Data Mining 2

Dr Kieran Murphy

Department of Department of Computing and Mathematics,
INSTITUTION.
(kmurphy@wit.ie)

Spring Semester, 2021

RESOURCE OUTLINE LABEL

- Fundamental concepts in neural networks
- Neural networks using scikit-learn.

Outline

Introduction

Aim

Introduce the fundamental concepts in neural networks and deep learning, and demonstrate the basics of modelling and training of networks using `scikit` and `Tensorflow`, using toy data sets (XOR, ...) and the MIST Handwritten Digits dataset.

Formalising a Simple Decision Process I

Say you want to decide whether you are going to attend a cheese festival this weekend. And say, the following three factors go into your decision:

Is the weather good?

Will a friend go also?

Is it near some pubs?

Go to festival ?
(Yes/No)

Assume

- Factors are binary: No/Yes, False/True, ... $\Longrightarrow$ 0/1
- Each input factor has its relative importance, given by its weight and we add effects.
- We have an initial threshold (inertial/bias...) that we need to exceed before we will decide "Yes".

Formalising a Simple Decision Process I

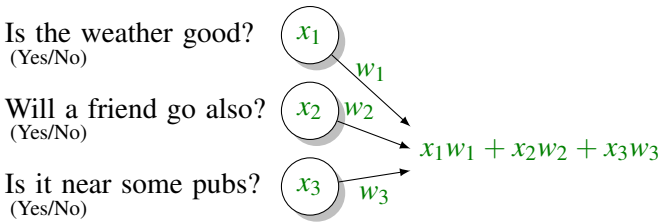
Say you want to decide whether you are going to attend a cheese festival this weekend. And say, the following three factors go into your decision:

Assume

- Factors are binary: No/Yes, False/True, ... $\Longrightarrow$ 0/1
- Each input factor has its relative importance, given by its weight and we add effects.
- We have an initial threshold (inertial/bias...) that we need to exceed before we will decide "Yes".

Formalising a Simple Decision Process I

Say you want to decide whether you are going to attend a cheese festival this weekend. And say, the following three factors go into your decision:

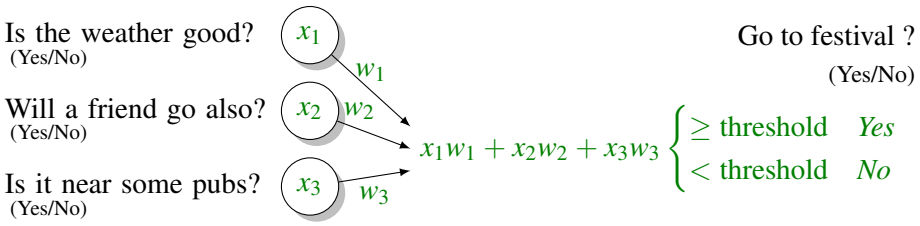


Assume

- Factors are binary: No/Yes, False/True, ... $\Longrightarrow$ 0/1
- Each input factor has its relative importance, given by its weight and we add effects.
- We have an initial threshold (inertial/bias...) that we need to exceed before we will decide "Yes".

Formalising a Simple Decision Process I

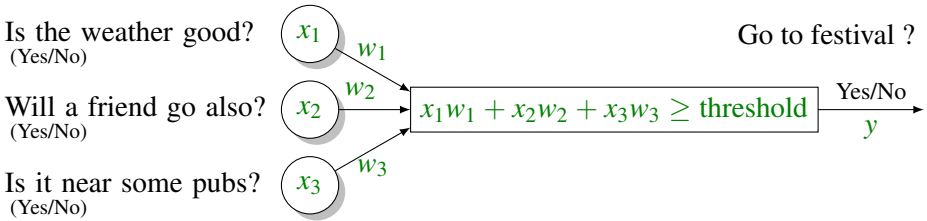
Say you want to decide whether you are going to attend a cheese festival this weekend. And say, the following three factors go into your decision:



Assume

- Factors are binary: No/Yes, False/True, ... $\Longrightarrow$ 0/1
- Each input factor has its relative importance, given by its weight and we add effects.
- We have an initial threshold (inertial/bias...) that we need to exceed before we will decide "Yes".

Formalising a Simple Decision Process II



For given input vector, $\boldsymbol{x}$, the decision is controlled by the weights and threshold:

- If you want decision to be 'Yes' whenever any two input factors are 'Yes', then set

$$w_1 = w_2 = w_3 = 1 \qquad \text{and} \qquad \text{threshold} = 2$$

- Fear of dying of thirst dominates your decision making process, then set

$$w_1 = w_2 = 1, \quad w_3 = 10 \qquad \text{and} \qquad \text{threshold} = 11$$

Formalising a Simple Decision Process III

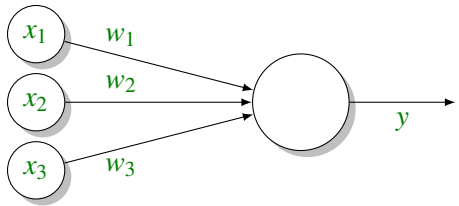
Rewriting function

$$y = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + x_3 w_3 \geq \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

so that the comparison is against zero we have

$$y = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + x_3 w_3 + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $b = -\text{threshold}$ is called the bias.



Formalising a Simple Decision Process III

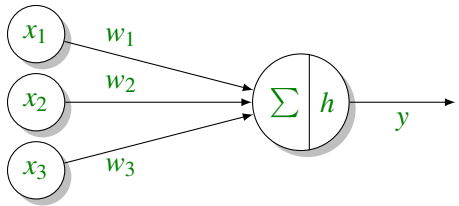
Rewriting function

$$y = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + x_3 w_3 \geq \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

so that the comparison is against zero we have

$$y = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + x_3 w_3 + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $b = -\text{threshold}$ is called the bias. This can treated as a (smooth) summation function followed by a (non-linear) activation function.



Formalising a Simple Decision Process III

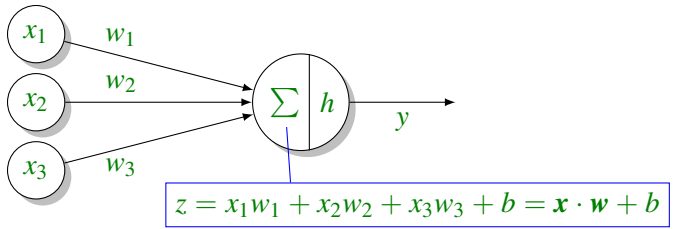
Rewriting function

$$y = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + x_3 w_3 \geq \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

so that the comparison is against zero we have

$$y = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + x_3 w_3 + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $b = -\text{threshold}$ is called the bias. This can treated as a (smooth) summation function followed by a (non-linear) activation function.



Formalising a Simple Decision Process III

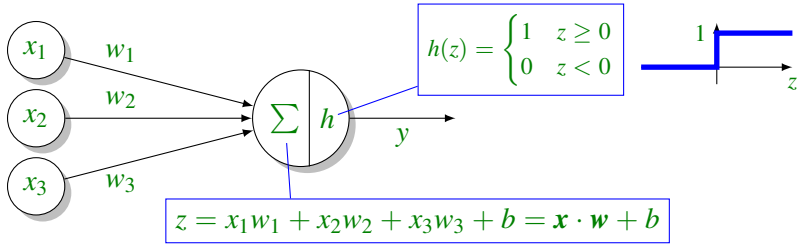
Rewriting function

$$y = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + x_3 w_3 \geq \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

so that the comparison is against zero we have

$$y = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + x_3 w_3 + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $b = -\text{threshold}$ is called the bias. This can treated as a (smooth) summation function followed by a (non-linear) activation function.



Formalising a Simple Decision Process III

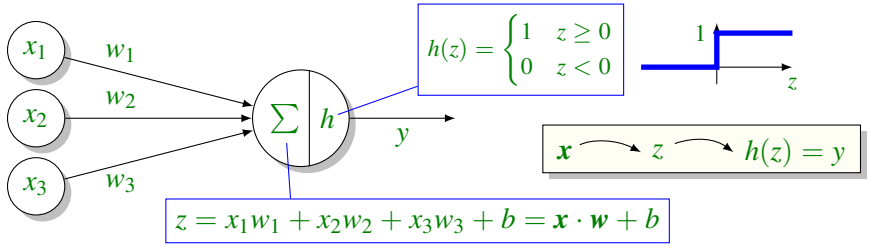
Rewriting function

$$y = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + x_3 w_3 \geq \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

so that the comparison is against zero we have

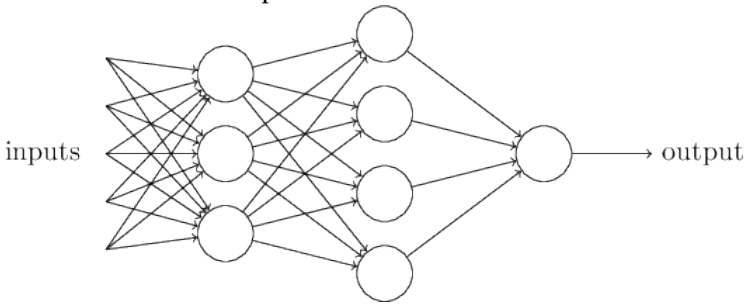
$$y = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + x_3 w_3 + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $b = -\text{threshold}$ is called the bias. This can treated as a (smooth) summation function followed by a (non-linear) activation function.



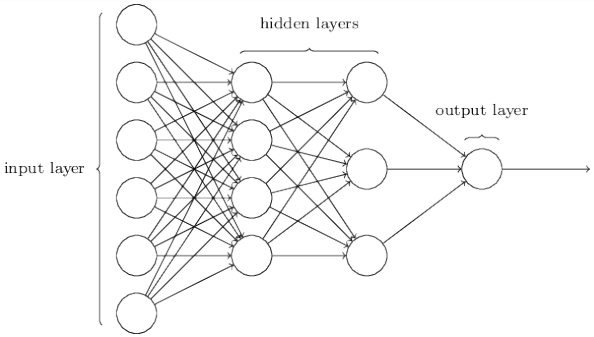
Arbitrary Complex Decisions $\rightsquigarrow$ Neural Networks I

Arbitrary complex decisions can be constructed by chaining/linking simple decisions, where the output of some decisions are used as input for other decisions.

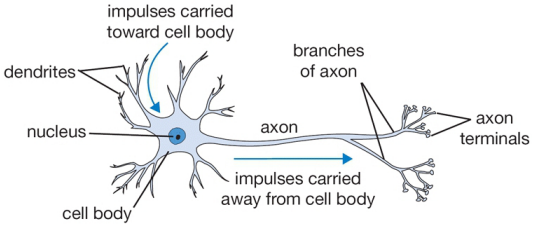


- Each simple decision node is called a neuron.
- Notes that while a decision might have arbitrary number of inputs it only has one output. (The multiple links out represent the same output.)
- In the simplest network the effect of decisions moves in one direction, this is called a feed forward network.

Arbitrary Complex Decisions ⇝ Neural Networks II



More than one hidden layer is "Deep learning"



- The input and outputs are typically represented as their own neurons, with the other neurons named hidden layers
- The biological interpretation of a neuron is this: when it emits a 1 this is equivalent to 'firing' an electrical pulse, and when it is 0 this is when it is not firing. The bias indicates how difficult it is for this particular node to send out a signal.

When to Consider Neural Networks?

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant.

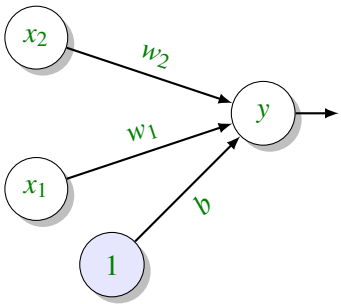
Examples

- Financial modelling — predicting the stock market
- Time series prediction — climate, weather, seizures
- Computer games — intelligent agents, chess, backgammon
- Robotics — autonomous adaptable robots
- Pattern recognition — speech recognition, seismic activity, sonar signals
- Data analysis — data compression, data mining
- Bioinformatics — DNA sequencing, alignment

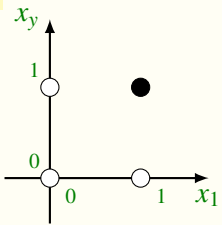
Implementation of Simple Logic Gates — AND

Function AND

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



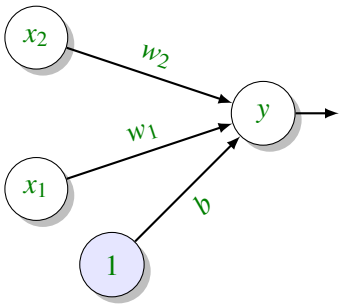
SVM



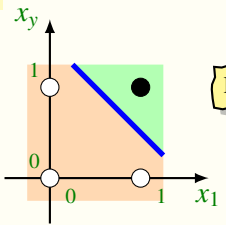
Implementation of Simple Logic Gates — AND

Function AND

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



SVM

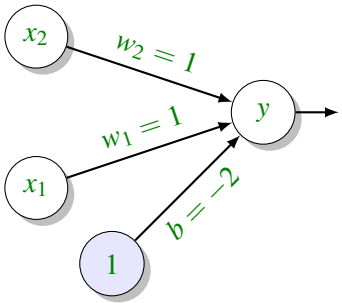


Linearly separable

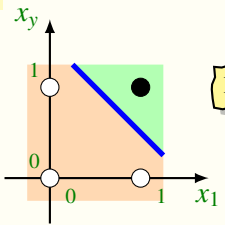
Implementation of Simple Logic Gates — AND

Function AND

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



SVM



Linearly separable

Calculation

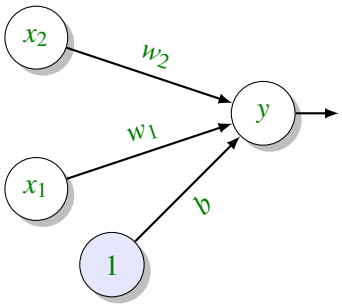
x_1	x_2	y	$z = \vec{x} \cdot \vec{w} + b$	$y = h(z)$
0	0	0	-2	0
0	1	0	-1	0
1	0	0	-1	0
1	1	1	0	1

Solution not unique — how many lines can separate classes in above SVM?

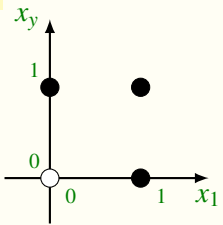
Implementation of Simple Logic Gates — OR

Function OR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



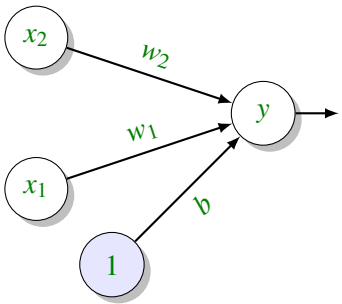
SVM



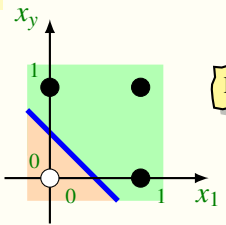
Implementation of Simple Logic Gates — OR

Function OR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



SVM

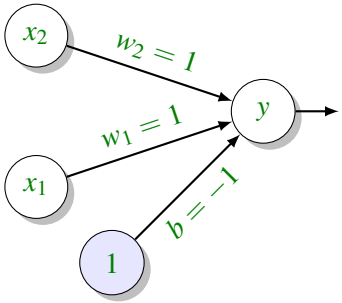


Linearly separable

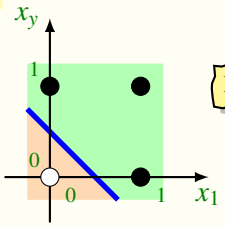
Implementation of Simple Logic Gates — OR

Function OR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



SVM



Linearly separable

Calculation

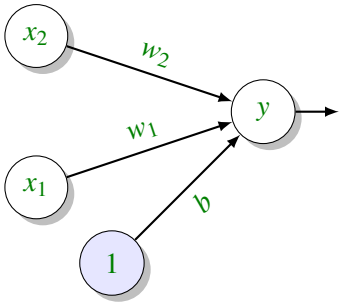
x_1	x_2	y	$z = \vec{x} \cdot \vec{w} + b$	$y = h(z)$
0	0	0	-1	0
0	1	0	0	1
1	0	0	0	1
1	1	1	0	1

Solution not unique — how many lines can separate classes in above SVM?

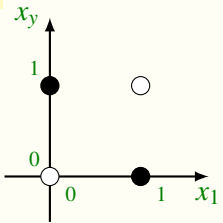
Implementation of Simple Logic Gates — XOR

Function $\vee$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



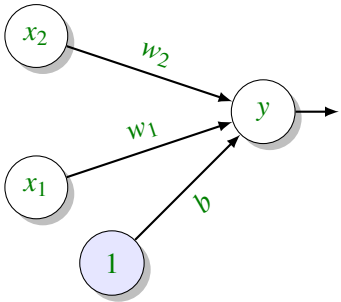
SVM



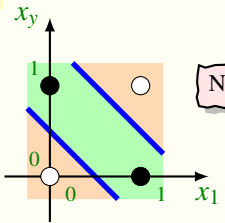
Implementation of Simple Logic Gates — XOR

Function $\vee$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



SVM

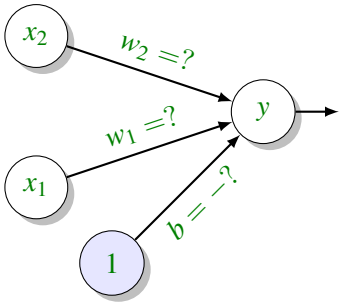


Not linearly separable

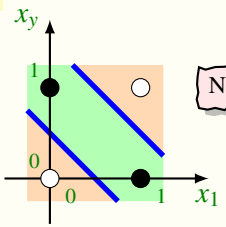
Implementation of Simple Logic Gates — XOR

Function $\vee$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



SVM



Not linearly separable

Calculation

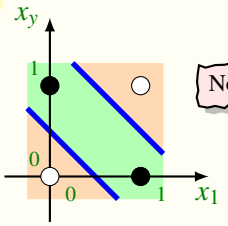
x_1	x_2	y	$z = \vec{x} \cdot \vec{w} + b \quad y = h(z)$
0	0	0	
0	1	1	
1	0	1	
1	1	0	

Implementation of Simple Logic Gates — XOR

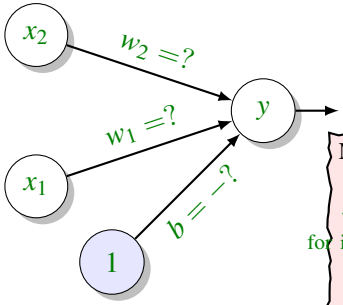
Function $\vee$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

SVM



Not linearly separable



Calculation

Not possible because require

$\underbrace{b < 0}_{\text{for input } [0,0]}$ $\underbrace{0 \geq w_1 + b}_{\text{for input } [1,0]}$ $\underbrace{0 \geq w_2 + b}_{\text{for input } [0,1]}$ $\underbrace{w_1 + w_2 + b < 0}_{\text{for input } [1,1]}$

$\Longrightarrow$ Need hidden layer

Expressive Capabilities of ANNs

Boolean functions

- Every boolean function* can be represented by network with single hidden layer ... but might require exponential (in number of inputs) hidden units.

Continuous functions

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

See Chapter 3 Neural Networks and Deep Learning for a nice non technical argument why these claims are true.

*this is not just the elementary functions ($\wedge$,$\vee$,$\neg$,etc.) but any logical expression

So What Now ?

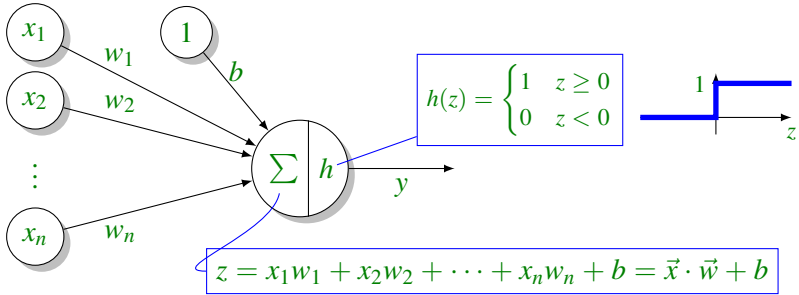
- How do determine the design/structure of our network?
  - Number and size of hidden layers?
  - How do decisions flow? feed forward networks, recurrent networks
  - Level of interconnections? fully connected[†]/ convolution[‡]
  - Which activation function should we use?
  - Use on-hot encoding for output?
- How do we compute the output of a network? forward propagation
- How we we measure the accuracy of our neural network?
  - Have cost functions measuring the difference between computed output and expected output, similar to regression/classification models.
- How do we train the network? back propagation
  - Determine optimal weights and biases — proper treatment requires calculus — we won't go there today.
- Standard modelling concerns
  - Overfitting, bias — standard techniques such as regularisation but many, many network specific techniques (eg. dropout).

[†]Are all neurons in each layer connected to all neurons in the next layer

[‡]Localised fully connected with multiple hidden layers.

Outline

Concept: Perceptrons



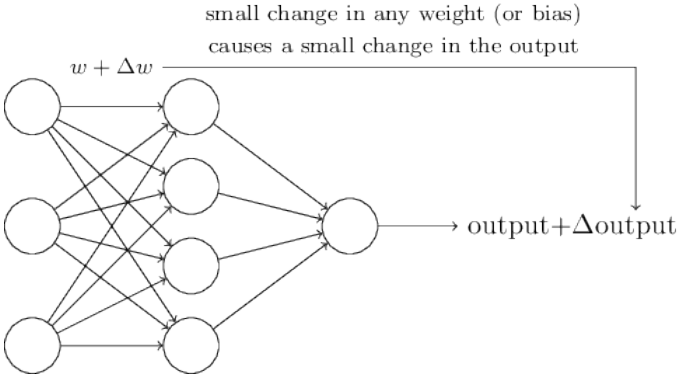
- Uses the step function as the activation function.
- ✔ A network of perceptrons can represent arbitrary boolean functions and arbitrary closely continuous functions — the difficulty is in the training.
- ✘ Was the original model for a neuron but rarely used now as zero sensitivity away from zero and chaotic[§] at zero.

[§] arbitrary small change when $z = 0$ results in a large change in output.

Working Towards Automated Training . . .

To train a network we could start with, say random values for weights and biases, then pick random weights/bias to change and see effect on the output. Two problems

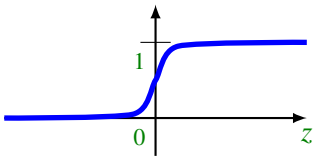
- Number of weights/bias can get very large, so need to be more focused than just random.
- Need structure where arbitrary small change in weights/bias results in a small change in the output — **not the case with perceptrons**.



Concept: Sigmoid Neuron

Uses the logistic/sigmoid function as the activation function ...

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

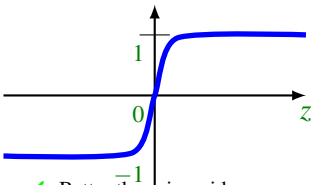


- ✔ For large positive or negative values of z, the result will be nearly the same as with the perceptron (i.e., output is near 0 or 1). For values close to the boundary of the separating hyperplane, values near 0.5 will be emitted.
- ✔ While sigmoid changes rapidly for z near zero, the change is smooth $\implies$ can use calculus.
- A single neuron with the sigmoid activation function is equivalent to logistic regression.

Concept: Activation Functions

hyperbolic tan

$\text{atanh}(z) = 2\sigma(z) - 1$

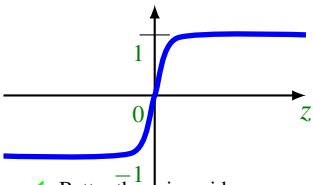


- ✔ Better then sigmoid, as optimisation is less likely to get trapped when inputs are negative.
- ✘ Vanishing derivative for large input

Concept: Activation Functions

hyperbolic tan

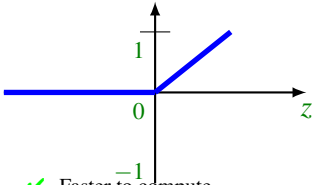
$$\text{atanh}(z) = 2\sigma(z) - 1$$



- ✔ Better then sigmoid, as optimisation is less likely to get trapped when inputs are negative.
- ✘ Vanishing derivative for large input

Rectified linear unit

$$\text{ReLU}(z) = \max(0, z)$$

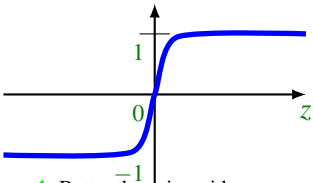


- ✔ Faster to compute.
- ✔ Results in faster ($\times 6$) learning.
- ✔ No vanishing derivative for positive input.
- ✘ Can be fragile during training and can 'die', if input caught in negative region.

Concept: Activation Functions

hyperbolic tan

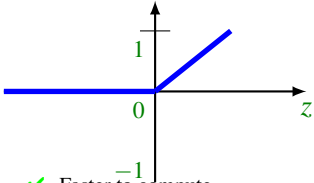
$$\text{atanh}(z) = 2\sigma(z) - 1$$



- ✔ Better then sigmoid, as optimisation is less likely to get trapped when inputs are negative.
- ✘ Vanishing derivative for large input

Rectified linear unit

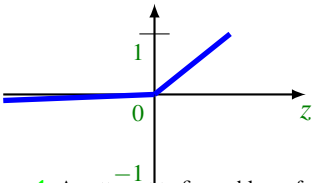
$$\text{ReLU}(z) = \max(0, z)$$



- ✔ Faster to compute.
- ✔ Results in faster ($\times 6$) learning.
- ✔ No vanishing derivative for positive input.
- ✘ Can be fragile during training and can 'die', if input caught in negative region.

Leaky ReLU

$$\text{LReLU}(z) = \begin{cases} z & z \geq 0 \\ 0.01z & z < 0 \end{cases}$$

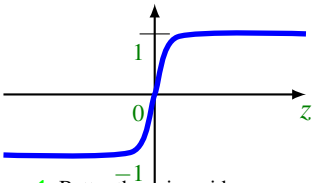


- ✔ An attempt to fix problem of 'dead' ReLU
- ✔ No vanishing derivative.
- Effect on improving training is inconsistent to date.
- Slope (0.01) for negative z can be parameterised.

Concept: Activation Functions

hyperbolic tan

$$\text{atanh}(z) = 2\sigma(z) - 1$$

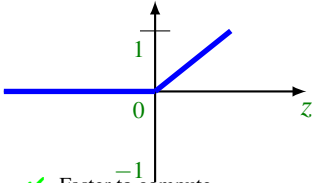


- ✔ Better then sigmoid, as optimisation is less likely to get trapped when inputs are negative.

Current best practice: Use ReLU and lower learning rates if see many (20-40%) 'dead' neurons.

Rectified linear unit

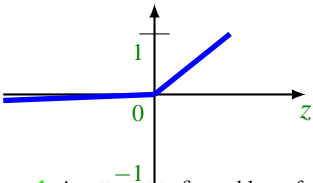
$$\text{ReLU}(z) = \max(0, z)$$



- ✔ Faster to compute.
- ✔ Results in faster ($\times 6$) learning.
- ✔ No vanishing derivative for positive input.
- ✘ Can be fragile during training and can 'die', if input caught in negative region.

Leaky ReLU

$$\text{LReLU}(z) = \begin{cases} z & z \geq 0 \\ 0.01z & z < 0 \end{cases}$$



- ✔ An attempt to fix problem of 'dead' ReLU
- ✔ No vanishing derivative.
- Effect on improving training is inconsistent to date.
- Slope (0.01) for negative z can be parameterised.

Example: Sigmoid (logistic) vs ReLU for XOR

xor_logistic_vs_relu .py

```
3  import numpy as np
4  from sklearn.neural_network import MLPClassifier
5
6  xs = np.array([[0, 0],[0, 1],[1, 0],[1, 1]])
7  ys = np.array([0, 1, 1, 0])
8
9  for activation in ['logistic', 'relu']:
10     print ("\nUsing activation %s" % activation)
11
12     model = MLPClassifier(
13         activation=activation,
14         max_iter=10000,
15         hidden_layer_sizes=(2,),
16         random_state=20)
17
18     model.fit(xs, ys)
19
20     print('score:', model.score(xs, ys))
21     print('predictions:', model.predict(xs))
22     print('expected:', ys)
```

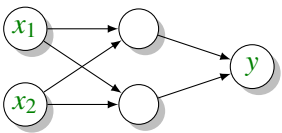
```
Using activation logistic
score: 0.5
predictions: [1 1 1 1]
expected: [0 1 1 0]

Using activation relu
score: 1.0
predictions: [0 1 1 0]
expected: [0 1 1 0]
```

Concept: One-Hot Encoding

Consider the following neural networks representations of the XOR function.

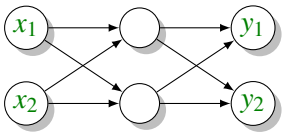
Minimal

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



One-Hot Encoding

x_1	x_2	y_1	y_2
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

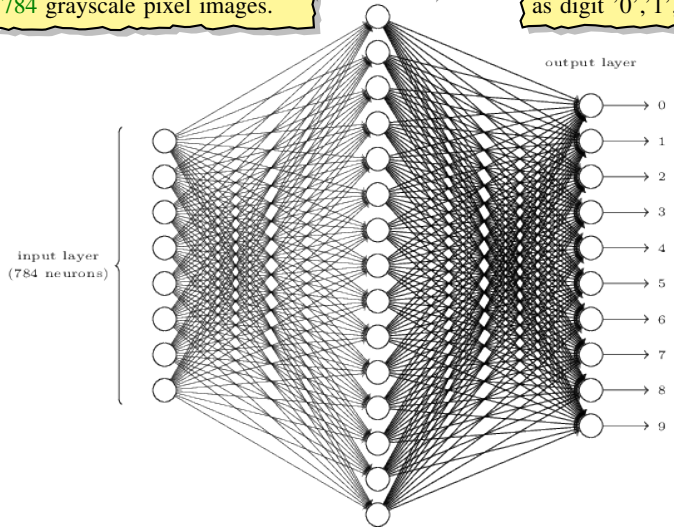


One-hot encoding with k output nodes is typically used in classification problems with k classes.

Example: One-Hot Encoding for MIST Digits Dataset

Input consists of $28 \times 28 = 784$ grayscale pixel images.

Desired output is classification as digit '0','1', ..., '9'.



Concept: Soft-Max

The soft-max is an activation function[¶] that is usually applied to the output layer of a network when one-hot encoding is used. It is defined by

$$\text{softmax}(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}} \qquad \text{for } j = 1, 2, \ldots, n$$

✔ Output from soft-max are in interval $[0, 1]$ and sum (over nodes in layer) to one. Hence can be treated as probabilities.

[¶] i.e., is applied to $z = \vec{w} \cdot \vec{x} + b$, the result of a summation operation.

Concept: Cost Functions

The cost function (also known as a error loss function) measures how well the network predicts outputs. The goal is to then find a set of weights and biases that minimises the cost.

Squared error loss, L_2 error

$$C(w,b) = \frac{1}{2n}\sum_{i}^{n}\left\{y_i - \hat{y}(x_i)\right\}^2$$

Absolute error loss, L_1 error

$$C(w,b) = \frac{1}{n}\sum_{i}^{n}\left|y_i - \hat{y}(x_i)\right|$$

Average Cross Entropy

$$C(w,b) = -\frac{1}{n}\sum_{i}^{n}\left[y_i \log\left(\hat{y}(x_i)\right) + (1-y_i)\log\left((1-\hat{y}(x_i))\right)\right]$$

The log function blows up at zero so magnifies larger errors —- the value of 0.000001 is much better than 0.000000001 when predicting 1.

Concept: Decomposable Cost Functions

- Notice in the previous slide, that to compute the cost function we had to compute its value for each of the n input-output pairs. What happens if n is large? . . . everything slooooows down $\implies$ not great.
- If we define C_i as the cost for the i-th input-sample pair, then the total cost is just

$$C = \frac{1}{n} \sum_i C_i$$

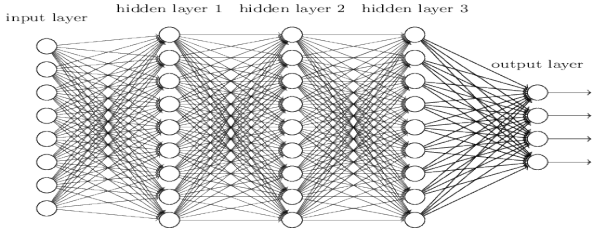
This has two big advantages

- Parallelisation — we can run over separate cores and compute C_i terms in parallel.
- Sampling — Consider taking a subset $M \subseteq \{1, 2, \ldots n\}$ with size m of the training set. It would seem that we can approximate the cost function using only this subsample of the data:

$$\frac{\sum_{i \in M} \nabla C_i}{m} \approx \frac{\sum_{i=1}^{n} \nabla C_i}{n} \approx \nabla C$$

So it seems that we can perhaps estimate the gradient using only a small subset of the entire training set.

Forward Propagation



Define:

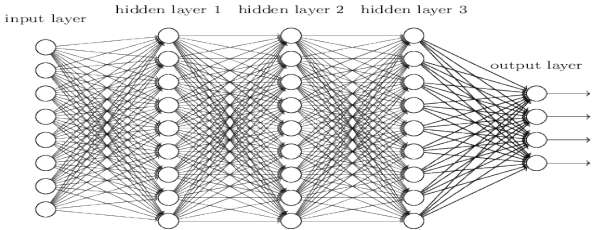
- L is number of layers (1 input + 1 output + $(L-2)$ hidden layers).
- a_j^l is output from applying the activation function on node j in layer l.

$\Rightarrow$ a_j^1 is value of input node j, and a_j^L is value of output node j.

- w_{jk}^l is the weight of node k in layer $l-1$ as applied by node j in layer l.
- b_j^l is the bias of node j of layer l.
- z_j^l is the result of the summation operation on node j in layer l.

$$z_j^l = \sum_k a_k^{(l-1)} \times w_{jk}^l + b_j^l = \boldsymbol{a}^{(l-1)} \cdot \boldsymbol{w}_j^l + b_j^l \qquad a_j^l = \sigma(z_j^l)$$

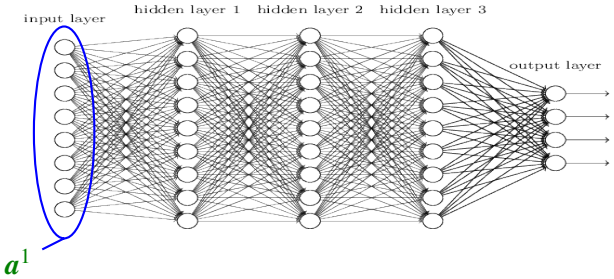
Forward Propagation II



where:

- h is the activation function: σ, atanh, ReLU, etc

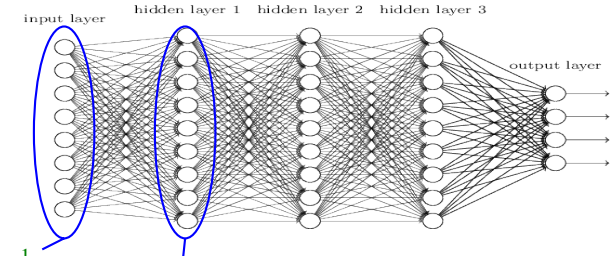
Forward Propagation II



where:

- h is the activation function: σ, atanh, ReLU, etc

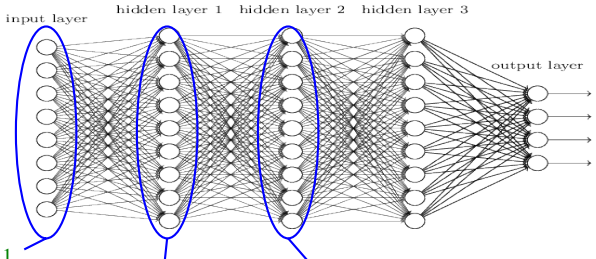
Forward Propagation II



where:

- h is the activation function: σ, atanh, ReLU, etc

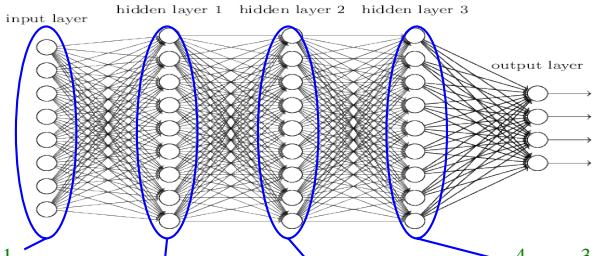
Forward Propagation II



where:

- h is the activation function: σ, atanh, ReLU, etc

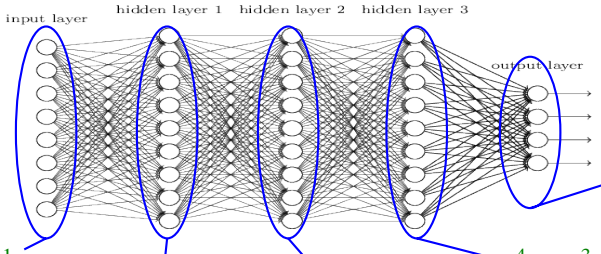
Forward Propagation II



where:

- h is the activation function: σ, atanh, ReLU, etc

Forward Propagation II



$$\boldsymbol{a}^1$$

$$z_j^2 = \boldsymbol{a}^1 \cdot \boldsymbol{w}_j^2 + b_j^2$$
$$a_j^2 = h(z_j^2)$$

$$z_j^3 = \boldsymbol{a}^2 \cdot \boldsymbol{w}_j^3 + b_j^3$$
$$a_j^3 = h(z_j^3)$$

$$z_j^4 = \boldsymbol{a}^3 \cdot \boldsymbol{w}_j^4 + b_j^4$$
$$a_j^4 = h(z_j^4)$$

$$z_j^5 = \boldsymbol{a}^4 \cdot \boldsymbol{w}_j^5 + b_j^5$$
$$a_j^5 = \text{softmax}(z_j^4)$$

where:

- h is the activation function: σ, atanh, ReLU, etc

Optimisation – Gradient Decent

- Given a cost function that depends on the weights and the biases, training the network is effectively an optimisation problem.
- Standard technique is

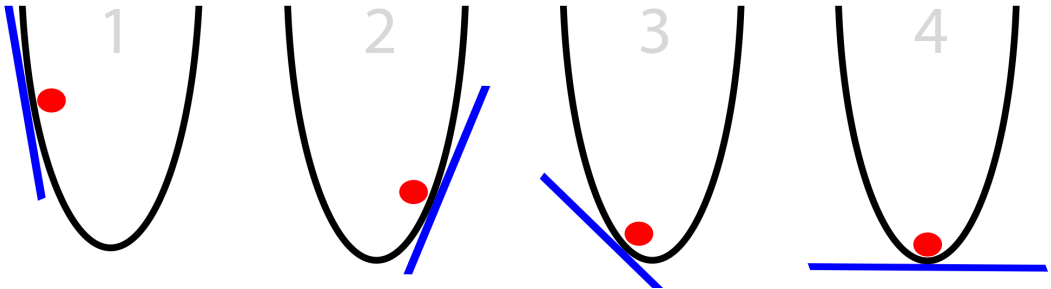
> **Gradient Descent**
> Compute the gradient function, then move a small amount in the opposite direction of the gradient (because we are minimising), and then recalculate the gradient on the new spot, and repeat.

- Mathematically, we can describe these updates, on the k-th iteration, as:

$$\vec{w}_{k+1} = \vec{w}_k - \eta \cdot \nabla_w C$$
$$\vec{b}_{k+1} = \vec{b}_k - \eta \cdot \nabla_b C$$

For some value $\eta > 0$. This tuning parameter is called the learning rate. Too low, and learning takes a very long time. Too small, and it is likely to have trouble finding the true minimum (as it will keep 'overshooting' it).

Gradient Descent in 1D



- Blue line represents the gradient.
  - If slope is negative, move right
  - If slope is positive, move left
  - (Repeat until slope == 0)

Optimisation – Stochastic gradient descent (SGD)

Stochastic gradient descent uses decomposable property of the cost function to speed up the process of doing gradient descent. Specifically, the input data are randomly partitioned into disjoint groups $M_1, M_2, \ldots, M_{n/m}$. We then do the following updates to the weights (biases are done at the same time, but omitted for sake of space):

$$w_{k+1} = w_k - \frac{\eta}{m} \sum_{i \in M_1} \nabla C_i$$

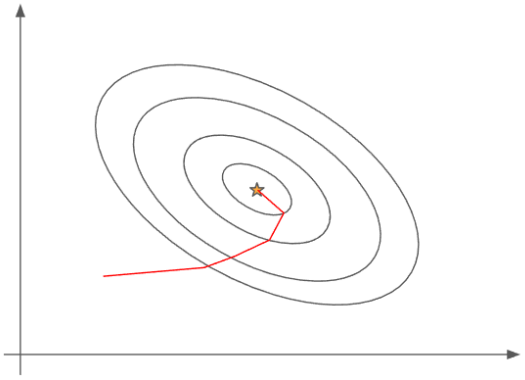
$$w_{k+2} = w_{k+1} - \frac{\eta}{m} \sum_{i \in M_2} \nabla C_i$$

$$\vdots$$

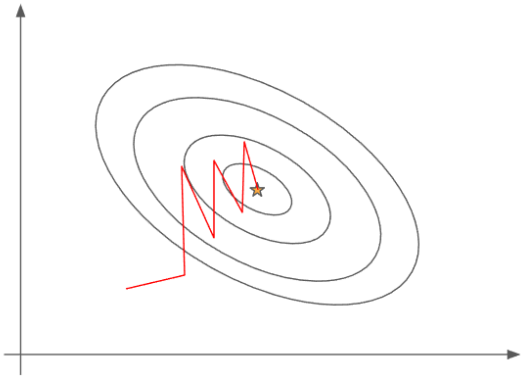
$$w_{k+n/m+1} = w_{k+n/m} - \frac{\eta}{m} \sum_{i \in M_{n/m}} \nabla C_i$$

Each set M_j is called a mini-batch and going through the entire dataset as above is called an epoch.

Gradient Descent vs Stochastic Gradient Descent



Gradient Descent



Stochastic Gradient Descent

- GD always head in the (locally) correct direction but each step takes a long time to compute.
- SGD will miss some local information so movement is more erratic. However computation of steps is much ($\times 100, \times 1000, \ldots$) faster.

Back Propagation

- Back propagation deals with computing the gradient of the cost function.
- We will not cover this today, as a proper treatment requires calculus — see resources at end of notes, if interested.
- To use neural networks you don't have to understand back propagation but it helps. See Medium article Yes you should understand backprop

Addressing Overfitting by Regularisation

As the size of neural networks grow, the number of weights and biases can quickly become quite large. State of the art neural networks today often have billions of weight values. In order to avoid over-fitting, one common approach is to add a penalty term to the cost function. Common choices are the ℓ_2-norm, given as:

$$C = C_0 + \lambda \sum_i w_i^2$$

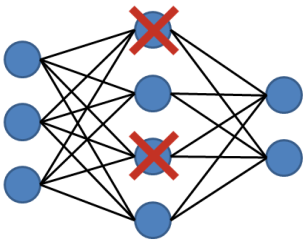
Where C_0 is the unregularized cost, and the ℓ_1-norm:

$$C = C_0 + \lambda \sum_i |w_i|.$$

The distinction between these is similar to the differences between lasso and ridge regression.

Addressing Overfitting by Dropout

A very different approach to avoiding over-fitting is to use an approach called *dropout*. Here, the output of a randomly chosen subset of the neurons are temporarily set to zero during the training of a given mini-batch. This makes it so that the neurons cannot overly adapt to the output from prior layers as these are not always present. It has enjoyed wide-spread adoption and massive empirical evidence as to its usefulness.



From about 2018, dropout is been replaced by Batch normalisation between convolution and activation layers in convolutional networks. This helps to protect against vanishing gradient during training — resulting in lower training time and better performance.

Momentum

Problem

- Learning is slow if the learning rate is set too low.
- Gradient may be steep in some directions but shallow in others.

$\Rightarrow$ add a momentum term α.

Solution

Momentum, adds a fraction of the past weight update to the current weight update. This helps prevent the model from getting stuck in local minima, as even if the current gradient is 0, the past one most likely was not, so it will as easily get stuck. By using momentum, the movements along the error surface are also smoother in general and the network can move more quickly throughout it.

Typical value for α is 0.5.

If the direction of the gradient remains constant, the algorithm will take increasingly large steps.

Outline

Resources

- Neural Networks and Deep Learning is a free online book by Michael Nielsen. This is a lovely read and my notes very much follow his approach.
- 3Blue1Brown has 4 amazing videos on neural networks (and many others on Calculous and Linear Algebra). See:
  - Chapter 1: But what *is* a Neural Network?
  - Chapter 2: Gradient descent, how neural networks learn.
  - Chapter 3: What is back propagation really doing?
  - Appendix: Back propagation calculus
- Learn TensorFlow and deep learning, without a PhD
  - Basics of neural networks with Tensorflow
  - Feed forward, convolution and recurrent networks,
  - MIST data set - training issues <— brilliant