# Introduction to Machine Learning for Particle Physics

Ian J. Watson
ian.james.watson@cern.ch

University of Seoul

KCMS Lectures
November 30, 2021

## Contents

- Overview of Machine Learning
- Decision Trees
- Neural Networks
- Example ML analysis using TMVA with CMS data - Quark/Gluon Separation
- Obviously, each of these topics could be expanded to several hours of lectures, today is really a brief overview to give a taste of what's possible!

## Introduction - Machine Learning

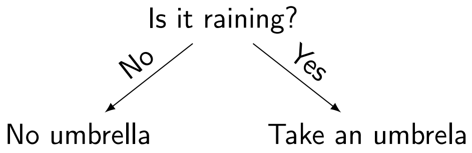
- **Machine learning** refers to sets of algorithms (techniques) that can "learn" from experience
- Given inputs and expected output can automatically learn to associate patterns in the input to the output and generalize on unseen inputs.
  - Outputs could be (a) real number(s): regression (photon energy)
  - Or it could be a *classification* into one of several classes ($\gamma$ v $e^-$ v $\mu^-$)
- As opposed to traditional algorithms which are explicitly pre-programmed to always act in specific ways
- Example: Does this signal in my detector correspond to a photon or a hadron? Feed the algorithm thousands of (simulated) photons & hadrons, and it will learn to distinguish them
  - More specifically, we will build a parameterized model which gives a "probability"* for an input datapoint to be photon or hadron, and the algorithms changes the parameters to better classify the photons or hadron examples you feed it

* Typically, its not actually a probability, but easier to think of it that way

## Classification

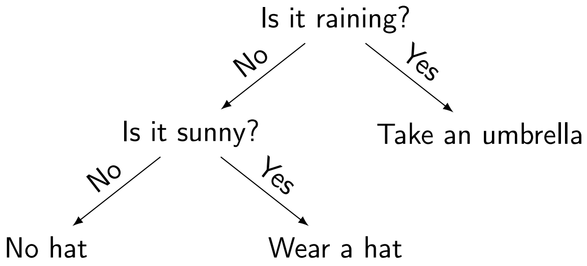
- Lets consider classification problems
  - In classification we have some data which could belong to one of several classes
  - We have some well-known data and we want to train a *classifier* which will tell us what class some new, unknown data comes from
  - E.g. classify images of pets into dogs and cats
  - Classify energy depositions in a calo into photons and electrons
  - Based on several indicators (age, height, weight, etc.) say whether someone will get diabetes
- Decisions can be complicated: many input variables with many non-trivial relationships differing by class, modelling this in general is called Multi-Variate Analysis (MVA)
- The goal is to find a *decision boundary*, one one side of the boundary we classify the input as a $\gamma$, on the other its a hadron
- We will start with a straightforward technique called Decision Trees

## Decision Trees



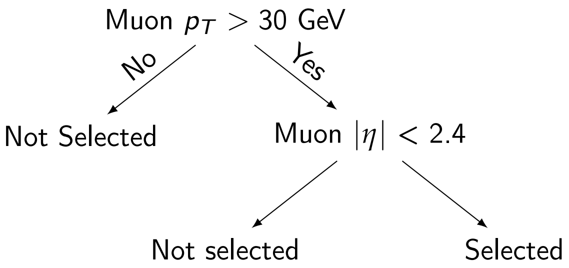
- Decision trees give a path to a result based on some conditions

## Decision Trees



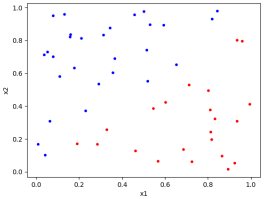
- Decision trees give a path to a result based on some conditions
- There could be several inputs, with multiple kinds of outputs
    - But always evaluate from top node down
- For true/false boolean inputs, straightforward to enumerate all options
- Write down all the paths through the questions, add a label at the end

## Some Decision Tree Examples



- In the case of real valued inputs, we have to be more careful
- We can create left/right branches by asking for a value to be above/below some cut-off
    - We turn a real value variable into a binary decision at each node

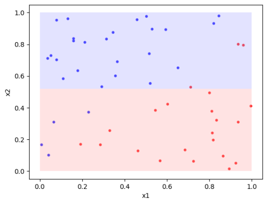
## Decision Trees with Real Numbers



- Given a set of data we want to split into red and blue spaces

## Decision Trees with Real Numbers

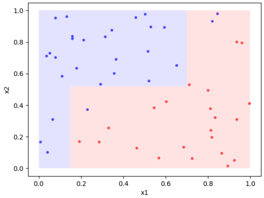
$$\begin{tikzcd}
 & x2 > 0.5 \arrow[dl] \arrow[dr] & \\
\text{Red} & & \text{Blue}
\end{tikzcd}$$



- Given a set of data we want to split into red and blue spaces
- The decision tree will partition the problem space into discrete regions

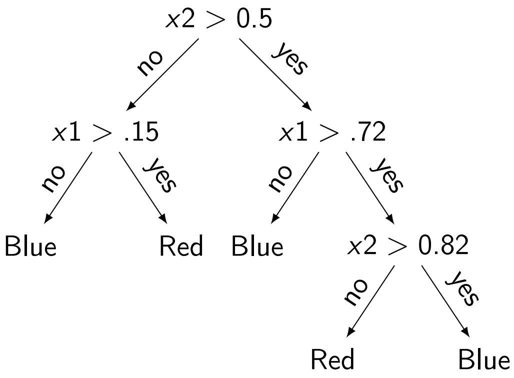
# Decision Trees with Real Numbers

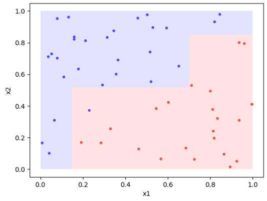
$$\begin{tikzcd}
 & & & x2 > 0.5 \arrow[dll, "no"'] \arrow[drr, "yes"] & & & \\
 & x1 > .15 \arrow[dl, "no"'] \arrow[dr, "yes"] & & & & x1 > .72 \arrow[dl, "no"'] \arrow[dr, "yes"] & \\
\text{Blue} & & \text{Red} & & \text{Blue} & & \text{Red}
\end{tikzcd}$$



- Given a set of data we want to split into red and blue spaces
- The decision tree will partition the problem space into discrete regions
- Can add *levels* to split the space up further and further

## Decision Trees with Real Numbers



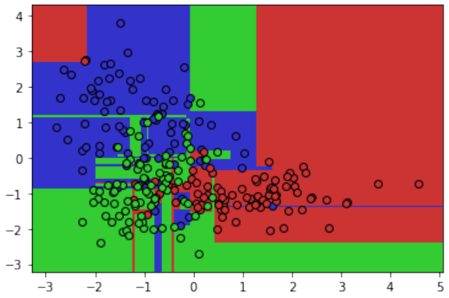


- Given a set of data we want to split into red and blue spaces
- The decision tree will partition the problem space into discrete regions
- Can add *levels* to split the space up further and further

## Training Decision Trees: CART algorithm, GINI coefficient

- To train the decision tree, scikit learn (and TMVA?) uses the CART algorithm to minimize the Gini coefficient $G$
- $G = 1 - \sum_k p_k^2$ summed over the classification classes $k$, where $p_k$ is the fraction of data in class $k$
  - If all the data is in one class then $G = 0$, if the data is split evenly over n classes then $G = \frac{n-1}{n}$
  - If the data is more unevenly split, the G value goes closer to 0
- The CART algorithm takes a dataset of $m$ datapoints and tries to make a split into two branches left and right, which minimize $J = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right}$
  - Its trying to minimize the number-of-datapoints-weighted-average Gini-coefficient of the left-right split at each *node*
- After the split, it then applies the same logic again onto each subtree until
  - A stopping parameter condition is met, i.e. the user can say only go to a certain *depth*
  - Each leaf (split dataset) only contains one class
  - It can't find a split which reduces the Gini-coefficient
- A *leaf* is a final decision, i.e. a node with no more splits

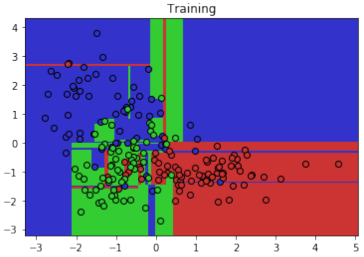
## Some Data

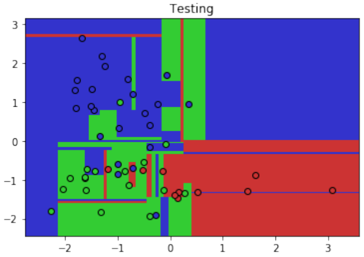


- You can see its *overfitting*, producing high variance changes in decisions based on the exact data, which probably won't generalize to unseen data

- Some synthetic data (i.e. created artificially) we will use to illustrate today's concepts
  - The circles are the datapoints, labelled by colors
- Has 3 categories with 2 variables, lots of overlaps
- Running a decision tree over the data gives a complicated decision boundary
  - The background colors show the decision for each point after training the tree

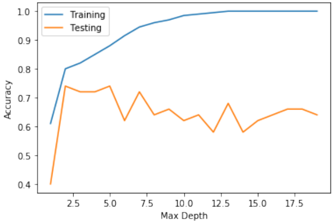
## Training and Testing





- Without a priori knowledge of the data distribution, we need some way to test if we are overfitting, using the data itself
- We can do this by setting aside some portion of the data, the testing set, and only train on the remaining data, the training set
- If we take 80% of the previous page data and fit the tree, overlaying the output shows clear overtraining (e.g. blue in cut out blocks of green)

# Bias vs Variance



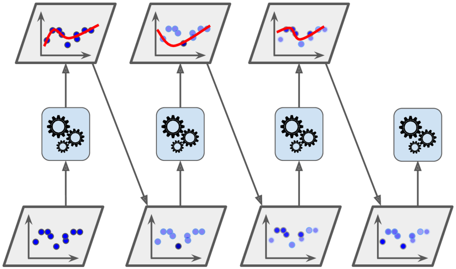
- We can see this particularly clearly by plotting accuracy (fraction of data classified correctly) against the depth of the tree (= complexity)

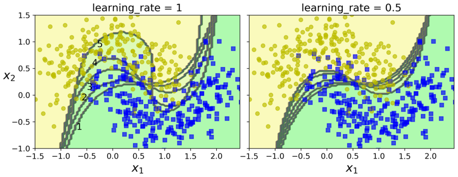
- As we increase the max depth, the training set gets more accurate, the test set diverges, and becomes less accurate, this is overfitting
- Bias: the difference between our model prediction and the datapoints
  - A high *bias* model has large differences between the datapoints
  - A model which doesn't have enough parameters *underfits* the data
- *Variance*, a measure of the fluctuations of the data or model, high variance models are typically fitting the intrinsic noise of the data
  - A high variance model with too many parameters *overfits* the data
- The *bias-variance trade off* is a theorem which tells us that you trade off model bias for variance and vice-versa, best to find a trade-off point between the two regions

## Boosting

- In order to reach that point, we need a *high capacity* model, one which has enough freedom to model our data
- We can use *ensemble classifiers* to increase performance
- In Boosting, we start with a weak classifier (barely better than random chance), and put them together to form a strong classifier
- This is done by *weighting* the data for each classifier we train
- E.g. start with a depth 1 decision tree, we can weight the misclassified data higher, and the correctly classified data lower
- Train a depth 1 tree on the reweighted data, this gives a different tree, since the gini index will be calculated based on the *weights*, instead of just taking the number of entries in each bucket
- This scheme is AdaBoost, there are also other variations, such as Gradient boosting, but in all the idea is to take a weak classifier and train up an ensemble of strong classifiers

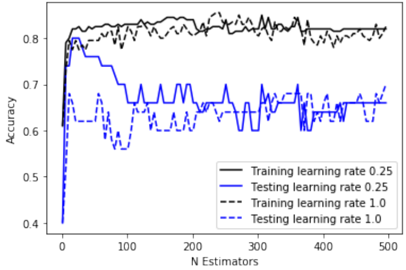
## Boosting





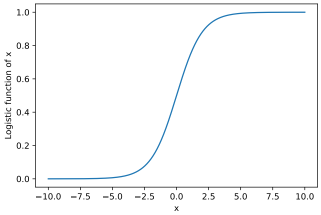
- Left shows a regression task, but the idea is the same, the further the curve is from a point, the higher its weighted in the next tree
- How quickly the weights change are controlled by the learning rate
- After training, the output is taking as a weighted average of the trees, the weight of each tree proportional to the number of correctly classified training datapoints it produced
    - Also produces a smoother decision boundary (average out fluctuations)
- See e.g. chapter 7 of "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition" for more details

## Boosting Decision Trees



- When we boost the decision tree learning, we end up with a Boosted Decision Tree (BDT), which really is a *forest* of decision trees
- Various number of depth 1 decision trees trained with AdaBoost on the previously shown dataset
- If the learning rate is too high, the boosting overcorrects too quickly, and we don't get good testing results
- With a lower learning rate, the boosting helps us get a more accurate classifier without overtraining

## Logistic Function

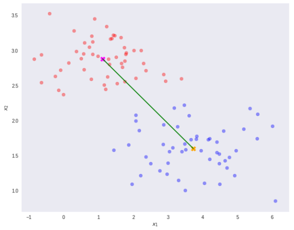


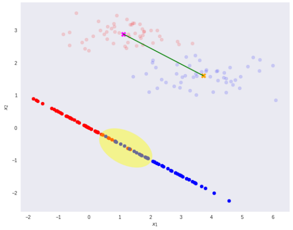
- The logistic function is defined as $f(x) = \frac{1}{1+e^{-x}}$
    - Looks like a classic "turn-on" curve
- "Logistic regression" fits this function from several variables
- Concentrate on the case of two classes (cat/dog or electron/photon), and ask what we want from a classifier output
    - We need to distinguish between the two classes using the output:
    - If the value is 0, it represents the classifier identifying one class (cat)
    - If its near 1, the classifier is identifies the other class (dog)
    - Thus, we need to transform the input variables into 1D, then pass through the logistic function

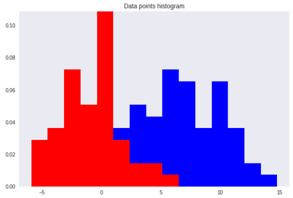
## Mathematics of Logistic regression

- Setup: we have data from two different classes, which can be described by the same independent variables, and we want to distinguish them based on those independent variables
- We want to build a function such that data from one class goes close to 1, from the other close to 0
- We will build a linear function of the variables, then pass it through the logistic function, and try to minimise the distance of data from 0 (for one class), or 1 (for the other)
- $y_i = f(\vec{\beta} \cdot \vec{x}_i) + \epsilon_i$, $y_i = 0$ if $x_i$ from class 0, 1 if $x_i$ from class 1
  - $\vec{\beta} \cdot \vec{x}_i = \beta_0 + \beta_1 x_1 + \ldots \beta_k x_k$ and $f(x) = \frac{1}{1+e^{-x}}$ the logistic function
- Find $\beta$ which minimizes a *loss function* e.g.: $MSE = \frac{1}{m} \sum_i (f(x_i) - y_i)^2$
- We end with the optimized parameters for the intercept and coefficients
  - The intercept sets *where* the turn occurs
  - Coefficients set how *quickly* the turn on occurs, larger coeff. imply fast turn on (sweep across the logistic function quicker)

## Illustration: 1D Projection

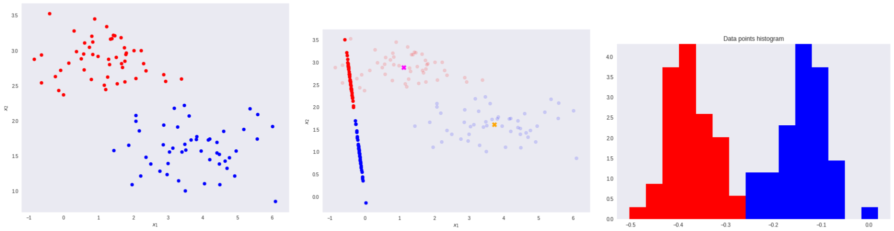






- $\vec{\beta} \cdot \vec{x}$ is a projection of the data onto a line
- Red and blue are two classes which can be measured in $(x_1, x_2)$
- We can take the mean of each class (left), form a line between, then project the data onto the line (middle) giving a distribution (right)
  - We have reduced the 2D data into a 1D projection
- After the projection, the logistic rejection chooses a cut point (via $\beta_0$) then sends things below the cut to 0, above to 1
- Here, we see some separation between the classes but a lot of overlap. We can do better

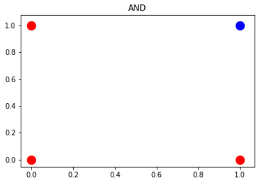
## Illustration: Better Fit

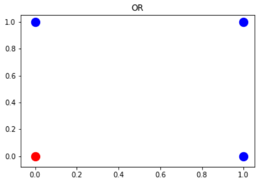


- Finding the *Fisher discriminant* for our illustrative dataset shows that these two classes are fully separable
- The Logistic Regression will place the cut point between the data and so all red go to 0, blue go to 1 after passing through the logistic function

https://medium.freecodecamp.org/an-illustrative-introduction-to-fishers-linear-discriminant-9484efee15ac

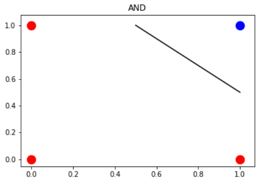
## Some very simple examples for simple logistic regression

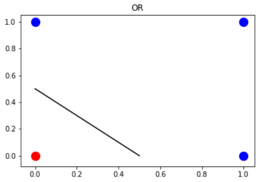




- Let's think about using logistic regression to approximate some simple binary functions, i.e. data in 2D, output red or blue, logistic regression will turn on at a boundary line
- OR and AND gates
  - OR is 0 (red) if both input are 0, 1 (blue) otherwise
  - AND is 1 if both inputs are 1, 0 otherwise
- Can we find logistic function approximations for this?
  - That is, $f(x_1, x_2)$ returns approximately 1 or 0 at the indicated points

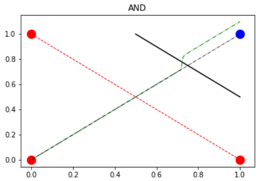
## Some very simple examples for simple logistic regression

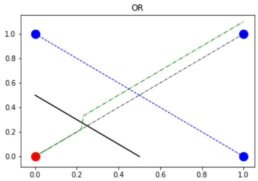




- Let's think about using logistic regression to approximate some simple binary functions, i.e. data in 2D, output red or blue, logistic regression will turn on at a boundary line
- OR and AND gates
  - OR is 0 (red) if both input are 0, 1 (blue) otherwise
  - AND is 1 if both inputs are 1, 0 otherwise
- Can we find logistic function approximations for this?
  - That is, $f(x_1, x_2)$ returns approximately 1 or 0 at the indicated points
- Yes! Take the projection perpendicular to the line

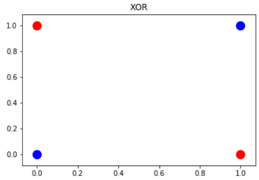
## Some very simple examples for simple logistic regression





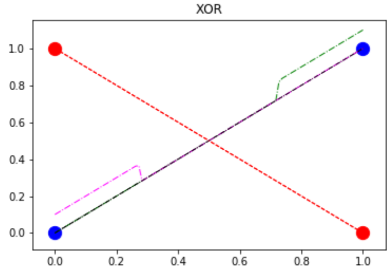
- Let's think about using logistic regression to approximate some simple binary functions, i.e. data in 2D, output red or blue, logistic regression will turn on at a boundary line
- OR and AND gates
  - OR is 0 (red) if both input are 0, 1 (blue) otherwise
  - AND is 1 if both inputs are 1, 0 otherwise
- Can we find logistic function approximations for this?
  - That is, $f(x_1, x_2)$ returns approximately 1 or 0 at the indicated points
- Yes! Take the projection perpendicular to the line
- and have the logistic turn on at the line
  - e.g. $f(x_1, x_2) = \sigma(2x_1 + 2x_2 - 1)$ for OR, $f(x_1, x_2) = \sigma(2x_1 + 2x_2 - 3)$ for AND [$\sigma$ is our logistic function]

## Very simple example with issues for Logistic Regression



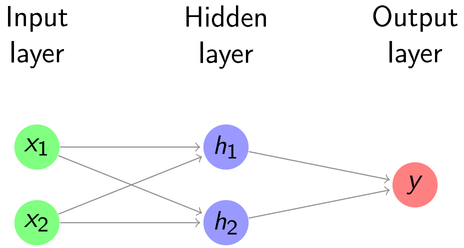
- Now consider the XOR gate: 1 if both inputs are the same, 0 otherwise
- The XOR gate can't be generated with a logistic function!
- Try it: no matter what line you draw, can't draw a logistic function that turns on only the blue!

## How to Fix: more logistic curves!



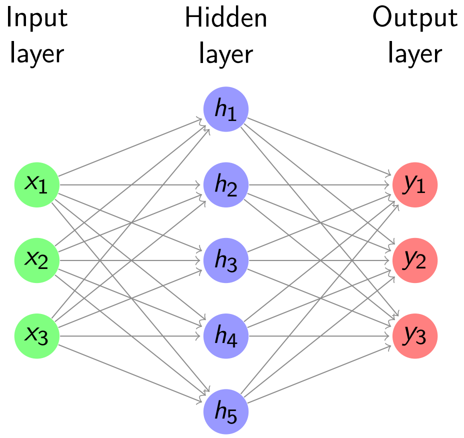
- Can fix by having 2 turn-on curves, one turning on either of the blue points, then summing the result
- $f(x_1, x_2) = \sigma(2x_1 + 2x_2 - 1) + \sigma(-2x_1 - 2x_2 + 1)$
- In general, this type of issue (complicated decision boundaries) is why we reach toward multivariate methods such as decision trees and neural networks

## The Feed-Forward Neural Network



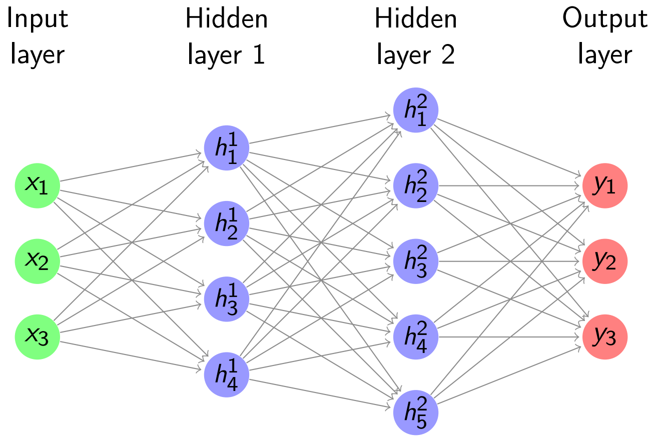
- Consider the structure of what we just made
  - $y = f(x_1, x_2) = \sigma(-1 + 2x_1 + 2x_2) + \sigma(1 - 2x_1 - 2x_2)$
- Decompose the function into:
  - the *input layer* of $\hat{x}$,
  - the *hidden layer* which calculates $h_i = \beta_i \cdot x$ then passes if through the *activation function* $\sigma$, ("logistic function" called "sigmoid" in NN)
    - as in logistic, there is an extra $\beta_0$, called the *bias*, which controls how big the input into the node must be to activate
  - the *output layer* which sums the results of the hidden layer and gives $y$
    - $y = 0 + 1 \cdot \sigma(h_1) + 1 \cdot \sigma(h_2)$

## Feed-Forward Neural Network



- In general, we could have several input variables, and output variables
- In the case of classification, we would usually have a final *softmax* applied to $\hat{y}$, but could use any *activation* $\varphi$ here also
  - *softmax* normalizes the output layer, so the outputs add to 1
  - The output layer then acts *like a* probability for each category

## Feed-Forward Neural Network



- We can even have several hidden layers
    - The previous layer acts the same as an *input layer* to the next layer
    - The layers can build up more complex features to discriminate with
- We call each node in the network a *neuron*

## Aside: Universal Approximation Thereom

Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let $I_m$ denote the $m$-dimensional unit hypercube $[0, 1]^m$. The space of real-valued continuous functions on $I_m$ is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer $N$, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$ such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi \left( w_i^T x + b_i \right)$$

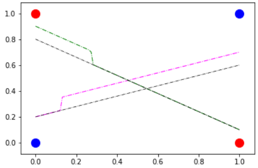
as an approximate realization of the function $f$; that is,

$$|F(x) - f(x)| < \varepsilon$$

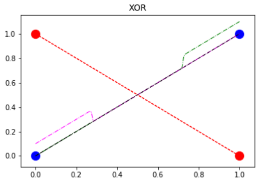
for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$. This still holds when replacing $I_m$ with any compact subset of $\mathbb{R}^m$.

- In brief: with a hidden layer (of enough nodes), any (sensible) function $f : \mathbb{R}^m \to \mathbb{R}$ can be approximated by a feed-forward NN
  - Any (sensible) activation $\varphi$ can work, not just $\sigma$
- Shows we won't run into the XOR issue with a neural network

# Training a Neural Network

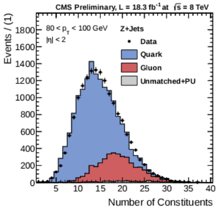


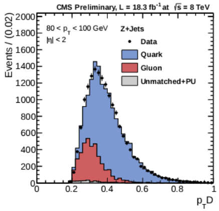
$\rightarrow$

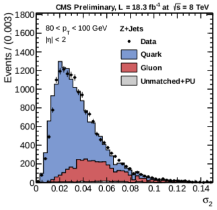


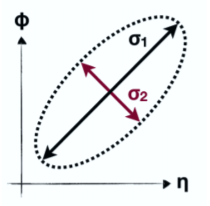
- What does it mean to train a neural network?
- Consider the XOR network
- There we set by hand, but could try to "train" the network
- Start with random weights and biases, reduce the loss function $C(x, y|w, b) = \sum_i |y_i^{\text{true}} - y(x_i)|^2$ where $i$ ranges over our 4 samples $(x_i, y_i)$ and $y(x_i)$ is the network output
  - Start with random weights so that different random features can be extracted by different nodes
  - As before, we train by trying to minimize the mean-squared error
  - There exists an efficient algorithm for doing this with neural networks called *backpropagation* (shown in backup)

# Quark-Gluon Jet Classification









- Quark and gluon jet production has subtly different properties
  - Gluon jets are more radiative so tend to be wider, produce more and softer particles, should tend to $C_A/C_F = 9/4$ at high $p_T$
- Proposed uses for discrimination in new physics studies, where more high-energy quark jets are expected than gluon jets
- At CMS, BDTs have been developed for q-g discrimination using variables constructed from PF inputs
  - Number of jet consitutents, $p_T D = \frac{\sqrt{\sum_i p_{T,i}^2}}{\sum_i p_{T,i}}$, jet ellipse axis lengths

"Quark-gluon Jet Discrimination At CMS", Cornelis for CMS (arxiv:1409.3072)

## Analysis of QG jets from CMS opendata

- We will use some simulated data from the CERN Open Data Portal to try out some of the techniques
  - Already online, so easy to download and analyse
  - But, was obtained using similar techniques to what Prof. Lee taught last week (analysing jets with CMSSW): producing ntuples from the MiniAOD output
- We can use SWAN to setup an environment and access the data
  - You could also work locally in the CMSSW environment you set up last time, but you will not be able to stream the data as I will be showing, you will have to download the file:

`http://opendata.cern.ch/record/12100/files/assets/cms/datascience/JetNtupleProducerTool/JetNTuple_QCD_RunII_13TeV_MC/JetNtuple_RunIISummer16_13TeV_MC_1.root`

- SWAN is the "Service for Web based ANalysis", which allows you (if you have a CERN account!) to easily run analyses on CERN's infrastructure, using a jupyter-based environment
  - `https://swan.web.cern.ch`

# Using SWAN

- Lets load up swan, create a directory for our analysis, open a notebook and try to load up the file:

```
import ROOT as rt
# A little setup for drawing
rt.gStyle.SetOptStat(0)
cvs = rt.TCanvas()
cvs.SetCanvasSize(800,600)

fname = ("root://eospublic.cern.ch//eos/opendata/cms/datascience/JetNtupleProducerTool/" +
         "JetNTuple_QCD_RunII_13TeV_MC/JetNtuple_RunIISummer16_13TeV_MC_1.root")
f = rt.TFile.Open(fname)
f.ls()
```

```
tree = f.AK4jets.jetTree
```

**For step by step instructions on starting swan, see p21. onward of:**
`https://indico.lip.pt/event/410/contributions/1043/attachments/1002/1149/Lisbon_TMVA_Tutorial.pdf`

**For more information on the various options, see the user guide:**

`https://root.cern.ch/download/doc/tmva/TMVAUsersGuide.pdf`

## Basic plots

- We have a tree with one entry per jet
- We can find light quark (gluon) jets using `isPhysUDS` (`isPhysG`)
- Lets draw a basic variable, the $p_T$ of the jets of the two different categories using the builtin `TTree::Draw` capabilities

```
tree.SetLineWidth(2)
tree.SetLineColor(rt.kBlue)
tree.Draw("jetPt>>gpt", "isPhysG", "")
tree.SetLineColor(rt.kRed)
tree.Draw("jetPt>>qpt", "isPhysUDS", "same")
rt.gpt.SetTitle(";p_{T} [GeV];# of Jets")
l=rt.TLegend(.6,.7,.9,.9)
l.AddEntry(rt.gpt, "Gluon")
l.AddEntry(rt.qpt, "UDS")
l.Draw()
cvs.Draw() # needed to display the output in Jupyter; don't need this on the command line
```

- The pt spectra are slightly different, we'd need to consider this in a real analysis
- Do the same for the qg separation variables:
  - `QG_mult` the multiplicity
  - `QG_axis2` the size of the minor axis
  - `QG_ptD` the ptD variable

## Prepare the Data for TMVA

- The Toolkit for MultiVariate Analysis (TMVA) is a library built into ROOT for doing machine learning
- We need to start by initializing TMVA and building a `Factory` which holds our analysis, and loading the data with a `DataLoader`

```python
# we can use this file later to analyse the results
outputFile = rt.TFile.Open('TMVA_output.root', 'recreate')

# we give it a name which it uses in the output, the outputfile, and
# some options (for instance, remove ! in front of Silent to suppress
# output)
factory = rt.TMVA.Factory('TMVAClassification', outputFile,
                          '!V:!Silent:Color:!DrawProgressBar:AnalysisType=Classification')

# create a dataloader and tell it the tree it sould use for signal and background
loader = rt.TMVA.DataLoader('dataset')
# in our case, the same tree holds signal and background, we will tell
# it later how to select the actual signal and background events we
# could also optionally add weights if we had several trees for
# e.g. different background processes
loader.AddSignalTree(tree)
loader.AddBackgroundTree(tree)

# now we define the variables to be used in the analysis, we can also
# give it a name for displaying nicely
loader.AddVariable('QG_mult', "multiplicity", "")
loader.AddVariable('QG_axis2', "#sigma_{2}", "")
loader.AddVariable('QG_ptD', "p_{T}D", "")

# finally tell it how to read signal and background and prepare the test/train
loader.PrepareTrainingAndTestTree("isPhysUDS", "isPhysG", # signal cut, then background cut
          "nTrain_Signal=4000:nTrain_Background=7000:SplitMode=Random:NormMode=NumEvents:!V")
```

## Train with TMVA

- Now we can "book" the methods we want TMVA to run, then train them, test them and evaluate them
- We can run as many as we want. Lets do a BDT and a Neural Network, see the user guide for more options
  - In the NN, the `HiddenLayers` takes a comma separated list of the number of neurons in each layer. `N` is the number of input variables

```
# Boosted Decision Trees
factory.BookMethod(loader,rt.TMVA.Types.kBDT, "BDT",
                   "!V:NTrees=200:MinNodeSize=2.5%:MaxDepth=2:BoostType=AdaBoost:AdaBoostBeta=0.5:"+
                   "UseBaggedBoost:BaggedSampleFraction=0.5:SeparationType=GiniIndex:nCuts=20")

# Multi-Layer Perceptron (= Neural Network)
factory.BookMethod(loader, rt.TMVA.Types.kMLP, "MLP",
                   "!H:!V:NeuronType=tanh:VarTransform=N:NCycles=100:HiddenLayers=N+5:"+
                   "TestRate=5:!UseRegulator")

# Train
factory.TrainAllMethods()
# Test
factory.TestAllMethods()
# Evaluate, these will compute various quantities of interest and output them into the output file
factory.EvaluateAllMethods()

# the output file will have the results of the training
outputFile.Close()
```

## Output Distributions

- A good overtraining check is to look at the output distribution for signal and background, comparing testing and training
- A well-trained model should have those distributions match each other
- An overtrained model will have the test distribution performing worse than the training

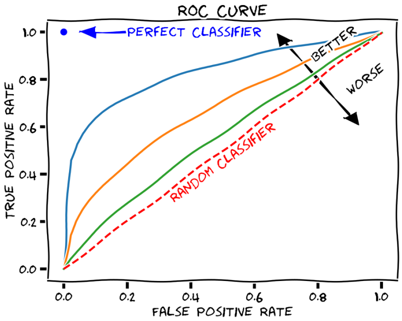
```python
# save pdfs instead of png
rt.TMVA.Config.Instance().fVariablePlotting.fPlotFormat = \
    rt.TMVA.Config.Instance().fVariablePlotting.kPDF
TMVA.mvas("TMVA_output.root", rt.TMVA.kCompareType)

# TMVA will make a canvas for each machine learning method
# check which is which
[c.GetTitle() for c in rt.gROOT.GetListOfCanvases()]

# and draw the one of interest
rt.gROOT.GetListOfCanvases().At(0).Draw()
```

## ROC Curves

- The Receiver Operating Characteristic (ROC) curve was defined during WW2 for displaying the abilities of radar receiver operators
- Gives the true positive rate (correctly identified signal) versus false positive rate (background incorrectly identified as signal) [TMVA shows 1−FPR]
- The area under the curve (AUC) is often used to summarize a classifier's performance
  - 0.5: completely random classifier
  - 1.0: perfect classifier



## ROC curves in TMVA (and python)

- Note, TMVA puts TPR on the x-axis, and shows 1-FPR on the y-axis, so we want to have the classifier move to the top right, not top left

```python
# we can do it through a factory method
cvs = factory.GetROCCurve(loader)
cvs.Draw()
bdt_auc = factory.GetROCIntegral(loader, "BDT")
mlp_auc = factory.GetROCIntegral(loader, "MLP")
print(f"AUCs: BDT {bdt_auc:.3f}, MLP {mlp_auc:.3f}")

# or similar to the way we did the output distributions, using the output file
rt.TMVA.efficiencies("dataset", "TMVA_output.root")
[c.GetTitle() for c in rt.gROOT.GetListOfCanvases()]
rt.c.Draw()
```

- Note that on the command line, we can run `rt.TMVA.TMVAGui("outputFile.root")` (or `TMVA::TMVAGui("outputFile.root")` in a root command line) and see a GUI with options to display various results, and some of the GUI options (mvaeffs) haven't been ported to be usable in the notebook (as far as I can tell), and they say they will modernize so sometime in the future all the plots should be drawable with the factory

## TMVA Reader

- When you are happy with your training, you will want to apply it to the data
- To do this, we save out the trained MVA and load it with a `Reader`
- In python we need to load our variables into an `array`, so it can interface with the C++ code (which uses pointers)
- The weights are saved after your training in `<dataset name>/weights/<factory name>_<method>`

```python
from array import array
reader = rt.TMVA.Reader("!Color:!Silent:!V")
mult = array('f', [0.])
axis2 = array('f', [0.])
ptD = array('f', [0.])
reader.AddVariable("QG_mult", mult)
reader.AddVariable("QG_axis2", axis2)
reader.AddVariable("QG_ptD", ptD)
reader.BookMVA("BDT", "dataset/weights/TMVAClassification_BDT.weights.xml")

# set your variables and evaluate (you would do this in an event loop)
ptD[0] = 0.8; mult[0] = 5; axis2[0] = 0.03
reader.EvaluateMVA("BDT")
```

# Running over a sample

**Lets take the second sample in the public data, and run our analyser, saving the output for quark v gluon (of course in real data we wouldn't have that tag. . . ).**

```
fname = ("root://eospublic.cern.ch//eos/opendata/cms/datascience/JetNtupleProducerTool/" +
         "JetNTuple_QCD_RunII_13TeV_MC/JetNtuple_RunIISummer16_13TeV_MC_2.root")
f = rt.TFile.Open(fname)
cvs = rt.TCanvas()
hq = rt.TH1F("quark", ";BDT output;Arb. Units", 100, -0.6, 0.6)
hg = rt.TH1F("gluon", ";BDT output;Arb. Units", 100, -0.6, 0.6)
for jet in f.AK4jets.jetTree:
    if not jet.isPhysUDS and not jet.isPhysG: continue
    ptD[0] = jet.QG_ptD
    mult[0] = jet.QG_mult
    axis2[0] = jet.QG_axis2
    bdt = reader.EvaluateMVA("BDT")
    if jet.isPhysUDS: hq.Fill(bdt)
    if jet.isPhysG: hg.Fill(bdt)

# DrawNormalized divides each histogram by the total number of entries
# before plotting, so we can see the shape of the distribution and
# ignore the difference in number of light quark v gluon jets
hg.DrawNormalized()
hq.DrawNormalized('same')
cvs.Draw()
```

## Exercises

- Draw all the input variable distributions
- Try changing the parameters of the ML methods, can you improve the performance?
  - For the BDT you could try changing the learning rate of AdaBoost (called `AdaBoostBeta` by TMVA) and the number of trees
  - For the neural network you could try changing the size and number of hidden layers
- Look through the TMVA tutorials for more examples of how to use TMVA
  - Note, that some of what I showed today is a bit of an older style, which is slowly being replaced. The current version of the tutorials use `RDataFrame` and `RReader` for instance, you should look into these, since it will be the future, but most of the material online will be closer to what I showed today (though a lot will be in C++!)
  - There are also options to integrate TMVAs data loading and evaluating facilities with modern libraries like XGBoost for BDTs, Tensorflow for Deep Learning, and so on

## Backup

## Analogy: Steepest descent

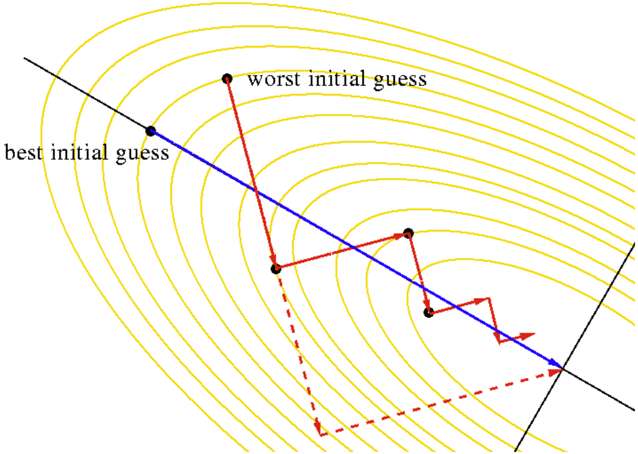
Question: How do we actually train these networks?

- A climber is trying to find his way down a mountain in deep fog, how should he proceed?
- One idea is to try to always go downhill the fastest way possible
- So, he figures out which direction has the steepest descent (ie which way is downhill), then takes a step in that direction
- After the step, he checks again, and takes another step
- He keeps proceeding in this manner until he cant go downhill anymore, he's reached the bottom

## Gradient Descent

- From calculus, $\nabla f(\mathrm{x})$ gives the direction of largest increase of $f$ at $x$ (if its 0, we are at a minimum and done)
- Equivalently, $-\nabla f(\mathrm{x})$ gives direction of largest decrease, so $f(\mathrm{x} - \gamma \nabla f(\mathrm{x})) < f(\mathrm{x})$ (at least, for some $\gamma$ small enough)
- We will define a sequence $\mathrm{x}_i$ to find the minimum:
  - Start with some random position $\mathrm{x}_0$
  - Iterate:
    - Find $\mathrm{x}_{n+1} = \mathrm{x}_n - \gamma_n \nabla f(\mathrm{x}_n)$
    - Stop if $|f(\mathrm{x}_{n+1}) - f(\mathrm{x}_n)| < \epsilon$, i.e. we're not reducing further, so we're close to the minimum
  - Return the final $\mathrm{x}_n$
- $\gamma_n$ can be different for each iteration, extensions to GD keep track of how quickly parameters are changing to update $\gamma$ also
- $\epsilon$ is the *tolerance*, how close to a minima do we need to be before stopping (again, there are various criteria we could choose here)

## Example function



- Lines are contours of equal value
- Shows how the algorithm picks out different paths depending on starting point

## Training Neural Networks: Backpropagation

- The algorithm to train neural networks is called **backpropagation**
- Its essentially a gradient descent implemented taking the network structure into account to speed up evaluation of the partials
- To apply gradient descent, we need a function to minimize, this is our loss function from earlier
  - $L(x_i; \theta) = \sum_i |f(x_i; \theta) - y_i|^2$ for inputs $x_i$ with known output $y_i$
- We start with the parameters $\theta$ set to arbitrary values, usually picked from e.g. the unit gaussian
- We run a forward pass through the network and calculate the loss, keeping track of the values at the intermediate nodes
- Using the chain rule, calculate the derivates *for all weights* backward from the loss to the higher layers to the inputs, in a single pass
- Propagate changes based on the gradient $\Delta\theta_i = -\eta \frac{\partial L}{\partial \theta_i}$
- For more on how backpropagation works: http://neuralnetworksanddeeplearning.com/chap2.html