Introduction to Machine Learning in (particle) physics

SNS - SCIENTIFIC DATA ANALYSIS SCHOOL

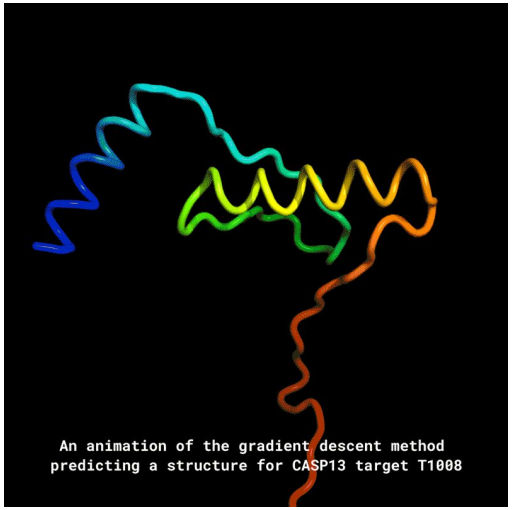
Andrea.Rizzi@unipi.it - 28/11/2019

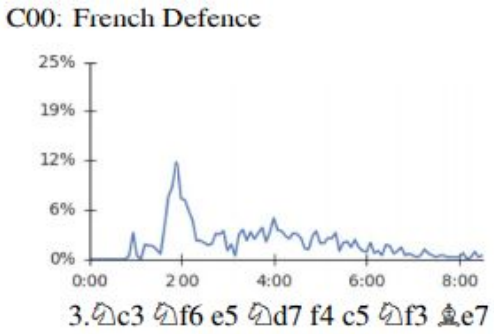
Why machine learning?

- See yesterday’s talk!
  - Big data
  - Powerful algorithms

My favorite performance examples

- Learning how to translate **without seeing a single translation example**, just having two independent monolingual corpora (https://arxiv.org/abs/1711.00043)
- Alpha-Fold: contest to predict protein folding, alphafold ranked first with 25 correct predictions out of 43 tests. The second ranked reached 3 out of 43.
- AlphaGo => AlphaZero: AlphaGo beat humans at “Go”, learning from human matches and know-how. Then AlphaZero learned from scratch. AlphaZero beat AlphaGo 100-0
- AlphaZero learned chess too, and beat the best existing chess program
- AI recently proved math theorems, 1200 of them
- Microsoft and Alibaba AIs beated humans in text understanding test (SQuAD)
- DeepFake: never ever believe what you see on a screen, even in videos





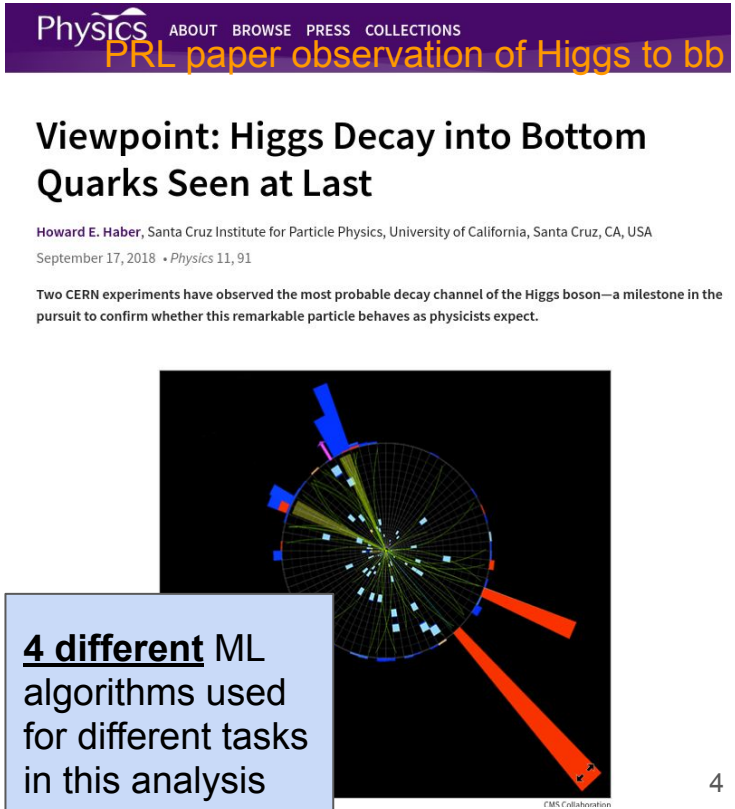
Machine learning is a key element of HEP analysis

examples are everywhere..

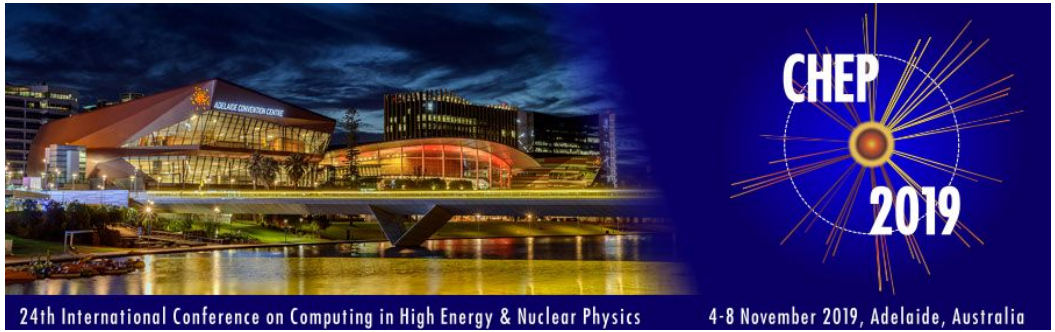
- Particle identification and kinematic measurement
- Signal to background discrimination (BDT and DNN are very popular in HEP experiments)
- Data quality anomaly detections
- Job processing optimization

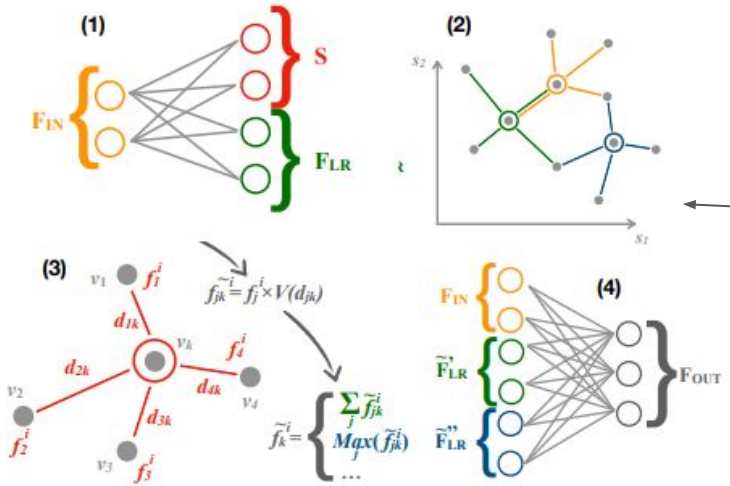
More to come:

- Reconstruction of charged particle trajectories (aka tracking)
- ...more applications…



Examples from CHEP





- Machine learning for QCD theory and data analysis
- BESIII drift chamber tracking with machine learning
- FPGA-accelerated machine learning inference as a service for particle physics computing
- Constraining effective field theories with machine learning
- Fast simulation methods in ATLAS: from classical to generative models
- Using ML to Speed Up New and Upgrade Detector Studies
- The Tracking Machine Learning Challenge
- *Particle Reconstruction with Graph Networks for irregular detector geometries*
- ...42 contribution with “Machine Learning” in the title/abstract

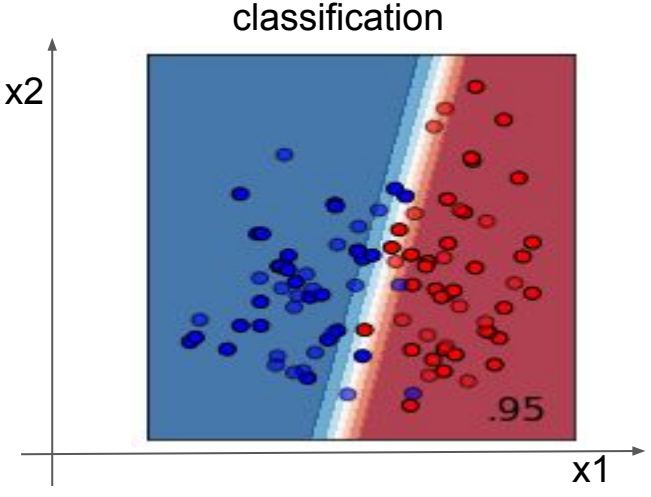
ML basics

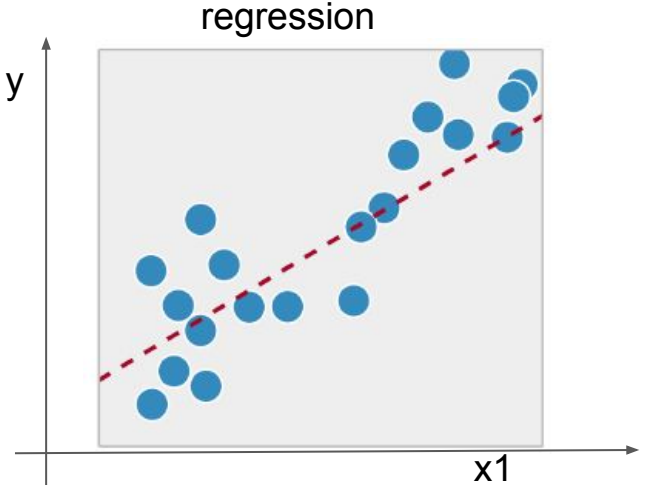
Types of typical ML problems

- **Classification:** which category a given input belong to.
- **Regression:** value of a real variable given the input.
- **Clustering:** group similar samples
- **Anomaly detection:** identify inputs that are different from others
- **Generation/synthesis of samples:** produce new samples, similar to the original data, starting from noise/random numbers
- **Denoising:** remove noise from an input dataset
- **Transcriptions:** describe in some language the input data
- **Translations:** translate between languages
- **Encoding and decoding:** transform input data to a different representation
- ...many more...

Function approximation

- The goal of a ML algorithm is to approximate an unknown function (typically the Probability Density Function of the data) given some example data
- The function is typically $f: R^n \rightarrow R^m$ (often $m=1$)
  - In **classification** we try to approximate the probability for each example, with inputs represented as a vector $\boldsymbol{x}$ to belong to a given category (y) (e.g. the probability to be a LHC Higgs signal event vs a Standard Model background one)
  - In **regression** we approximate the function that given the inputs ($\boldsymbol{x}$) returns the value of the variable to predict (y)





Model

- A model for the functions that can be used to approximate the PDF must be specified. The model can be simple (e.g. sum of polynomials up to degree **N**) or complex (e.g. all the functions that could be coded in **M** lines of C++)
- Different ML techniques are based on different “models”
  - Each technique further allow to specify the exact model
  - The parameters describing the exact model are called “hyper-parameters” (e.g. the degree **N** of the polynomial, or the maximum number of C++ line **M** can be considered hyper parameters)
- We will see example of models with different complexity:
  - Linear regression
  - Decision trees
  - **Artificial Neural Networks**

Parameters

- A specific model typically have parameters (e.g. the coefficient of the polynomials or the characters of the 10 lines of C++).
- Parameters are learned in the “training phase”.
- Different models or similar model with different hyper-parameters settings have different *n.d.o.f.* in the parameters phase space

Objective function

- A goal for what is “a good approximation” have to be defined
- This is called objective function (or **loss function** or error function …)
- Is a function that returns higher(or lower) value depending how good or bad the approximation is
  - *Loss* functions have to be *minimized*
- Example functions
  - Classification problems: binary cross entropy
  - Regression problems: Mean Square Error (i.e. the chi2 with sigma=1, I hope you are not surprised by this choice!)

The process is not very different from a typical phys-lab1 chi2 fit… but the number of parameters can be several orders of magnitude larger (10^3 to 10^6)

Objective function: binary cross entropy

- In classification problems the function to approximate is typically R^n -> [0,1]
  - Where, for example, 0 means background and 1 means signal
- The binary cross entropy is defined as follows:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

- The above function has large negative value when an example with y=1 is classifed with a p ~ 0 and no loss when p ~ 1
  - Viceversa if y=0, p ~ 1 has large loss and p ~ 0 has no loss
- Minimizing the binary cross-entropy we maximize the likelihood in a process with 0 or 1 outcome:

$$L = \prod_i p_i^{y_i}(1 - p_i)^{1-y_i}$$

$$-log(L) = -log(\prod_i p_i^{y_i}(1 - p_i)^{1-y_i}) = -\sum_i [y_i log(p_i) + (1 - y_i) log(1 - p_i)]$$

Learning / Training

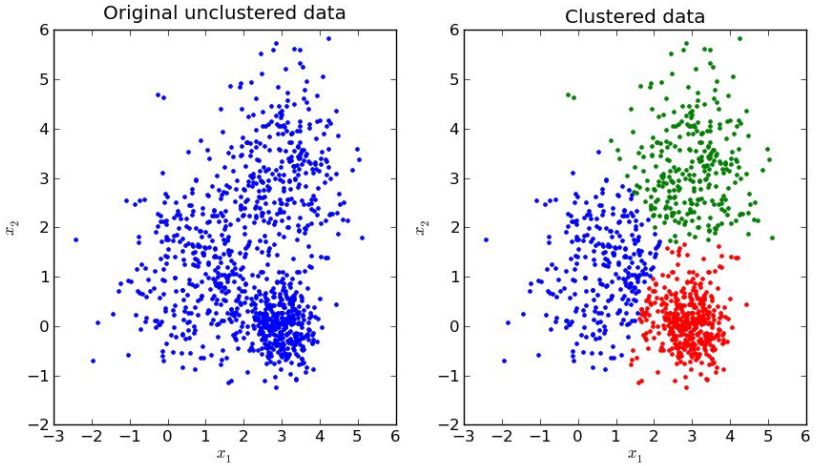
- For a given model, and given set of hyper-parameters, how do we infer the parameters that minimize the objective function?
- The idea of ML is to get the parameters from “data” in a so called “training” step
- Each ML technique has a different approach to training
- Different types of training
  - **Supervised**: i.e. for each example we know the correct answer
  - **Unsupervised**: we do not know “what is what”, we ask the ML algorithm to learn the probability density function of the examples in the features phase space
  - **Reinforcement learning**: have agents playing a punishment/reward game

Supervised learning

- We want to teach something we (the supervisors) already know (at least on the training samples)
- For each example we need to have the “right answer” / “truth” , for example:
  - Labels telling if a given example **signal** or **background**
  - Labels classifying the content of an image (multiple labels are possible)
  - Correct values of some quantity, e.g. generated energy of a particle
- Sample can be labelled in various ways:
  - Humans labelling existing data
  - Data being “generated” from known functions (e.g. simulations)
- Learn the probability of the label y, given the input x, i.e. $P(y \mid x)$

Unsupervised learning

- Often we do not have labels (or we have labels only for few data points)
- Unsupervised learning techniques allow to train networks that can perform similar tasks as the supervised ones, e.g.
  - Classification of “common” patterns
  - Dimensionality reduction, compression
  - Prediction of missing inputs
  - Anomaly detection



- In practice learn the Probability Density Function of the data, independently of any “label” variable, i.e. P($\boldsymbol{x}$)

Supervised vs unsupervised

Supervised and unsupervised are not as different as one would imagine, in fact

- P($\boldsymbol{x}$) can be seen as *n* supervised problems, one for each feature

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(\mathrm{x}_i \mid \mathrm{x}_1, \ldots, \mathrm{x}_{i-1})$$

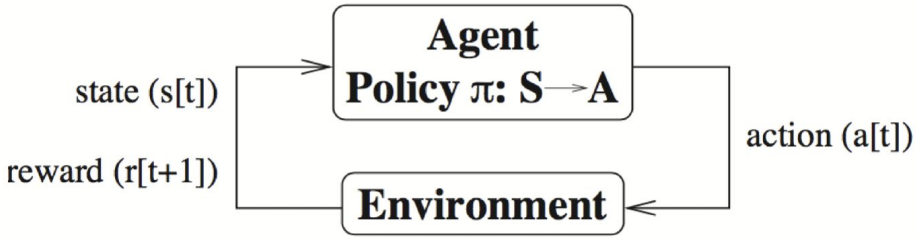
- P($y \mid \boldsymbol{x}$) can also be computed, if we treat y as an "$\boldsymbol{x}$" in unsupervised learning deriving hence $p(\mathbf{x}, y)$, as

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y} p(\mathbf{x}, y')}$$

Reinforcement learning

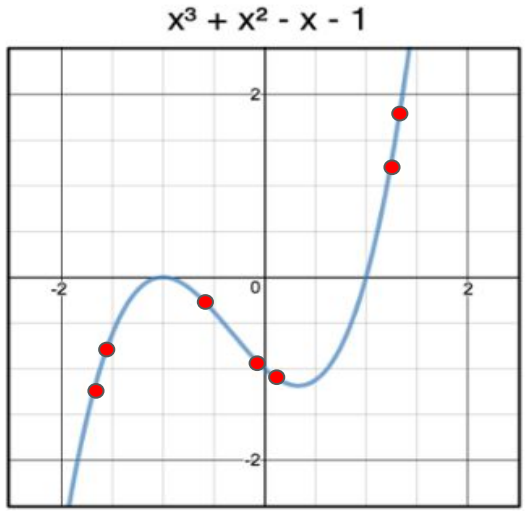
Applies to “agents” acting in an “environment” that updates their state

- It is similar to supervised learning as a “reward” has to be calculated
- The *supervisor* anyhow doesn’t necessarily know what is the best action to perform in a given state to interact with the environment, it just computes the reward
- Learn to make best decision in a given situation
  - The right move in chess or go match
  - Drive a car in the traffic
  - Etc..



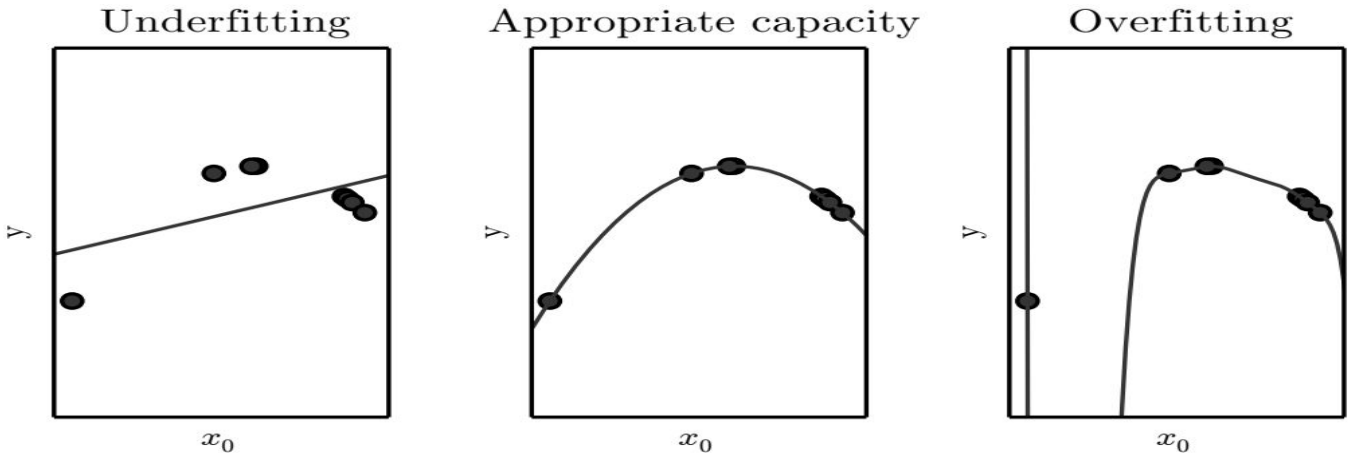
Capacity and representational power

- Different models (i.e. techniques/hyper-parameters values) allow to represent different type of functions
- Models with more free parameters typically can approximate a larger number of functions => **higher capacity**
- Remember: we do not know the actual function to aproximate, we just want to **learn from examples**
- With limited samples we have a tradeoff to handle:
  - accuracy in representation **vs** generalization of the results



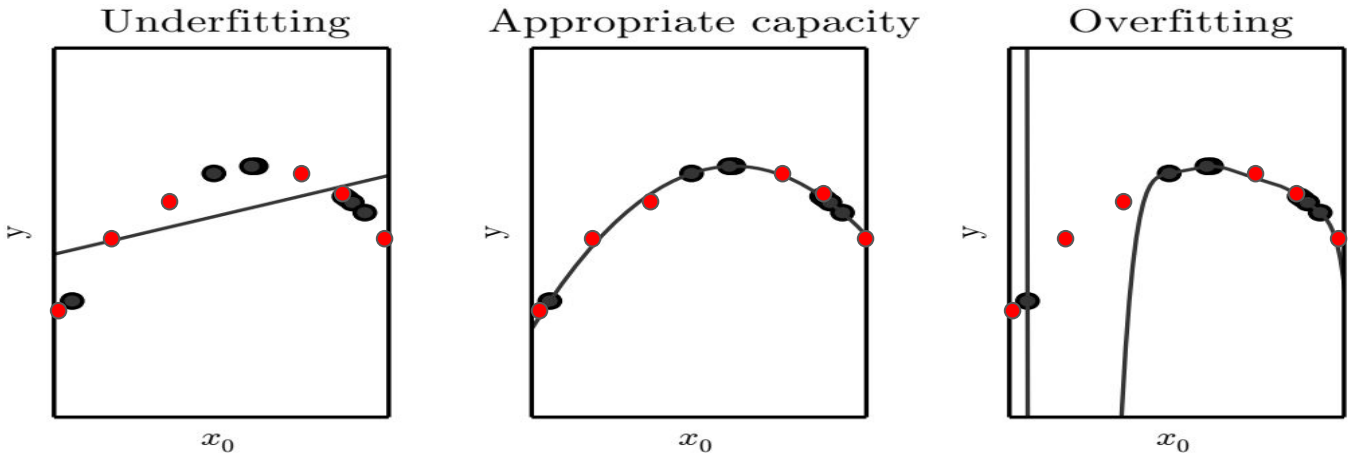
Capacity and representational power

- Underfitting: the sample is badly represented
- Overfitting / Appropriate capacity are less obvious to define
    - Lack of “generalization” -> overfitting



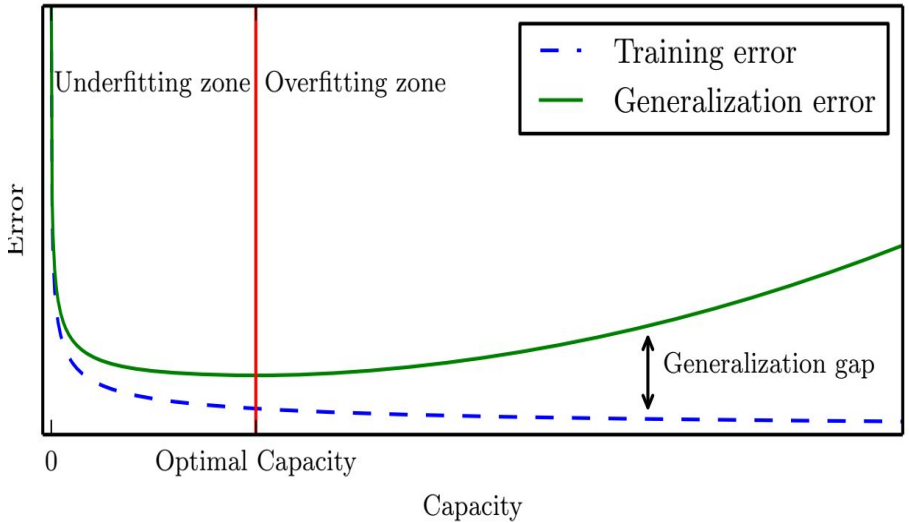
Capacity and representational power

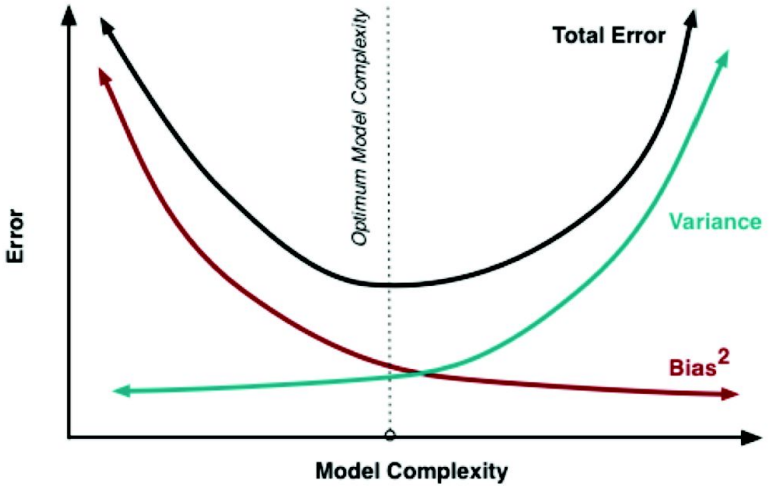
- Underfitting: the sample is badly represented
- Overfitting / Appropriate capacity are less obvious to define
    - Lack of “generalization” -> overfitting
    - Typical method is to check on independent sample
        - Or just split your sample in two and use only half for training



Generalization

- We can compare the accuracy between the “training” sample and the “generalization/validation” sample





- Bias/variance trade-off
  - y: function (with random noise)
  - h(x): approximated function

$$E[(y - h(x))^2] = E[(y - \bar{y})^2] + (\bar{y} - \bar{h}(x))^2 + E[(h(x) - \bar{h}(x))^2]$$

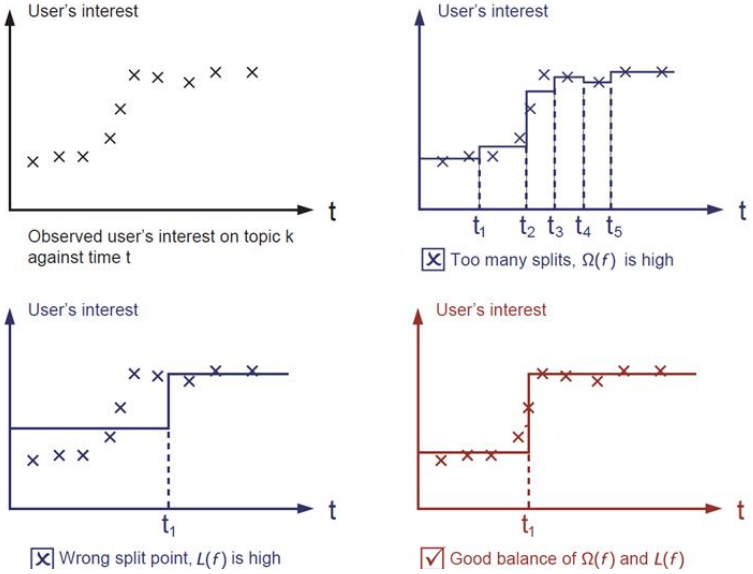
Noise — Bias Squared — Variance

Regularization

In order to control the “generalization gap”

- the objective function can be modified adding a regularization term
  - Introduce a “cost” in increasing the capacity of the model or in accessing some parts of the model-parameters space
- the examples in training dataset can be increased with augmentation techniques
  - Adding stochastic noise to existing examples
  - Transforming the existing examples with transformation that are known to be invariant for the solution we look for

$$\text{obj}(\theta) = L(\theta) + \Omega(\theta)$$



https://xgboost.readthedocs.io/en/latest/tutorials/model.html

Hyperparameters(model) optimization

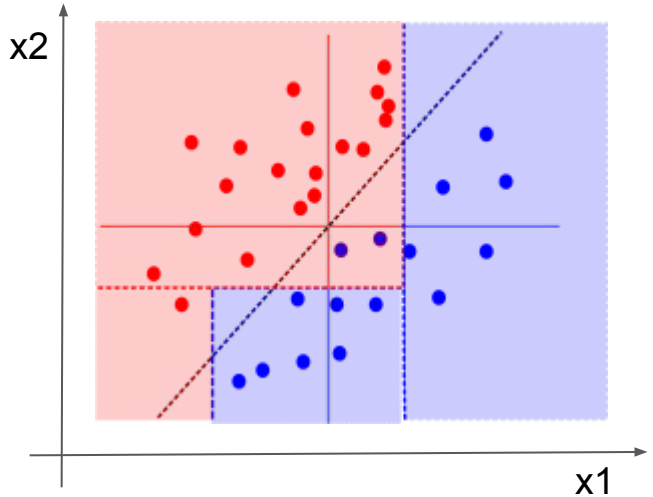
- It is normal to have to test a few, if not several, configurations in the model hyper-parameter space
  - Scans of hyper-parameters are often performed
  - Different techniques used
- Effectively a “second” minimization is done
  - First minimization is on the parameter => minimize on the “training dataset”
  - Second minimization is on the hyper-parameters => minimize on the “validation dataset”
- A third dataset (“test dataset”) is then also needed
  - To assess the performance of the algorithm in an unbiased way
  - To make an unbiased prediction of the algorithm output
- Original dataset is typically split in uneven parts to be used as *training, validation* and *test*

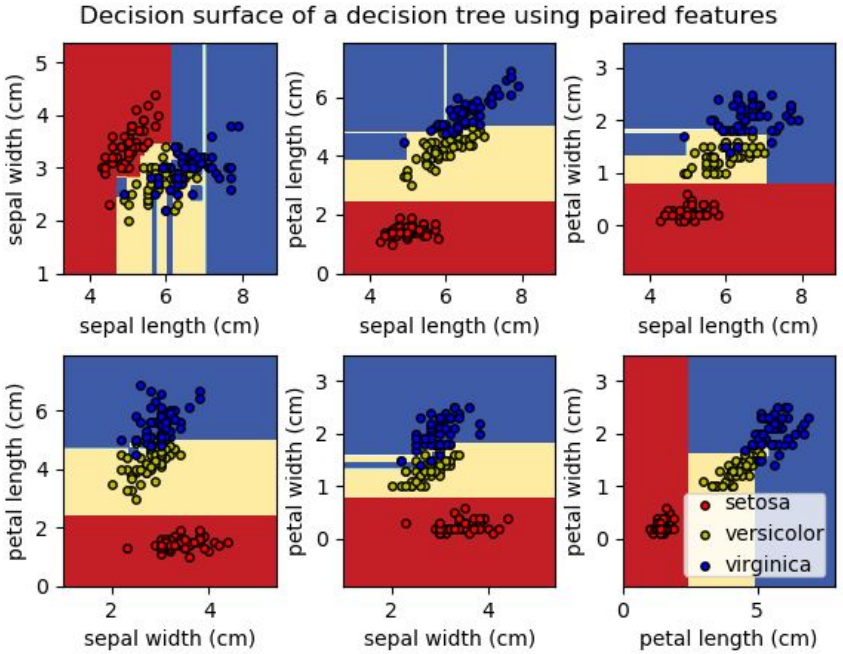
Inference

- A ML model that has been trained can than be used to act on some new data (or on the test dataset if a prediction has to be made)
- The evaluation of the algorithm output on the “unseen” data is called *inference*
- From a computing point of view *inference* is usually faster than *training*

Limitations of decision trees

- Cuts are axis aligned
- Classification of x1 > x2 is a a hard problem for a decision tree

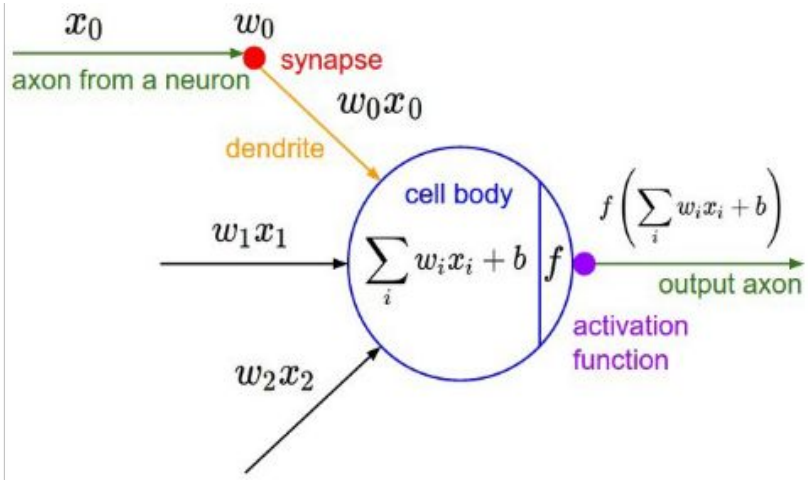


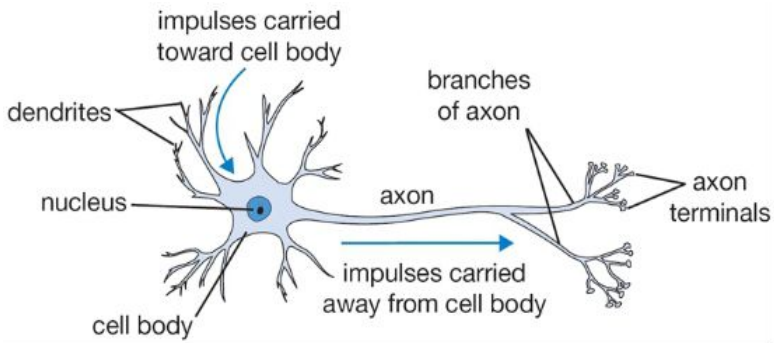


Artificial Neural Networks

(Artificial) neural networks

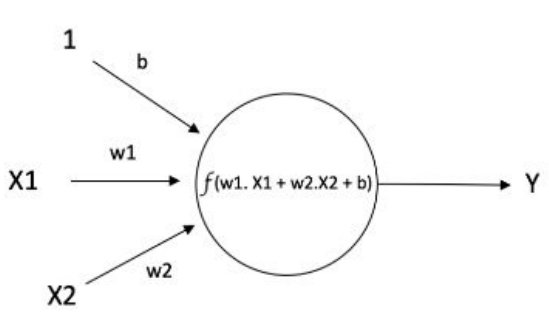
- Computation achieved with a network of elementary computing units (neurons)
- Each basic units, a neuron, has:
  - **Weighted** input connections to other neurons
  - A **non linear** activation function
  - An output value to pass to other neurons
- Biologically inspired to brain structure as a network of neuron
  - But artificial NN goal is not that of “simulating” a brain!



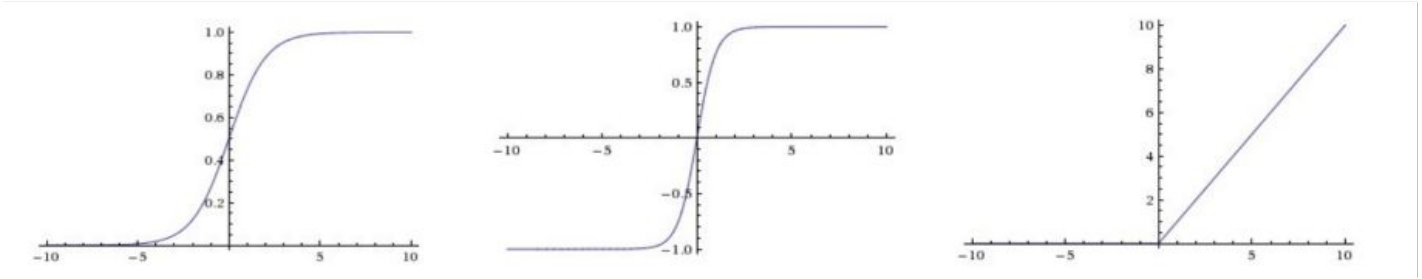


A neural network node: the artificial neuron

- The elementary processing unit, a neuron, can be seen as a node in a directed graph
- Inputs are **summed**, with **weights**, and an **activation function** is evaluated on such sum
- Nodes are typically also connected to an input “bias node” that has a fixed output value of 1
- Different activation functions can be used, common ones are: sigmoid, atan, relu (rectified linear unit)

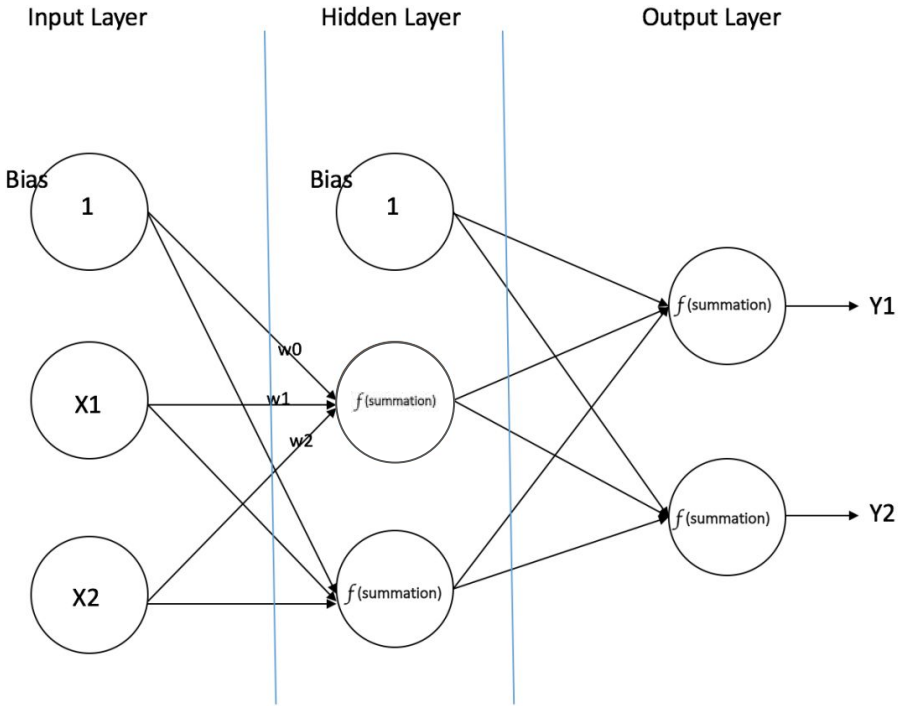


$$Y = f(w1.\,X1 + w2.X2 + b)$$



The MLP model

- The most common NN in the ‘90 was the Multi Layer Perceptron (MLP)
- Graph structure organized in “layers”
  - Input layer (nodes filled with input value)
  - Hidden layer
  - Output layer (node(s) where output is read out)
- Nodes are connected **only from one layer to the next** and **all possible connections** are present (known as “dense” or “fully connected” layer)
  - No intra-layer connections
  - No direct connections from input to output
- Size of input and output layers are fixed by the problem
- **Hyperparameters** are
  - The size of the hidden layers
  - The type of activation function
- The **parameters** to learn are the weights of the connections



Universal approximation theorem

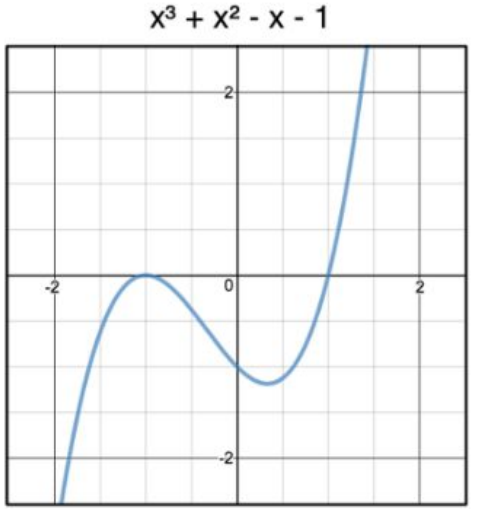
“One hidden layer is enough to *represent* (not *learn*) an approximation of any function to an arbitrary degree of accuracy” (I. Goodfellow et al. 2016)

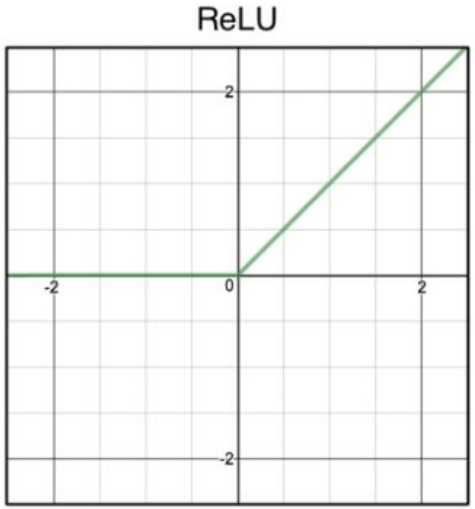
- You can approximate any function with arbitrary precision having **enough hidden nodes** and **the right weights**
- How do you get the right weights? You need a “training” for your network
  - The theorem does not say that one hidden layer (+ some training algorithm) is enough to find the optimal weights, just that they exists!
- Achieving some (even modest with some metric) level of accuracy may need an unmanageable hidden layer size
  - And may need an unreasonable number of “examples” to learn from

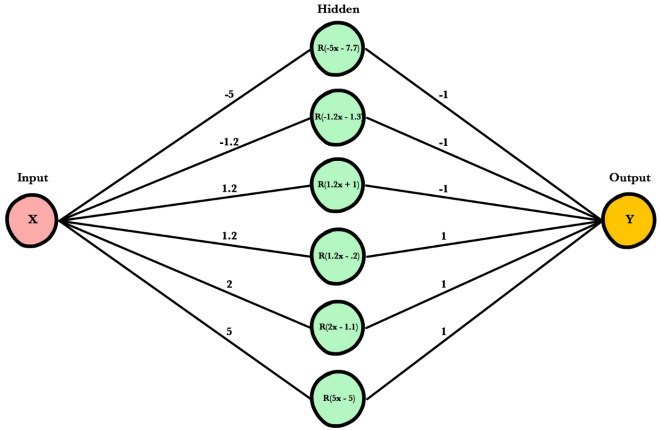
Example (1-D input)

Approximate this function

With a weighted sum of functions like this one

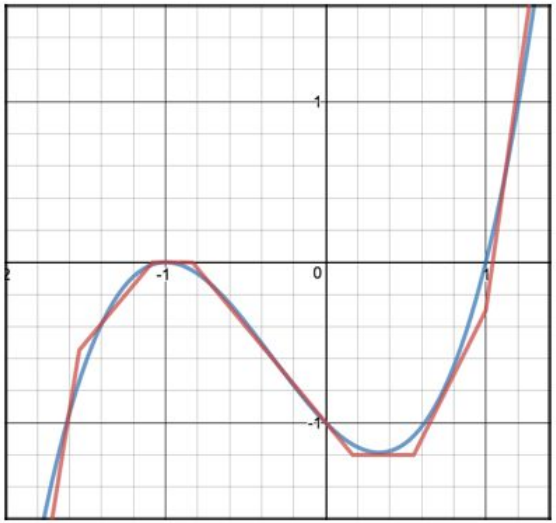






Example

- The Universal Approximation Theorem says that increasing #nodes I can increase the accuracy as much as I want
- More hidden nodes, higher “capacity” => more accuracy



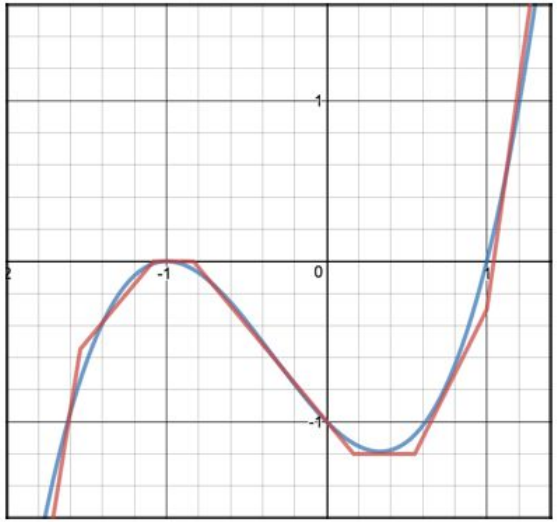
$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$

https://towardsdatascience.com/can-neural-networks-really-learn-any-function-65e106617fc6

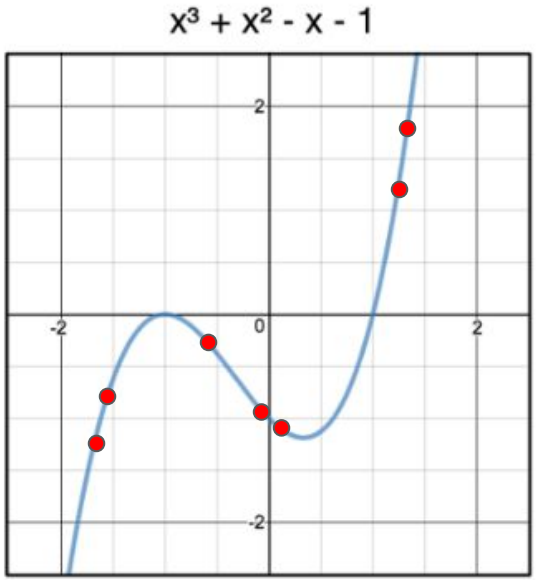
Training of an MLP

- How do I get the weights?
- Especially if I have only a few samples?

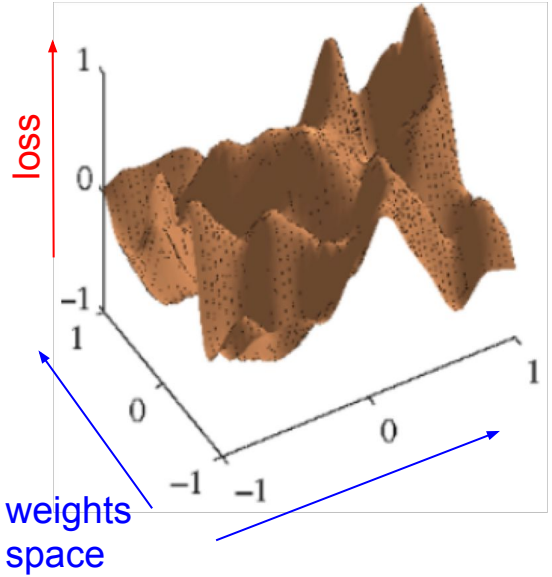


$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$



Training a NN

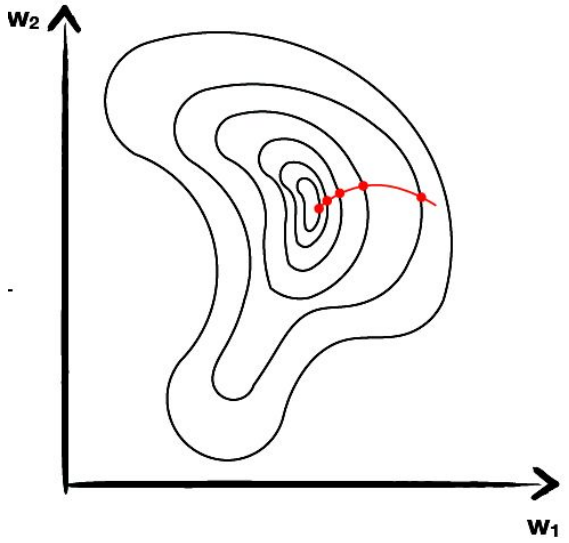


- The goal of training is to minimize the objective function (possibly both on the training and validation sample)
  - I.e. we want to minimize the loss as a function of the model parameters (i.e. the weights)
- For a MLP the basic idea is the following
  a. Start with random weights
  b. Compute the prediction for a given input ***x*** and check the difference with target *y* and the loss (repeat for a few example, aka “one batch”)
  c. Estimate an update for the weights that **reduces the loss**
  d. Iterate from point (b), repeating for all samples
  e. When the sample has been used completely (end of an epoch), iterate from (b) again on all samples
  f. Repeat for multiple epochs
- The important point is how to implement point (c) => (stochastic) gradient descent

How to find a minimum?

Gradient Descent

- We know the loss function value in a point in the weights phase space (e.g. the initial set of random weights, or the iteration N-1), computed numerically as the mean or the sum of the losses for each (or only some) of our training examples
- We can compute the gradient of the loss function in that point, we expect the minimum on “the way down” hence we adjust our set of weights doing a “step” in the direction pointed by the gradient with a step size that is proportional to the length of the gradient

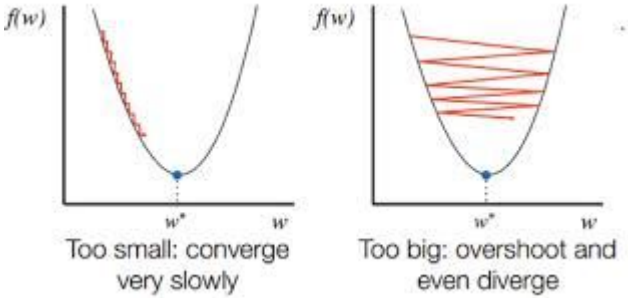


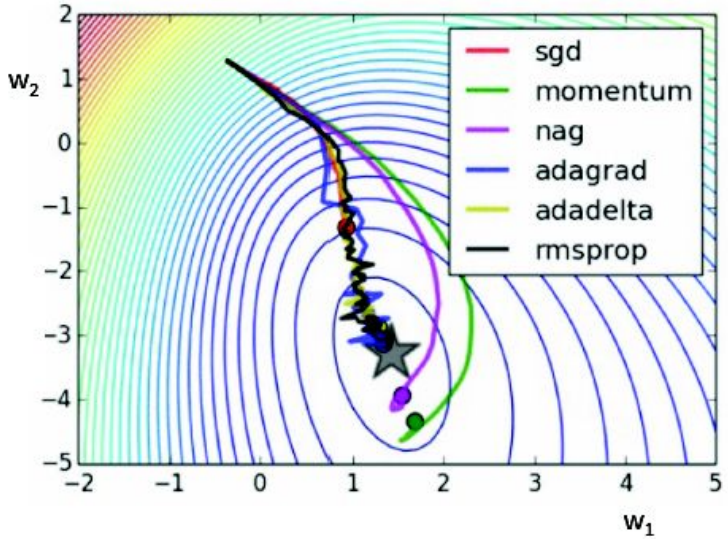
Stochastic Gradient Descent (SGD):

- Compute the gradient on “**batches**” of events rather than full sample
- The “noise” may help avoiding local minima

Not as simple as you would imagine

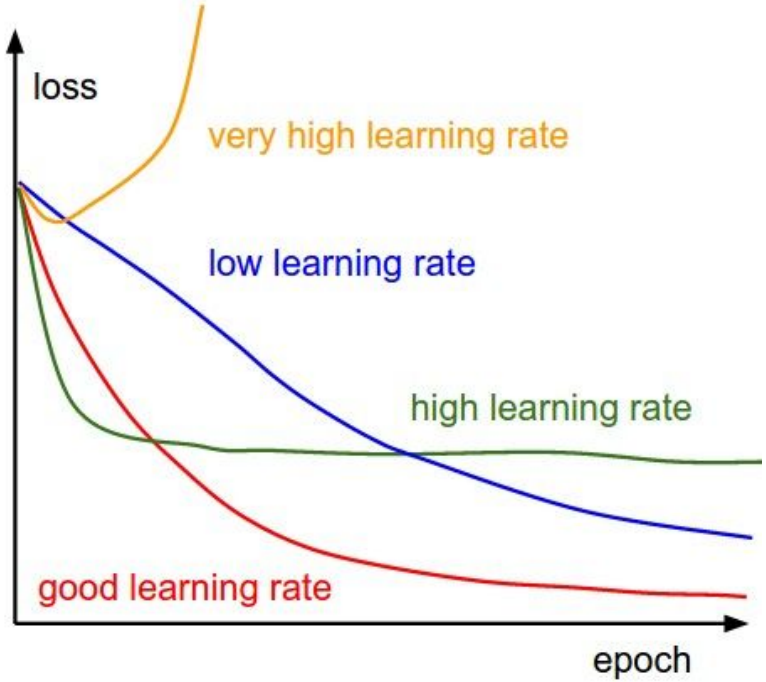
- A parameter named **learning rate** controls how big the step in the direction of the gradient is
  - A too large step may let you bounce back and forth on the walls of your “valley”
  - A too small step would make your descent lasting forever
- Several variants of SGD
  - Include “momentum” from previous gradient calculations (may help overcome local obstacles)
  - Reduce step size over time
  - Adadelta, Adagrad, Adam, and many more



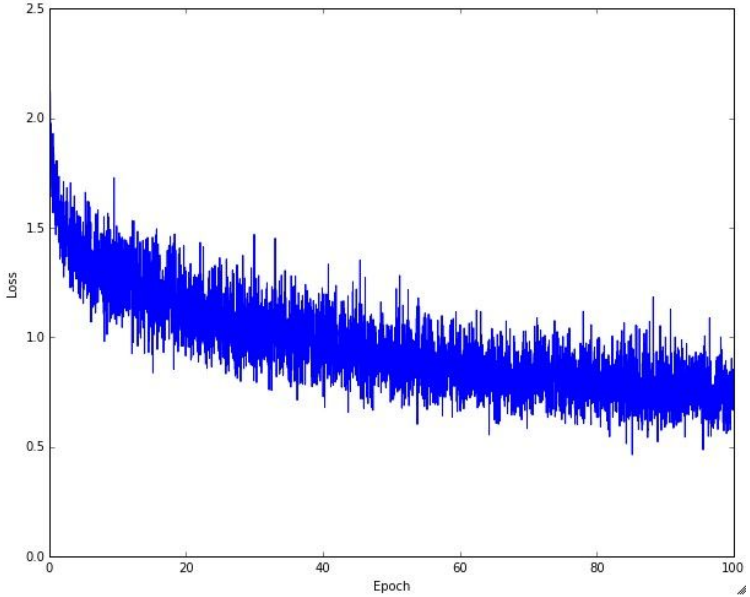


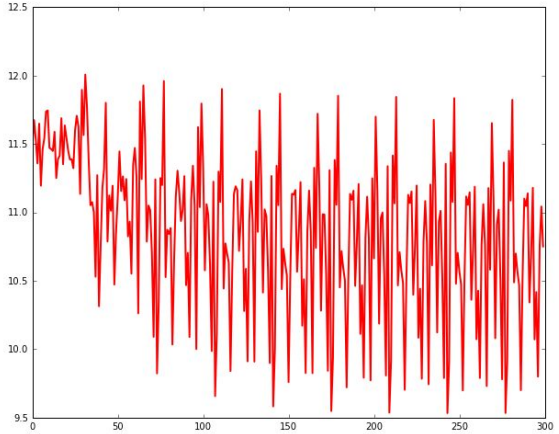
Learning rate, epochs and batches

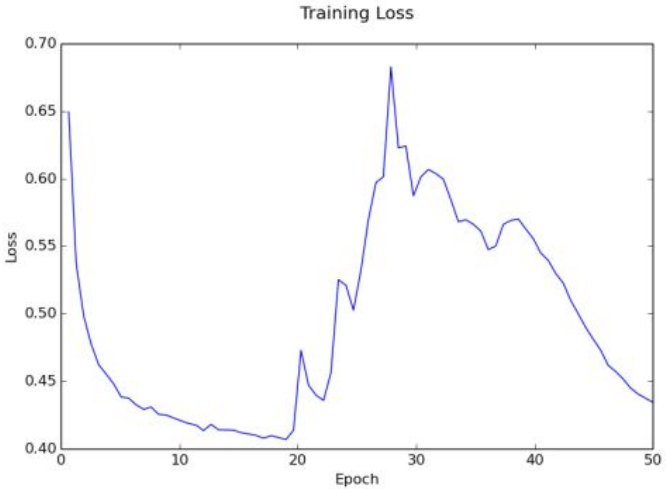
- The gradient update (in SGD) is repeated for each “batch” of events
- A full pass of the whole dataset (i.e. all batches) is called an **epoch**
- A typical training foresee iteration on multiple epochs
- The size of the update step can be controlled with a multiplicative factor called “**learning rate**”
    - Learning rate can be adapted over time



In reality

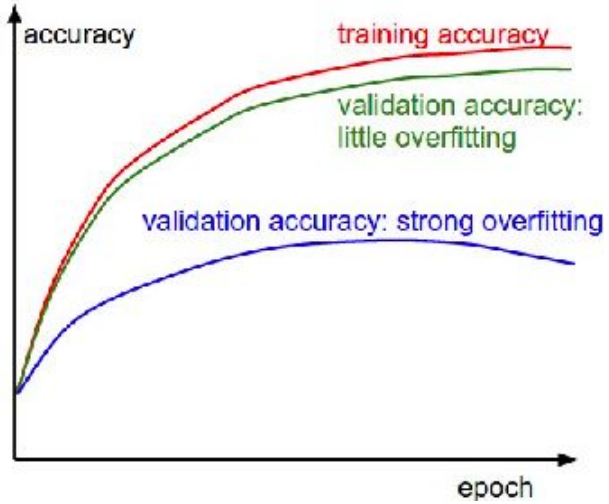

Loss
Epoch
0.0
0.5
1.0
1.5
2.0
2.5
0
20
40
60
80
100


9.5
10.0
10.5
11.0
11.5
12.0
12.5
0
50
100
150
200
250
300


Training Loss
Loss
Epoch
0.40
0.45
0.50
0.55
0.60
0.65
0.70
0
10
20
30
40
50

Training and overfitting

- As discussed earlier if the capacity is large enough the network could “overfit” on the training dataset
- Have a separate, stat independent, validation/generalization sample
- Evaluate performance (with “loss” or with other metrics) on the validation sample
- Training results depends on many choices
  - Size of batches (amount of “noise”)
  - Learning rate (how much you move along the gradient at each iteration)
  - Gradient Descent algorithm
  - Capacity of the network



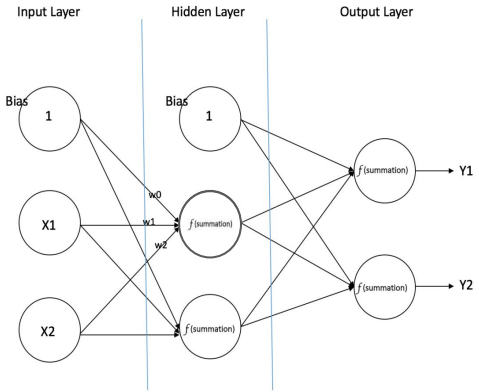
Deep networks

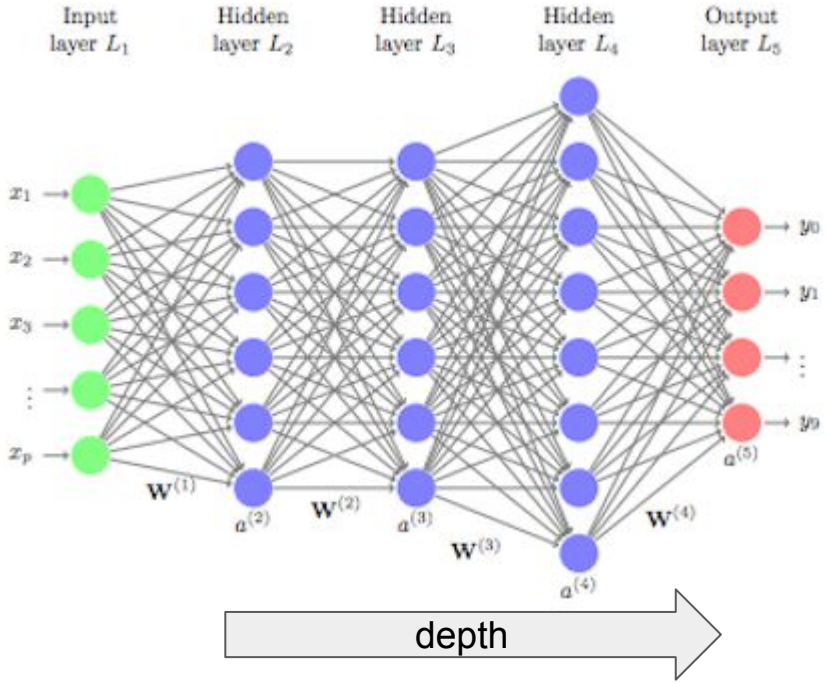
Deep Feed Forward networks

The simplest extension to the MLP is to just add more hidden layers

Other names of this network architecture

- Deep Feed Forward network
- Deep Dense network, i.e. made (only) of **Dense**(ly) connected layers





Why going deeper?

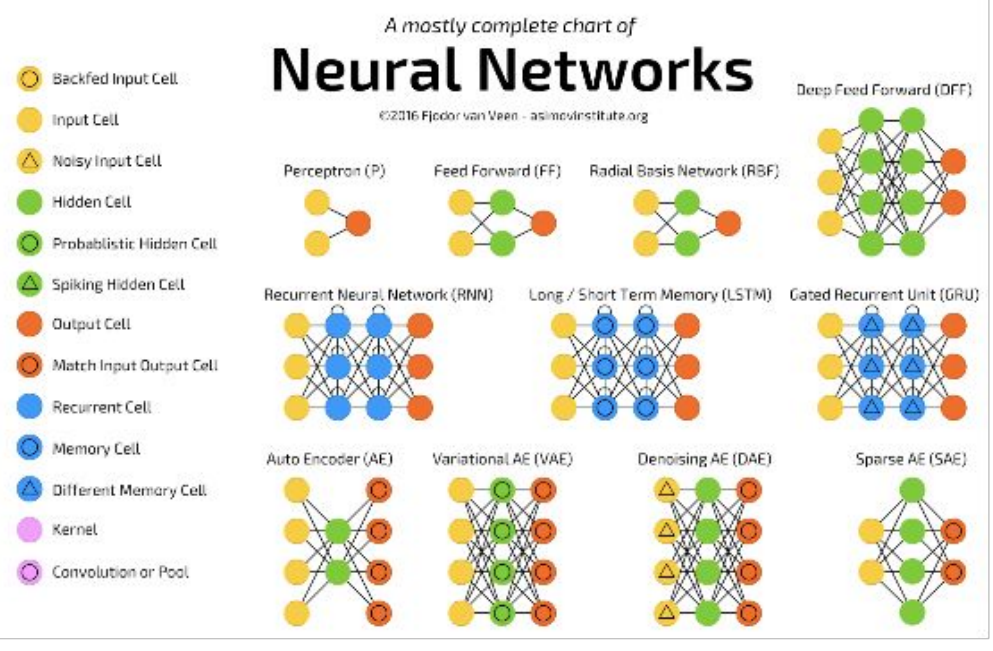
Hold on… wasn’t there a theorem saying that MLP is good enough ? Yes but…

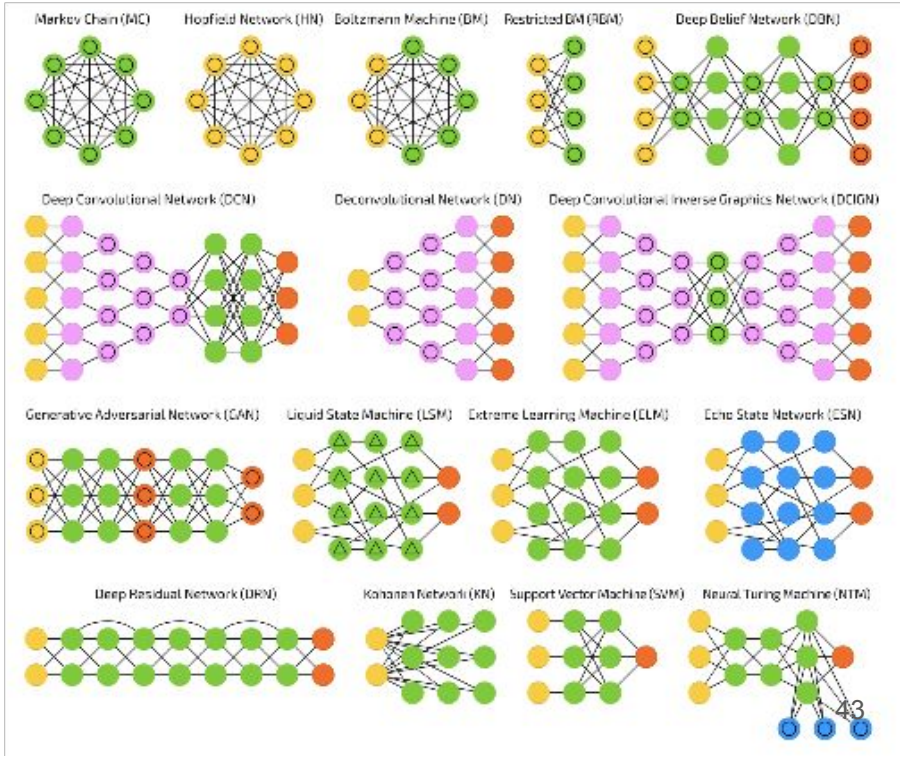
- Amount of nodes to represent complex functions can be too high
- Learning the weights on finite samples could be too difficult

Advantages of Deep architectures

- Hierarchical structure can allow easier “abstraction” by the network with early layers computing low level features and deeper layers representing more abstract properties
- Number of neurons and connections needed to represent the same function highly reduced in many realistic cases

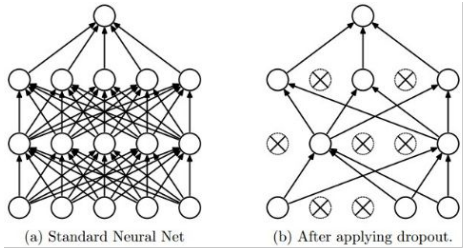
Deep architectures

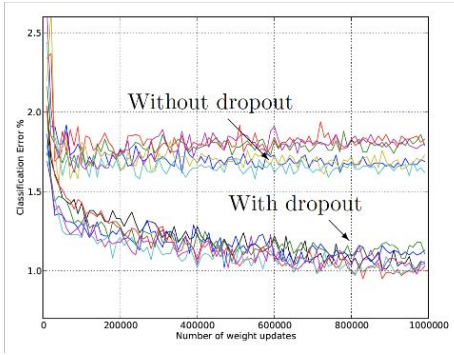




Dropout and regularization methods

- NN training is a numerical process
- Often the number of samples is limited hence the gradient accuracy is not great
- Several regularization methods exists to avoid being dominated by stochastic effects
  - Caps to the weights (so that individual nodes cannot be worth more than some amount)
  - **Dropout** techniques: during the training a fraction of nodes is discarded, randomly, at each iteration
    - NN more robust to noise
    - Effectively “augmenting” the input dataset

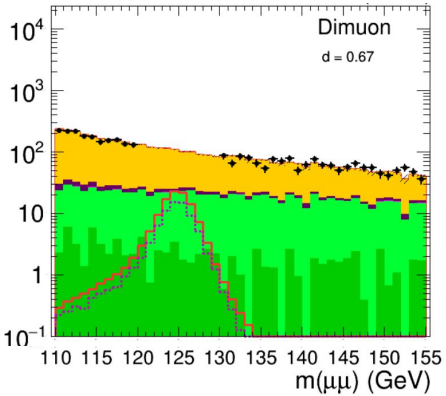


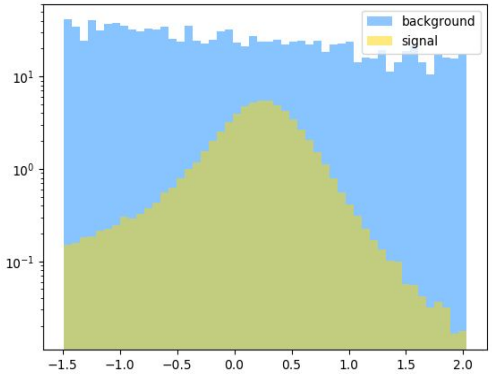


(Batch) normalization

- Input features have typically different ranges, means, variance
- It is generally useful to “normalize” the input distribution
  - Mean zero
  - Variance 1
- Often it could be practical to compute the normalization on individual batches rather than full sample
  - Batch vs full sample ? may depend on your use case

A typical observable, e.g. the invariant mass of a pairs of leptons





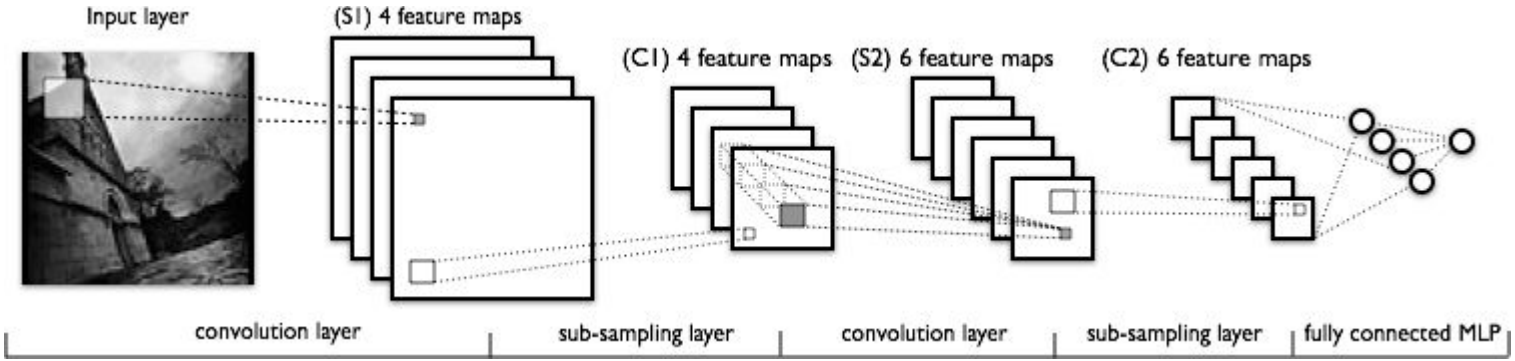
Normalized version

Exploit invariance and locality

- Suppose you want to count windows in a 800x600 picture with houses
  - With an MLP or DFF you have 800x600x3(RGB)=1.4M inputs
  - Each node process independently some part of the image
  - The initial “Dense” connection should converge to something with lot of “zero” weights because far away pixel points have no reason to be considered at the same time in order to detect **local** features
  - => the problem cannot be managed this way
- But the problem is translation invariant!
  - “Windows” are local features, you can just analyze a patch of the image **(locality)**
  - A window is a window no matter if it is top left or bottom right of your image **(Invariance)**
  - And actually windows are made of even more local features (some borders/frame, some uniform area, a squared shape)

Can we exploit problem invariance?

- Convolutional neural networks (CNN) attempt to exploit invariance against spatial translations
  - Smaller networks
  - Acting on a single patch of the image
  - Stacking multiple such Convolutional Layers one after the other

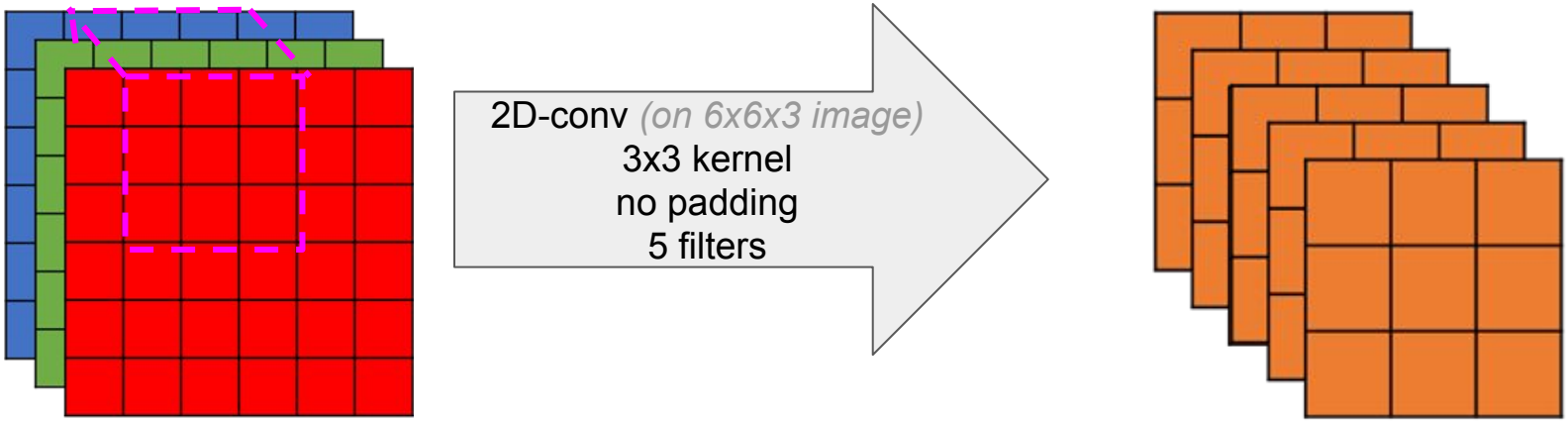


Limitations

- The linear algebra formalism we use can handle nicely images, hence implement nicely CNN (translation invariance along x and y)
- There are more invariances out there!
  - Rotation
  - Scale
  - Luminosity
  - … you name it…
- So currently the networks have to learn them all
  - We can do tricks to increase the number of samples in our datasets with augmentation techniques (i.e. apply random transformations of scale, rotation etc..)
  - “Built-in” invariance (such as the x-y one) has the advantage of reducing by orders of magnitude the number of weights to learn

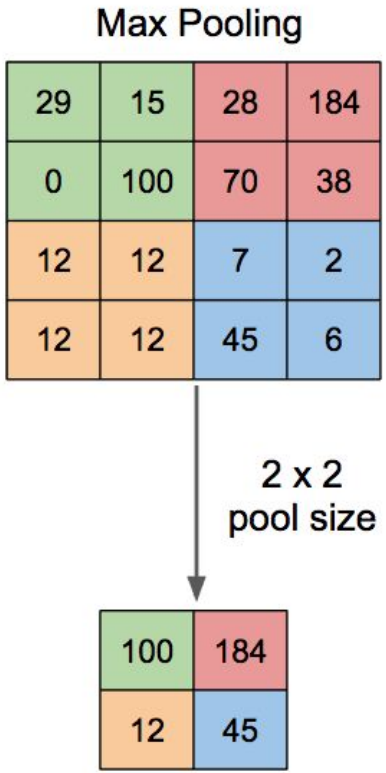
Understanding the dimensions of the convolution

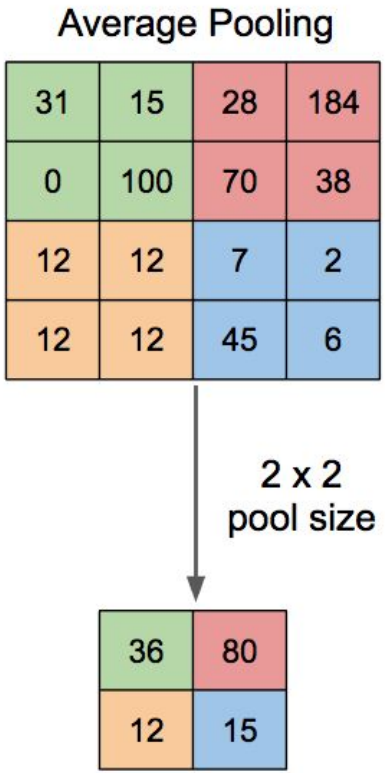
- Convolution can be 1D, 2D, 3D
- Kernel size, typically square (MxM) with M even (but can be any shape)
- *Padding*: how to we handle borders? We can do only “valid” windows (no padding) or process borders as if there were zeros (or other values) outside
- Each “point” in the 1D, 2D, 3D matrix can have multiple features (e.g. R,G,B)
- Each Convolutional layer have mutiple outputs (filters) for every “patch” it scans on (one optimized to detect if the patch is uniformly filled, one looking for vertical lines, etc..)



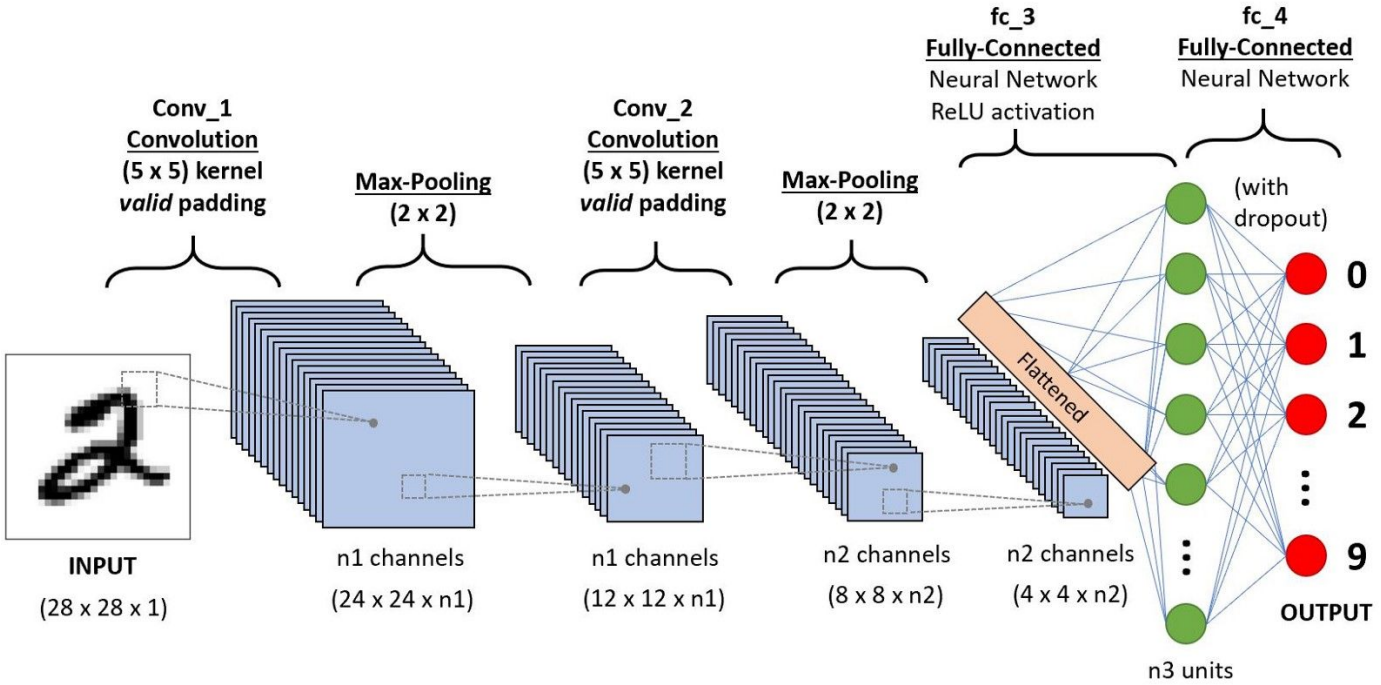
Pooling

- Pooling layers are simply finding maxima or computing average in patches of some convolution layer output
- Pooling is used to reduce the space dimensionality after a convolutional layer
  - The Conv “filters” look for features (e.g. a filter may look for cats eyes)
  - The Pooling layer checks if in a given region some filtered fired (there was a cut eye somewhere in this broad region)





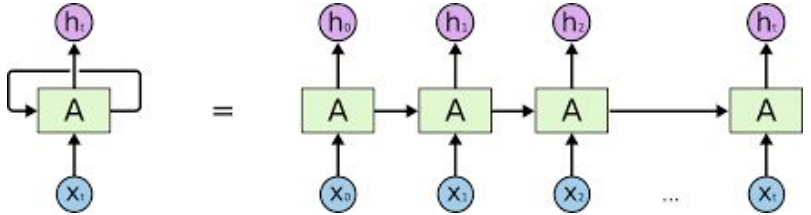
Typical CNN architecture



Exploiting time invariance

- Some problems are “time invariant”
  - E.g. recognize words in a sentence (written or spoken)
- Order matters and some causality is implied in the sequence
- Length of the inputs or the output may not be fixed

Recurrent Networks (RNN)

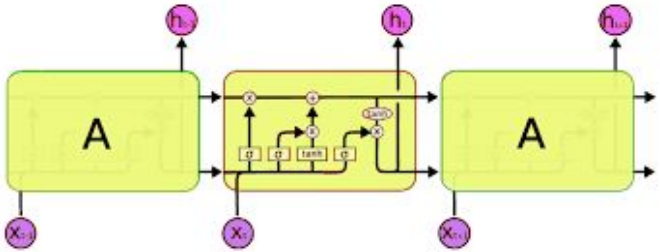


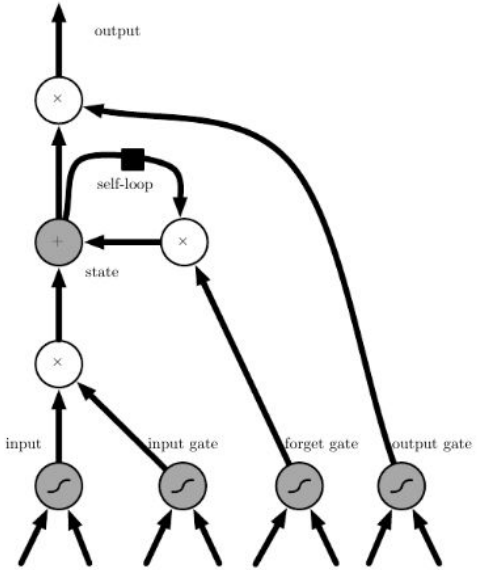
- Iterative networks with output passed again as input
  - Allow some “memory” of the previous inputs and/or some internal “state” of what the network understood so far in the sequence
- Most commonly used RNN are LSTM (Long Short Term Memory) and GRU (Gated Recurrent Unit)

LSTM and GRU

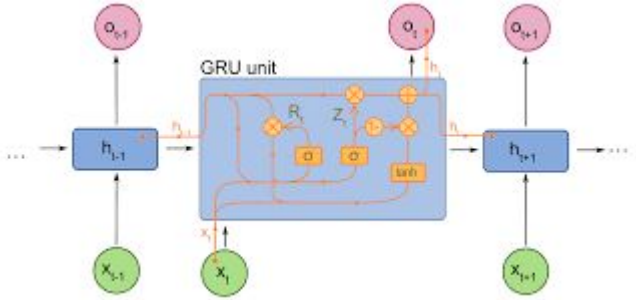
- LSTM and GRU are RNN units with additional features to *control* their “memory”
- “Gates” allow to control (keep or drop) *input*, *output* and *internal state*
- The advantage of gated units is that they *can forget* so that when processing a sequence they focus on the relevant part (e.g. when processing a text we may know that each time we encounter a space the word is over)

LSTM



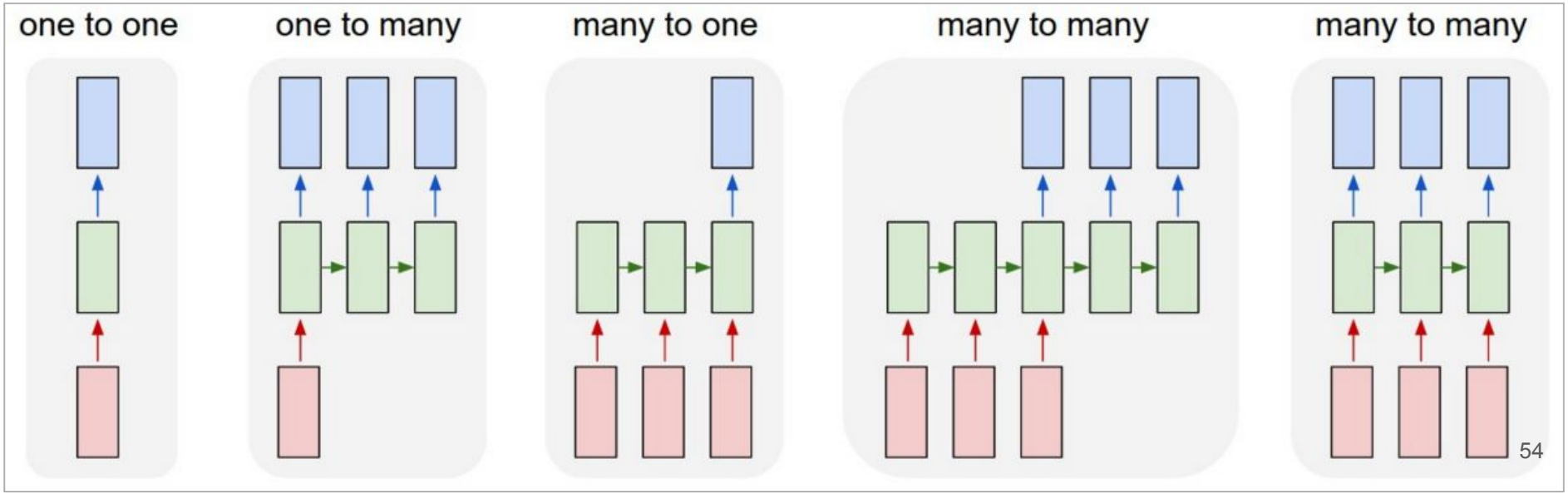


GRU



Different ways of processing time series

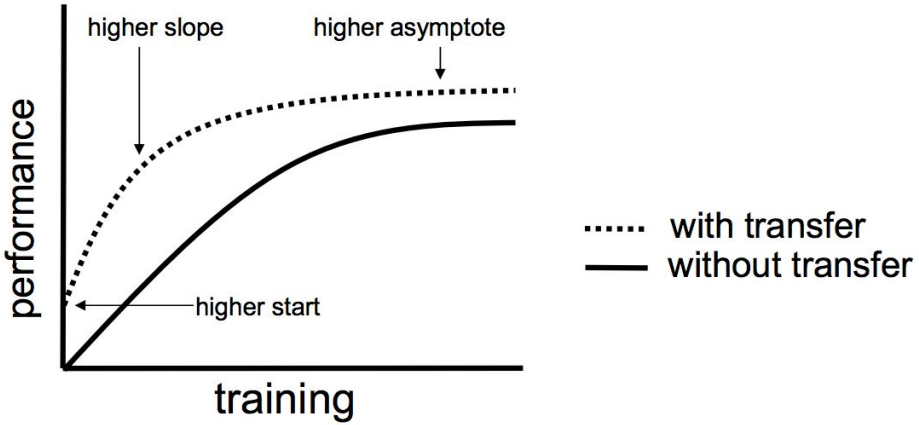
- Recurrent Networks can be used to implement networks with variable number of inputs and outputs
  - Encoding, Decoding, Sequence2Sequence



Transfer learning

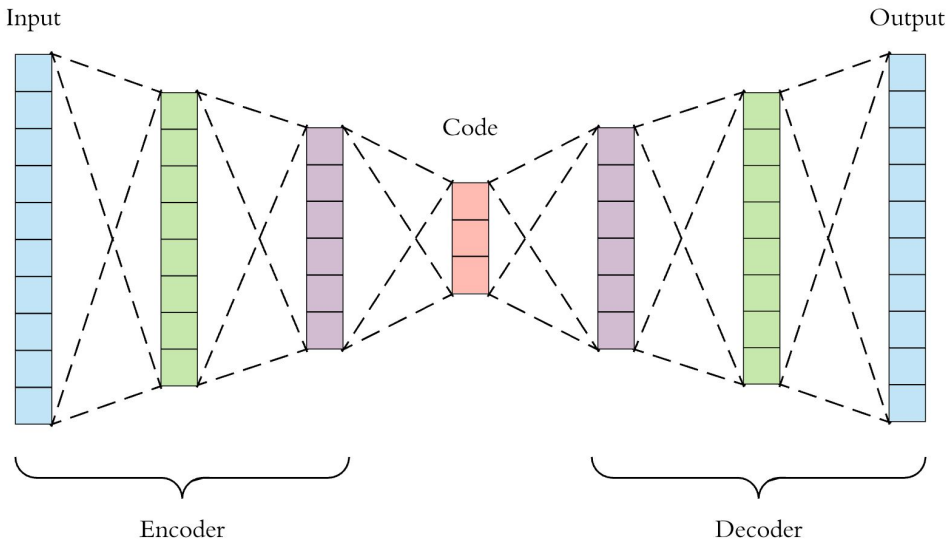
Transfer learning is a technique to reuse a network training for a task to perform **another task** with reduced retraining

- E.g. a Conv2D network meant for image processing have initial layers processing “local features”... that is not very domain specific (if yo trained on flowers images it may work on animals too)
- Very useful when the available sample of the proper domain is small
  - E.g. annotated medical images are harde to get than labelled real world pictures



Example of unsupervised architecture: autoencoders

- An example architecture that doesn't need labelled data is the "autoencoder"
- The network is shaped so that one of the inner layer has a much smaller dimension than the input
- The target is an **output matching the input**
- Being able to reproduce the input from the "coded" information in the inner layer means that the network was able to extract the important features of the input



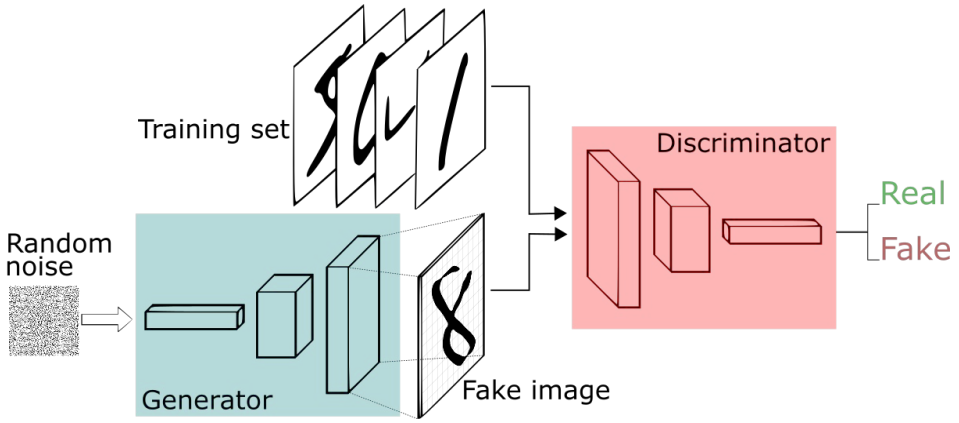
Generative Adversarial Networks

How about generating realistic samples ?

- E.g. generate pictures of animals or showbiz like faces
- Or something more useful in (HE)physics such as generating hadronic shower of a quark/gluon

GAN works with two independent networks:

- A generator
- A discriminator



- The **discriminator** separates samples of the training dataset from samples generated by the **generator**
- No label is needed in the training set as we know where each sample comes from
- **Generator** loss is controlled by the **discriminator** being able to recognize the fake

GAN progress

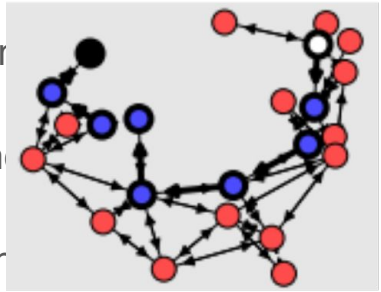
2014: “dogs with three heads”

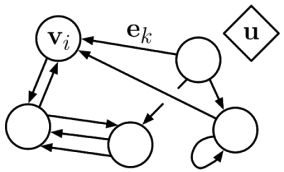
2018: coherent generation of faces

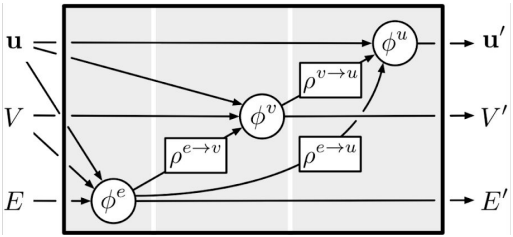
See also https://thispersondoesnotexist.com/

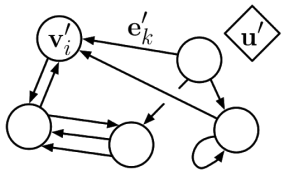
Graph nets and message passing

- CNN and LSTM exploit cartesian **invariance** to reduce the problem dimensio
- What if the problem have different **symmetries**?
- E.g. you can represent on a graph many type of data and have some invarian you want to perform on each node/edge
- **Locality** in this case is driven by “how many connections are separating two n
  - In order to let the information go from one node to another a “message passing” schema is used (i.e. the evaluation is iterative and at each step the information from nearest node can flow in)







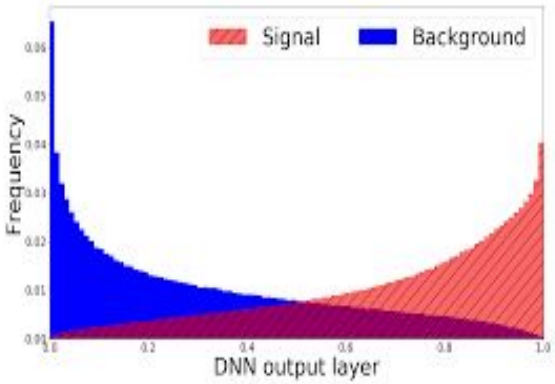


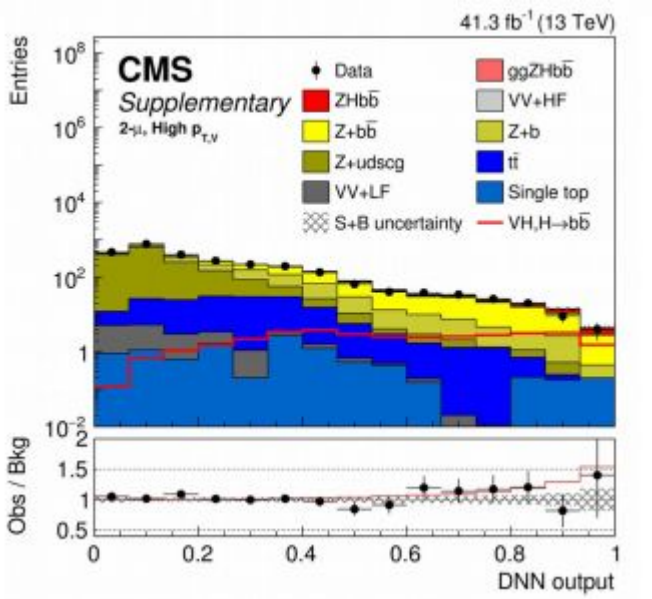
https://github.com/deepmind/graph_nets

Some example HEP applications

High level signal to background discrimination

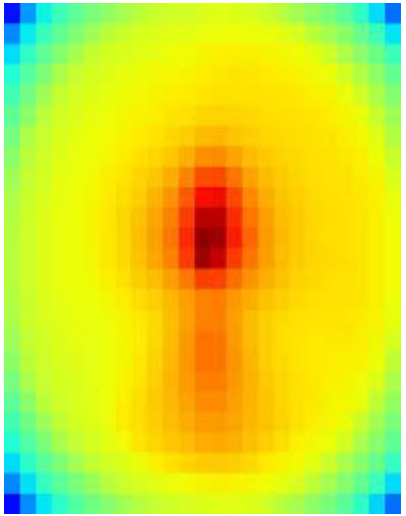
- The most common application of ML algorithms (BDT or DNN) is for final S/B discrimination.
  - Inputs are handcrafted features (invariant masses, decay hypothesis, angles in different reference frames, etc...)
  - Training is supervised using MC simulation as datasets (labels are known from MC truth)

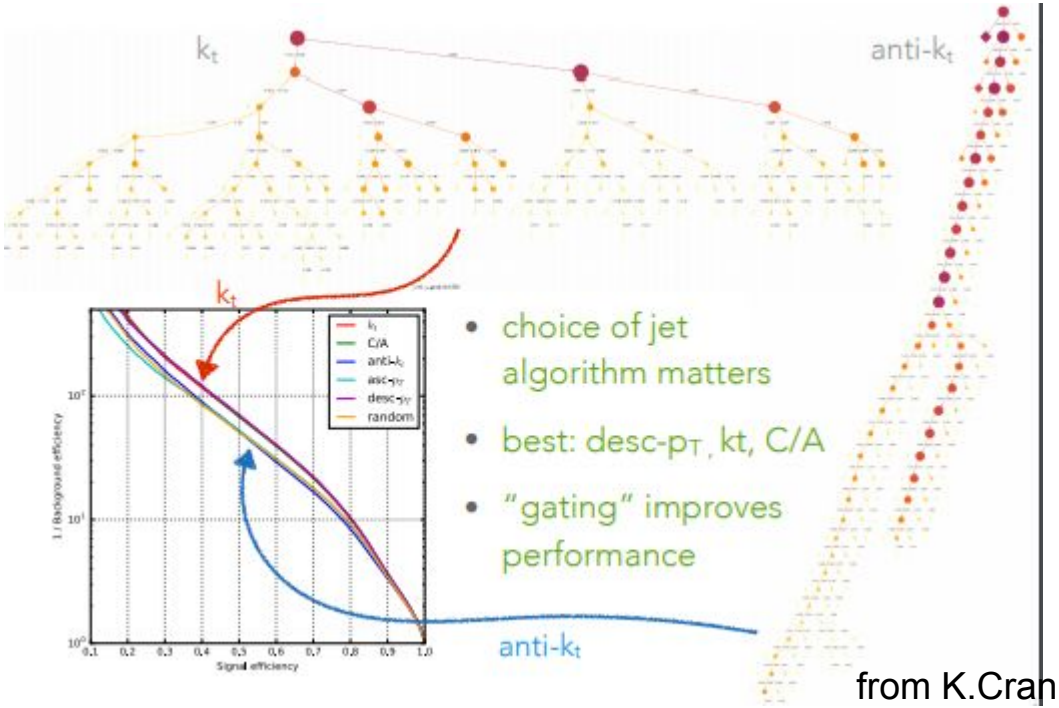




Jet identification

- B-tagging: identify jets of particles originating from b-quarks
  - See this afternoon hands-on session
- Boosted jets: distinguish (fat) jets originating from a vector boson hadronic data wrt QCD backgroun
  - Several techniques tested

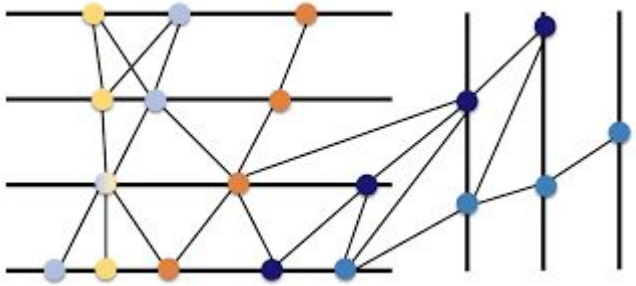


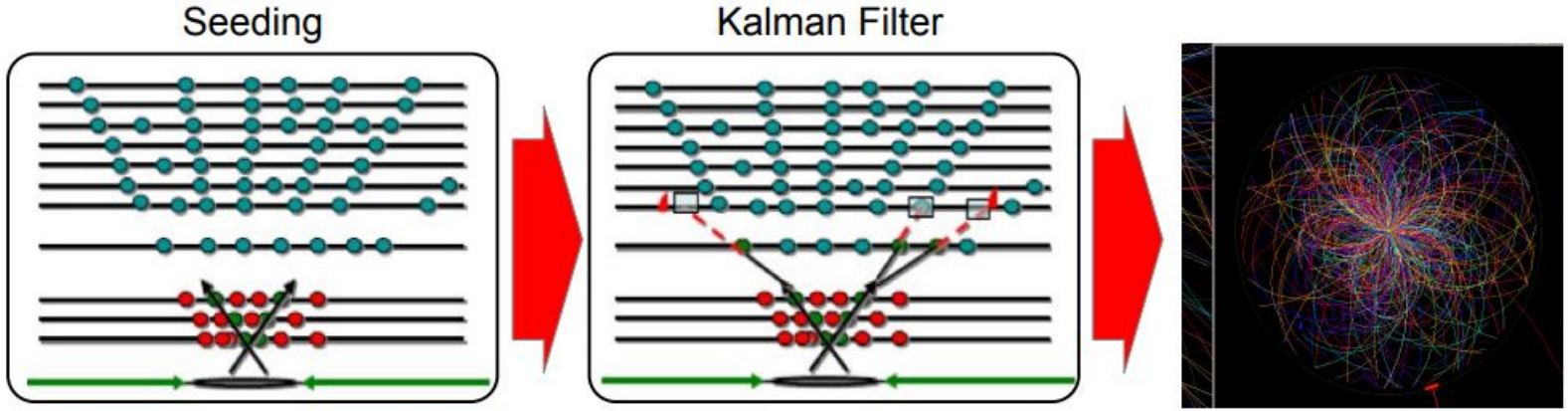


from K.Cranmer

Tracking with graph nets

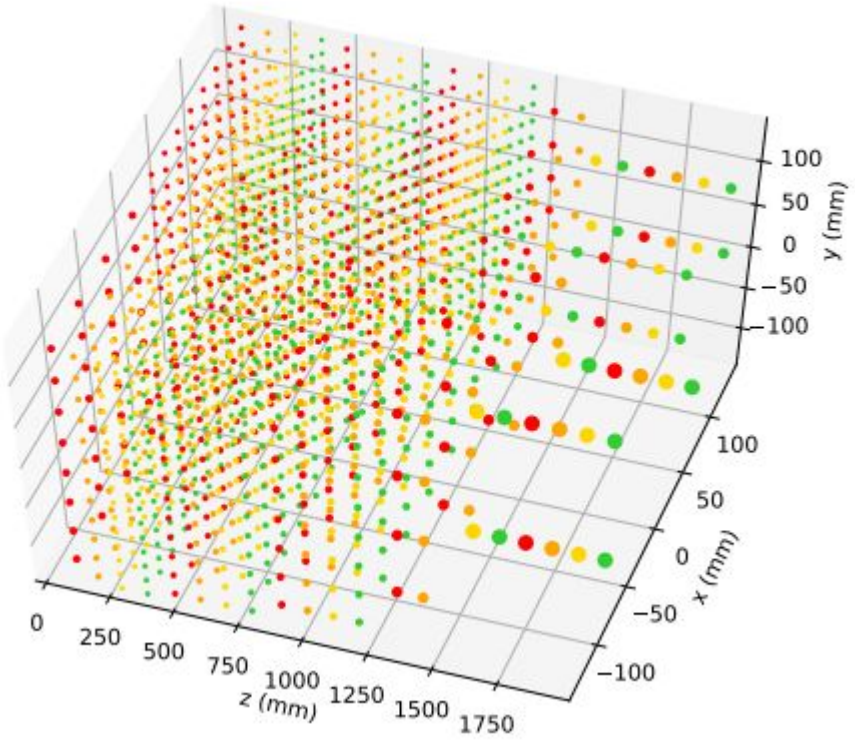
- Track reconstruction is the most time expensive algorithm at CMS/ATLAs
  - Currently using seeding + kalman filter
- Several group testing ML based ideas
  - Graph nets: the connection of hits to tracks are like a graph to prune





Reconstruction of complex detector topologies

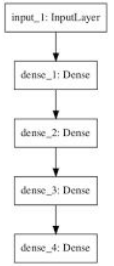
- Some (planned) detectors have irregular geometries so that clustering of measurements belonging to the same particle is not obvious
  - Use graph networks to represent non trivial connection map
- Particle Flow algorithm can also be designed to exploit graph nets to associate information from different detectors

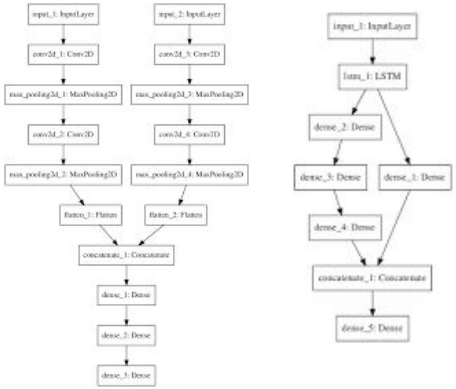


DNN Tools

Keras

- Keras is a python library that allow to build, train and evaluate NN with many modern technologies
- Keras supports multiple backends for actual calculations
- Two different syntax are usable to build the network architecture
  - Sequential: simple linear “stack” of layers
  - Model (functional API): create more complex topologies
- Multiple type of “Layers” are supported
  - Dense: the classic fully connected layer of a FF network
  - Convolutional layers
  - Recurrent layers
- Multiple type of activation functions
- Various optimizers and gradient descent techniques





Other common tools

Common alternative to keras

- Pytorch
- Sonnet
- Direct usage of TensorFlow (or other backends such as Theano, Torch, …)
  - Need to write yourself some of the basics of NN training
  - Especially useful to develop new ideas (e.g. a new descent technique, a new type of basic unit/layer)

Keras Sequential example

```
# first neural network with keras tutorial
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
# define the keras model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10)
# evaluate the keras model
_, accuracy = model.evaluate(X, y)
print('Accuracy: %.2f' % (accuracy*100))
```

Keras “Model” Functional API

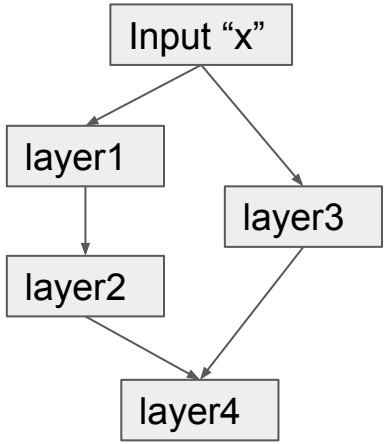
A NN can be seen as the composition of multiple functions (one per layer), e.g.

- A simple stack of layers is: $y=f_5(f_4(f_3(f_2(f_1(\boldsymbol{x})))))$
- A more complex structure could be something like

$$y=f_4(\ f_2(f_1(\boldsymbol{x})),f_3\ (\boldsymbol{x})\ )$$

- The functional API allow to express the idea that each layer is evaluate on the output of a previous layer, i.e.

```
x = Input()
layer1=FirstLayerType(parameters) (x)
layer2=SecondLayerType(parameters) (layer1)
layer3=ThirdLayerType(parameters) (x)
layer4=FourthLayerType(parameters)([layer2,layer3])
```

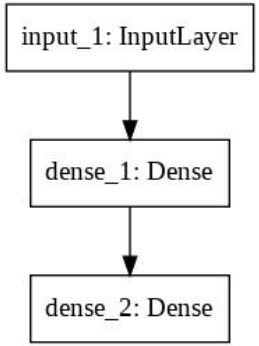


An MLP in keras

```
from keras.models import Model
from keras.layers import Input, Dense
x = Input(shape=(32,))
hid = Dense(32)(x)
out = Dense(1)(hid)
model = Model(inputs=x, outputs=out)


model.summary()
from keras.utils import plot_model
plot_model(model, to_file='model.png')
```

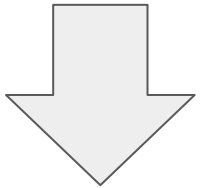
```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 33
=================================================================
Total params: 1,089
Trainable params: 1,089
Non-trainable params: 0
_________________________________________________________________
```



From the ~1995 to ~2010

```
from keras.models import Model
from keras.layers import Input, Dense

x = Input(shape=(32,))
hid = Dense(32)(x)
out = Dense(1)(hid)
model = Model(inputs=x, outputs=out)
```



```
from keras.models import Model
from keras.layers import Input, Dense

x = Input(shape=(32,))
b = Dense(32)(a)
c = Dense(32)(b)
d = Dense(32)(c)
e = Dense(32)(d)
model = Model(inputs=x, outputs=e)
```

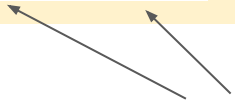
Training a model with Keras

```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```



Those are numpy arrays with your data

Keras layers

Keras basic layers

- Basic layers
  - Inputs
  - Dense
  - Activation
  - Dropout
- Convolutional layers
  - Conv1D/2D/3D
  - ConvTranspose or “Deconvolution”
  - UpSampling and ZeroPadding
  - MaxPooling, AveragePooling
  - Flatten
- More stuff
  - Recursive layers
  - ...check the keras docs...

Simple Exercise

- Partition a 2D region with a simple function
  - E.g. x>y or x*2>y
- Generate few samples
- Build a MLP or a DNN with similar number of parameters that approximate the given function

https://colab.research.google.com/drive/1X9uppZENlN9PbX4Tr4e959JSyq4ePv3S