# Machine Learning in HEP

A. Sznajder

UERJ
Instituto de Fisica

May - 2019

# Outline

1. What is Machine Learning ?
2. Neural Networks
3. Learning as a Minimization Problem ( Gradient Descent and Backpropagation )
4. Deep Learning
5. Deep Architectures and Applications

# What is Machine Learning ?

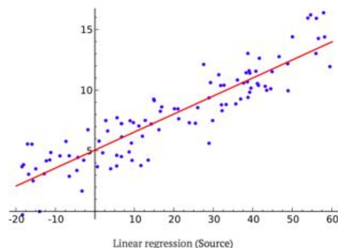
## Machine Learning (ML)

Machine learning (ML) is the study of computer algorithms capable of building a mathematical model out of a data sample, by learning from examples. The algorithm builds a predictive model without being explicitly programmed to do so.

# Centuries Old Machine Learning [1]

Take some points on a 2D graph, and fit a function to them. What you have just done is generalized from a few $(x, y)$ pairs (examples) , to a general function that can map any input $x$ to an output $y$

The Centuries Old Machine Learning Algorithm



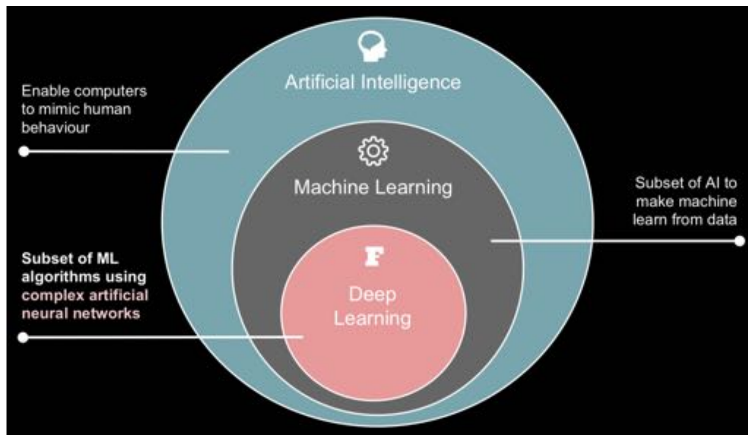

Linear regression (Source)

Linear regression is a bit too wimpy a technique to solve the problems of image , speech or text recognition, but what it does is essentially what supervised ML is all about: learning a function from a data sample

[1] http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learnin

# Artificial Inteligence

None of the systems we have nowadays are real AI [2]. The brain learns so efficiently that no ML method can match it




[2] Yann LeCun , Epistemology of Deep Learning : https://www.youtube.com/watch?v=gG5NCkMerHU&t=944s , https://www.youtube.com/watch?v=cWzi38-vDbE&t=768s

# Introduction to Machine Learning

Machine learning can be implemented by many different algorithms (ex: SVM, BDT, Bayesian Net , Genetic Algo ...) , but we will discuss only Neural Networks(NN)

## 1) Neural Network Topologies

- Feed Forward NN
- Recurrent NN

## 2) Learning Paradigms

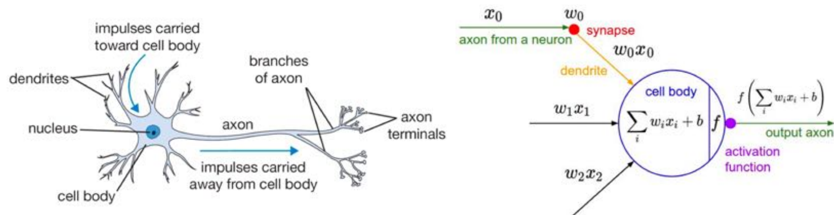
- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

## 3) Neural Networks Architectures

- Multilayer Perceptron(MLP)
- Convolutional Neural Network (CNN)
- Recurrent Neural Network (RNN)
- Long Short Time Memory(LSTM)
- Autoencoder(AE)
- Variational Autoencoder(VAE)
- Generative Adversarial Network(GAN)

# Neural Networks

Artificial Neural Networks (NN) are computational models vaguely inspired [3] by biological neural networks. A Neural Network (NN) is formed by a network of basic elements called neurons, which receive an input, change their state according to the input and produce an output




Original goal of NN approach was to solve problems like a human brain. However, focus moved to performing specific tasks, deviating from biology. Nowadays NN are used on a variety of tasks: image and speech recognition, translation, filtering, playing games, medical diagnosis, autonomous vehicles, ...

[3] Design of airplanes was inspired by birds, but they don't flap wings to fly !

# Artificial Neuron

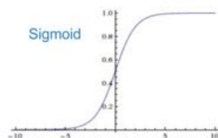
## Artificial Neuron (Node)

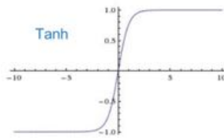
Each node of a NN receives inputs $\vec{x} = \{x_1, \ldots, x_n\}$ from other nodes or an external source and computes an output $y$ according to the expression

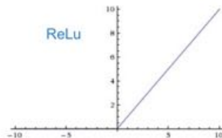
$$y = F\left(\sum_{i=1}^{n} W_i x_i + B\right) = F\left(\vec{W}.\vec{x} + B\right) \tag{1}$$

, where $W_i$ are connection weights, $B$ is the treshold and $F$ the activation function [4]

There are a variety of possible activation function and the most common ones are



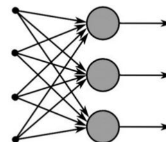

[4] The neuron output is sometimes called the activation

# Neural Network Topologies

Neural Networks can be classified according to the type of neuron interconections and the flow of information

## Feed Forward Networks

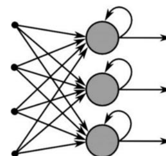
A feedforward NN is an neural network wherein connections between the nodes do not form a cycle. In a feed forward network information always moves one direction, from input to output, and it never goes backwards. Feedforward NN can be viewed as mathematical models of a function $F : \mathbb{R}^N \rightarrow \mathbb{R}^M$



Feed-Forward Neural Network

## Recurrent Neural Network

A Recurrent Neural Network (RNN) is a neural network that allows connections between nodes in the same layer, with themselves or with previous layers. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequential input data.



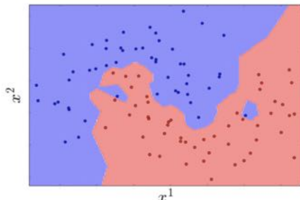
Recurrent Neural Network

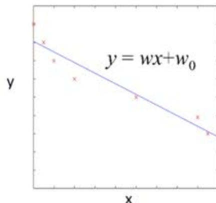
A NN layer is called a dense layer to indicate that it's fully connected. One also uses terms like recurrent, convolutional, pooling, normalizing and softmax layers

# Supervised Learning

## Supervised Learning

- During training a learning algorithm adjust the network's weights to values that allows the NN to map the input to the correct output.
- It calculates the error between the target output and a given NN output and use error to correct the weights.
- Given some labeled data $D = \{(\vec{x}_1, \vec{t}_1), \ldots, (\vec{x}_n, \vec{t}_n)\}$ with features $\{\vec{x}_i\}$ and targets $\{\vec{t}_i\}$ , the algorithm finds a mapping $\vec{t}_i = F(\vec{x}_i)$
- **Classification:** $\{\vec{t}_1, \ldots, \vec{t}_n\}$ ( finite set of labels )
- **Regression:** $\vec{t}_i \in \mathbb{R}^n$

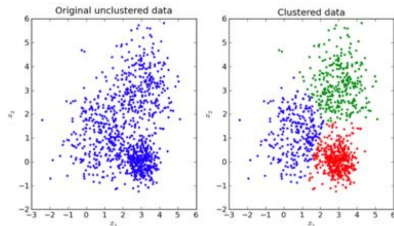


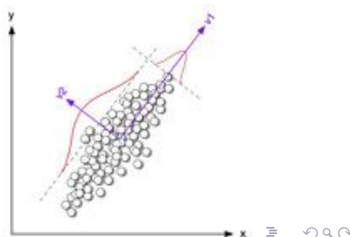



# Unsupervised Learning

## Unsupervised Learning

- No labeled data is used at training and the NN finds patterns within input data
- Given some data $D = \{\vec{x}_1, \ldots, \vec{x}_n\}$ , but no labels, find structures in the data
  - **Clustering:** partition the data into sub-groups $D = \{D_1 \cup D_2 \cup \ldots . D_k\}$
  - **Dimensional Reduction:** find a low dimensional representation of the data with a mapping $\vec{y} = F(\vec{x})$ , where $F : \mathbb{R}^n \to \mathbb{R}^m$ with $n >> m$



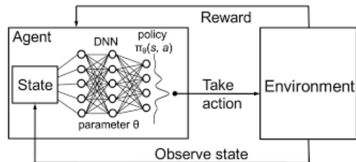



# Reinforcement Learning

## Reinforcement Learning [5]

Similar to supervised learning but instead of a target output, a reward is given based on how well the system performed. The algorithm take actions in an environment so as to maximize some notion of cumulative reward.

- Inspired by behaviorist psychology and strongly related with how learning works in nature
- Maximize the reward the system receives through trial-and-error
- Algorithm learns to make the best sequence of decisions to achieve goal.
- Requires a lot of data, so applicable in domains where simulated data is readily available: gameplay, robotics, self-driving vehicles
- Reinforcement learning algorithms: Q-learning, policy gradients ...

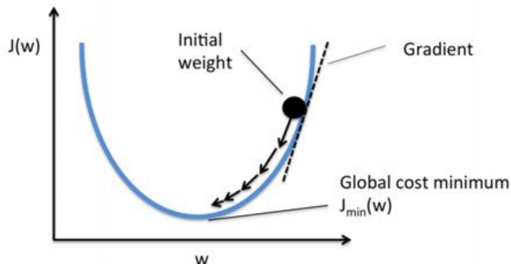



[5] https://www.youtube.com/watch?v=eG1Ed8PTJ18

# Supervised Learning - Training Process

## Learning as an Error Minimization Problem

1. Random NN parameters ( weights and bias ) initialisation
2. Choose a Loss Function , differentiable with respect to model parameters
3. Use training data to adjust parameters ( gradient descent + back propagation ) that minimize the loss
4. Repeat until parameters values stabilize or loss is below a chosen treshold



# Supervised Learning - Loss Function

The Loss function quantifies the error between the NN output $\vec{y} = F(\vec{x})$ and the desired target output $\vec{t}$.

## Squared Error Loss ( Regression )

If we have a target $t \in \mathcal{R}$ and a real NN output $y$

$$L = \frac{1}{2}\left(y - \vec{t}\right)^2$$

## Cross Entropy Loss [6] ( Classification )

If we have $m$ classes with binary targets $t \in \{0, 1\}$ and output probabilities $y$:

$$L = -\sum_{i=1}^{m} t_i log(y_i)$$

## Cost Function

The Cost function is the mean Loss over a data sample $\{\vec{x}_i\}$

$$\bar{L} = \frac{1}{n}\sum_{i=1}^{n} L(\vec{x}_i)$$

The activation function type used in the output layer is directly related to loss used for the problem !

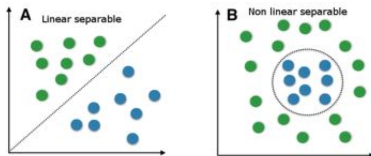
[6] For $m = 2$ we have the Binary Cross Entropy $L = -t \log y - (1 - t) \log(1 - y)$

# The Perceptron

The perceptron algorithm is a binary linear classifier invented in 1957 by F.Rosenblatt. It's formed by a single neuron that takes input $\vec{x} = (x_1, \ldots, x_n)$ and outputs $y = 0, 1$ according to

**Perceptron Model** [7]

$$y = \begin{cases} 1, \; if \; (\vec{W}.\vec{x} + B) > 0 \\ 0, \; otherwise \end{cases}$$




To simplify notation define $W_0 = B$ , $\vec{x} = (1, x_1, \ldots, x_n)$ and call $\theta$ the Heaviside step function

**Perceptron Learning Algorithm**

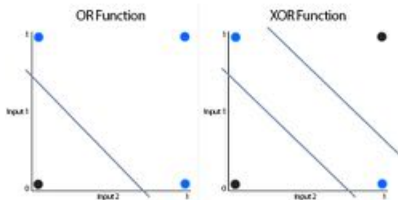
Initialize the weights and for each example $j$ in training set $D = \{(\vec{x}_1, t_1), \ldots, (\vec{x}_m, t_m)\}$ , where $t_j$ are output targets, and perform the following steps for each $(\vec{x}_j, t_j) \in D$

1. Calculate the actual output and the ouput error: $y_j = \theta(\vec{W}.\vec{x}_j)$ and $Error = 1/m \sum_{j=1}^{m} |y_j - t_j|$
2. Modify(update) the weights to minimize the error: $\delta W_i = r.(y_j - t_j).X_i$ , where $r$ is the learning rate
3. Return to step 1 until output error is acceptable

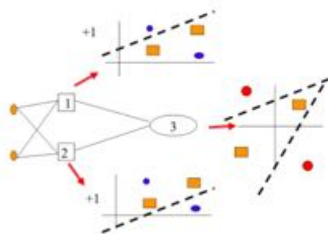
[7] Equation of a plane in $\mathbb{R}^n$ is $\vec{W}.\vec{x} + B = 0$

# Perceptron and XOR Problem

Perceptrons limitations with non linearly separable problems makes its unable to learn the Boolean *XOR* function




We need a multi layer architecture to solve the *XOR* problem in two-stages



# Multilayer Perceptron

The Multilayer Perceptron(MLP) is a fully connected NN with at least 2 layers ( hidden and output ). The layers uses nonlinear activation functions $F$ which can differ between layers [8] . The MLP is the simplest feed forward neural network and has a model [9]

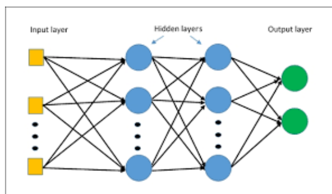
## Multilayer Perceptron Model

For a MLP with inputs nodes $\vec{x}^{(0)}$, one hidden layer of nodes $\vec{x}^{(1)}$ and output layer of nodes $\vec{x}^{(2)}$ , we have

$$\begin{cases} \vec{x}^{(1)} = \vec{F}^{(1)}\left(\vec{W}^{(1)}.\vec{x}^{(0)}\right) \\ \vec{x}^{(2)} = \vec{F}^{(2)}\left(\vec{W}^{(2)}.\vec{x}^{(1)}\right) \end{cases}$$

Eliminating the hidden layer variables $\vec{H}$ we get

$$\Rightarrow \quad \vec{x}^{(2)} = \vec{F}^{(2)}\left(\vec{W}^{(2)}.\vec{F}^{(1)}\left(\vec{W}^{(1)}.\vec{x}^{(0)}\right)\right) \tag{2}$$




A MLP can be seen as a parametrization of a mapping (function) $F_{w,b} : \mathbb{R}^n \rightarrow \mathbb{R}^m$

---

[8] A MLP with $m$ layers using linear activation functions can be reduced to a single layer !

[9] The tresholds $\vec{B}$ are represented as weights by redefining $\vec{W} = (B, W_1, \ldots, W_n)$ and $\vec{x} = (1, x_1, \ldots, x_n)$ ( bias is equivalent to a weight on an extra input of activation=1 )

# Multilayer Perceptron as a Universal Approximator

## Universal Approximation Theorem

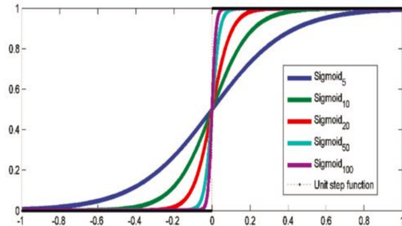
A single hidden layer feed forward neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden neurons [10]

The theorem doesn't tell us how many neurons or how much data is needed !

## Sigmoid → Step Function

For large weight $W$ the sigmoid turns into a step function, while $B$ gives its offset

$$y = \frac{1}{1 + e^{-(w.x+b)}} \quad (3)$$



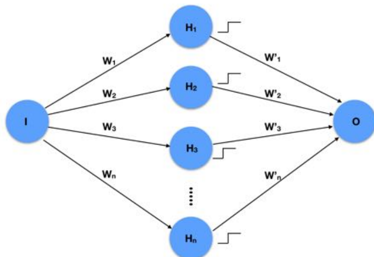

[10] Cybenko,G.(1989) Approximations by superpositions of sigmoidal functions, Math.ofCtrl.,Sig.,andSyst.,2(4),303
Hornik,K.(1991) Approximation Capabilities of Multilayer Feedforward Networks, Neural Networks, 4(2), 251

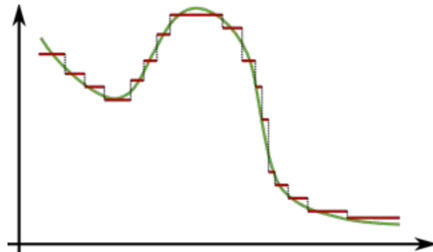
# Multilayer Perceptron as a Universal Approximator

## Approximating $F(x)$ by Sum of Steps

A continuous function can be aproximated by a finite sum of step functions. The larger the number of steps(nodes), the better the aproximation

Consider a network composed of a single input , $n$ hidden nodes and a single output. Tune the weights such that the activations approximate steps functions with appropriate tresholds and add them together !



The same works for any activation function $f(x)$, limited when $x \to \pm\infty$. One can always tune weights $W$ and tresholds $B$ such that it behaves like a step function !

# Learning as a Minimization Problem ( Gradient Descent and Backpropagation )

# Gradient Descent Method (Loss Minimization)

The learning process is a loss $L(\vec{W})$ minimization problem, where weights are tunned to achieve a minimum network output error. This minimization is usually achieved by applying the Gradient Descent iterative method

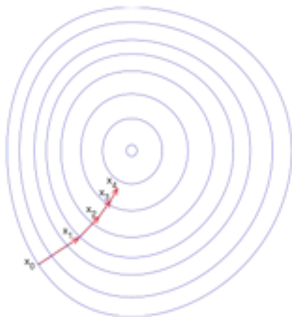
## Gradient Descent

A multi-variable function $F(\vec{x})$ decreases fastest in the direction of its negative gradient $-\nabla F(\vec{x})$ [11].

Choosing an initial point $\vec{x}_0$, one can use a recursion formula to obtain a sequence of points $\{\vec{x}_1, \ldots, \vec{x}_n\}$ leading to the minimum

$$\vec{x}_{n+1} = \vec{x}_n - \lambda \nabla F(\vec{x}_n) \quad \text{, where } \lambda \text{ is the step}$$

The monotonic sequence $F(\vec{x}_0) \geq F(\vec{x}_1) \geq \ldots \geq F(\vec{x}_n)$ indicates it converges to local minimum !



Gradient Descent uses only first order derivatives, which is efficiently and simply calculated by backpropagation. Minimization methods like Newton and BFGS needs second order derivative(Hessian), which is computationally costly and memory inefficient.

[11] The directional derivative of $F(\vec{x})$ in the $\vec{u}$ direction is $D_{\vec{u}} = \hat{u}.\nabla F$

# Stochastic Gradient Descent

## Stochastic Gradient Descent(SGD)

SGD [12] is called stochastic because data samples are selected randomly (shuffled), instead of the order they appear in the training set. This allows the algorithm to try different "minimization paths"at each training epoch

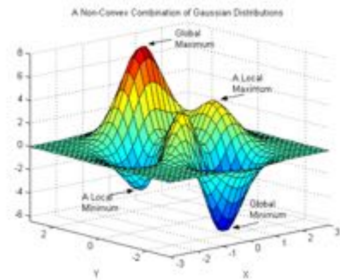
- It can also average gradient with respect to a set of events (minibatch)
- Noisy estimates average out and allows "jumping" out of bad critical points
- Scales well with dataset and model size

## Convex Loss

- Single global minimum
- Iterations toward minimum

## Non-Convex Loss

- Get stuck in local minima
- Convergence issues
- Adaptive variants

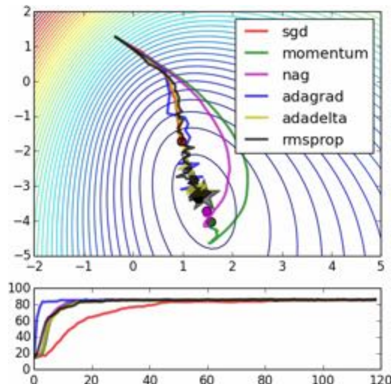



[12] https://deepnotes.io/sgd-momentum-adaptive

# SGD Algorithms Improvements [14]

## SGD Algorithms [13] :

- **Vanilla SGD**
- **Momentum SGD :** uses update $\Delta w$ of last iteration for next update in linear combination with the gradient
- **Annealing SGD :** step, exponential or 1/t decay
- **Adagrad :** adapts learning rate to updates of parameters depending on importance
- **Adadelta :** robust extension of Adagrad that adapts learning rates based on a moving window of gradient update
- **Rmsprop :** rescale gradient by a running average of its recent magnitude
- **Adam :** rescale gradient averages of both the gradients and the second moments of the gradients




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

[14] http://ruder.io/optimizing-gradient-descent

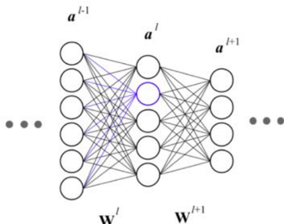
# Backpropagation

Backpropagation is a calculation technique for applying the gradient descent method to multilayer networks. An error is computed at the network output and distributed backwards to each layer, as a consequence of recursive use of the chain rule [15]

## MLP Loss Function

Consider a MLP with $n$ layers (not counting input layer) with a quadratic loss. One can view the MLP loss as a $n$ layer composite function of the weights, where $W_{ij}^{(l)}$ connects the neuron $j$ in layer $(l-1)$, to neuron $i$ in layer $l$. Defining the activation $a_i^{(l)} = F(W_{ij}^{(l)} a_j^{(l-1)})$ as the output of neuron $i$ in layer $(l)$, we have

$$L(\vec{W}) = \frac{1}{2}\left[a_i^{(n)} - t_i\right]^2 = \frac{1}{2}\left[F(W_{ij}^{(n)} \cdots F(W_{rs}^{(1)} a_s^{(0)}) \cdots) - t_i\right]^2$$



# Backpropagation

Let's define $z_i^{(l)} = W_{ij}^{(l)} a_j^{(l-1)}$, such that $a_i^{(l)} = F(z_i^{(l)})$. The loss gradients in $l$-layer are

$$\frac{\partial L}{\partial W_{kj}^{(l)}} = \left[\frac{\partial L}{\partial z_k^{(l)}}\right] \underbrace{\frac{\partial z_k^{(l)}}{\partial W_{kj}^{(l)}}}_{a_j^{(l-1)}} = \left[\left(\sum_m \frac{\partial L}{\partial z_m^{(l+1)}} \underbrace{\frac{\partial z_m^{(l+1)}}{\partial a_k^{(l)}}}_{W_{mk}^{(l+1)}}\right) \underbrace{\frac{\partial a_k^{(l)}}{\partial z_k^{(l)}}}_{F'}\right] \underbrace{\frac{\partial z_k^{(l)}}{\partial W_{kj}^{(l)}}}_{a_j^{(l-1)}}$$

$$\Rightarrow \frac{\partial L}{\partial z_k^{(l)}} = \left(\sum_m \frac{\partial L}{\partial z_m^{(l+1)}} W_{mk}^{(l+1)}\right) F'(z_k^{(l)})$$

## Backpropagation Formulas

The backpropagation master formulas for the gradients of the loss function in layer-$l$ are given by

$$\boxed{\frac{\partial L}{\partial W_{kj}^{(l)}} = \delta_k^{(l)} a_k^{(l-1)}} \quad \text{and} \quad \boxed{\delta_k^{(l)} = \left(\sum_m \delta_m^{(l+1)} W_{mk}^{(l+1)}\right) F'(z_k^{(l)})}$$

, where the 'errors' in each layer are defined as $\delta_k^{(l)} = \frac{\partial L}{z_k^{(l)}}$

→ The only derivative one needs to calculate is F' !

# Backpropagation Algorithm

Given a training dataset $D = \{(x_i, t_i)\}$ we first run a *forward pass* to compute all the activations throughout the network, up to the the output layer. Then, one computes the network output "error" and backpropagates it to determine each neuron error contribution $\delta_k^{(l)}$

## Backpropagation Algorithm

1. Initialize the weights $W_{kj}^{(l)}$ randomly
2. Perform a feedforward pass, computing the arguments $z_k^{(l)}$ and activations $a_k^{(l)}$ for all layers
3. Determine the network output error: $\delta_i^{(n)} = \left[a_i^{(n)} - t_i\right] F'(z_i^{(n)})$
4. Backpropagate the output error: $\delta_k^{(l)} = \left(\sum_m \delta_m^{(l+1)} W_{mk}^{(l+1)}\right) F'(z_k^{(l)})$
5. Compute the loss gradients using the neurons error and activation: $\dfrac{\partial L}{\partial W_{kj}^{(l)}} = \delta_k^{(l)} a_k^{(l-1)}$
6. Update the weights according to the gradient descent: $\Delta W_{kj}^{(l)} = -\lambda \dfrac{\partial L}{\partial W_{kj}^{(l)}}$
7. Return to step (2)

So far we have focused on fully connected neural network, but this reasoning can be applied to other NN architectures (different neuron connectivity patterns)

# Evaluation of Learning Process

Split dataset into 3 independent parts , one for each learning phase

**Training**

Train(fit) the NN model by iterating through the training dataset ( an epoch )

- High learning rate will quickly decay the loss faster but can get stuck or bounce around chaotically
- Low learning rate gives very low convergence speed

**Validation**

Check performance on independent validation dataset and tune hyper-parameters

- Evaluate the loss over the validation dataset after each epoch
- Examine for overtraining(overfitting), and determine when to stop training

**Test**

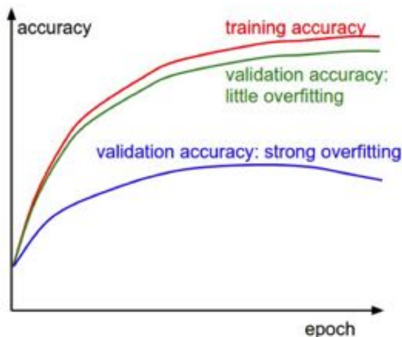
Final performance evaluation after finished training and hyper-parameters are fixed. Use the test dataset for an independent evaluation of performance obtaining a ROC curve

# Overtrainging(Overfitting)

## Overtraining(Overfitting)

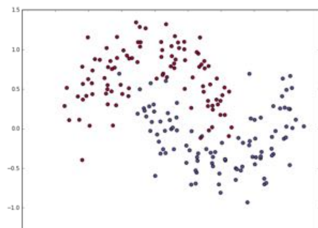
Gap between training and validation accuracy indicates the amount of overfitting

- If validation error curve shows small accuracy compared to training it indicates overfitting ⇒ add regularization or use more data.
- If validation accuracy tracks the training accuracy well, the model capacity is not high enough ⇒ use larger model



# Overfitting - Hidden Layers Size X Decision Boundary [16]

- Hidden layer of low dimensionality nicely captures the general trend of data.
- Higher dimensionalities are prone to overfitting (“memorizing” data) as opposed to fitting the general shape
- If evaluated on independent dataset (and you should !), the smaller hidden layer generalizes better
- Can counteract overfitting with regularization, but picking correct size for hidden layer is much simpler



Hidden Layer size 1

Hidden Layer size 2

Hidden Layer size 3

Hidden Layer size 4

Hidden Layer size 5

Hidden Layer size 20

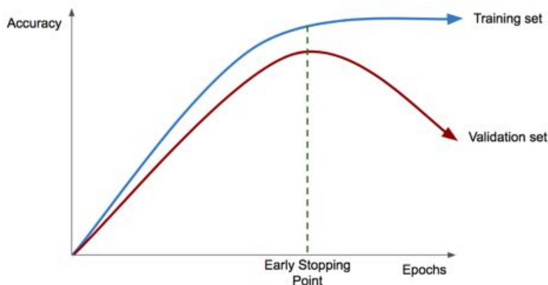
# Regularization

Regularization techniques prevents the neural network from overfitting

## Early Stopping

Early stopping can be viewed as regularization in time. Gradient descent will tend to learn more and more the dataset complexities as the number of iterations increases.

Early stopping is implemented by training just until performance on the validation set no longer improves or attained a satisfactory level. Improving the model fit to the training data comes at the expense of increased generalization error.

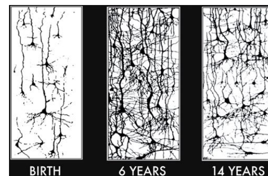


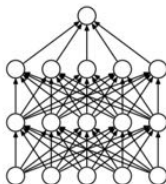
# Dropout Regularization

## Dropout Regularization

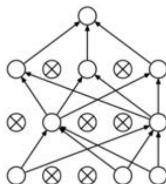
Regularization inside network that remove nodes randomly during training.

- It's ON in training and OFF in validation and testing
- Avoid co-adaptation on training data
- Essentially a large model averaging procedure

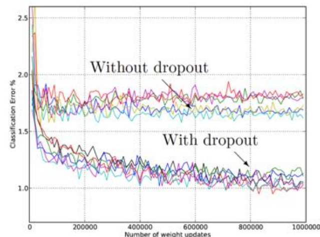


(a) Standard Neural Net



(b) After applying dropout.




Usually worsens the results during training, but improves validation and testing results !

# L1 & L2 Regularization

Between two models with the same predictive power, the 'simpler' one is to be preferred ( NN Occam's razor )

L1 and L2 regularizations add a term to the loss function that tames overfitting

$$L'(\vec{w}) = L(\vec{w}) + \alpha\Omega(\vec{w})$$

**L1 Regularization**

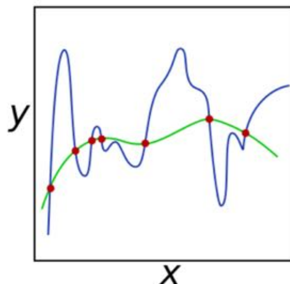
Works by keeping weights sparse

$$\Omega(\vec{w}) = |\vec{w}|$$

**L2 Regularization**

Works by penalising large weights

$$\Omega(\vec{w}) = |\vec{w}|^2$$




The combined $L1 + L2$ regularization is called Elastic

# Hyperparameter Optimization

Hyperparameters are NN parameters that don't get updated during learning ( $\#$ neurons , learning rate , weight decay ... ). Finding the optimal network architecture and hyperparameters values is another optimization task

## Hyperparameter Optimization Algorithms

- Grid search: systematically try out many configurations
- Random search: try out configurations randomly
- Bayesian optimization

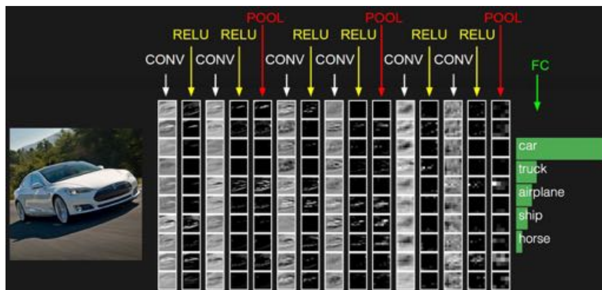
# Deep Learning

# Deep Neural Networks - Need for Depth

Universal approximation theorem says a single hidden layer is enough to learn any function. But as data complexity grows, one needs exponentially large hidden layer to capture all structure in data

## Deep Neural Networks(DNN)

DNN is a NN with more than one hidden layer. This allows it to factorize the data features, distributing its representation across the layers, exploring the compositional character of nature [17]




⇒ DNN allows a hierarchical representation of data features !

[17] Deep Learning , Y.LeCunn, J.Bengio, G.Hinton , Nature , vol. 521, pg. 436 , May 2015

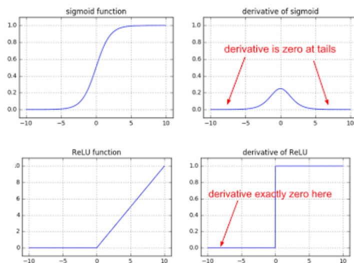
# Vanishing Gradient Problem

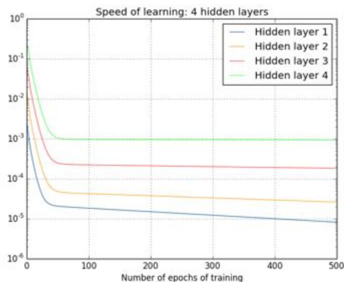
## Vanishing Gradient Problem [18]

Backpropagation computes gradients iteratively by multiplying the activation function derivate $F'$ through $n$ layers.

$$\delta_k^{(l)} = \left( \sum_m \delta_m^{(l+1)} W_{mk}^{(l+1)} \right) F'(z_k^{(l)})$$

For $\sigma(x)$ and $Tanh(x)$ the derivate $F'$ is asymptoticaly zero, so weights updates gets vanishing small when backpropagated $\Rightarrow$ Then, earlier layers learns much slower than later layers !!!



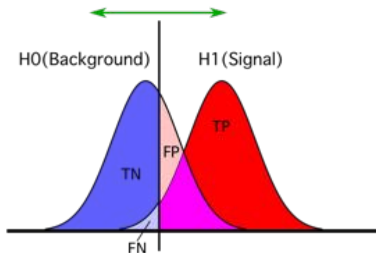



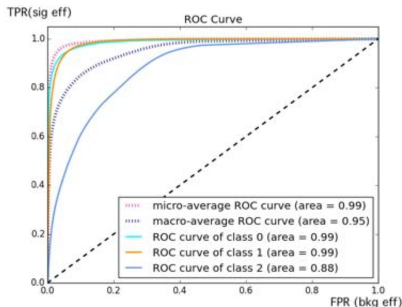

[18] http://neuralnetworksanddeeplearning.com/chap5.html

# Evaluating Classifier Perfomance: ROC

## ROC Curve

A Receiver Operating Characteristic ( **ROC** ) curve shows the discrimination of a binary classifier ( **TPR x FPR** ) as its discrimination threshold is varied

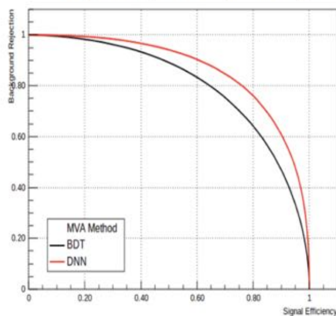


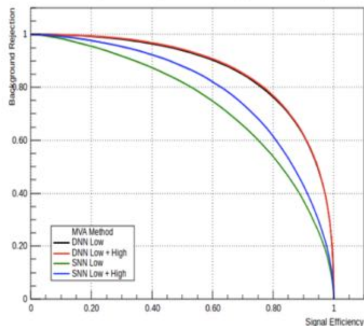

The Area Under the Curve ( **AUC** ) is used as a figure of merit

# DNN × BDT Perfomance

DNN × BDT (same input)




LOW × HIGH level (feature engineered) input [19]

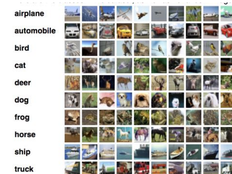


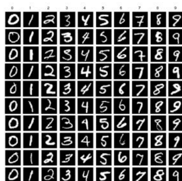

[19] https://arxiv.org/pdf/1402.4735.pdf

# Deep Learning

## Deep Learning Revolution

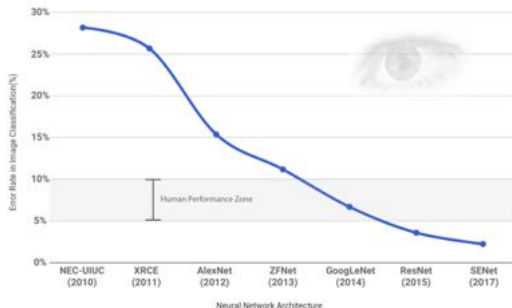
The main catalyst for the Deep Learning revolution has been the availability of new and large open source datasets for machine learning research [20]

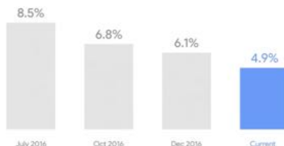


[20] https://en.wikipedia.org/wiki/List_of_datasets_for_machine-learning_research

# Image and Speech Recognition: DNN versus Humans




https://arxiv.org/pdf/1409.0575.pdf

# Deep Architectures and Applications

# Neural Network Zoo [21]

NN Architecture and nodes connectivity can be adapted for the problem at hand

[21] http://www.asimovinstitute.org/neural-network-zoo

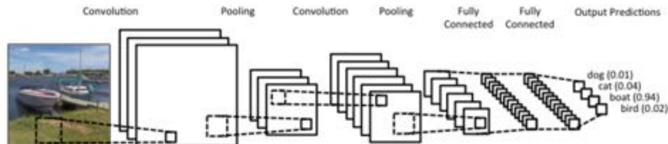
# Convolutional Neural Network

When dealing with high-dimensional inputs such as images, learning features with fully connected networks is computationally too expensive ( ex: 1000x1000 image has $O(10^6)$ pixels ).
Convolutional neural network [22] emulate the behavior of a visual cortex, where individual neurons respond to stimuli only in a restricted region of the visual field.

## Convolutional Neural Network (CNN or ConvNet)

A CNN mitigate the challenges of high dimensional inputs by restricting the connections between the input and hidden neurons. It conects only a small contiguous region of input nodes, exploiting local correlation. Its architecture is a stack of distinct and specialized layers:

1. Convolutional
2. Pooling (Downsampling)
3. Fully connected (MLP)




[22] http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution

# CNN - Convolutional Layer

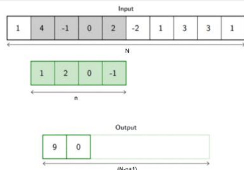
## Convolutional Layer

CNN convolutional layer [23] is its core building block and its parameters are learnable filters (kernels), that extracts image features corresponding to a small receptive field.

A convolution is like a sliding window transformation, where the window applies a filter to extract image features

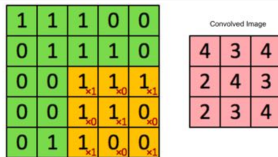
## 1D Discrete Convolution

$$y_i^{(l+1)} = \sum_{a=0}^{n-1} w_a x_{i+a}^{(l)}$$




## 2D Discrete Convolution

$$y_{ij}^{(l+1)} = \sum_{a=0}^{n-1} \sum_{b=0}^{m-1} w_{ab} x_{(i+a)(j+b)}^{(l)}$$




[23] http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution

# Convolution Filters

An image convolution [24] with filters designed to produce some effect ( sharpen, blurr ) or detect some image feature ( edges, texture )

| 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|
| 0 | 0 | -1 | 0 | 0 |
| 0 | -1 | 5 | -1 | 0 |
| 0 | 0 | -1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |

| 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |

| | | | | |
|---|---|---|---|---|
| | 0 | 1 | 0 | |
| | 1 | -4 | 1 | |
| | 0 | 1 | 0 | |
| | | | | |

| | | | | |
|---|---|---|---|---|
| | -2 | -1 | 0 | |
| | -1 | 1 | 1 | |
| | 0 | 1 | 2 | |
| | | | | |

[24] https://docs.gimp.org/2.6/en/plug-in-convmatrix.html

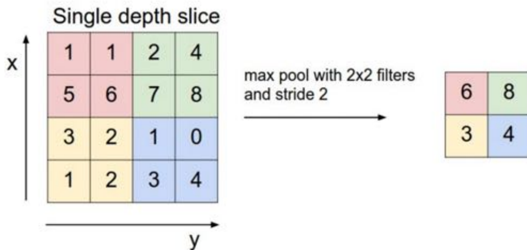
# CNN - Pooling Layer

## Pooling(Downsampling) Layer

The pooling (downsampling) layer [25] has no learning capabilities and serves a dual purpose:

- Decrease the representation size ⇒ reduce computation
- Make the representation approximately invariant to small input translations and rotations

Pooling layer partitions the input in non-overlapping regions and, for each sub-region, it outputs a single value (ex: max pooling, mean pooling)




[25] http://ufldl.stanford.edu/tutorial/supervised/Pooling

# CNN - Fully Connected Layers

## Fully Connected Layer

CNN chains together convolutional(filtering) , pooling (downsampling) and then fully connected(MLP) layers.

- After processing with convolutions and pooling, use fully connected layers for classification
- Architecture allows capturing local structure in convolutions, and long range structure in later stage convolutions and fully connected layers

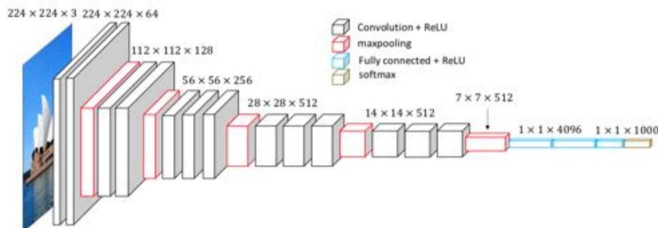
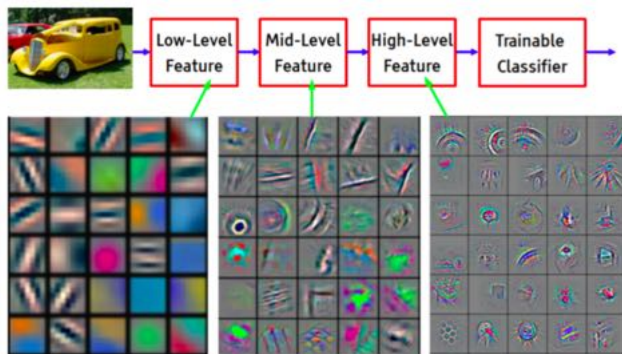



Figure 2: The architecture of VGG16 model .

# CNN Feature Visualization

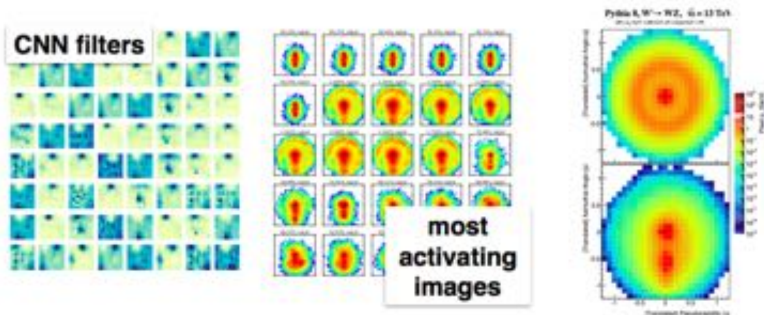
Each CNN layer is responsible for capturing a different level of features as can be see from from ImagiNet [26]




Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

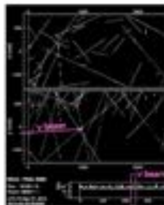
[26] https://arxiv.org/pdf/1311.2901

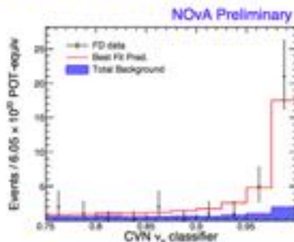
# CNN Application in HEP: Jet ID



# CNN Application HEP: Neutrino ID

## Neutrinos with convolutional nets

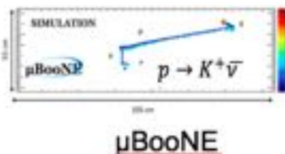


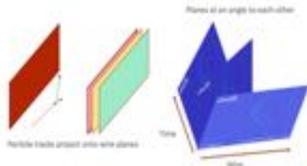



76% Purit
73% Effic

An equivalent increased exposure of 30%

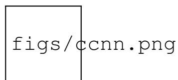
Aurisiano et al. 2016



μBooNE

# Capsule Convolution Neural Network (CCNN)



# Recurrent Neural Network(RNN)

Sequential data is a data stream (finite or infinite) which is interdependent (ex: text , speech and video). Feed forward networks can't learn any sort of correlation between previous and current input !

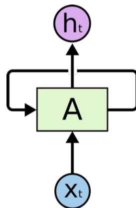
## Recurrent Neural Networks (RNN)

Recurrent neural networks are networks that use feedback loops to process sequential data. These feedbacks allows information to persist, which is an effect often described as memory.

## RNN Layers and Loss [27]

The hidden layer output depends not only on the current input, but also on the entire history of past inputs. The output layer is squashed by a softmax and error evaluated by a log loss

1. **Hidden Layer:** $\vec{h}^{[t]} = Tanh(\vec{W}_{xh}\vec{x}^{[t]} + \vec{W}_{hh}\vec{h}^{[t-1]} + b_h)$
2. **Output Layer:** $p^{[t]} = Softmax(\vec{W}_{hy}\vec{h}^{[t]} + b_y)$
3. **Loss Function:** $L = -log(p^{[t]})$



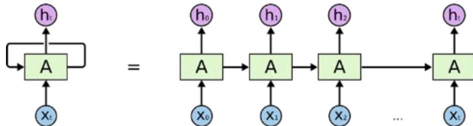

[27] https://eli.thegreenplace.net/2018/understanding-how-to-implement-a-character-based-rnn-

# Recurrent Neural Network(RNN)

RNNs [28] process an input sequence one element at a time, maintaining in their hidden units a 'state vector' that contains information about all the past elements history.

## RNN Unrolling

A RNN can be thought of as multiple copies of the same network, each passing a message to a successor. Unrolling is a visualization tool which views a RNN hidden layer as a sequence of layers that you train one after another with backpropagation.




## Backpropagation Through Time (BPTT)

Backpropagation through time is just a fancy buzz word for backpropagation on an unrolled RNN. The error is back-propagated from the last to the first timestep

RNNs unfolded in time can be seen as very deep networks $\Rightarrow$ difficult to train !

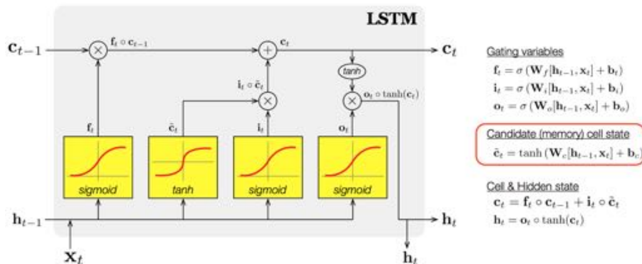
[28] http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-

# Long Short Term Memory(LSTM)

## Long Short-Term Memory (LSTM) Network

LSTM [29] networks are a special RNNs, capable of learning long-term dependencies.It categorize data into **short** and **long** term memory cells, deciding its importance and what to remember or forgot.

The LSTM unit cell has three gates: **input**, **forget** and **output**. The **input** defines how much of the newly computed state for the current input is accepted. The **forget** defines how much of the previous state is accepted. Finally, the **output** defines how much of the internal state is expose to the external network (other layers and the next time step)



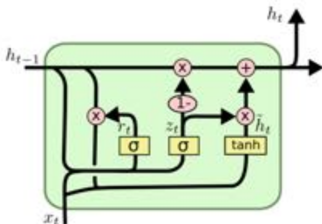

[29] http://colah.github.io/posts/2015-08-Understanding-LSTMs

# Gated Recurrent Network (GRU)

## Gated Recurrent Network (GRU)

GRU [30] networks have been proposed as a simplified version of LSTM, which also avoids the vanishing gradient problem and is even easier to train.

In a GRU the **reset** gate determines how to combine the new input with the previous memory, and the **update** gate defines how much of the previous memory is keept




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

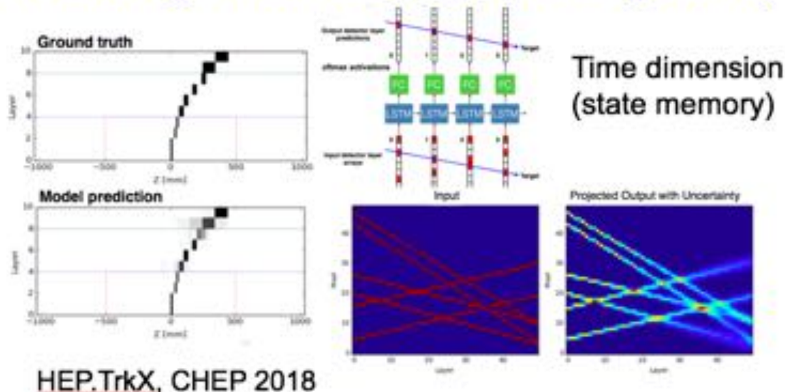
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

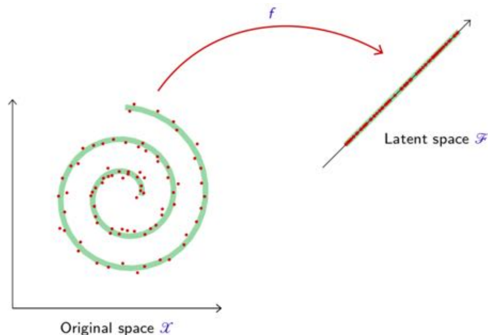
[30] https://isaacchanghau.github.io/post/lstm-gru-formula

# LSTM Application in HEP: Tracking



# Autoencoder

Many applications such as data compression, denoising and data generation require to go beyond classification and regression problems. This modeling usually consists of finding “meaningful degrees of freedom”, that can describe high dimensional data in terms of a smaller dimensional representation



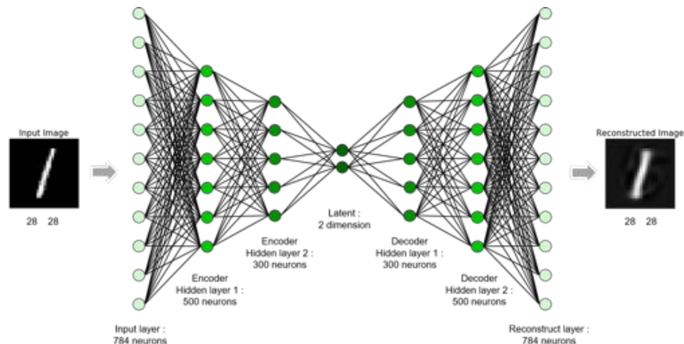

Traditionally, autoencoders were used for dimensionality reduction and denoising. Recently, autoencoders are being used also in generative modeling

# Autoencoder

## Autoencoder(AE)

An Autoencoder(AE) is a neural network that is trained to attempt to copy its input to its output in an **unsupervised way**. In doing so, it learns a representation(encoding) of the data set features $l$ in a low dimensional latent space.

It may be viewed as consisting of two parts: an encoder $l = f(x)$ and a decoder $y = g(l)$.

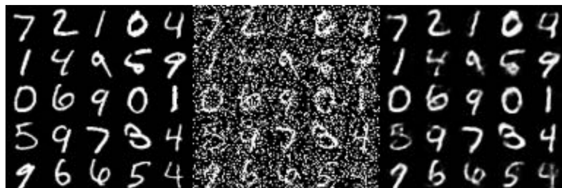



$\Rightarrow$ Understanding is data reduction

# Denoising Autoencoder (DAE)

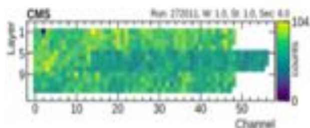
## Denoising Autoencoder (DAE)

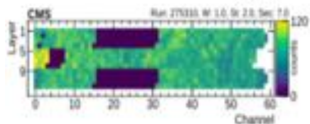
The DAE is an extension of a classical autoencoder where one corrupts the original image on purpose by adding random noise to it's input. The autoencoder is trained to reconstruct the input from a corrupted version of it and then used as a tool for noise extraction

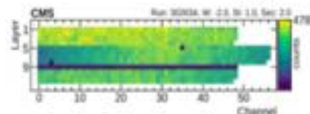


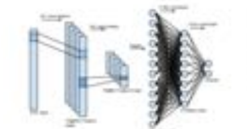
# Autoencoder Application in HEP: DQM

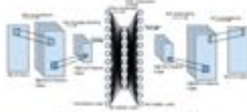
Monitoring the data taking to spot failures ( anomalies ) in the detector systems

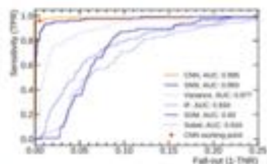


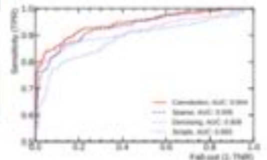



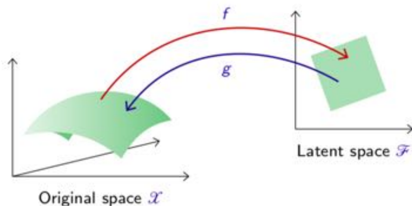




# Autoencoder (AE)

An autoencoder combines an encoder $f$ from the original space $\mathscr{X}$ to a latent space $\mathscr{F}$, and a decoder $g$ to map back to $\mathscr{X}$, such that the composite map $g \circ f$ is close to the identity when evaluated on data.




**Autoencoder Loss Function**

$$L = \| X - f \circ g(X) \|^2$$

- An AE becomes useful by having a latent hidden layer smaller than the input layer, forcing it to create a compressed representation of the data by learning correlations.
- AE layers can be feed forward, convolutional or even recurrent, depending on the application
- One can train an AE on images and save the encoded vector to reconstruct (generate) it later by passing it thrugh the decoder. The problem is that two images of the same number (ex: 2 written by different people) could end up far away in latent space !

# Variational Autoencoder(VAE)

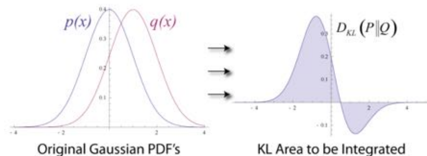
## Variational Autoencoder(VAE)

A Variational Autoencoder(VAE) [31] is an autoencoder with a loss function penalty (KL divergence) that forces it to generate latent vectors that follows a unit gaussian distribution. To generate images with a VAE one just samples a latent vector from a unit gaussian and pass it through the decoder.

The Kullback–Leibler(KL) divergence, or relative entropy, is a measure of how one probability distribution differs from a second, reference distribution

## Kullback–Leibler Divergence

$$D_{KL}(p|q) = \int_{-\infty}^{+\infty} dx \; p(x) \; log\left(\frac{p(x)}{q(x)}\right)$$




[31] http://kvfrans.com/variational-autoencoders-explained

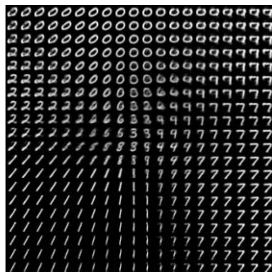
# Variational Autoencoder(VAE)

## VAE Loss

The VAE loss [32] function is composed of a mean squared error (generative) loss that measures the reconstruction accuracy, and a KL divergence (latent) loss that measures how close the latent variables match a gaussian.

$$L = \| \, x \; - \; f \circ g(x) \, \|^2 + D_{KL}(p(z|x)|q(z|x))$$

Bellow we have an example of a set of a VAE generated numbers obtained by gaussian sampling a 2D latent space

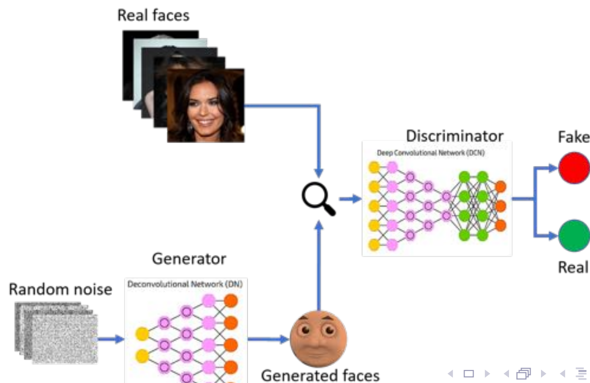


[32] https://tiao.io/post/tutorial-on-variational-autoencoders-with-a-concise-keras-implement

# Generative Adversarial Network(GAN)

## Generative Adversarial Networks (GAN)

Generative Adversarial Networks (GANs) are composed by two neural networks, where one generates candidates (generative), while the other classifies them (discriminative). Typically, the generative network learns a map from a latent space to data, while the discriminative network discriminates between generated and real data.

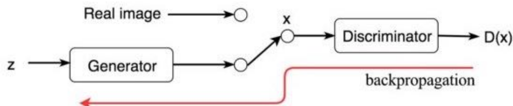


# Generative Adversarial Network(GAN)

The GAN adversarial training [33] works by the two neural networks competing and training each other. The generator tries to ”fool”the discriminator, while the discriminator keeps trying to uncover the fake data.

## GAN Training

1. The discriminator receives as input samples synthesized by the generator and real data . It is trained just like a classifier, so if the input is real, we want output=1 and if it's generated, output=0
2. The generator is seeded with a randomized input that is sampled from a predefined latent space (ex: multivariate normal distribution)
3. We train the generator by backpropagating this target value all the way back to the generator
4. Both networks are trained in alternating steps and in competition to improve themselves




[33] https://medium.com/@jonathan_hui/gan-whats-generative-adversarial-networks-and-its-appli

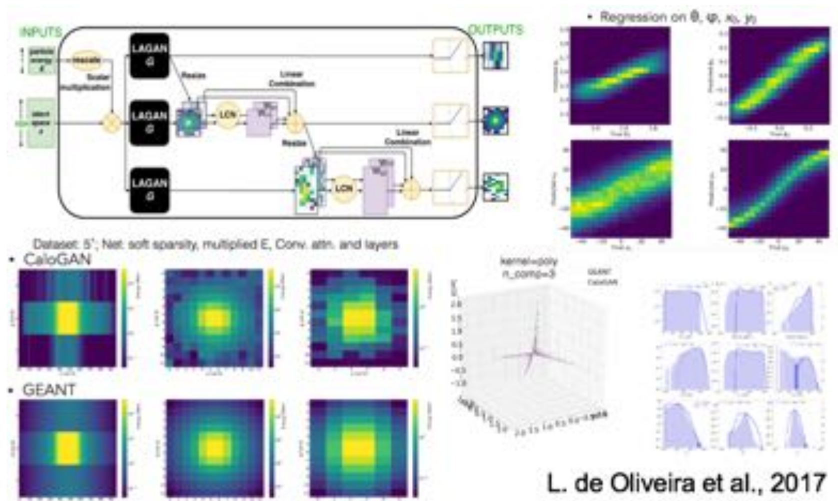
# Generative Adversarial Network(GAN)

GANs are quite good on faking celebrities images and Monet style painitings !

Training Data
(CelebA)

Sample Generator
(Karras et al, 2017)

# GAN Application in HEP: MC Simulation



# Aditional Topics:

- Hyperparameter Optimization: Grid search , Random search and Bayesian Optimization ...
- New architectures: Capsule Networks , Graph Neural Networks , Brain reverse engineering ...
- New computing frameworks: GPU , TPU , FPGA ( Amazon EC2 )
- What's the NN learning ? Debuging a CNN ?
  - Which pixels of a picture most contributed to a prediction?
  - Which pixels were in contradiction to prediction?
  - "Sensitivity analysis and heatmaps (based on "input gradient" df/dxi).

# THE END