Imperial College
London

COURSEWORK

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

# Introduction To Machine Learning

*Author:*
Your Name (CID: your college-id number)

Date: March 22, 2019

# 1 Introduction

## 1.1 Machine Learning Approaches

### 1.1.1 Supervised Learning

**Supervised learning** is where you have input variables $X_i$ and an output variable $Y_i$ and you use an algorithm to learn the mapping function from the input to the output $Y = f(X)$.

The **training set** is in the form $D = \{(X_i, Y_i)\}_{i=1}^{N}$ where $N$ is the number of samples.

Each $X_i$ element is in the **feature space**. In the simplest setting, each input $X_i$ is a D-dimensional vector called **features**, attributes or covariants.

Each $Y_i$ element is in the **label space**. The three main types of Label Space are:

1. Categorical: $Y_i$ is a category from a finite set
2. Real-valued scalar: $Y_i$ is a scalar value
3. Ordinal regression: $Y_i$ has some natural ordering

### 1.1.2 Unsupervised Learning

**Unsupervised learning** is where you only have input data $X$ and no corresponding output variables.

The goal for unsupervised learning is to model the underlying structure or distribution in the data in order to explain the data $X$ in a more efficient way. This is possible in broadly two ways:

1. Dimensionality reduction: Reducing the dimensions in the data $X$
2. Clustering: Assigning the data $X$ to automatically defined categorical labels

### 1.1.3 Reinforcement Learning

**Reinforcement Learning** is about finding the suitable action to maximise reward in a particular situation.

Reinforcement learning differs from the supervised learning in a way that in supervised learning the correct is given whereas in reinforcement learning the agent only gives a reward signal.

In the absence of training dataset, the machine is bound to learn from its experience.

## 1.2 Machine Learning Problems

1. **Classification** is about predicting the right label for an unknown sample
2. **Regression** is about approximating an unknown function

3. **Clustering** is about grouping data in such a way that data points in the same group (cluster) are more similar to each other than to those in other clusters.

4. **Dimensionality reduction** is about reducing the dimensionality of the observed data

5. **Density Estimation** is about estimating unobservable underlying probability density function based on observed data

6. **Policy Search** is about finding which action an agent should take, depending on its current state, to maximise the received rewards.

## 1.3 Lazy Learning and Eager Learning

### 1.3.1 Lazy Learning

In **Lazy Learning**, the generalisation beyond the training data is delayed until a query is made to the system.

Lazy learning is suitable for complex and incomplete problem domains.

The disadvantage of lazy learning includes large space requirement and long query time.

### 1.3.2 Eager Learning

In **Eager Learning**, the system tries to construct a general, input-independent target function during training of the system

Eager learning systems have better memory efficiency, and usually low query time.

The disadvantage of eager learning is that it is unable to provide good local approximations in the target function.

# 2 Instance Based Learning

**Instance-based learning** is a family of learning algorithms that, instead of performing explicit generalisation, compares new problem instances with instances seen in training, which have been stored in memory.

### 2.0.1 K-Nearest Neighbours (KNN)

KNN is a lazy learning algorithm that can be used for both classification and regression predictive problems:

In classification problems, we consider the class in majority.

In regression problems, we consider the value in majority.

The classifier considers the K nearest neighbours (in the feature space) of the current instance and assign the class in the majority.

The KNN algorithm is usually a quite powerful approach however it might be slow if the dataset is large

The distance between the points can be defined as:

1. Manhanttan Distance

$$d(x,y) = \sum_{i=1}^{m} |x_i - y_i|$$

2. Euclidian Distance (Most Commonly Used)

$$d(x,y) = \sqrt{\sum_{i=1}^{m} (x_i - y_i)^2}$$

3. Chebyshev Distance

$$d(x,y) = max_{i=1}^{m} |x_i - y_i|$$

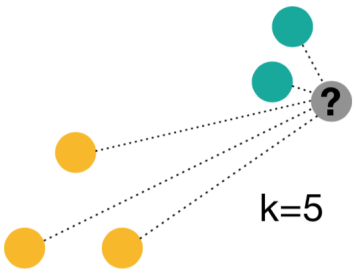
The influence of K:

1. A small K gives a good resolution of the borderlines between classes but is sensitive to noise.

2. A large K gives a bad resolution of the borderlines between classes but is robust to noise.

**Distance Weighted KNN**

A refinement of the KNN algorithm is to assign a weight $w_i$ to each neighbour $x_i$ of the query instance $x_q$, such that the greater distance, the smaller the weight.




Distance Weighted KNN algorithm is robust to noisy training data since the classification is based on a weighted combination of all K nearest neighbours.

If K = number of samples then the algorithm is a **global method**

If K ¡ number of samples then the algorithm is a **local method**

Some weighting strategies include:

1. Inverse of the distance

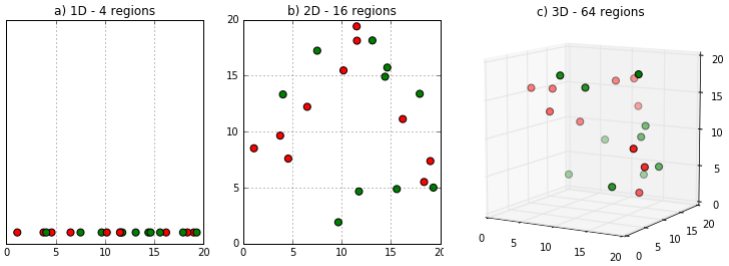
$$w_i = \frac{1}{d(x_i,\ x_q)}$$

2. Gaussian distribution

$$w_i = \frac{1}{\sqrt{2\pi}} e^{\frac{-d(x_i,\ x_q)^2}{2}}$$

**Curse of dimensionality**

The **curse of dimensionality** refers to various phenomena that arise when analysing and organising data in high-dimensional spaces that do not occur in low-dimensional settings.




As dimensions or number of features grows, dimensions space increases exponentially. This exponential growth in data causes high sparsity in the data set and unnecessarily increases storage space and processing time for the KNN algorithm.

# 3 Decision Trees Algorithms

Decision Trees is an eager learning algorithm that can be used for both classification and regression problems.

**Decision Tree Algorithms** are methods for approximating discrete classification functions by means of a tree-based representation. These algorithms employ top-down greedy search through the space of possible solutions.

**Iterative Dichotomiser (ID3) algorithm** is one of the most commonly used Decision Tree learning algorithms.

## 3.1 Iterative Dichotomiser (ID3) algorithm

The ID3 Algorithm consists of the following steps:

1. Search for a **split point**(or attribute) using a **statistical test** of each attribute to determine how well it classifies the training examples when considered alone.

2. Split your dataset according to your split point (or attribute)
3. Repeat steps 1, 2 on each of the created subsets.

Several statistical tests can be used when searching for a split point but the most common statistical test is **Information Gain**. Other statistical tests include GINI impurity and Variance reduction (Mainly used in regression).

### 3.1.1 Information Gain

**Information Entropy** tells how much **information** there is in an event. The higher the entropy of an event, the more information it will contain.

Consider the example with two sentences:

1. Sentence 1 contains all 'A'. Highly ordered, predictable and low in entropy.
2. Sentence 2 contains random letters. Highly disordered and high in entropy.

It is obvious that Sentence 2 contains more information than Sentence 1.

Here is another way to see information entropy. The higher the entropy of a variable, the less **knowledge** we know about the variable.

Consider the example with 3 boxes:

1. Box 1 contains 4 red balls (Least entropy)
2. Box 2 contains 3 red balls and 1 blue ball
3. Box 3 contains 2 red balls and 2 blue balls (Highest entropy)

If we have to pick 1 ball from each box. We have the most knowledge about Box 1, the ball we pick must be red. We have the least knowledge or certainty about Box 3 because we cannot predict the outcome.

Information Entropy is calculated by:

$$H(X) = -\sum_{i=1}^{M} p_i \log_2(p_i)$$

The **Information Gain** from knowing attribute $a$ is calculated by

$$Gain(a) = H(dataset) - \left( \frac{|subset\ A|}{|dataset|} H(subset\ A) + \frac{|subset\ B|}{|dataset|} H(subset\ B) \right)$$

Where $subset\ A$ and $subset\ B$ are subsets by splitting the $dataset$ using attribute $a$.

### 3.1.2 Types of Inputs

There are mainly two types of inputs:

1. Symbolic Values
2. Ordered Values (Real Numbers)

For **symbolic values**, search for the most informative feature and then create as many branches as there are different values for this feature.

For **ordered values**, sort the values for each attribute and then evaluate the information gain for each mid-points between adjacent values. We can save computation time by only considering split points that lie between examples of different classes.

### 3.1.3 Example

The following table includes decision factors to play tennis.

| Day | Outlook | Temp. | Humidity | Wind | Decision |
|---|---|---|---|---|---|
| 1 | Sunny | Hot | High | Weak | No |
| 2 | Sunny | Hot | High | Strong | No |
| 3 | Overcast | Hot | High | Weak | Yes |
| 4 | Rain | Mild | High | Weak | Yes |
| 5 | Rain | Cool | Normal | Weak | Yes |
| 6 | Rain | Cool | Normal | Strong | No |
| 7 | Overcast | Cool | Normal | Strong | Yes |
| 8 | Sunny | Mild | High | Weak | No |
| 9 | Sunny | Cool | Normal | Weak | Yes |
| 10 | Rain | Mild | Normal | Weak | Yes |
| 11 | Sunny | Mild | Normal | Strong | Yes |
| 12 | Overcast | Mild | High | Strong | Yes |
| 13 | Overcast | Hot | Normal | Weak | Yes |
| 14 | Rain | Mild | High | Strong | No |

**Entropy of Dataset**

There are a total of 14 decision instances: 9 yes and 5 no. The entropy is calculated as:

$$H(Decision) = -\left[\frac{9}{14}\log_2\left(\frac{9}{14}\right) + \frac{5}{14}\log_2\left(\frac{5}{14}\right)\right] = 0.940$$

**Information Gain Using Outlook Attribute**

There are 5 Sunny instances: 2 yes and 3 no. The entropy is calculated as:

$$H(Sunny) = -\left[\frac{2}{5}\log_2\left(\frac{2}{5}\right) + \frac{3}{5}\log_2\left(\frac{3}{5}\right)\right] = 0.971$$

There are 4 Overcast instances: 4 yes and 0 no. The entropy is calculated as:

$$H(Overcast) = -\left[\frac{4}{4}\log_2\left(\frac{4}{4}\right) + \frac{0}{4}\log_2\left(\frac{0}{4}\right)\right] = 0$$

There are 5 Rain instances: 3 yes and 2 no. The entropy is calculated as:

$$H(Rain) = -\left[\frac{3}{5}\log_2\left(\frac{3}{5}\right) + \frac{2}{5}\log_2\left(\frac{2}{5}\right)\right] = 0.971$$

The Information Gain is therefore

$$Gain(Outlook) = H(Decision) - \left(\frac{5}{14}H(Sunny) + \frac{4}{14}H(Overcast) + \frac{5}{14}H(Rain)\right)$$
$$= 0.940 - \left(\frac{5}{14}(0.971) + \frac{4}{14}(0) + \frac{5}{14}(0.971)\right) = 0.246$$

Performing same calculations on all other attributes to find the information gain:

$$Gain(Outlook) = 0.246$$
$$Gain(Temperature) = 0.029$$
$$Gain(Humidity) = 0.151$$
$$Gain(Wind) = 0.048$$

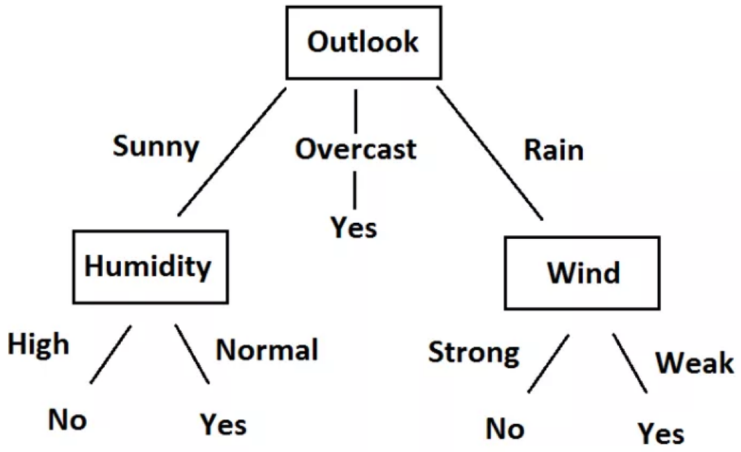
The Outlook factor on decision produces the highest information gain and so we first split up the data using the Outlook attribute to produce three subsets.

| Day | Outlook | Temp. | Humidity | Wind | Decision |
|---|---|---|---|---|---|
| 3 | Overcast | Hot | High | Weak | Yes |
| 7 | Overcast | Cool | Normal | Strong | Yes |
| 12 | Overcast | Mild | High | Strong | Yes |
| 13 | Overcast | Hot | Normal | Weak | Yes |

| Day | Outlook | Temp. | Humidity | Wind | Decision |
|---|---|---|---|---|---|
| 1 | Sunny | Hot | High | Weak | No |
| 2 | Sunny | Hot | High | Strong | No |
| 8 | Sunny | Mild | High | Weak | No |
| 9 | Sunny | Cool | Normal | Weak | Yes |
| 11 | Sunny | Mild | Normal | Strong | Yes |

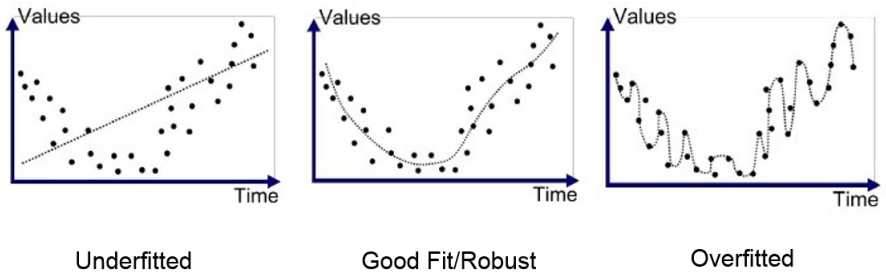
| Day | Outlook | Temp. | Humidity | Wind | Decision |
|---|---|---|---|---|---|
| 4 | Rain | Mild | High | Weak | Yes |
| 5 | Rain | Cool | Normal | Weak | Yes |
| 6 | Rain | Cool | Normal | Strong | No |
| 10 | Rain | Mild | Normal | Weak | Yes |
| 14 | Rain | Mild | High | Strong | No |

We then repeat the same process for each of the subsets until we build a decision tree.




## 3.2 Overfitting

Like many machine learning algorithms, decision trees can overfit. **Overfitting** refers to a model that models the training data too well.




For decision trees, the solution to overcome overfitting is **Pruning**. Pruning is the process of going through all the nodes that are only connected to leaves and check if the accuracy on the validation dataset would increase if this node is turned into a leaf.

We have to do this recursively as when you turn nodes into leaves, you might create new nodes that are connected to two leaves.

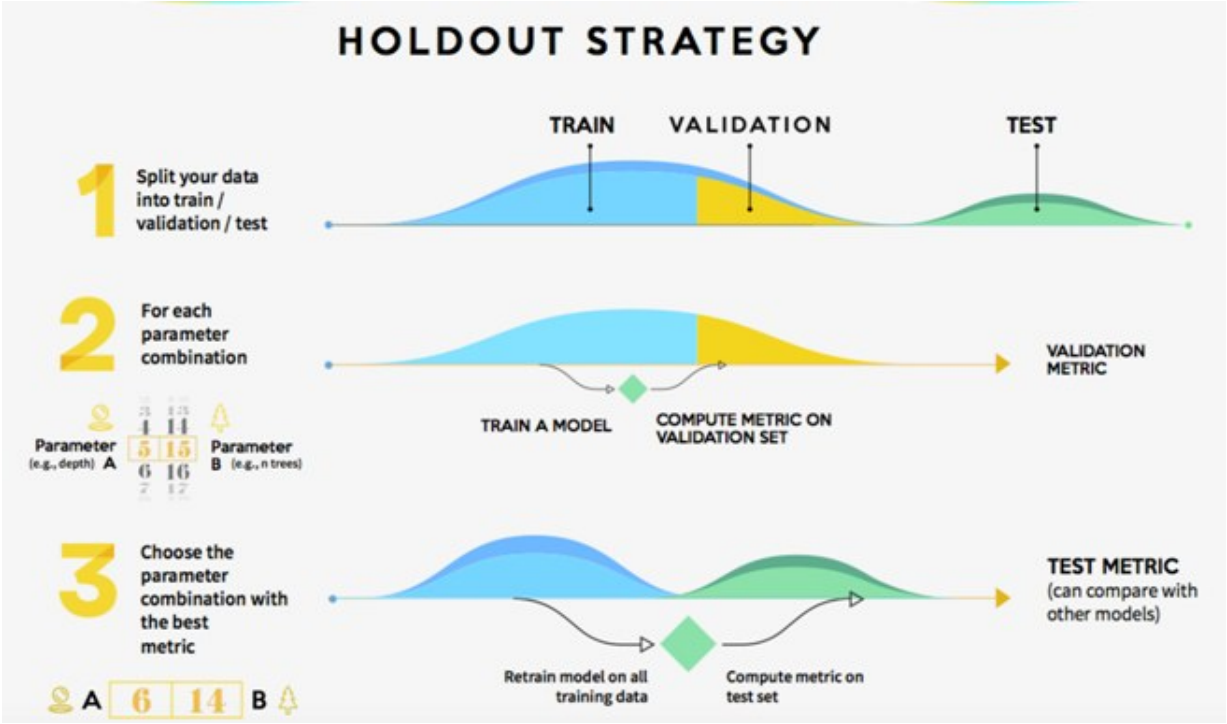
# 4 Evaluation Hypothesis

## 4.1 Model Evaluation

The ultimate goal in machine learning is to create models or algorithms that can generalise unknown data.

When evaluating machine learning models, the validation step helps you find the best parameters for your model while also preventing it from becoming overfitted. Two of the most popular strategies to perform the validation step are the **holdout strategy** and the **k-fold strategy**.

### 4.1.1 Holdout Method




1. Split the dataset into **Training set**, **Validation set** and **Test set**.

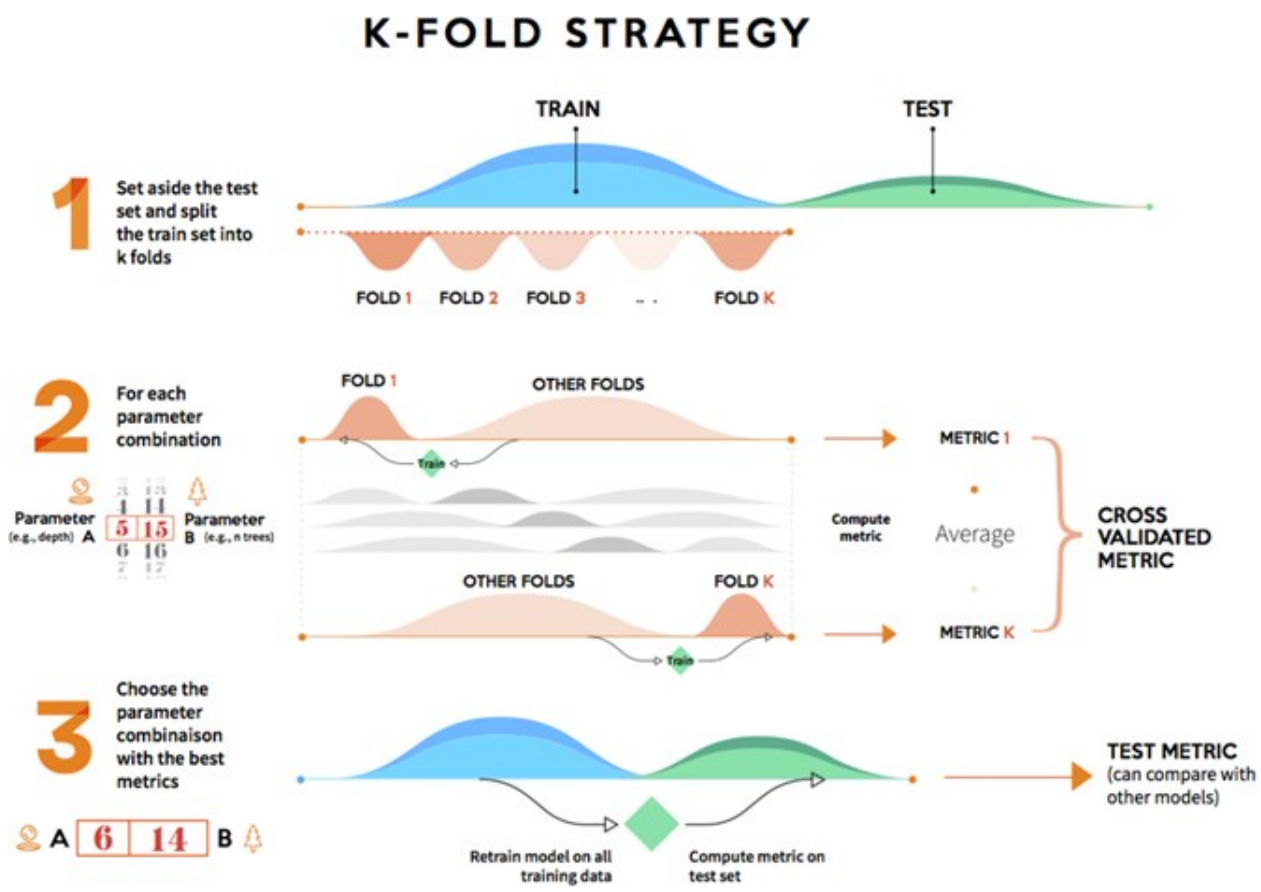
2. Try different parameter values on the training dataset, and select the best according to the accuracy on the validation dataset.

3. Now that the parameters are fixed, combine the training and validation datasets to train a new model and perform the final evaluation on the test dataset.

The advantage of Holdout Method is that it only needs to be run once so has lower computational costs.

The disadvantage of Holdout Method is that the performance evaluation has higher variance if data size is small.

### 4.1.2 K-fold Cross Validation




When we have a small sample size then a good alternative is **cross validation**.

1. Split the data into **Training + Validation set** and **Test set**.

2. Split the Training + Validation set into K folds, then for each parameter set run the K fold cross-validation. This is done by training the model K times, each time leaving a different fold out of the training data and using it instead as a validation set. At the end, the error is averaged across all K tests.

3. Now that the parameters are fixed, use the Training + Validation sets to train a new model and perform the final evaluation on the test dataset.

The advantage of cross validation is that the performance evaluation has lower variance.

The disadvantage of cross validation is the high computation cost.

## 4.2 Performance Metrics

A confusion matrix is a table that is often used to describe the performance of a classification model on a set of test data for which the true values are known. In the simplest case we have two classes: Class1 (Positive) and Class2 (Negative).

| | Class 1 Predicted | Classe 2 Predicted |
|---|---|---|
| Class 1 Actual | **TP**: True Positive | **FN**: False Negative |
| Class 2 Actual | **FP**: False Positive | **TN**: True Negative |

True positives (TP): Classified as Positive. It is Positive.

True negatives (TN): Classified as Negative. It is Negative.

False positive (FP): Classified as Positive. It is Negative.

False negatives (FN): Classified as Negative. It is Positive.

When there are multiple classes, we can define one class as positive and the others as negative. Then compute the metrics for each class.

| | Class 1 Predict | Classe 2 Predicted | Classe 3 Predicted | Classe 4 Predicted |
|---|---|---|---|---|
| Class 1 actual | **TP** | **FN** | **FN** | **FN** |
| Class 2 Actual | **FP** | **TN** | ? | ? |
| Class 3 actual | **FP** | ? | **TN** | ? |
| Class 4 actual | **FP** | ? | ? | **TN** |

There is a list of rates that are often computed from a confusion matrix. The most common ones are:

1. Classification Rate / Accuracy
2. Recall
3. Precision
4. F Score

### 4.2.1 Classification Rate / Accuracy

Accuracy is the number of correctly classified samples over the total number of samples

$$CR = \frac{TP + TN}{TP + TN + FP + FN}$$

### 4.2.2 Recall

Recall is the number of correctly classified positive examples divided by the total number of positive examples

$$Recall = \frac{TP}{TP + FN}$$

A high recall means that most of the positive examples are correctly recognised.

Recall can be calculated for each class if there are multiple classes.

The **Unweighted Average Recall (UAR)** is the mean recall for all classes. We compute recall for each class: class1 (R1), class2 (R2), class3 (R3) ... classN (RN)

$$UAR = \frac{R1 + R2 + R3 + ... + RN}{N}$$

### 4.2.3 Precision

Precision is the number of correctly classified positive examples divided by the total number of predicted positive examples

$$Recall = \frac{TP}{TP + FP}$$

A high recall means that if the model predicts a positive it is very likely to be positive.

Precision can be calculated for each class if there are multiple classes.

### 4.2.4 F Score

When there is high recall and low precision, the model can correctly recognise positive examples but there are many false positive.

When there is low recall and high precision, the model misses lots of positive examples but the ones classified as positive are very likely to be positive.

The F score is a weighted average of recall and precision.

$$F_\alpha = (1 + \alpha^2)\frac{Precision \times Recall}{\alpha^2 \times Precision + Recall}$$
$$F_1 = 2\frac{Precision \times Recall}{Precision + Recall}$$

F Score can be calculated for each class if there are multiple classes.

## 4.3 Problems

### 4.3.1 Imbalanced Datasets

Imbalanced data refers to a problem with classification problems where the classes are not represented equally.

An imbalanced dataset results in misleading metrics.

The classification rate (CR) follows the performance of the majority class

The Unweighted Average Recall (UAR) can help to detect that one class is completely misclassified but it does not give us any information about FP

**Solution**

One solution to imbalanced dataset problem is to divide by the total number of examples per class.

| | Class 1 Predicted | Classe 2 Predicted |
|---|---|---|
| Class 1 Actual | 700 | 300 |
| Class 2 Actual | 10 | 90 |

**Divide by the total number of examples per class** →

| | Class 1 Predicted | Classe 2 Predicted |
|---|---|---|
| Class 1 Actual | 0.7 | 0.3 |
| Class 2 Actual | 0.1 | 0.9 |

Other solutions include:

1. Upsampling the minority class
2. Downsampling the majority class
3. Repeat this procedure several times and train a classifier each time with a different training set

### 4.3.2 Overfitting

Overfitting refers to a model that models the training data too well. An overfitting model will have good performance on the training data but poor generalisation to other data.

Overfitting can occur when the learning is performed too long, or when the examples in the training set are not representative of all possible situations.

**Solution**

Solutions to overfitting include stopping the training earlier or getting more data.

## 4.4 Confidence Interval

The **true error** of a model $h$ is the probability that it will misclassify a randomly drawn example $x$ from distribution $D$

$$error_D(h) = \Pr_{x \in D}[f(x) \neq h(x)]$$

The **sample error** of a model $h$ and data sample $S$ is the proportion of examples $h$ misclassifies

$$error_S(h) = \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$

$$\delta(f(x), h(x)) = \begin{cases} 1, & \text{if } f(x) \neq h(x) \\ 0, & \text{if } f(x) = h(x) \end{cases}$$

We can measure the sample error $error_S(h)$, but we do not know true error $error_D(h)$.

We want to know how well $error_S(h)$ can estimate $error_D(h)$ using confidence interval

If $S$ contains $n$ examples, drawn independently of $h$ and each other. Then with approximately $N\%$ probability, $error_D(h)$ lies in interval

$$error_S \pm z_N \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

## 4.5 Comparing Algorithms

When comparing two models $h_1$ and $h_2$, we need to run a statistical test to tell us if there is indeed a difference between the two models.

There are several statistical tests that we can perform but in general, a statistical test will return a p-value represents the probability that the null hypothesis can be rejected.

The null hypothesis in our case means that there is no performance difference between the two algorithms tested.

If the statistical tests returns a p-value lower than 0.05, then we can conclude that the performance difference between two algorithms is statistically significant.

Notice that a p-value higher than 0.05 does not mean the two algorithms are similar, it simply means that we cannot observe a statistical difference

# 5 Artificial Neural Network

## 5.1 The Task

Given some inputs $x_1$, $x_2$, $x_3$... with desired output $y_1$, $y_2$, $y_3$... The task is to learn a function $y = f(x, \theta)$ that maps the input to the output by adjusting the parameters $\theta$.

### 5.1.1 Inputs and Outputs

Everything needs to be encoded as numbers to work with neural networks. For inputs such as text and speech, they get encoded into some number scheme before they are fed into a neural network.

Having numerical features doesn't necessarily mean that one can just feed them into a neural network. There might be outliers, or perhaps the range of values might be skewed.

Data can be separated into continuous and discrete data. You can roughly think of continuous data as measured and discrete as counted.

#### Continuous

These inputs and outputs corresponding to numerical values that are innately a numeric value and are continuous.

#### Discrete

This type of data is more common in classification. We can use **one-hot encoding** to feed discrete data into the neural network.

Given $N$ discrete values we construct a vector $v \in R^N$ such that $i^{th}$ index is equal to one.

## 5.2 Neural Networks

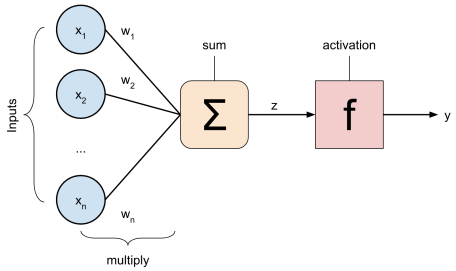
The building block of neural networks is **neuron**. The **architecture**, the way these neurons are connected, give rise to different sub-classes of neural networks.

Here we will focus on **feed-forward neural networks** also known as **multi-layer perceptrons (MLP)**.

### 5.2.1 Neuron

A neuron computes the following function

$$y = f(z) = f\left(\sum_i w_i x_i\right) = f(w^T x)$$



Every **input** is multiplied by a corresponding **weight** and then summed to the **pre-activation value** $z$, which is then passed to the **activation function** $f$.

We often write these in vector form, $w \in R^{n\times 1}$ and $x \in R^{n\times 1}$, when taking the dot product of two vectors $w \cdot x = w^T x$.

Often we will see $wx + b$ as the equation, with $b$ as the **bias**. In this case, we absorb the bias into $w$ by having an extra input whose value is always 1.

The weights allow the neuron to create a custom, learnable combination of inputs while the bias acts like a learnable threshold.

### 5.2.2 Perceptron

Perceptron is an algorithm for supervised binary classification. There are 2 classes we want to predict, class 0 and class 1. We use a **threshold function** as the activation function.

$$f(w \cdot x) = \begin{cases} 1, \text{ if } w \cdot x \geq 0 \\ 0, \text{ otherwise} \end{cases}$$

The **perceptron learning rule** update the weights by

$$w_{i+1} = w_i + \alpha(y - f(w_i \cdot x))x_i$$

where $\alpha$ is the **learning rate**. The learning rule works as follow:

1. If $y = f(w_i \cdot x)$: then $\alpha(y - f(w_i \cdot x))x_i = 0$ so the weight stays the same

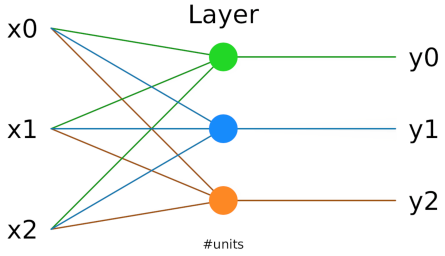
2. If $y = 1$, $f(w_i \cdot x) = 0$: The weight increase if $x_i$ is positive, and decrease if $x_i$ is negative. This change in weight can make $w \cdot x$ larger so that $f(w_i \cdot x) = 1$.

3. If $y = 0$, $f(w_i \cdot x) = 1$. The weight increase if $x_i$ is positive, and decrease if $x_i$ is negative. THis change in weight can make $w \cdot x$ smaller so that $f(w_i \cdot x) = 0$.

We can learn any **linearly separable function** using this algorithm. Furthermore, this algorithm gives hint at how we can train neural networks by adjusting the weights in the direction of the desired output.

### 5.2.3 Layer

A layer is a collection of neurons that share the same inputs but have different weights.



In vector form, $x \in R^{N\times 1}$. the weights can be collected into a matrix $W \in R^{N\times M}$, with bias $b \in R^{M\times 1}$ to produce output $y \in R^{M\times 1}$.

$$y = f(W^T x + b)$$

A neuron creates a learnable linear combination of its inputs passed through a non-linear activation function $f$. With layer, it now computes $M$ many different transformations of the input.

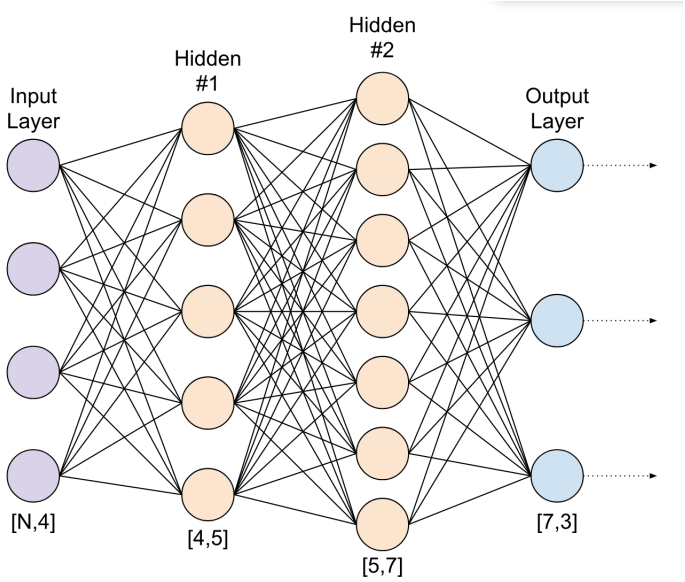
There is no requirement to apply the activation within the layer. We can separate the activation from the linear transformation.

$$z = W^T x + b$$
$$y = f(z)$$

## 5.3 Feed-Forward Networks

Feed Forward Networks are a collection of layers chained together. The construction of a feed-forward network is relatively straightforward, we just take some layers and connect the outputs of one to the inputs of the next one, here we noted a layer as $h_i$.

The **depth** of the network is often referred to the number of layers. The **width** is the number of neurons. The layers in between the input and output are called **hidden layers**.




The mathematical steps of the network

$$A^{(l)} = f^{(l)}(Z^{(l)}) \qquad Z^{(l)} = W^{(l)} \cdot A^{(l-1)}$$

We also often do not explicitly place a layer for the inputs, hence $A^{(0)} = x$

### 5.3.1 Initialising Weights

Most deep learning libraries have weight initialising in built, the most common ones are

**Zeros**: just sets the corresponding parameters to 0

**Normal**: sets the parameters from a normal distribution $W \sim N(\mu, \sigma^2)$ often mean to 0 and variance to 1

What we want is values to be random but vary in a reasonable range across the layers. If we get unlucky and assign small values, the outputs will start to vanish whereas assigning large weights will make things explode causing numerical overflows. Xavier Glorot and Yoshua Bengio proposed a normalised initialisation method

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

Where $U$ is uniform distribution. $n_j$ is the number of inputs and $n_{j+1}$ is the number of outputs

### 5.3.2 Activation Functions

The most common activation functions include:

**Linear (Identity)**: This is the same as having no activation function.

$$f(x) = x \qquad f'(x) = 1$$

**Sigmoid**: Compresses the output to the range between 0 and 1. Instead of suddenly jumping from 0 to 1 at a given threshold, it varies smoothly.

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \qquad sigmoid'(x) = sigmoid(x)(1 - sigmoid(x))$$

**tanh**: It is a scaled version of sigmoid. It ranges between -1 and 1.

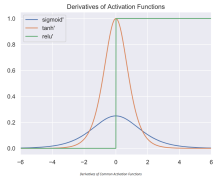
$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \qquad tanh'(x) = 1 - tanh^2(x)$$

**ReLU**: This activation is the **most commonly used one for feed-forward networks** since it preserves the desirable properties of a linear function while introducing non-linearity.

$$relu(x) = \begin{cases} x, \text{ if } x > 0 \\ 0, \text{ otherwise} \end{cases} \qquad relu'(x) = \begin{cases} 1, \text{ if } x > 0 \\ 0, \text{ otherwise} \end{cases}$$

**Softmax**: can be thought as the n-dimensional version of sigmoid, it compresses the sum of the output vector to be 1

$$softmax(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$




If you look at the gradients of sigmoid and tanh they strictly range between 0 and 1. When there is a very deep network, it causes the **vanishing gradient problem**. This is why sigmoid and tanh are only works well for shallow network, and ReLU is the most preferred function.

### 5.3.3 Loss Function

The loss function is the function we are trying to minimise such that when we do so we learn the relationship between the given inputs and the desired outputs.

Selecting the correct loss function is crucial and it depends on what we are trying to do.

#### Regression

When the task is to predict a continuous variable we have a regression problem. For such problem we use **squared error** loss function

$$L(y^{(i)}, a^{(i)}) = (a^{(i)} - y^{(i)})^2$$

Where $a^{(i)}$ is the network output and $y^{(i)}$ is the desired output.

When there are multiple outputs, we use the **mean-squared error** loss function

$$\frac{1}{N}\sum_{k}^{N}(a_k^{(i)} - y_k^{(i)})^2$$

#### Classification

In classification, we assume every input belongs to one class and one class only. **Binary classification** refers to the situation when we have two classes. **Multi-class classification** is when we have more than two classes. When we want to predict multiple classes or labels we have **multi-label classification**.

1. For Binary classification, we use the **Sigmoid** activation function and **Binary cross entropy** as the loss function, also known as **negative log likelihood**

$$L(y^{(i)}, a^{(i)}) = -\left[y^{(i)} log(a^{(i)}) + (1 - y^{(i)}) log(1 - a^{(i)})\right]$$

2. For Multi-class classification, we use the **Softmax** activation function in the final layer and **categorical cross entropy** as the loss function

$$L(y^{(i)}, a^{(i)}) = -\sum_{k} y_k^{(i)} log(a_k^{(i)})$$

Where $k$ is the number of classes we have.

3. For Multi-label classification, we use the **Sigmoid** activation function and **Binary cross entropy** as the loss function.

## 5.4 Training

Our network has **trainable parameters**, or **weights**. Training means adjusting those parameters to fit the desired data.

### 5.4.1 Back Propagation

To decide whether to increase or decrease the weight, we need to understand how the weight affects our loss function. For this, we use the **derivative of the loss function with respect to the weight**. Applying chain rule

$$\frac{\partial Loss}{\partial W^{(L)}} = \frac{\partial Loss}{\partial A^{(L)}} \cdot \frac{\partial A^{(L)}}{\partial Z^{(L)}} \cdot \frac{\partial Z^{(L)}}{\partial W^{(L)}}$$

Similarly, we can find how the previous hidden layer affect the loss function

$$\frac{\partial Loss}{\partial W^{(L-1)}} = \frac{\partial Loss}{\partial A^{(L)}} \cdot \frac{\partial A^{(L)}}{\partial Z^{(L)}} \cdot \frac{\partial Z^{(L)}}{\partial A^{(L-1)}} \cdot \frac{\partial A^{(L-1)}}{\partial Z^{(L-1)}} \cdot \frac{\partial Z^{(L-1)}}{\partial W^{(L-1)}}$$

This is the back-propagation algorithm, we propagate the gradients backwards through the network layers or in general back through the computation graph.

**Example**

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix}$$

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix}$$

$$B = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 \end{bmatrix}$$

$$Z = \begin{bmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} + b_1 & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} + b_2 & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} + b_3 \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} + b_1 & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} + b_2 & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} + b_3 \end{bmatrix}$$

If we do not consider the activation function, then

$$\frac{\partial Loss}{\partial W} = \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial W} \qquad \frac{\partial Loss}{\partial b} = \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial b} \qquad \frac{\partial Loss}{\partial X} = \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial X}$$

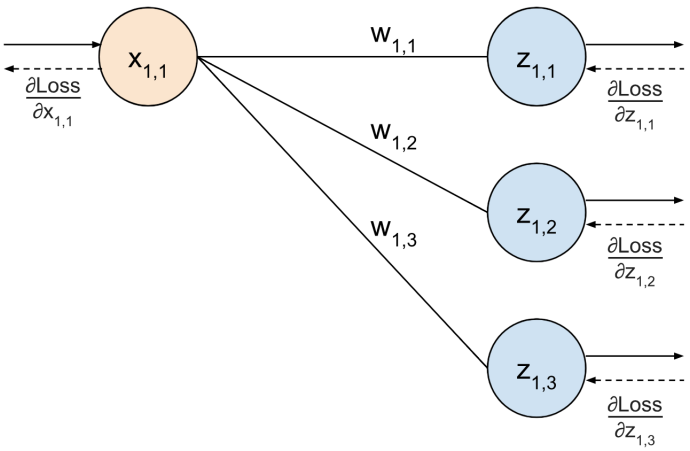
Hence all we need is

$$\frac{\partial Loss}{\partial Z} = \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} & \frac{\partial Loss}{\partial z_{1,2}} & \frac{\partial Loss}{\partial z_{1,3}} \\ \frac{\partial Loss}{\partial z_{2,1}} & \frac{\partial Loss}{\partial z_{2,2}} & \frac{\partial Loss}{\partial z_{2,3}} \end{bmatrix}$$

**Derivative of Loss with respect to X**

$$\frac{\partial Loss}{\partial X} = \begin{bmatrix} \frac{\partial Loss}{\partial x_{1,1}} & \frac{\partial Loss}{\partial x_{1,2}} \\ \frac{\partial Loss}{\partial x_{2,1}} & \frac{\partial Loss}{\partial x_{2,2}} \end{bmatrix}$$

Let's investigate one element




$$\begin{aligned} \frac{\partial Loss}{\partial x_{1,1}} &= \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial x_{1,1}} = \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} & \frac{\partial Loss}{\partial z_{1,2}} & \frac{\partial Loss}{\partial z_{1,3}} \\ \frac{\partial Loss}{\partial z_{2,1}} & \frac{\partial Loss}{\partial z_{2,2}} & \frac{\partial Loss}{\partial z_{2,3}} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ 0 & 0 & 0 \end{bmatrix} \\ &= \frac{\partial Loss}{\partial z_{1,1}} w_{1,1} + \frac{\partial Loss}{\partial z_{1,2}} w_{1,2} + \frac{\partial Loss}{\partial z_{1,3}} w_{1,3} \end{aligned}$$

Intuitively, how $x_{1,1}$ affects the loss is determined by the weights it multiplies with and then whatever is using that output upstream.

Repeating the process for all the elements, we have

$$\begin{aligned} \frac{\partial Loss}{\partial X} &= \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} w_{1,1} + \frac{\partial Loss}{\partial z_{1,2}} w_{1,2} + \frac{\partial Loss}{\partial z_{1,3}} w_{1,3} & \frac{\partial Loss}{\partial z_{1,1}} w_{2,1} + \frac{\partial Loss}{\partial z_{1,2}} w_{2,2} + \frac{\partial Loss}{\partial z_{1,3}} w_{2,3} \\ \frac{\partial Loss}{\partial z_{2,1}} w_{1,1} + \frac{\partial Loss}{\partial z_{2,2}} w_{1,2} + \frac{\partial Loss}{\partial z_{2,3}} w_{1,3} & \frac{\partial Loss}{\partial z_{2,1}} w_{2,1} + \frac{\partial Loss}{\partial z_{2,2}} w_{2,2} + \frac{\partial Loss}{\partial z_{2,3}} w_{2,3} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} & \frac{\partial Loss}{\partial z_{1,2}} & \frac{\partial Loss}{\partial z_{1,3}} \\ \frac{\partial Loss}{\partial z_{2,1}} & \frac{\partial Loss}{\partial z_{2,2}} & \frac{\partial Loss}{\partial z_{2,3}} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \\ w_{1,3} & w_{2,3} \end{bmatrix} \\ &= \frac{\partial Loss}{\partial Z} \cdot W^T \end{aligned}$$

**Derivative of Loss with respect to W**

Again, let's start from one element

$$\frac{\partial Loss}{\partial w_{1,1}} = \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial w_{1,1}} = \frac{\partial Loss}{\partial z_{1,1}} x_{1,1} + \frac{\partial Loss}{\partial w_{2,1}} x_{2,1}$$

Repeating the process for all the elements, we have

$$\begin{aligned}\frac{\partial Loss}{\partial W} &= \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}}x_{1,1} + \frac{\partial Loss}{\partial w_{2,1}}x_{2,1} & \frac{\partial Loss}{\partial z_{1,2}}x_{1,1} + \frac{\partial Loss}{\partial w_{2,2}}x_{2,1} & \frac{\partial Loss}{\partial z_{1,3}}x_{1,1} + \frac{\partial Loss}{\partial w_{2,3}}x_{2,1} \\ \frac{\partial Loss}{\partial z_{1,1}}x_{1,2} + \frac{\partial Loss}{\partial w_{2,1}}x_{2,2} & \frac{\partial Loss}{\partial z_{1,2}}x_{1,2} + \frac{\partial Loss}{\partial w_{2,2}}x_{2,2} & \frac{\partial Loss}{\partial z_{1,3}}x_{1,2} + \frac{\partial Loss}{\partial w_{2,3}}x_{2,2} \end{bmatrix} \\ &= \begin{bmatrix} x_{1,1} & x_{2,1} \\ x_{1,2} & x_{2,2} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} & \frac{\partial Loss}{\partial z_{1,2}} & \frac{\partial Loss}{\partial z_{1,3}} \\ \frac{\partial Loss}{\partial z_{2,1}} & \frac{\partial Loss}{\partial z_{2,2}} & \frac{\partial Loss}{\partial z_{2,3}} \end{bmatrix} \\ &= X^T \cdot \frac{\partial Loss}{\partial Z}\end{aligned}$$

**Derivative of Loss with respect to B**

$$\frac{\partial Loss}{\partial B} = \frac{\partial Loss}{\partial Z}$$

**Considering Activation Function**

If we consider the activation function as well, consider the example

$$Z = f(X) = \begin{bmatrix} f(x_{1,1}) & f(x_{1,2}) \\ f(x_{2,1}) & f(x_{2,2}) \end{bmatrix}$$

Consider only one element

$$\begin{aligned}\frac{\partial Loss}{\partial x_{1,1}} &= \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial x_{1,1}} = \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} & \frac{\partial Loss}{\partial z_{1,2}} \\ \frac{\partial Loss}{\partial z_{2,1}} & \frac{\partial Loss}{\partial z_{2,2}} \end{bmatrix} \cdot \begin{bmatrix} f'(x_{1,1}) & 0 \\ 0 & 0 \end{bmatrix} \\ &= \frac{\partial Loss}{\partial z_{1,1}} f'(x_{1,1})\end{aligned}$$

Putting everything together

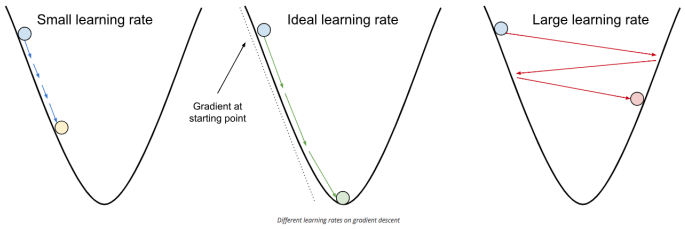
$$\begin{aligned}\frac{\partial Loss}{\partial X} &= \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} f'(x_{1,1}) & \frac{\partial Loss}{\partial z_{1,2}} f'(x_{1,2}) \\ \frac{\partial Loss}{\partial z_{2,1}} f'(x_{2,1}) & \frac{\partial Loss}{\partial z_{2,2}} f'(x_{2,2}) \end{bmatrix} \\ &= \frac{\partial Loss}{\partial Z} \circ f'(X)\end{aligned}$$

### 5.4.2 Gradient Descent

From back propagation, we have the direction to alter the weights to minimise the loss. Gradient descent is a general optimisation technique where we iteratively descend in the direction of the gradient slowly minimising the loss. There is no guarantee we will converge to the global minimum when just walk in the direction of the gradient.

$$W \leftarrow W - \alpha \frac{\partial L}{\partial W}$$

Where $\alpha$ is the learning rate. We often set $\alpha = 0.01$ or $0.001$.




Although currently the learning rate is fixed, there are many extensions to gradient descent that make it adaptive such that the training becomes more efficient. The simplest of them is **learning rate decay** which reduces the learning rate by a factor each epoch

**Stochastic Gradient Descent**

Ideally we would like to compute the gradient using the entire training data, but that is computationally expensive and slow if for example we have thousands of data points. Instead we estimate the true gradient using small random batches.

1. We take training data of size $N$ each with $D$ features, input $X \in R^{N \times D}$
2. Shuffle the network
3. Partition it into batches of size $B$ to get $N/B$ batches with shape $B \times D$
4. Compute the forward pass to collect the network output one batch at a time.
5. Compute the derivative of the loss with respect to the network outputs.
6. Back-propagate the gradients to compute the derivative of the loss with respect to every parameter in the network
7. Update the weights using the given learning rate
8. Repeat 4 to 7 for every batch.
9. Once we do this for every batch, we finished an epoch. Repeat step 2 to 8 until we have done the required number of epochs or reached a convergence criteria.

**Hyperparameters**

Anything that is not learned and is set by the user is a hyper-parameter. Such as number of layers, number of units in each layer, the learning rate.

It is up to the user to find, guess or tune until this entire training process gives desirable outputs.

## 5.5 Evaluation

To ensure that our networks, models are learning something beyond just the training data which is what we use to train with, we need test or validation data that are separate from training.

We use the validation data to tune the hyper parameters of the model, changing

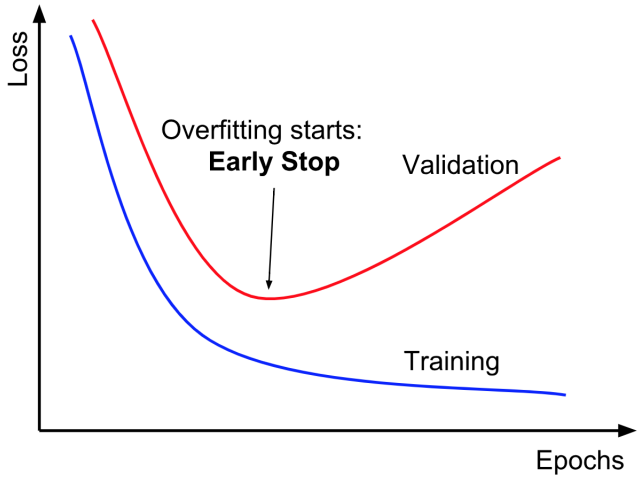
number of layers, units etc. Then the best model on the validation set becomes our final model to test on the test data we have.

### 5.5.1 Overfitting

Over-fitting occurs when the network learns properties specific to the training data rather than the general paradigm.

Under-fitting happens when the network cannot learn the training data at all.

One method for avoiding over-fitting is **early stopping**. It stops the training as the validation loss starts to stray away from the training loss.




Over-fitting starts to happen when validation loss starts to increase while the training loss continues to decrease. The network is learning things that do not apply to the validation set, things that are specific to the training set since the loss for the training continues to decrease.

The best method to any over-fitting problem is to get more data.

#### Capacity

Capacity refers to the number of layers and units those layers have. More layers, more units means the network has more parameters and therefore a bigger capacity to learn more complex functions.

If the network is not learning on the training data, then it might not have enough capacity to learn.

If the network is over-fitting, it has too much capacity and we can look at either reducing it by lowering the number units, layers or apply regularisation

### 5.5.2 Regularisation

Regularisation is way to penalise the model in some way to stop it from over-fitting.

If there is a very useful feature that describes only the training data, the network will latch onto the information to learn the function, but the validation data might not have that.

For example, a neural network is trying to learn about housing prices and the location of the house has great influence on the price. The neural network might solely rely on the location and neglect other factors such as size and age of the house.

We can use regularisation to prevent the neural network to use only one property.

The **L2 regularisation** adds the squared weight $w^2$ to the objective function along side the loss. So if the weight gets larger, so does the loss, in effect to reduce the loss the network needs to keep the weights small as well.

$$J(\theta) = Loss(L, A) + \lambda \sum w^2$$
$$W \leftarrow W - \alpha \left( \frac{\partial L}{\partial W} + 2\lambda W \right)$$

The **L1 regularisation** just adds the absolute value of the weight itself to the objective function.

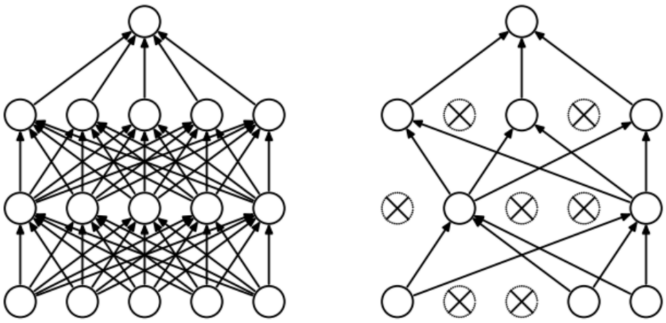
$$J(\theta) = Loss(L, A) + \lambda \sum |w|$$
$$W \leftarrow W - \alpha \left( \frac{\partial L}{\partial W} + \lambda W \right)$$

The $\lambda$ tells how much we want to regularise, we usually set it to 0.01.

Regularisation reduce over-fitting by pushing weights towards 0, a weight of 0 means no connection. No connection means less capacity so in effect we are simplifying the network which removes its ability to learn very specific patterns of the training set.

### 5.5.3 Dropout

Dropout randomly sets outputs of layers to 0 essentially turning off neurons.



(a) Standard Neural Net (b) After applying dropout.

Source: Dropout: A Simple Way to Prevent Neural Networks from Overfitting

With probability $p$ we set the output of the neuron to 0. Every time we apply dropout we get a different sub-network. It is as if we are training smaller networks inside a

larger one which reduces inter-dependency between neurons across layers since it might be dropped next round. So the overall network learns to be more robust with less ability to extract specific features of the training data.

Since we basically dissect our network into smaller ones during training, we might need more epochs to train it increasing the overall training time

### 5.5.4 Data-Preprocessing

Although getting more data is always the best option, more often than not we can better utilise our data by applying some pre-processing.

One common approach is to do **data augmentation** which enhances our existing data. We can generate artificial data based on the original data by adding noise. This gives better generalisation and reduces over-fitting.

Another common technique is **data normalisation** in which the input and potentially the output data is normalised. There are two ways to normalise data:

1. Map the largest value to $b$ and smallest value to $a$, usually [0,1] or [-1,1]

2. Normalise using the mean and standard deviation of the data to make the data have mean 0 and standard deviation 1.

Normalisation is used because updating weights which uses this gradient relies on the magnitude of the input. Thus, having really large or really small inputs destabilises or hampers training respectively.

# 6 Unsupervised Learning

Unsupervised Learning is usually done by clustering. Clustering is the task of grouping a set of objects in such a way that objects in the same cluster are more similar to each other than to those in other clusters.

## 6.1 K Means Clustering

K-means clustering is one of the simplest and popular unsupervised machine learning algorithms.

K means clustering consists of the following steps:

1. Initialisation: Randomly place $K$ centroids in you feature space

2. Assignment: Assign each datapoint to the nearest centroid

3. Update: Update the position of each centroid by computing the mean position of all the data points associated to the centroid.

The algorithm halts when either:

1. The centroids have stabilised?, meaning ?there is no change in their values

2. The defined number of iterations has been achieved.

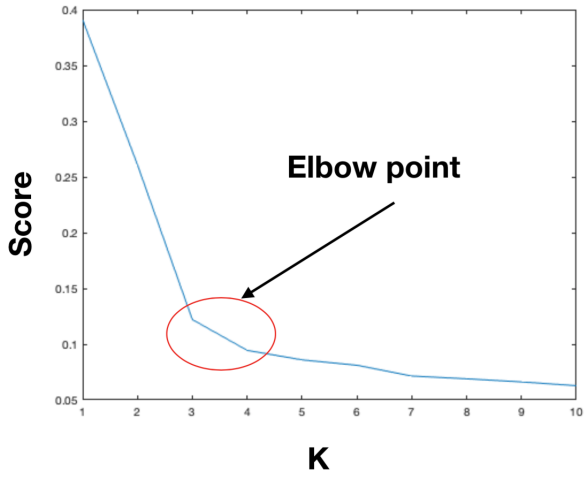
### 6.1.1 Selecting K – Elbow Method

The elbow method runs K-means clustering on the dataset for a range of values for $K$ and then for each value of $K$ computes an average score for all clusters. Usually the score is the average of the mean distance to centroid.

$$score = \frac{1}{K} \sum_{k} Mean\ Distance\ of\ cluster\ k$$

Plot a line chart of the score for each value of k. The line chart looks like an arm, and the ”elbow” on the arm is the best value of $K$.

The idea is that we want to minimise the score, but that the score tends to decrease toward 0 as we increase $K$. So our goal is to choose a small value of $K$ that still has a low score, and the elbow usually represents where we start to have diminishing returns by increasing $K$.




### 6.1.2 Selecting K – Cross Validation

Perform cross validation on various values of $K$ and pick the best configuration.

Select $K$ such that further increase in number of clusters leads to only a small improvement in the average score.

### 6.1.3 Strengths

K means clustering is easy to understand and to implement.

K means clustering is considered a linear algorithm, meaning it is very efficient

### 6.1.4 Weaknesses

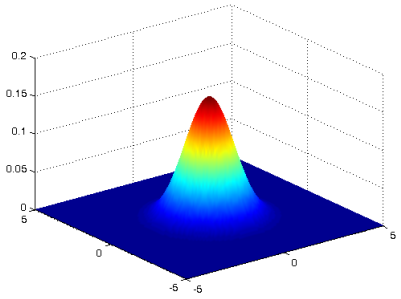
Defining a $K$ is not easy and the algorithm is sensitive to the initial position of the centroids.

The algorithm is sensitive to outliers and it is not suitable for discovering clusters that are not hyper-ellipsoids (hyper spheres)

## 6.2 Density Estimation

Density estimation is the problem of reconstructing the probability density function using a set of given data points. Namely, we observe $X_1, \ldots, X_n$ and we want to recover the underlying probability density function generating our dataset.

### 6.2.1 Multivariate Gaussian Distribution




$$\boldsymbol{\mu} = \frac{1}{N}\sum_{n=0}^{N}\boldsymbol{x}_n \qquad \sum = \frac{1}{N}\sum_{n=0}^{N}(\boldsymbol{x}_n - \boldsymbol{\mu})(\boldsymbol{x}_n - \boldsymbol{\mu})'$$

### 6.2.2 Quantifying the Quality of the Model

Likelihood describes the plausibility, given specific observed data, of a parameter value of the statistical model which is assumed to describe that data.

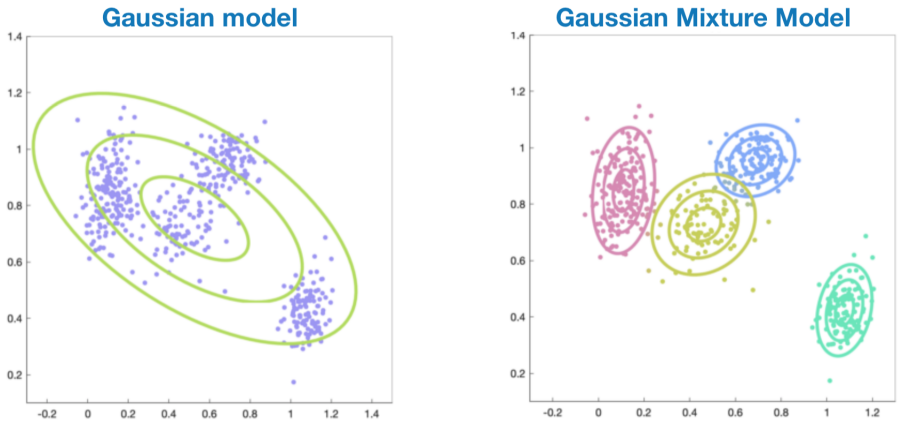
$$Likelihood = p(X|\theta) = \prod_{n=1}^{N} p(x_n|\theta)$$

Likelihood characterises the probability of observing data x from a dataset X, given the parameters of the model $\theta$

Maximum likelihood estimation is a method that determines values for the parameters of a model. The parameter values are found such that they maximise the likelihood that the process described by the model produced the data that were actually observed.

## 6.3 Gaussian Mixture Model (GMM)

Most of the time, the gaussian distribution might not be sufficient to properly model the data distribution. Therefore we might use the combination of several distributions to construct our model. The most famous one is Gaussian Mixture Model




To find the parameters, we use an iterative approach called **Expectation Maximisation (EM)**

Given a Gaussian mixture model, the goal is to maximise the likelihood function with respect to the parameters comprising the means and covariances of the components and the mixing coefficients.

1. Initialise the means, covariances, and mixing coefficients. Then evaluate the initial value of the log likelihood.

2. E-Step: Evaluate the responsibilities using the current parameter values. The responsibility corresponds to the posterior probability of data point $x$ to belong to the mixture component $K$.

3. M-Step: Use the updated responsibilities to update the parameters

4. Evaluate the likelihood log function. If there is no convergence, return to step 2.

# 7 Genetic Algorithms

Evolutionary algorithms are a heuristic-based approach to solving problems that cannot be easily solved in polynomial time.

Evolutionary algorithms are inspired by biological evolution, and use mechanisms that imitate the evolutionary concepts of reproduction, mutation, recombination and selection.

An Evolutionary Algorithm (EA) contains the following steps:

1. Randomly generate a **population** of solutions, where each solution is represented by a **genotype**

2. The genotype can be developed into a **phenotype**

3. A **fitness function** to used to measure the performance of the phenotypes.

4. The fittest individuals have a higher probability to pass part of their genotype to the offspring.

Evolutionary algorithms are often defined according to three main **genetic operators**:

1. Selection operator: Selects the solutions that will be reproduced.

2. Cross-over operator: mixes the parents genotype.

3. Mutation operator: Variations applied to the genotype after reproduction

Different Evolutionary algorithms differ in their **definition of genotype** and their **genetic operators**

## 7.1 Genetic Algorithm

We will use the MasterMind as an example. The goal of the algorithm is to find the correct combination.

### 7.1.1 Fitness Function

Every solution is evaluated by a fitness function to determine how well they can solve the problem. The higher the score, the better the solution.

The fitness function is problem specific. For the Mastermind game, the fitness function is

$$F(x) = p_1 + 0.5p_2$$

Where $p_1$ is the number of pieces with right colour and right position

$p_2$ is the number of pieces with right colour but wrong position

### 7.1.2 Genotype and Phenotype

Genotype and Phenotype represent the potential solutions to the problem. The genotype is the complete heritable genetic identity. The phenotype is a description of the actual physical characteristics.

Genotype and Phenotype are problem specific. For the Mastermind game, the genotype can be represented by a binary string with $N \times 3$ bits where $N$ is the number of pieces. The genotype can be transformed into phenotype by aggregating the bits 3 by 3 and transform each trio into an integer.

000011101001

000 011 101 001

0 3 5 1

Red, Blue, Orange, yellow.

### 7.1.3 Selection operator

The selection operator select the parents for the next generation.

**Biased roulette wheel**

Each solution is associated with a probability. The higher the fitness the higher the probability. Then generate a new population based on the probabilities.

**Tournament**

Randomly draw two solutions from the population, then select the better one and add it to the next generation.

**Elitism**

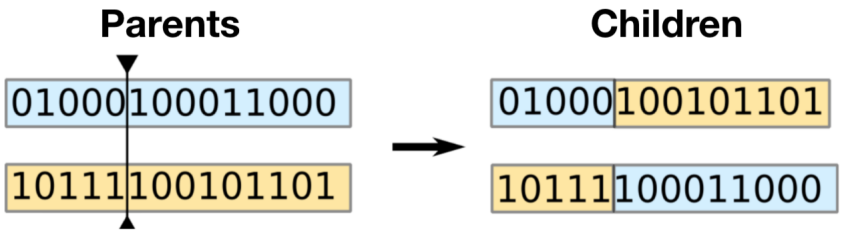
In the general description of an EA, the next generation completely replaces the current generation.

It is often useful to keep in the new generation a fraction (10%) of the best individual found so far. This will guarantee the fitness of the best individual in the population will strictly increase.

### 7.1.4 Cross-over operator

The selection operator combine the traits of the parents.

One way to combine parents is to select a random split point then produce a child by exchanging the portions of genotype




### 7.1.5 Mutation operator

The mutation operator is used to explore nearby solutions.

A standard mutation on binary strings is to randomly generate a number between 0 and 1 for each bit, if the value is lower than $1/length\ of\ genotype$, flip the bit.

## 7.2 Evolutionary Strategy

Evolution Strategy is an Evolutionary Algorithm for real numbers. Different to genetic algorithm, it does not need cross-over method and selection method.

### 7.2.1 $(\mu + \lambda) - ES$

The algorithm works as follow:

1. Randomly generate a population of $\mu + \lambda$ individuals
2. Evaluate the population
3. Select the $\mu$ best individuals from the population as parents $(X)$
4. Generate $\lambda$ offsprings $(Y)$ from the parents: $y_i = x_j + N(0, \sigma)$, where $x_j$ is randomly chosen.
5. The new population is the union of parents and offsprings
6. Repeat from step 2?

The value of $\sigma$ is crucial:

1. A large sigma means the population will move quickly to the solution but has hard time to refine it
2. A small sigma means the population moves slowly and might be affected by local optimums.