Statistics and Data Science: Lecture 3

Roger Barlow
Huddersfield University

Cockcroft Lecture Series

6^{th} June 2022

We are very good at telling cats from dogs...

Rapidly, accurately, by using partial or even inaccurate information

Feature-space very far from raw pixel space

Not using flowchart-type algorithms

Classification

A very general statistics problem

- Physicist: is this event signal or background?
- Astronomer: is this blob a star or a galaxy?
- Engineer: is this girder reliable or will it fracture?
- Doctor: has this patient got flu or Ebola virus?
- Banker: will this loan be repaid or defaulted?
- Employer: will this applicant be a good employee?
- Accelerator physicist: is this bunch going to reach the target or be lost?

Often many such cases present and need rapid decisions.

Can we imitate the brain's decision making process?

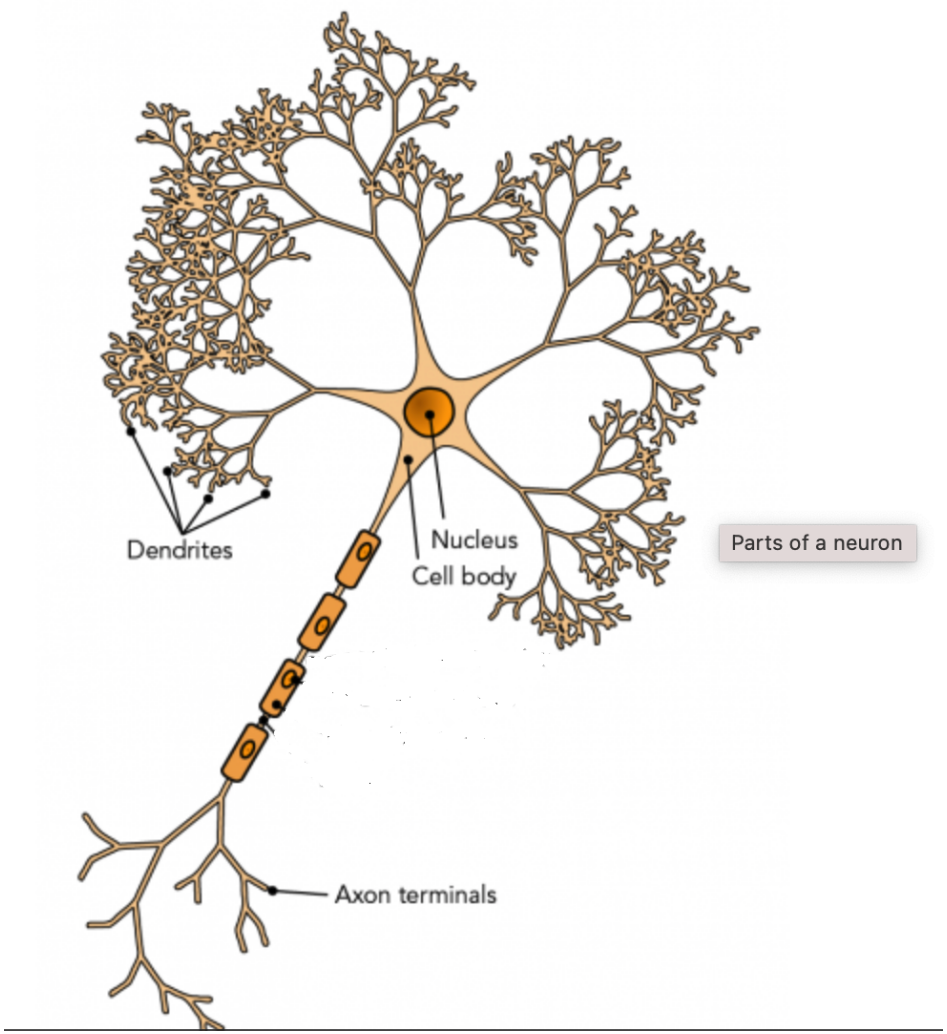
The brain

A very over-simplified view

The brain is made of MANY (around 86,000,000,000) neurons.

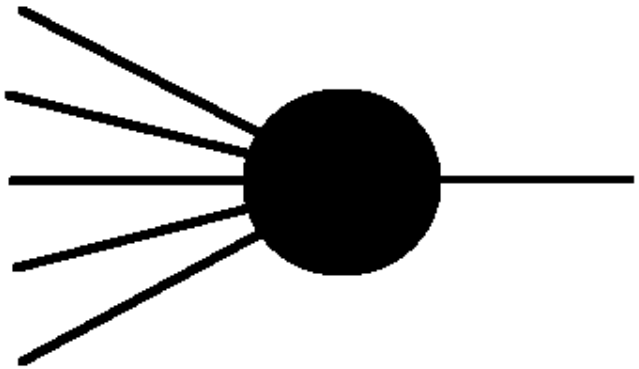
Each has MANY inputs (dendrites: from eyes, ears, etc and from other neurons)
These are combined to form a value

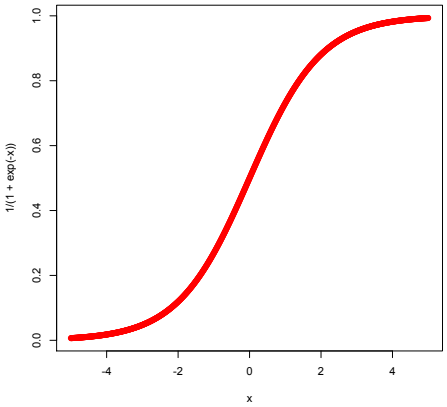
Each then has MANY outputs (axon: to hands, tongue, etc and to other neutrons)



The node

Imitating the neuron





Suppose node i has inputs U_j

- Simplest arrangement: output $\sum_j U_j$
- Better: output $\sum_j w_{ij} U_j$ to weight more important inputs and allow for positive and negative factors
- Even better: feed this through some thresholding function: $y_i = f(\sum_j w_{ij} U_j + z_i)$

Various forms of f used:

Logistic: $f(x) = 1/(1 + e^{-x})$

Tanh: $f(x) = (e^x - e^{-x})/(e^x + e^{-x})$

ReLU: $f(x) = xH(x)$

Google Swish: $f(x) = x/(1 + e^{-x})$

The network

Imitating the brain

Multilayer Perceptron or ANN

Nodes arranged in layers

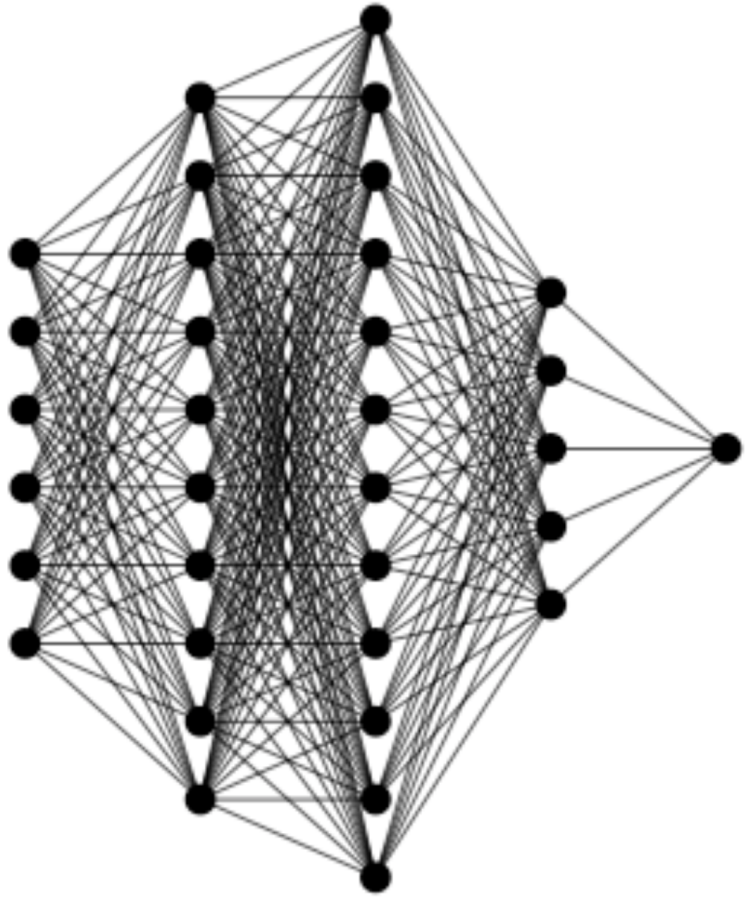
Each node has inputs from all nodes in previous layer, outputs to all nodes in next layer, but not to any other layers

First layer is input - defined by data size

Final layer is output - ideally 1 for S and 0 for B (or whatever)

Intermediate layers are 'hidden'

Lends itself to parallelisation of all nodes within a layer



Easy to code and run - but contains many weights (one for each thin line) and thresholds. How do you set them to give the right answers?

Training

Back-propagation

Let $w_{ij}^{(\ell)}$ be weight to node i in layer ℓ from node j in previous layer

If some instance has desired output T (0 or 1 for B or S) and network output U, define badness $B = \frac{1}{2}(U - T)^2$

Adjust each $w_{ij}^{(\ell)}$ by $-\alpha \frac{\partial B}{\partial w_{ij}^{(\ell)}}$ (thresholds $z_i^{(\ell)}$ similar)

α is a 'learning parameter', typically 0.1

For final layer: $U = f(S)$ with $S = \sum_j w_{1j}^{\ell} U_j^{\ell-1} - z_1^{(\ell)}$ and

$$\frac{\partial B}{\partial w_{ij}^{(\ell)}} = (U - T) f'(S) \frac{\partial S}{\partial w_{ij}^{(\ell)}} = (U - T) f'(S) U_j^{(\ell-1)}$$

$f'(S)$ from simple algebra - for logistic, $f'(S) = f(S)(1 - f(S))$

For previous layers, just continue the differentiations following the network backwards (codes neatly using recursion)

Run over the whole sample, with equal numbers of signal and background.

Do this many times if necessary (epochs), adjusting as you go

Training-testing-validation

Overtraining

Training on the same data set many times eventually leads the network to recognising individual events. Gives bad results and misleadingly good performance measure.

Testing

Separate data into training and testing samples (maybe in 80:20 ratio). Train on the large sample but measure performance on the small sample, and stop training when that stops improving.

Validation

May need to know the performance (e.g. for absolute measurements, and for comparing different techniques). 2-stage train-test cycle overoptimistic as you stop when the test data result looks good. So need 3 samples.

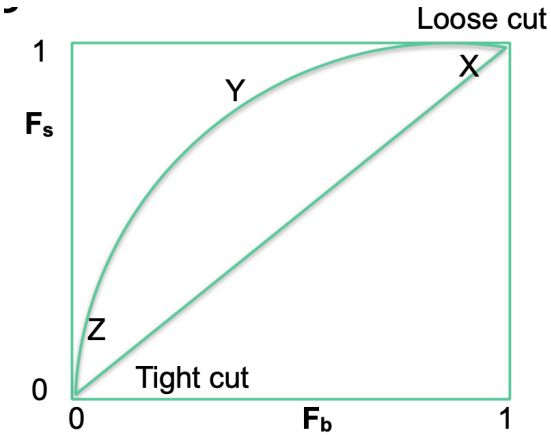
The terms 'validation' and 'test' are interchanged in the literature.

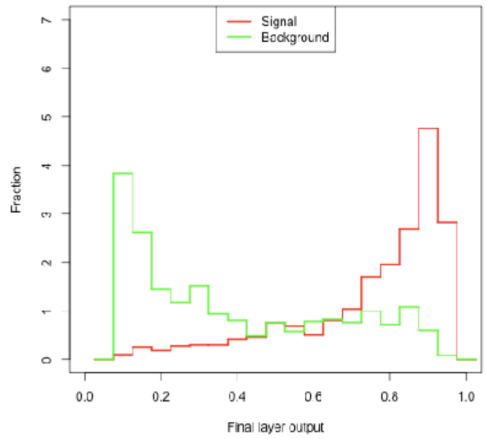
Performance

ROC plots - Receiver Operating Characteristics

Typical network output -where should you place the cut?
0.5?? Where the histograms cross??
You need more information

- The actual S and B fractions in the data (unlikely to be 50:50)
- The 'cost' of a Type I error (rejecting an S) and a Type II error (accepting a B)





Vary cut from 0 to 1 and plot fractions of B and S accepted
Start at top right X - 100% of both
Increase cut, move left (quickly) and down (slowly) Y to bottom left Z.
Diagonal is zero discrimination. The further from it the better.
Used to compare networks, and to combine with fraction and cost info to give cut

ANNs for Regression

or fitting

Suppose you want to model $y = f(x_1, x_2 \ldots x_n)$ but with not preconceived form for f?

Feed the x_i into a neural network, giving output u, and adjust weights to minimise $\frac{1}{2}\sum(y_j - u_j)^2$
This is the same as training for classification except that the y values are continuous, not just 0 and 1
(May need to scale the final output)

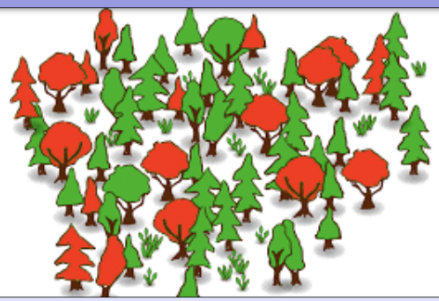
Boosted Decision Trees

Another ML technique

Basic Tree: Find most sensitive variable and best cut on it, dividing sample in two (branching)
Repeat in each of 2,4,8,16… branches until all events classified

Random Forest: as Bagging, but each tree restricted to random subset of input variables. Avoids trees all going for the obvious and looking the same

Bagging (Bootstrap Aggregation): Avoid overtraining by creating many trees, sampling with replacement. Then take majority vote

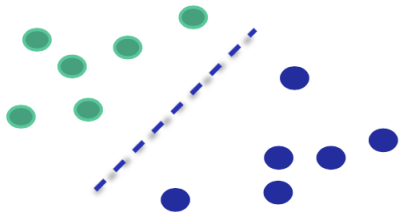


Boosting: weight events that early trees find hard to classify, so later trees focus on them

Support Vector Machines

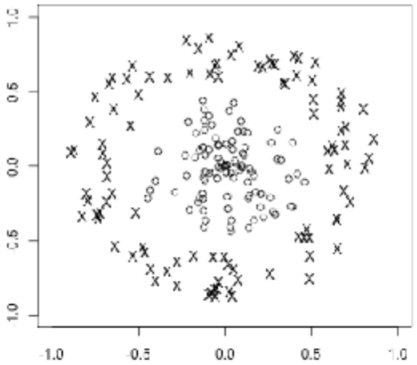
Yet another ML technique

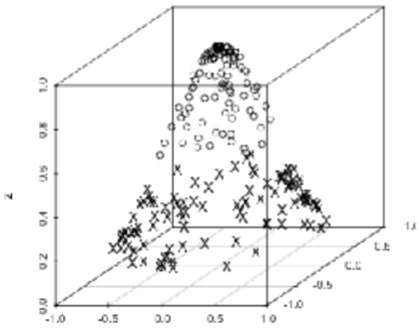
Draw the best line/plane/hyperplane dividing events into 2 classes (widest margin + fewest misclassified)



In choosing this line/plane/whatever only the datapoints near the boundary need be considered - the 'support vector'

As necessary, distort into higher dimensions to make this work





Comparing the 3 methods

All 3 share the problems of overtraining, testing and validation, and their performance is evaluated using ROC plots

Problem is different so no general 'best buy' technique, just guidance:

- SVMs do well when the signal/background samples can be cleanly demarcated, even if this is some complicated shape region
- For logical (true/false) or classified data, BDTs are a natural choice. For numeric data, ANNs look more appropriate

Multiple outputs

Distinguishing more than two patterns
Classic example: recognising written characters
Output layer with several nodes, one for each pattern.
Training as before
Running: take node with highest value

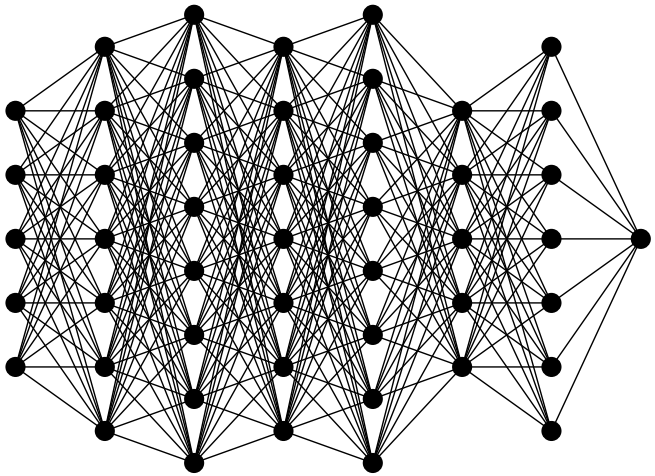
S and B - two output nodes or 1?

Multinode output can give null result - even highest value is very small
Makes sense for character recognition - might not be a character to recognise - but not in cases which must be one or the other
'Highest value' choice does not include prior probability and cost factor

Deep Learning

No generally accepted definition

Hot topic - much activity
Lots of funding and effort from tech companies
Use GPUs and other parallelisation techniques
Some impressive results



Neural networks with many layers (Whatever many means. Certainly more than 2.)

Back-propagation fails due to noise

Various cunning compute-intensive feature-extracting techniques used

CNNs

Convolutional Neural Networks

Typical application in image processing. Is a cat anywhere in this picture?
Convolutional layer

1. Apply ANN to small region in top left corner
2. Scan across and then down, usually 1 pixel at a time, and repeat
3. Train looking for 'features' present in some but not all regions
4. Do this with several ANNs. Ensure features useful & not duplicated

This is followed by a Pooling layer

1. Split the image into small (typically 2x2) regions
2. Within each region, give maximum(or average) CNN output

Then more C+P layers applied to the outputs of all the ANNs. A first layer might pick out eyes, ears, and claws. A second might use these to form larger features like faces and legs. A third would pick out cats and dogs. As the process proceeds the scale increases, and the fine detail is incorporated in the features found.

GANs - Generative Adversarial Networks

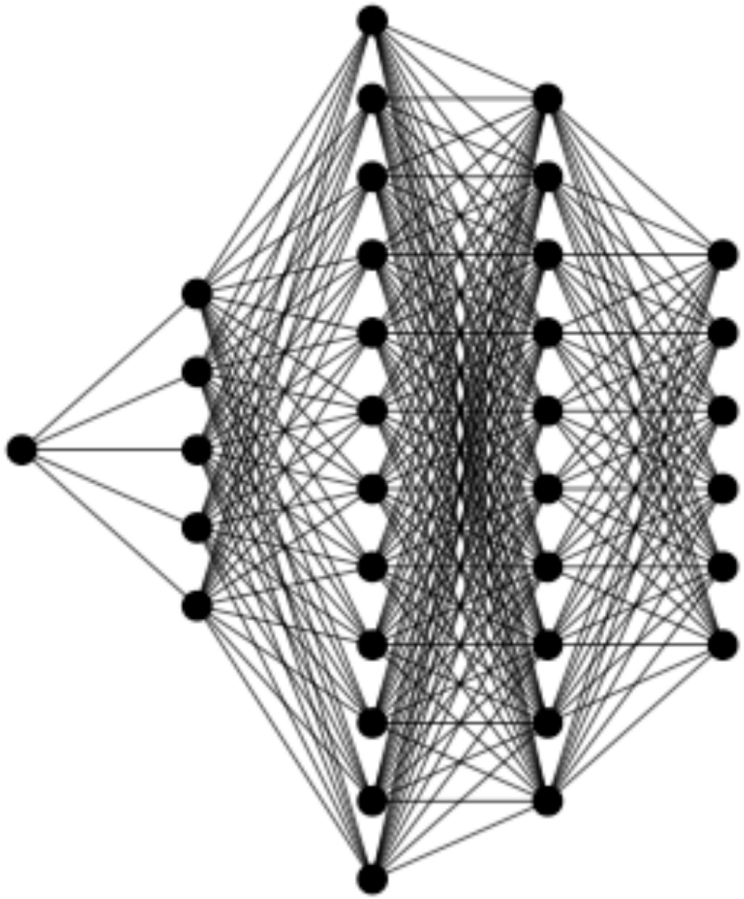
Creating simulated data

Run a network backward and feed it random numbers

Outputs of this network have same structure as original data

Train a second network, with usual topology, to distinguish between original data and generated data

Train backward network to reduce the performance score of the analyser network



End of training is a network which produces simulated data indistnguishble from the real thing

TensorFlow - preliminary

Has become the standard platform for ML

Installation (do once. Details depends on your system)

```
pip install tensorflow
```

Import packages

```
import tensorflow as tf
from tensorflow.keras import Input
from tensorflow.keras.Models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np

import matlib.pyplot as plt
```

Keras is an ML package that sits on top of TensorFlow

Tensorflow (Set up)

Define a model - suppose 5 - 8 - 4 - 1 topology

```
model=tf.keras.Sequential()
model.add(tf.keras.Input(shape=(5)))
model.add(tf.keras.layers.Dense(8,activation='sigmoid')
model.add(tf.keras.layers.Dense(4,activation='sigmoid')
model.add(tf.keras.layers.Dense(1,activation='sigmoid')
model.compile(loss='binary_crossentropy')
```

Let's invent some data

```
smear=3.5.  # Adjust this.  Maybe use keyboard input?
key=[] # empty
data=np.ndarray((0,7)) # empty but structured
for i in range(5000):  # make 10000 values
    key=np.append(key,0)
    data=np.vstack([data,[1,2,3,4,5,6,7]+np.random.normal(0,smear,7)])
    key=np.append(key,1)
    data=np.vstack([data,[7,6,5,4,3,2,1]+np.random.normal(0,smear,7)])
```

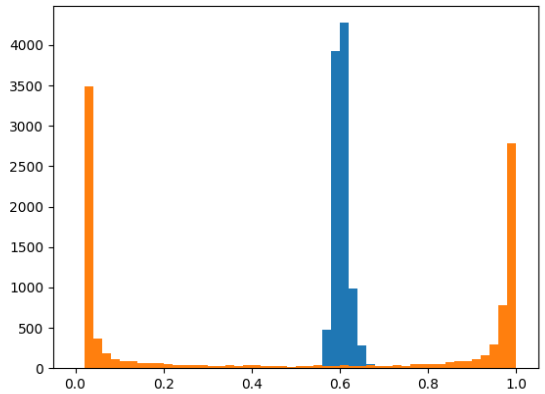
Tensorflow (Running it)

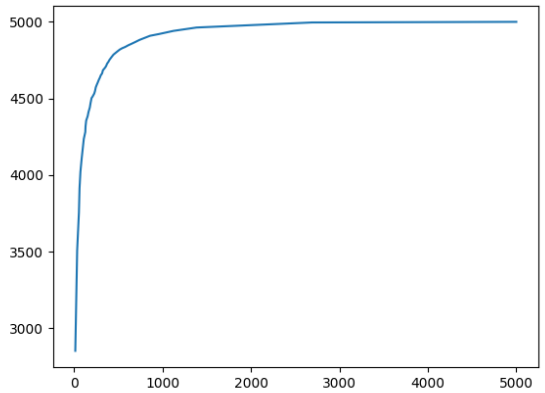
Try it!

```
before=model.predict(data)
model.fit(data,key,epochs=10)
after=model.predict(data)
```

How did it do?!

```
binning=arange(0,1.01,.02)
plot.hist(before,binning,color='blue')
plt.hist(after,binning,color='orange')
hbak=plt.hist(after[kk==0],binning)
hsig=plt.hist(after[kk==1],binning)
# draw ROC plot
plt.plot(sum(hbak[0])-np.cumsum(hbak[0]),
       sum(hsig[0])-np.cumsum(hsig[0]))
```





Exercise

1. Form groups as usual. All in python this week (sorry!)
2. Download data files `http://barlow.web.cern.ch/barlow/good.dat` and `bad.dat`. They are 5000 samples of 15 numbers representing some monitor signal. Plot a few of them to see what they look like
3. Use TensorFlow to discriminate between good and bad signals. Remember to separate training and testing. Try different topologies and different activation functions and other options. Draw the ROC plot for your best result (and maybe others)
4. In supposedly real life, good signals outnumber bad signals by a factor of 2000. A bad signal results in beam loss and sprays radiation everywhere doing an estimated £350 of damage. You can abort the beam, which loses a good beam shot which is worth £2 to the users. Where should you put the cut on the network output?
5. Prepare some plots and slides, and come back at 2 pm and present your results in a short talk.