Introduction to Machine Learning

There are three types of machine learning:

Supervised Learning
Learning from examples. *E.g. object detection.*

Reinforcement Learning
Learning by trial and error. *E.g. traffic control.*

Unsupervised Learning
Learning the structure of the data without knowing what you're looking for. *E.g. customer segmentation.*

Within these, you can have **parametric** and **non-parametric models**.

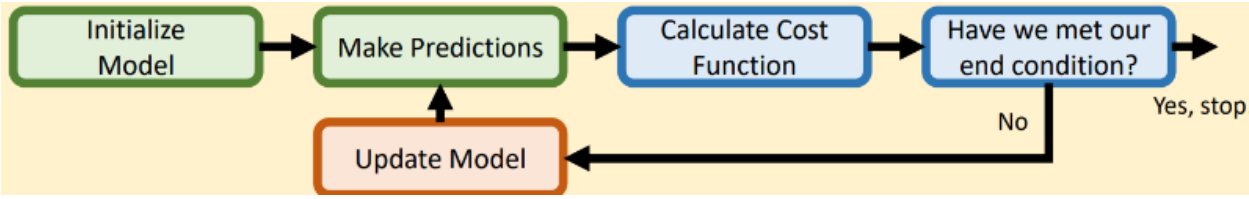
Parametric Models	***Non-parametric Models***
Assume a **predefined structure** which relies on parameters. Training data optimises parameters and **isn't needed to make predictions**.	**Doesn't assume structure** and can grow with data complexity. No parameter optimisation but training **data is needed to make predictions**, so they're **very reliant** and **sensitive on training data.**

Supervised Learning

To create a model, you need to know:

- **Type:** Parametric or non-parametric.
- **Hypothesis Function:** Converts inputs to output predictions.
- **Loss/Cost/Objective Function:** Evaluates model performance/error/cost.
- **Optimiser:** Determines improvements.

Training a model generally works like this:



Types of Data Sets

- **Training:** Finds the correct parameters.
- **Validation:** Optimises the hyperparameters. *e.g. the number of nearest neighbours.*
- **Testing:** Evaluates the model.

Linear Regression

Regression problems predict continuous values. Linear regression tries to find the **line of best fit** in a dataset by optimising the **parameters**.

Hypothesis Function

The same as **the equation of a line**. Linear regression focusses on **optimising** this.

Let x be the input vector, θ be the parameters and n be the number of features.

$$x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

It always has a **bias**, a parameter without input, to shift the line.

The hypothesis function is:

$$h_\theta(x) = \sum \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_n x_n$$

Finding the Output for a Set of Parameters

"A gem merchant wants to create a model to predict the price of a gem based on its weight and purity."

$$\text{The hypothesis function is: } h_\theta(x) = \theta_0 + \theta_1 x_{weight} + \theta_2 x_{purity}$$

Test a set of parameters $\theta = \{2, 2, 2\}$ using the input and parameter vectors:

$$h_\theta(x^0) = \sum \theta^T x = \sum \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix}^T \begin{bmatrix} 1 \\ 5 \\ 100 \end{bmatrix} = \sum [2 \quad 2 \quad 2] \begin{bmatrix} 1 \\ 5 \\ 100 \end{bmatrix} = \sum \begin{bmatrix} 2 \\ 10 \\ 200 \end{bmatrix} = 212$$

Note: The first input is always a bias of 1!

Cost Function

Mean-squared error evaluates the cost of a set of parameters. The output vector of testing samples is compared to their actual output vector y.

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i)^2$$

m - the number of predictions.

Example

$$J(\theta) = \frac{1}{2(3)} \left((212 - 500)^2 + (46 - 40)^2 + (167 - 120)^2 \right)$$

$$J(\theta) = 14198.167$$

Weight	Purity	Price	Predictions
5	100	500	212
2	20	40	46
2.5	80	120	167

Optimiser

Linear regression uses **gradient descent**, an iterative algorithm which finds optimal parameters by minimising the cost function. For each parameter θ_j, we do:

$$\theta_j = \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

$$\theta_j = \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_j^i$$

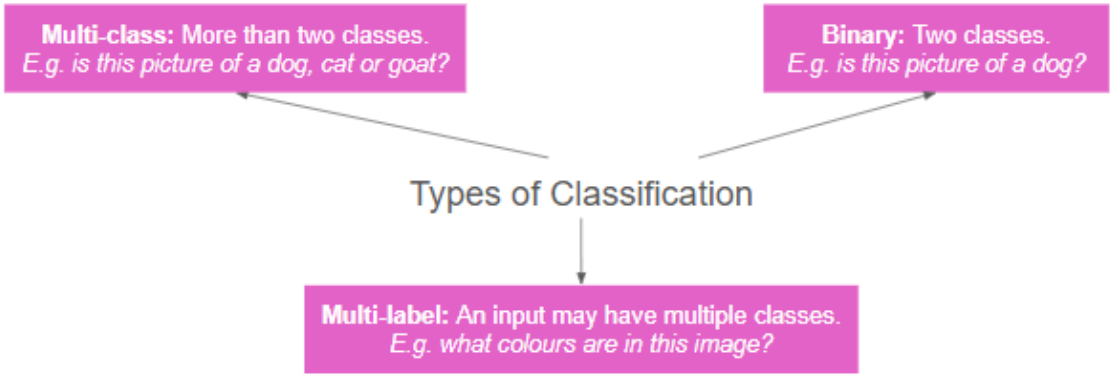
It iteratively steps down the gradient, with the rate of change dictated by the **learning rate** α. If α is:

- Too high it will struggle to find a minimum.
- Too long it will take too many epochs.

See the walkthrough on the PPT!

Logistic Regression

Logistic regression **produces a probability,** using **parameters,** which is used for **classification**!



Hypothesis Function

The **logistic/sigmoid function** gives the estimated **probability of a binary class**.

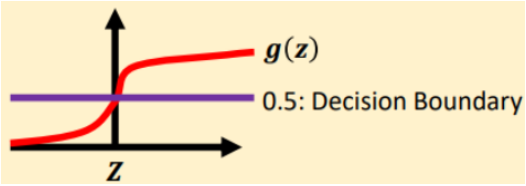
Increasing the magnitude accounts for non-linear decision boundaries.

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

$$g(x) = \frac{1}{1 + e^{-x}}$$

Decision Boundaries

Specifies the boundary in a binary classification problem. *e.g. if it's 0.5, any results over 0.5 are classified as that class and any under aren't.*



The sigmoid function always crosses the y-axis on the decision boundary. i.e.

$$h_\theta(x) = g(\theta^T x) \geq 0.5 \; where \; \theta^T x \geq 0$$

Using this, substituting values in and recalling that $x_0 = 1$ gives an equation for the decision boundary.

$$\begin{aligned} \theta^T x \geq 0 &\implies \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 \geq 0 \\ &\implies \theta_0 + \theta_1 x_1 + \theta_2 x_2 \geq 0 \\ &\implies -100 + x_1 + x_2 \geq 0 \end{aligned}$$

Cost Function

With sigmoid, MSE leads to a **non-convex cost function**, so the **average log loss function** is used instead! 0 means the prediction is perfect and -1 it's very wrong!

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^i log(h_\theta(x^i)) + (1-y^i)log(1-h_\theta(x^i))]$$

m - the number of predictions.

Example - One Sample

$$J(\theta) = y^i \log\left(h_\theta\left(x^i\right)\right) + \left(1-y^i\right)\log\left(1-h_\theta\left(x^i\right)\right)$$
$$J(\theta) = 1\log\left(h_\theta\left(0.8\right)\right) + \left(1-1\right)\log\left(1-h_\theta\left(0.8\right)\right)$$
$$J(\theta) = \left(0.8\right) + \left(0\right)\log\left(0.2\right) = \log\left(0.8\right) = -0.1$$

Pred. $(h_\theta(x))$	Truth (y)
0.8	1

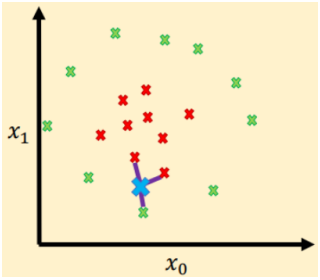
Multi-class Classification

Run the input through hypothesis functions, one for each class, and assign it the class with the highest probability over the decision boundary.

K-Nearest Neighbours (KNN)

A **non-parametric classification** approach. Given a training data set, select the nearest k neighbours to the point and select the majority class.

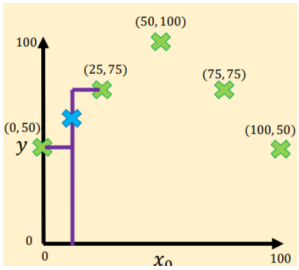
- Useful for **non-linearly separable** datasets.
- k dictates how **generalised the model is** to the dataset.



Using for Regression

Calculate the average output of the nearest k neighbours.

- $k = 1$ prevents **extrapolation**.
- $k = n$ takes the average of all points.



Advantages	*Disadvantages*
• No training needed. • **Extrapolates**. • Easily **explainable**.	• **Lots of Memory**: Must store training data. • **Sensitive** to training data & hyperparameters.

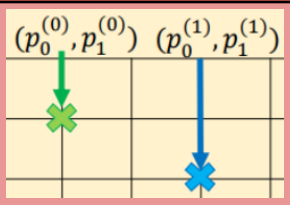
Distance Metrics

Used in many models, like KNN. You **can't combine them to combine categorical and continuous data** because they're different scales.

Euclidean Distance

As-the-crow-flies distance between two points.

- For **Continuous** Data!
- **Best on Low-dimensional Data:** Hides deviations at high dimensions.



$$\sqrt{\sum_{i=0}^{d}(p_i^{(0)} - p_i^{(1)})^2}$$

Manhattan Distance

The absolute difference between two points.

- For **Continuous** Data!
- Good for **Sparse Data.**
- **Lower Computational Cost.**

$$\sum_{i=0}^{d}\left|p_i^{(0)} - p_i^{(1)}\right|$$

Cosine Similarity

The magnitude of the angle between two points.

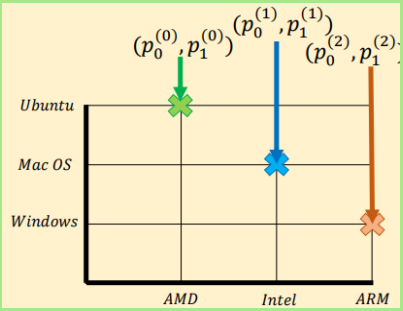
- For **Continuous** Data!
- Best for **Comparing Different Scales.**

$$\frac{\sum_{i=0}^{d}(p_i^{(0)}p_i^{(1)})}{\sqrt{\sum_{i=0}^{d}(p_i^{(0)})^2}\sqrt{\sum_{i=0}^{d}(p_i^{(1)})^2}}$$

Hamming Distance

Represent points using **one-hot encoding** and calculate the differences between them.

- For **Categorical** Data!



$$p^{(0)} = ['AMD', 'Ubuntu']$$
$$p^{(0)} = [[1,0,0],[0,0,1]]$$
$$p^{(0)} = [1,0,0,0,0,1]$$

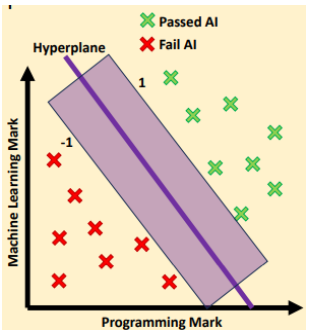
$$HD(p^{(0)}, p^{(1)}) = 4$$
$$HD(p^{(0)}, p^{(2)}) = 4$$
$$HD(p^{(1)}, p^{(2)}) = 4$$
$$HD(p^{(0)}, p^{(3)}) = 2$$

Jaccard Similarity	
Treat points as sets and calculate the difference between them. • For **Categorical** Data!	$J(p^{(x)}, p^{(y)}) = \frac{\lvert p^{(x)} \cap p^{(y)} \rvert}{\lvert p^{(x)} \cup p^{(y)} \rvert}$

Maximal Margin Classification

Another method of finding a **decision boundary** to separate classes, used in SVMs. However, it **optimises the boundary's position** by maximising its margin using **parameters**.

- Mitigates overfitting.
- Works for **linearly separable** datasets.



Hypothesis Function

MMCs have a **center hyperplane** given by:

$$\mathbf{wx} - b = 0$$

The hypothesis function **labels input vectors by their sign**, a.k.a. which side they're on:

$$h_w(x) = sign(\mathbf{wx} - b)$$

Cost Function

$$\|\mathbf{w}\| = \sqrt{\sum_{j=1}^{D} (w^{(j)})^2}$$ The Euclidean norm of **w**.

Optimiser

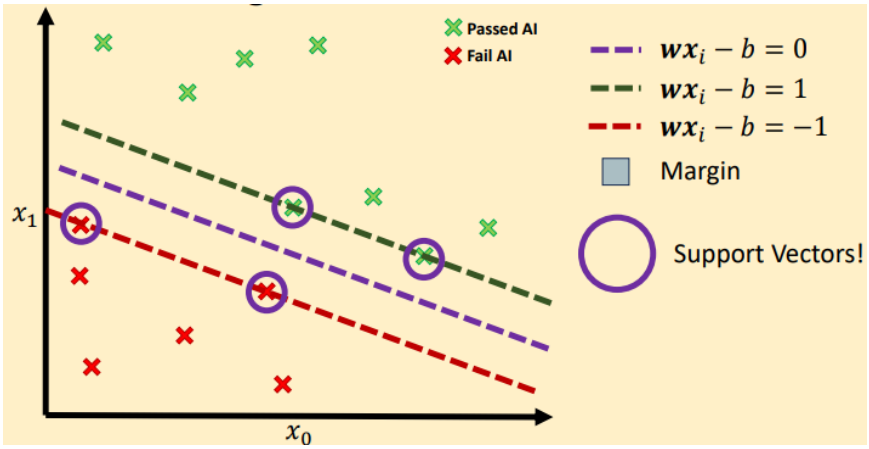
Optimises the parameters **w** and b to make the **widest margin between points** by **minimising the Euclidean norm of w** while conforming to the constraints:

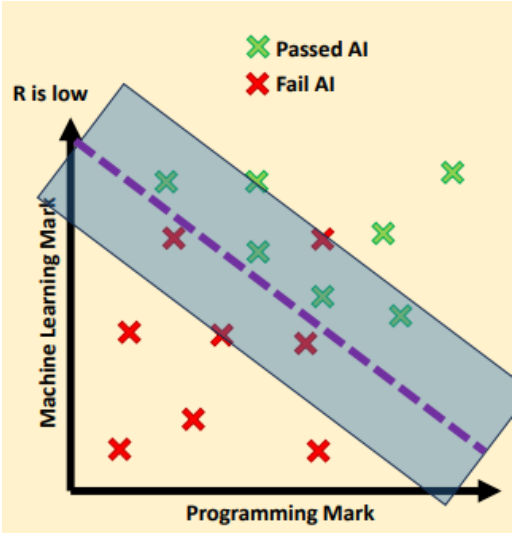
$$wx_i - b \geq +1 \quad \text{if } y_i = +1 \qquad wx_i - b \leq -1 \quad \text{if } y_i = -1$$

So, we have a **constraint satisfaction problem!**

You don't need to know how this is solved!

Solving this generates two parallel hyperplanes with **support vectors** from each classification.





Soft-Margin Classification

Similar to MMCs but can classify **non-linearly separable data** using a new cost function "**Hinge Loss**":

$$R\|\mathbf{w}\| + \frac{1}{N}\sum_{i=1}^{N}\max(0, 1 - y_i(\mathbf{w}\cdot\mathbf{x}_i - b))$$

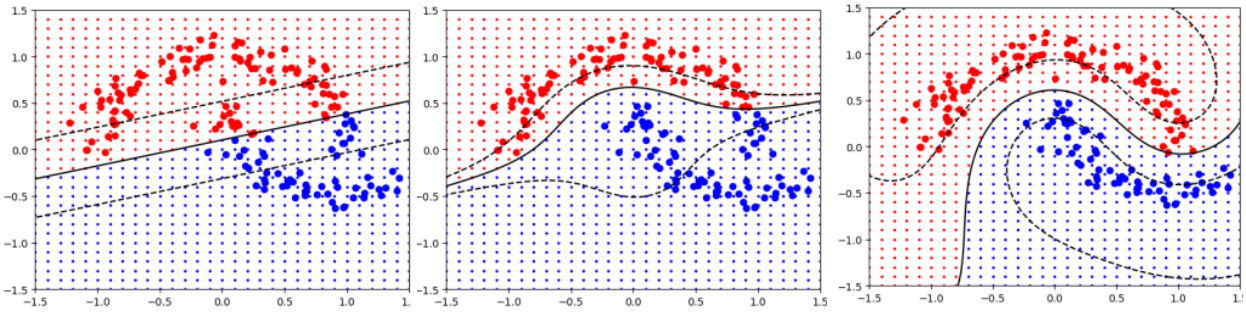
Correct items have a cost of 0 and incorrect items have a **cost proportional to their distance from the margin**.

R dictates the strength of the constraints.

- Low R = Large margin.
- High R = Small margin.

Support Vector Machines (SVMs)

SVMs utilise these classification methods and apply a **kernel to the data** to enable more effective classification. **Kernels map data to higher dimensions**, making it easier to linearly separate, and model **non-linear relationships**.



No Kernel *Polynomial Kernel (d=3)* *RBF Kernel*

Decision Trees

A **non-parametric classification model** consisting of:

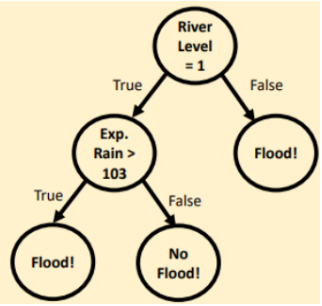
- **Decision Nodes** - Ask binary questions. *e.g. is* $x \leq 500$
- **Lead Nodes -** Provide a final result.
- **Edge Connections**

Constructing Decision Trees

1. Calculate the number of features & classes.
2. Use the **entropy information game** to find the feature which best splits the output.

Example

River	River Level (m)	Expected Rainfall (mm)	Flood?
Ray	1	103	No
Ray	2	200	Yes
Ray	3	45	Yes
Ray	1	62	No
Ray	1	300	Yes



Picking a feature uses a **greedy approach**, which means our choice only depends on the current state, because **full optimisation is too computationally expensive**.

The Entropy Information Game

Tests each feature to see which provides the most information about splitting the output classes.

Homogeneity: Similarity **Heterogeneity:** Difference

Entropy: A measure of homogeneity in a dataset.

- $E(x) = 1$ means the number of times each class occurs is equal.
  Max heterogeneity!
- $E(x) = 0$ means that only one class appears in the data set.
  Max homogeneity!

$$E(x) = -\sum_{i=1}^{c} P(i) * log_2(P(i))$$

where:

- $E(x)$ is the entropy of a feature.
- c is the number of output classifications.

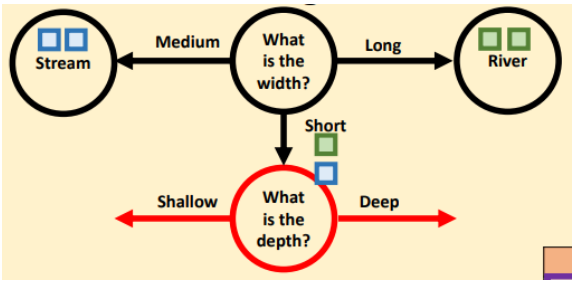
Steps - See PowerPoint for Example

1. Calculate the probability of each output class appearing.
$$P(x) = \frac{num\ of\ occurances\ of\ x}{num\ of\ occurances}$$
2. Calculate the entropy of the output classes using the formula.
3. For each feature, calculate the output class entropy for the samples which take that feature:
   a. For each class, calculate the entropy of the output classes.
   b. Calculate the feature's average entropy.
4. Work out the information gain for each feature:
$$G(y) = E(x) - E(y)$$

where:

- $G(y)$ - information gain from splitting on feature y.
- $E(x)$ - entropy before splitting.
- $E(y)$ - entropy after splitting.

5. Split on the feature with the highest information gain.
6. If this doesn't fully split the output classes, narrow down to the feature class which still has mixed output classes and repeat the process with the data that makes it to there.



Handling Continuous Features

1. Rank order the elements.
2. Calculate the midpoints between each.
3. Calculate the information gain for all midpoints.
4. Select the midpoint with the highest gain.

Min. Width	True Label
2	River
1	River
3	River
0.5	Stream
0.5	Stream

$0.5, 0.5, 1, 2, 3$

$0.5, 0.75, 1.5, 2.5$

$G(< 0.5), G(< 0.75), G(< 1.5), G(< 2.5)$

Pick the lowest!

Hyperparameters for Generalisation

Max Depth

Setting a max depth mostly avoids overfitting by preventing it from fitting exactly to the values.

Minimum Number of Samples

Set a min. number of samples, preventing nodes under from developing more branches and mitigating overfitting.

Random Forest

A **non-parametric ensemble model** comprised of a **collection of decision trees**. They **overcome** decision tree's **overfitting problem** by allowing the model to **generalise**. The collective output of these trees is used to select the output:

- **Classification:** Select the modal output.
- **Regression:** Select the mean or median.

Constructing Random Forests

1. Create a **bootstrapped data set**.
    a. Select a random row from the training data.
    b. Append it to a new dataset.
    c. Repeat until the new dataset is the same size as the training dataset.
2. Use this to **create a large number of decision trees**:
   At each node, only consider splitting on a random subset of features, ensuring tree variety.

Flood Forecasting(Training)			
Date	Type	Rainfall	Flood
21/02/2023	A	207	True
02/02/2023	B	103	False
02/02/2023	C	95	False
19/01/2023	A	195	True

Not every item must be used and they can be duplicated!

Computational Limits

Creates multiple decision trees and combines their outputs, and is non-parametric so only needs to store the training data. So, they need **a lot more processing power than memory.**

Advantages	***Disadvantages***
• No complex hyperparameters. • Don't overfit. • Robust to outliers.	• Computationally inefficient. • Reliant on the number of trees. • Difficult to explain results.

Naïve Bayes

A **non-parametric probabilistic classification model** based on **Bayes' Theorem**.

Bayes' Theorem

A way of using **prior knowledge** to calculate a **conditional probability.** Where:

$$P(A|B) = P(A) \times \frac{P(B|A)}{P(B)}$$

- $P(A|B)$ - a **posterior**.
- $P(A)$ - a **prior**.
- $P(B|A)$ - a **likelihood**.
- $P(B)$ - **a marginal probability**.

It's **naïve** because we assume each feature is **independent**!

Example

Given the following data, what is the probability of most students walking to university on a sunny day?

Weather Type	Majority Walk
Sunny	Yes
Rainy	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Sunny	No
Overcast	No

$$P(\text{Yes} \mid \text{Sunny}) = P(\text{Yes}) \times \frac{P(\text{Sunny} \mid \text{Yes})}{P(\text{Sunny})}$$

Calculating the values:

$$P(\text{Yes}) = \frac{\text{No.Yes}}{\text{No. Items}} = \frac{5}{8} \qquad P(\text{Sunny}) = \frac{\text{No. Sunny}}{\text{No. Items}} = \frac{4}{8}$$

$$P(\text{Sunny}|\text{Yes}) = \frac{\text{No. Maj. Walk on Sunny}}{\text{No. Maj. Walks}} = \frac{3}{5}$$

Substituting them in: $$P(\text{Yes}|\text{Sunny}) = \frac{5}{8} \times \frac{\frac{3}{5}}{\frac{4}{8}} = \frac{3}{4}$$

Multi-Feature Naïve Bayes

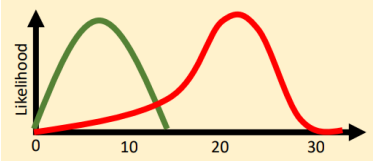
To calculate the **probability of an outcome given a set of conditions**:

$$P(A \mid x_0 \cap x_1 \cap \ldots \cap x_n) = P(A) \times \frac{P(x_0 \cap x_1 \cap \ldots \cap x_n \mid A)}{P(x_0 \cap x_1 \cap \ldots \cap x_n)}$$

which is equal to:

$$P(A \mid x_0 \cap x_{\ldots} \cap x_n) = P(A) \times \frac{P(x_0 \mid A) \cdot P(x_{\ldots} \mid A) \cdot P(x_n \mid A)}{P(x_0) \cdot P(x_{\ldots}) \cdot P(x_n)}$$

Morning Rain (mm)	Majority Bus
0.5	Yes
7	No
...	...
1.2	Yes
10	Yes



Handling Continuous Features: Gaussian Naïve Bayes

1. Fit Gaussian distributions to each class (values which A can take) by finding the mean, standard deviation and applying the formula.
2. Use them to find the probabilities with continuous values.

e.g. you'll need it to calculate:

$$P(\text{10mm of rain} \mid \text{Maj. Bus})\ P(\text{10mm of rain} \mid \text{Maj. Bus})$$

Metrics For Evaluating Multi-Class Classifiers

Accuracy

The percentage of correctly classified cases.

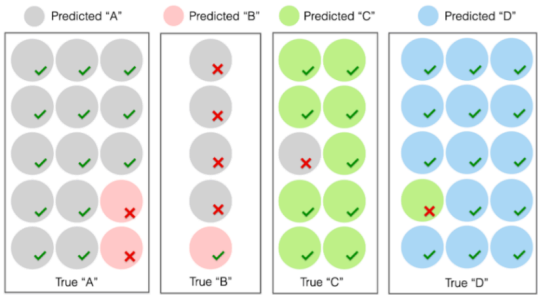
$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{All Predictions}}$$

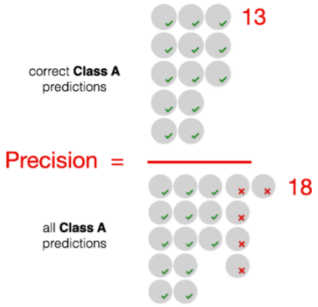
Precision

The model's ability to correctly identify instances of a specific class. For a given class, calculate:

$$Precision = \frac{True\ Positives}{(True\ Positives)\ +\ (False\ Positives)}$$

Example



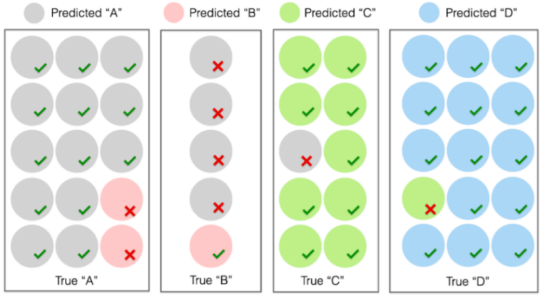


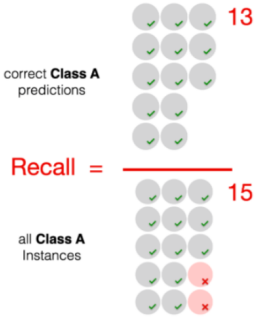
Recall

The model's ability to correctly identify all instances of a particular class. For a given class, calculate:

$$Recall = \frac{True\ Positives}{(True\ Positives)\ +\ (False\ Negatives)}$$

Example





Mean Absolute Error

Similar to MSE but doesn't exacerbate errors because it doesn't square anomalies.

$$MAE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(|y_i - \hat{y}_i|)$$

Root Mean Squared Error

Uses the same units as the model output.

$$RMSE(y, \hat{y}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

R^2 Metric

How much change in your target variable can be explained by the input variables.

If R^2 = 0.75, 75% of the changes in the dependent/target variable can be predicted by the independent variables, while 25% is due to other factors not included in the model.

Feature Engineering

Correlation doesn't imply causation!

If there's no causation, we can't rely on the relationships the model learns for future cases!
Prioritise data which has a **causal impact** on the output variable.

Feature Creation

Creating features in our dataset can provide more causal information.
e.g getting price per meter from the area and total price.

Distance-based Methods *e.g. k-nearest neighbours*
Use when the distance between features is more important.

Grouping-based Methods *e.g. decision trees*
Splits the data into independent groups.

Regression-based Methods *e.g. linear regression*
Use when labels have a linear dependency on features.

Total Price (£)	£ per m^2	City
500,000	18,518	Bath
250,000	25,000	Bath
403,000	9,372	Swindon
...	...	...

Handling Categorical Features

Categorical features, like city here, must be converted to be handled by a model.

Just labelling them doesn't work because it implies distance when there isn't any.

One-hot Encoding
Adding artificial features for each option which represent a binary choice.

Total Price (£)	£ per m^2	City (Bath)	City (Swindon)
500,000	18,518	1	0
250,000	25,000	1	0
403,000	9,372	0	1
...	...	...	

Creating New Categorical Features

Turning continuous features categorical can help to generalise and prevent overfitting!

...	Age	...
...	17	...
...	16	...
...	19	...
...	25	...
...	...	...

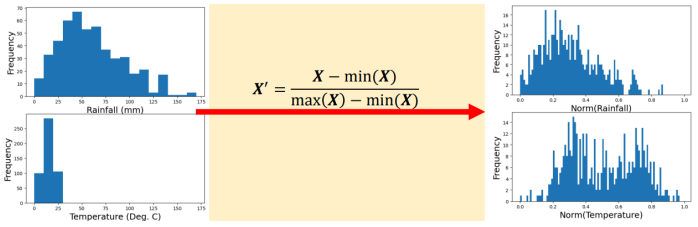
...	Child	Young Adult	...
...	1	0	...
...	1	0	...
...	0	1	...
...	0	1	...
...	...		...

Data Scaling

Features with large differences will **disproportionately influence the model**. In gradient descent, they would cause vastly **different parameter updates** for each feature and a **noisy curve**. There are two types of **scaling** to prevent this:

Normalisation

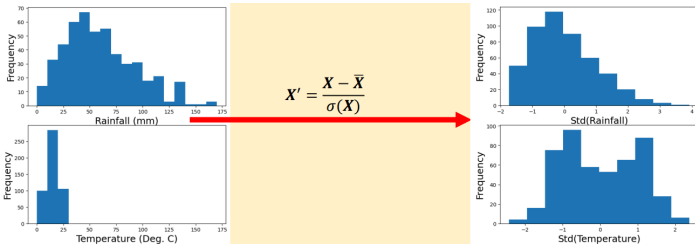
- Scales data between 0 and 1.
- Doesn't distort the range.
- Outliers cause condensation around the mean.



Don't memorise.

Standardisation

- Scales data by the standard deviation.
- Useful with outliers.
- Bad if you want all features on the same scale.



Don't memorise.

Governance: Overseeing the control and direction of something.

Governance, Ethics & Legislation

Governance

The datasets, models and results are artefacts unique to machine learning which can be monitored and tracked with **data cards** and **model cards**.

Data Cards

Should be created for each data set.

- **Clarifies the Data's Lifecycle:** how it was collected, processed and will evolve.
- **Helps Select Data**.
- **Identifies Issues During Monitoring**.

e.g. name, description, authors, purpose, applications, intended use-cases, motivations, data types, etc.

Model Cards

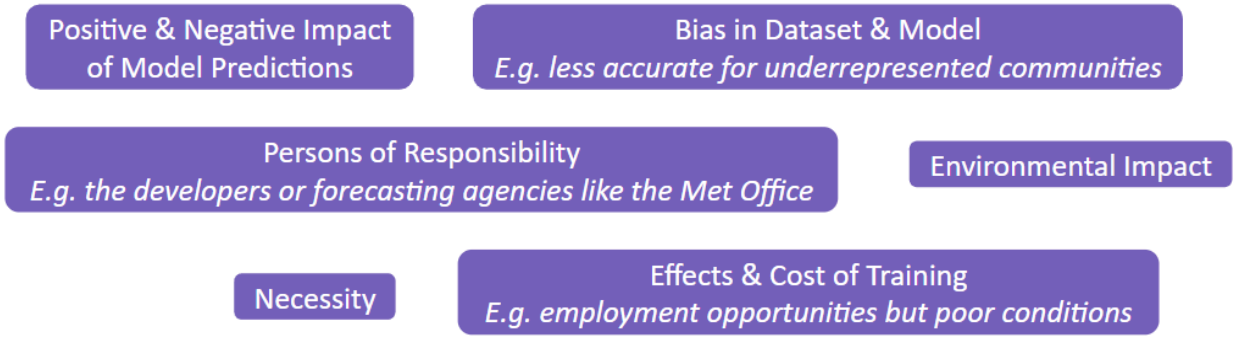
Should be created for each model.

- **Clarifies Intentions.**
- **Describes External Factors:** Could be relevant later. *e.g. environmental conditions, demographics.*
- **Specifies How to Check Performance.**

e.g. developer, description of model, references, use cases, factors, metrics, datasets and ethical considerations.

Ethics

Ethical considerations should be made at the planning stage.



Legislation

There isn't any explicit AI legislation in the UK. Instead, it's regulated through:

- The Data Protection Act
- Equality & Human Rights Laws
- Intellectual Property Laws

US Legislation

Being written and set to cover:

- Safety features
- Algorithmic bias & discrimination protections.
- Data privacy.
- Notice of use and expectations.
- Human alternatives.

The EU AI Act

Different types of AI have different rules and restrictions:

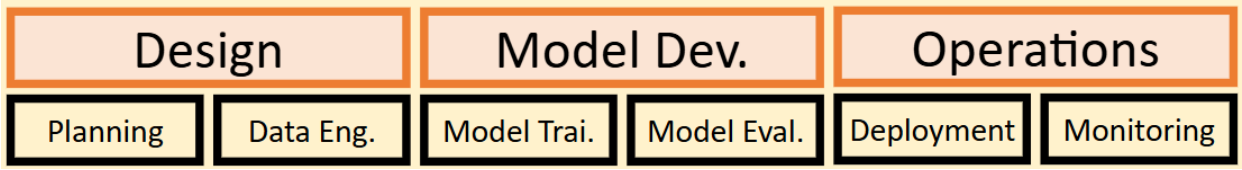
General Purpose
- Transparency requirements.
- Design limitations.
- Copyrighted data must be summarised.

High Risk *e.g. toys, aviation, cars, medical devices*
Models relating to critical infrastructure, like education, employment and law, must be registered.

Unacceptable Risk
Banned applications:
- Cognitive manipulation of people.
- Social scoring systems.
- Biometric identification & classification.
- Real-time and remote biometric identification.

Machine Learning Operations

Developing a machine learning system is similar to software development:



- Requirements analysis.
- Feasibility studies.
- Identifying data.
- Model selection.
- Data management & governance.
- Data pre-processing.
- Data selection.

- Hyperparameter tuning.
- Documentation.
- Client/customer feedback.

- Cloud infrastructure.
- Software dev.
- Hardware & performance monitor.
- Data monitoring.

Potential Issues

Changing data streams & model expectations are common problems with real-time datasets. **Monitoring** these aspects enables identification and model modification.

Data Drift/Skew

Sudden or slow changes to the input data distribution over time, reducing model performance.
e.g. developing a model for person detection with security cameras and they're changed, or developing a model to detect culvert blocking but global warming changes the data.

Concept/Model Drift

The mapping from input to output changes over time.
e.g. the definition of spam changing over time.

Software as a Service (SaaS) Applications

Enables data science to be done **in the cloud** by creating model pipelines using prebuilt components, usually built like a flowchart.

Unsupervised Learning

See probability & linear algebra notes!

ML datasets are stored as $X = \begin{pmatrix} x_1^T \\ \vdots \\ x_n^T \end{pmatrix}$ where each $x_1^T = [x_{1,1} \quad x_{1,2} \quad \cdots \quad x_{1,d}]$ is a sample/observation.

They're actually row vectors but are stored as column vectors so the features line up vertically.

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}$$

Sample/Observation

Feature/Variable

Principal Component Analysis (PCA)

Simplifies complex datasets by **transforming data into a smaller set of orthogonal** (perpendicular) **components, redefining the coordinate system to emphasise the variance**. It reduces the dimensions while retaining as much data as possible, improving data analysis efficiency.

Principal components are new variables created which capture the max variance. They're the eigenvectors with the largest eigenvalues and represent the directions with the highest variance.

Dimensionality Reduction

Can be used reduce the number of features in the dataset from n to k while preserving as much variance as possible, because it indicates meaningful patterns and relationships.

Assume the data follows a multivariate Gaussian distribution, if it does PCA is particularly effective, because it has clear variance patterns which are well-captured by the covariance matrix. The multivariate normal distribution uses a **mean vector** and a **covariance matrix**, which are used in PCA!

Multivariate Gaussian Dist. Formula

$$Norm(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})}{2}\right]$$

The optimal parameters formulae (maximum likelihood estimation) finds the best parameters for describing the Gaussian distribution and so the data.

Optimal (maximum likelihood) parameters for n samples:

$$\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i, \quad \boldsymbol{\Sigma} = \frac{1}{n-1}\sum_{i=1}^{n}(\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^T$$

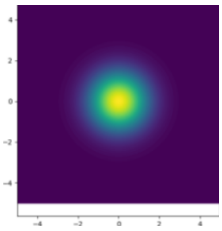
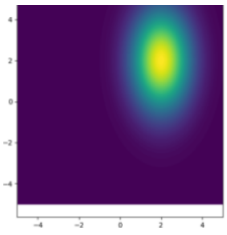
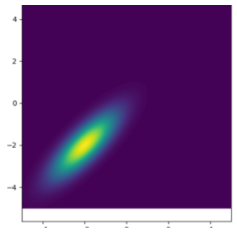
Covariance/Multivariate Gaussian Density

Indicates the direction of the linear relationship between variables.

- **Positive:** Increase together.
- **Negative:** One increases & one decreases.

See covariance formula in linear algebra notes.

Plotting the covariance matrix, there are three types:

Spherical	***Diagonal***	***Full***
Only diagonal and all have the same variance - **no correlation between variables.**	Only diagonal with different variances - **no correlation between variables.**	All have different variances - **Maybe correlations between variables.**
$\Sigma = \begin{bmatrix} \sigma^2 & 0 \\ 0 & \sigma^2 \end{bmatrix}$	$\Sigma = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}$	$\Sigma = \begin{bmatrix} \sigma_{11}^2 & \sigma_{12} \\ \sigma_{21} & \sigma_{22}^2 \end{bmatrix}$
		

Eigendecomposition

Decomposing the variance helps to find the principal components. $\Sigma = V\Lambda V^T$

V is the matrix of eigenvectors, the direction of each principal component, and Λ is the matrix of eigenvalues, the variance of each principal component.

Spherical

$$\boldsymbol{\mu} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \Sigma = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Diagonal

$$\boldsymbol{\mu} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \Sigma = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

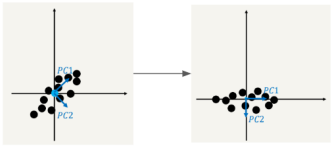
Full

$$\boldsymbol{\mu} = \begin{pmatrix} -2 \\ -2 \end{pmatrix}, \Sigma = \begin{pmatrix} -0.71 & 0.71 \\ 0.71 & 0.71 \end{pmatrix} \begin{pmatrix} 0.2 & 0 \\ 0 & 1.8 \end{pmatrix} \begin{pmatrix} -0.71 & 0.71 \\ 0.71 & 0.71 \end{pmatrix}$$

Changing an eigenvalue to zero reduces the dimensionality! So, **taking the highest eigenvalues and their vectors keeps the most variance but reduces the dimensions**.

Rotating the Data to Emphasize Variance

Once the eigenvectors & values have been found, rotate the data to the principal axes/components to emphasise the variance (see formula below).



The PCA Algorithm

1. Input $X = \begin{pmatrix} x_1^T \\ \vdots \\ x_n^T \end{pmatrix}$
2. Find the mean $\bar{x}^T$ and calculate the matrix $\mathrm{B} = \begin{pmatrix} x_1^T - \bar{x}^T \\ \vdots \\ x_n^T - \bar{x}^T \end{pmatrix}$
3. Calculate the covariance matrix $\Sigma = \frac{1}{n-1} B^T B$.
4. Compute eigenvectors/eigenvalues of $\Sigma = VDV^T$
   - $D \in \mathbb{R}^{d \times d}$ is a diagonal matrix with the eigenvalues in the diagonal
   - $\mathrm{V} \in \mathbb{R}^{d \times d}$ a matrix with columns the corresponding eigenvectors
5. $\mathrm{P} = BV$ gives a rotated representation of the data on the principal axes, if we want to whiten the data, we can normalize by the standard deviation $\mathrm{P} = BVD^{-\frac{1}{2}}$

To reduce dimensions, take a subset of the eigenvectors & values.

White Noise

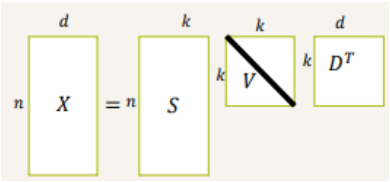
A vector of variables is a white noise vector if the **mean is zero** and its **covariance matrix is the identity matrix**. Means the variables are **independent** and the vector is **rotation invariant**. Whitening data can improve ML performance. PCA can derive a whitening transform.

A **whitening transform** is a linear transform that converts data's covariance matrix to the identity matrix.

Singular Value Decomposition

SVD is similar to eigendecomposition but can decompose **any, not just square, linear transformation matrix** into three components which represent the scaling and rotations it carries out. Removing the smallest singular values and their vectors creates the **optimal low-rank approximation** of the matrix, capturing the **most important relationships**, **reducing size and noise**.

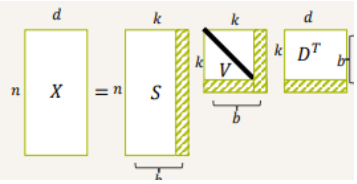
Any matrix $X \in R^{n*d}$ can be broken down to $X = SVD^T$. Where:



- S & D's columns are **orthonormal basis / singular vectors** representing the directions of X's transformation.
- V is a diagonal matrix of X's **ordered singular values** representing the importance of each direction.

k = min(n, d) gives a **full rank** decomposition, which describes X exactly!

Rank: The number of linearly independent columns/rows.



Rank-Deficient Matrices

If $rank(X) < min(n, d)$, the lower diagonal parts of V will be 0, so the vectors and values can be omitted. Set $k = rank(X)$ every time to remove them.

Low-Rank Approximation

Retaining the highest k singular values and vectors, produces $X^{(k)}$ which the **best possible rank-k approximation of X** in terms of minimising the error between them.

As you remove singular values, remove their vectors like this:

$$\begin{pmatrix} 2 & 0 \\ 1 & 2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix}$$

The Frobenius Norm

Measures the error between matrices.

$$\|X - X^{(k)}\|_F = \sqrt{\sum_{ij} (X_{ij} - X_{ij}^{(k)})^2}$$

e.g. $\left\| \begin{pmatrix} 2 & 0 & 4 \\ 1 & 2 & 4 \\ 0 & 1 & 1 \end{pmatrix} - \begin{pmatrix} 2 & 0 & 4 \\ 1 & 0 & 2 \\ 0 & 0 & 0 \end{pmatrix} \right\|_F = \left\| \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 2 \\ 0 & 1 & 1 \end{pmatrix} \right\|_F = \sqrt{10}$

Example

1. Break down X. (Don't need to know how.)

$$\begin{pmatrix}2&0&4\\1&2&4\\0&1&1\end{pmatrix}=\begin{pmatrix}-0.68&0.7\\-0.71&-0.55\\-0.18&0.45\end{pmatrix}\begin{pmatrix}6.31&0\\0&1.77\end{pmatrix}\begin{pmatrix}-0.33&-0.25&-0.91\\0.48&-0.88&0.07\end{pmatrix}$$

2. Retain the k highest singular values & vectors.

$$\begin{pmatrix}2&0&4\\1&2&4\\0&1&1\end{pmatrix}=\begin{pmatrix}-0.68&0.7\\-0.71&-0.55\\-0.18&0.45\end{pmatrix}\begin{pmatrix}6.31&0\\0&1.77\end{pmatrix}\begin{pmatrix}-0.33&-0.25&-0.91\\0.48&-0.88&0.07\end{pmatrix}$$

3. Recalculate X.

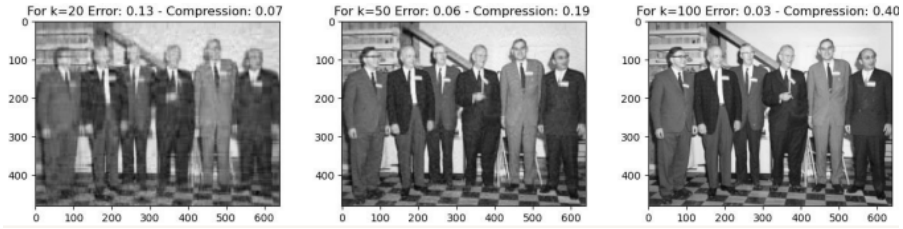
$$\begin{pmatrix}2&0&4\\1&2&4\\0&1&1\end{pmatrix}\approx\begin{pmatrix}-0.68\\-0.71\\-0.18\end{pmatrix}(6.31)(-0.33\quad-0.25\quad-0.91)=\begin{pmatrix}1.4&1.03&3.9\\1.48&1.12&4.08\\0.38&0.28&1.03\end{pmatrix}$$

4. Calculate the Frobenius norm.

$$\left\|\begin{pmatrix}2&0&4\\1&2&4\\0&1&1\end{pmatrix}-\begin{pmatrix}1.4&1.03&3.9\\1.48&1.12&4.08\\0.38&0.28&1.03\end{pmatrix}\right\|_F=1.78$$

Use Cases

Image Compression



	Alien	Inception	Matrix	Chocolat	La la Land	Amelie
User 1	5	0	4	0	1	0
User 2	5	4	5	1	0	0
User 3	4	5	0	0	1	0
User 4	0	4	4	0	1	0
User 5	0	0	1	5	4	0
User 6	1	0	0	4	5	0
User 7	0	1	0	4	0	5
User 8	0	0	0	0	4	5

Recommendation Systems

Given a matrix of ratings, removing singular components gives estimated ratings of films people haven't watched yet.

$$\underset{X}{\begin{pmatrix}5&0&4&0&1&0\\5&4&5&1&0&0\\4&5&0&0&0&0\\0&4&4&0&1&0\\0&0&1&5&4&0\\1&0&0&4&5&0\\0&1&0&4&0&5\\0&0&0&0&4&5\end{pmatrix}}\approx\underset{S}{\begin{pmatrix}-0.41&0.18\\-0.61&0.30\\-0.37&0.24\\-0.34&0.14\\-0.26&-0.48\\-0.27&-0.47\\-0.19&-0.40\\-0.16&-0.43\end{pmatrix}}\underset{V}{\begin{pmatrix}12.37&0.00\\0.00&10.06\end{pmatrix}}\underset{D^T}{\begin{pmatrix}-0.56&-0.47&-0.51&-0.30&-0.30&-0.14\\0.29&0.26&0.23&-0.56&-0.57&-0.41\end{pmatrix}}$$

$$\underset{X}{\begin{pmatrix}5&0&4&0&1&0\\5&4&5&1&0&0\\4&5&0&0&0&0\\0&4&4&0&1&0\\0&0&1&5&4&0\\1&0&0&4&5&0\\0&1&0&4&0&5\\0&0&0&0&4&5\end{pmatrix}}\approx\underset{\hat{X}}{\begin{pmatrix}3.37&2.89&3.04&0.56&0.54&-0.02\\5.07&4.35&4.56&0.59&0.56&-0.2\\3.25&2.8&2.91&0.04&0.02&-0.36\\2.76&2.36&2.49&0.53&0.52&0.03\\0.41&0.31&0.56&3.66&3.71&2.45\\0.45&0.34&0.6&3.64&3.69&2.43\\0.16&0.1&0.3&2.97&3.01&2.\\-0.18&-0.2&-0.01&2.99&3.03&2.06\end{pmatrix}}$$

(i = 8, j = 4)

They would be recommended Chocolat as it has the highest rating of films they haven't seen.

K-Means Clustering

Clustering aims to group close data points into a set of clusters. K-means uses **distance measurements** and **hard assignment**.

Hard Assignment: Assigns data points to one cluster

Soft Assignment: Assigns data points to multiple clusters.

Overview

- Separate data, $x_1, x_2, \ldots, x_n$ into k clusters
- Consider $C_1, \ldots C_k$, to be index sets holding the indices of the datapoints that belong to a cluster.
- K-means is use for **hard assignment** to a cluster.
- In other words, each data point is uniquely assigned to a single cluster based on proximity to the cluster centroids.

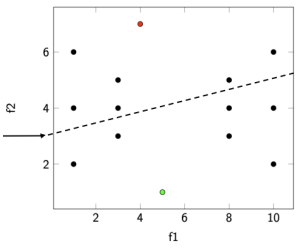
1. Initialise
   - select the number of clusters k
   - randomly choose k centroids from the data set.
2. Repeat the process till convergence
   - Step 1: assign clusters
     - each data point x_i moves into k clusters
     - calculate the distance between x_i and each centroids
     - assign x_i to the cluster with the closest centroid.
   - Step 2: Update Centroids
     - for each cluster, update its centroid by calculating the mean of all data points
   - Repeat Steps 1 and 2 until centroids stabilize or reach maximum iterations.

Cost Function

Minimises the **squared error distance between the data points**, minimising intra-cluster variance and increasing compactness.

$$\min_{c_1,\ldots c_k} \left\{ \sum_{k=1}^{K} \sum_{i \in C_k} \sum_{d=1}^{D} (x_d^k - \overline{x}_{kd})^2 \right\}$$

- K - Number of clusters
- k - a cluster
- C_k - kth index set
- i - Data point
- D - Number of data points
- $\overline{x}_{kd}$ - centroid of kth cluster
- x_d^k - data point belonging to cluster k



The Decision Boundary

The decision boundary for assigning each datapoint to a cluster is **all points equidistant for two or more clusters**.

Complexity

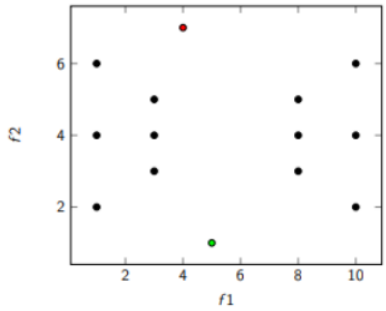
$\theta(T(K+1)ND)$ where :

- T: no. iterations.
- K: no. clusters.
- N: no. points.
- D: no. features.

The Centroids & The Process

Each cluster is characterised by the mean of the data points, the **centroid**.

1. Randomly assign the centroids.



Random mean [4,7] → ●

Random mean [5,1] → ●

2. Calculate the **Euclidean Distance** between each data point and each centroid.

$$d(p_1, p_2) = \sqrt{(c_1 - x_1)^2 + (c_2 - x_2)^2}$$

Data points		Distance to			
f_1	f_2	4	7	5	1
		RED		GREEN	
1	6		10		41
10	6		37		50
1	2		34		17
3	5		5		20
8	3		32		13
3	3		17		8

Example 1: $(4-1)^2+(7-6)^2 -> 3^2+1^2=10$

3. Assign each point to the closest centroid.

RED	GREEN	f1	f2
(10)	41	1	6
(37)	50	10	6
34	(17)	1	2
(5)	20	3	5
32	(13)	8	3
17	(8)	3	3
(20)	25	8	5
45	(34)	10	4
(18)	25	1	4
61	(26)	10	2
(10)	13	3	4
25	(18)	8	4

10 < 41 → YES [RED]

4. Calculate the mean for each cluster using all data points in it and update the centroid's value.

e.g. for the green cluster:

$f1$	$f2$
1	6
10	6
1	2
3	5
8	3
3	3
8	5
10	4
1	4
10	2
3	4
8	4

New mean of $f1$

$(1+8+3+10+10+8)/6 = 6.66$

New mean of $f2$

$(2+3+3+4+2+4)/6 = 3$

Note: f1 = x-values and f2 = y-values

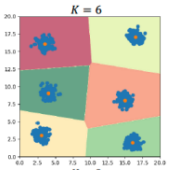
Repeat until the centroids stop moving or max iters. reached!

Advantages	***Disadvantages***
• Easy to **implement** and **understand**. • **Efficient** for large datasets. • Linear complexity with no. of points.	• Initial random k-values. • Bad K values lead to **non-convergence.** • Struggles to find clusters if not spherical. (Must be **convex**) • **Sensitive to outliers.**

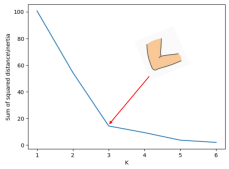
Random Initialisation Problems

Bad initialisation values **lead to convergence with incorrect clusters**. Solve by selecting the first centroid at random, calculating the squared difference between it and all points, setting the furthest away to the next centroid and repeating for until you have all centroids needed.

Evaluation Metrics



Select the optimal number of clusters by inspecting the decision boundaries after classification, or using evaluation metrics.



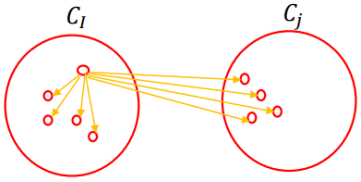
Elbow Scoring

Plot the cost function at convergence for different K values, select the one after which you get diminishing returns.

Silhouette Score

Gauges a data point's similarity to its cluster vs others. Getting the average gives a general idea of cohesion.

a_i is similarity to its cluster and b_i is some other. So we want $a_i > b_i$.



Hierarchical Clustering

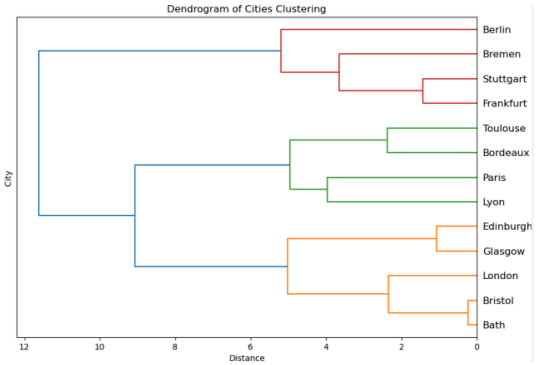
A **soft assignment** method.

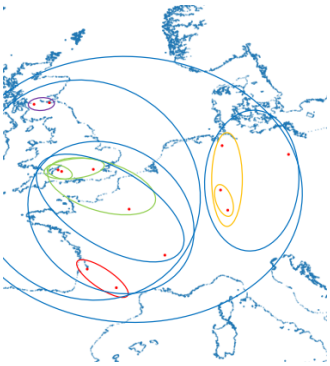
Agglomerative (Bottom-up)

1. Starts with N clusters, each with one data point.
2. **Finding Closest Clusters:** Merge the two closest clusters.
3. **Repeat** until all data points are in a single group.

Linkage Criteria: Determines which points in each cluster to measure the distance between.

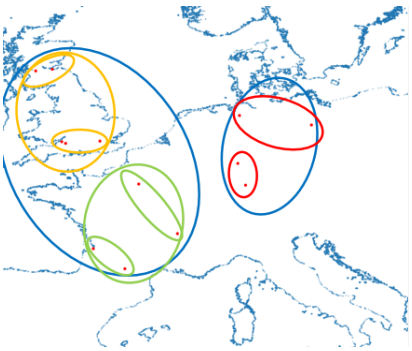
- *Single:* Closest members of the two clusters.
- *Complete:* Furthest members.
- *Average:* Mean distance.





See Example on the PPT!

Divisive (Top-Down)



1. Start with all data points in one cluster.
2. **Split the Cluster in Two**
    a. Identify the most remote datapoint from the mean and create a new cluster with it.
    b. Separate the two clusters by calculating the distance of the datapoints from the centres of each cluster, can do with k-means ($k = 2$).
3. **Repeat** until every cluster has one data point.

Agglomerative	***Divisive***
• **Complexity:** $O(N^2)$ to $O(N^3)$ • **More Flexible**: Linkage options. • **Easier to Implement.**	• **Complexity**: $O(N^3)$ but can be reduced with k-means. • **Difficult to Implement.**
Use Cases Detailed data, a few thousand samples. *e.g. clustering species*	*Use Cases* Clear cluster separation, millions of samples. *e.g. product categories*

DBSCAN

Density-based spatial clustering can cluster **non-convex sets,** unlike k-means, and **identify outliers**. It's **robust against noisy datasets** but needs **parameter tuning**.

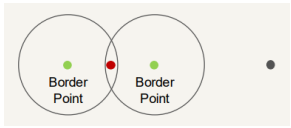
Parameters

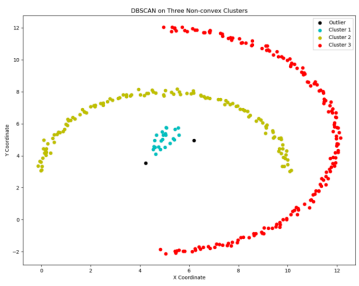
- ϵ - boundary around each point.
- N - density of points expected in the boundary.

Three Types of Data Points

- **Core:** Points with $\geq N$ neighbours within ϵ.
- **Border:** Points with $< N$ neighbours within ϵ, but one is a core point.
- **Noise:** Points with $< N$ neighbours within ϵ.

e.g. For $N = 3$ and some ϵ:



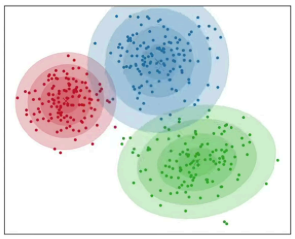


The Approach

Look at each data point's neighbours and classify them. A set of **neighbouring core points and their border points form a cluster**. Shared border points are assigned randomly.

Gaussian Mixture Models

A **generative probabilistic** model which assumes that a dataset is **a combination of several Gaussian distributions**. Each of these distributions is a **component/cluster**. Given a data point, it can give the probability that it came from the model, and the probability of it coming from a certain cluster in the model! It does **soft assignment**!



60% **red**
30 % **blue**
10% **green**

Given data points x, assume there's an **underlying discrete distribution** represented by a **latent** (missing) **variable** c, indicating which cluster generated a point. The probability of each cluster is the **prior**: $p(c|\theta) = \pi_c \; s.t. \; \sum_c \pi_c = 1$ and each cluster has parameters: μ_c and σ_c.

The probability of a point coming from the model is:

$$p(x|\theta) = \sum_c p(x, c|\theta) = \sum_c p(x|c, \theta)\, p(c|\theta) \quad \text{with parameters } \theta = \{\pi_1, \mu_1, \sigma_1, \; \ldots \; \pi_n, \mu_n, \sigma_n\} \text{ for } n \text{ Gaussians.}$$

Also assume that each cluster *c* can be modelled by a Gaussian distribution (**Noise model**):

$$p(x|c, \theta) = \mathcal{N}(x; \mu_c, \sigma_c)$$

Generating/Clustering Points

When creating a new point, the model randomly selects a prior biased on their weights, and samples the point from that cluster's Gaussian distribution.

Loss Function

Finding the best parameters θ which maximise $p(x|\theta)$, so the loss function is the joint probability of all data points given the parameters. Gives us the **log likelihood:**

$$L(\theta) = p(x^{(1)}, \ldots, x^{(N)} \mid \theta) = \prod_{n=1}^{N} p(x^n|\theta) = \sum_{n=1}^{N} log(p(x^{(n)} \mid \theta))$$

Logarithms make it **numerically stable.**

Optimiser: Expectation-Maximisation

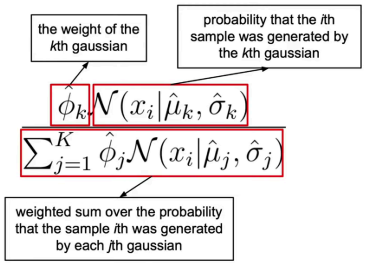
Expectation-maximisation **estimates model parameters** where there's missing or incomplete data, like the parameters of our gaussians, **maximising the log-likelihood**. Need to repeat the steps for each cluster/gaussian on each iteration until it converges or limit reached!

Expectation Step

For each datapoint, calculate the probability that it belongs to the cluster/gaussian.

$$p(c|x^{(n)}, \theta) = \frac{p(x^{(n)}|c, \theta)p(c|\theta)}{\sum_c p(x^{(n)}|c, \theta)p(c|\theta)}$$

Alternative notation w/ annotations:



e.g.

$$p(c = 1|x, \theta) = \frac{\frac{1}{\sqrt{2\pi}\sigma_1} exp\left(-\frac{1}{\sigma_1^2}(x^{(n)} - \mu_1)^2\right)\pi_1}{\frac{1}{\sqrt{2\pi}\sigma_1} exp\left(-\frac{1}{\sigma_1^2}(x^{(n)} - \mu_1)^2\right)\pi_1 + \frac{1}{\sqrt{2\pi}\sigma_2} exp\left(-\frac{1}{\sigma_2^2}(x^{(n)} - \mu_2)^2\right)\pi_2}$$

Maximisation Step

Use these to recalculate the optimal parameters:

These formulae are for 1D, they get more complex when using multivariate Gaussians.

$$\mu_c = \frac{\sum_{n=1}^{N} p(c|x^{(n)}, \theta) x^{(n)}}{\sum_{n=1}^{N} p(c|x^{(n)}, \theta)} \qquad \sigma_c = \sqrt{\frac{\sum_{n=1}^{N} p(c|x^{(n)}, \theta)(x^{(n)} - \mu_c)^2}{\sum_{n=1}^{N} p(c|x^{(n)}, \theta)}} \qquad \pi_c = \frac{\sum_{n=1}^{N} p(c|x^{(n)}, \theta)}{\sum_{n=1}^{N} \sum_{k=1}^{K} p(\mathrm{c} = \mathrm{k}|x^{(n)}, \theta)}$$

Connection to K-Means

K-means is a GMM which assumes the same probability for all clusters and the Gaussians are spherical.

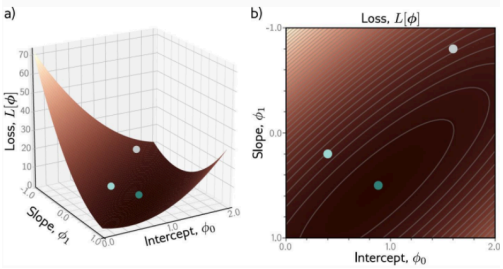
The Evidence Lower Bound

Log summation in the log likelihood is computationally expensive. ELBO is a lower bound which is easier to calculate.

$$L(\theta) \geq \mathcal{F}(q, \theta) \qquad \sum_{n=1}^{N} log(p(x^{(n)}|\theta)) \geq \sum_{n=1}^{N} \sum_{n=1}^{K} q(c) \log(p(x, c|\theta)) - \sum_{n=1}^{N} \sum_{n=1}^{K} q(c) \log(q(c))$$

Optimisation

Gradient descent minimises the loss function by calculating the derivative with respect to each parameter and adjusting them downwards in each epoch. Plotting the loss function output with different parameters looks like:



Convex VS Non-Convex Loss Functions

Convex: Any two points can be connected with a line on or above the curve. Gradient descent **always finds the global minima**. **Linear models** have convex loss functions.

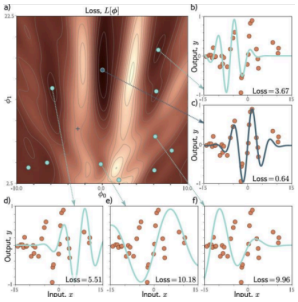
Non-convex: Has peaks and valleys, causing **optimisation challenges** with gradient descent **getting stuck in local minima**. **Non-linear models** have non-convex noss functions.

A 2D optimisation curve is convex if the 2nd derivative of the loss function is positive. In higher dimensions, it's if the Hessian matrix is positive everywhere.

Example: Göbel Model

Another type of model for **non-convex relationships** using two parameters, making it good for visualising the non-convex optimisation problem.

$$\mathrm{f}[x, \boldsymbol{\phi}] = \sin[\phi_0 + 0.06 \cdot \phi_1 x] \cdot \exp\left(-\frac{(\phi_0 + 0.06 \cdot \phi_1 x)^2}{8.0}\right)$$



Each dataset has its own map of the loss function over changing parameters. Normal **gradient descent only finds the global minimum if it starts in the right valley**.

Stochastic Gradient Descent

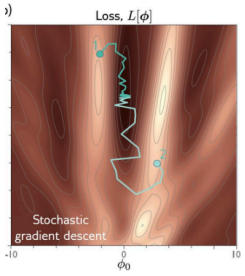
Computes the gradient on a subset (**batch**) of the data points, rather all of them, without replacement. Introduces **randomness during descent to avoid getting stuck in local minima**. Less computationally expensive, uses all data equally and finds better solutions!

Before (full batch descent)

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \sum_{i=1}^{I} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}},$$

After (SGD)

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}},$$



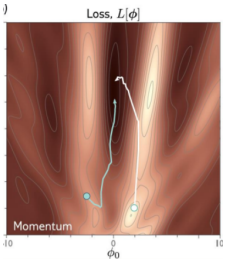
Momentum

Adding a weighted sum of the current gradient and direction moved in the previous steps to make steps smoother. Each gradient step is a weighted sum of the previous gradients.

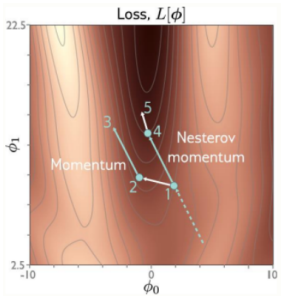
$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}$$

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_{t+1}$$

- Can lead to oscillations.
- Converges faster.
- β controls the degree of smoothing.



Nesterov Accelerated Momentum



Improves normal momentum by **moving in the predicted direction first / applying momentum first** and **calculating the gradient of the point ahead** of the current position for **increased accuracy and smoothing**.

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_t]}{\partial \boldsymbol{\phi}}$$

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_{t+1}$$

Adaptive Movement Estimation (ADAM)

Optimises models using **adaptive learning rates** and **normalised gradients** for each parameter for **more efficient**, **smoother updates** and **faster optimisation**. Different parameters need different step sizes or you get **overshooting**, **stuck** or a large number of iterations.

Normalising Gradients

Ensures **gradients are on a similar scale so contribute equally to parameter updates**.

The Mean Gradients

Calculates the gradient of the loss function with respect to the parameters at time step *t* over time.

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}$$

The Pointwise Squared Gradient

$$\mathbf{v}_{t+1} \leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2$$

At the start, there's no history of momentum and pointwise squared gradients so we apply a **bias correction**:

$$\tilde{\mathbf{m}}_{t+1} \leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}} \qquad \tilde{\mathbf{v}}_{t+1} \leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}}$$

The Update Rule

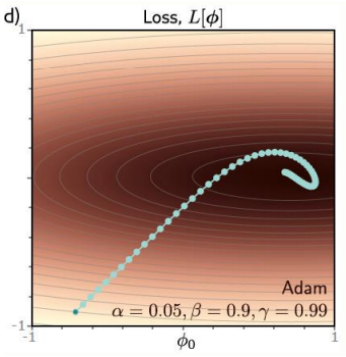
$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}$$

- α - learning rate.
- ϵ - small constant preventing division by zero.

The Algorithm

At each epoch:

1. Computes the mean and pointwise squared gradients with momentum.
2. Apply bias correction at the start.
3. Update the parameters.



Shallow Neural Networks

Shallow neural networks have **three layers** and enable many inputs and outputs. They're **flexible** enough to describe complex input/output mappings.

$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11} x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21} x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31} x] \end{aligned}$$

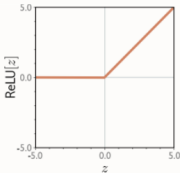
- They're made of activation functions a which take on the equation of a line.
- The parameters determine the particular function. In this example, there are ten: $\phi = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \ldots\}$

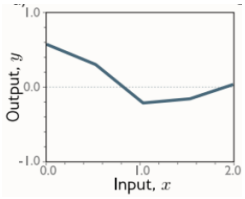
Activation Functions

There are many different types. Rectified linear unit (ReLU) is the most popular.

$$\mathrm{a}[z] = \mathrm{ReLU}[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}.$$

Rectified Linear Unit
(particular kind of activation function)





They enable **piecewise linear functions**. So, three activation functions will have three joints.

They're expressed as **hidden units/nodes** for simplification. Hidden units are **inactive if they're zero and active if they're non-zero**.

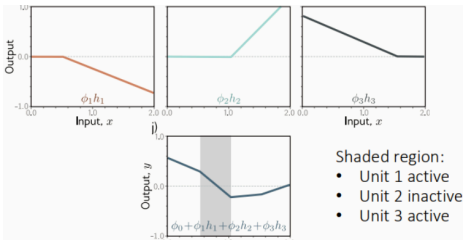
Break down into two parts:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

where:

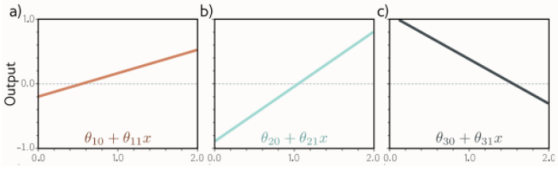
Hidden units

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11} x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21} x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31} x] \end{aligned}$$

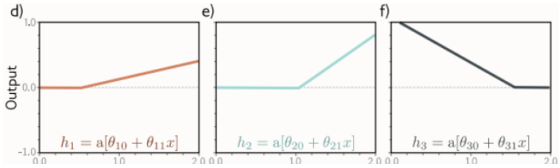


Computing Hidden Units

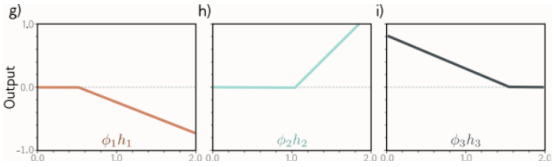
1. Compute the linear functions.



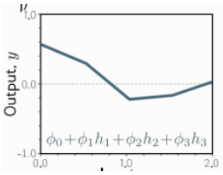
2. Weight them based on the activation function.



3. Multiply them by the scalars ϕ.



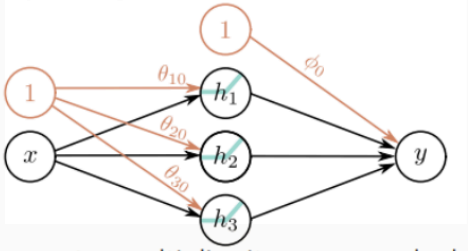
4. Sum the resulting values.



Representing as Conventional Neural Networks

This equation is a neural network! Connections always add a constant (the **bias**), so the orange lines aren't usually illustrated.

$$h_1 = a[\theta_{10} + \theta_{11}x]$$
$$h_2 = a[\theta_{20} + \theta_{21}x] \qquad y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$
$$h_3 = a[\theta_{30} + \theta_{31}x]$$



Each parameter multiplies its source and adds to its target

The Universal Approximation Theorem

A shallow network with enough hidden units can describe any continuous function on a compact subset of R^D to arbitrary precision. Basically, **any relationship can be described with enough hidden units** but a deep network might be more computationally efficient.

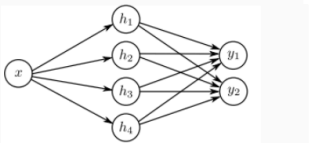
Multiple Outputs

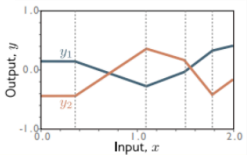
Two outputs means two equations that **share hidden units** so have the same boundaries.

1 input, 4 hidden units, 2 outputs

$$h_1 = a[\theta_{10} + \theta_{11}x]$$
$$h_2 = a[\theta_{20} + \theta_{21}x]$$
$$h_3 = a[\theta_{30} + \theta_{31}x]$$
$$h_4 = a[\theta_{40} + \theta_{41}x]$$

$$y_1 = \phi_{10} + \phi_{11}h_1 + \phi_{12}h_2 + \phi_{13}h_3 + \phi_{14}h_4$$
$$y_2 = \phi_{20} + \phi_{21}h_1 + \phi_{22}h_2 + \phi_{23}h_3 + \phi_{24}h_4$$





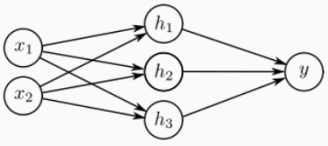
Multiple Inputs

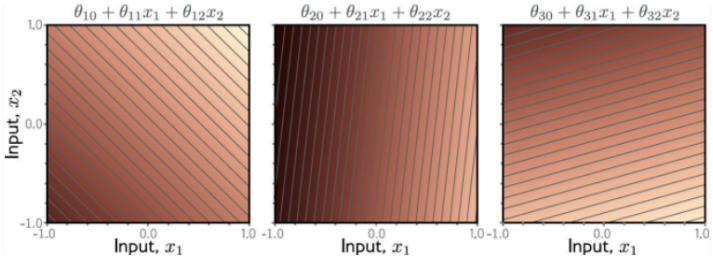
Two inputs means two inputs (x_1, x_2) in each node. The output can be viewed graphically. Higher outputs are brighter and lower outputs are darker.

2 inputs, 3 hidden units, 1 output

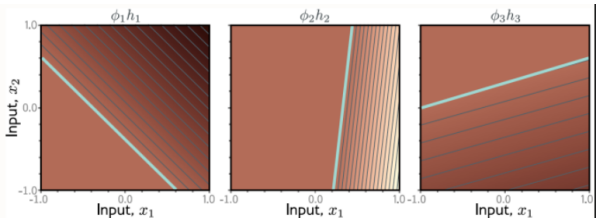
$$h_1 = a[\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2]$$
$$h_2 = a[\theta_{20} + \theta_{21}x_1 + \theta_{22}x_2]$$
$$h_3 = a[\theta_{30} + \theta_{31}x_1 + \theta_{32}x_2]$$

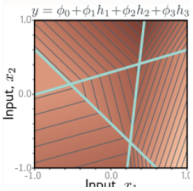
$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$





Putting them through Relu: the area without lines is where it is set to 0 because it was negative. Then scaling them: the gradients change. *e.g. the leftmost image is scaled by a negative value*.





Summing them all creates a shape made of **convex polygons**.

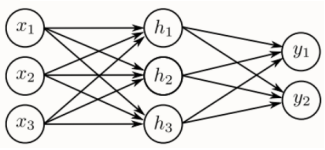
Multiple Inputs and Outputs

Two inputs and multiple outputs adds more polygon outputs, it doesn't change the hidden units or boundaries. Increasing the inputs makes it difficult to visualise.

An arbitrary number of inputs, hidden units, and outputs, can be expressed as:

D_o Outputs, D hidden units, and D_i inputs

$$h_d = \mathrm{a}\left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i\right] \qquad y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_d$$



Identifying Parameters

1. Count the weights: *3x3 + 3x2 = 9 + 6 (the number of lines)*
2. Count the biases: 3 + 2 *(one for each node outside the input layer)*

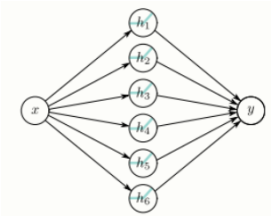
Number of Output Regions

Each output consists of D_j dimensional **convex polytopes**. The number of hidden units/parameters to the number of regions scales very fast!

Num. of Planes = Num of Hidden Units/Nodes

The number of regions created by D planes in D_i dimensions, where $D > D_i$ is:

$$2^{D_i} < \sum_{j=0}^{D_i} \binom{D}{j} < 2^D$$



Identifying the Number of Polytopes

Each hidden unit adds one region, and there's already one with ReLU (where it's zero because of negative values). So, in this example there are 7.

Networks for Classification

For classification, the output layers will have activation functions! **Softmax with multiple output nodes for multi-class classification**, **sigmoid with one for binary classification** or **sigmoid with multiple for multi-label classification**.

Deep Neural Networks

Form more complex functions by employing **more than one hidden layer**, so can **model more complex relationships easier**. They have hyperparameters: **network depth** (K) and **width** (D_k).

Composing Shallow Networks

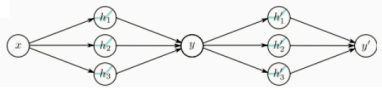
Feeding the output of the first network to the input of the second. You can view the overall output as sections, where for each section in the first network ranges from -1 to 1 so repeats some version of the second network.

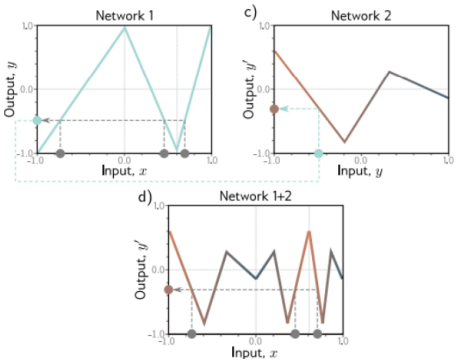
Network 1:

$$h_1 = a[\theta_{10} + \theta_{11}x]$$
$$h_2 = a[\theta_{20} + \theta_{21}x]$$
$$h_3 = a[\theta_{30} + \theta_{31}x]$$
$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

Network 2:

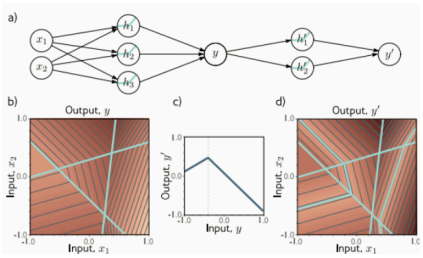
$$h'_1 = a[\theta'_{10} + \theta'_{11}y]$$
$$h'_2 = a[\theta'_{20} + \theta'_{21}y]$$
$$h'_3 = a[\theta'_{30} + \theta'_{31}y]$$
$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3$$





2D Networks

Combining a 2D network and a 1D network, the 1D network has two segments and so doubles the number of complex polytopes.



Fully Combining Two Networks

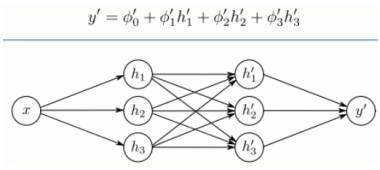
Express the second set of hidden units in terms of the first.

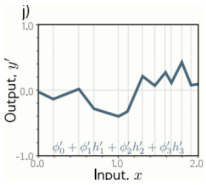
$$\begin{aligned}
h'_1 &= a[\theta'_{10} + \theta'_{11}y] &&= a[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \\
h'_2 &= a[\theta'_{20} + \theta'_{21}y] &&= a[\theta'_{20} + \theta'_{21}\phi_0 + \theta'_{21}\phi_1 h_1 + \theta'_{21}\phi_2 h_2 + \theta'_{21}\phi_3 h_3] \\
h'_3 &= a[\theta'_{30} + \theta'_{31}y] &&= a[\theta'_{30} + \theta'_{31}\phi_0 + \theta'_{31}\phi_1 h_1 + \theta'_{31}\phi_2 h_2 + \theta'_{31}\phi_3 h_3]
\end{aligned}$$

Then rewrite them for simplicity. The coefficients of the second layer's hidden nodes are **weightings of the first**. Applying Relu, weightings and summing the second layer gives:

$$\begin{aligned}
h'_1 &= a[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
h'_2 &= a[\psi_{20} + \psi_{21}h_2 + \psi_{22}h_2 + \psi_{23}h_3] \\
h'_3 &= a[\psi_{30} + \psi_{31}h_2 + \psi_{32}h_2 + \psi_{33}h_3]
\end{aligned}$$

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3$$





Mathematical Notation

$$\begin{aligned}
&h_1 = a[\theta_{10} + \theta_{11}x] \\
&h_2 = a[\theta_{20} + \theta_{21}x] \\
&h_3 = a[\theta_{30} + \theta_{31}x]
\end{aligned}
\longrightarrow
\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \theta_{10} \\ \theta_{20} \\ \theta_{30} \end{bmatrix} + \begin{bmatrix} \theta_{11} \\ \theta_{21} \\ \theta_{31} \end{bmatrix} x\right]
\longrightarrow
\mathbf{h} = \mathbf{a}\left[\boldsymbol{\theta}_0 + \boldsymbol{\theta} x\right]
\longrightarrow
\mathbf{h}_1 = \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}]$$

$$\begin{aligned}
&h'_1 = a[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
&h'_2 = a[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\
&h'_3 = a[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3]
\end{aligned}
\longrightarrow
\begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \psi_{10} \\ \psi_{20} \\ \psi_{30} \end{bmatrix} + \begin{bmatrix} \psi_{11} & \psi_{12} & \psi_{13} \\ \psi_{21} & \psi_{22} & \psi_{23} \\ \psi_{32} & \psi_{32} & \psi_{33} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}\right]
\longrightarrow
\mathbf{h}' = \mathbf{a}\left[\boldsymbol{\psi}_0 + \boldsymbol{\Psi}\mathbf{h}\right]
\longrightarrow
\mathbf{h}_2 = \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1]$$

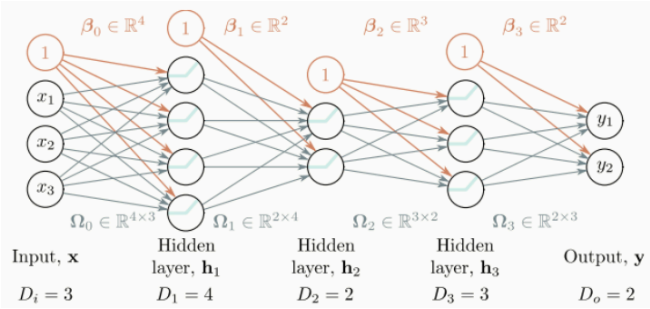
$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3
\longrightarrow
y' = \phi'_0 + \begin{bmatrix} \phi'_1 & \phi'_2 & \phi'_3 \end{bmatrix} \begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix}
\longrightarrow
y = \phi'_0 + \boldsymbol{\phi}'\mathbf{h}'
\longrightarrow
\mathbf{y} = \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2$$

The most important thing to remember is that **every hidden layer has a bias vector** β and **weight matrix** Ω**.** Which are weightings of the parameters from prior layers.

General Equation for a Deep Network

$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathbf{a}\left[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{a}\left[\ldots \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{a}\left[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{a}\left[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}\right]\right]\ldots\right]\right]$$

Example



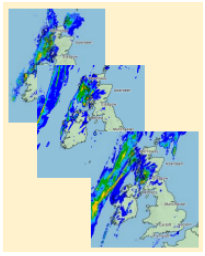
Shallow VS Deep Networks

- **Function Approximation:** Both obey the Universal Approximation Theorem.
- **Number of Linear Regions Per Parameter:** Deep networks create more regions per parameter, with dependencies between them.
- **Depth Efficiency:** Deep networks can approximate some functions with fewer hidden units than shallow networks.
- **Lots of Inputs:** Fully connected networks have a large number of parameters so aren't practical.
- **Fitting & Generalisation:** Fitting deep networks with less than 20 layers is easy but past that requires tricks.

Neural Network Optimisation

See physical notes on forward & backpropagation.

Recurrent Neural Networks



What if we want to produce an output based on **related timesteps**? With a normal neural network, you'd have to feed all of them at once because they're related, **requiring a massive number of parameters**. Or, feed them independently, but the **relationships** between timesteps **won't be modelled**!

Recurrent Units

Hold a **hidden state** (memory) of the previous input state, allowing the model to **select information from timesteps** and reduce developer assumptions. They're **fully connected**.

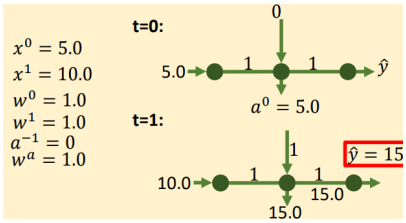
Each hidden unit **applies an activation function**, and **passes the activation value to the same recurrent unit** on the next iteration with the next timestep. We visualise this as a series of layers.

More Flexible: The number of timesteps used to train the model can be changed.

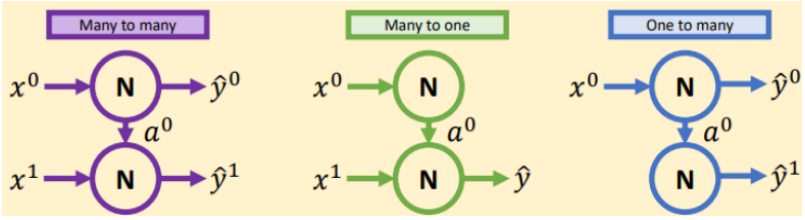
Less Parameters: Parameters are shared between recurrent units.

Calculating Values

Calculate the **output** and **activation passed on by each timestep** using the weights and inputs. Here we assume the biases are 0 (if not they'd be added to the 2nd and 3rd nodes) and a ReLU activation function.



Variants



Many to many

Tasks where the inputs and output sequences need to remain aligned.

Many to one

Tasks where a sequence is required to make a prediction.

One to many

Tasks where a single input should produce a sequence.

Limitations

Inference is slow

As each time step must be computed sequentially there is limited opportunity for parallelisation.

No look ahead

There is no opportunity for the network to look at the next time step or what's coming up in the time series.

Long-term memory

The network will struggle to access data from many time-steps previously.

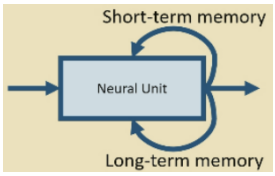
The Vanishing/Exploding Gradient Problem

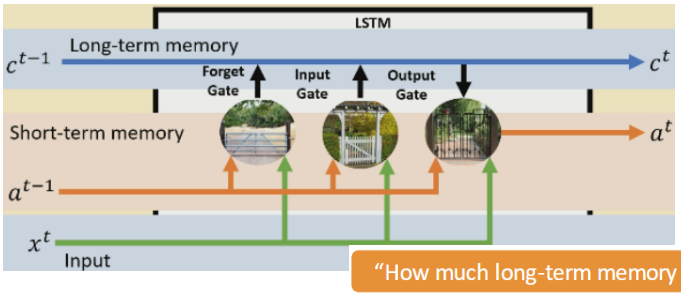
The connecting weights can make the activation values really big or small. **The more timesteps, the more sensitive it is to the memory weight!** In backpropagation, this leads to the gradient step being large (exploding) or really small (vanishing), leading the network to **ignore or overweight previous timesteps**! Can be generalised by the equation: $x^0 \times (w^a)^t = \hat{y}$

Long Short-Term Memory (LSTM) Networks

Fully connected!

RNNs only work well with short-term memory, great when there's a long period between timesteps. **LSTMs have short & long-term memory** and **mitigate the vanishing/exploding gradient problem** using $tanh(x)$ and $sigmoid$ to regulate gradients and the multiple propagation routes to preserve the error signals back through time.





The Three Gates

LSTMs have three gates: forget, input and output. The **short-term memory is the output**!

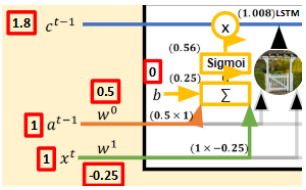
- t - current timestep.
- x^t - input scalar/vector.

"How much long-term memory should we remember?"

The Forget Gate

A neuron with a **sigmoid activation function** because it acts like a percentage. Computes the dot product of the short-term and input vectors/scalars with their weights, sums them w/ bias and calculates the sigmoid.

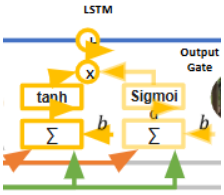
$$f = \sigma(W_h h + W_x x)$$



The Input Gate

"How should we update the long-term memory?"

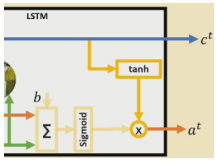
Uses a $tanh(x)$ **activation** to work out the potential new long-term memory because it can produce negative values, allowing it to remove some, and a **sigmoid activation** to work out how much to remember.



The Output Gate

"How should we update our short-term memory or output?"

Uses $tanh(x)$, connected to the long-term, to work out the potential short-term memory and sigmoid to get how much should be remembered.



Accepting Multiple Features

RNNs and LSTMs can accept multiple features using vectors, and the units can be organised in several ways.

Mid-layer allows feature extraction first, sequential allows them to depend on each other and stacking enables each unit to remember something different.

Convolutional Neural Networks (CNNs)

Deep learning classification models, which **aren't fully connected** because not every pixel in the input impacts the pixels in each convolution. Output values share kernel weights so are **faster to optimise**. More layers & filters = captures more complex features.

Convolutional Layers

Perform convolutions on the input, using randomly predefined kernels/filters, to identify features. Filters have a few hyperparameters:

Size

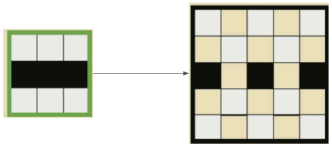
Stride: Steps between filter applications. Reduces noise, output size and time but increases data loss.

Padding

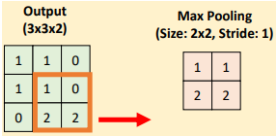
Dilation: Space between kernel filters. Increases receptive field without more parameters.

Dilation Example

Dilating by 1:



See PPT for walkthroughs.



Pooling Layers

Increase network efficiency by downsampling information, retaining the most important values.

Fully-connected Layers *a.k.a. dense layers*

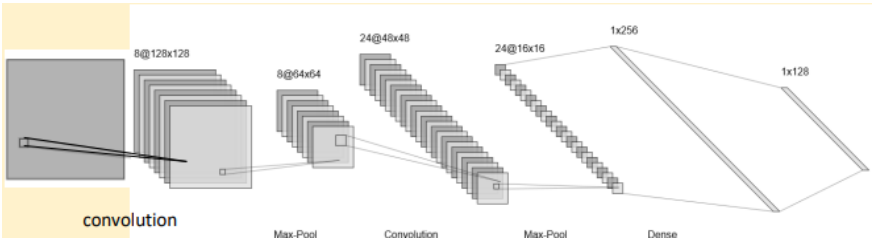
The final layer which takes features and outputs the intended class/regressive value.

Applications

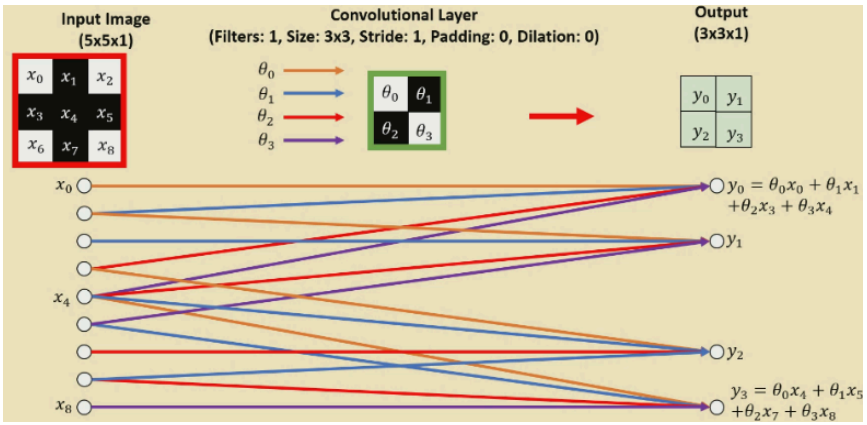
1D Kernels: Timeseries data.

2D Kernels: Images.

3D Kernels: Videos



Mathematical Interpretation

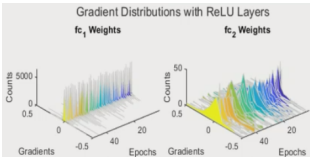


Identifying the Gradient Problem

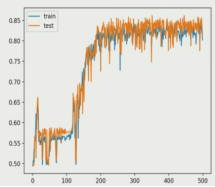
The **more layers** a model has, the **higher the chance of the exploding/vanishing gradient problem** occurring in back propagation! There are three signifiers:

Abnormal Gradient Distributions

Plot the changes in gradients over epochs to check they're only adjusting slightly, rather than loads or none at all.



The gradient of ReLU is 0 or 1, and sigmoid is between 0 and 1, so **sigmoid is more likely to lead to vanishing gradients!**



Chaotic Learning Curves

Plot the **train-test learning curve**, the performance on training and testing data over epochs. **High-variability** between them indicates the gradient problem.

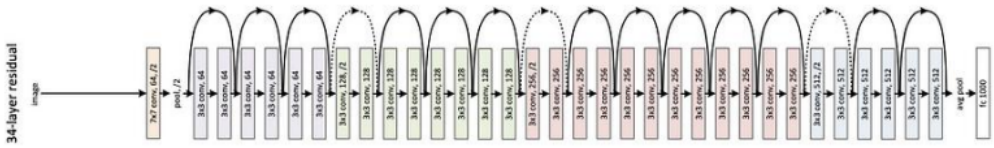
Irregular Outputs

Outputs zero-ing out (*i.e. producing a black image is better than trying to produce something*) or producing disproportionately high outputs.

Residual Neural Networks/Blocks *a.k.a. the ResNets*

LSTM's long-term memory lets them remember information from previous layers. Residual blocks replicate this in CNNs by allowing information to be passed around layers without modification, ignoring them.

Can ignore layers that cause the gradient problem!



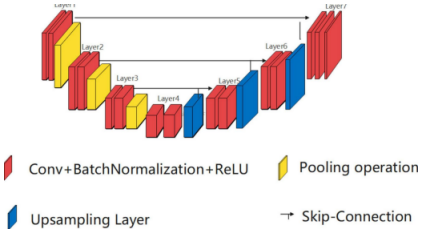
Different residual block variants exist depending on the number of layers and kernel sizes.

Disadvantages

Computational Cost	Lack of interpretability	Identity Shortcuts
Additional parameters introduced through residual connections increase cost of optimisation.	Skip connections introduce difficulties in attempting to explain the decision making.	Some blocks may become redundant due to the skip connections.

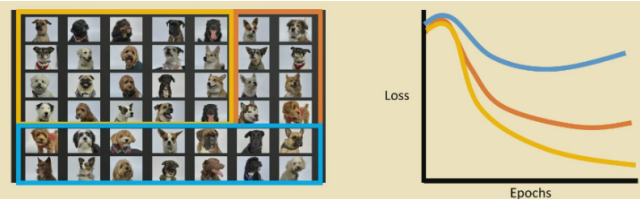
U-Nets

Use skip connections, different to residual connections.



Preventing Overfitting with Regularisation

Generally, we end up overfitting and getting a performance like:

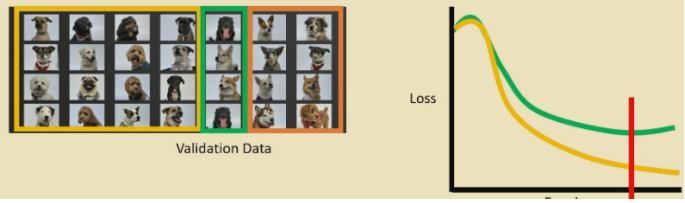


- **Yellow:** Training data.
- **Orange:** Testing data.
- **Blue:** Outside data.

Implicit Regularisation

Holdout-early Stopping

Prevents overfitting using a **validation set which assesses the model's performance** after each epoch and **stops training** when its loss starts to increase.



If this is a local minimum, the global minimum can be found using **checkpoints** to store the best parameters so far.

Data Augmentation

Expands the training dataset to include more examples, but risks overfitting and methods should depend on the context.

Images

- Flip/rotate.
- Noise.
- Colour changes.

Audio

- Noise.
- Shifting audio clips.
- Speed & pitch variation.

Text

- Sentence suffling.
- Word replacement.
- Random word insertion & deletion.

Dropout Layers

Neural networks can become over-reliant on particular features. Adding a **dropout rate** (normally between 0.2 and 0.7) mitigates this by introducing a chance that a feature isn't trained in an iteration.

Must be deactivated for testing & deployment.	Increased computational time.	Inconsistency and replicability.

Explicit Regularisation

Constraints a model's weights, ensuring they're consistent and around zero, by **modifying the loss function**.

L1 Regularisation (Lasso)

Adds the absolute value of the weights.

$$\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2+\lambda\sum_{i=1}^{m}|w_i|$$

Choice of λ should depend on the bias-variance tradeoff.

L2 Regularisation (Ridge)

Adds a penalty proportional to the square of the coefficients.

$$\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2+\lambda\sum_{i=1}^{m}w_i^2$$

Combination/Elastic

Uses both with a mixing parameter to give them a total weight of 1.

$$\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2+\lambda Z$$

$$Z=\alpha\sum_{i=1}^{m}|w_i|+(\alpha-1)\sum_{i=1}^{m}w_i^2$$

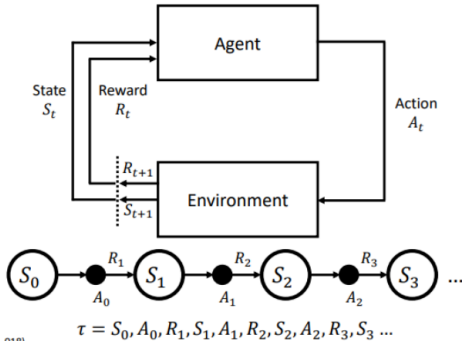
Reinforcement Learning

The agent uses its experience to learn how to solve **sequential decision problems** - how to **map states to actions** to **maximise total reward in the long run**.

Not supervised or unsupervised!

Applications

It's an **abstract** and **flexible** framework that can be applied to many different problems in different ways! *e.g. controlling nuclear reactors and autonomous driving.*



The Agent-Environment Interaction

Agents build up a **trajectory**, the sequence of states, actions and rewards, as they interact with their environment.

They learn a **policy** $\pi_t(s, a)$ which gives the probability of selecting action a in state s at time-step t, telling it which action to take.

They aim to maximise the **total discounted return**, the sum of future rewards. γ is a discount factor.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$
$$= \Sigma_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

RL algorithms use experience generated by the agent to **modify the policy** to maximise the discounted return.

Value Functions

State-Value: The return in a state if the agent follows its policy.

Action-Value: The return in a state if the agent takes an action and then follows its policy.

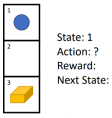
In DeepRL, neural networks approximate value functions.

Q-Learning

Learns the **action-value function** for the **optimal policy**, the discounted return it can expect if it takes an action and follows the optimal policy after. It stores these values in an **action-value table**!

Example - One Q-Value Reassignment

Step 1



State: 1
Action: ?
Reward:
Next State:

Actions: ↑ Up ↓ Down

Rewards:
- +10 for reaching gold
- -1 otherwise

Action-Value Table Q

State-Action Pair	Q-Value
1, ↑	0
1, ↓	−1
2, ↑	0
2, ↓	10
3, ↑	0
3, ↓	0

Explore Randomly or **Act Greedily**

Algorithm chooses to explore randomly or act greedily using the table.

Step 2

State: 1
Action: Down
Reward: -1
Next State: 2

Actions: ↑ Up ↓ Down

Rewards:
- +10 for reaching gold
- -1 otherwise

Action-Value Table Q

State-Action Pair	Q-Value
1, ↑	0
1, ↓	−1
2, ↑	0
2, ↓	10
3, ↑	0
3, ↓	0

$$Q(1,\downarrow) \leftarrow 0 \times \underbrace{Q(1,\downarrow)}_{\text{Old Estimate}} + 1 \times \underbrace{(-1 + 1 \times \max\{Q(2,\uparrow), Q(2,\downarrow)\})}_{\text{New Estimate}}$$

$\alpha = 1.0, \gamma = 1.0$

Uses a formula which takes the max q-value from the resulting state and the old estimate.

Step 3

State: 1
Action: Down
Reward: -1
Next State: 2

Actions: ↑ Up ↓ Down

Rewards:
- +10 for reaching gold
- -1 otherwise

Action-Value Table Q

State-Action Pair	Q-Value
1, ↑	0
1, ↓	9
2, ↑	0
2, ↓	10
3, ↑	0
3, ↓	0

$$Q(1,\downarrow) \leftarrow 0 \times -1 + 1 \times (-1 + 10) = 9$$

Updates the q-value for the starting state.