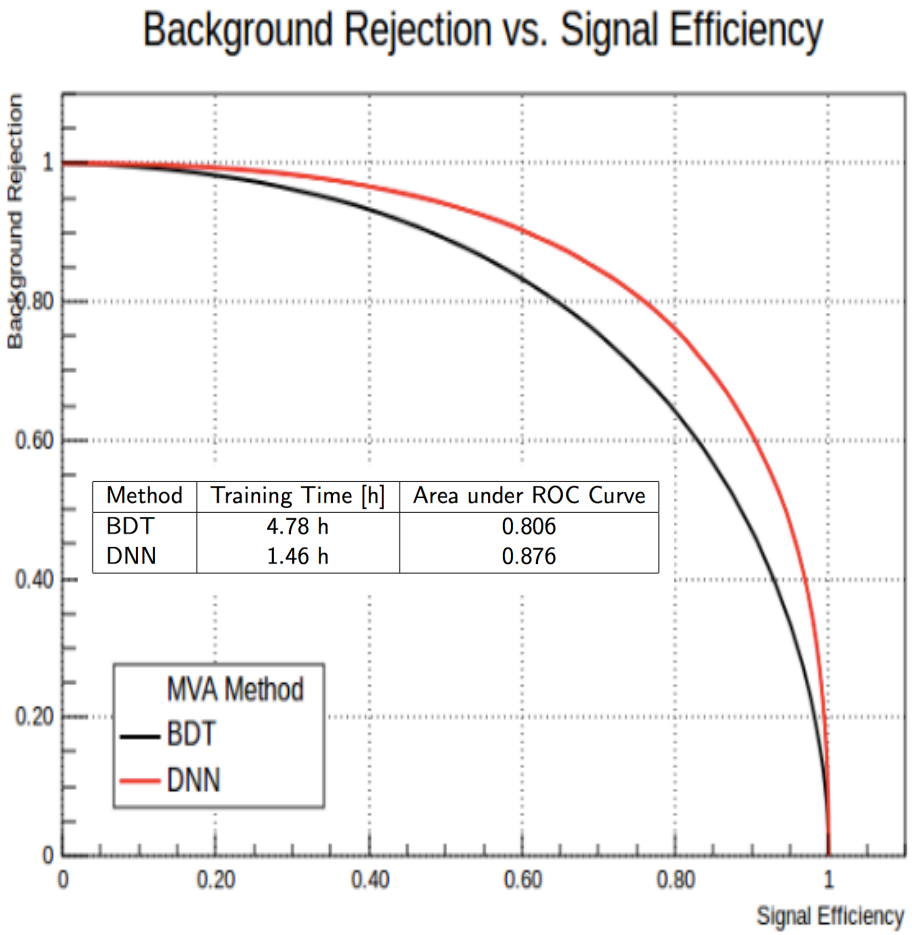
# Imagenet Challenge



# Deep Learning Performances

Example of Deep Learning for classification on HEP data set

## DNN vs BDT

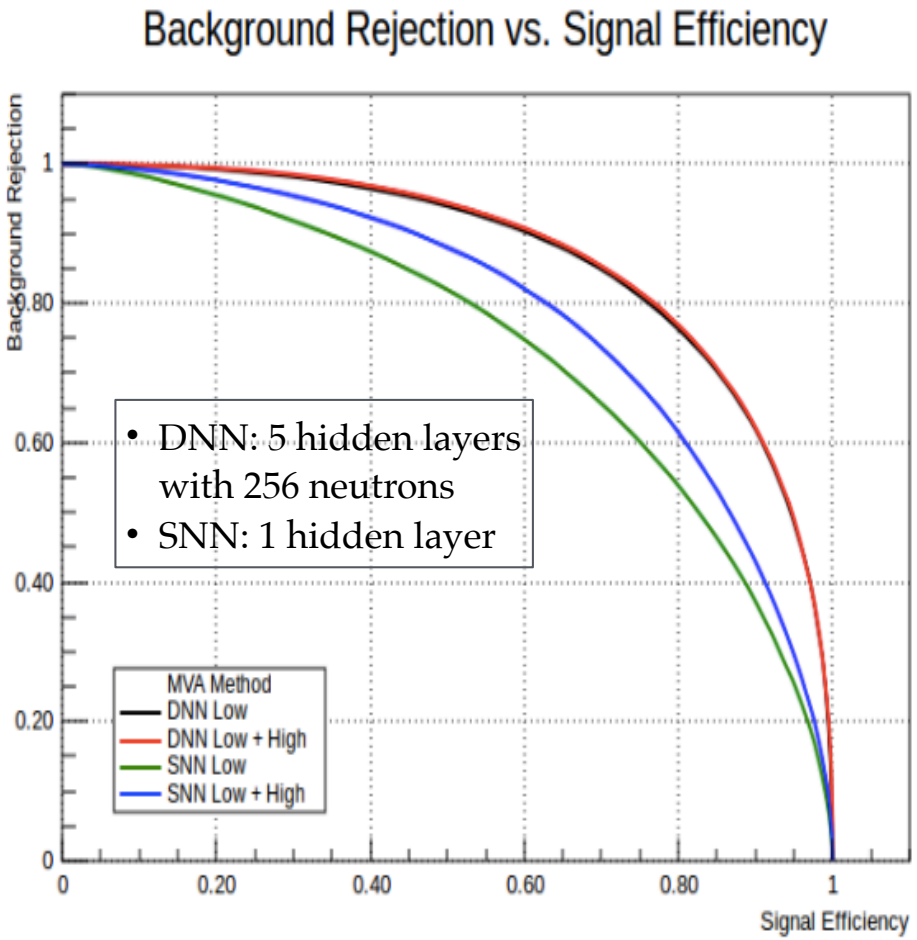



High classification performance compared to other ML methods

# Deep Learning Performances

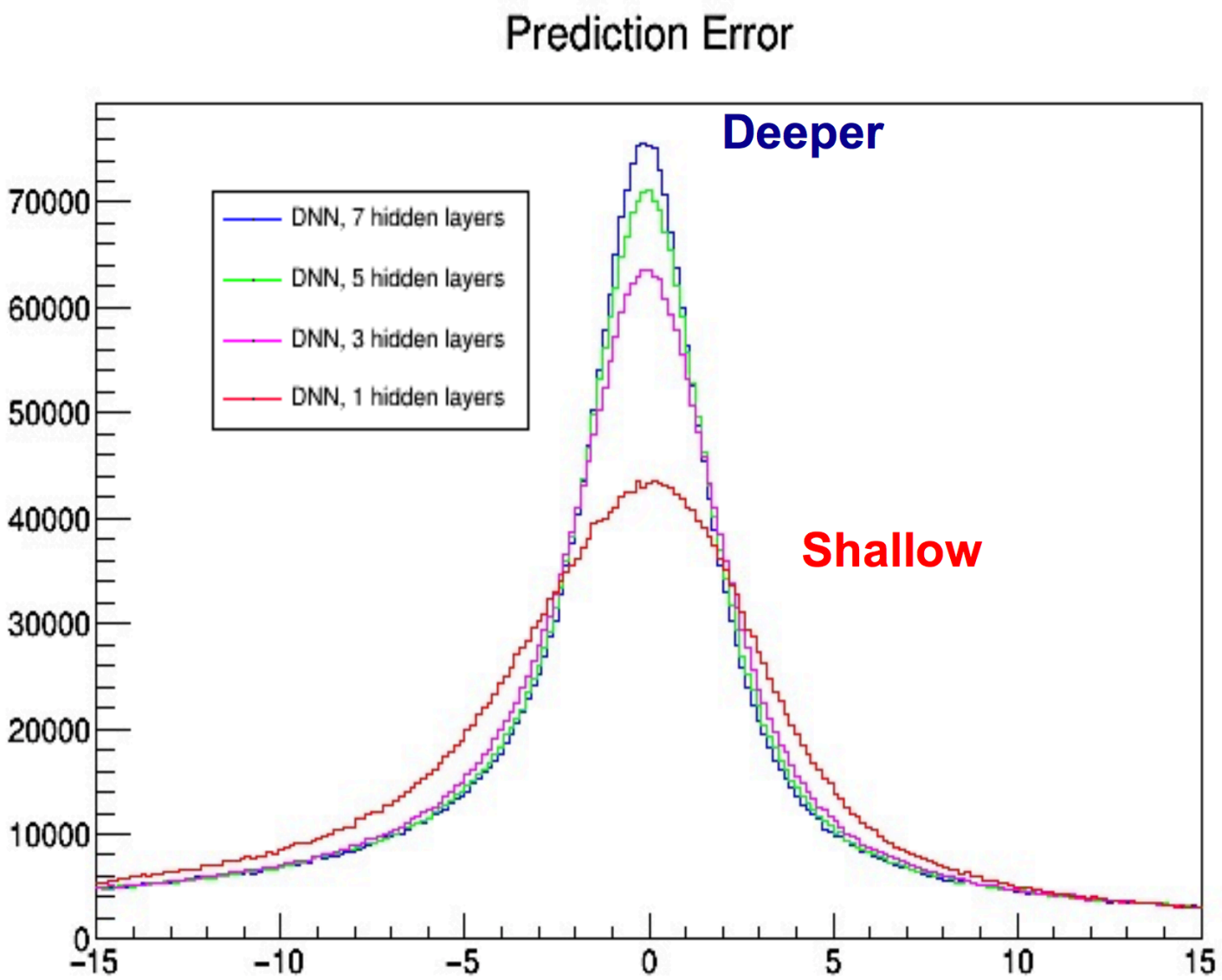
Example of Deep Learning for classification on a HEP data set

## DNN vs Standard ANN



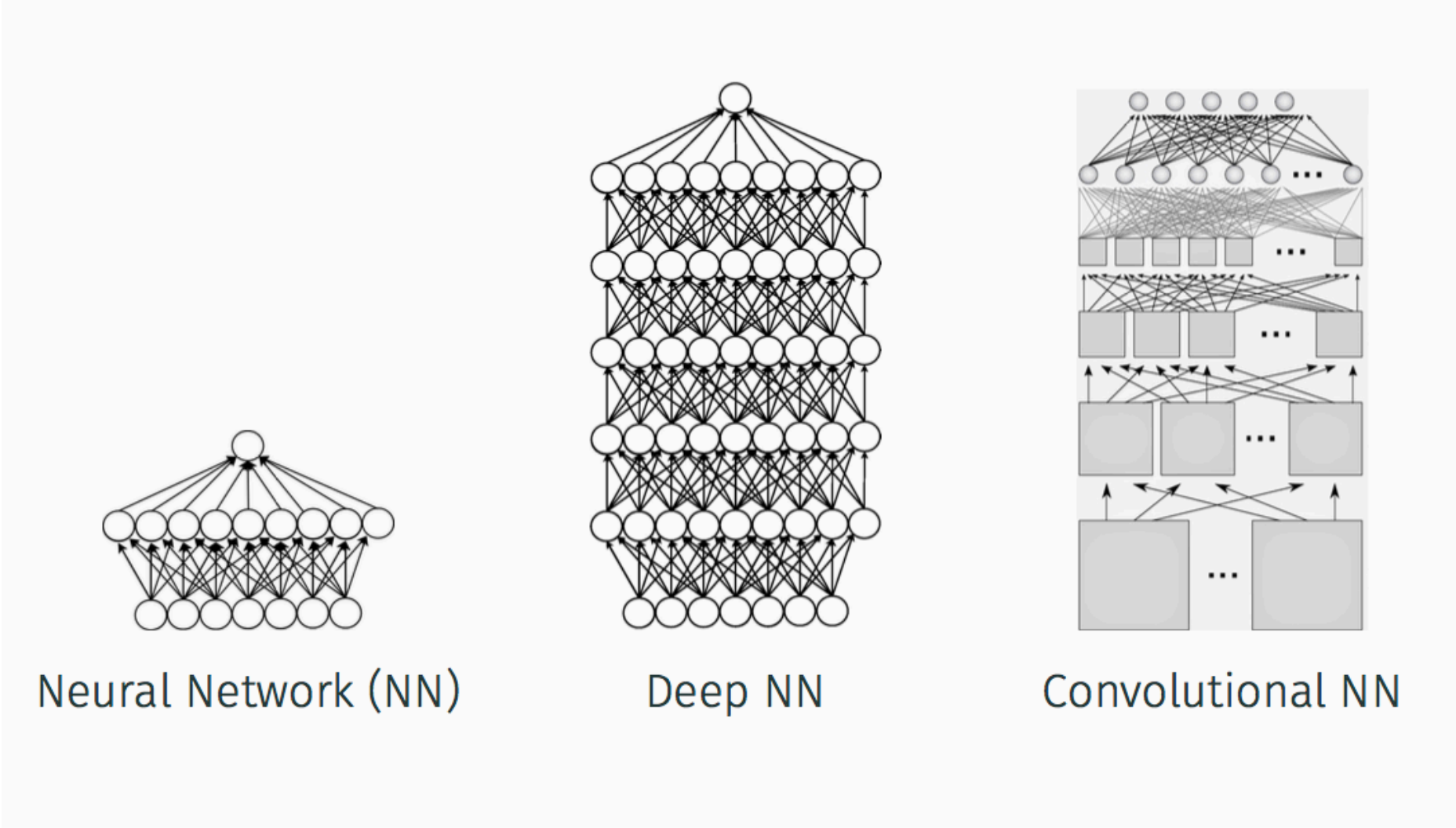

- Significant performance improvement in deep vs. shallow
- DNN learns the high level features. Almost no difference in performance when using only low level features

# Deep Learning for Regression



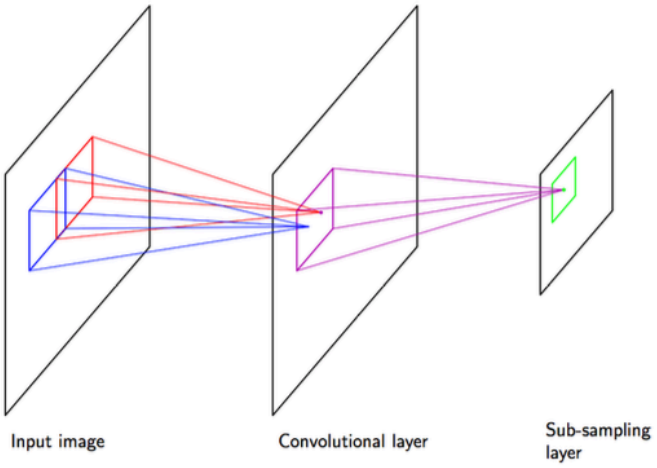

[S. Gleyzer]

# Convolutional Neural Networks

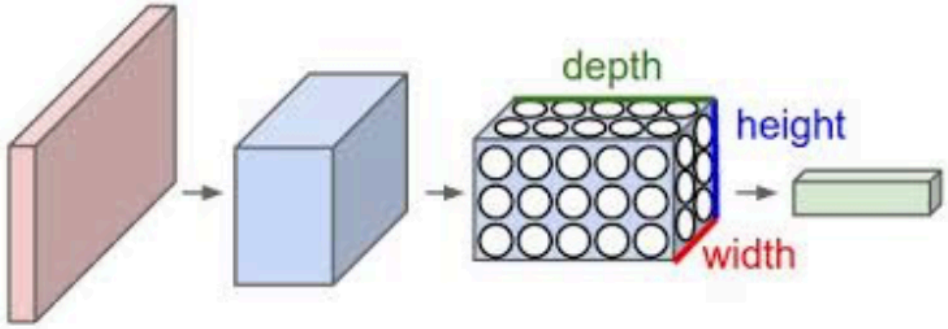


# Convolutional Neural Networks

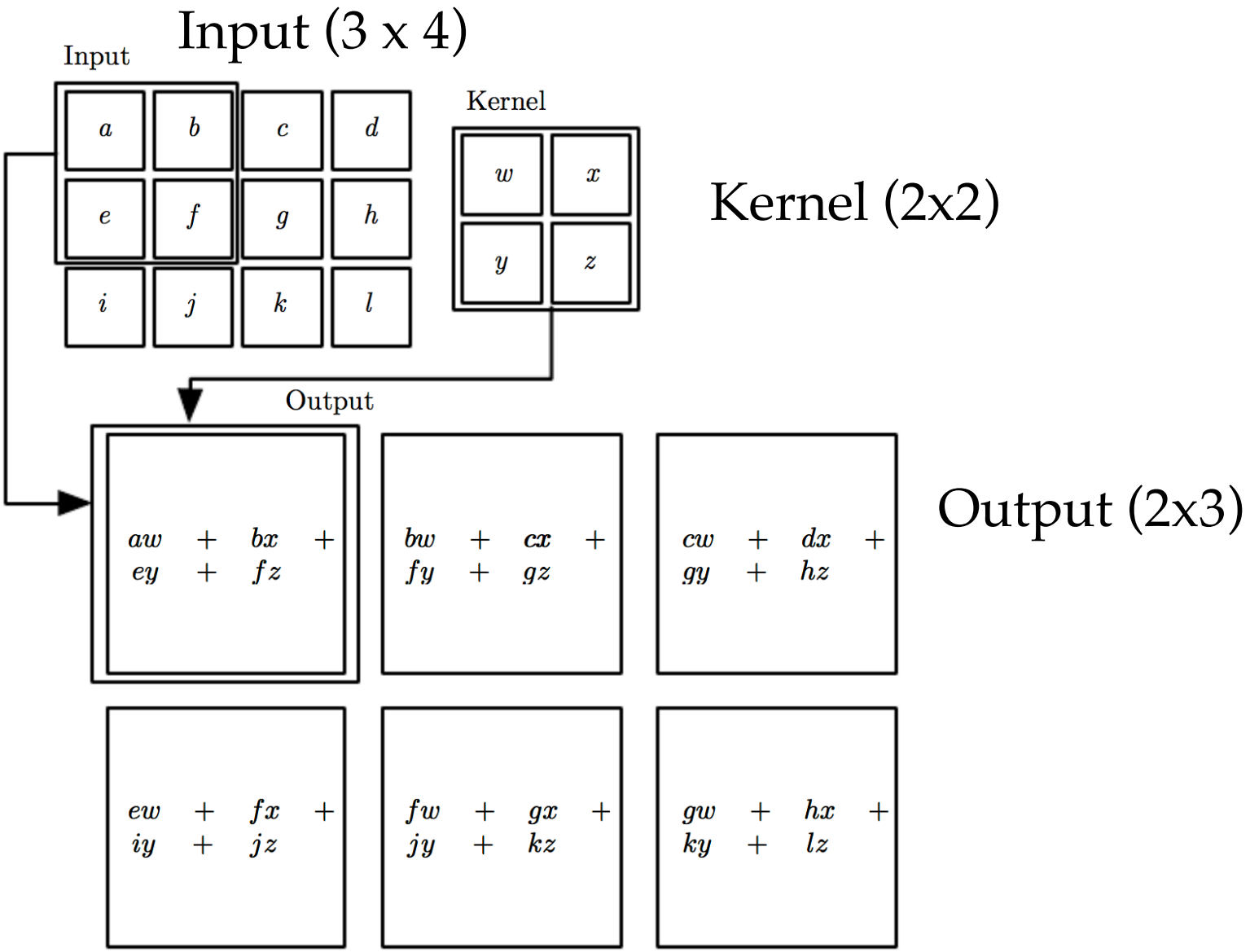
- Shared weights between neurons
- Neurons take only subsets of input
  - much less parameters than a traditional DNN
  - able to handle a large number of  inputs
    - e.g every pixel in an 2D or 3D image
- Very powerful for image recognition



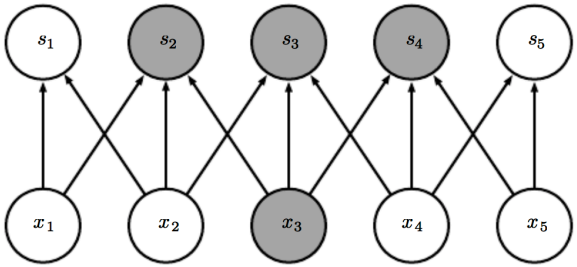

[Bishop]



# Convolutional Neural Networks



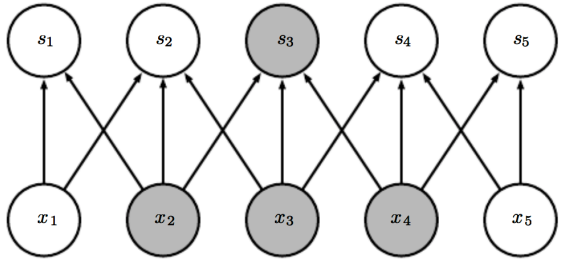
# Convolutional Neural Networks

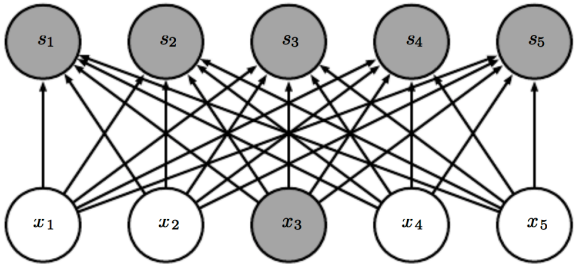



## Convolution:

**sparse connectivity**

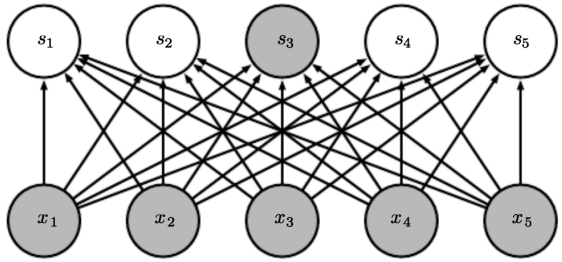
e.g. kernel width=3

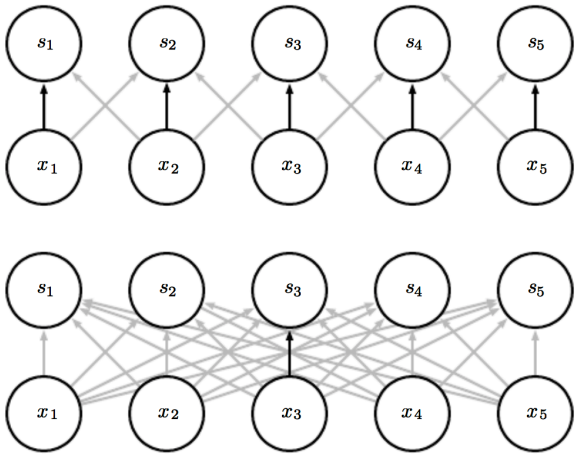



## Dense architecture:

every input connects to all output and vice versa



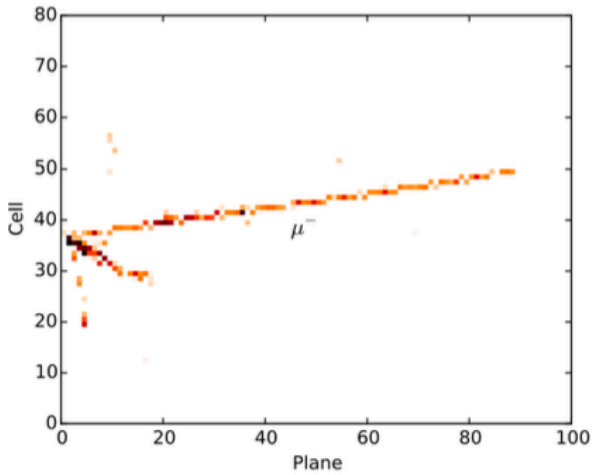

[www.deeplearningbook.org]

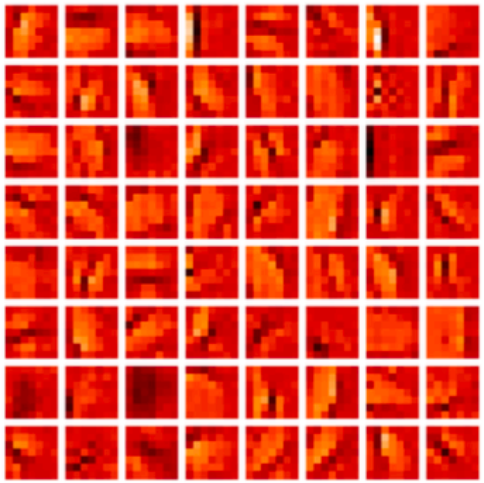
**Parameter sharing**:

same weights across neurons

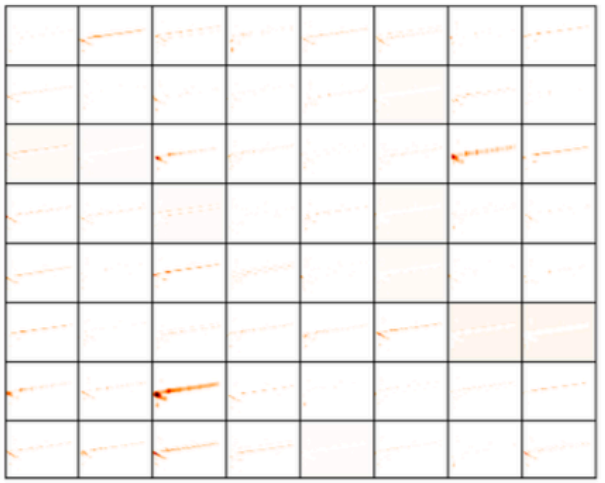
# CNN in HEP

- Neutrino event reconstruction



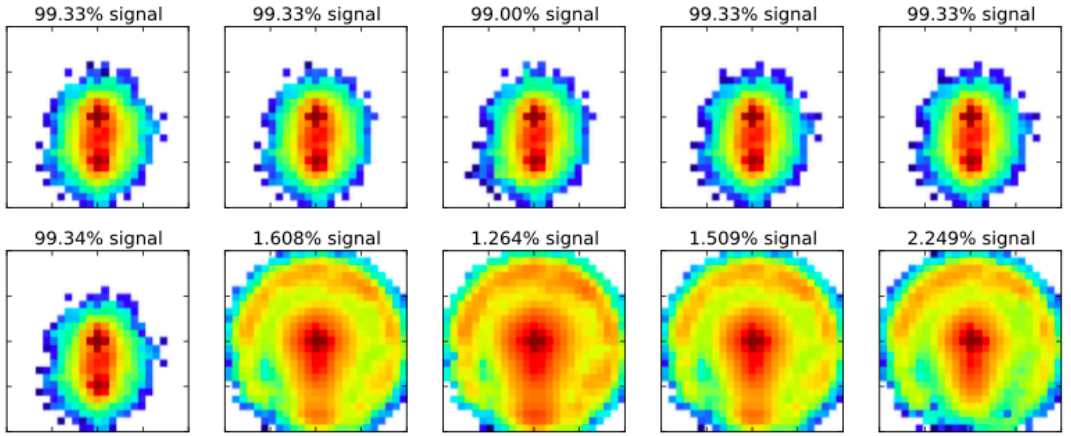



arXiv:1604.01444



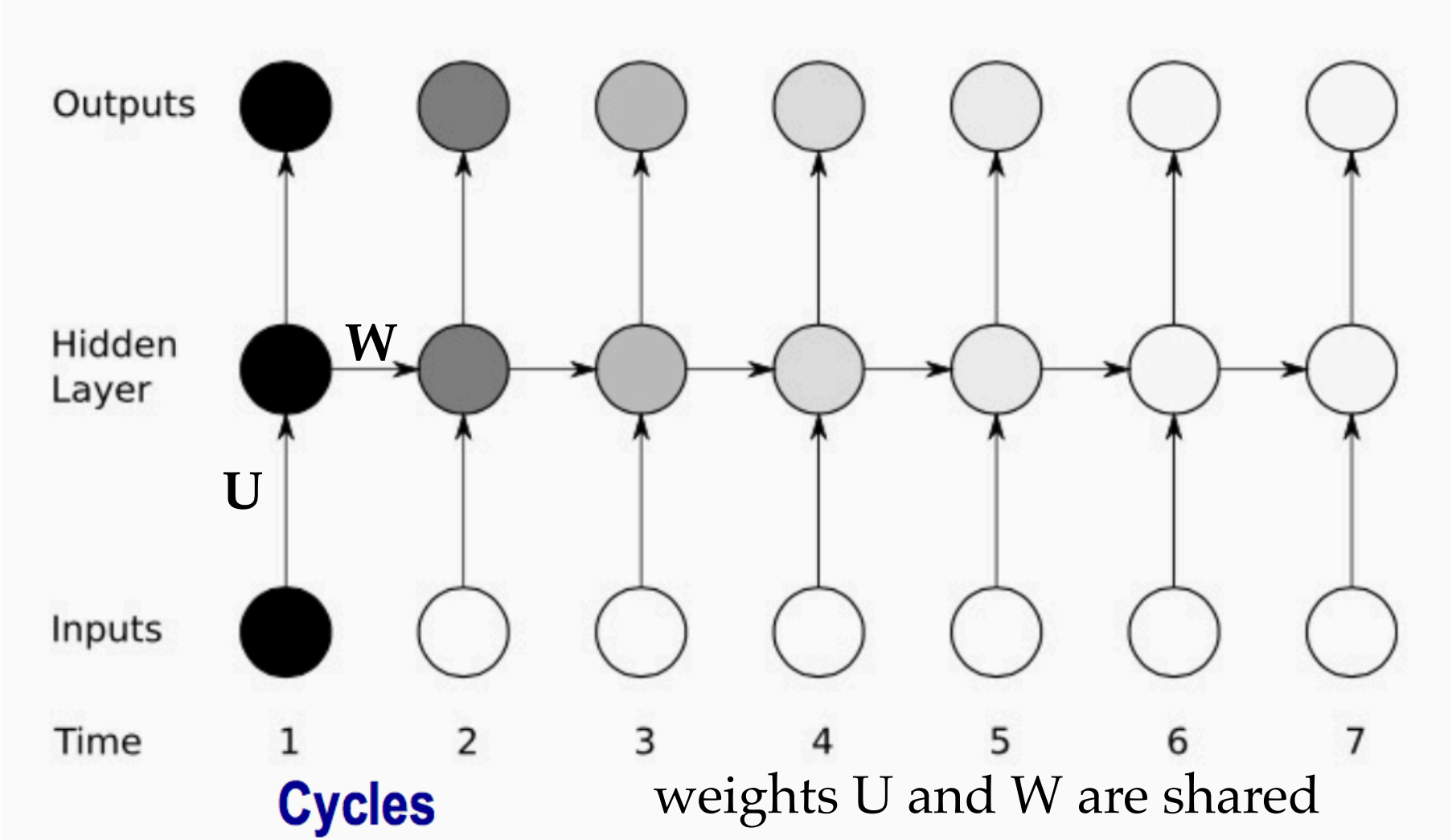
- Separate W-jets from quark/gluon jets

https://arxiv.org/abs/1511.05190

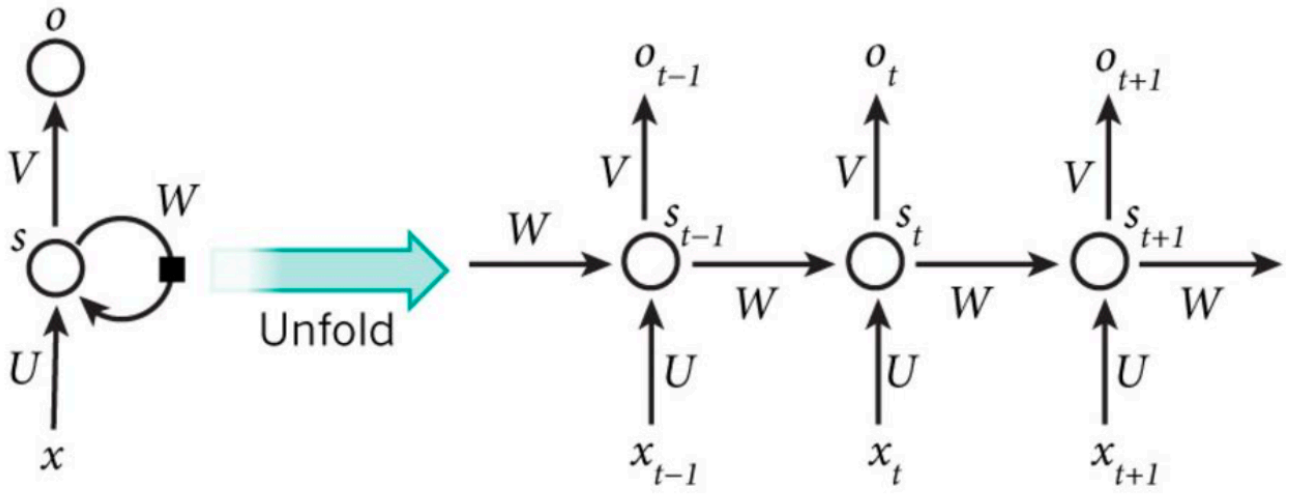


# Recurrent Neural Networks

- Able to process a sequence of data $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(t)}\}$
  - e.g. time dependent data



# Model of a RNN




$o_t = g(V\, s_t)$

$s_t = f(W s_{t-1} + U\, x_t)$

*RNN Computation, source: Nature*

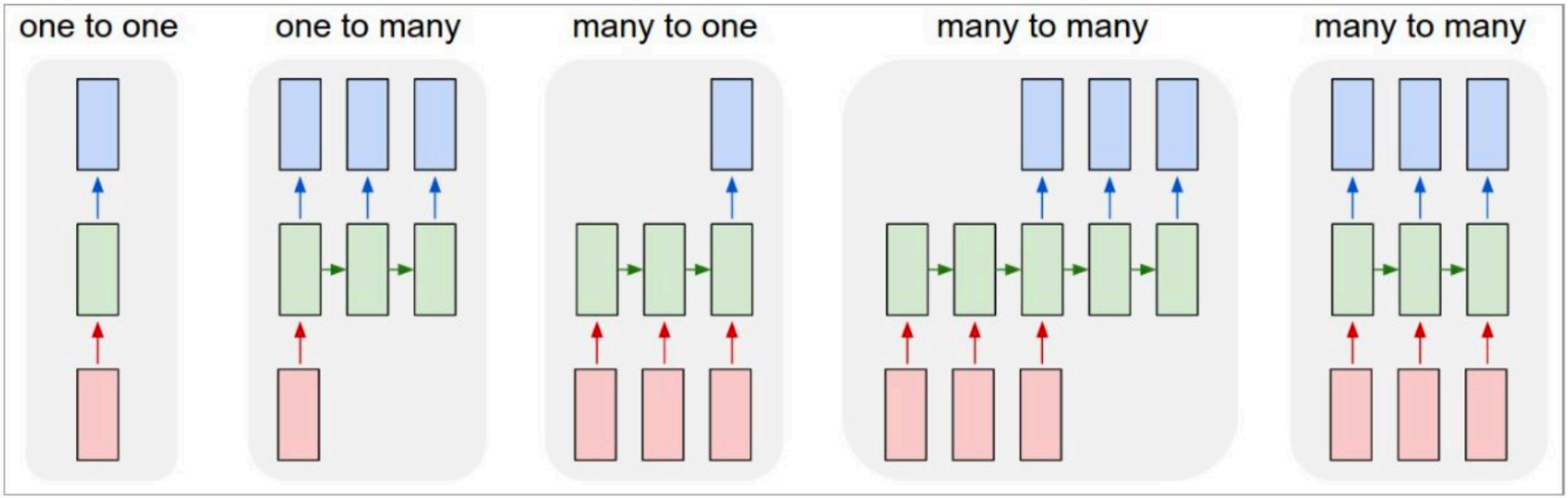
**V**: hidden-output weights

**W** : hidden-hidden weights

**U** : input-hidden weights

All nodes share the weights **U,W,V**

# Power of RNN

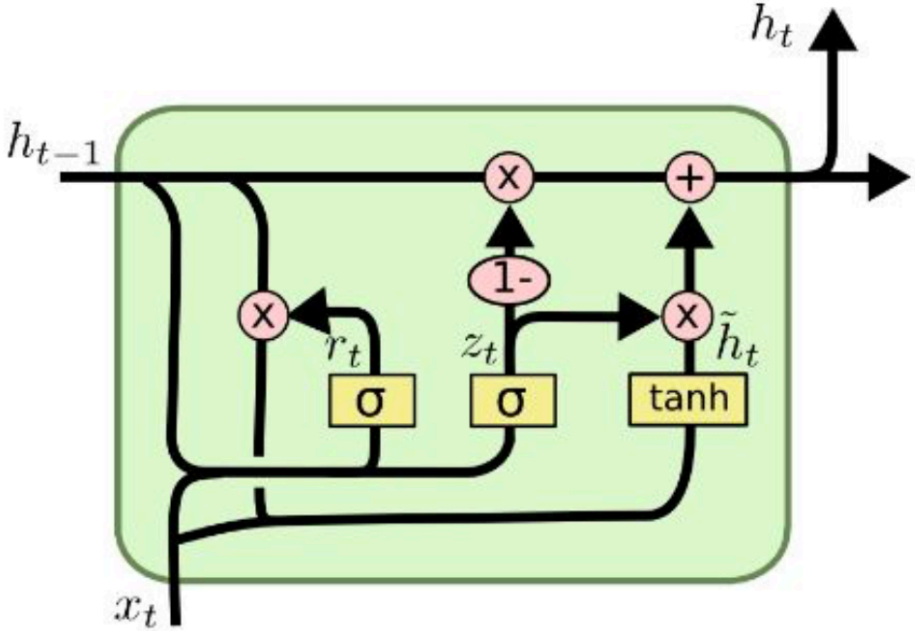



Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (more on this soon). From left to right: **(1)** Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification). **(2)** Sequence output (e.g. image captioning takes an image and outputs a sentence of words). **(3)** Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). **(4)** Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). **(5)** Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

*Source: Andrej Karapathy*

# LSTM

- RNN suffers from problem to preserve long recurrence memory vs to the short ones
  - gradient may explode or vanish due to recursive relations (e.g. $\mathbf{s}^t = \mathbf{W}\,\mathbf{s}^{t-1} \rightarrow \mathbf{s}^t = \mathbf{W}^t\mathbf{s}^1$)
- LSTM cells is a modified RNN cells introducing gates to prevent this problem and preserving better long term memories




$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

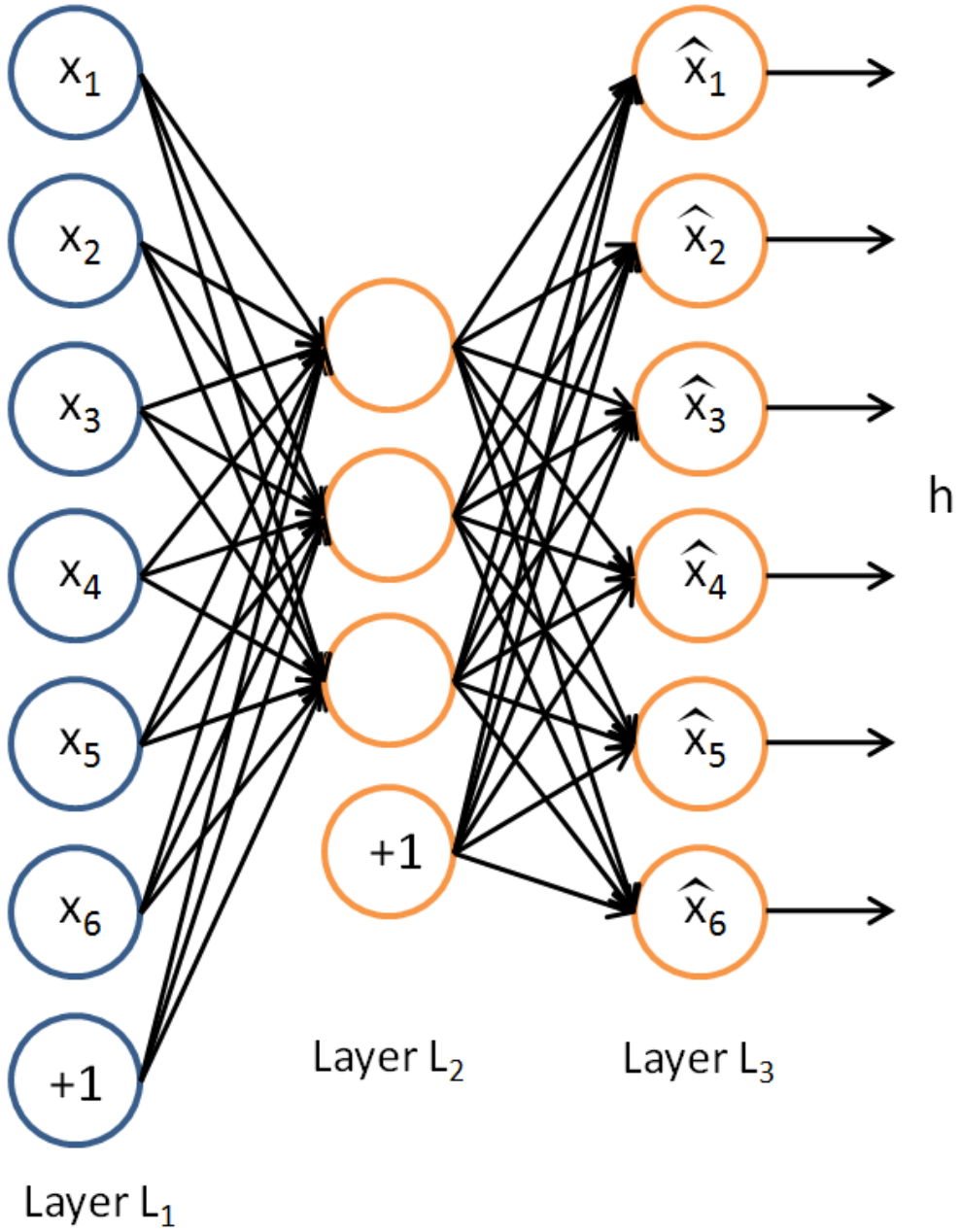
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

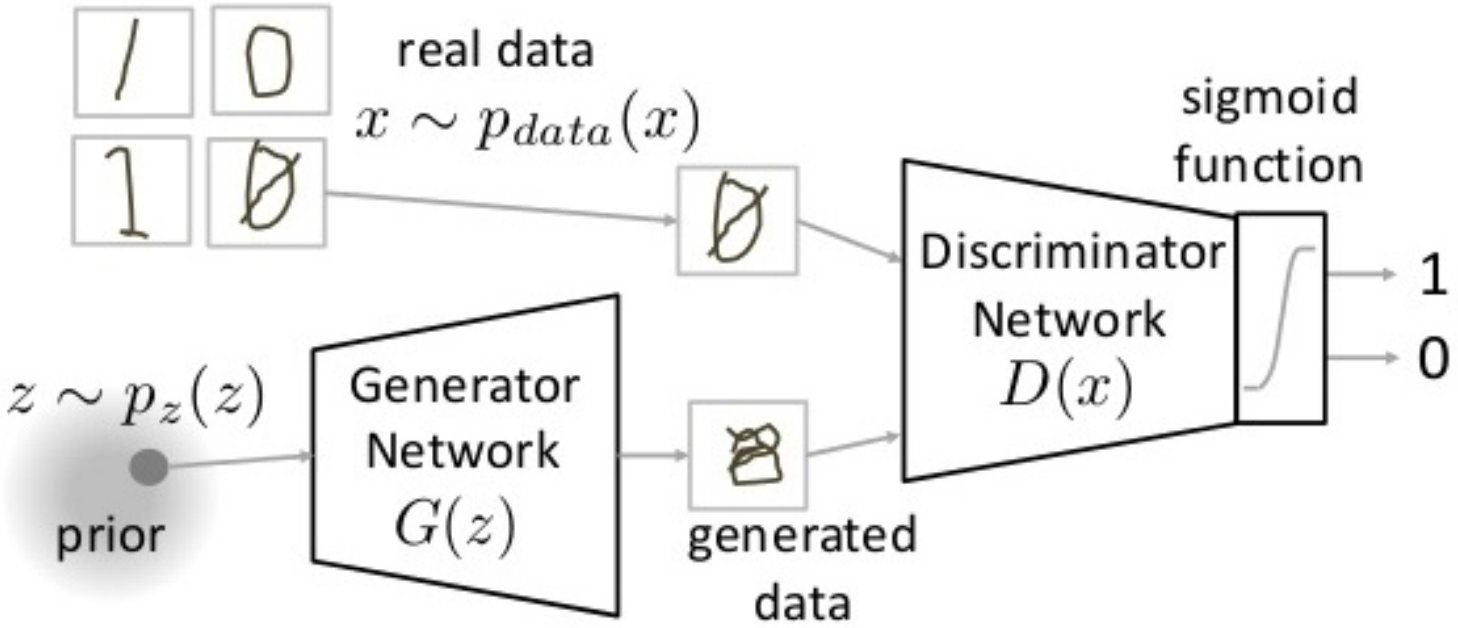
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

# Deep Autoencoder

- An unsupervised neural network
- Trained by setting the target values $y_i$ equal to the inputs $x_i$
- Can be used for **dimensionality reduction** or **anomaly detection**
  - and as a generator (**variational auto-encoders**)

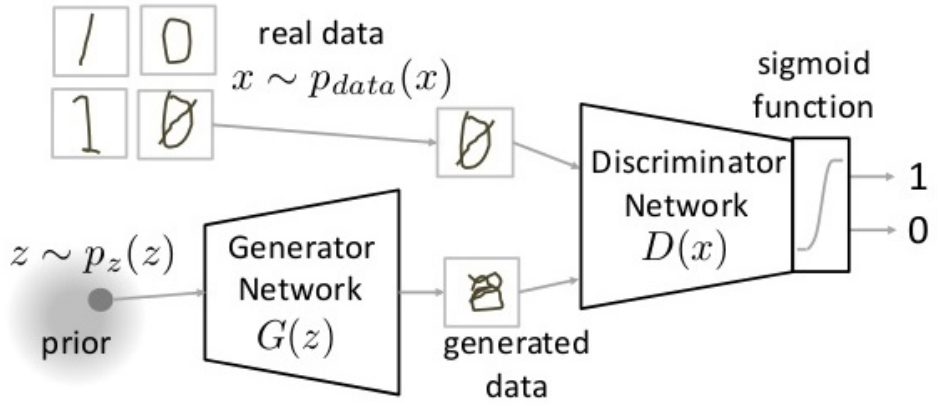


# GAN: Generative Adversarial Network




- **Generator network**:
  - output data from a random input G(x)
- **Discriminator network**:
  - discriminate the generated data from real ones
  - output probability D(x) that data are from real input

# GAN Optimization

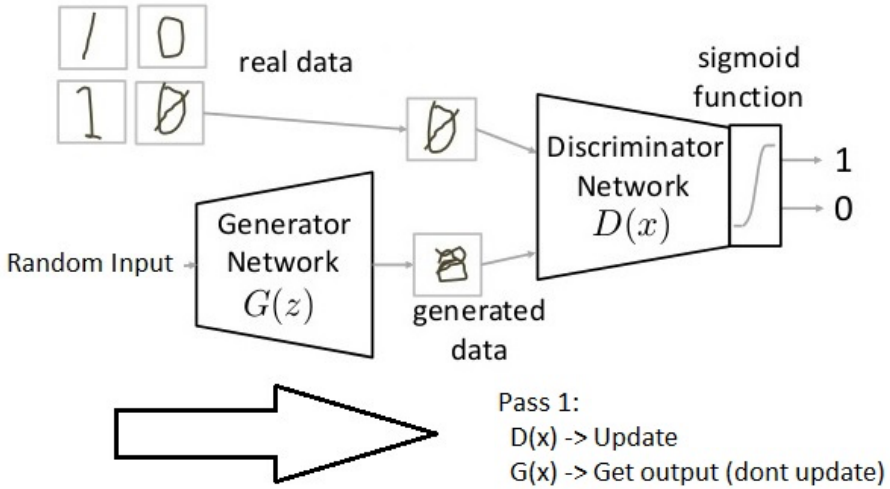



- Want to find discriminator parameters such that data coming from training sample and real one are as similar as possible
- Find generator parameters that make random (fake) generated data unlikely
  - classified by the discriminator as fake.
  - minimize for G cross-entropy log(1 - D(G(z))
- **Optimization of a GAN is then a min-max player game**

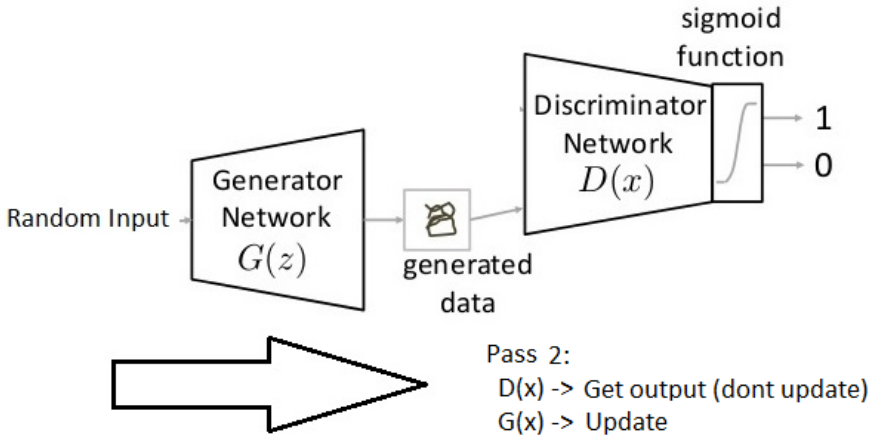
$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

# Training of GAN

- Iteratively procedure:
  - Train first discriminator with real data and un-trained output from generator (fake data)
    - learn to discriminate
  - Train generator with a fixed discriminator
  - Repeat the procedure for few iterations




Pass 1: Train Discriminator

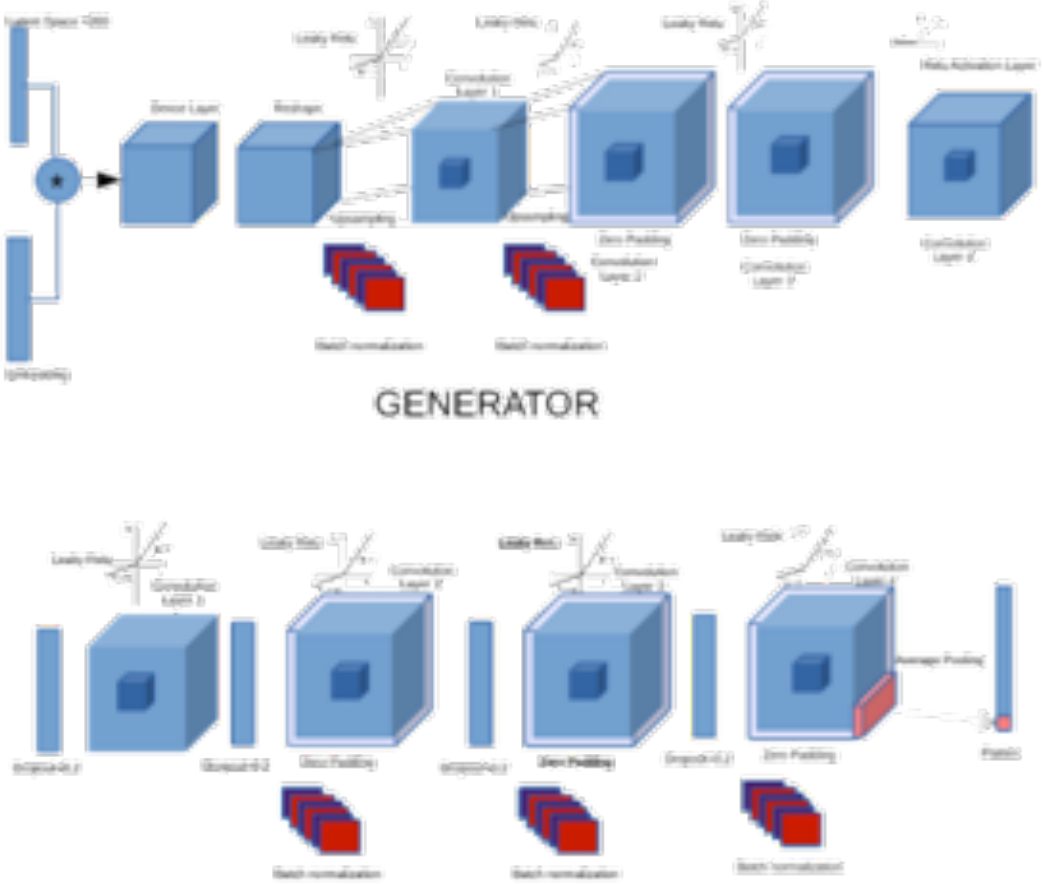


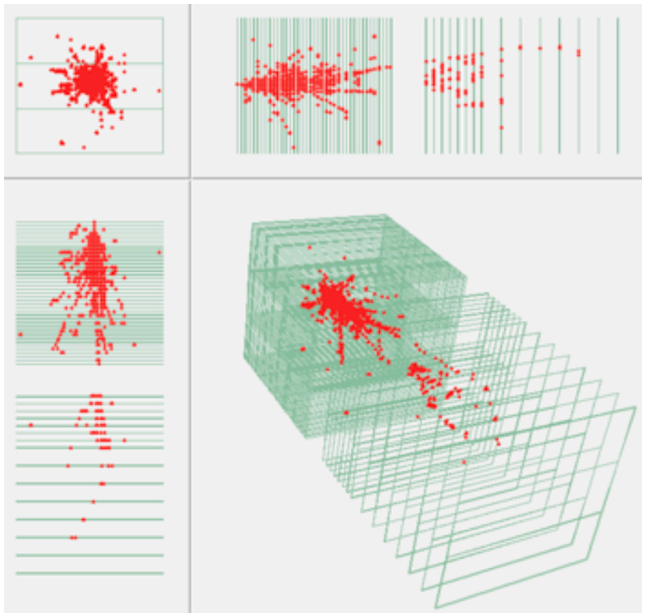

Pass 2: Train Generator

# Example: 3d GAN for Calo Images

GAN as possible fast simulations of calorimetric images

- Train with full simulated (Geant4) images
- use 3d convolutions



[S. Vallecorsa]

# Some General Advices

- There is no a-priori algorithm that will work best for every supervised learning problem
  - a model could work very well on one problem and poorly on another
  - need to try several algorithms
- Let's look at some empirical conclusions

# Empirical Analysis

- From **structured data**, expressing high level features (e.g. with physical meaning )
  - Decision tree based algorithms (Random forest, boosted decision trees) work very well
- From **unstructured data** (low level data)
  - deep learning algorithms are winning
    - network learns high level structures
  - CNN for image classification
  - RNN for text and speech recognition

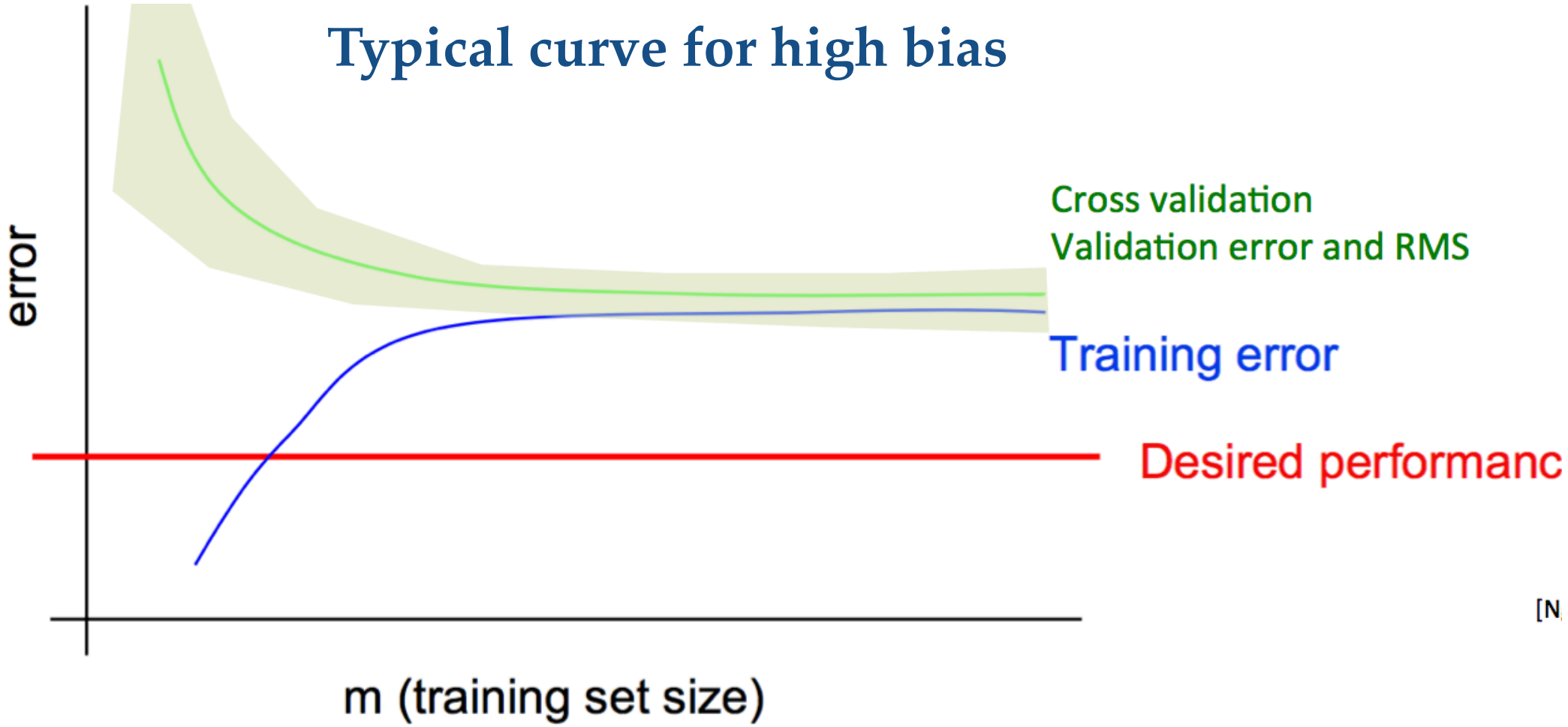
# Learning Curve: High Variance

Look at the learning curve

## Typical curve for high variance

# Learning Curve: High Bias

Look at the learning curve



# Possible Fixes

- Fixes to try:
  - Get more training data — Fixes high variance
  - Try smaller feature set size — Fixes high variance
  - Try larger feature set size — Fixes high bias
  - Try different features — Fixes high bias

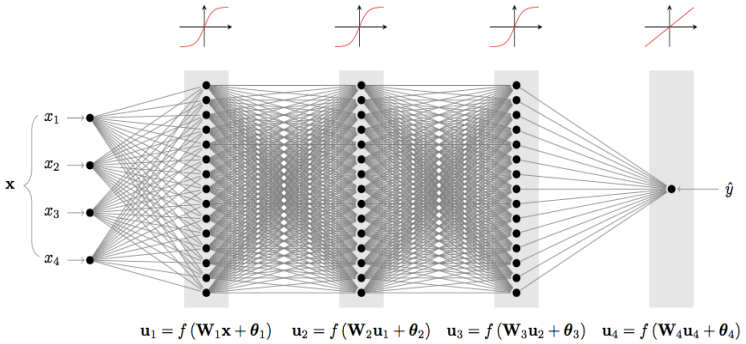
- Did the training converge?
  - Run gradient descent a few more iterations — Fixes optimization algorithm
    - or adjust learning rate
  - Try different optimization algorithm — Fixes optimization algorithm

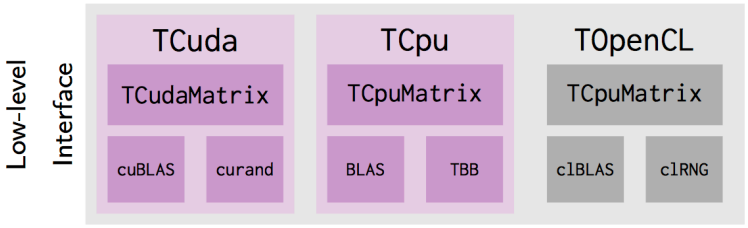
- Is it the correct model / objective for the problem?
  - Try different regularization parameter value — Fixes optimization objective
  - Try different model — Fixes optimization objective

[M. Kagan]

# Deep Learning in TMVA

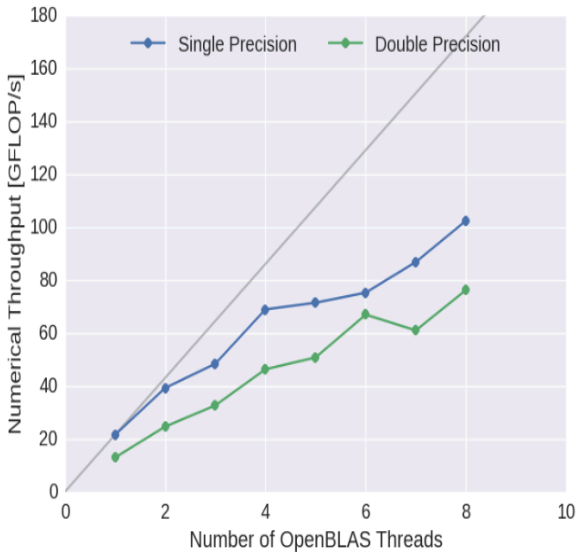
- Deep Learning library in ROOT/TMVA
  - parallel evaluation on CPU
    - implementation using OpenBLAS and TBB
  - GPU support
    - CUDA
    - OpenCL
  - Excellent performance and high numerical throughput
- For more information see
  - https://indico.cern.ch/event/565647/contributions/2308666/attachments/1345668/2028738/tmva_dnn_gpu.pdf



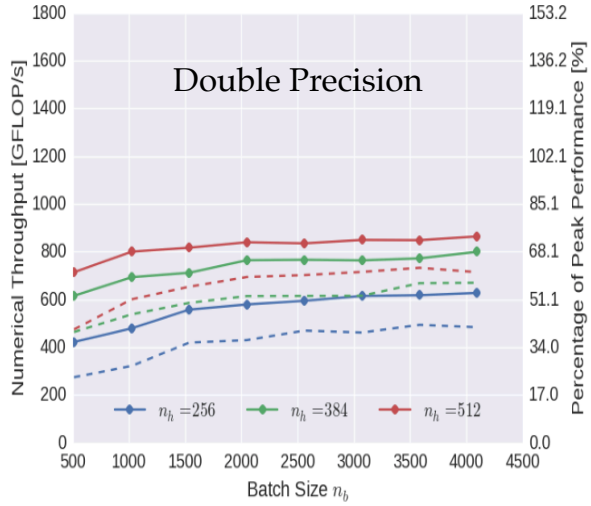



# Deep Learning Performance

- CPU Performance
  - Intel Xeon E5-2650, 8 × 4 cores
  - Estimate peak performance:
    - 16 GFLOP/s / core

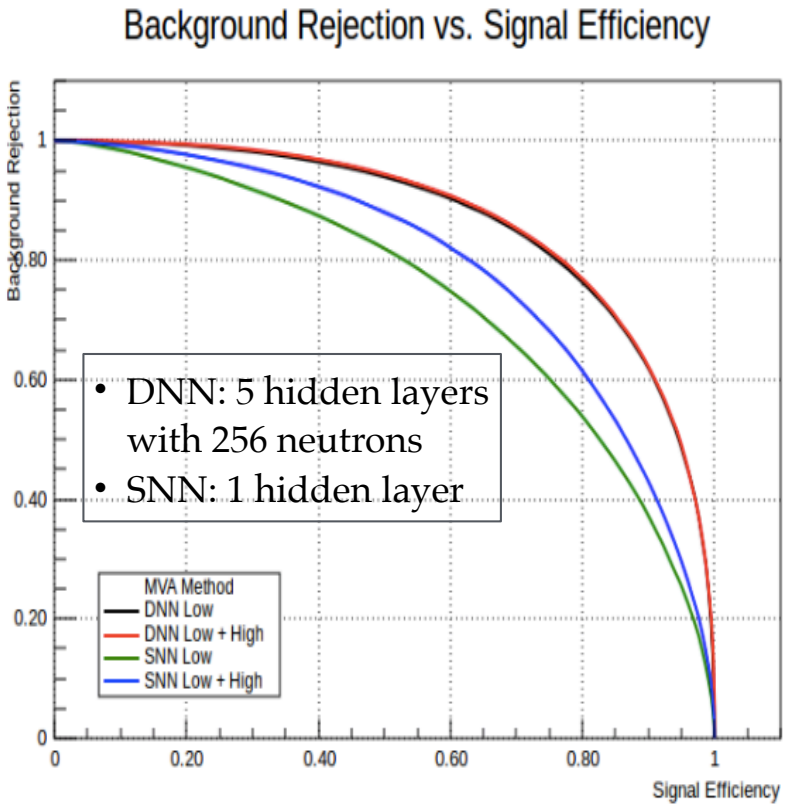



- GPU Performance
  - NVIDIA Tesla K20
  - Peak performance:
    - 1.17 TFLOP/s with double precision

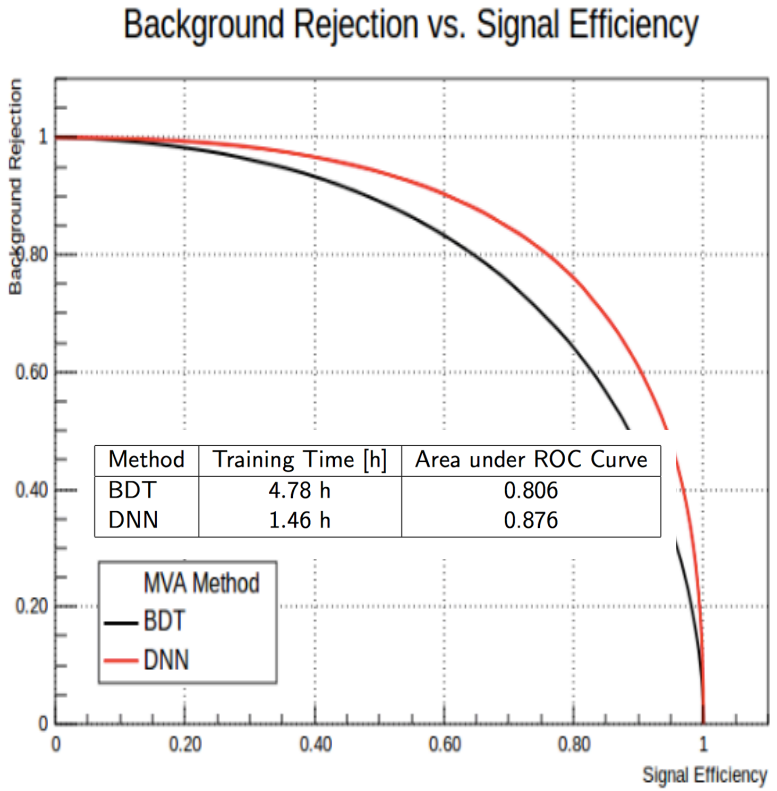


# Deep Learning Performance

## DNN vs Standard ANN



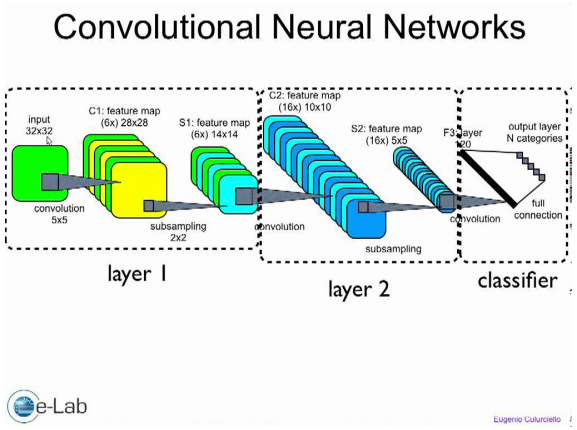

## DNN vs BDT

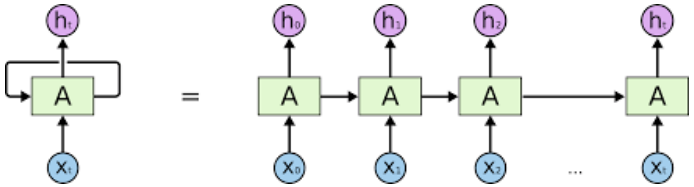


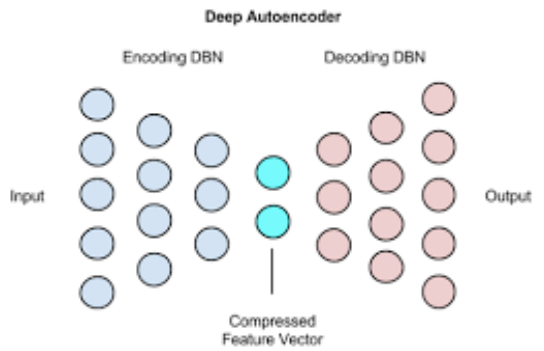

- Using Higgs public dataset with 11M events
- Significant improvements compared to shallow networks and BDT

# Deep Learning Developments in TMVA

- Focus on Deep Learning tools
- Extend existing Deep Neural Network classes by adding:
  - Convolutional Neural Network
    - very powerful for image data sets
  - Recurrent Neural Network
    - useful for time-dependent data
  - Deep Auto Encoder
    - useful for dimensionality reduction (pre-processing tool)
    - can be used as unsupervised tool (e.g. for anomaly detection)

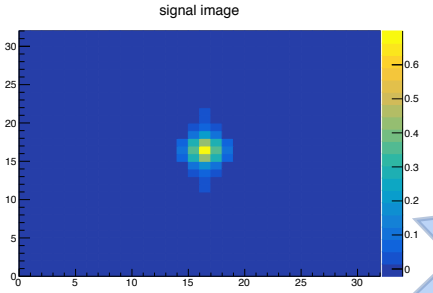


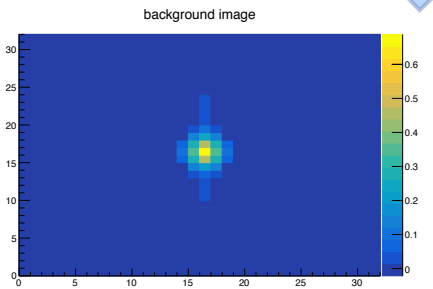



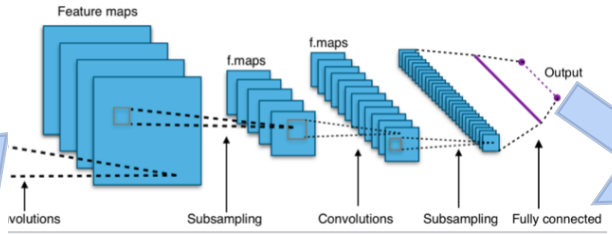
# Convolutional Neural Network

- Development well advanced, plan is to integrate soon in ROOT master, for next ROOT development release after 6.12
- Supporting both CPU and GPU
  - parallelisation and code optimisation is essential

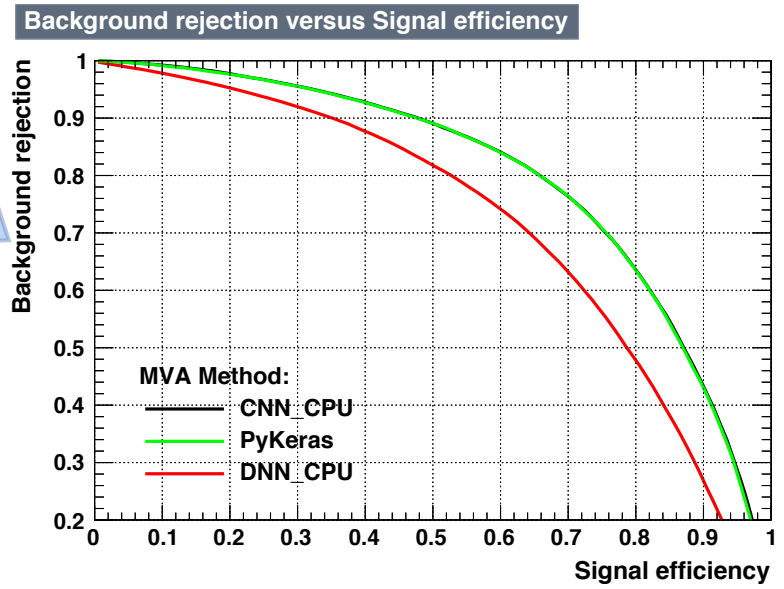



input 32x32 images



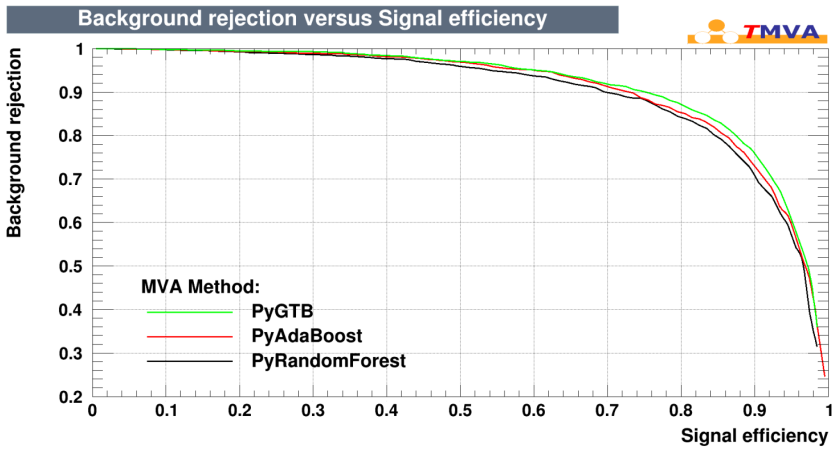

Convolutional + Pooling + Dense layers



# TMVA Interfaces

External tools are available as additional methods in TMVA and they can be trained and evaluated as any other internal ones.

- RMVA: Interface to Machine Learning methods in R
  - c50, xgboost, RSNNS, e1071
    - see http://oproject.org/RMVA
- PYMVA: Python Interface
  - **scikit-learn** with RandomForest, Gradient Tree Boost, Ada Boost)
    - see http://oproject.org/PYMVA
  - **Keras** (Theano + Tensorflow)
    - support model definition in Python
    - see https://indico.cern.ch/event/565647/contributions/2308668/attachments/1345527/2028480/29Sep2016_IML_keras.pdf
  - Input data are copied internally from TMVA to Numpy array



# Conclusions

- **Machine learning is a powerful branch of data science**
  - Many methods and applications
  - Lectures covered basics and advanced methods (boosted decision trees, deep neural networks,…)
  - Very exciting field (e.g. deep learning) with lots of new developments and applications
- ROOT provides Machine Learning Tools in TMVA and interfaces to most popular tools (scikit-learn, Tensor flow, keras,..)

# References

Lots of materials presented taken from these lectures:

- *M. Kagan*: CERN Academic Training Lectures (2017)
- *S. Gleyzer*: TAE 2017 Lectures
- *A. Rogozhnikov*: Lecture at Yandex summer school of Machine Learning in HEP (2016)
- E. von Toerne: Desy Terascale School of Statistics (2016)

Books:

- Elements of Statistical Learning (*Friedman et al…*)
- Pattern Recognition and Machine learning (*Bishop*)
- Deep Learning (*I. Goodfellow et al.*)