

7CCSMPNN

Pattern Recognition, Neural Networks, and Deep Learning

If you spot anything wrong, please let me know!

discord: @insertish

izzy · insrt.uk

Contents

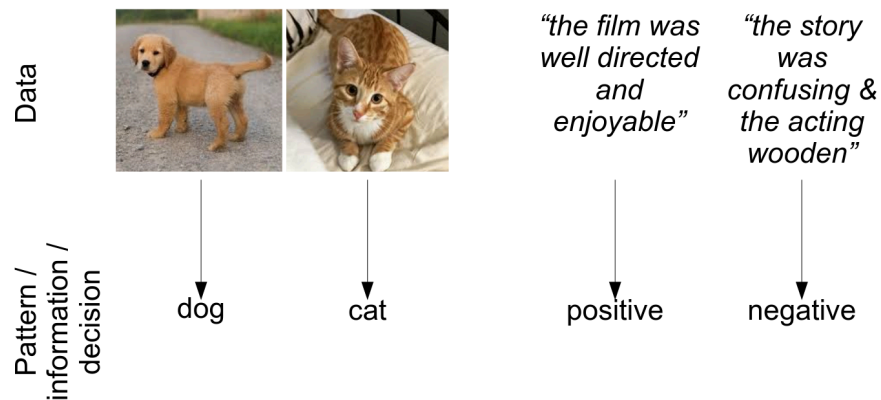
1. Introduction to Pattern Recognition	4
1.1. The Classifier Design Cycle	5
1.1.1. Data Collection (Input Data)	5
1.1.2. Feature Selection (Feature Vectors)	6
1.1.3. Models, Tuning, & Training	7
1.1.4. Evaluation	8
2. Discriminant Functions	9
2.1. Dichotomizers	10
2.2. Linear Discriminant Functions	11
2.2.1. Decision Boundaries & Hyperplanes	12
2.2.2. Generalised Linear Discriminant Functions	13
2.3. Learning Decision Boundaries (Perceptron Learning)	14
2.3.1. Gradient Descent Procedures	15
2.3.1.1. Batch Perceptron Learning Algorithm	16
2.3.1.2. Sequential Perceptron Learning Algorithm	16
2.3.1.3. Multiclass Perceptron Learning Algorithm	17
2.4. Learning Decision Boundaries (Minimum Squared Error procedures)	18
2.4.1. Widrow-Hoff (LMS) Learning Algorithm	19
2.5. k-Nearest Neighbour Classifier	20
3. Introduction to Neural Networks	21
3.1. Biological Neural Networks	21
3.2. Artificial Neural Networks	24
3.2.1. Linear Threshold Units (Perceptrons)	26
3.2.1.1. Equivalence to Linear Discriminant Functions	27
3.2.1.2. Delta Learning Algorithm	28
3.2.2. Hebbian Learning Rule	29
3.2.3. Competitive Learning Networks	30
3.2.4. Negative Feedback Networks	31
3.2.5. Autoencoder Networks	32
4. Multi-layer Perceptrons and Backpropagation	34
4.1. Feedforward Neural Networks (MLPs)	34
4.1.1. Neuron Model	34
4.1.2. Activation/Transfer Functions	35
4.1.3. General Feed-Forward Operation	36
4.1.4. Network Topology	36

4.2.	Backpropagation Algorithm	37
4.2.1.	Network Learning	37
4.2.2.	Hidden-to-Output Weights	38
4.2.3.	Input-to-Hidden Weights	39
4.2.4.	Pseudo-Code	40
4.2.5.	Learning Curves	41
4.3.	Radial Basis Function Neural Networks	42
4.3.1.	RBF Network Learning	44
4.3.1.1.	Determination of centres	45
4.3.1.2.	Determination of output weights	46
4.3.2.	Comparison of RBF and MLP Networks	47
5.	Deep Discriminative Neural Networks	48
5.1.	Difficulties Training Deep Neural Networks	50
5.1.1.	Solutions	51
5.1.1.1.	Activation functions with non-vanishing derivatives	51
5.1.1.2.	Better weight initialisation	51
5.1.1.3.	Adaptive variations on standard backpropagation	52
5.1.1.4.	Batch normalisation	54
5.1.1.5.	Skip connections	55
5.2.	Convolutional Neural Networks (CNNs)	56
5.2.1.	Convolutional Layers	56
5.2.2.	Pooling Layers	60
5.2.3.	Fully-Connected Layers	61
5.3.	Issues with CNNs/Deep Networks	62
6.	Deep Generative Neural Networks	64
6.1.	Generative Model Taxonomy	66
6.1.1.	Explicit Model with Trackable Density: PixelRNN	67
6.1.2.	Explicit Model with Approx. Density: Variational Autoencoder (VAE)	68
6.2.	Generative Adversial Networks (GANs)	69
6.3.	Problems with GANs	72
6.4.	Improving training of GANs	73
6.4.1.	Network Design	73
6.4.2.	Vector arithmetic with latent data	74
6.4.3.	Cost functions	74
6.4.4.	Optimisation	75
7.	Feature Extraction	77
7.1.	Feature Engineering, Selection, & Extraction	77
7.1.1.	Principal Component Analysis (PCA)	79
7.1.1.1.	Karhunen-Loève Transform (KLT)	80
7.1.1.2.	Neural Networks for PCA	80
7.1.1.3.	Whitening	82
7.1.2.	Linear Discriminant Analysis (LDA)	82
7.1.3.	Independent Component Analysis (ICA)	83
7.1.4.	Random Projections	84
7.1.5.	Sparse Coding	85
8.	Support Vector Machines	87
8.1.	Linear SVMs: Linearly Seperable Case	88
8.2.	Linear SVMs: Non-separable Case	91
8.3.	Nonlinear SVMs	94

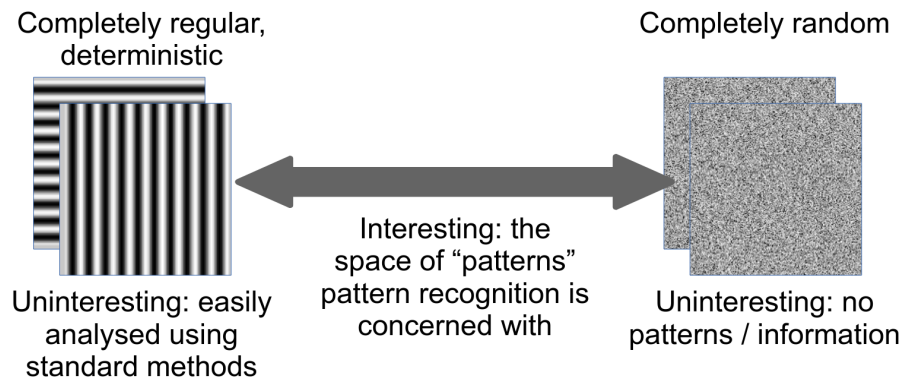
8.3.1. Kernel Mappings	95
8.4. Multi-class SVMs	97
8.4.1. One-against-one approach	97
8.4.2. One-against-all approach	97
8.4.3. Binary decision tree	98
8.4.4. Binary coded approach	98
9. Ensemble Methods	99
9.1. Bagging	99
9.2. Boosting	100
9.2.1. AdaBoost (Adaptive Boost)	101
9.2.2. Stacked Generalisation	103
9.2.3. Structure of ensemble classifiers	104
9.2.4. Decision Trees	105
9.2.4.1. Bagging Decision Trees	106
9.2.4.2. Random Forests	107
10. Clustering	108
10.1. k-Means Clustering	108
10.2. Fuzzy k-Means Clustering	109
10.3. Iterative Optimisation	110
10.4. Hierarchical Clustering	111
10.5. Competitive Learning	112
10.6. Clustering for Unknown No. of Clusters	114
11. Glossary	115

1. Introduction to Pattern Recognition

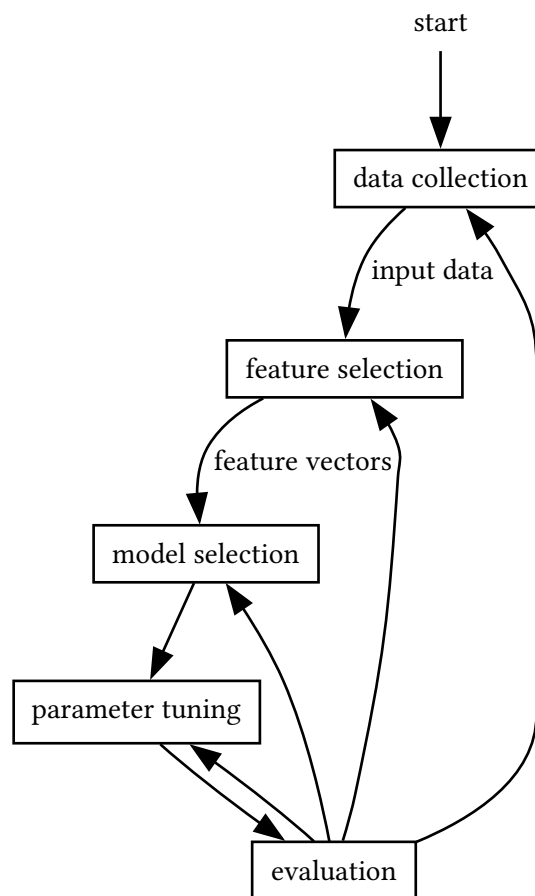
Pattern Recognition is creating algorithms that can recognise useful patterns in data, extract useful information, or making decisions based on data



Pattern is a predictable repetition



1.1. The Classifier Design Cycle



1.1.1. Data Collection (Input Data)

We need to make decisions on:

- What data to collect? — is it possible to obtain? what data is useful?
What sources are there? e.g. one or more transducers like camera, microphone, etc
- How much data to collect? — is it representative?
- Does data need cleaning? — reprocessing/cleaning (e.g. segmentation, noise reduction, outlier removal, dealing with missing values, etc)

In this module, assume we have input data and concentrate on subsequent work.

Discrete Data is integer/symbolic valued

Continuous Data is real/continuous valued

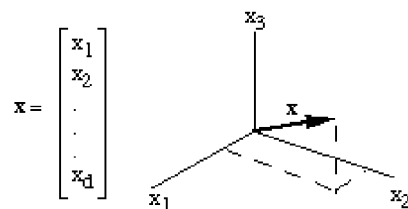
Univariate Data is containing one variable

Multivariate Data is containing many variables

1.1.2. Feature Selection (Feature Vectors)

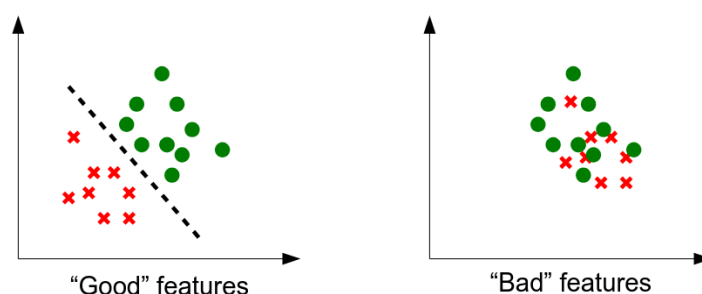
An **exemplar** is a particular datapoint which is represented by a feature vector (also called item, sample, instance, ...)

A **feature vector** is a d -dimensional column vector which is a combination of d features



The **feature space** is the (multidimensional) space defined by the feature vectors in the dataset

How datapoints are distributed in the feature space will influence classifier performance! Ideally they help us discriminate between classes.



A **dataset** is the collection of feature vectors for all exemplars

$$\begin{aligned} \text{sample 1} &= \begin{bmatrix} x_{11} \\ x_{21} \\ \vdots \\ x_{d1} \end{bmatrix} & \text{sample 2} &= \begin{bmatrix} x_{12} \\ x_{22} \\ \vdots \\ x_{d2} \end{bmatrix} & \dots & \text{sample } n &= \begin{bmatrix} x_{1n} \\ x_{2n} \\ \vdots \\ x_{dn} \end{bmatrix} \\ \text{dataset} &= \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \dots & \vdots \\ x_{d1} & x_{d2} & \dots & x_{dn} \end{bmatrix} & & \text{dataset} \end{aligned}$$

Linearly Separable is exemplars from two classes can be separated by a hyperplane in feature space

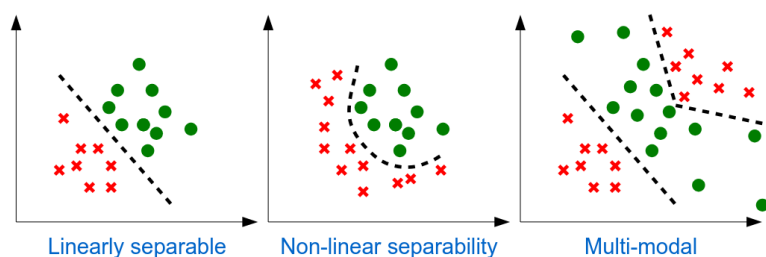


Figure 2: Certain types of distributions have different names

1.1.3. Models, Tuning, & Training

A model is the method used to perform the classification, there are several design considerations:

- What model do we use? e.g. NN, SVM, Random Forest, etc.
Can lead to different results.
- How to use data to define parameters? What hyper-parameters to use?

Generalisation is how well a model performs on new data

Overfitting is making the model so specific to the training data that it fails to generalise to new data

A **hyper-parameter** is a value used by the learning algorithm in its search for the optimal parameters of the classifier

Grid Search is a method of trying to find suitable hyper-parameters that searches all possible combinations of values within defined ranges

Training Data is collection of feature vectors used by learning algorithm to tune the parameters of the classifier

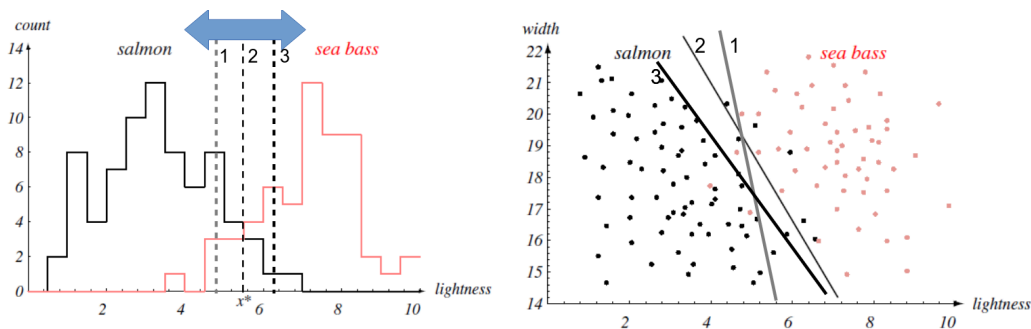


Figure 3: Salmon vs. Sea Bass classification example

We want parameters that give best performance!

Types of Learning

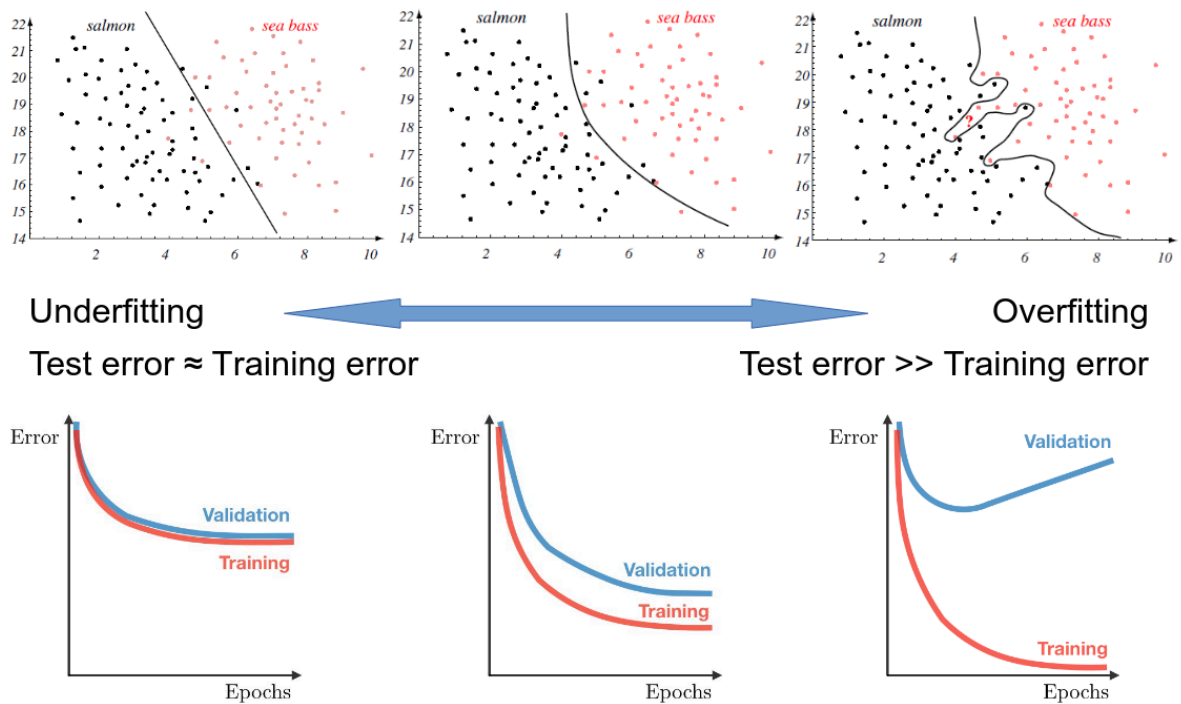
- **Supervised:** desired output defined for each feature vector in the training set
 - **Regression:** outputs are continuous variables (real numbers)
 - **Classification:** outputs are discrete variables (category labels)
- **Weakly-supervised:** inexact/inaccurate labels used in training; typically cheaper to produce
- **Unsupervised learning:** desired output is undefined for each feature vector
 - **Clustering:** discover natural groupings of the data
 - **Embedding:** discover a low-level dimensional manifold or surface near which the data lives
 - **Factorisation:** discover a set of components that can be combined together to reconstruct the samples in the data
- **Reinforcement:** for each input, learning algorithm makes a decision and a critic/teacher compares the decision $\omega^* = g(\mathbf{x})$ to the correct answer and occasionally rewards or punishes
- **Transfer:** train classifier on task where data is abundant then fine-tune on a data set where data is less abundant; expectation is pre-training will help it perform better on the second task

1.1.4. Evaluation

Decision Theory is the methods for making decisions that reduce cost rather than misclassification rate

A **dicotomizer** is a classifier that places exemplars in one two classes (also called a binary classifier)

Test Data is the collection of feature vectors used by to evaluate the performance of the trained classifier (this dataset should be distinct from the training data to ensure generalisation)



For metrics, e.g. confusion matrix, error rate, accuracy, TP/FN, etc, refer to DM1.

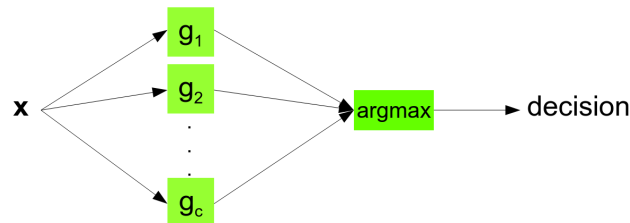
2. Discriminant Functions

A **discriminant function**, g , is a scalar function defined over a feature vector $\mathbf{x}$ that determines class membership:

$$g_i(\mathbf{x}) \text{ for } i = 1 \dots c \text{ (classes)}$$

We use the output of a discriminant function to make categorisation decisions:

assign feature vector $\mathbf{x}$ to class ω_j if $g_j(\mathbf{x}) > g_i(\mathbf{x}) \forall i \neq j$



Note

The following have no effect on discriminant functions:

- $g_i(\mathbf{x}) \leftarrow g_i(\mathbf{x}) + a$ (add a constant)
- $g_i(\mathbf{x}) \leftarrow b \cdot g_i(\mathbf{x})$ (multiply by a constant)
- $g_i(\mathbf{x}) \leftarrow f(g_i(\mathbf{x}))$ where f is a monotonically increasing function

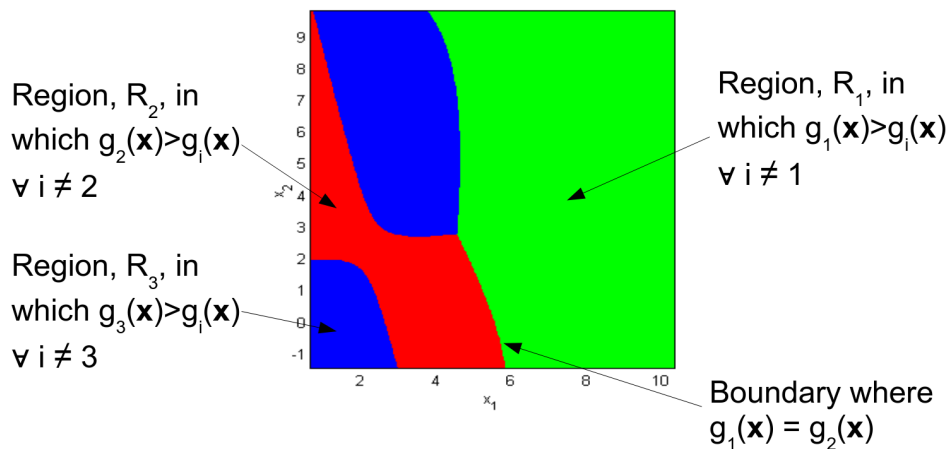


Figure 5: Discriminant functions divide feature space into regions, which are separated by boundaries

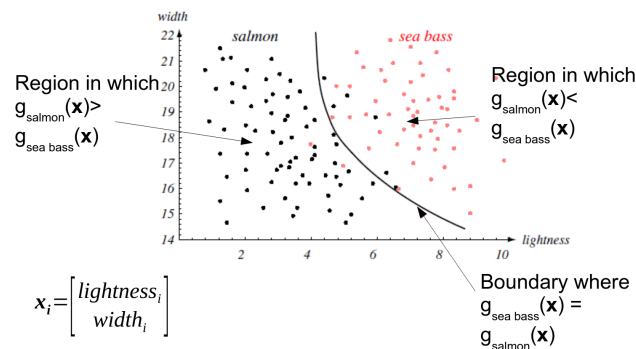


Figure 6: Decision regions for fish function

2.1. Dichotomizers

A **dichotomizer** is a classifier that places exemplars in one two classes (also called a binary classifier)

It can be implemented either by comparison of two functions or as one function comparing signs:

Assign x to ω_1 if $g_1(x) > g_2(x)$
otherwise assign x to ω_2

Define function $g(x) = g_1(x) - g_2(x)$
Hence assign x to ω_1 if $g(x) > 0$
otherwise assign x to ω_2

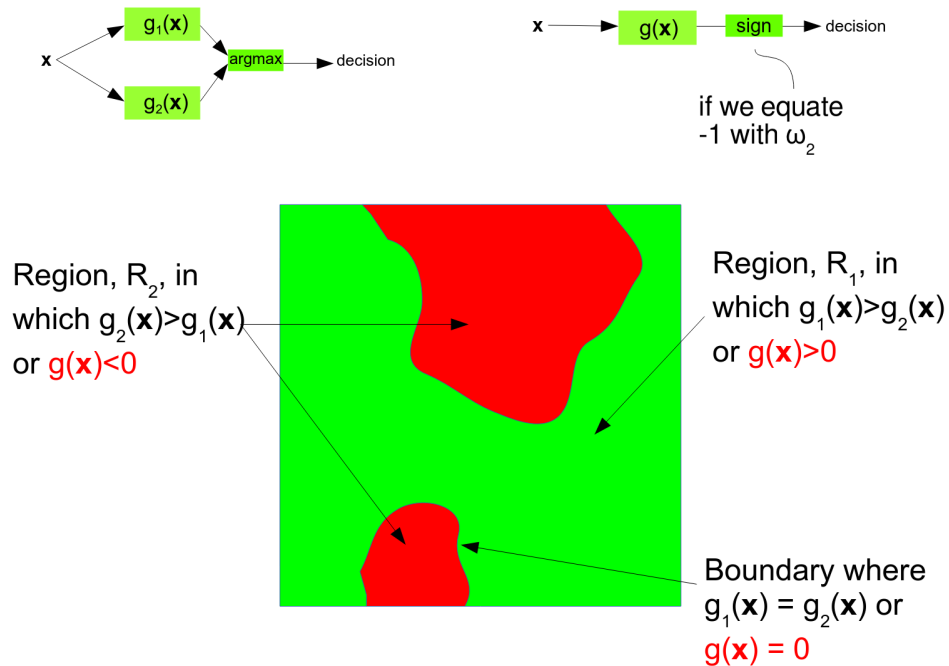


Figure 7: There are only two decision regions which are not necessarily contiguous

Note

The fish example (Figure 6) is a dichotomizer!

2.2. Linear Discriminant Functions

A **linear discriminant function** is defined by a linear combination of feature values:

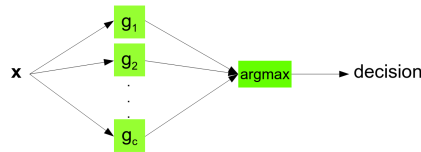
$$g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$$

Where $\mathbf{x}$ is the feature vector, $\mathbf{w}$ is the column vector of parameter/weight values, w_0 is a scalar called the bias/threshold weight, and subscript T means transpose.

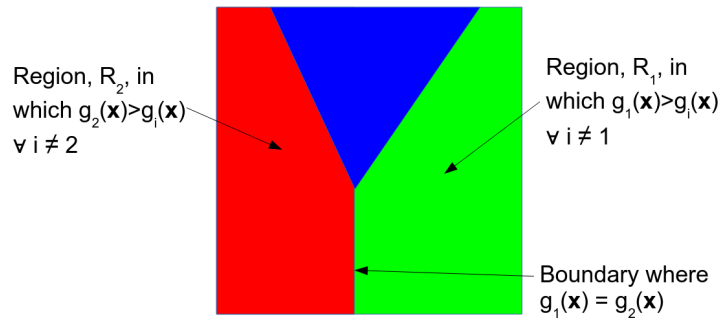
We use the classification rule:

- assign feature vector $\mathbf{x}$ to class ω_j if $g_j(\mathbf{x}) > g_i(\mathbf{x}) \forall i \neq j$
- i.e. choose ω_j if $\mathbf{w}_j^T \mathbf{x} + w_{j0} > \mathbf{w}_i^T \mathbf{x} + w_{i0} \forall i \neq j$

We call this classifier a **linear machine**.



When $g(\mathbf{x})$ is linear, feature space is divided into exactly c contiguous regions



As before, when we have only two categories, we can define a single function:

$$\begin{aligned}
 g(\mathbf{x}) &= g_1(\mathbf{x}) - g_2(\mathbf{x}) \\
 &= (\mathbf{w}_1^T \mathbf{x} + w_{10}) - (\mathbf{w}_2^T \mathbf{x} + w_{20}) \\
 &= (\mathbf{w}_1 - \mathbf{w}_2)^T \mathbf{x} + (w_{10} - w_{20}) \\
 &= \mathbf{w}^T \mathbf{x} + w_0
 \end{aligned}$$

Categorisation rule is then assign $\mathbf{x}$ to ω_1 if $g(\mathbf{x}) > 0$ otherwise assign to ω_2 ,
or, assign $\mathbf{x}$ to ω_1 if $\mathbf{w}^T \mathbf{x} > -w_0$ otherwise assign to ω_2 .

Note

To simplify the maths, we can 'augment' the weight and data vectors, instead of defining $g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$, instead define $g(\mathbf{x}) = \mathbf{a}^T \mathbf{y}$ where $\mathbf{a} = [w_0 \mathbf{w}^T]^T$ and $\mathbf{y} = [1 \mathbf{x}^T]^T$.

Note

Equivalent ways of writing linear discriminant functions:

$$g(\mathbf{x}) = \mathbf{a}^T \mathbf{y} \quad g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 \quad g(\mathbf{x}) = w_0 + \sum_i w_i x_i$$

2.2.1. Decision Boundaries & Hyperplanes

The location of the decision boundary is defined by the weights.

In 2D feature space the **hyperplane** is defined as:

$$\begin{aligned}x_2 &= mx_1 + c \\&= \frac{-w_1}{w_2}x_1 + \frac{-w_0}{w_2} \\&\rightarrow w_0 + w_1x_1 + w_2x_2 = 0\end{aligned}$$

Properties of hyperplanes

- The distance of the hyperplane from the origin is proportional to w_0 (so w_0 determines the location of the hyperplane)
- The vector normal to the hyperplane is w (so w determines the orientation of the hyperplane)
- The vector normal to the hyperplane points towards feature vectors for which $g(x)$ is **positive** And vice-versa, points away from feature vectors that are **negative**
- The value of $g(x)$ provides a measure of how far x is from the decision boundary. The actual distance is given by $\frac{|g(x)|}{\|w\|}$
- These properties are also true for linear discriminant functions in higher-dimensional space

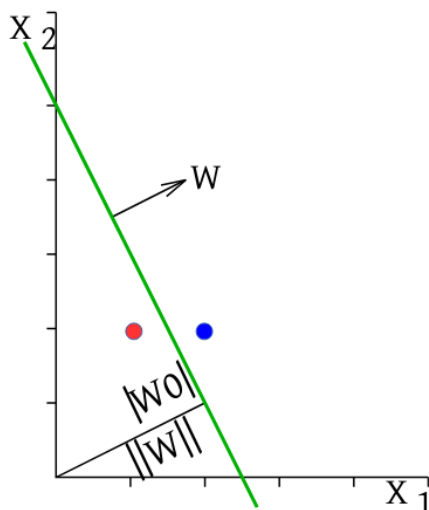


Figure 9: Hyperplane in 2D

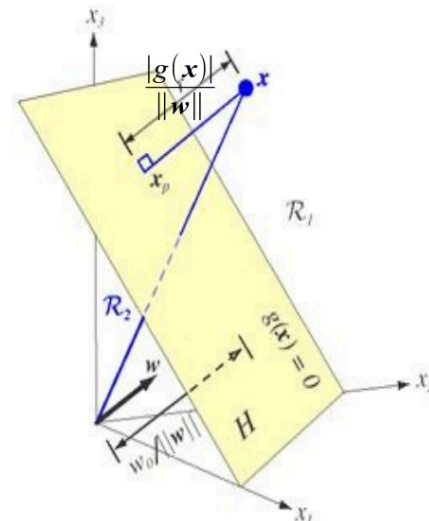
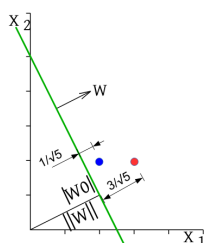


Figure 10: Hyperplane in 3D

Example values



Suppose $w_0 = -5$, $w = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$ and some $x_1 = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$, $x_2 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$

distance of hyperplane from origin

vector values

$$\frac{|w_0|}{\|w\|} = \frac{5}{\sqrt{2^2 + 1^2}} = 2.24$$

$$g(x_1) = 4 + 2 - 5 = +1$$

$$g(x_2) = 2 + 2 - 5 = +3$$

2.2.2. Generalised Linear Discriminant Functions

A **linear** discriminant function can be written as

$$g(\mathbf{x}) = w_0 + \sum_i w_i x_i$$

A **quadratic** discriminant function can be written as

$$g(\mathbf{x}) = w_0 + \sum_i w_i x_i + \sum_{i,j} w_{ij} x_i x_j$$

This can also be thought of as a linear discriminant function in an expanded feature space:

$$g(\mathbf{x}) = \mathbf{a}^T \mathbf{y} \text{ where } \mathbf{a} = [w_0 \ w_1 \ w_2 \ w_{11} \ w_{12} \ w_{22} \ \dots]^T$$

$$\mathbf{y} = [1 \ x_1 \ x_2 \ x_1 x_1 \ x_1 x_2 \ x_2 x_2 \ \dots]^T$$

A **generalised discriminant function** can be written as

$$g(\mathbf{x}) = w_0 + w_1 f_1(\mathbf{x}) + w_2 f_2(\mathbf{x}) + \dots + w_N f_N(\mathbf{x})$$

Where $f_i(\mathbf{x})$ for $i = 1..N$ can be any scalar function of $\mathbf{x}$

This allows us to define any arbitrary decision surface in the original feature space, $\mathbf{x}$

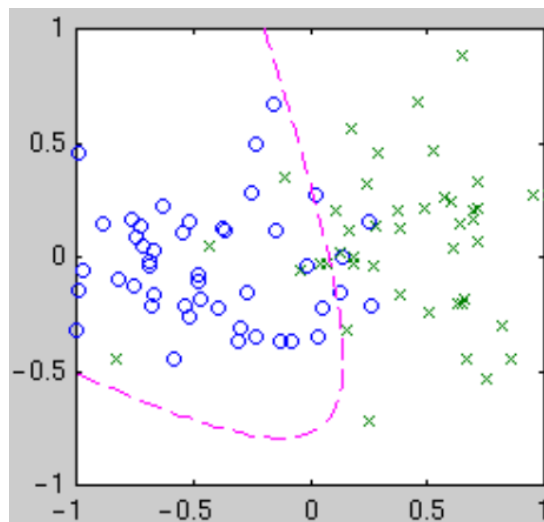


Figure 11: Boundaries can now be quadratic curves or hyperquadratic surfaces

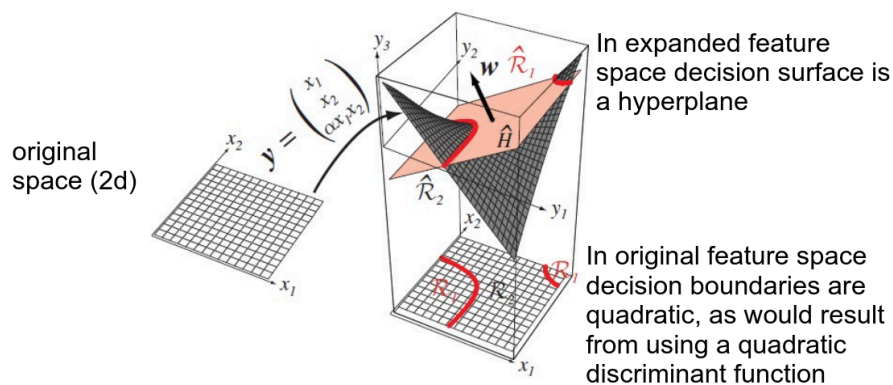


Figure 12: More complex boundaries can be generated using linear discriminant functions in expanded feature space

2.3. Learning Decision Boundaries (Perceptron Learning)

So far we have assumed we know $g(\mathbf{x})$, and hence, that we can determine the class of a new input vector $\mathbf{x}$. But typically, we only have training data and we need to determine the appropriate form of $g(\mathbf{x})$ from this data.

Only need to consider how to learn linear discriminant functions as we can learn any arbitrary decision surfaces by changing the feature space and learning a linear discriminant function in the new feature space.

Our goal is, given a training dataset $(\mathbf{x}_1, \omega'_1), (\mathbf{x}_2, \omega'_2), \dots, (\mathbf{x}_n, \omega'_n)$, to determine the weights $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$ such that $g_j(\mathbf{x}_k) > g_i(\mathbf{x}_k) \forall i \neq j$ where $j = \omega'_k$ for all k . If a solution exists, then the data set is [linearly separable](#).

Suppose a *two-category linearly separable case*:

- We have a set of n linearly separable samples $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n$; some labelled ω_1 , some ω_2
- We want to use these samples to determine weight vector $\mathbf{a}$ of a dichotomizer
- A sample $\mathbf{y}_k$ is correctly classified if $\mathbf{a}^T \mathbf{y}_k > 0$ and $\mathbf{y}_k$ is labelled ω_1 ; similarly if < 0 and ω_2

Sample Normalisation

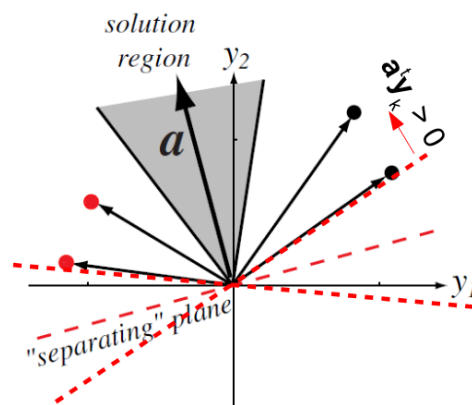
Replace all samples from class ω_2 by their negatives:

$$\mathbf{y} \leftarrow -\mathbf{y} \forall \mathbf{y} \in \omega_2$$

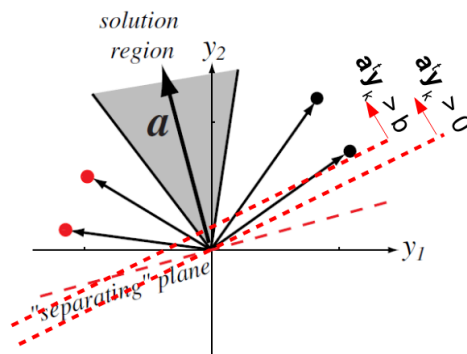
Now, a sample $\mathbf{y}_k$ is correctly classified if $\mathbf{a}^T \mathbf{y}_k > 0$ regardless if $\mathbf{y}_k$ is labelled ω_1 or ω_2 .

Hence, we can ignore class labels and look for a vector $\mathbf{a}$ such that $\mathbf{a}^T \mathbf{y}_k > 0 \forall k$.

There are potentially many solutions that satisfy $\mathbf{a}^T \mathbf{y}_k > 0 \forall k$:

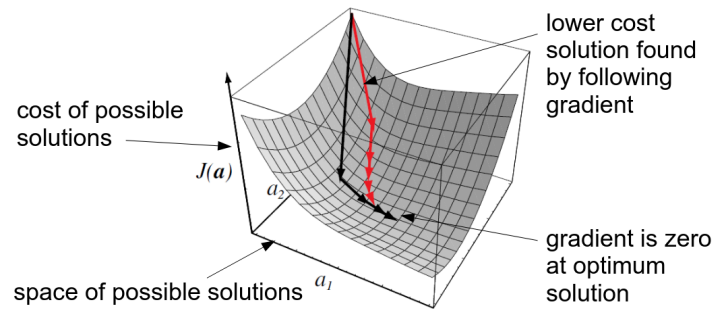


Furthermore, there are fewer solutions that satisfy $\mathbf{a}^T \mathbf{y}_k > b \forall k$ for a given k , +ve value (margin)



2.3.1. Gradient Descent Procedures

We define a cost(/objective/loss) function $J(\mathbf{a})$ that is minimised if $\mathbf{a}$ is a solution vector.



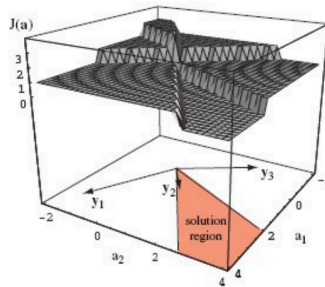
This problem can be solved by gradient descent, the basic procedure being as follows:

- Initialise $\mathbf{a}$ to arbitrary solution.
- Repeat:
 - Compute gradient vector at $\mathbf{a}$: $\nabla J(\mathbf{a})$
 - Move solution in direction of steepest descent

$$\mathbf{a} \leftarrow \mathbf{a} - \eta \nabla J(\mathbf{a})$$

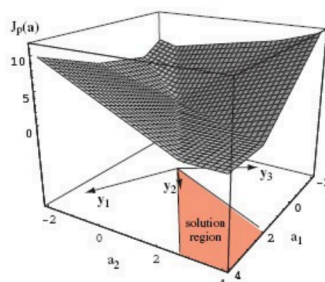
Where η is the learning rate/step size.

The simplest approach is to define a cost function based on no. of samples misclassified by $\mathbf{a}$, the **number misclassified criterion function**, but this would be piecewise constant and hence have no gradient to follow (not 'convex').



Instead, we can define a cost function based on the sum of the distances to the decision boundary for all misclassified samples, **perceptron criterion function**. Given $\mathcal{X}$ is the set of samples misclassified, we find:

$$J_p(\mathbf{a}) = \sum_{\mathbf{y} \in \mathcal{X}} (-\mathbf{a}^T \mathbf{y})$$



2.3.1.1. Batch Perceptron Learning Algorithm

The gradient of the perceptron cost function is:

$$\nabla J_p(\mathbf{a}) = \sum_{\mathbf{y} \in \mathcal{X}} (-\mathbf{y})$$

Hence the update rule becomes:

$$\mathbf{a} \leftarrow \mathbf{a} + \eta \sum_{\mathbf{y} \in \mathcal{X}} \mathbf{y}$$

Which converges when $\mathcal{X}$ is empty, hence $\nabla J_p \mapsto 0$ (i.e. no more errors)

Will only converge if data is linearly separable

Gradient descent with the **batch perceptron cost function** is:

- Set value of hyper-parameter (η)
- Initialise $\mathbf{a}$ to arbitrary solution
- Repeat:
 - Compute gradient vector at $\mathbf{a}$: $\nabla J_p(\mathbf{a}) = \sum_{\mathbf{y} \in \mathcal{X}} (-\mathbf{y})$
 - Move solution in direction of steepest descent: $\mathbf{a} \leftarrow \mathbf{a} + \eta \sum_{\mathbf{y} \in \mathcal{X}} \mathbf{y}$

2.3.1.2. Sequential Perceptron Learning Algorithm

Update based only on single sample errors (updating of weight vector is ‘sequential’/‘online’):

- Set value of hyper-parameter (η)
- Initialise $\mathbf{a}$ to arbitrary solution
- Repeat:
 - For each sample $\mathbf{y}_k$ in the data set:
 - If $\mathbf{y}_k$ is misclassified:
 - Move solution towards $\mathbf{y}_k$: $\mathbf{a} \leftarrow \mathbf{a} + \eta \cdot \mathbf{y}_k$

Note

Perceptron updates move weights towards misclassified exemplars. This only works for exemplars in ω_2 if we apply sample normalisation (multiply by -1).

Alternatively, we can move towards misclassified exemplars from ω_1 and away from misclassified exemplars from ω_2 , if we were to not use sample normalisation.

e.g. let $w_1 = 1, w_2 = -1$, hence update rules become:

- **batch**: $\mathbf{a} \leftarrow \mathbf{a} + \eta \sum_{\mathbf{y} \in \mathcal{X}} \omega' \mathbf{y}$
- **sequential**: $\mathbf{a} \leftarrow \mathbf{a} + \eta \cdot \omega'_k \mathbf{y}_k$

Note

Example included: `7ccmpnn-wk2-sequential-perceptron-learning-example.pdf`

2.3.1.3. Multiclass Perceptron Learning Algorithm

Assign feature vector to $\mathbf{x}$ to class ω_j if $g_{j(\mathbf{x})} > g_{i(\mathbf{x})} \forall i \neq j$.

If classification is wrong, adjust weights of discriminant functions:

- move weights for required class towards input
- move weights of wrongly selected class away from input

Hence algorithm:

- Set value of hyper-parameter (η)
- For each possible class, c , initialise $\mathbf{a}_c$ to arbitrary solution
- Repeat:
 - For each sample $(\mathbf{y}_k, \omega_k)$ in the dataset:
 - Classify $c' = \arg \max g_c(\mathbf{x}_k)$
 - If $\mathbf{y}_k$ is mis-classified then:
 - Move $\mathbf{a}_{\omega_k}$ towards $\mathbf{y}_k$: $\mathbf{a}_{\omega_k} \leftarrow \mathbf{a}_{\omega_k} + \eta \cdot \mathbf{y}_k$
 - Move $\mathbf{a}_{c'}$ away from $\mathbf{y}_k$: $\mathbf{a}_{c'} \leftarrow \mathbf{a}_{c'} - \eta \cdot \mathbf{y}_k$

2.4. Learning Decision Boundaries (Minimum Squared Error procedures)

In perceptron learning:

- we use gradient descent to calculate suitable linear discriminant functions
- learning is driven by misclassified exemplars
- converges only if data is linearly separable
- solves linear inequalities $\mathbf{a}^T \mathbf{y}_k > 0 \forall k$

With MSE procedures:

- solve linear equations $\mathbf{a}^T \mathbf{y}_k = b_k \forall k$
- provides a result even if data is not linearly separable
- learning depends on all exemplars
- uses matrix inversion (or gradient descent) to calculate l.d. functions

Suppose a *two-category case with sample normalisation*:

- We have a set of n samples $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n$; some labelled ω_1 , some ω_2
- We replace all samples from second class by negatives: $\mathbf{y} \leftarrow -\mathbf{y} \forall \mathbf{y} \in \omega_2$
- Now, a sample $\mathbf{y}_k$ is correctly classified if $\mathbf{a}^T \mathbf{y}_k > 0 \forall k$
- Also if $\mathbf{a}^T \mathbf{y}_k > b_k \forall k$ where b_k is a positive value (the margin)

MSE via Pseudoinverse

Let Y be a matrix, each row of which contains one exemplar, $\mathbf{y}_k^T$ from the training data (in augmented and sample normalised format). Then we can express:

$$\mathbf{Y} \mathbf{a} = \mathbf{b} \quad \begin{bmatrix} y_{10} & y_{11} & \dots & y_{1d} \\ \vdots & & & \\ y_{k0} & y_{k1} & \dots & y_{kd} \\ \vdots & & & \\ y_{n0} & y_{n1} & \dots & y_{nd} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} = \begin{bmatrix} b_0 \\ \vdots \\ b_k \\ \vdots \\ b_n \end{bmatrix}$$

Typically Y is rectangular, so the solution is given by $\mathbf{a} = \mathbf{Y}^\dagger \mathbf{b}$

Where $\mathbf{Y}^\dagger$ is the matrix pseudo-inverse (Moore-Penrose inverse)

MSE via Gradient Descent

The pseudo-inverse provides a closed-form solution to find $\mathbf{a}$ which minimises the squared error:

$$e^2 = \| \mathbf{Y} \mathbf{a} - \mathbf{b} \|^2$$

An alternative method to find a solution is to define a cost function $J(\mathbf{a})$ that is to be minimised:

$$J_s(\mathbf{a}) = \frac{1}{2} \| \mathbf{Y} \mathbf{a} - \mathbf{b} \|^2$$

2.4.1. Widrow-Hoff (LMS) Learning Algorithm

The gradient of the MSE cost function is:

$$\nabla J_s(\mathbf{a}) = \mathbf{Y}^T(\mathbf{Y}\mathbf{a} - \mathbf{b})$$

Hence, the batch update rule becomes:

$$\mathbf{a} \leftarrow \mathbf{a} - \eta \cdot \mathbf{Y}^T(\mathbf{Y}\mathbf{a} - \mathbf{b})$$

The corresponding sequential update rule is:

$$\begin{aligned} \mathbf{a} &\leftarrow \mathbf{a} - \eta(\mathbf{a}^T \mathbf{y}_k - b_k) \mathbf{y}_k \\ \text{or, } \mathbf{a} &\leftarrow \mathbf{a} + \eta(b_k - \mathbf{a}^T \mathbf{y}_k) \mathbf{y}_k \end{aligned}$$

Sequential Widrow-Hoff Learning Algorithm

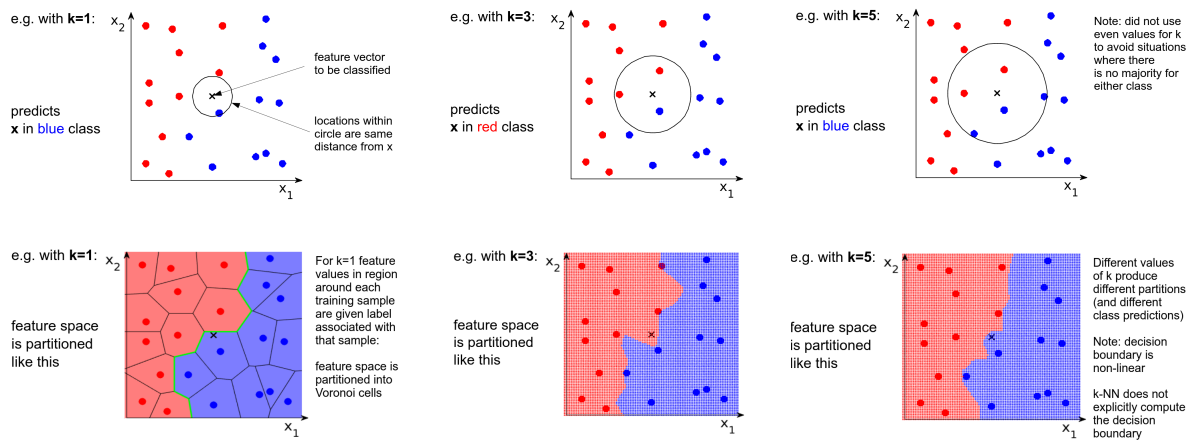
- set values of hyper-parameters $(\eta, \theta, \mathbf{b})$
- initialise $\mathbf{a}$ to an arbitrary solution
- repeat until $|\sum_k (b_k - \mathbf{a}^T \mathbf{y}_k) \mathbf{y}_k| < \theta$:
 - for each sample $\mathbf{y}_k$ in the dataset:
 - update solution: $\mathbf{a} \leftarrow \mathbf{a} + \eta(b_k - \mathbf{a}^T \mathbf{y}_k) \mathbf{y}_k$

Note

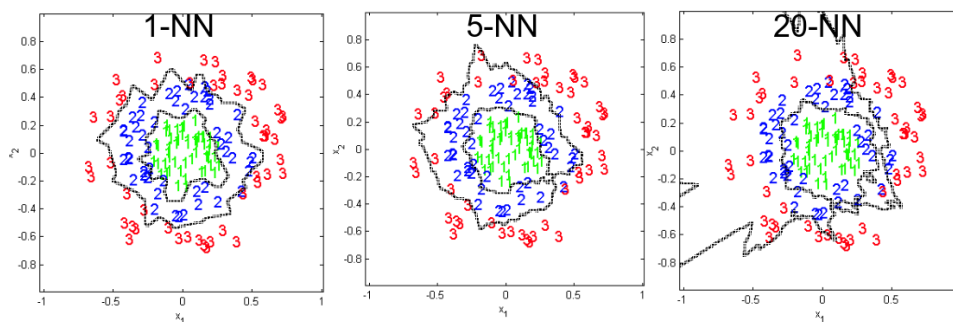
Example included: `7ccsmpnn-wk2-sequential-widrow-hoff-example.pdf.pdf`

2.5. k-Nearest Neighbour Classifier

The **k-Nearest Neighbours Classifier** is a classifier that partitions feature-space without learning; a feature vector x is classified into the same class as the majority of the k ([hyper-parameter](#)) most similar training samples



k-NN can also succeed with multi-modal/non-linearly separable data!



A simple classification algorithm only requires:

- a metric to measure distance; typically Euclidean
- a value for hyper-parameter k
 - results highly depend on choice
 - avoid even numbers to avoid ties
 - larger values of k can allow:
 - +‘ve: smoother decision boundaries (smooths over outliers in training data)
 - +‘ve: more probabilistically accurate (prediction based on more samples)
 - -‘ve: include samples far from x (classification based on irrelevant samples)
 - -‘ve: increase computational costs

k-NN classifier means (positives/negatives):

- +‘ve: no training time (“lazy learning”)
- +‘ve: classification accuracy can be excellent if number of samples is very high
- +‘ve: can work when data is multi-modal or non-linearly separable
- +‘ve: can use proportion of neighbours in each class to give an estimate of probability that the sample belongs to each class
- -‘ve: large storage requirements (can be ameliorated)
- -‘ve: computationally expensive at runtime
- -‘ve: no. of training samples required increases exponentially with dimensionality of feature space

3. Introduction to Neural Networks

3.1. Biological Neural Networks

A **neural network** is a network of connected elements called neurons.

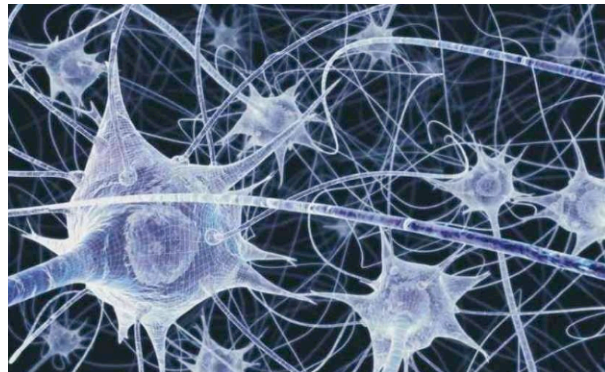


Figure 19: Neurons found in a biological brain

A **brain** is a highly complex, non-linear and parallel information processing system, it is composed of lots of neurons (~ 85 billion for humans, ~ 1 billion for cats, ~ 1 million for bees).

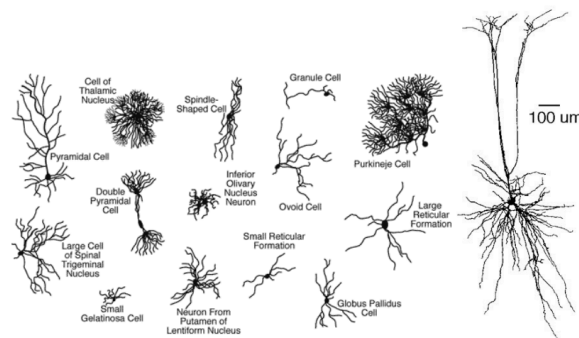


Figure 20: A neuron is a type of cell
They come in many different types

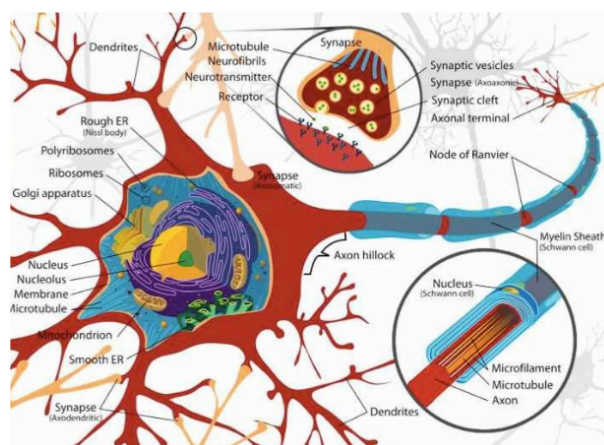
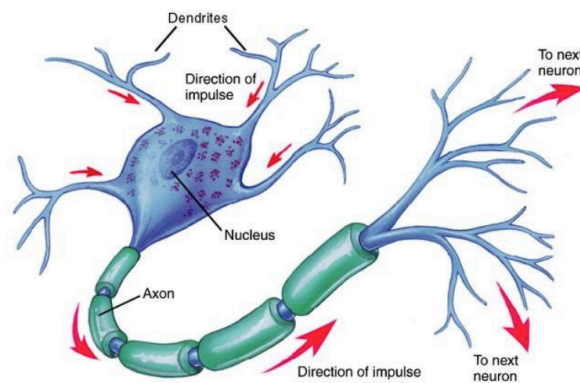


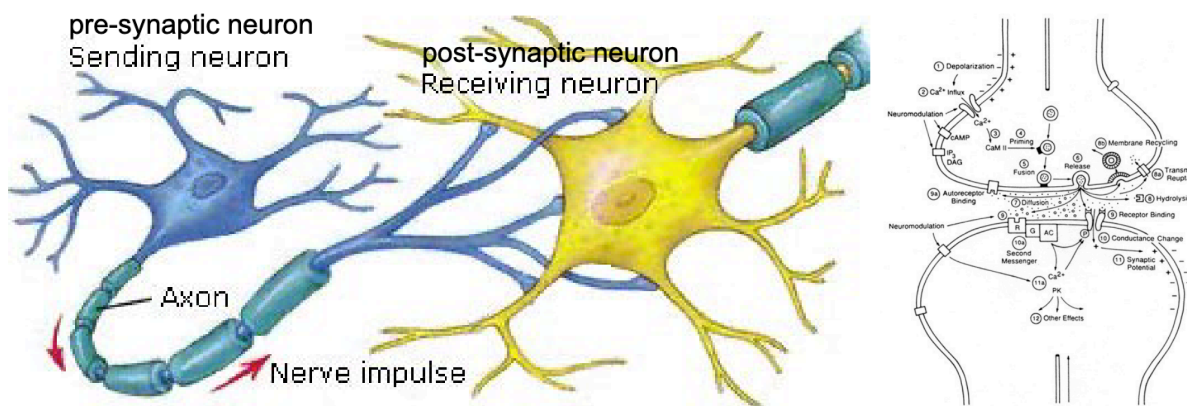
Figure 21: They are small, complex, biological machines

They consist of a cell body (or 'nucleus'/'soma'), many dendrites, an axon, and many synapses.



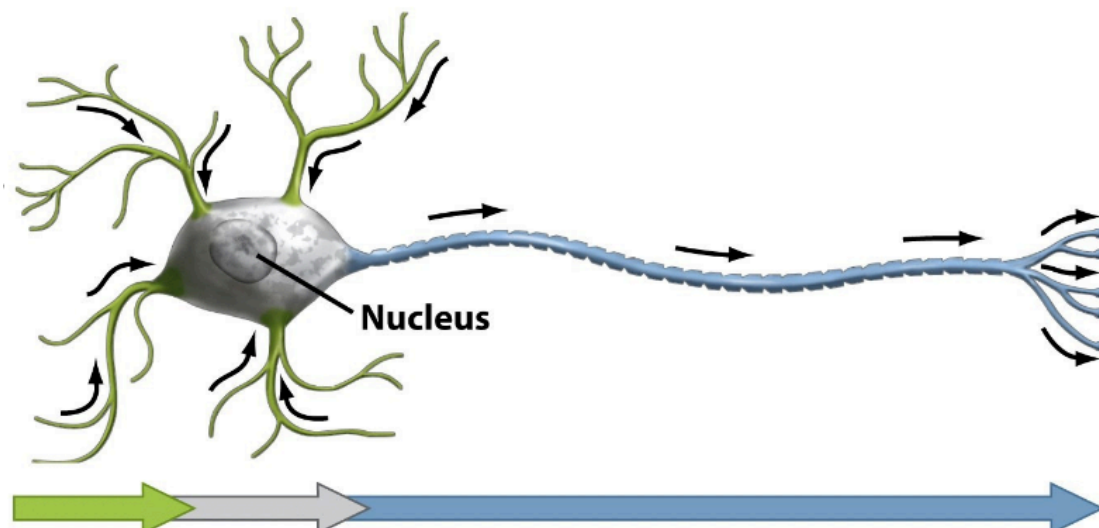
A **synapse** is a connection between the axon of one neuron and the dendrites of another neuron. Each synapse is also a small, complex, biological machine.

- Activity in the pre-synaptic neuron causes the release of neurotransmitters from synaptic vesicles.
- These neurotransmitters diffuse across the gap to the receptors on the post-synaptic neuron and cause activity there.
- There are ~ 100 different neurotransmitters, e.g. dopamine, serotonin, acetylcholine

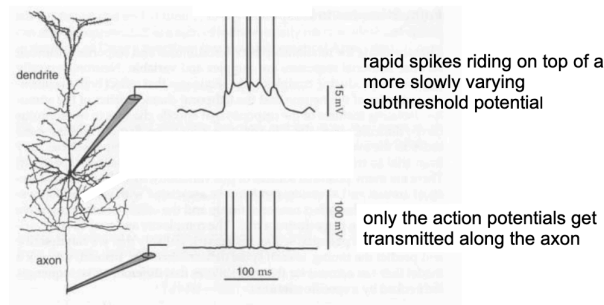


The dendrites transmit input signals from the synapses to the soma.

The soma integrates inputs and if total is sufficiently strong, then it emits an action potential ('fires'/'spikes') which is then transmitted by the axon to the dendrites of other neurons.



An **action potential** is an electrical impulse. A neuron is an electronic device with a voltage potential, the dendrites and axons act like (highly non-linear wires).



There is a threshold for spiking, reaching the **spiking threshold** requires either:

- repetitive stimulation of the same synapses (temporal summation)
- simultaneous stimulation of a large number of synapses (spatial summation)
- or typically, a combination of both

The arrival of an impulse at a synapse may have the opposite effect, i.e. may render the post-synaptic neuron less activated by other stimuli (inhibitory).

Hence, we define two types of synapse:

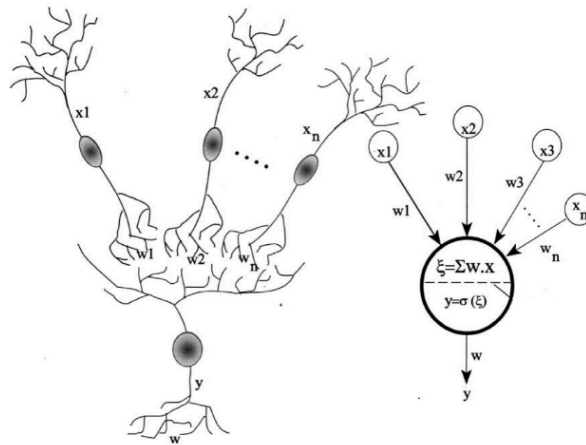
- **Excitatory**: tend to cause spiking in post-synaptic neuron, e.g. glutamate
- **Inhibitory**: tend to prevent spiking in the post-synaptic neuron, e.g. GABA

Different synapses may have stronger/weaker effects on post-synaptic neuron; synapses may have different strengths ('weights') or may change their strength. **Synaptic plasticity** permits the nervous system to adapt to its environment (learning, memory, new computations); it works by two mechanism of creation of new synaptic connections between neurons and modification of existing neurons.

A **biological neural network** is a network of neurons interconnected through synapses.

3.2. Artificial Neural Networks

An **artificial neural network** is a parallel computational system consisting of a network of simple processing units which communicate by sending signals to each other over weighted connections. These processing units are analogous to neurons, and weighted connections are analogous to synapses. It is a hugely simplified model.



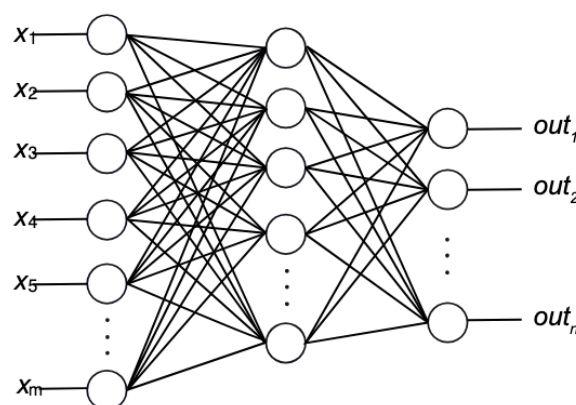
ANNs are also known as parallel distributed processing (PDP) or connectionist models.

We study ANNs as:

- they can help us understand human intelligence/brain dysfunction through models
- they can help us build AI; mimic cognitive function of humans/animals
- they are extremely powerful computational devices; Turing equivalent, universal computers
- any continuous function from input to output can be implemented in a three-layer ANN
e.g. universal function approximators or universal classifiers

While they can provide any calculation, it is problematic to find an appropriate NN architecture.

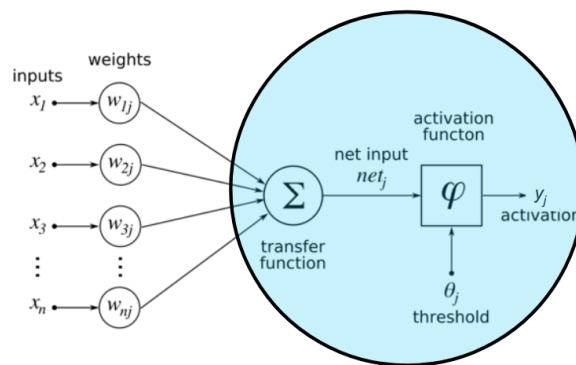
Typically, processing units are arranged into populations/layers; which can either be visible (receive/output information) or hidden (only connect with other processing units).



Visible layers are either:

- input layers: receive signals from environment
input layers are linear as they do not perform any processing, hence often ignored
- output layers: send signals to environment
for classification, we typically “1-hot” encode categories

Each unit receives a vector of inputs, $\mathbf{x}$, from other units or environment. Each input is associated with a weight, w , which determines strength/influence of each input, which can be either + 've (excitatory) or - 've (inhibitory).



The output, y , is computed as a simple function of $\mathbf{x}$ and $\mathbf{w}$.

Response function is often split into:

- a transfer function: determines how inputs are integrated
- an activation function: determines output the neuron produces

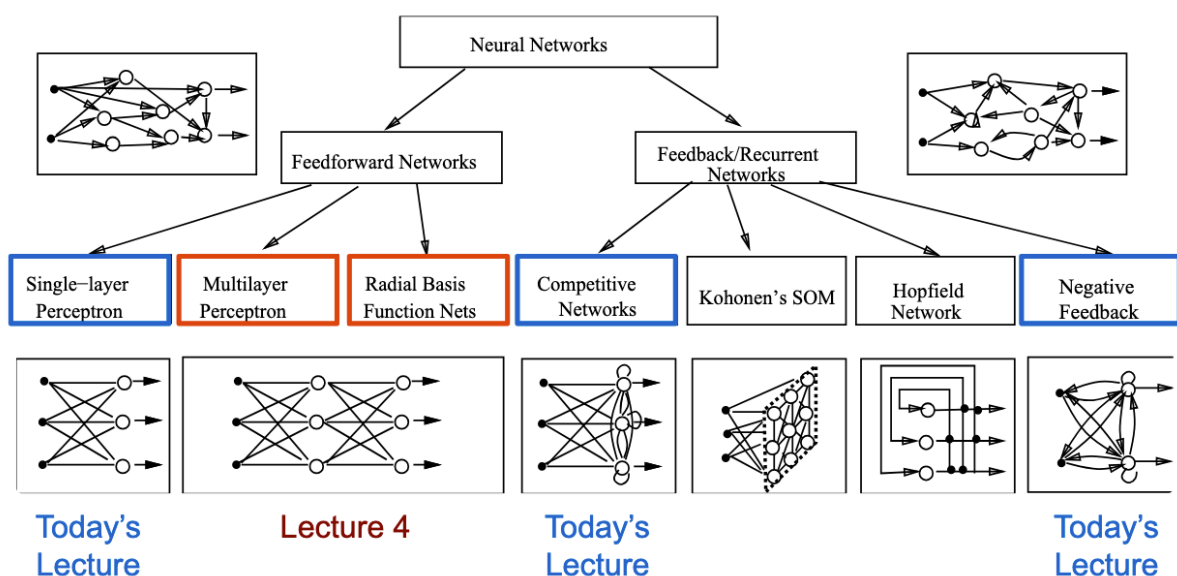
Connection weights can be defined by:

- setting weights explicitly using prior knowledge
- optimising connectivity to achieve some objective
- training network by feeding it training data and allowing it to adapt

Generic ANN

- an environment in which the system operates
- set of processing units
- set of weighted connection units, w_{ij} , which determine influence of unit i on j
- a transfer function
- an activation function
- an activation state, y_j , for every unit
- a method for setting/changing connection weights

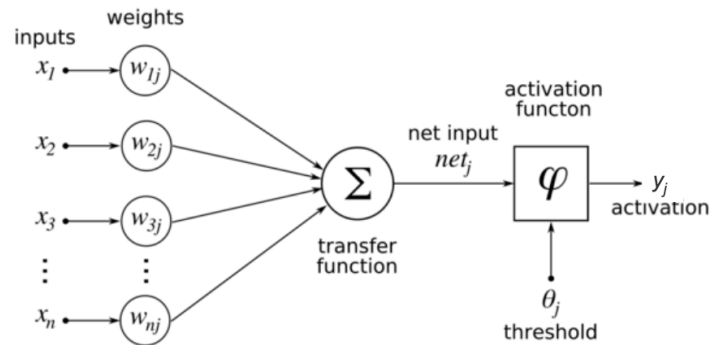
There exist many types of neural networks: which will be covered **now** or **later**



3.2.1. Linear Threshold Units (Perceptrons)

A **linear threshold unit** or **perceptron** is a basic processing unit.

$$y_j = H \left(\underbrace{\sum_i w_{ji} x_i - \theta_j}_{\text{activation function}} \right) \quad \text{e.g. } H(u) = \text{heaviside step function} = \begin{cases} 1, & \text{if } u \geq 0 \\ 0, & \text{otherwise} \end{cases}$$



Simplified notation: $y = H(\sum_i w_i x_i - \theta)$

Vector notation: $y = H(\mathbf{w}\mathbf{x} - \theta)$

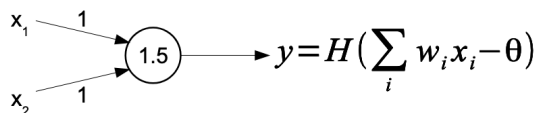
Augmented vector notation: $y = H(\mathbf{w}\mathbf{x})$

Note

There also exists, a restricted case, where weights and activations are binary, known as a Threshold Logic Unit or McCulloch-Pitts neuron.

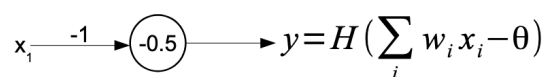
Example units

Logical AND



x_1	x_2	Transfer fn, $\sum w_i x_i$	Activation fn, y
0	0	$(1 \times 0) + (1 \times 0) = 0$	$H(0 - 1.5) = 0$
0	1	$(1 \times 0) + (1 \times 1) = 1$	$H(1 - 1.5) = 0$
1	0	$(1 \times 1) + (1 \times 0) = 1$	$H(1 - 1.5) = 0$
1	1	$(1 \times 1) + (1 \times 1) = 2$	$H(2 - 1.5) = 1$

Logical NOT

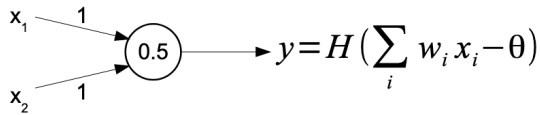


x_1	Transfer fn, $\sum w_i x_i$	Activation fn, y
0	$(-1 \times 0) = 0$	$H(0 - (-0.5)) = 1$
1	$(-1 \times 1) = -1$	$H(-1 - (-0.5)) = 0$

Given just AND and NOT gates, it is possible to build any computational device using a network of linear threshold units!

More example units

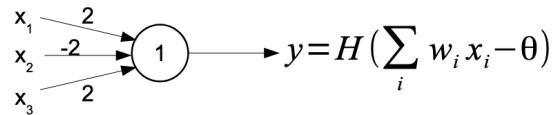
Logical OR



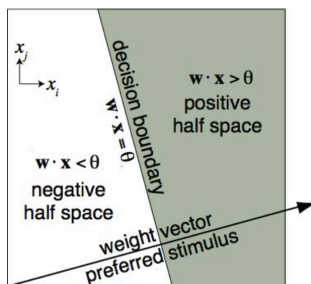
x_1	x_2	Transfer fn, $\sum w_i x_i$	y
0	0	0	0
0	1	1	1
1	0	1	1
1	1	2	1

Arbitrary Function

$$(x_1 \wedge \neg x_2) \vee (x_1 \wedge x_3) \vee (\neg x_2 \wedge x_3)$$



x_1	x_2	x_3	Transfer fn, $\sum w_i x_i = 2x_1 - 2x_2 + 2x_3$	y
0	0	0	0	0
1	0	0	2	1
0	1	0	-2	0
1	1	0	0	0
0	0	1	2	1
1	0	1	4	1
0	1	1	0	0
1	1	1	2	1



w and θ define a hyperplane that divides the input space into two parts, this hyperplane is called the “decision boundary”

AND and OR are linearly separable functions so this works, however not all logical functions are linearly separable, e.g. XOR

	x_1	x_2	AND	OR	XOR
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	1	0

3.2.1.1. Equivalence to Linear Discriminant Functions

A single linear threshold unit is **equivalent** to linear discriminant functions.

And can therefore act as a dichotomizer:

$$\begin{aligned} \text{assign } \mathbf{x} \text{ to } \omega_1 \text{ if } g(\mathbf{x}) > 0 & \quad \text{where } g(\mathbf{x}) = \mathbf{a}^T \mathbf{y}; \mathbf{y} = [1 \ \mathbf{x}^T]^T \\ \equiv \text{assign } \mathbf{x} \text{ to } \omega_1 \text{ if } y = 1 & \quad \text{where } y = H(\mathbf{w}\mathbf{x}) \end{aligned}$$

A layer of linear threshold units is equivalent to a set of linear discriminant functions and can therefore act as a linear machine.

$$\begin{aligned} \text{assign } \mathbf{x} \text{ to } \omega_j \text{ if } g_{j(\mathbf{x})} > g_{i(\mathbf{x})} \quad \forall i \neq j & \quad (g(\mathbf{x} = \mathbf{a}^T \mathbf{y})) \\ \equiv \text{assign } \mathbf{x} \text{ to } \omega_j \text{ if } y_j > y_i \quad \forall i \neq j & \quad (y = \mathbf{w}\mathbf{x}) \end{aligned}$$

Recall, generalised linear discriminant functions can produce non-linear boundaries in the original feature space by applying a linear discriminant function to an expanded feature space. A linear threshold unit can implement the same function as a generalised linear discriminant function:

$$y = w_0 + w_1 f_1(\mathbf{x}) + w_2 f_2(\mathbf{x}) + \dots + w_N f_{N(\mathbf{x})}$$

Both the expansion of the feature-space and the discrimination can be performed by different layers of neurons in a multi-layer neural network!

3.2.1.2. Delta Learning Algorithm

Weights for a linear threshold unit can be learnt using the same methods used to set the parameters of a linear discriminant function, e.g. perceptron learning or MSE learning (Widrow-Hoff).

But in addition, we can also use the delta learning rule.

The **delta learning rule** is a supervised algorithm where we adjust weights in proportion to the difference between the desired output, t , and the actual output, y .

- Sequential (online) update is: $\mathbf{w} \leftarrow \mathbf{w} + \eta(t - y)\mathbf{x}^T$
- Batch update is: $\mathbf{w} \leftarrow \mathbf{w} + \eta \sum_p (t_p - y_p)\mathbf{x}_p^T$

Similar to:

- Widrow-Hoff: except arbitrary margin, b , replaced by desired output, t
- Perceptron: learning only occurs for misclassified exemplars, when $y \neq t$

There are two types of update: (consider online for following equations)

- False negative when $t = 1, y = 0$, make $\mathbf{w}$ more like $\mathbf{x}$

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \mathbf{x}^T$$

- False positive when $t = 0, y = 1$, make $\mathbf{w}$ less like $\mathbf{x}$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \mathbf{x}^T$$

Similarity with multi-class perceptron learning.

The delta learning rule is implemented using gradient descent:

- set value of hyper-parameter η
- initialise $\mathbf{w}$ to arbitrary solution
- repeat:
 - for each sample $(\mathbf{x}_k, t_k)$ in the dataset:
 - update weights: $\mathbf{W} \leftarrow \mathbf{w} + \eta(t_k - H(\mathbf{w}\mathbf{x}_k))\mathbf{x}_k^T$

Run through all samples in turn as many times as necessary for convergence.

If the dataset is not linearly separable, learning will not converge.

Note

Example included: 7ccsmpnn-wk3-sequential-delta-learning-example.pdf

3.2.2. Hebbian Learning Rule

Hebbian Learning Rule is unsupervised and strengthens the connection between an active neuron and any active inputs. Sequential (online) update is:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \cdot \boldsymbol{y} \cdot \boldsymbol{x}^T$$

The Hebbian learning rule is a simple rule that states that when two neurons fire together, the connection between them should strengthen.

3.2.3. Competitive Learning Networks

In **competitive learning networks**, output units compete for the right to respond to the input. Competition means activity of some neurons is suppressed by other neurons.

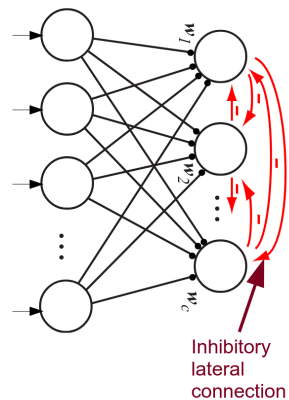


Figure 30: Example of a competitive learning network

Competition can be implemented using:

- **inhibitory lateral weights** between output neurons.
requires outputs to be determined iteratively
- a **selection process** that chooses the winning neuron(s).
generally simpler and more stable

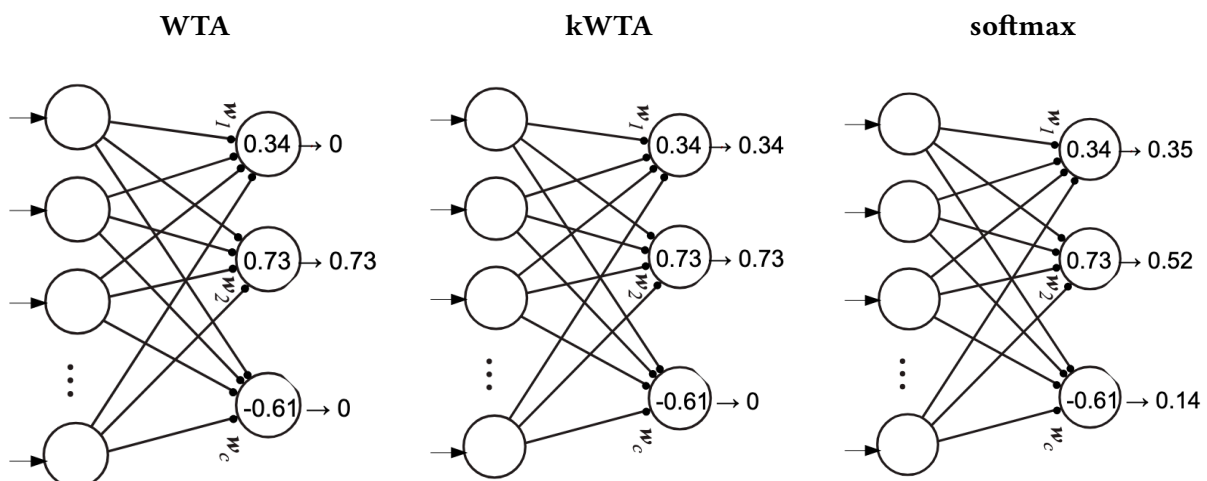
The selection process can be implemented using the following *activation functions*:

- **winner-takes-all (WTA)**: the neuron with the largest response is the winner and remains active; all other neurons have their response set to zero
WTA can be used with Hebbian learning to perform clustering
- **k-winners-take-all (kWTA)**: k neurons with largest response remain active (k is a [hyper-parameter](#)); all other neurons have their response set to zero
- **softmax**: activation of each neuron is determined by

$$y_j \leftarrow \frac{e^{\beta y_j}}{\sum_k e^{\beta y_k}}$$

Where β is a [hyper-parameter](#). Outputs of all neurons will sum to 1.

Popular as activation function used in last layer of networks applied to classification tasks!



3.2.4. Negative Feedback Networks

Negative feedback networks present an alternative way to implement competition by using **inhibitory feedback** connections between the output neurons and the input neurons.

Output neurons compete to *receive input* rather than to produce output.

Using inhibitory feedback weights requires the outputs to be determined iteratively.

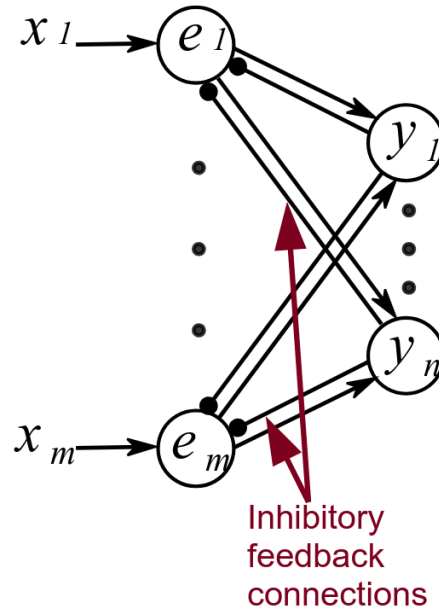


Figure 31: Example of a negative feedback network

They work as follows:

- Activation: try to minimise response from input units, e (i.e. minimise reconstruction error); try to find y values that accurately reconstruct the input (in terms of minimising reconstruction error)

initialise y to zero

perform iterations of:

$$e = x - W^T y \quad \text{update input units}$$

$$y \leftarrow y + \alpha W e \quad \text{update output units}$$

- Learning: adjust weights to accurately reconstruct the input (in terms of minimising the reconstruction error).

$$W \leftarrow W + \beta y e^T$$

We can consider:

- $W^T y$ as a reconstruction of the input
- e as the error between the input and the reconstruction

3.2.5. Autoencoder Networks

An autoencoder network builds on negative feedback networks but instead represents the reconstruction using a separate neural population, r , and does not include the inhibitory feedback connections. Neural networks therefore no longer have to be calculated iteratively but we can still learn weights as to minimise the error between the input and the reconstruction.

An **autoencoder** is a three (two) layer neural network whose aim it is to reconstruct the input.

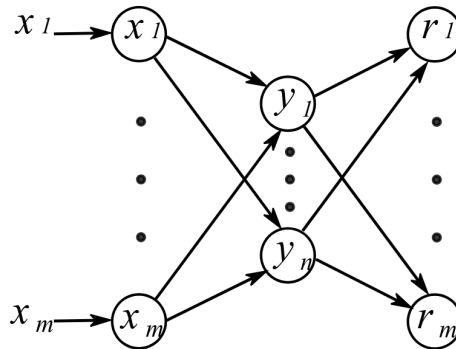
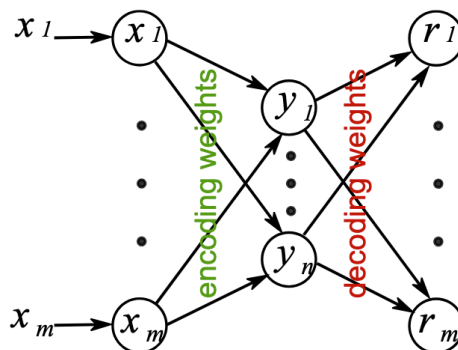


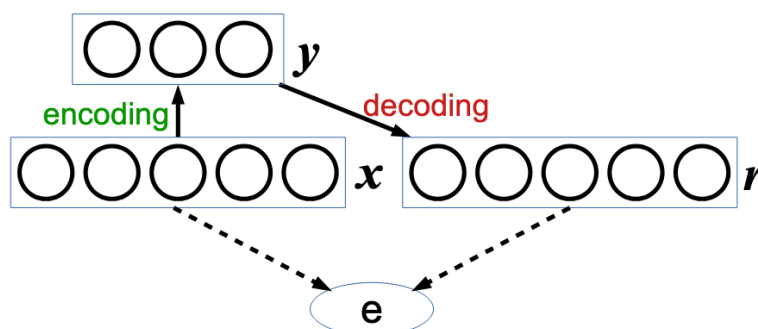
Figure 32: Example of an autoencoder network

Hidden units impose a ‘information bottleneck’ (limited no. of hidden nodes/simultaneously active nodes - sparsity constraint). The hidden units learn a useful representation of the data.

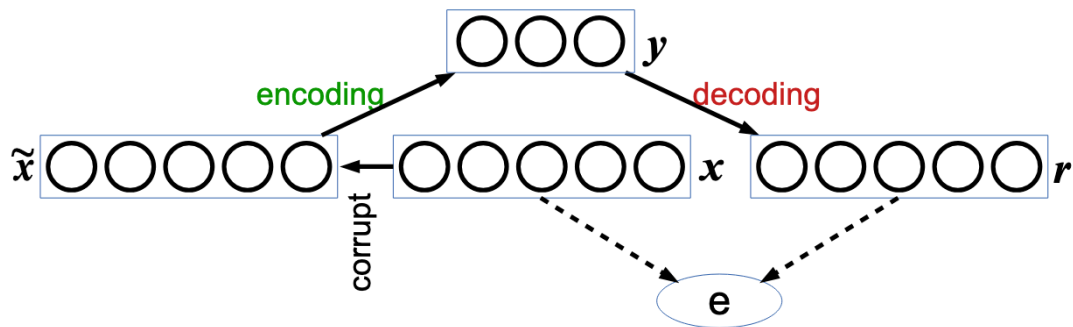
Typically, weights between the hidden and output layer equal corresponding weights between input and hidden layer, i.e. the **decoding weights** is equal to the transpose of the **encoding weights**.



Weights are defined to minimise reconstruction error. Then, the error can be measured using a number of metrics, e.g. Euclidean distance $e = \|x - r\|_2$. Finally, trained using backpropagation.



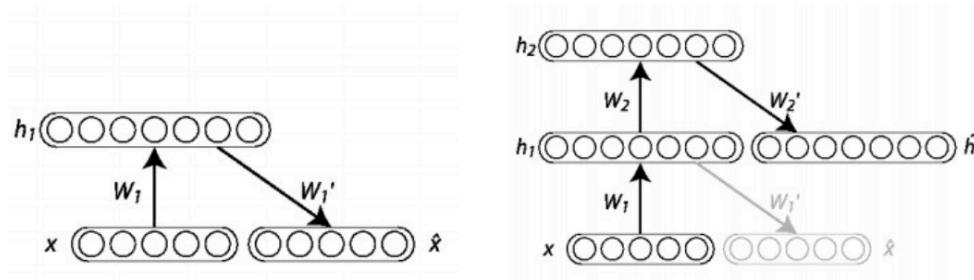
To avoid overfitting, **de-noising autoencoders**, add noise to inputs used for learning. As in, we encode with a corrupted input then perform the decoding with the original input.



Note

Stacked autoencoders form the basis for some deep learning architectures.

The **Stacked Restricted Boltzmann Machines (RBMs)** are called deep belief networks.



Deep architectures produce State-of-the-Art performance on many pattern recognition tasks.

4. Multi-layer Perceptrons and Backpropagation

4.1. Feedforward Neural Networks (MLPs)

A feed-forward neural network (multi-layer perceptron, MLP) consists of one input layer, some hidden layers, and one output layer

- Each layer is an array of neurons
- Layers interconnected by links
- Each link is associated with a connection weight

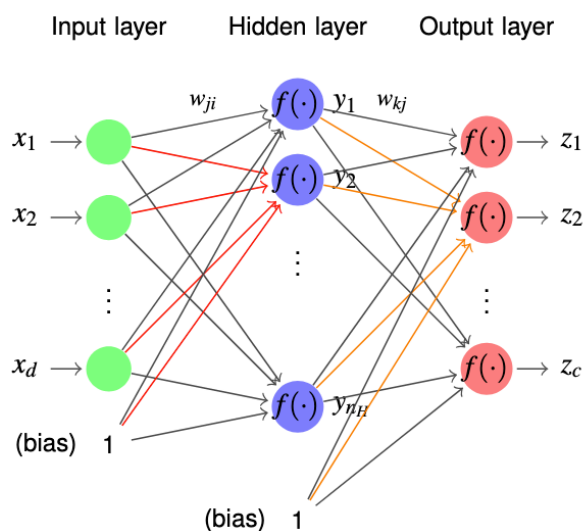


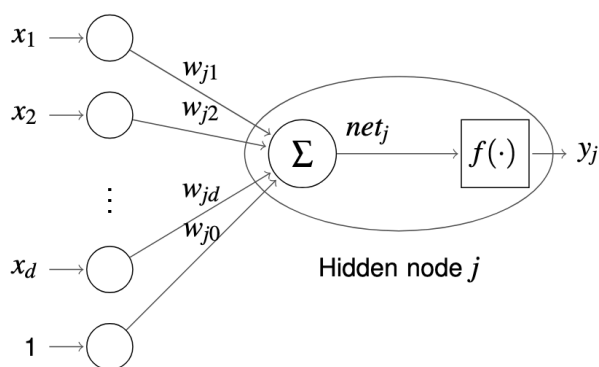
Figure 37: Three-Layer Fully-Connected FF-NN

4.1.1. Neuron Model

A single 'bias unit' is connected to each unit other than the input units.

$$net_j = x_1 w_{j1} + x_2 w_{j2} + \dots + x_D w_{jd} + w_{j0}$$

$$y_j = f(net_j)$$



Note

Okay, there was more here, but I cut it, why?

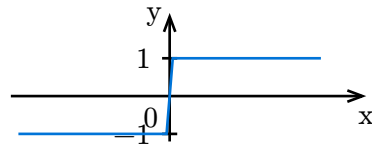
Because it's just repeating everything from the last section w/ different notation!

4.1.2. Activation/Transfer Functions

Diagram copied from **DM1**, but some are swapped out.

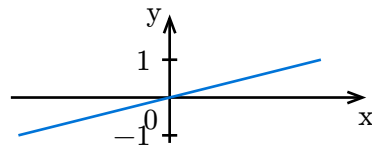
Sign function

$$g(s) = \begin{cases} 1, & s > 0 \\ -1, & s < 0 \end{cases}$$



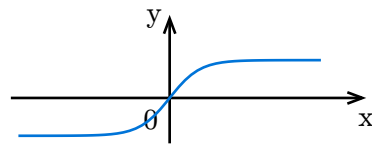
Linear transfer function

$$g(s) = s$$



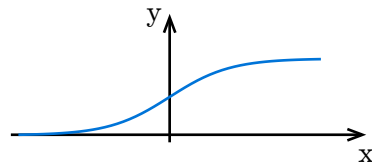
Symmetric Sigmoid function

$$g(s) = \frac{2}{1 + e^{-2n}} - 1$$



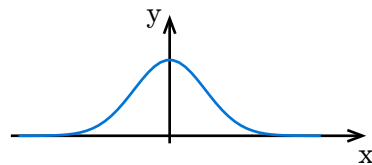
Logarithmic Sigmoid function

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

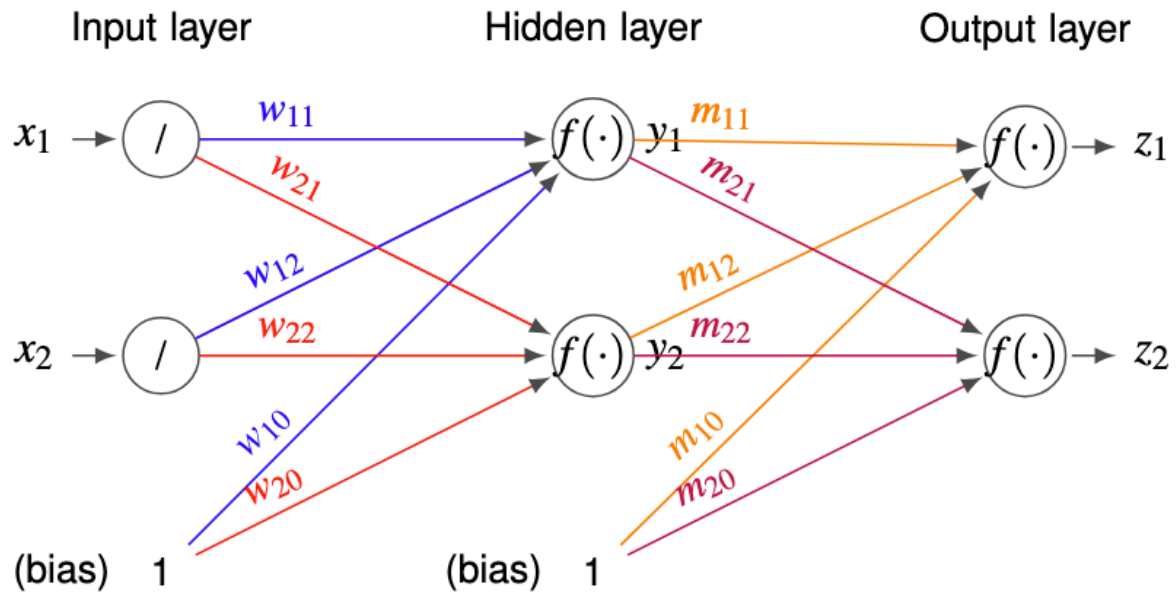


Radial basis

$$g(s) = e^{-s^2}$$



4.1.3. General Feed-Forward Operation



$$y_1 = f(w_{11}x_1 + w_{12}x_2 + w_{10}); \quad z_1 = f(m_{11}y_1 + m_{12}y_2 + m_{10})$$

$$y_2 = f(w_{21}x_1 + w_{22}x_2 + w_{20}); \quad z_2 = f(m_{21}y_1 + m_{22}y_2 + m_{20})$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = f \left(\underbrace{\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}}_{\mathbf{W}_{ji}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{\mathbf{x}} + \underbrace{\begin{bmatrix} w_{10} \\ w_{20} \end{bmatrix}}_{\mathbf{W}_{j0}} \right) \Rightarrow \mathbf{Y} = f(\mathbf{W}_{ji}\mathbf{x} + \mathbf{W}_{j0})$$

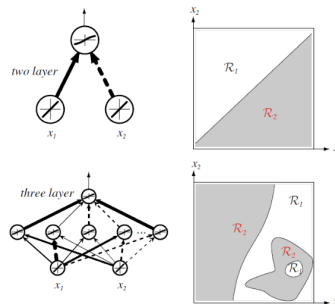
$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = f \left(\underbrace{\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}}_{\mathbf{W}_{kj}} \underbrace{\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}}_{\mathbf{Y}} + \underbrace{\begin{bmatrix} m_{10} \\ m_{20} \end{bmatrix}}_{\mathbf{W}_{k0}} \right) \Rightarrow \mathbf{Z} = f(\mathbf{W}_{kj}\mathbf{Y} + \mathbf{W}_{k0})$$

$$\mathbf{Z} = f(\mathbf{W}_{kj}\mathbf{Y} + \mathbf{W}_{k0}) = f(\mathbf{W}_{kj}(f(\mathbf{W}_{ji}\mathbf{x} + \mathbf{W}_{j0})) + \mathbf{W}_{k0})$$

4.1.4. Network Topology

Note

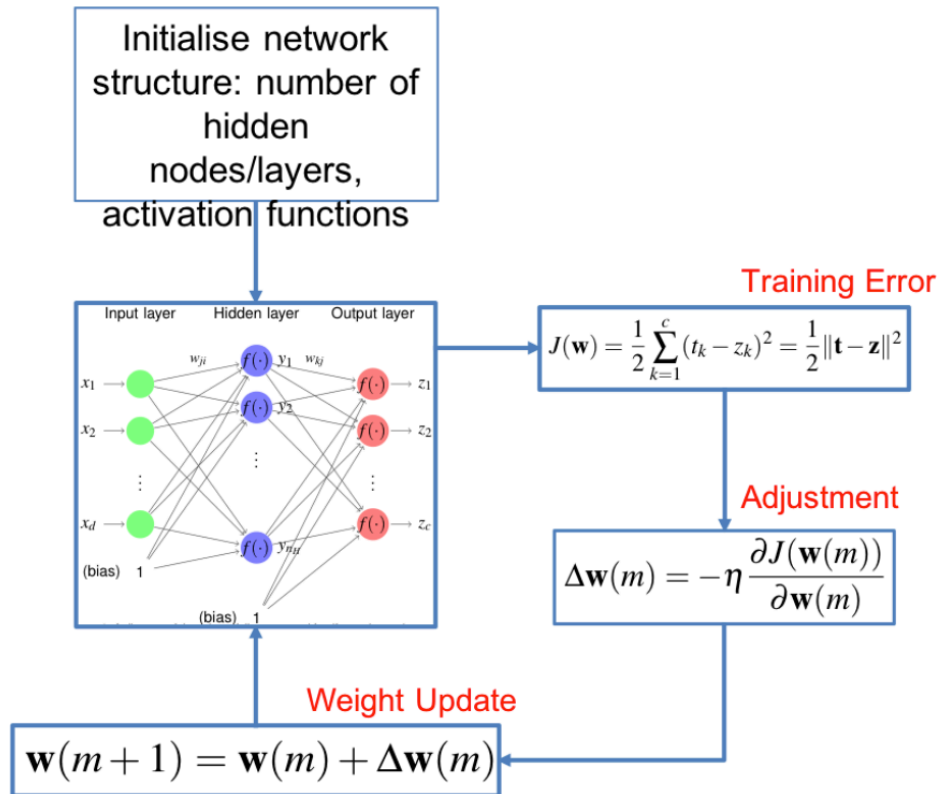
Any function from input to output can be implemented as a three-layer neural network, i.e. universal approximators. These results are generally more theoretically interesting than practical, since constructing the network requires knowledge of the non-linear functions and their weights.



4.2. Backpropagation Algorithm

The goal of **backpropagation** is to set the weights based on training data and desired output. It is one of the simplest and most general methods for supervised training of multilayer neural networks.

4.2.1. Network Learning



Training Error (at any instance)

$$J(\mathbf{w}) = \frac{1}{2} \sum_{k=1}^c (t_k - z_k)^2$$

$$= \frac{1}{2} \|\mathbf{t} - \mathbf{z}\|^2$$

Where t_k is the k -th desired/target output and z_k is the k -th computed output.
 $\mathbf{w}$ are the weights of the network and $\|\cdot\|$ is the Euclidean norm operator.

Weight Update (weights are changed in direction that reduces error)

$$\mathbf{w}(m+1) = \mathbf{w}(m) + \Delta \mathbf{w}(m)$$

$$\Delta \mathbf{w}(m) = -\eta \frac{\partial J(\mathbf{w}(m))}{\partial \mathbf{w}(m)}$$

Where m is the iteration number and $\eta > 0$ is the learning rate.

4.2.2. Hidden-to-Output Weights

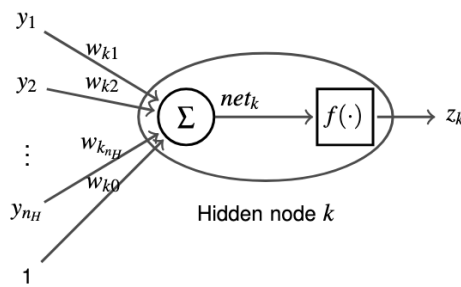
Error on the hidden-to-output weights (at m -th iteration)

$$\begin{aligned}\frac{\delta J}{\delta w_{kj}} &= \frac{\delta J}{\delta \text{net}_k} \frac{\delta \text{net}_k}{\delta w_{kj}} \\ &= -\delta_k \frac{\delta \text{net}_k}{\delta w_{kj}}\end{aligned}$$

Where the sensitivity of the unit k is defined as:

$$\delta_k = -\frac{\delta J}{\delta \text{net}_k}$$

Which describes how the overall error, i.e. J , changes with the activation function of the unit's net.



We know that:

$$\begin{aligned}\text{net}_k &= \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} \\ &= \sum_{j=0}^{n_H} y_j w_{kj} \\ &= \mathbf{w}_k^T \mathbf{y}\end{aligned}$$

$$\text{hence } \frac{\delta \text{net}_k}{\delta w_{kj}} = y_j$$

and $y_0 = 1$ for the bias

So the weight update for hidden-to-output weights is:

$$\begin{aligned}\Delta w_{kj} &= -\eta \frac{\delta J}{\delta w_{kj}} \\ &= \eta \cdot \delta_k \cdot y_j \\ &= \eta (t_k - z_k) f'(\text{net}_k) y_j\end{aligned}$$

Which means the new weight w_{kj} at the next iteration is given by:

$$\begin{aligned}w_{kj}(m+1) &= w_{kj}(m) + \Delta w_{kj}(m) \\ \text{where } j &= 0 \text{ (bias), } 1, \dots, n_H \\ k &= 1, \dots, c \\ y_0 &= 1\end{aligned}$$

4.2.3. Input-to-Hidden Weights

Error on the input-to-hidden units (at m -th iteration)

$$\frac{\delta J}{\delta w_{ji}} = \frac{\delta J}{\delta y_j} \frac{\delta y_j}{\delta \text{net}_j} \frac{\delta \text{net}_j}{\delta w_{ji}}$$

We know that:

$$\begin{aligned} J &= \frac{1}{2} \sum_{k=1}^c (t_k - z_k)^2 \\ y_j &= f(\text{net}_j) \\ \text{net}_j &= \sum_{i=1}^d x_i w_{ji} + w_{j0} \end{aligned}$$

So then:

$$\begin{aligned} \frac{\delta J}{\delta y_j} &= \sum_{k=1}^c \frac{\delta J}{\delta z_k} \frac{\delta z_k}{\delta y_j} \\ &= - \sum_{k=1}^c (t_k - z_k) \frac{\delta z_k}{\delta y_j} \\ &= - \sum_{k=1}^c (t_k - z_k) \frac{\delta z_k}{\delta \text{net}_k} \frac{\delta \text{net}_k}{\delta y_j} \\ &= - \sum_{k=1}^c \underbrace{(t_k - z_k) f'(\text{net}_k)}_{\delta_k} w_{kj} \\ \frac{\delta y_j}{\delta \text{net}_j} &= f'(\text{net}_j) \\ \frac{\delta \text{net}_j}{\delta w_{ji}} &= x_i \quad (x_0 = 1 \text{ for } w_{j0} \text{ (bias)}) \end{aligned}$$

Going back to the original equation, we find:

$$\frac{\delta J}{\delta w_{ji}} = - \underbrace{\sum_{k=1}^c \delta_k w_{kj} f'(\text{net}_j)}_{\delta_j} x_i$$

The sensitivity at a hidden unit, δ_k , is the sum of the individual sensitivities at the output units, weighted by the hidden-to-output weights w_{kj} ; all multiplied by $f'(\text{net}_j)$.

And so we can define our learning rule and new weight:

$$\begin{aligned} \Delta w_{ji} &= -\eta \frac{\delta J}{\delta w_{ji}} \\ &= \eta \cdot \delta_j \cdot x_i \\ &= \eta \cdot f'(\text{net}_j) \left[\sum_{k=1}^c w_{kj} \delta_k \right] x_i \\ w_{ji}(m+1) &= w_{ji}(m) + \Delta w_{ji}(m) \end{aligned}$$

4.2.4. Pseudo-Code

For the following, the **stopping criteria** is when the change in criterion function $J(\mathbf{w})$ is smaller than some preset value δ .

Algorithm 1: Stochastic Backpropagation

```
1 initialise network topology  $(d, n_H, c)$ ,  $\mathbf{w}$ , criterion  $\theta$ ,  $\eta$ 
2  $m \leftarrow 0$  – layers
3 while  $|\Delta J(\mathbf{w}(m))| = |J(\mathbf{w}(m)) - J(\mathbf{w}(m-1))| \geq \theta$  do
4    $\mathbf{x}^m \leftarrow$  randomly selected pattern
5    $w_{kj} \leftarrow w_{kj} + \eta \cdot \delta_k \cdot y_j$ 
6    $w_{ji} \leftarrow w_{ji} + \eta \cdot \delta_k \cdot x_i$ 
   Remark:  $\delta_j$  and  $\delta_k$  are calculated with old values of  $w_{ji}$  and  $w_{kj}$ 
```

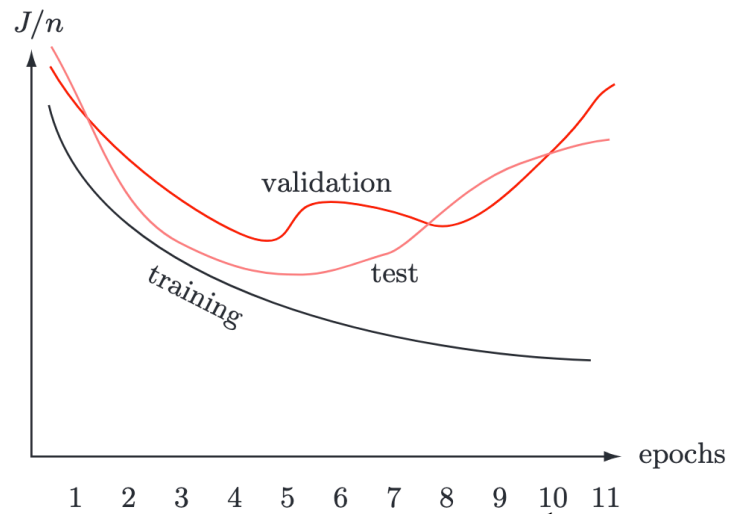
Algorithm 2: Batch Backpropagation

```
1 initialise network topology  $(d, n_H, c)$ ,  $\mathbf{w}$ , criterion  $\theta$ ,  $\eta$ 
2  $r \leftarrow 0$  – epochs
3 while  $|\Delta J(\mathbf{w}(r))| = |J(\mathbf{w}(r)) - J(\mathbf{w}(r-1))| \geq \theta$  do
4    $\Delta w_{kj} \leftarrow 0$ 
5    $\Delta w_{ji} \leftarrow 0$ 
6   for  $m \leftarrow 1..n$  do – for each pattern
7      $\mathbf{x}^m \leftarrow$  select the  $m^{\text{th}}$  pattern
8      $\Delta w_{kj} \leftarrow \Delta w_{kj} + \eta \cdot \delta_k \cdot y_j$  – accumulate for input patterns
9      $\Delta w_{ji} \leftarrow \Delta w_{ji} + \eta \cdot \delta_k \cdot x_i$  – accumulate for input patterns
10   $w_{kj} \leftarrow w_{kj} + \Delta w_{kj}$  – batch update
11   $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$  – batch update
```

4.2.5. Learning Curves

The dataset is partitioned into three sets, training, validation, test. As we train, error on the training set falls. Error per pattern depends on amount of training data and expressive power of the network.

Average error on independent test set is always higher than on training set, and can decrease as well as increase. A validation set is used in order to decide when to stop training, we do not want to overfit so we stop training at a minimum error on the validation set to avoid overfitting.



4.3. Radial Basis Function Neural Networks

A **radial basis function neural network** is a three-layer network that has three layers:

- input unit: linear transfer function
- hidden unit: radial basis function, $h(\cdot)$
- output unit: any activation function, $f(\cdot)$

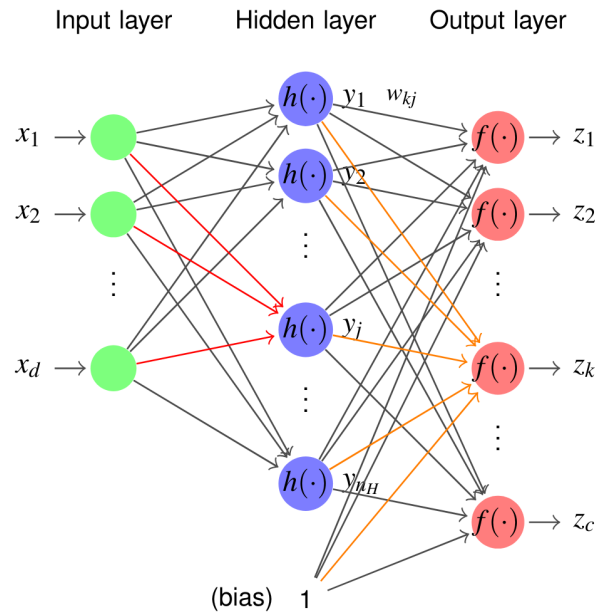
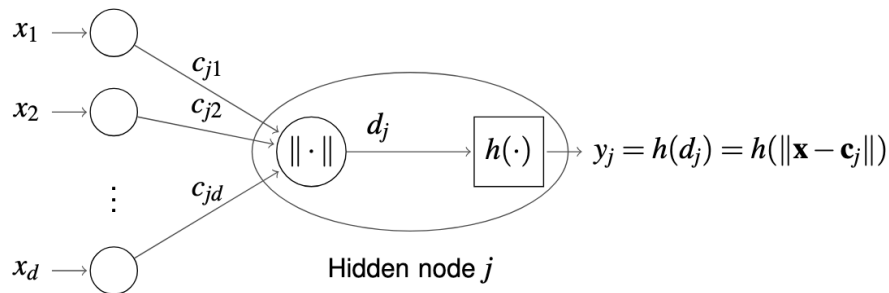


Figure 45: Example of a radial basis function neural network

In a radial basis function network, the hidden units define $d_j = \| \mathbf{x} - \mathbf{c}_j \|$ where $\mathbf{x}$ is the input vector and $\mathbf{c}_j$ is a vector representing the centre of the basis function. It may be preset or determined by a training algorithm.



Commonly used basis functions are specified below:

Function	Formula	Parameters
Gaussian function	$h(d_j) = e^{-\frac{d_j^2}{2\sigma^2}}$	$\sigma > 0$
Multi-quadric function	$h(d_j) = (d_j^2 + \sigma^2)^{\frac{1}{2}}$	$\sigma > 0$
Generalized multi-quadric function	$h(d_j) = (d_j^2 + \sigma^2)^\beta$	$\sigma > 0, 0 < \beta < 1$
Inverse multi-quadric function	$h(d_j) = (d_j^2 + \sigma^2)^{-\frac{1}{2}}$	$\sigma > 0$
Generalized inverse multi-quadric function	$h(d_j) = (d_j^2 + \sigma^2)^{-\beta}$	$\sigma > 0, \beta > 0$
Thin plate spline function	$h(d_j) = d_j^2 \ln(d_j)$	–
Cubic function	$h(d_j) = d_j^3$	–
Linear function	$h(d_j) = d_j$	–

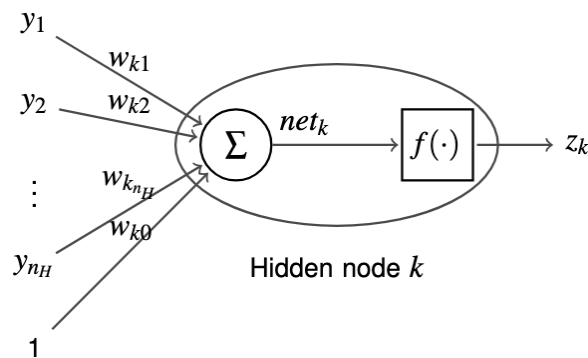
4.3.1. RBF Network Learning

Output at hidden units

$$\begin{aligned} y_{j(x)} &= h(d_j) \\ &= h(\| \mathbf{x} - \mathbf{c}_j \|), \quad j = 1, \dots, n_H \end{aligned}$$

Output at output units

$$\begin{aligned} z_{k(x)} &= f\left(\sum_{j=1}^{n_H} w_{kj} y_j(\mathbf{x}) + w_{k0}\right) \\ &= f\left(\sum_{j=1}^{n_H} w_{kj} h(\| \mathbf{x} - \mathbf{c}_j \|) + w_{k0}\right), \quad k = 1, \dots, c \end{aligned}$$



Our **training goal** is minimising the sum of squared errors:

$$\begin{aligned} J &= \sum_{p=1}^n J_p \\ &= \frac{1}{2} \sum_{p=1}^n \underbrace{\| \mathbf{t}_p - \mathbf{z}_p \|^2}_{J_p} \\ &= \frac{1}{2} \sum_{p=1}^n + (p=1) \underbrace{\sum_{k=1}^c (t_{pk} - z_{pk})^2}_{J_p} \end{aligned}$$

There are two phases of training:

- (unsupervised) determine centres $\mathbf{c}_j$ (and func. parameters)
 - fixed centres selected at random
 - or, clustering based approach
- (supervised) determine output weights w_{kj}
 - compute output weights using least squares
 - or, gradient descent/backpropagation

4.3.1.1. Determination of centres

Select at random

1. Choose number of hidden units, n_H
2. Pick randomly n_H input pattern from data set (with n input patterns) as centres c_j
3. When Gaussian function is employed, define $\sigma_j = \frac{\rho_{\max}}{\sqrt{2n_H}}$ or $\sigma_j = 2\rho_{\text{avg}}$ for all j where $\rho_{\max}$ is the maximum distance between the chosen centres and ρ_{avg} is the average distance between the chosen centres
4. Choose all other parameters randomly

Example initialisation

Given a dataset of 10 input samples: $x_1, x_2, \dots, x_{10}$

Choose 3 centres, i.e. $n_H = 3$, so $c_1 = x_5, c_2 = x_2, c_3 = x_9$

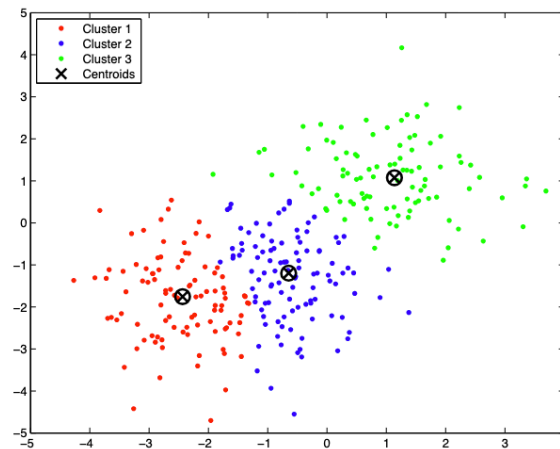
So then pick either:

$$\rho_{\max} = \max\{\|c_1 - c_2\|, \|c_1 - c_3\|, \|c_2 - c_3\|\}$$
$$\text{or, } \rho_{\text{avg}} = \frac{\|c_1 - c_2\| + \|c_1 - c_3\| + \|c_2 - c_3\|}{3}$$

Use K-Means Clustering

Centre c_j is the mean/centroid of the input patterns in cluster S_j .

n_H is determined prior.



4.3.1.2. Determination of output weights

Compute using least squares method

$f(\cdot)$ must be a linear function

$$z_k(\mathbf{x}) = \sum_{j=1}^{n_H} w_{kj} y_j(\mathbf{x}) + w_{k0}, \quad k = 1, \dots, c$$

$$\underbrace{\begin{bmatrix} y_1(\mathbf{x}_1) & y_2(\mathbf{x}_1) & \cdots & y_{n_H}(\mathbf{x}_1) & 1 \\ y_1(\mathbf{x}_2) & y_2(\mathbf{x}_2) & \cdots & y_{n_H}(\mathbf{x}_2) & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ y_1(\mathbf{x}_p) & y_2(\mathbf{x}_p) & \cdots & y_{n_H}(\mathbf{x}_p) & 1 \end{bmatrix}}_{\Phi(\mathbf{x})} \underbrace{\begin{bmatrix} w_{k1} \\ w_{k2} \\ \vdots \\ w_{kn_H} \\ w_{k0} \end{bmatrix}}_{\mathbf{W}_k} = \underbrace{\begin{bmatrix} t_{k1} \\ t_{k2} \\ \vdots \\ t_{kp} \end{bmatrix}}_{\mathbf{t}_k}, \quad k = 1, \dots, c$$

$$\Phi(\mathbf{x}) \mathbf{W}_k = \mathbf{t}_k, \quad k = 1, \dots, c$$

So now, if $\Phi(\mathbf{x})$ is a square matrix then:

$$\mathbf{W}_k = \Phi(\mathbf{x})^{-1} \mathbf{t}_k, \quad k = 1, \dots, c$$

Otherwise, if non-square matrix:

$$\mathbf{W}_k = (\Phi(\mathbf{x})^T \Phi(\mathbf{x}))^{-1} \Phi(\mathbf{x})^T \mathbf{t}_k, \quad k = 1, \dots, c$$

Compute using gradient descent/backpropagation

Can be applied to non-linear output function $f(\cdot)$

Parameters we are learning are w_{kj} , σ_j , and even c_j

Compute the following:

$$\begin{aligned} \Delta w_{kj} &= -\eta \frac{\delta J}{\delta w_{kj}} \\ \Delta \sigma_j &= -\eta \frac{\delta J}{\delta \sigma_j} \\ \Delta c_{ji} &= -\eta \frac{\delta J}{\delta c_{ji}} \end{aligned}$$

Then update parameters iteratively:

$$\begin{aligned} w_{kj}(m+1) &\leftarrow w_{kj}(m) + \Delta w_{kj}(m) \\ \sigma_j(m+1) &\leftarrow \sigma_j(m) + \Delta \sigma_j(m) \\ c_{ji}(m+1) &\leftarrow c_{ji}(m) + \Delta c_{ji}(m) \end{aligned}$$

One can use stochastic or batch updates here.

Use input normalisation if necessary.

4.3.2. Comparison of RBF and MLP Networks

Similarities

- Both are universal approximators
- Both are nonlinear feedforward neural networks

Differences

- RBF networks only have one hidden layer; MLP networks may have more than one
- RBF networks use different basis functions from activation functions used in MLP networks
- RBF networks compute *distance between input patterns & centres* while MLP networks compute the *inner product of input pattern & weights*.
- RBF is trained with two-phase algorithm but MLP networks are trained with a single-phase algo.

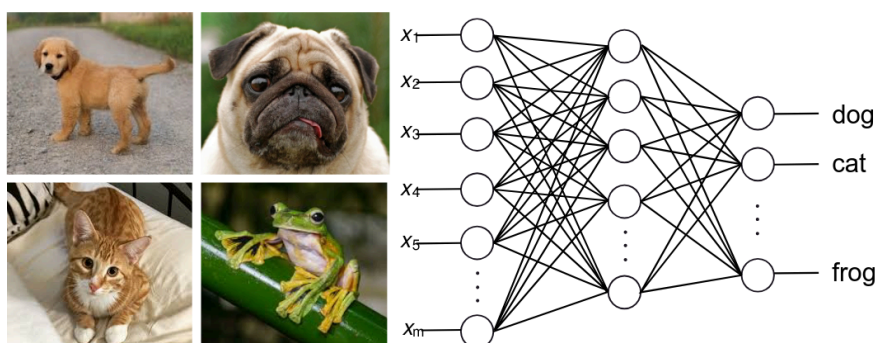
5. Deep Discriminative Neural Networks

A **deep network** is a neural network with > 3 layers (typically $\gg 3$); ‘deep learning’ refers to training one of these deep networks

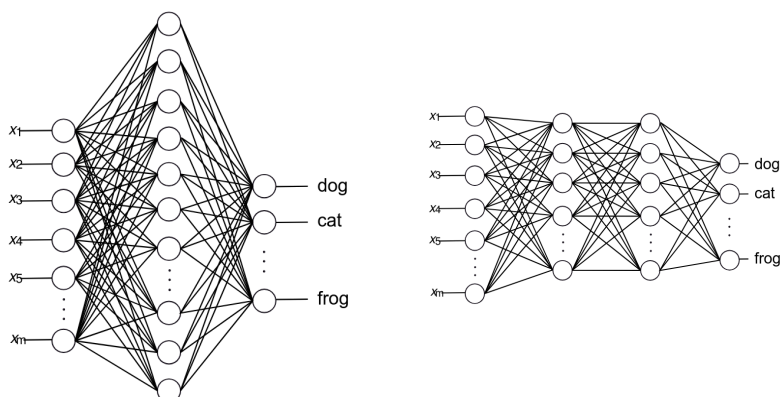
A **shallow network** is a neural network with ≤ 3 layers

Three-layer neural networks are already able to approximate any continuous non-linear function arbitrarily well (universal function approximators, Can solve any pattern recognition task, in theory).

But to solve more complex tasks (i.e. perform complex mappings), we need larger networks with much more parameters.



We could either:



Build a wide network

- large increase in parameters to achieve certain level of perf. on a task
- tends to memorise training input and not generalise

Build a deep network

- smaller increase required to achieve higher perf.
- aims to capture natural ‘hierarchy’ of task to improve generalisation

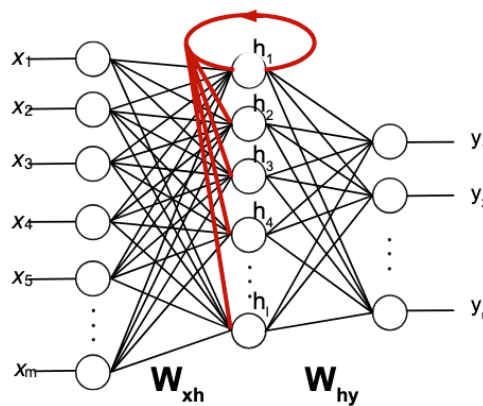
Introducing hierarchy is natural for dealing with many tasks:

- image recognition: pixel $\rightarrow$ edge $\rightarrow$ texton/contour $\rightarrow$ part $\rightarrow$ object
- text recognition: character $\rightarrow$ word $\rightarrow$ clause $\rightarrow$ sentence $\rightarrow$ story
- speech recognition: sound $\rightarrow$ phone $\rightarrow$ phoneme $\rightarrow$ word $\rightarrow$ clause $\rightarrow$ sentence

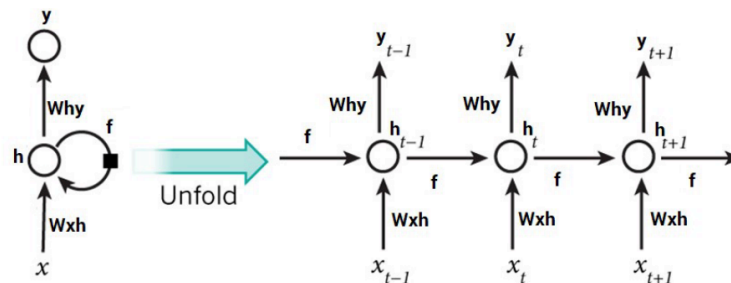
Deep learning methods aim to learn feature hierarchies with features from higher levels of the hierarchy formed by the composition of lower level features, i.e. build complicated concepts out of simpler ones.

Note

The need to build deep neural networks also arises from recurrent networks (these process temporal information, where x changes over time) where connections take output of every hidden neuron and feed it in as additional input at the next iteration.



Training a recurrent network is achieved by unfolding the network and then using backpropagation on the unfolded network. There is one layer for in the unfolded network for each time-step, so recurrent networks processing long sequences are equivalent to very deep networks.

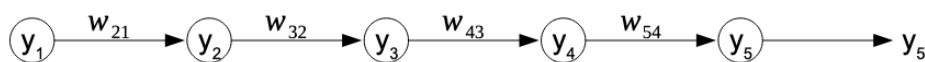


5.1. Difficulties Training Deep Neural Networks

Backpropagation is inherently unstable and gradients can vanish or explode easily. To train deep networks it is necessary to understand and mitigate these issues.

Vanishing gradient problem

Consider a deep network with one neuron per layer:



Apply backpropagation:

$$\Delta w_{54} = \eta(t - y_5) \delta'(w_{54} y_4) y_4$$

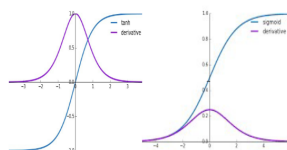
$$\Delta w_{54} = \eta(t - y_5) \delta'(w_{54} y_4) w_{54} \delta'(w_{43} y_3) y_3$$

$$\Delta w_{54} = \eta(t - y_5) \delta'(w_{54} y_4) w_{54} \delta'(w_{43} y_3) w_{43} \delta'(w_{32} y_2) y_2$$

Each time the error is propagated further backwards, it is multiplied by a factor of form $\delta'(w_{ji} x_i) w_{ji}$.

This means when:

- using standard activation function, tanh or sigmoid: derivative is small for most values of $w x$



- initialising weights from a distribution with mean zero: this term may be very small

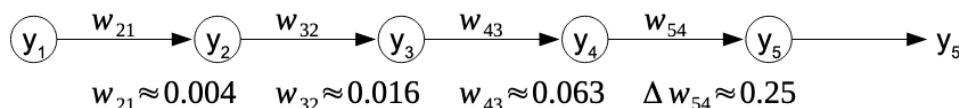


Figure 54: error being multiplied by $\delta'(w_{ji} x_i) w_{ji} \ll 1$

If we're not careful, the gradient will get smaller as we move through the hidden layers which means neurons in the earlier layers will learn much slower than the ones in the later layers. This means early layers contribute nothing to solving the task (keep initial weights) hence making the network deeper does not improve performance.

Exploding gradient problem

We can have the opposite problem where if weights are initialised to, or learn, large values, then $\delta'(w_{ji} x_i) w_{ji} > 1$ which means when the error is propagated backwards and exponentially grows.

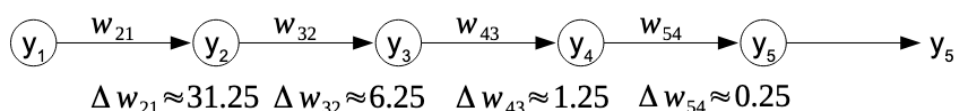
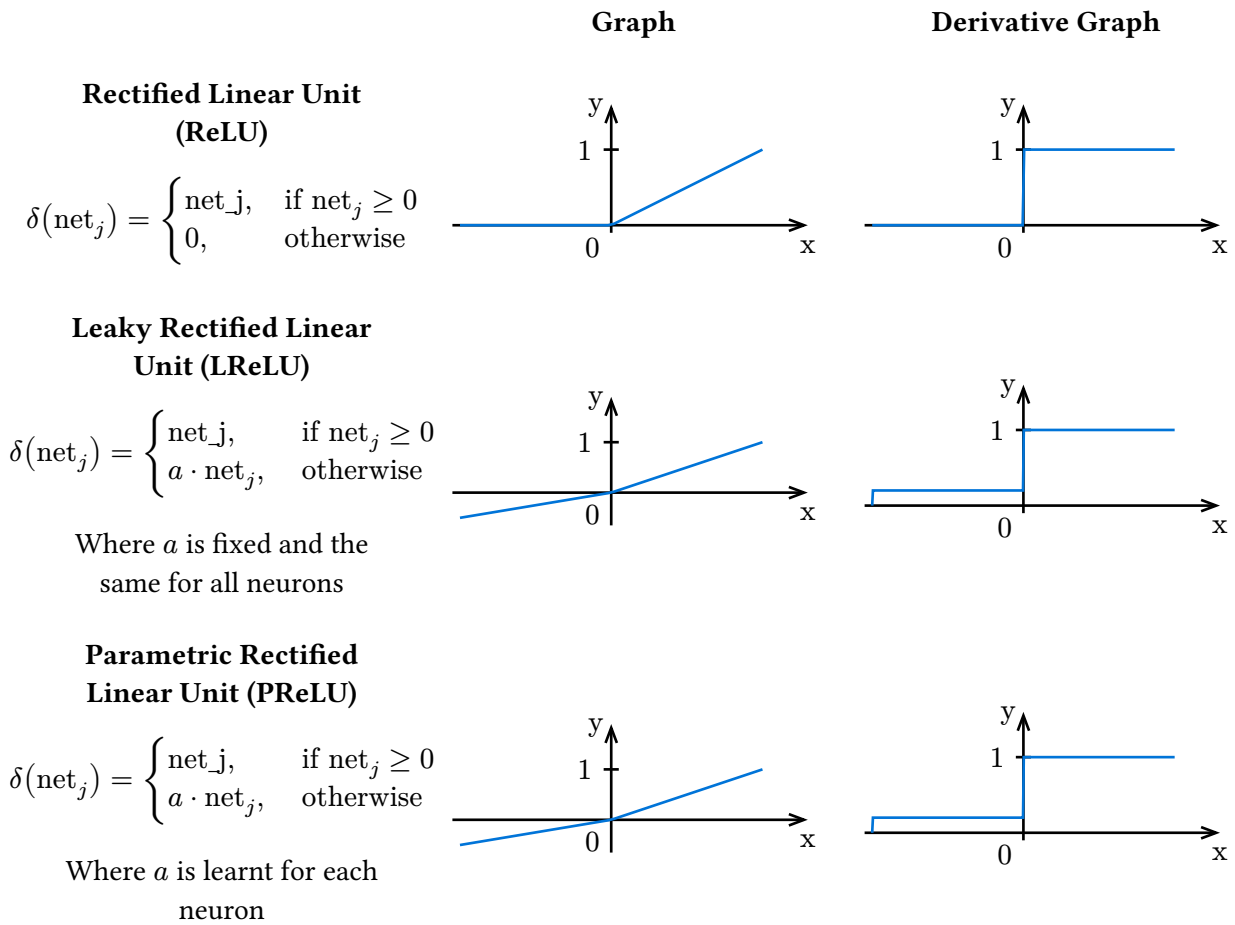


Figure 55: error being multiplied by $\delta'(w_{ji} x_i) w_{ji} > 1$

If we're not careful, the gradient will get larger as we move backwards through the hidden layers which means neurons in the earlier layers make, large, often random changes in weights. Later layers cannot learn due to constantly changing output of earlier layers hence making the network deeper does not improve performance.

5.1.1. Solutions

5.1.1.1. Activation functions with non-vanishing derivatives



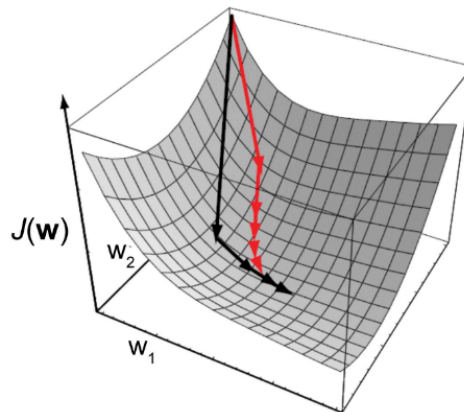
5.1.1.2. Better weight initialisation

For weights connecting m inputs to n outputs:

Method	Activation Function	Choose weights from	
		Uniform distribution range:	Normal distribution with $\mu = 0$ and standard deviation:
Xavier / Glorot	sigmoid / tanh	$\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$	$\sqrt{\frac{2}{m+n}}$
Kaiming He	(L/P)ReLU	$\left(-\sqrt{\frac{6}{m}}, \sqrt{\frac{6}{m}}\right)$	$\sqrt{\frac{2}{m}}$

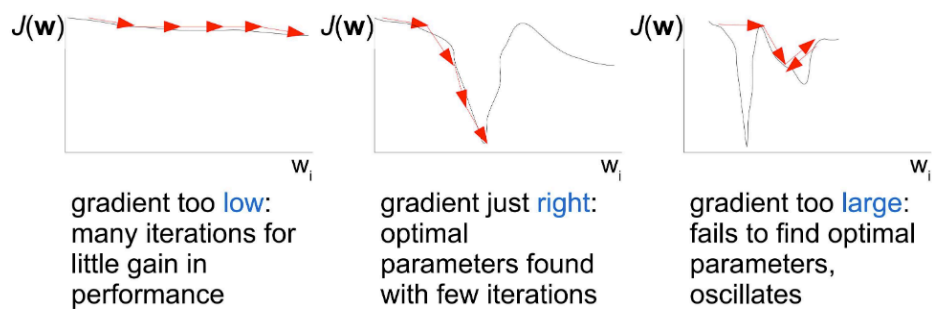
5.1.1.3. Adaptive variations on standard backpropagation

Backpropagation struggles to deal with gradients in the cost function $J(\mathbf{w})$ that are too small, or too large. Recall that we perform gradient descent to find parameters that minimise a cost.

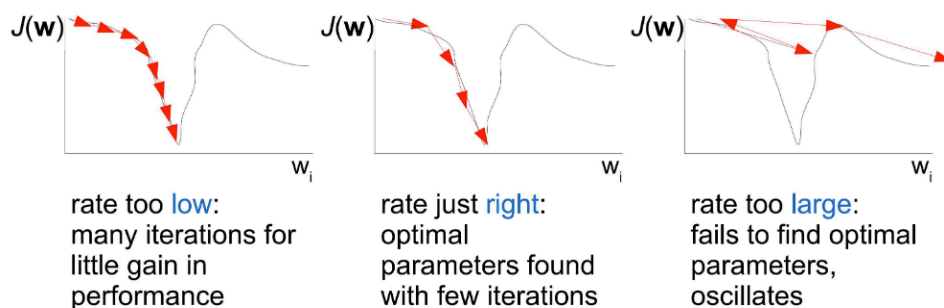


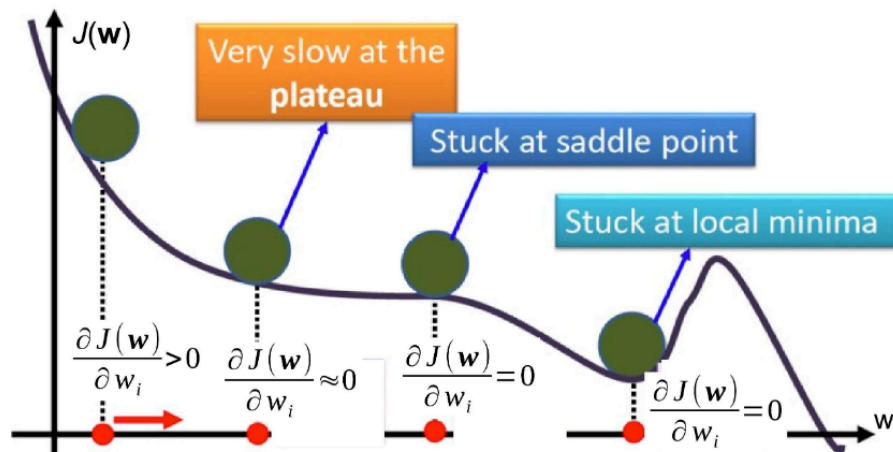
Variation in the magnitude of the gradient may occur between:

- different layers (vanishing/exploding gradient)
- different parts of a cost function for a single neuron
- different directions for a multi-dimensional function

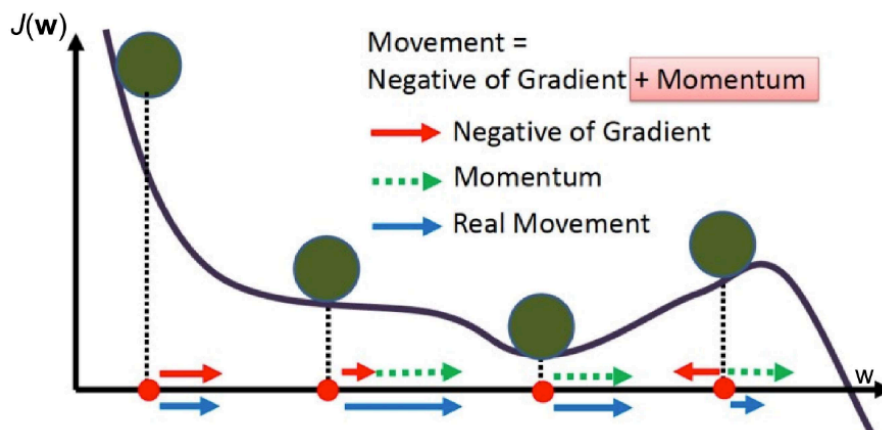


Similarly, for selecting an appropriate learning rate:

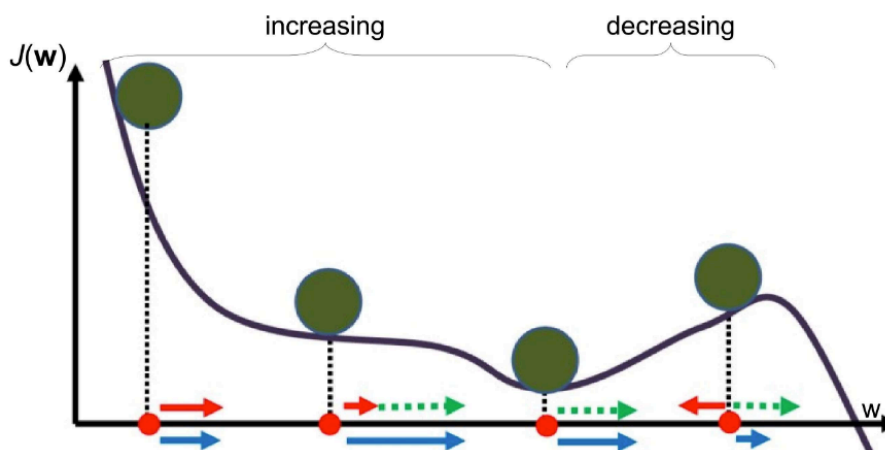




We could add **momentum** where we add moving average of previous gradient to current gradient, which increases step size when weight changes are consistently in same direction, helping eliminate plateaus and local minima.



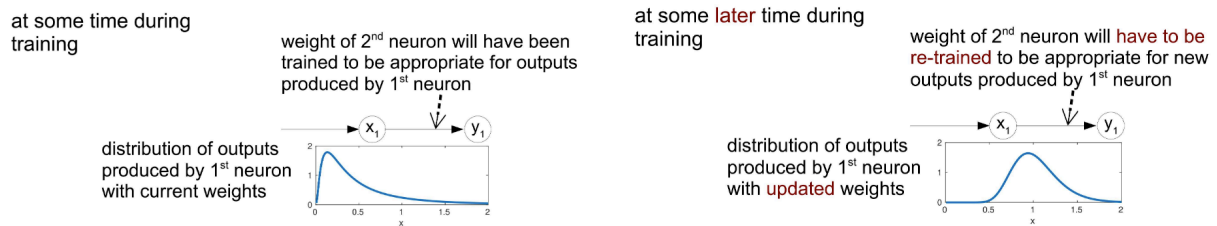
We could have an **adaptive learning rate** where we vary the learning rate (for individual parameters) during training, increasing if cost is decreasing, and vice-versa.



Backpropagation algorithms with adaptive learning are AdaGrad and RMSprop.
Backpropagation algorithms with adaptive learning and momentum are ADAM and Nadam.

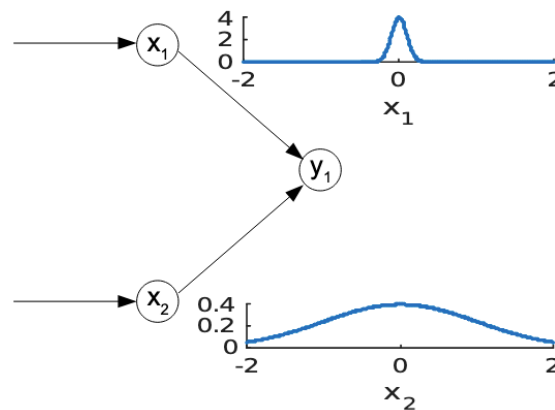
5.1.1.4. Batch normalisation

Learning in one layer of a network will change the distribution of inputs received by subsequent layers of the network:



Learning is slow as later layers are always having to compensate for changes made to earlier layers!

Different inputs to one neuron can have very different scales and inputs with smaller scales will tend to have less influence than ones with a larger scale (even if smaller scale inputs are more discriminative).



Batch normalisation aims to solve issues with large/small scale of input values by scaling the output of each individual neuron so that it has a mean close to 0 and a standard deviation close to 1.

$$\text{BN}(x) = \beta + \gamma \frac{x - \mathbb{E}(x)}{\sqrt{\text{Var}(x) + \varepsilon}}$$

Where $\mathbb{E}$ is the mean/expected value and $\text{Var}(x)$ is the variance, given by $\frac{1}{N} \sum_n (x_n - \mathbb{E}(x))^2$.

In batch normalisation, we learn β and γ by backpropagation, and set a constant ε to avoid divide-by-zero errors. $\mathbb{E}(x)$ (mean) and $\text{Var}(x)$ (variance) can be computed from values of x in current batch or using all training data presented thus far.

Batch normalisation can be applied before or after the activation function:

$$y_j = \delta(\text{BN}(w_{ji}x_i))$$

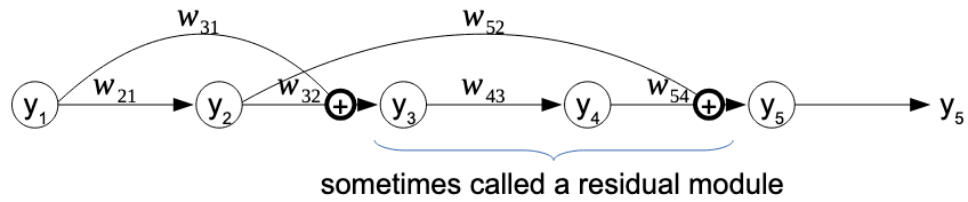
$$\text{or, } y_j = \text{BN}(\delta(w_{ji}x_i))$$

Batch normalisation enables saturating activation functions to be used because it limits activations to the range where gradients are non-zero (e.g. tanh and sigmoid).

Batch normalisation makes weight initialisation less critical and generally stabilises learning (less vanish/exp.).

5.1.1.5. Skip connections

Skip connections are those that skip one or more layers of the network



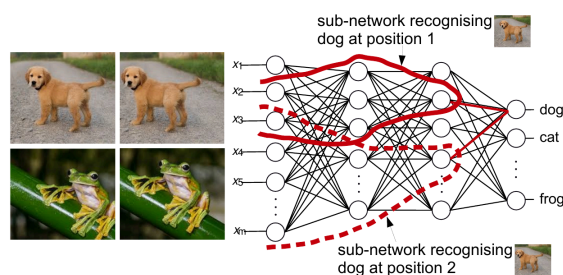
Skip connections let gradients bypass parts of the network where the gradient has vanished, network effectively becomes shallower (but this may be temporary, skips can be unlearned).

5.2. Convolutional Neural Networks (CNNs)

A **Convolutional Neural Network** is any neural network in which at least one layer has a transfer function implemented using convolution/cross-correlation (as opposed to vector multiplication)

Note

The idea of CNNs is motivated by the desire to recognise patterns with tolerance to position. Tolerance to location could be achieved by learning to recognise the pattern at each location independently:



It would be more computationally efficient to share the weights between the sub-networks (have multiple copies of the same sub-network processing different locations).

5.2.1. Convolutional Layers

Weight sharing is achieved using cross-correlation as the transfer function.

A neuron's weights are defined as an array (mask/filter/kernel).

The values (weights) are learnt using backpropagation.

-1	1
0	-2

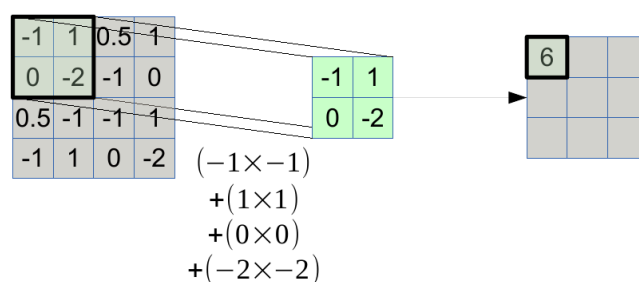
The input is also an array, e.g. an image or output of preceding layer.

-1	1	0.5	1
0	-2	-1	0
0.5	-1	-1	1
-1	1	0	-2

We can use 1D kernels for a 1D input, e.g. audio, or 3D kernels for a 3D input, e.g. video.

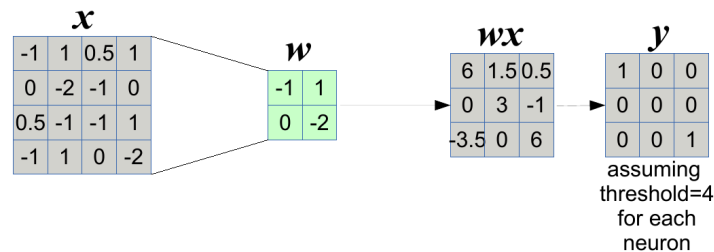
Cross-correlation works by, for each location in the input array:

1. Multiply each weight by corresponding value in the input
2. Sum these values and write into output array

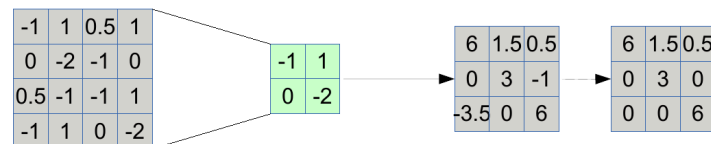


After using cross-correlation to implement the transfer function, the final output can be calculated by applying an activation function to each output (e.g. implement a layer of linear threshold units).

$$y_j = H \left(\underbrace{\sum_i w_{ji} x_i}_{\text{activation}} - \theta_j \right)$$



Example using ReLU as activation function:



A **feature map** is the output produced by a single mask (before or after application of the activation function, or pooling) in a convolutional layer of a CNN. It shows the responses produced by identical neurons at different spatial locations.

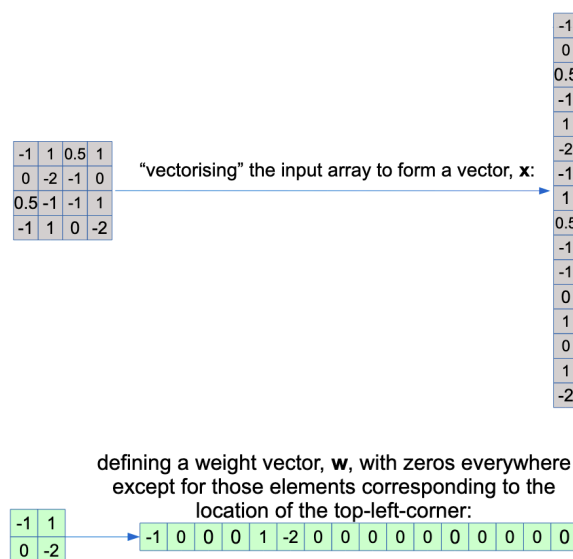
Some observations that indicate we have tolerance to location:

- Output is strongest where the location in the input matches weights in the kernel.
- Output is the same at the two locations in the input with the same values.

Note

Cross-correlation is equivalent to vector multiplication.

For example, we could vectorise the two vectors mentioned above:



Note

Despite being called ‘convolutional’ layers (and resulting networks, ‘convolutional’), the operation performed by conv. layers is cross-correlation/linear filtering and not convolution.

cross-correlation:

$$Y(i, j) = X \star H = \sum_{k, l} X(i+k, j+l) H(k, l)$$

-1	1
0	-2

convolution:

$$Y(i, j) = X * H = \sum_{k, l} X(i+k, j+l) H(-k, -l)$$

convolution same as cross-correlation with a rotated mask:

-2	0
1	-1

The mask defines the weights of a neuron, a neuron only receives input from a (small) region of the input array (where mask values are non-zero), this is the **receptive field**. Different masks have different RF sizes:

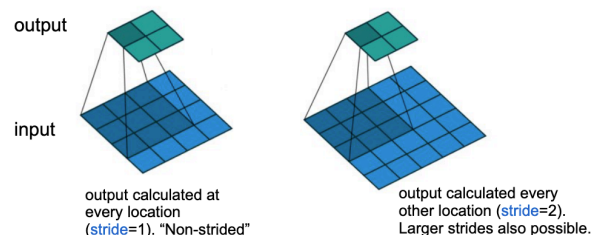
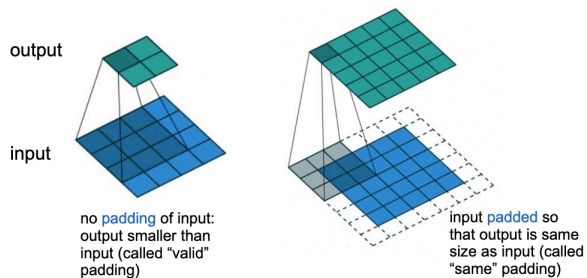
2x2 mask	<table><tr><td>-1</td><td>1</td></tr><tr><td>0</td><td>-2</td></tr></table>	-1	1	0	-2					
-1	1									
0	-2									
3x3 mask	<table><tr><td>-1</td><td>1</td><td>0</td></tr><tr><td>0</td><td>-2</td><td>-2</td></tr><tr><td>-1</td><td>1</td><td>0.1</td></tr></table>	-1	1	0	0	-2	-2	-1	1	0.1
-1	1	0								
0	-2	-2								
-1	1	0.1								
1x5 mask	<table><tr><td>1</td><td>0</td><td>-2</td><td>-2</td><td>-1</td></tr></table> (for 1D signal)	1	0	-2	-2	-1				
1	0	-2	-2	-1						

Cross-correlation generates ‘copies’ of the same neuron at each location; the output is the response of multiple, identical, neurons with the same weights but different RF locations.

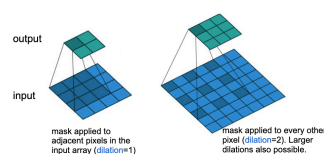
There are some variations on standard cross-correlation:

Padding defines the number of 0 valued pixels appended to each edge of the input, increasing padding increases size of output.

Stride defines the number of pixels moved between calculations of the output, increasing stride reduces size of output.

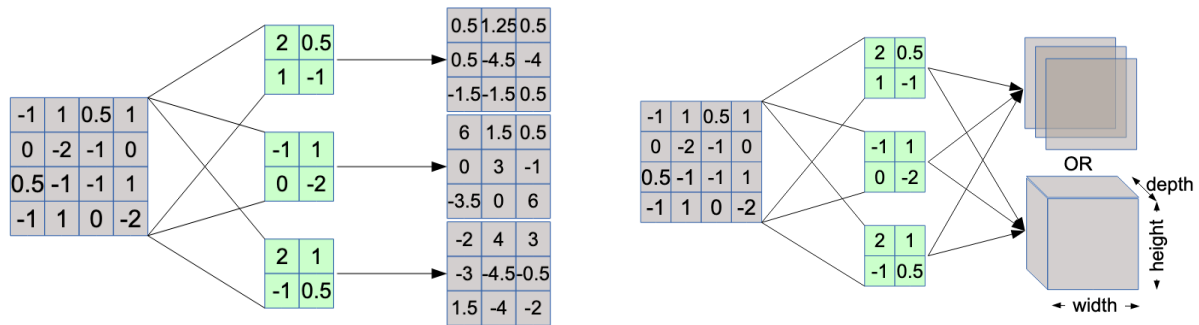


Dilation defines the spacing between the pixels to which the mask is applied. Increasing dilation increases RF size.

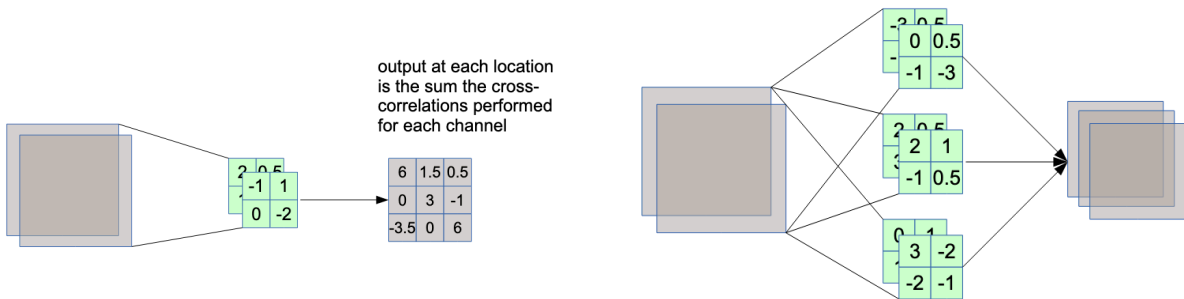


5.2.1 Convolutional Neural Networks (CNNs) > Convolutional Layers

We can have a *multi-channel* output if we provide multiple masks to identify multiple patterns: Often the output is identified as a stack of squares or a cuboid.



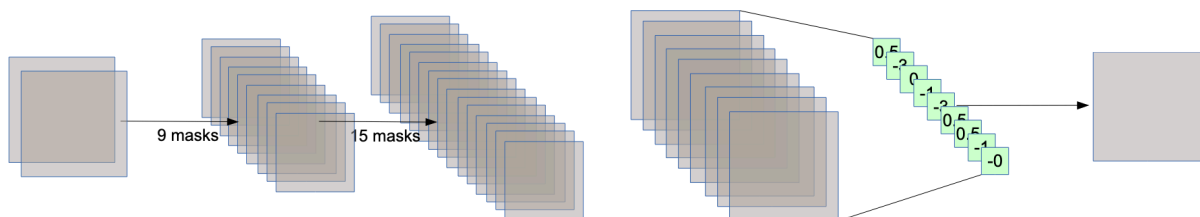
A neuron can have a *multi-channel* input (e.g. different colour channels, or a previous convolutional layer). Typically, the input, output and (multiple) masks are all multi-channel:



It may be necessary to use many masks to learn to recognise many different sub-patterns in the data. However, this may result in many channels in the input to the next layer. A **1x1 convolution** can be used to reduce the number of channels, and reduce the computational burden of the next layer.

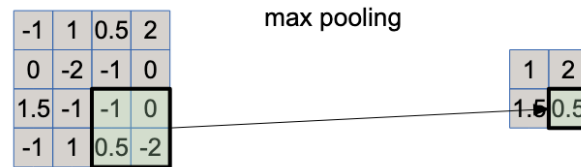
A 1x1 convolution:

- contains a single weight in each channel
- is always applied with stride=1, and no padding
- provides a weighted sum of input channels at each location
- multiple 1x1 filters can be used to produce differently weighted sums of input channels
- can also be used to increase no. of channels if many 1x1 filters are used



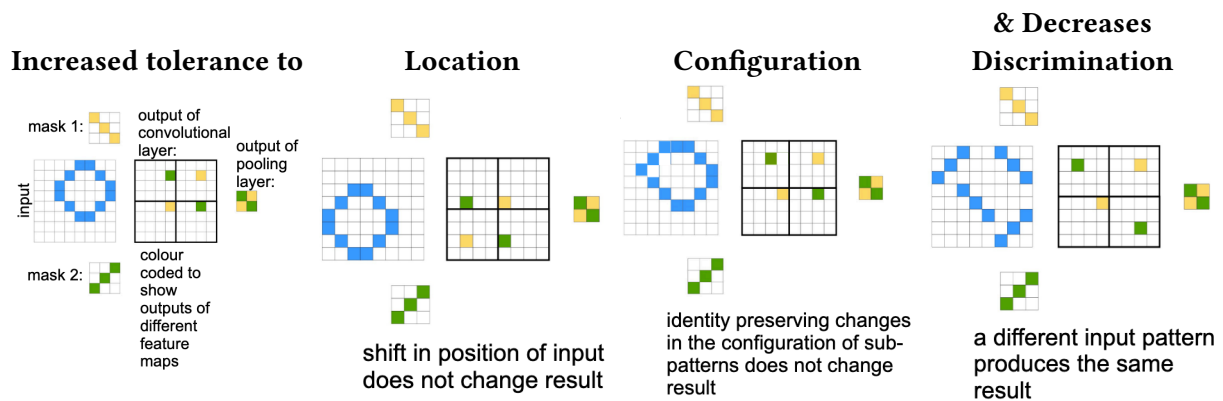
5.2.2. Pooling Layers

CNNs can also contain **pooling layers** to increase tolerance to the location of patterns and the configuration of sub-patterns. Pooling layers replace values within a region of the input array with a single value calculated, typically the mean ("average pooling") or maximum ("max pooling").



Some notes on pooling:

- typically has a stride>1 to decrease spatial size of layer
- different sized pooling regions (RFs) can be used
 - if it's the same size as the input array, output is a scalar value ('global pooling')
- pooling operation is user-specified, not learnt
- pooling is also called 'sub-sampling'/'down-sampling'

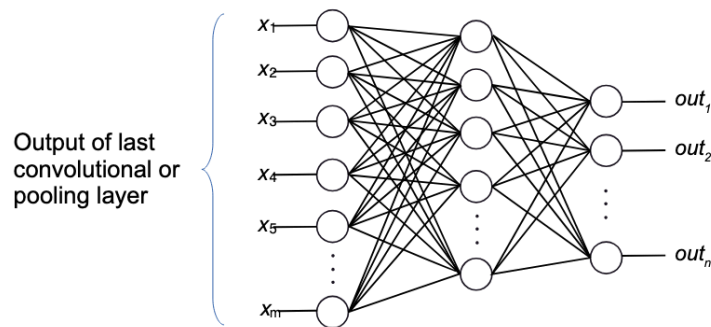


Note

Typically, pooling is applied separately to each input array so the number of channels is equal to the number of input channels, but it is possible to perform cross-channel pooling to reduce the resulting number of channels.

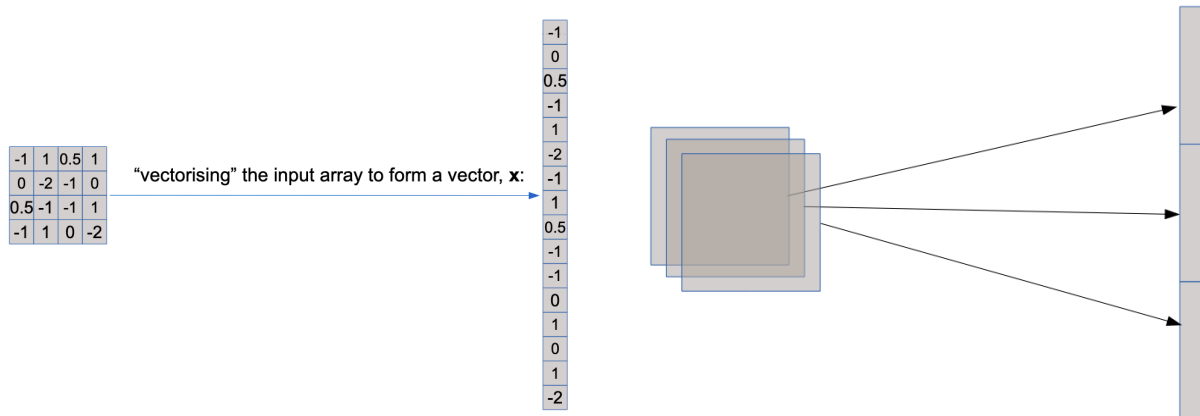
5.2.3. Fully-Connected Layers

Typically, particularly when applied to classification tasks, the final layers of a CNN are fully connected layers: they work exactly the same as a traditional, feed-forward NN (e.g. MLP).

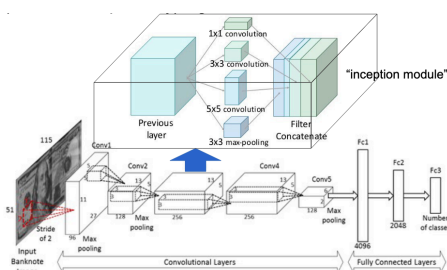
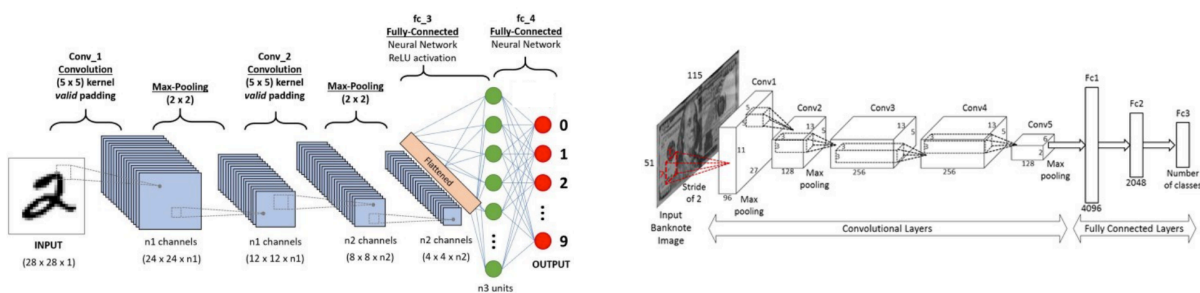


Classification is then performed using final layer of the fully connected layer.

To create the input into the 1st fully connected layer, it is necessary to 'flatten' the output of the last convolutional/pooling layer. This is the same as vectorisation. Also, the vectors from each feature map are concatenated into one vector.



The final network may contain multiple convolutional/pooling and fully connected layers connected sequentially, for example:



It's also possible for operations to be performed in parallel.

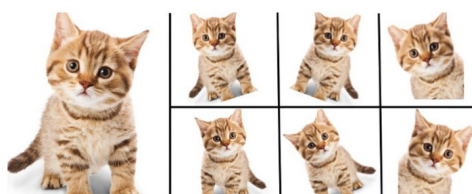
5.3. Issues with CNNs/Deep Networks

INCOMPLETE slides 37-41 TODO-CARDS

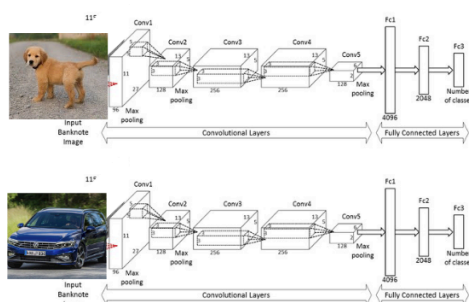
Deep NNs/CNNs have lots of tunable parameters (typically 4 – 140+ million for classifying colour images); there is a trend to increasing depth and complexity. Hence:

1. We need lots of training data
2. Training is computationally intensive
3. Danger of over-fitting the training data

Data Augmentation can tackle need for lots of training data by applying class-preserving transformations to the image (shifts, rotations, scaling, flips, shears, crops, ...)

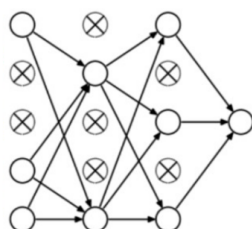


Transfer Learning can be used to address the need for lots of data and the computationally intensive nature of training. We can first train a network on a related task with lots of available data or select a pre-trained network from the internet, and then train the network further with little data or fewer epochs.

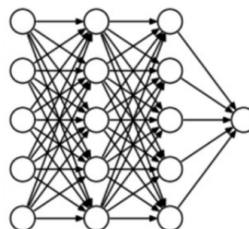


We can use **regularisation** where we modify the learning algorithm to improve generalisation. One of the most effective methods is dropout: force random sub-networks to correctly classify each sample preventing recognition of individual samples being reliant on individual neurons.

During training, at each iteration,
randomly select a fraction of neurons
and set their activity to be zero.



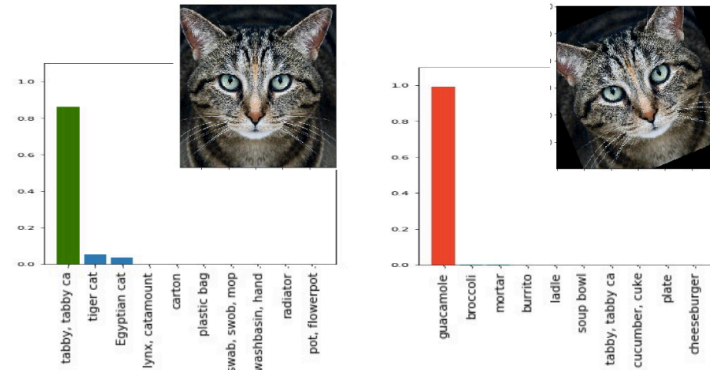
During usage, all neurons
behave normally.



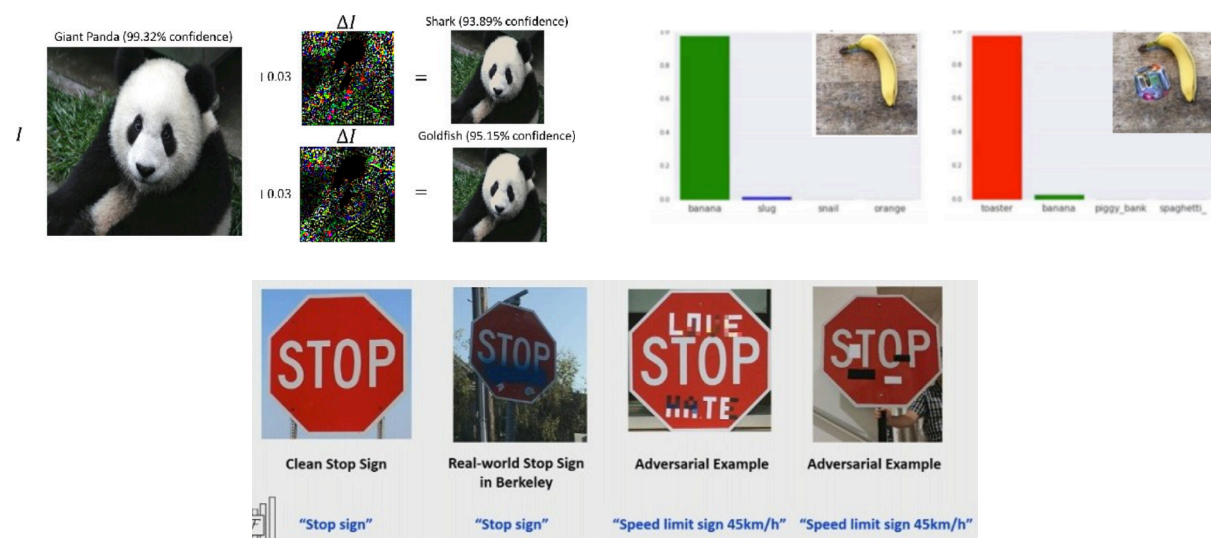
5.3 Deep Discriminative Neural Networks > Issues with CNNs/Deep Networks

Even if all precautions are taken against over-fitting, the network may still learn to exploit unintended characteristics such as certain lighting conditions, smudges on lens, certain contexts, dead pixels, etc.

Deep NNs can produce excellent results on testing data set but still fail to generalise.

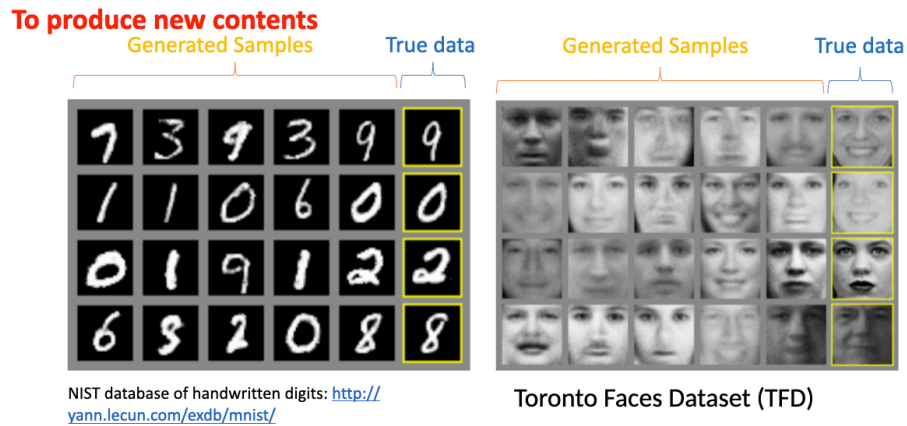


This is a particular problem as it makes it easy for someone to manipulate an image to be misclassified.

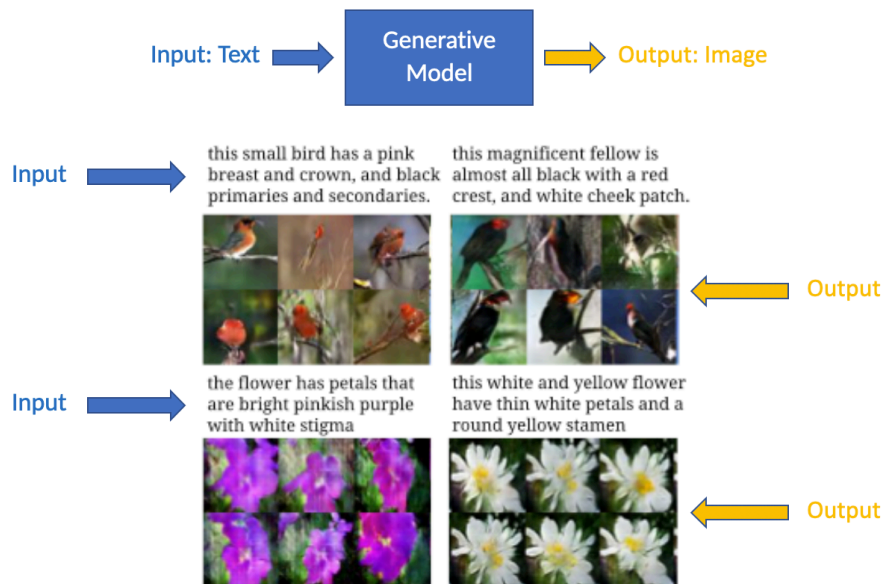


6. Deep Generative Neural Networks

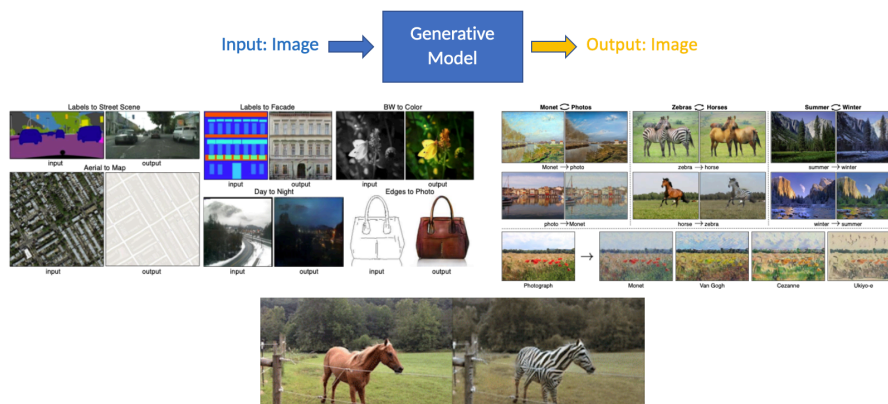
GANs are generative models that can produce new contents:



- text-to-image generation

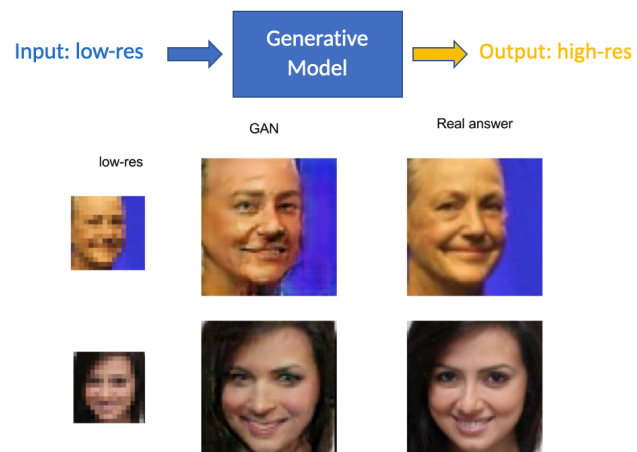


- image-to-image translation

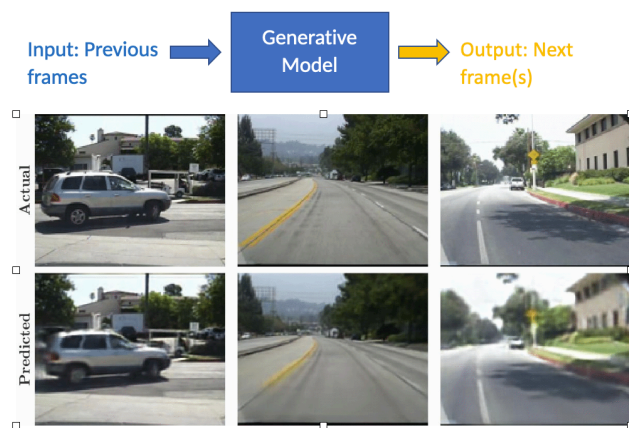


6 Deep Generative Neural Networks

- increasing image resolution

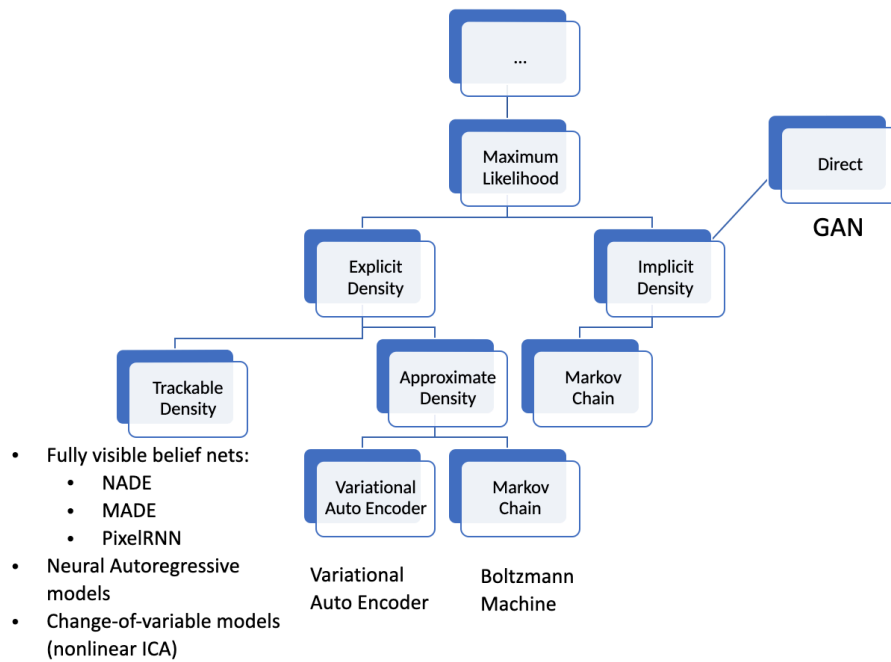


- predicting next video frame



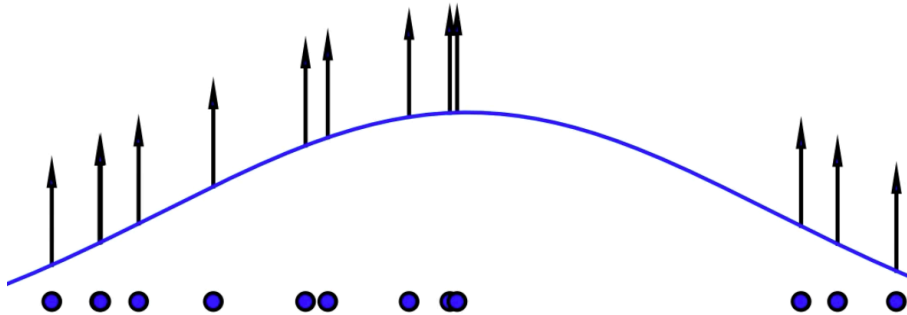
- ..., music generation, face generation

6.1. Generative Model Taxonomy



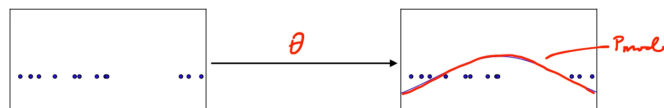
Maximum likelihood is given by:

$$\theta^* = \arg_{\theta} \max \mathbb{E}_{x \sim p_{\text{data}}} \log p_{\text{model}}(x | \theta)$$

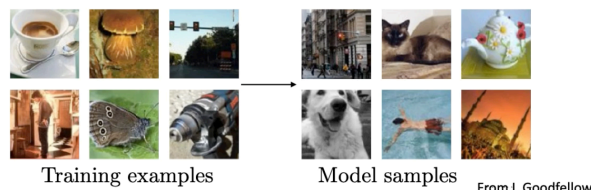


We want to find θ parameter which allows to best approximate the distribution.

Density Estimation



Sample Generation



6.1.1. Explicit Model with Trackable Density: PixelRNN

PixelRNN is a recurrent neural network model that uses trackable density (which in turns uses explicit density $\rightarrow$ maximum likelihood). An explicit formula is needed.



PixelCNN elephants
(van den Ord et al 2016)

We need a probability density model for each pixel, i , given by $p_{\text{model}}(x_i)$.

The explicit model is therefore based on a chain:

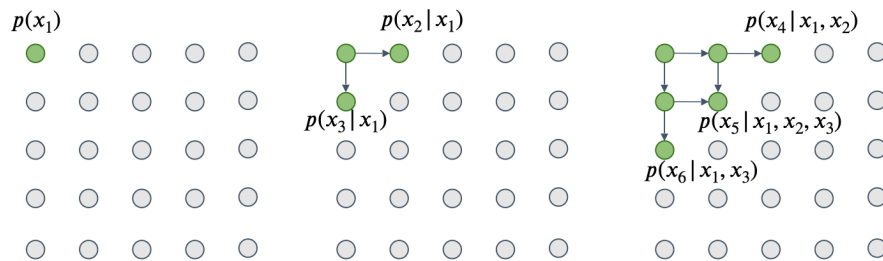
$$p_{\text{model}}(\mathbf{x}) = p_{\text{model}}(x_1) \prod_{i=2}^n p_{\text{model}}(x_i \mid x_1, \dots, x_{i-1})$$

PixelRNN has two main disadvantages:

- Propagation is needed to generate a sample
- Generation is not controlled by a latent code (a feature of an image)

Demonstration

$$\underbrace{p(\mathbf{x})}_{\text{likelihood of image } \mathbf{x}} = \prod_{i=1}^n \underbrace{p\left(x_i \mid \overbrace{x_1, \dots, x_{i-1}}^{\text{does not incl. } x_i}\right)}_{\substack{\text{probability of } i\text{-th pixel value} \\ \text{given all previous pixels}}}$$



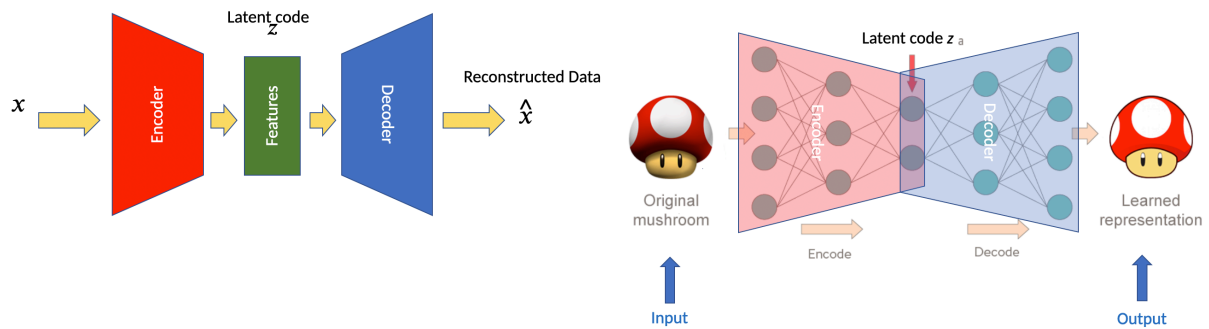
6.1.2. Explicit Model with Approx. Density: Variational Autoencoder (VAE)

Variational Autoencoder (VAE) is an explicit model that tries to approximate the density.

$$p_{\theta}(x) = \int p_0(z)p_0(x|z).dz$$

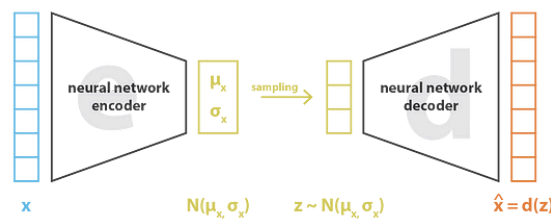
Where $p_{\theta}(x)$ is the density function/model, z is the latent code, and θ is the model parameter vector. So $\theta = [\theta_1, \theta_2, \dots, \theta_n]$. This is an intractable density function so we cannot optimise it directly, we must optimise the lower bound of the likelihood instead.

With an autoencoder, when we input some image x , we generate latent code Z , and then the decoder reconstructs $\hat{x}$, we try to minimise the error in training, for example with the L2 loss function which is given by $\min_{\theta} \sum_{i=1}^N \|x_i - \hat{x}_i\|^2$:

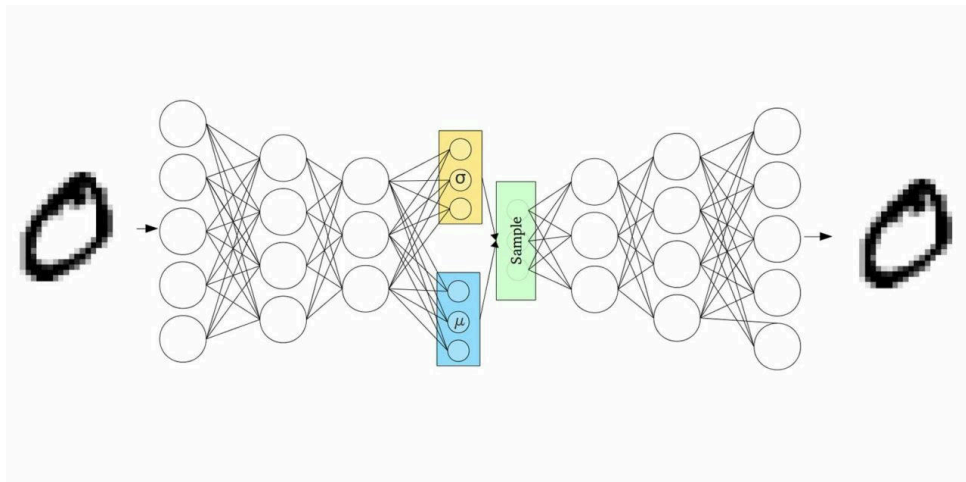


In a VAE, instead of learning the latent code, we learn the data distribution instead.

The parameter layers, μ and σ , capture this information, then we take a **sample** of data as given by the distribution function which the decoder then tries to reconstruct the image from.



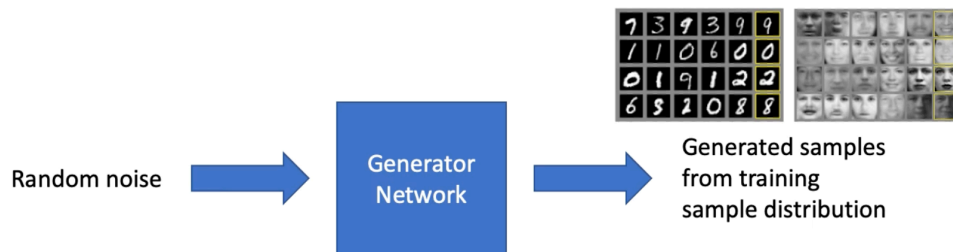
$$\text{loss} = \|x - \hat{x}\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)] = \|x - d(z)\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)]$$



6.2. Generative Adversial Networks (GANs)

A **Generative Adversial Network (GAN)** is a model where we learn the implicit density function.

Our problem is that we want to sample from complex and high-dimensional training sample distribution. A solution is to sample from a simple distribution such as random noise, and transform it into a training distribution (by a generator network).

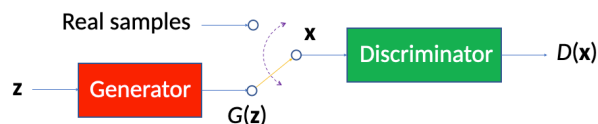


A simple idea is to generate sample x using latent code z however we have no guidance to train the generator (we cannot give a label to random noise) and random code will generate output noise.



Instead, we use the **two-player game** structure (Actor-Critic approach) consisting of two elements:

- generator: maps random noise z to produce a meaningful sample
If not well trained, produced sample is a noise
- discriminator: to decide if the sample x is real or produced from generator (fake)



When working with GANs, we have the following data:

- Real sample: $x \sim p_{\text{data}}(x), x \in X$
- Latent code: $z \sim p_z(z), z \in Z$
 $p_z(z)$ can be a uniform or Gaussian distribution

The generator $G(z, \theta_g)$ denotes the generator's model and maps the Z space to X space, where θ_g is the parameters of the generator. It could be a NN with weights θ_g or an expression with params. We define implicitly, a distribution $p_g(x, \theta_g)$.

The discriminator $D(x, \theta_d) \in [0, 1]$ returns a value where 1 denotes a real sample and 0 denotes a fake sample, it can be a range between these two values, $X \rightarrow \mathbb{R}$. θ_d denotes the parameters of the discriminator.

The **cost/objective/loss function (cross entropy)**, where $\mathbb{E}(\cdot)$ is the expectation operator:

$$V(D, G) = \underbrace{\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})]}_{\text{for real samples}} + \underbrace{\mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]}_{\text{for fake samples}}$$

① Note

- $\log(\cdot)$ within this GAN topic will refer to natural logarithm, $\ln(\cdot)$; other bases can be used
- minibatch can be used in training, i.e. calculating gradient of $V(D, G)$

Training proceeds as a minimax game where we repeat: ($V(D, G)$ - cost function)

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]$$

- Maximise $V(D, G)$ with respect to D by adjusting parameters θ_d (can repeat a few times)
- Minimise $V(D, G)$ with respect to G by adjusting parameters θ_g

Property of log function

$$\log(a) = \begin{cases} 0, & a = 1 \\ (-\infty, 0), & a \in (0, 1) \\ -\infty, & a = 0 \end{cases}$$

Our training objectives are:

- discriminator: maximise chance to recognise real samples as real; generated samples as fake

$$\max_D V(D, G) = \underbrace{\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})]}_{\text{recognise better for real samples}} + \underbrace{\mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]}_{\text{recognise better for generated samples}}$$

- generator: minimise the chance generated samples are recognised as fake

$$\min_G V(D, G) = \underbrace{\mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]}_{\text{make the generated samples to fool the discriminator}}$$

Algorithm 3: Minibatch stochastic gradient descent training of GANs.

 $k \leftarrow$ number of steps to apply to discriminator (hyper-parameter)1 **for** *no. of training iterations* **do**2 **for** *k steps* **do**3 sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$ 4 sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(z)$

update the generator by its descending stochastic gradient:

5
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\log(D(x^{(i)})) + \log(1 - D(G(z^{(i)})))]$$

6 sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$

update the generator by its ascending stochastic gradient:

7
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log(1 - D(G(z^{(i)})))$$

remark: the gradient-based updates can use any standard gradient-based learning rule

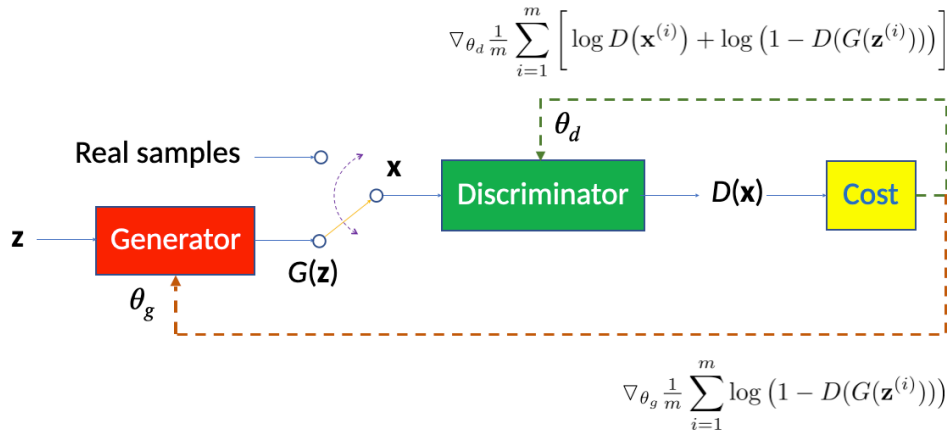
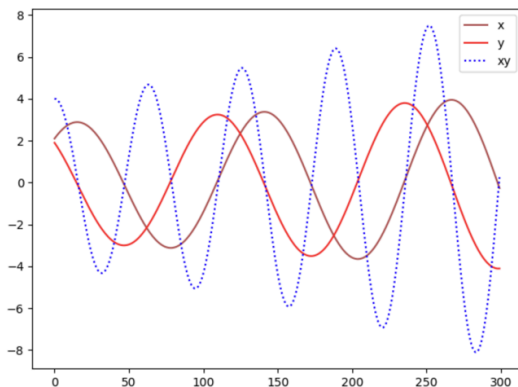


Figure 83: Training via backpropagation

6.3. Problems with GANs

GANs exhibit several problems:

- *Non-convergence*: oscillation of model parameters
- *Mode collapse*: generator collapses causing the generation of limited varieties of samples
- *Diminished gradient (gradient saturation)*: generator gradient vanishes and does not learn
- *Imbalance between generator and discriminator*: causing overfitting (e.g. discriminator is too good)
- *High sensitivity to hyper parameters*: learning rate, net. parameters can cause issue



$$\min_y \max_x V = xy$$

gradients are $(\delta xy)(\delta x) = y$

$(\delta xy)(\delta y) = x$

updates are $x + \eta(\delta xy)(\delta x) = x + \eta x$

$x - \eta(\delta xy)(\delta x) = x - \eta x$

where $\eta > 0$ denotes learning rate

Figure 84: Non-converging 'chasing' game of two players
Discriminator vs. generator (zero-sum game: one player wins/loses)

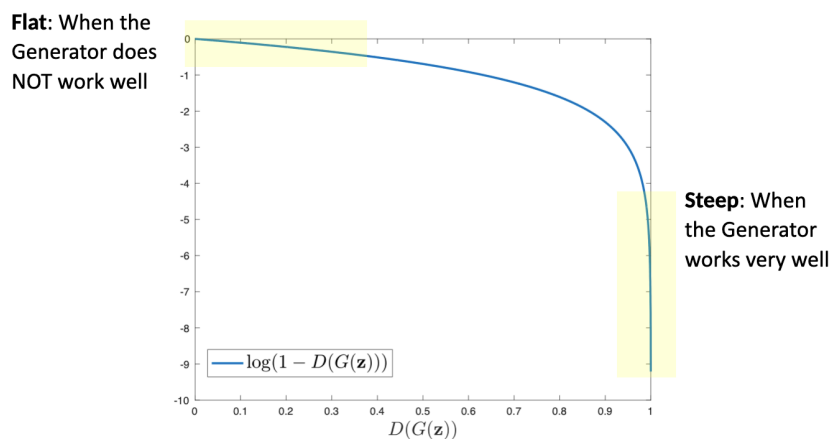


Figure 85: Diminished gradient: $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log(1 - D(G(\mathbf{z}^{(i)})))$

6.4. Improving training of GANs

6.4.1. Network Design

We could use a better network design such as **DCGAN**¹ (deep convolutional generative adversarial networks) where:

- we replace any pooling layers with strided convolutions (discriminator) and fractional (generator); transposed convolution used for upsampling
- batch normalisation is used in both generator and discriminator
- remove fully connected hidden layers for deeper architectures
- use ReLU activation in generator for all layers except output (tanh)
- use leakyReLU activation in discriminator for all layers

Note

You definitely do not need to know these details for the exam, lecturer just skimmed over this slide and said to read the paper for details.

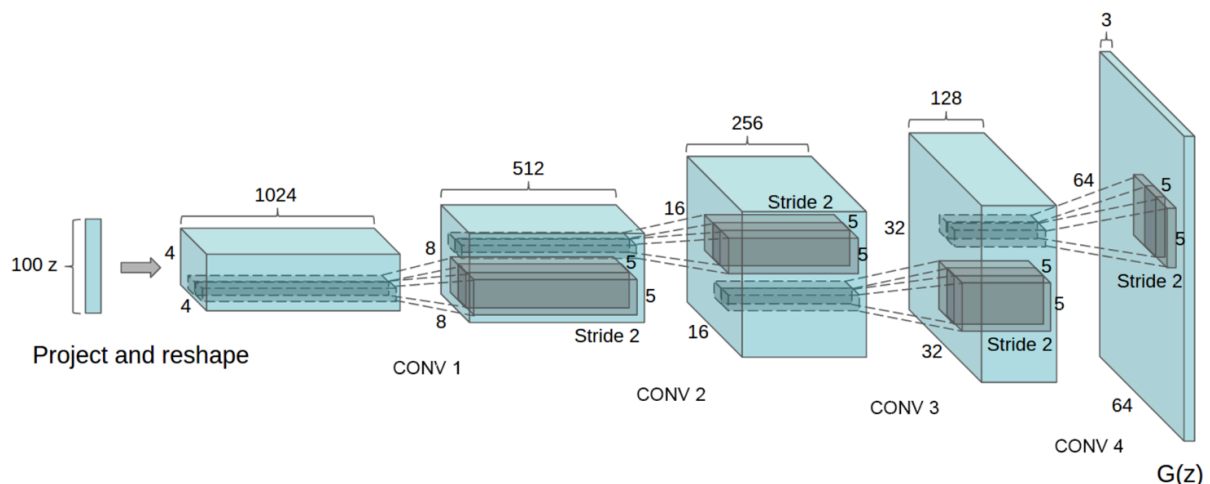


Figure 87: generated bedrooms by DCGAN

¹<https://arxiv.org/pdf/1511.06434>

6.4.2. Vector arithmetic with latent data

We can apply vector arithmetic to the z vector to fine-tune what we want out of the model:



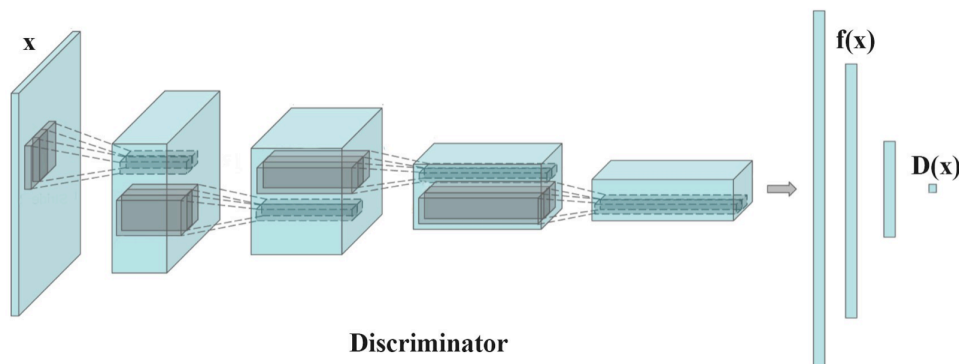
6.4.3. Cost functions

One-sided label smoothing is where we penalise the discriminator when its output is over 0.9 for real samples to avoid overconfidence.²

Feature Matching is where we minimise the difference between the features of the real and generated samples when training the discriminator, we use the cost function:

$$\| \mathbb{E}_{x \sim p_{\text{data}}(z)} f(x) - \mathbb{E}_{z \sim p_z(z)} f(G(z)) \|_2^2$$

Where $\| \cdot \|_2$ denotes the L2/Euclidean norm.



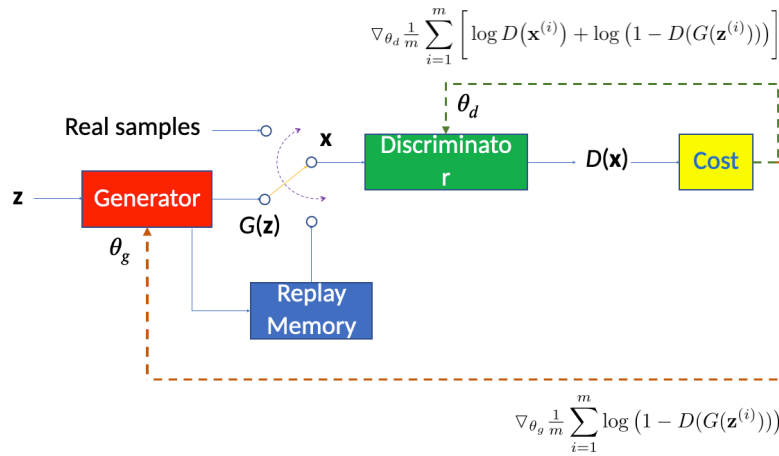
Note

Other cost functions are included in the tutorial for this section.

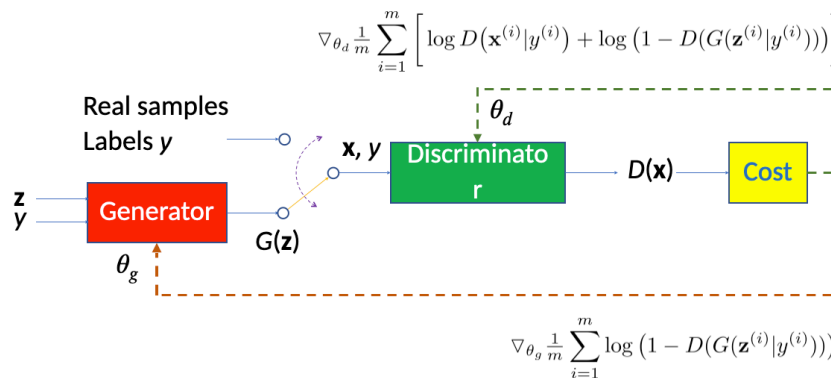
²<https://arxiv.org/pdf/1606.03498>

6.4.4. Optimisation

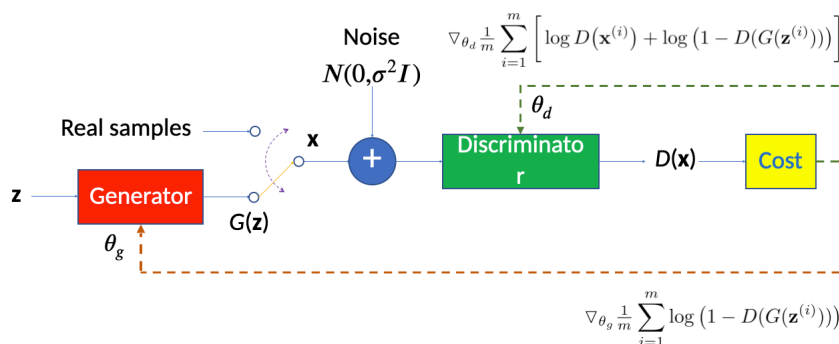
Experience replay³ is where we introduce a buffer to store previously generated samples. This technique is exclusively used in reinforcement learning (deep Q learning). The stored samples and current samples generated by the generator will be used for training.



Training with Labels (Conditional GAN – CGAN⁴⁵) is where we provide extra information such as the label y of real/generated samples to facilitate learning. When the label y is fed into the generator, the generated samples are of class y .



Adding noise (when training discriminator) allows us to make the gradient of the discriminator's cost function be proportional to the gradient of the Jensen Shannon Divergence (JSD). The JSD is not zero when the distribution of real and generated samples are very different. Noise with zero-mean normal distribution with variance σ^2 .



³<https://arxiv.org/pdf/1701.04862>

⁴<https://arxiv.org/pdf/1611.07004>

⁵<https://arxiv.org/pdf/1606.03657>

Unrolling GAN stabilises training and alleviates problem of mode collapse by allowing the generator to look a few N steps ahead to evaluate how the discriminator reacts. The generator is then updated according to the evaluated/predicted cost, i.e. cost obtained in N -th step.

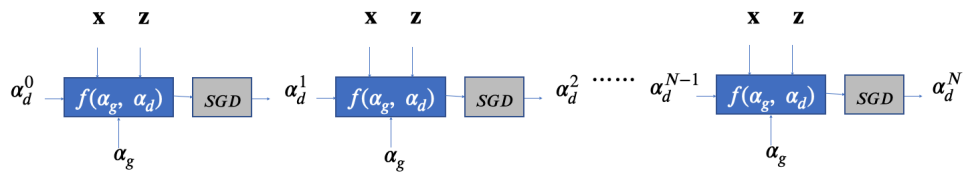
Discriminator is updated the same as original GAN.

Algorithm 4: Predict the move of the discriminator by unrolling

Consider that we have parameters θ_g and θ_d in the i -th iteration, so $\theta_g \equiv \theta_g^i$ and $\theta_d \equiv \theta_d^i$.

Assume $k = 1$

- 1 Keep parameters θ_g and θ_d
- 2 $\alpha_g = \theta_g, \alpha_d = \theta_d$ – make a copy of parameters
- 3 Unroll N times for the discriminator



remark: update in α_g does not change θ_g

- 4 $\theta_d^{i+1} \leftarrow \theta_d^i + \eta \nabla_{\theta_d} f(\theta_g, \theta_d) |_{\theta_g=\theta_g^i, \theta_d=\theta_d^i}$ – update discriminator; gradient ascending
 - 5 $\theta_g^{i+1} \leftarrow \theta_g^i - \eta \nabla_{\theta_g} f(\theta_g, \theta_d) |_{\theta_g=\theta_g^i, \theta_d=\alpha_d^N}$ – update generator; gradient descending
-

7. Feature Extraction

7.1. Feature Engineering, Selection, & Extraction

Feature Engineering: modify measurements to be appropriate for the task, e.g. more useful for classification; it is the process of choosing the most discriminative subset of data on which to perform the classification process

Feature engineering helps with:

- reducing risk of over-fitting
- reducing training time

Some measurements may not be selected because they are unobtainable or too costly to obtain. Some may just be uninformative. Some may be redundant. Sometimes it may not be obvious, because we don't know if a measurement is useful or not (esp. if we don't have the data and don't know what it means yet).

If domain knowledge cannot help us, it may be necessary to search for the best features, e.g. perform classification with different possible subsets of features to find those that produce the least error.

Example feature engineering

Translate the date into the day-of-week:

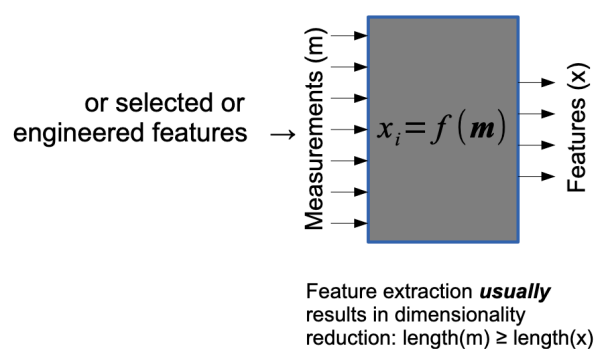
- 09/08/2019 → Friday
- useful to make predictions about future sales based on past sales
- not useful to make predictions about chances of death on date-of-birth

Translate categorical features into numbers:

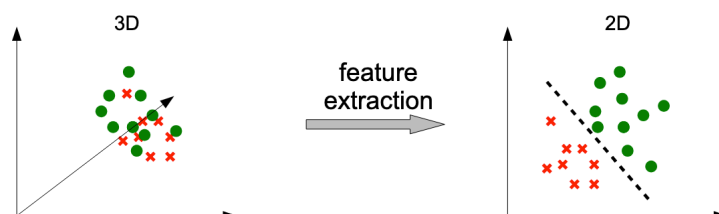
- "Monday", "Tuesday" → 1, 2
- Or one-hot encoded: "Monday" → [1,0,0,0,0,0]

For classifying fish species, useful (discriminative) features are size, colour, DNA, but not gender or eye shape.

Feature Selection: choose discriminative features as functions of the raw data



Feature Extraction: transform raw data/selected/engineered features into new feature space

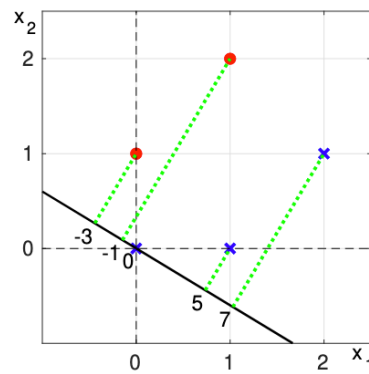


Example of feature extraction

Suppose we want to project 2D data into a 1D feature space.

Let w define the projection from the old to new feature space, e.g. $w = [5 \ -3]$.

coordinates in original feature space x^T	class	coordinates in new feature space y^T
[0,0]	1	[0]
[1,0]	1	[5]
[2,1]	1	[7]
[0,1]	0	[-3]
[1,2]	0	[-1]



$$y_1 = wx_1 = [5 \ -3] \begin{bmatrix} 0 \\ 0 \end{bmatrix} = [0]$$

$$y_2 = wx_2 = [5 \ -3] \begin{bmatrix} 1 \\ 0 \end{bmatrix} = [5]$$

$$y_3 = wx_3 = [5 \ -3] \begin{bmatrix} 2 \\ 1 \end{bmatrix} = [7]$$

$$y_4 = wx_4 = [5 \ -3] \begin{bmatrix} 0 \\ 1 \end{bmatrix} = [-3]$$

$$y_5 = wx_5 = [5 \ -3] \begin{bmatrix} 1 \\ 2 \end{bmatrix} = [-1]$$

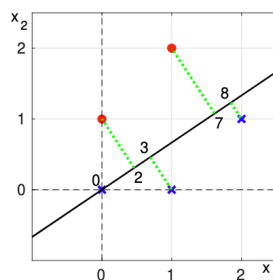
Note

NB.

- We can apply projection to the whole dataset in one go: $Y = wX$
- The scale of new coordinates depends on $\|w\|$
- Sign of new coordinates depends on sign of w
- A new axis with same slope but a different origin would produce the same new coordinate values with a fixed shift in value
- w defines the axis of the new space in the original feature space; each new coordinate is a linear combination of the original coordinates; same projection applied to all classes
- a different w will define a different axis for the new space

$$Y = wX$$

$$\begin{bmatrix} 0 & 3 & 8 & 2 & 7 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 0 & 1 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 0 & 1 & 2 & 0 & 1 \\ 0 & 0 & 1 & 1 & 2 \end{bmatrix}$$



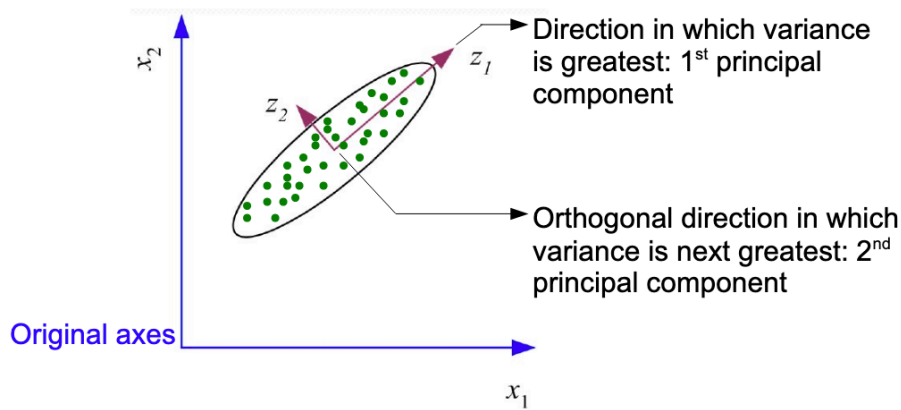
- we can project to multiple new axes simultaneously

7.1.1. Principal Component Analysis (PCA)

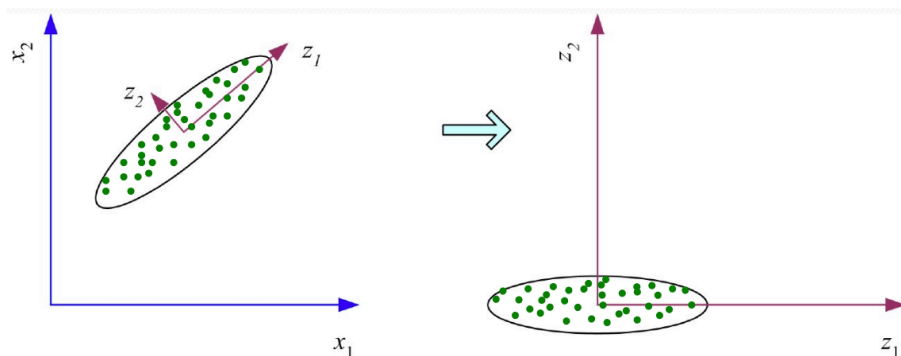
Principal Component Analysis (PCA) projects data onto orthogonal directions in which the data have the most variance

PCA Procedure

- Takes N -dimensional data and finds the M ($M \leq N$) orthogonal directions in which the data has the most variance.



- These M principal directions form a subspace. We can represent an N -dimensional datapoint by its projection onto the M principal directions.



- Typically $M < N$, so the subspace has smaller dimensionality than the original space. We choose the first M principal components, those with largest variance.

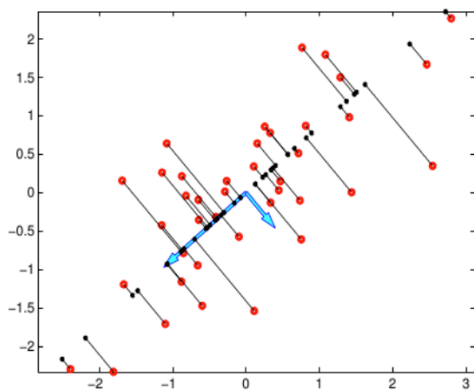


Figure 96: 2D data (red) projected on to 1st principal component (black)

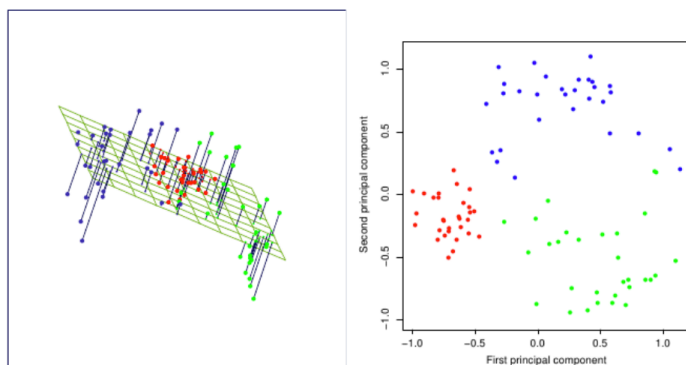


Figure 97: 3D data projected on to 1st two principal components

7.1.1.1. Karhunen-Loève Transform (KLT)

The **Karhunen-Loève Transform** is a method used for applying PCA:

- Calculate the mean of all data vectors

$$\mu = \frac{1}{N} \sum_{i=1}^N x_i$$

- Calculate covariance matrix of zero-mean data

$$C = \frac{1}{N} \sum_{i=1}^N (x_i - \mu)(x_i - \mu)^T$$

- Find eigenvalues, E , and eigenvectors, V , of C

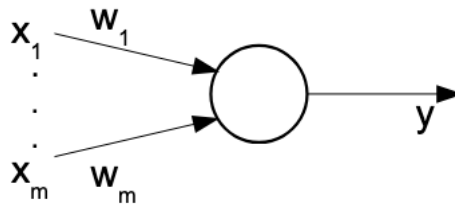
$$C = V \cdot E \cdot V^T$$

- Order eigenvalues from large to small, and discard small eigenvalues and their respective vectors
- Form matrix, $\hat{V}$, of remaining eigenvectors and apply to data (w/ mean subtracted)

$$y_i = \hat{V}^T (x_i - \mu)$$

7.1.1.2. Neural Networks for PCA

Using zero-mean data as x , a linear neuron (linear threshold without heaviside function), and output y as the projected data.



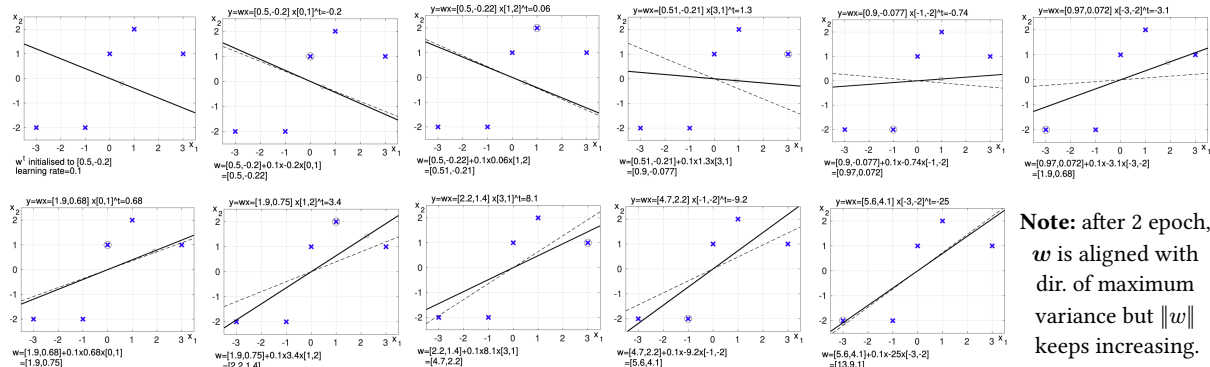
We can apply either Hebbian Learning or Oja's rule.

Using Hebbian Learning

$$\nabla w = \eta \cdot y \cdot x^t$$

For one neuron, w is aligned with 1st principal component and grows without bounds.

Suppose we have the following zero-mean 2D data, and we apply Hebbian learning to update the weights, w , of a linear unit:

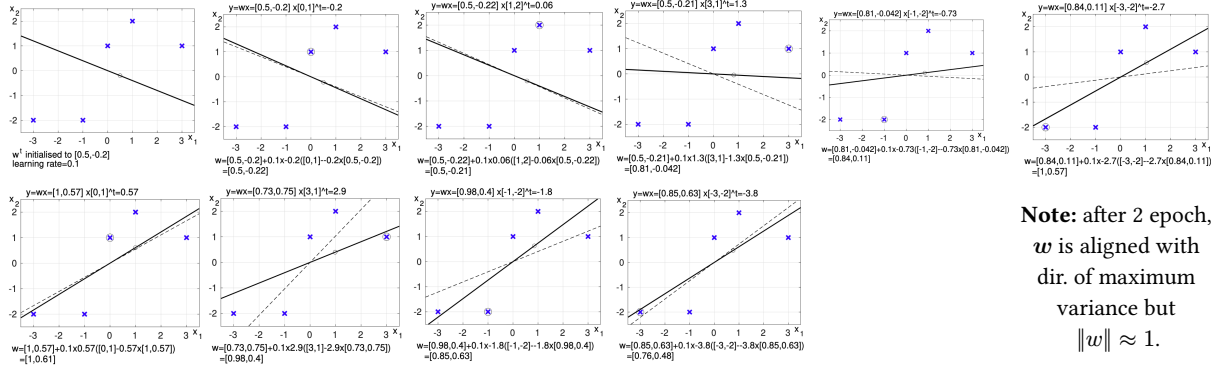


Using Oja's rule

$$\nabla w = \eta \cdot y \cdot (x^T - y \cdot w)$$

For one neuron, w is aligned with 1st principal component but weight decay term causes w to approach unit length.

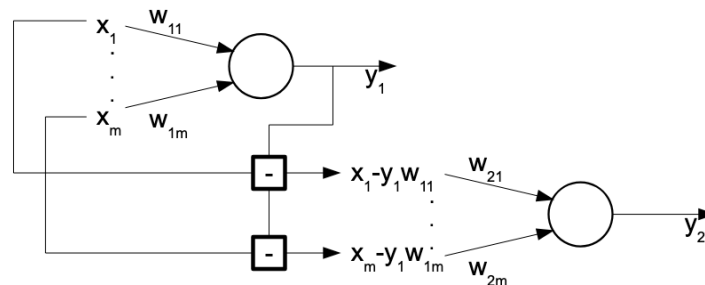
Suppose we have the following zero-mean 2D data, and we apply Oja's rule to update the weights, w , of a linear unit:



Sanger's rule

$$\nabla w_{ji} = \eta \cdot y_j \left(x_i - \sum_{k=1}^j y_k w_{ki} \right)$$

For each neuron, subtract from x the contribution of the neurons representing the $1^{st} j - 1$ principal components, then apply Oja's rule. w_j will learn the j^{th} principal component.



Oja's subspace rule

$$\nabla W = \eta \cdot y (x - W^T y)^T$$

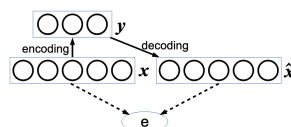
Learn the same subspace spanned by the $1^{st} n$ principal components (i.e. PCs are in random order).

NB. Oja's subspace rule is identical to rule used to update a negative feedback network.

Using Autoencoders

An autoencoder (with linear units) can also learn PCA using:

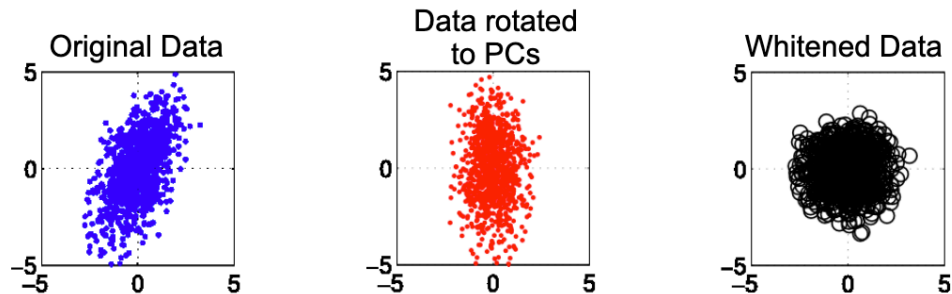
- Negative feedback learning (= Oja's subspace rule)
- Backpropagation of error



7.1.1.3. Whitening

PCA gives us a subspace where the data dimensions are uncorrelated (orthogonal) – the covariance matrix in the new space is diagonal

Whitening projects data onto orthogonal directions with equal variance; it makes the covariance matrix in the new space equal to the identity matrix



Unlike PCA, no dimensionality reduction is typically performed.

7.1.2. Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis (LDA) projects data onto directions in which the data from different classes is most distinct

Because PCA is unsupervised, it may eliminate discriminative dimensions.

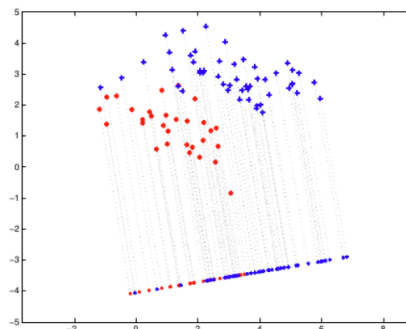


Figure 102: Overlap between elements from different categories is increased by projection to 1st principal component

With **LDA**, we seek a projection that is maximally discriminative. Requires feature extraction to be informed by class labels (i.e. supervised).

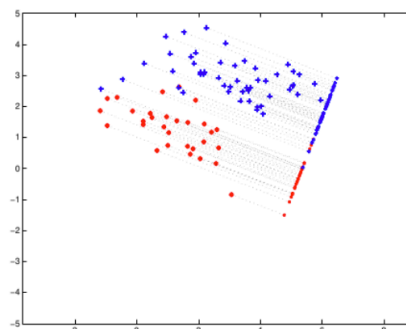
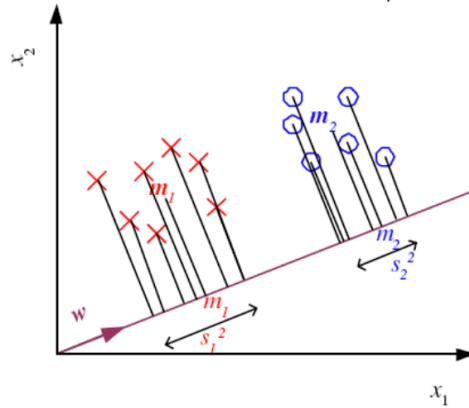


Figure 103: Overlap between elements from different categories is minimised

Between class scatter (sb) needs to be maximised but within class scatter (sw) needs to be minimised, so we can use **Fisher's method** where we find w that maximises

$$J(w) = \frac{sb}{sw}$$



$$sb = (w(m_1 - m_2))^2$$

$$\text{where } m_i = \frac{1}{n_i} \sum_{x \in \omega_i} x$$

$$sw = s_1^2 + s_2^2$$

$$\text{where } s_i^2 = \sum_{x \in \omega_i} (w(x - m_i))^2$$

7.1.3. Independent Component Analysis (ICA)

PCA finds components that are uncorrelated; $\langle y_i y_j \rangle = \langle y_i \rangle \langle y_j \rangle$

Independent Component Analysis (ICA) projects data such that new directions are independent, finding components that are statistically independent; $\langle g_1(y_i) g_2(y_j) \rangle = \langle g_1(y_i) \rangle \langle g_2(y_j) \rangle$ where g_1 and g_2 are any functions

In probability terms: $P(y_i, y_j)P(y_i)P(y_j), P(y_i|y_j) = P(y_i)$.

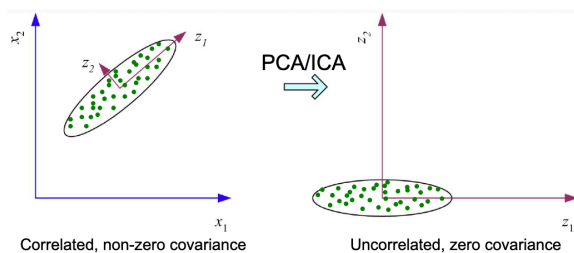


Figure 105: For Gaussian distributions: uncorrelatedness is the same as independence, so PCA produces independent components

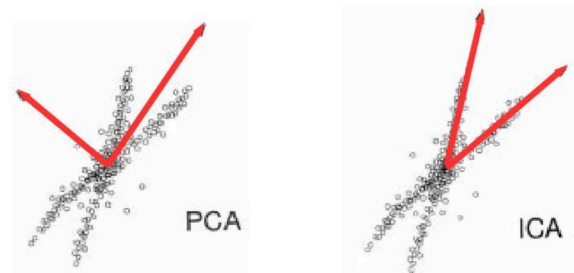


Figure 106: For non-Gaussian distributions, PCA and ICA are quite different.

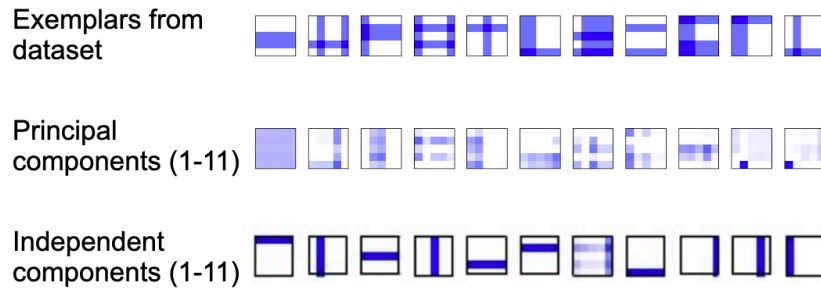


Figure 107: Further examples of differences between PCA and ICA for non-Gaussian distributions.

Neural Networks for ICA

We can also use neural networks for ICA, specifically **Oja's subspace rule**:

$$\nabla W = \eta \cdot g(y)(x - W^T g(y))^T$$

Where g is a non-linear function (e.g. sigmoid). It will learn independent components. Requires the input to be zero-mean and whitened.

There are also many other algorithms for ICA.

7.1.4. Random Projections

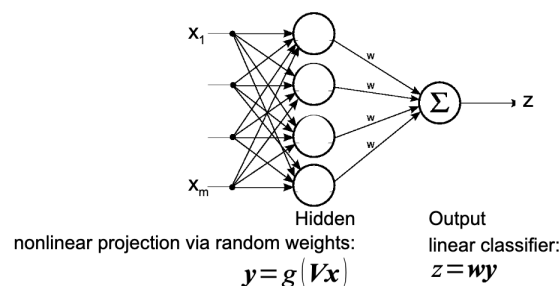
Random Projection projects data onto a high-dimensional space with random directions

These methods usually combine feature extraction using random projections with a simple, linear classifier. Both implemented using a neural network, e.g.:

- Extreme Learning Machines
- Echo State Networks
- Liquid State Machines
- Reservoir Computing

Example: Extreme Learning Machines

Identical to an RBF network, except hidden layer activation is not defined in terms of distance from a centre.



Output weights are determined so as to minimize the MSE between the hidden-layer outputs and the desired targets. If desired output is t , then we want $wy \approx t$ so for all training data, $z = wY \approx T$, weights that minimise MSE $w = TY^\dagger$. If multiple outputs (i.e. representing multiple classes), weights for each can be calculated in the same way.

Output weights calculated by $w = TY^\dagger$ where Y is hidden layer output for each training exemplar and T is required output for each training exemplar.

7.1.5. Sparse Coding

Sparse Coding projects data onto a few directions in high-dimensional space

$$\mathbf{y} = g(\mathbf{V}\mathbf{x})$$

Such that $\mathbf{y}$ contains only a few non-zero elements and $\mathbf{V}$ is a matrix of weights.

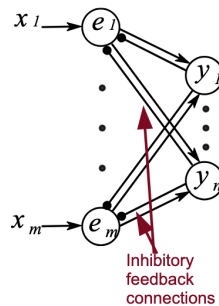
Like random projections, data is projected into high-dim space in hope that this will make the classes more separable, but unlike: only a few of the dimensions are used to represent any one exemplar (exemplars projected onto a subspace) and projection is not random.

We can define $\mathbf{V}$ by:

- learning through optimising a cost function
- using a random selection of exemplars from training set
- using all exemplars from training set

Neural Network Approach for Sparse Coding

Introduce competition so that activity of some neurons is suppressed by others. We can use a negative feedback network to achieve this. Competition selects neurons (elements of $\mathbf{y}$) that best represent the input, $\mathbf{x}$.



One measure of ability to represent input is

$$\|\mathbf{x} - \mathbf{V}^T \mathbf{y}\|_2$$

Instead of using a neural network, we can minimise an objective function such as:

$$\min_{\mathbf{y}} (\|\mathbf{x} - \mathbf{V}^T \mathbf{y}\|_2 + \lambda \|\mathbf{y}\|_0)$$

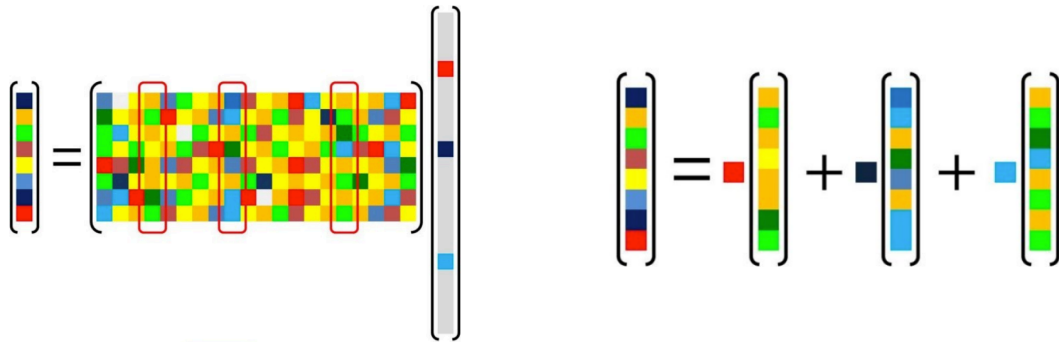
Where λ is a positive parameter defining trade-off between accuracy and sparsity.

Dictionary Approach for Sparse Coding

We can think of matrix V^T as a dictionary, where the columns are 'basic vectors'/'elements'. A vector x can be represented as a linear combination of columns sparsely selected from the dictionary.

$$x \approx V^T y$$

sample $\approx$ dictionary $\cdot$ sparse coefficients



For one sample: $x \approx V^T y$

For all samples: $X \approx V^T Y$

Maximally-sparse y has one non-zero element	Non-sparse y has any number of non-zero elements
nearest neighbour classifier if V equals training data	principal component analysis (PCA) if columns of V^T are orthogonal and rows of Y are mutually orthogonal
vector quantization (VQ) if V is learnt	independent components analysis (ICA) if rows of Y are statistically independent
	non-negative matrix factorisation (NMF) if X , V , and Y contain only elements with non-negative values

8. Support Vector Machines

Support Vector Machines (SVMs) are based on linear discriminant functions/linear machines with margins, it is a binary classifier where we compute optimal weights instead of through training. It relies on preprocessing of the data in a high dimension using Kernel functions.

Two-Class Classification Problem

Given the following information:

- Labelled training samples: $s = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$
- $\mathbf{x}_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{id} \end{bmatrix}$
- $y_i \in \{-1, 1\}, i = 1, 2, \dots, N$
- where N denotes no. of training samples

Our goal is to design a hyperplane $f(\mathbf{x}) = 0$ which can correctly classify all of the training samples.

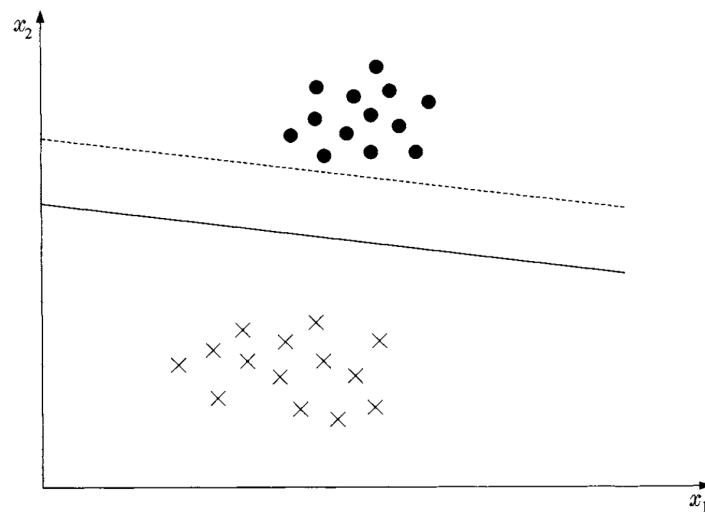


Figure 110: linearly separable two classes

To design the optimal classifier, i.e. design optimal hyperplane $f(\mathbf{x})$, we need to have:

- no errors, i.e. no mis-classification
- maximal distance/margin between nearest support vectors and separating plane

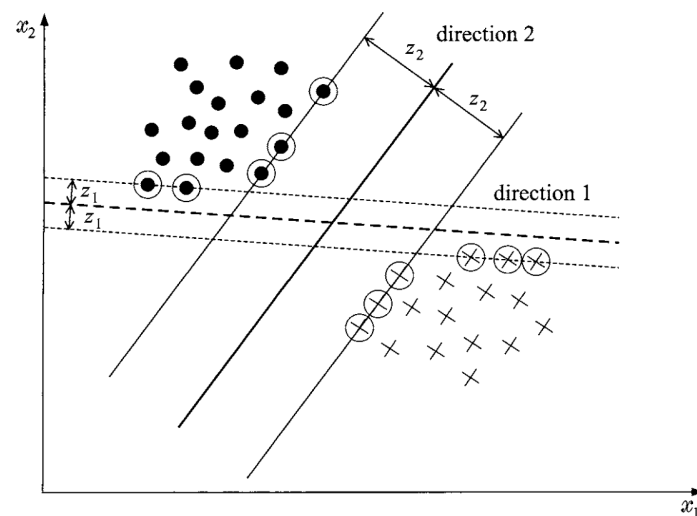


Figure 111: two linear classifiers

8.1. Linear SVMs: Linearly Seperable Case

To deal with the linearly separable case:

$$f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 = 0 \text{ where } \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}$$

Find $\mathbf{w}$ and w_0 such that the margin is optimal and:

$$\begin{cases} \mathbf{w}^T \mathbf{x} + w_0 \geq 1, & \forall \mathbf{x} \in \text{class 1 (" + 1")} \\ \mathbf{w}^T \mathbf{x} + w_0 \leq -1, & \forall \mathbf{x} \in \text{class 2 (" - 1")} \end{cases}$$

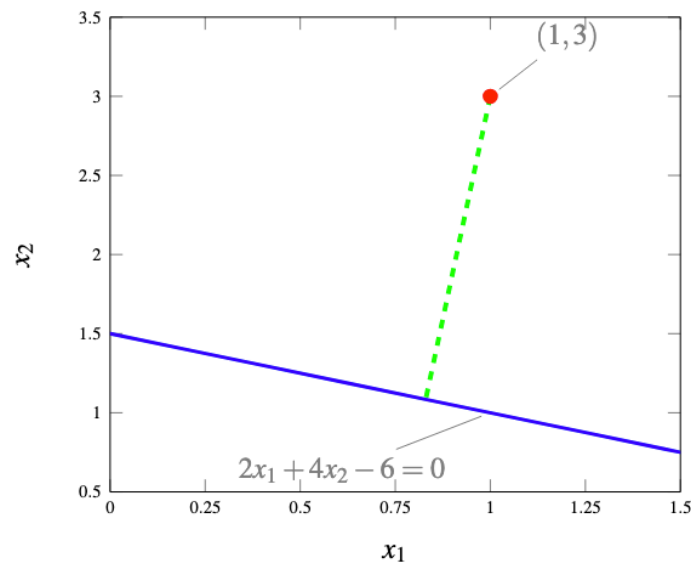
Optimal margin:

Distance of a point from a hyperplane: $\mathcal{z} = \frac{|f(\mathbf{x})|}{\|\mathbf{w}\|}$ where $\|\cdot\|$ is l^2 /Euclidean norm operator

Note

Example: given $f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + w_0 = 2x_1 + 4x_2 - 6 = 0$,

Then distance from a point $(x_1, x_2) = (1, 3)$ is $\mathcal{z} = \frac{|2 \times 1 + 4 \times 3 - 6|}{\sqrt{2^2 + 4^2}}$.



We want to achieve a maximum margin (distance): find the largest margin $\mathcal{z}$ between hyperplane and support vectors. The margin is given by:

$$\begin{aligned} \min_{\mathbf{x}_i; y_i = -1} \frac{|f(\mathbf{x}_i)|}{\|\mathbf{w}\|} + \min_{\mathbf{x}_i; y_i = +1} \frac{|f(\mathbf{x}_i)|}{\|\mathbf{w}\|} &= \frac{1}{\|\mathbf{w}\|} \left(\min_{\mathbf{x}_i; y_i = -1} |f(\mathbf{x}_i)| + \min_{\mathbf{x}_i; y_i = +1} |f(\mathbf{x}_i)| \right) \\ &= \frac{2}{\|\mathbf{w}\|} \end{aligned}$$

Constrained optimisation problem

$$\min_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{2} \|\mathbf{w}\|^2$$

$$\text{subject to } y_i(\mathbf{w}^T \mathbf{x}_i + w_0) \geq 1, \quad i = 1, 2, \dots, N$$

Remark: minimising " $\frac{1}{2} \|\mathbf{w}\|^2$ " is equivalent to maximising the margin. The constraint " $y_i(\mathbf{w}^T \mathbf{x}_i + w_0) \geq 1$ " is to make sure all samples $\mathbf{x}_i$ are correctly classified.

Method of Lagrange multipliers

Primal problem/Lagrangian: $\mathcal{L}(\mathbf{w}, w_0, \lambda) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N \lambda_i (y_i (\mathbf{w}^T \mathbf{x}_i + w_0) - 1)$ (1)

where $\lambda = [\lambda_1 \ \lambda_2 \ \dots \ \lambda_N]$, $\lambda_i \geq 0$ for all $i = 1, 2, \dots, N$

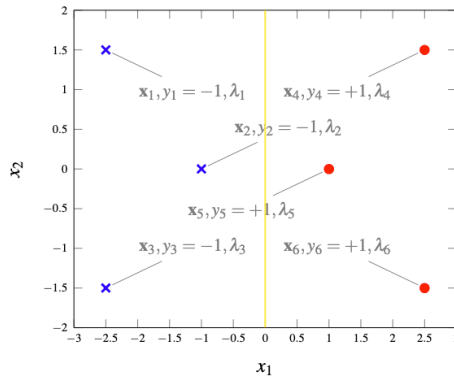
This primal problem can be transformed to the following dual problem:

$$\max_{\lambda \geq 0} \min_{\mathbf{w}, w_0} (\mathcal{L}(\mathbf{w}, w_0, \lambda)) \equiv \max_{\lambda \geq 0} \min_{\mathbf{w}, w_0} \left(\frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N \lambda_i (y_i (\mathbf{w}^T \mathbf{x}_i + w_0) - 1) \right)$$

Remark: Constraint $\lambda \geq 0$ is included in subscript of max and $\min_{\mathbf{w}, w_0}$ can be written as $\min_{\mathbf{w}}$.

$$\frac{\partial \mathcal{L}(\mathbf{w}, w_0, \lambda)}{\partial \mathbf{w}} = \mathbf{0} \Rightarrow \mathbf{w} = \sum_{i=1}^n \lambda_i y_i \mathbf{x}_i \quad (2)$$

$$\frac{\partial \mathcal{L}(\mathbf{w}, w_0, \lambda)}{\partial w_0} = 0 \Rightarrow \sum_{i=1}^N \lambda_i y_i = 0 \quad (3)$$

Example

From Equation 2, $\mathbf{w} = \lambda_1 y_1 \mathbf{x}_1 + \lambda_2 y_2 \mathbf{x}_2 + \lambda_3 y_3 \mathbf{x}_3 + \lambda_4 y_4 \mathbf{x}_4 + \lambda_5 y_5 \mathbf{x}_5 + \lambda_6 y_6 \mathbf{x}_6$

From Equation 3, $\lambda_1 y_1 + \lambda_2 y_2 + \lambda_3 y_3 + \lambda_4 y_4 + \lambda_5 y_5 + \lambda_6 y_6 = 0$

So, putting Equation 2 and Equation 3 into $\mathcal{L}(\mathbf{w}, w_0, \lambda)$ in Equation 1, we now have:

$$\begin{aligned} \mathcal{L}(\lambda) &= \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N \lambda_i (y_i (\mathbf{w}^T \mathbf{x}_i + w_0) - 1) \\ &= \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j - \sum_{i=1}^N \lambda_i y_i \mathbf{w}^T \mathbf{x}_i - \sum_{i=1}^N \lambda_i y_i w_0 + \sum_{i=1}^N \lambda_i \\ &= \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j - \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \mathbf{w}^T \mathbf{x}_j - \sum_{i=1}^N \lambda_i y_i w_0 + \sum_{i=1}^N \lambda_i \\ &= \sum_{i=1}^N \lambda_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \end{aligned}$$

The dual problem is reduced to:

$$\begin{aligned} \max_{\lambda \geq 0} & \left(\sum_{i=1}^N \lambda_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \right) \\ \text{subject to} & \sum_{i=1}^N \lambda_i y_i = 0 \\ & \lambda_i \geq 0, i = 1, 2, \dots, N \end{aligned} \quad (4)$$

- The solution λ_i to Equation 4 can be found using a quadratic programming solver.
- It is a scalar function which does not depend explicitly on the dimension of the input space.
- The solution of the Lagrange multipliers λ_i may not be unique but the hyperplane characterised by $\mathbf{w}$ and w_0 is unique.
- For those $\mathbf{x}_i$ with $\lambda_i \neq 0$, they are known as support vectors. As a result, $\mathbf{w} = \sum_{i=1}^{N_s} \lambda_i y_i \mathbf{x}_i$, $N_s \leq N$ denotes the number of support vectors. *Note: $\mathbf{x}_i$ here refers to a support vector and not sample.*
- The support vectors lie on two hyperplanes satisfying $\mathbf{w}^T \mathbf{x} + w_0 = \pm 1$ where $\mathbf{x} \in \text{support vecs.}$

INCOMPLETE

what is the point of slide 18?

Summary

The linear SVM classifier for the linearly separable case can be found by solving the solution ($\mathbf{w}$, w_0 , and λ_i) to the following conditions:

$$\begin{aligned} \frac{\partial \mathcal{L}(\mathbf{w}, w_0, \lambda)}{\partial \mathbf{w}} = \mathbf{0} & \Rightarrow \mathbf{w} = \sum_{i=1}^n \lambda_i y_i \mathbf{x}_i \\ \frac{\partial \mathcal{L}(\mathbf{w}, w_0, \lambda)}{\partial w} = 0 & \Rightarrow \sum_{i=1}^N \lambda_i y_i = 0 \\ & \lambda_i \geq 0, i = 1, 2, \dots, N \\ & \lambda_i (y_i (\mathbf{w}^T \mathbf{x}_i + w_0) - 1) = 0, i = 1, 2, \dots, N \end{aligned}$$

Hence, the **hard classifier** is given by:

$$\begin{aligned} f(\mathbf{x}) &= \text{sgn}(\mathbf{w}^T \mathbf{x} + w_0) \\ \text{where } \text{sgn}(z) &= \begin{cases} -1 & \text{if } z < 0 \\ +1 & \text{if } z \geq 0 \end{cases} \end{aligned}$$

Hence, the **soft classifier** is given by:

$$\begin{aligned} f(\mathbf{x}) &= h(\mathbf{w}^T \mathbf{x} + w_0) \\ \text{where } h(z) &= \begin{cases} -1 & \text{if } z < -1 \\ z & \text{if } -1 \leq z \leq 1 \\ +1 & \text{if } z > 1 \end{cases} \end{aligned}$$

Note

You do not need to know derivations, but you **need to know the results given above.**

8.2. Linear SVMs: Non-separable Case

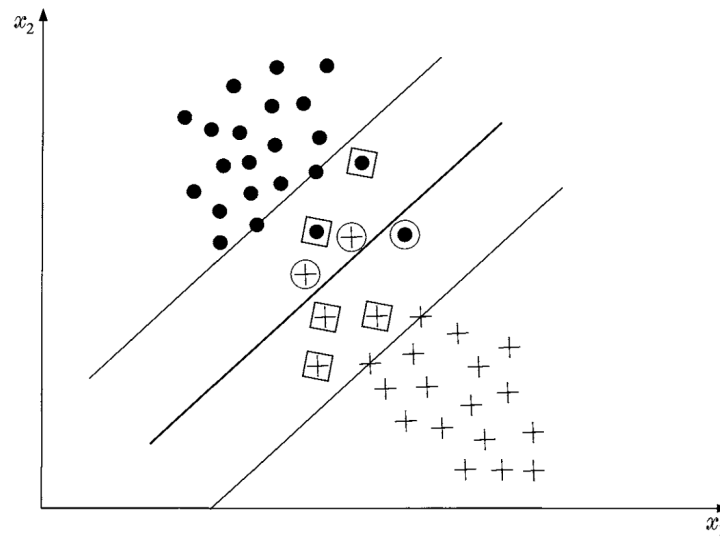


Figure 114: Two non-separable classes

There are three categories of input samples:

- samples fall outside the band and are correctly classified:

$$y_i(\mathbf{w}^T \mathbf{x}_i + w_0) \geq 1$$

- samples fall inside the band and are correctly classified:

$$0 \leq y_i(\mathbf{w}^T \mathbf{x}_i + w_0) < 1$$

- samples are misclassified:

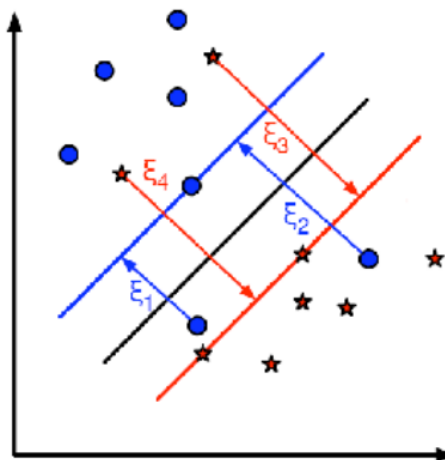
$$y_i(\mathbf{w}^T \mathbf{x}_i + w_0) < 0$$

All 3 categories can be described as:

$$y_i(\mathbf{w}^T \mathbf{x}_i + w_0) \geq 1 - \xi_i$$

Where ξ_i is known as the *slack variable*, e.g. $\xi = 0$ for first category, $0 < \xi_i \leq 1$ 2nd, third: $\xi_i > 1$.

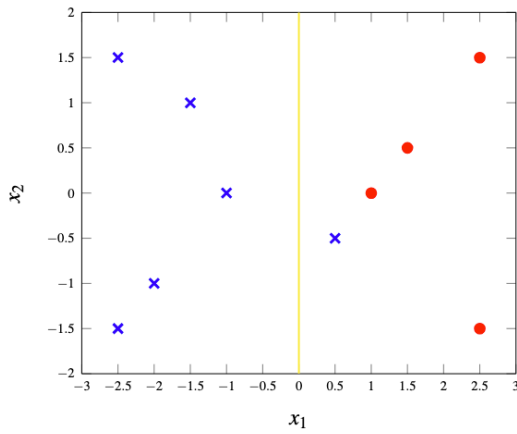
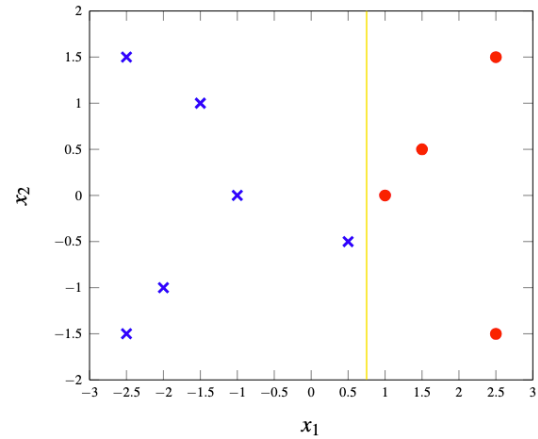
Remark: ξ_i for sample $\mathbf{x}_i$ in the second/third category is obtained as $\xi_i = 1 - y_i(\mathbf{w}^T \mathbf{x}_i + w_0)$.



We want to maximise the margin and minimise the number of misclassified points (minimise margin violations). We formulate the constrained optimisation problem as:

$$\begin{aligned} \min_{\mathbf{w}, w_0, \xi} J(\mathbf{w}, \xi) &= \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^N \xi_i \\ \text{subject to } y_i(\mathbf{w}^T \mathbf{x}_i + w_0) &\geq 1 - \xi_i, i = 1, 2, \dots, N \\ \xi_i &\geq 0, i = 1, 2, \dots, N \end{aligned}$$

Where $\xi = [\xi_1 \ \xi_2 \ \dots \ \xi_N]$ and $0 \leq C \leq +\infty$ is a pre-set constant scalar, which controls the influence of the two competing terms. This is known as the **soft-margin method**, the classifier is known as a *soft-margin classifier*. (not to be confused with soft/hard classifiers)

Figure 116: Small value of C Figure 117: Large value of C

The constrained optimisation problem is formulated as the following primal problem using method of Lagrange multipliers:

Primal problem/Lagrangian:

$$\mathcal{L}(\mathbf{w}, w_0, \xi, \lambda, \mu) = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^N \xi_i - \sum_{i=1}^N \mu_i \xi_i - \sum_{i=1}^N \lambda_i (y_i(\mathbf{w}^T \mathbf{x}_i + w_0) - 1 + \xi_i)$$

$$\text{where } \mu = [\mu_1 \ \mu_2 \ \dots \ \mu_N],$$

$$\lambda = [\lambda_1 \ \lambda_2 \ \dots \ \lambda_N]$$

are Lagrange multipliers $\mu_i \geq 0, \lambda_i \geq 0$ for all $i = 1, 2, \dots, N$

The above primal problem can be transformed to the following dual problem:

$$\min_{\mathbf{w}, w_0, \xi} \max_{\lambda \geq 0, \mu \geq 0} \mathcal{L}(\mathbf{w}, w_0, \xi, \lambda, \mu)$$

$$\frac{\partial \mathcal{L}(\mathbf{w}, w_0, \xi, \lambda, \mu)}{\partial \mathbf{w}} = \mathbf{0} \Rightarrow \mathbf{w} = \sum_{i=1}^N \lambda_i y_i \mathbf{x}_i \quad (1)$$

$$\frac{\partial \mathcal{L}(\mathbf{w}, w_0, \xi, \lambda, \mu)}{\partial w_0} = 0 \Rightarrow \sum_{i=1}^N \lambda_i y_i = 0 \quad (2)$$

$$\frac{\partial \mathcal{L}(\mathbf{w}, w_0, \xi, \lambda, \mu)}{\partial \xi_i} = 0 \Rightarrow C - \mu_i - \lambda_i = 0 \quad (3)$$

Putting Equation 1, Equation 2, and Equation 3 into $\mathcal{L}(\mathbf{w}, w_0, \xi, \lambda, \mu)$, we find:

$$\begin{aligned}
 \mathcal{L}(\lambda, \xi) &= \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^N \xi_i - \sum_{i=1}^N \lambda_i (y_i (\mathbf{w}^T \mathbf{x}_i + w_0) - 1 + \xi_i) \\
 &= \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j + C \sum_{i=1}^N \xi_i - \sum_{i=1}^N \mu_i \xi_i - \sum_{i=1}^N \lambda_i y_i \mathbf{w}^T \mathbf{x}_i - \sum_{i=1}^N \lambda_i y_i w_0 + \sum_{i=1}^N \lambda_i - \sum_{i=1}^N \lambda_i \xi_i \\
 &= \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j + C \sum_{i=1}^N \xi_i - \sum_{i=1}^N \mu_i \xi_i - \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \mathbf{w}^T \mathbf{x}_j - \sum_{i=1}^N \lambda_i y_i w_0 + \sum_{i=1}^N \lambda_i - \sum_{i=1}^N (C - \mu_i) \xi_i \\
 &= \sum_{i=1}^N \lambda_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j
 \end{aligned}$$

The same remarks from linearly separable case apply. This dual problem is the same as that of linearly separable case except a bound is given to λ_i .

The dual problem is reduced to:

$$\begin{aligned}
 &\max_{\lambda \geq 0} \left(\sum_{i=1}^N \lambda_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \right) \\
 &\text{subject to } \sum_{i=1}^N \lambda_i y_i = 0 \\
 &\quad 0 \leq \lambda_i y_i \leq C, i = 1, 2, \dots, N
 \end{aligned}$$

Summary

The linear SVM classifier for the non-separable case can be found by solving the solution $(\mathbf{w}, w_0, \xi_i, \lambda_i, \mu_i)$ to the following conditions:

$$\begin{aligned}
 \frac{\partial \mathcal{L}(\mathbf{w}, w_0, \xi, \lambda, \mu)}{\partial \mathbf{w}} = 0 &\Rightarrow \mathbf{w} = \sum_{i=1}^N \lambda_i y_i \mathbf{x}_i \\
 \frac{\partial \mathcal{L}(\mathbf{w}, w_0, \xi, \lambda, \mu)}{\partial w_0} = 0 &\Rightarrow \sum_{i=1}^N \lambda_i y_i = 0 \\
 \frac{\partial \mathcal{L}(\mathbf{w}, w_0, \xi, \lambda, \mu)}{\partial \xi_i} = 0 &\Rightarrow C - \mu_i - \lambda_i = 0 \\
 \text{s.t. } \mu_i \geq 0, \lambda_i \geq 0 &\quad i = 1, 2, \dots, N \\
 \mu_i \xi_i = 0 &\quad i = 1, 2, \dots, N \\
 \lambda_i (y_i (\mathbf{w}^T \mathbf{x}_i + w_0) - 1 + \xi_i) = 0 &\quad i = 1, 2, \dots, N
 \end{aligned}$$

As before, hard classifier is $f(\mathbf{x}) = \text{sgn}(\mathbf{w}^T \mathbf{x} + w_0)$, soft classifier is $f(\mathbf{x}) = h(\mathbf{w}^T \mathbf{x} + w_0)$.

Note

You do not need to know derivations, but you **need to know the results given above**.

8.3. Nonlinear SVMs

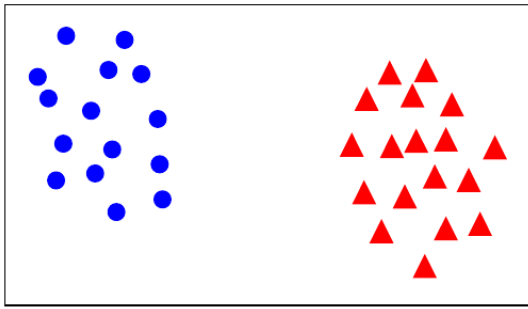


Figure 118: Linearly separable case

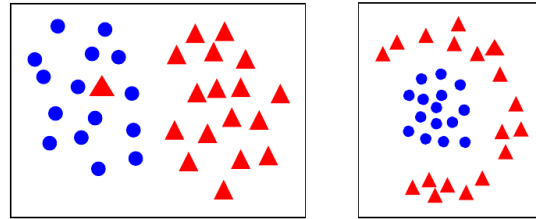
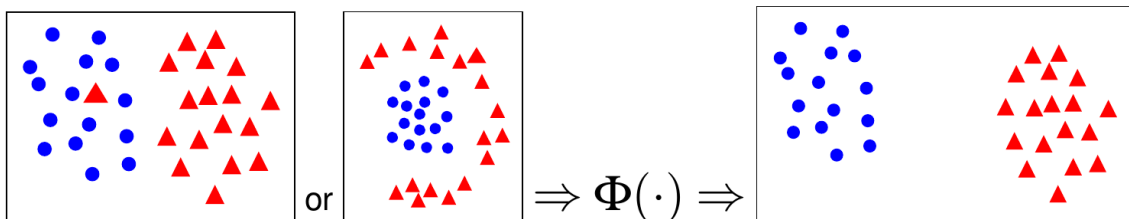


Figure 119: Nonlinearly separable case

Does there exist a mapping $\Phi(\cdot)$ such that:



Non-linear SVMs

We use a feature mapping:

$$z_i = \Phi(x_i), i = 1, 2, \dots, N$$

We use the same analysis results (formulas):

Linear SVM hyperplane:

$$w = \sum_{i \in SV_s} \lambda_i y_i x_i \Rightarrow w^T x + w_0 = 0$$

Nonlinear SVM hyperplane in z space:

$$w = \sum_{i \in SV_s} \lambda_i y_i z_i \Rightarrow w^T z + w_0 = 0$$

hyperplane in x space:

$$\underbrace{\sum_{i \in SV_s} \lambda_i y_i \Phi(x_i)^T}_{w^T} \underbrace{\Phi(x)}_z + w_0 = 0$$

The hard classifier is given by:

$$f(x) = \text{sgn} \left(\sum_{i \in SV_s} \lambda_i y_i \Phi(x_i)^T \Phi(x) + w_0 \right)$$

The soft classifier is given by:

$$f(x) = h \left(\sum_{i \in SV_s} \lambda_i y_i \Phi(x_i)^T \Phi(x) + w_0 \right)$$

8.3.1. Kernel Mappings

Certain Kernels satisfy Mercer's Theorem which allows implicit mapping to high-dimensional feature space:

$$K(\mathbf{x}_i, \mathbf{x}) = \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x})$$

Instead of finding the mapping function, $\Phi(\cdot)$, it is easier to find the kernel function, $K(\cdot)$.

The hard classifier is given by:

$$f(\mathbf{x}) = \text{sgn} \left(\sum_{i \in SV_s} \lambda_i y_i K(\mathbf{x}_i, \mathbf{x}) + w_0 \right)$$

The soft classifier is given by:

$$f(\mathbf{x}) = h \left(\sum_{i \in SV_s} \lambda_i y_i K(\mathbf{x}_i, \mathbf{x}) + w_0 \right)$$

Commonly used kernel functions

- Linear kernel/standard inner product

$$K(\mathbf{x}_i, \mathbf{x}) = \mathbf{x}_i^T \mathbf{x} + c \text{ where } c \text{ is an optional constant}$$

- Polynomial kernel of degree

$$K(\mathbf{x}_i, \mathbf{x}) = (\alpha \mathbf{x}_i^T \mathbf{x} + c)^q, q > 0$$

- Radial basis function (RBF) kernel (exponential kernel)

$$K(\mathbf{x}_i, \mathbf{x}) = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}\|^2}{2\sigma^2}}$$

- Multi-quadratic kernel

$$K(\mathbf{x}_i, \mathbf{x}) = \sqrt{\|\mathbf{x}_i - \mathbf{x}\|^2 + c}$$

- Inverse multi-quadratic kernel

$$K(\mathbf{x}_i, \mathbf{x}) = \frac{1}{\sqrt{\|\mathbf{x}_i - \mathbf{x}\|^2 + c}}$$

- Power kernel

$$K(\mathbf{x}_i, \mathbf{x}) = -\|\mathbf{x}_i - \mathbf{x}\|^q$$

- Log kernel

$$K(\mathbf{x}_i, \mathbf{x}) = -\log(\|\mathbf{x}_i - \mathbf{x}\|^q + 1)$$

- Sigmoid Function (Hyperbolic Tangent)

$$K(\mathbf{x}_i, \mathbf{x}) = \tanh(\beta \mathbf{x}_i^T \mathbf{x} + \lambda)$$

- A kernel can be constructed from other kernels:

- Linear combination of kernels

$$\sum_k \alpha_k K_k(\cdot), \alpha_k > 0, \forall k$$

- Product of kernels

$$\prod_k \alpha_k K_k(\cdot)$$

Why use kernel functions?

Original feature space is mapped to a high-dimensional feature space through $\Phi(\cdot)$. Instead of computing $\Phi(\cdot)$ and then $\Phi(\cdot)^T \Phi(\cdot)$ (two-steps), we can compute the kernel function $K(\cdot)$ in one step saving computational demand.

Example function

Consider quadratic kernel function (polynom degree two):

$$\begin{aligned} K(z, x) &= (z^T x)^2 \\ &= z_1^2 z_2^2 + 2z_1 z_2 x_1 x_2 + z_2^2 z_2^2 \end{aligned}$$

It can be shown that the feature mapping function is:

$$\Phi(x) = \begin{bmatrix} x_1 \\ \sqrt{2}x_1x_2 \\ x_2 \end{bmatrix} \Rightarrow K(z, x) = \Phi(z)^T \Phi(x) = (z^T x)^2$$

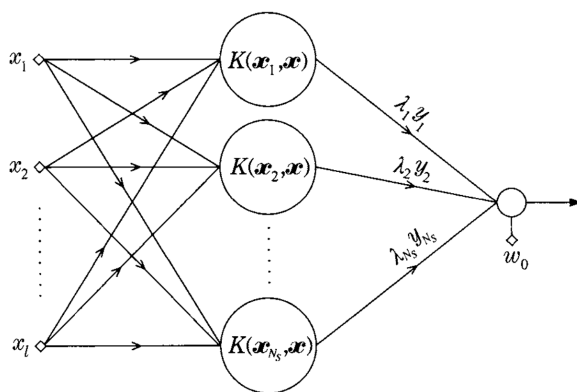


Figure 121: A diagram of SVM classifier with kernel functions $\sum_{i \in SV_s} \lambda_i y_i K(x_i, x) + w_0$

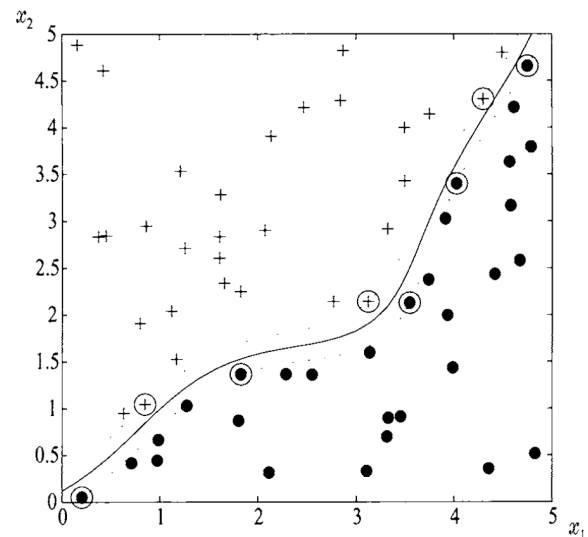


Figure 122: A nonlinearly separable classification example using nonlinear SVM classifier

Note

- When RBF kernel function is used, the SVM architecture is the same as the RBF network structure. However, the number of hidden units and the centres are determined by the optimisation procedure.
- When sigmoid kernel function is used, the SVM architecture is the same as a three-layer fully-connected feed-forward neural network structure. However, the number of hidden units is determined by the optimisation procedure.
- There is no systematic method to determine the best Kernel function and its parameters, and the parameter C (hyper-parameters), which are usually chosen by trial and error
- *Applications exist in:* Speaker verification, Face detection, Hand-writing recognition, Biomed: cancer diagnosis, epilepsy diagnosis (EEG), cardiac arrhythmia (ECG), cardiovascular disease, etc.

8.4. Multi-class SVMs

There are many approaches combining a number of two-class SVMs (linear or nonlinear) for multi-class classification. Approaches are listed below.

8.4.1. One-against-one approach

In the one-against-one approach, we need $\frac{R(R-1)}{2}$ of 2-class classifiers for an R -class problem. We train each 2-class classifier for each pair of classes. Then during decision, we choose class with majority votes.

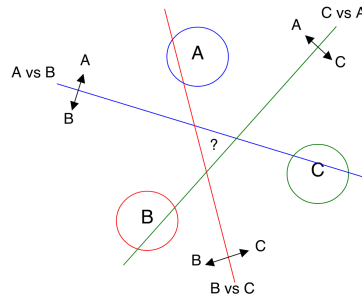


Figure 123: one-against-one approach for 3 classes

Note

- Example: 2-class hard SVM classifier $\text{sgn} \left(w_{ij}^{(ij)} x + w_0^{(ij)} \right)$ is able to classify if input x belongs to class C_i (label +1) or C_j (label -1).
- Example: when considering 3 classes, we need $3 \frac{3-1}{2} = 3$ classifiers, i.e 1 against 2, 2 against 3, 1 against 3. Majority vote wins.
- **Some regions cannot be classified.**

8.4.2. One-against-all approach

In the one-against-all approach, we need R of 2-class classifiers for an R -class problem. We train each 2-class classifier for each class against all other classes. Then, choose majority vote.

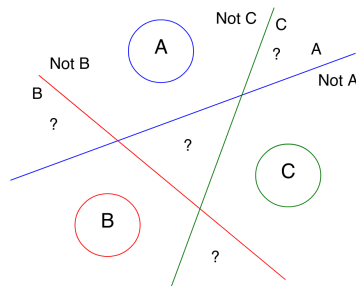


Figure 124: one-against-all approach for 3 classes using hard SVM classifiers

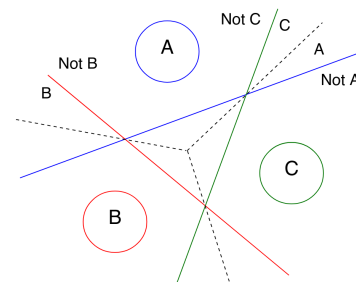


Figure 125: one-against-all approach for 3 classes using soft SVM classifiers

Note

- Example: 2-class hard SVM classifier $w_i^{(i)} x + w_0$ is able to classify if input x belongs to class C_i (positive value) or the rest (negative value).
- Example: when considering 3 classes, we need 3 classifiers, one for each.
- **Some regions cannot be classified using hard SVM classifiers.**
- **All regions can be classified using soft SVM classifiers.**
However, misclassification may occur near boundaries.

8.4.3. Binary decision tree

A **binary decision tree** requires $R - 1$ SVMs for a problem with R classes. It consults at most $\lceil \log_2 R \rceil$ SVMs to make a final decision for a sample x . The classification accuracy depends heavily on SVMs in upper layers.

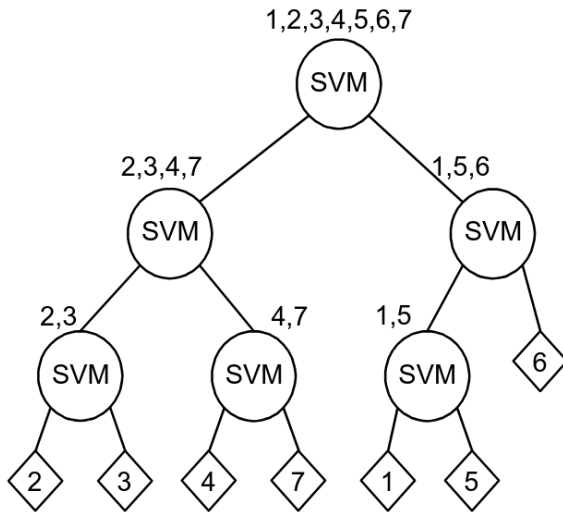


Figure 126: Binary decision tree for 7 classes

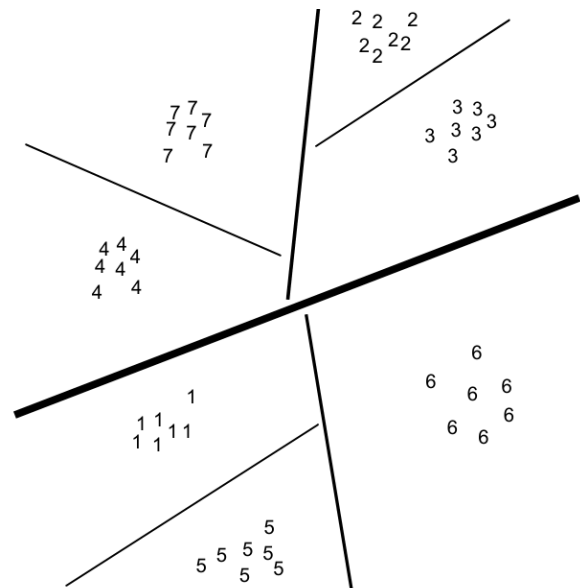


Figure 127: Binary decision tree divisions for 7 classes

8.4.4. Binary coded approach

The **binary coded approach** requires $\lceil \log_2 R \rceil$ SVMs for a problem with R classes.

Class	SVM 1	SVM 2	SVM 3
1	+1	+1	+1
2	+1	+1	-1
3	+1	-1	+1
4	+1	-1	-1
5	-1	+1	+1
6	-1	+1	-1
7	-1	-1	+1

Class assignments:

- SVM 1: $\underbrace{1234}_{+1} | \underbrace{567}_{-1}$
- SVM 2: $1256 | 347$
- SVM 3: $1357 | 246$

Figure 128: Example classification of 7 classes

$$\lceil \log_2 7 \rceil = \lceil 2.8074 \rceil = 3 \text{ SVMs}$$

9. Ensemble Methods

The idea of **ensemble methods** is to generate a collection of ‘weak’ classifiers from a given dataset, where each ‘weak’ classifier is a simple classifier (e.g. half-space classifier; can be slightly better than choosing randomly/blindly, or given by a different dataset; generated by resampling) and then combined to create a final classifier (e.g. by averaging or voting).

Advantage of ensemble methods

- Reduces variance: higher precision of the match
Results are less dependent on peculiarities of a single training set
- Reduces bias: more accurate or better quality of the match
- Improves estimate if learning algorithm is unstable
Learner is unstable if a small change to training set causes large change in output classifier
e.g. decision trees and neural networks

9.1. Bagging

Bagging (derived from ‘bootstrap aggregating’) uses multiple versions of a training set, each created by drawing $n' < n$ samples from $\mathcal{D}$ with replacement. Each bootstrap data set is used to train a different weak classifier. Final classification decision is based on vote of each weak classifier.

Algorithm 5: Bagging Procedure

- 1 Start with dataset $\mathcal{D}$
- 2 Generate M dataset $\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_M$
 - Draw $n' < n$ samples from $\mathcal{D}$ with replacement
 - Some samples can appear more than once; others may not appear at all
- 3 Learn weak classifier for each dataset, $f_i(\mathbf{x})$ for dataset $\mathcal{D}_i, i = 1, 2, \dots, M$
- 4 Combine all weak classifiers using majority voting or averaging scheme

Averaging:

$$f_{\text{final}}(\mathbf{x}) = \text{sgn} \left(\sum_{i=1}^M \frac{1}{M} f_i(\mathbf{x}) \right)$$

9.2. Boosting

Boosting is where we create a classifier with accuracy on training set greater than the average, then add new component classifiers to form an ensemble whose joint rule has arbitrarily high accuracy on the training set.

The goal is to improve the accuracy of any given learning algorithm.
Classification is boosted by combining a number of ‘weak’ classifiers.

Algorithm 6: Example using 2 dim 2-category classification task with 3 weak classifiers

- 1 Start with dataset $\mathcal{D}$
 - 2 Select $n_1 \leq n$ patterns (w/o replacement) from $\mathcal{D}$ to train a weak classifier, C_1 , min. 50% accuracy.
 - 3 Create ‘informative’ dataset D_2 ($n_2 \leq n$) of which roughly half of the patterns should be correctly classified by C_1 and the rest wrongly. Train classifier C_2 using D_2 .
 - 4 Repeat step 3, creating dataset D_3 , training classifier C_3 .
 - 5 Final decision is based on votes of weak classifiers, e.g. by first two if agree; third if first two don’t
-

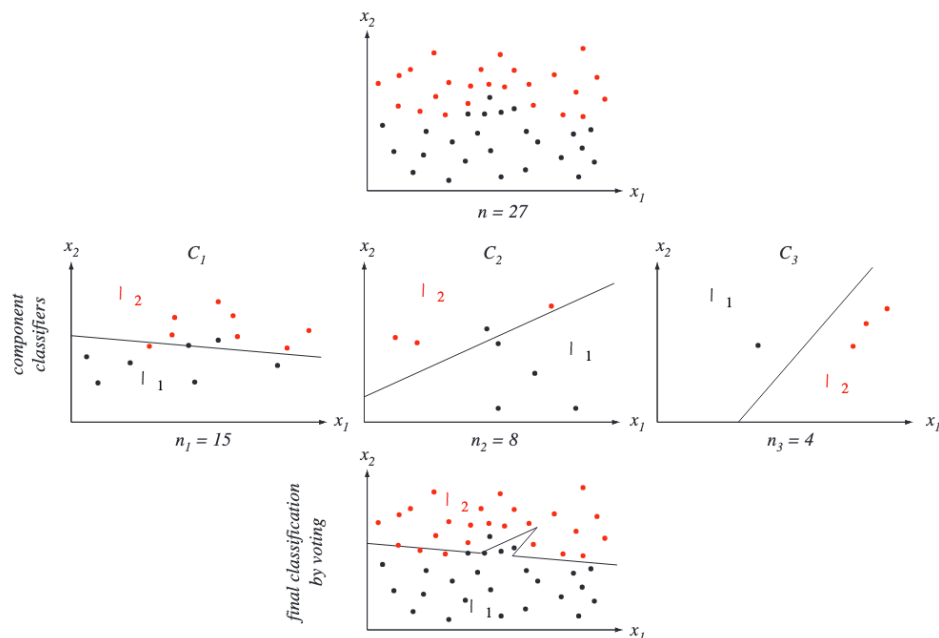


Figure 129: 3 weak classifiers in a boosting algorithm

9.2.1. AdaBoost (Adaptive Boost)

AdaBoost is the procedure of adding weak classifiers to the final classifier until a desirable classification accuracy is achieved. Each training pattern is assigned a weight (lower is assigned to patterns being identified correctly; vice-versa). AdaBoost takes more care about 'difficult' patterns.

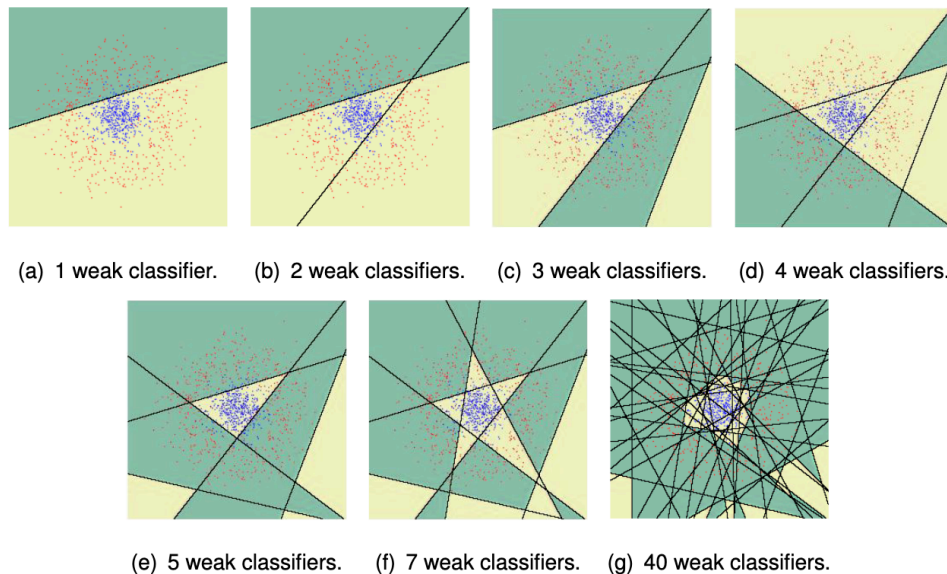


Figure 130: An example of AdaBoost classifier

Advantages	Disadvantages
Simple for implementation	Solution is suboptimal
Good classification accuracy and generalisation	Sensitive to noisy data and outliers
Can be applied to different classifiers	

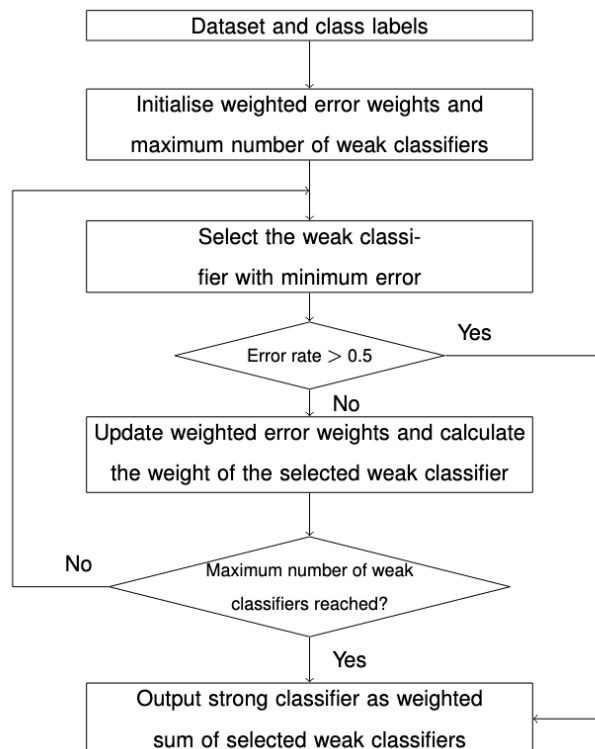


Figure 131: Flowchart of Adaboost algorithm

Algorithm: Adaboost

Given $\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$, $y_i \in \{-1, 1\}$, $i = 1, \dots, n$
begin initialise $k_{\max}$, $W_1(i) = 1/n$, $i = 1, \dots, n$
 $k \leftarrow 0$
do $k \leftarrow k + 1$
 $\hat{h}_k = \arg \min_{h_j \in \mathcal{H}} E_j$ where $E_j = \sum_{i=1, y_i \neq h_j(\mathbf{x}_i)}^n W_k(i)$ (Weighted error rate)[†]
 ε_k = overall weighted error rate of classifier $\hat{h}_k$ (i.e., the minimum E_j)
 if $\varepsilon_k > 0.5$
 $k_{\max} = k - 1$
 Stop
 end
 $\alpha_k = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_k}{\varepsilon_k} \right)$
 $W_{k+1}(i) = \frac{W_k(i) e^{-\alpha_k y_i \hat{h}_k(\mathbf{x}_i)}}{Z_k}$, $i = 1, \dots, n$ (Update $W_k(i)$)[‡]
until $k = k_{\max}$
end

[†] Find the classifier $h_k \in \mathcal{H}$ that minimises the error ε_k with respect to the distribution $W_k(i)$.

[‡] Z_k is a normalization factor chosen so that $W_{k+1}(i)$ is a normalised distribution.

Algorithm 7: AdaBoost Algorithm

Final classifier is therefore:

$$H(x) = \text{sgn} \left(\sum_{k=1}^{k_{\max}} \alpha_k \hat{h}_k(x) \right)$$

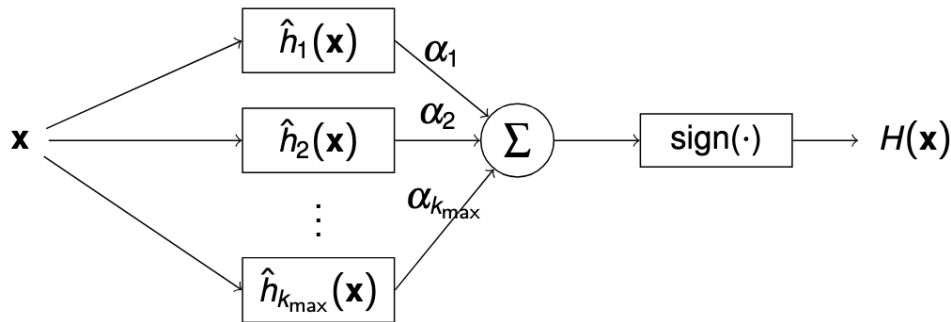


Figure 132: Structure of AdaBoost classifier

Note

Example included: 7ccsmpnn-wk9-adaboost-example.pdf

9.2.2. Stacked Generalisation

In **stacked generalisation**, the overall classification is done in two levels: The input will be processed by a group of classifiers in the first level. The output from the first layer will go through the classifier in the second level to make the final decision.

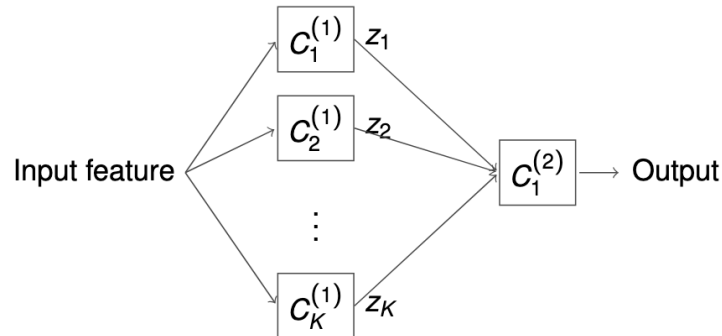


Figure 133: Stacked generalisation

Algorithm 8: Training Procedure of dataset $\mathcal{D}$ with n samples, K classifiers

- 1 Prepare K sub-training dataset: $\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_K$ of different size and samples.
 - 2 Train the k^{th} classifier using training dataset $\mathcal{D}_k$, $k = 1, 2, \dots, K$
Create a new training dataset for the second-level classifier. Outputs of first-level classifiers $(z_1, z_2, \dots, z_K)$ and the target output (y) of the classification problem (supervised) are taken to form the new dataset $\mathcal{D}_{\text{new}}$.
 - 3
 - 4 Train the second-level classifier using the dataset $\mathcal{D}_{\text{new}}$.
-

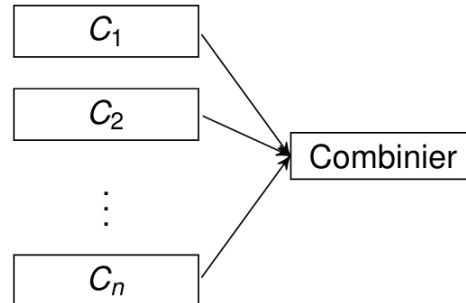
9.2.3. Structure of ensemble classifiers

Parallel Structure

All classifiers make decision independently, result is combined by a combiner.

Combination strategies incl. averaging, voting, weighted voting, adaptive, voting, etc.

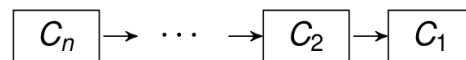
Approaches such as: bagging, boosting, AdaBoost



Serial Structure

A particular structure of hierarchical (tree) structure. Information and decision result are propagated to the later level for making decision.

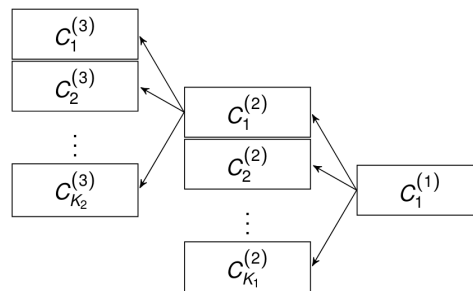
- Rough decision can be made in the upper levels for efficiency.
- Accuracy can be fine tuned in the lower levels; however error will propagate.



Hierarchical

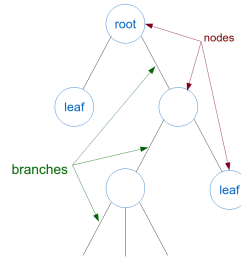
General version of parallel/serial structures.

Combination of parallel and serial structures can be used.



9.2.4. Decision Trees

A **decision tree** is a tree structure that can perform classification.



The training data consists of feature vectors and class labels, $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$.

Training Decision Trees

- Determine tree structure
- Determine decision rules.

Typically, choose feature, x_j , and threshold, t , that best splits the data into distinct classes.
- Several metrics for best split, such as
 - Gini impurity
 - Information gain
 - Variance reduction

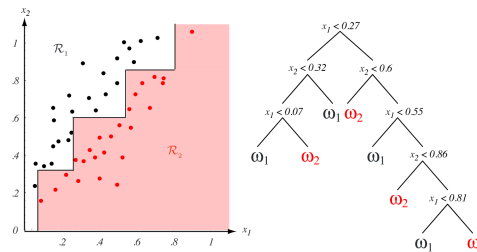


Figure 138: Example of a 2-class decision tree

Issues with Decision Trees

- Decisions at nodes could be more complex - implemented by a more complex classifier
- Prediction performance can be poor (tends to over-fit training data)
- Unstable (small changes to training set cause large changes in the classification accuracy)

Algorithm 1: Basic algorithm for constructing a binary decision tree

```

begin
  Start with the root node;
  foreach new node do
    foreach feature  $x_i$ ,  $i = 1, 2, \dots, d$ , do
      foreach threshold,  $t_i \in \{t_{1i}, t_{2i}, \dots, t_{N_i}\}$ , do
        Consider the decision rule " $x_i \leq t_i$ " for the left branch;
        Consider the decision rule " $x_i > t_i$ " for the right branch;
        Compute the impurity decrease  $\Delta i(n)$  corresponding to  $x_i$  and  $t_i$ ;
      end
    end
    Denote  $x_i$  and  $t_i$  leading to maximum impurity decrease  $\Delta i(n)$  as  $x_{best}$  and  $t_{best}$ , respectively;
    if Stop-splitting rule is met then
      Declare this node as a leaf and designate it with a class label;
    else
      Grow the tree at this node by constructing:
      the left branch using the decision rule:  $x_{best} \leq t_{best}$ ;
      the right branch using the decision rule:  $x_{best} > t_{best}$ ;
    end
  end
end

```

9.2.4.1. Bagging Decision Trees

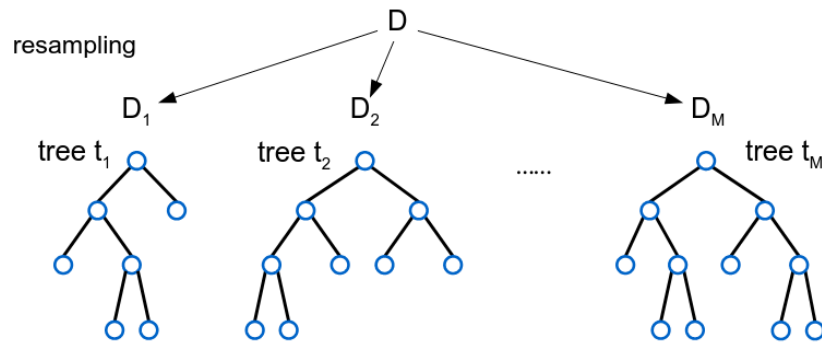


Figure 140: Training

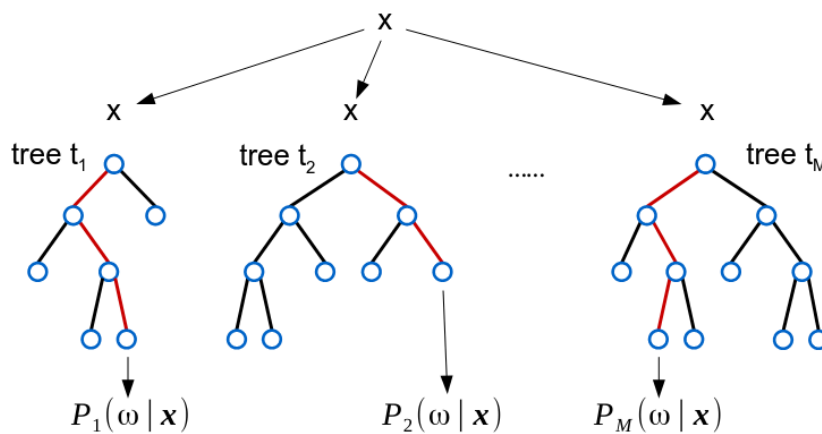


Figure 141: Classifying

Overall classification is therefore:

$$P(\omega|\mathbf{x}) = \frac{1}{M} \sum_{t=1}^M P_t(\omega|\mathbf{x})$$

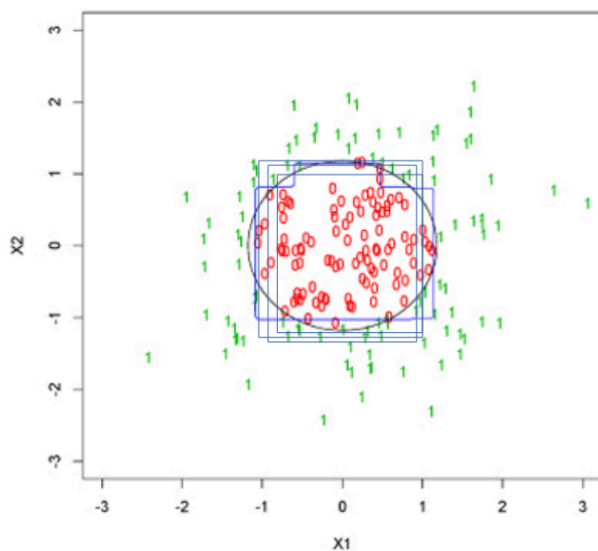


Figure 142: Each DT will define a different decision boundary

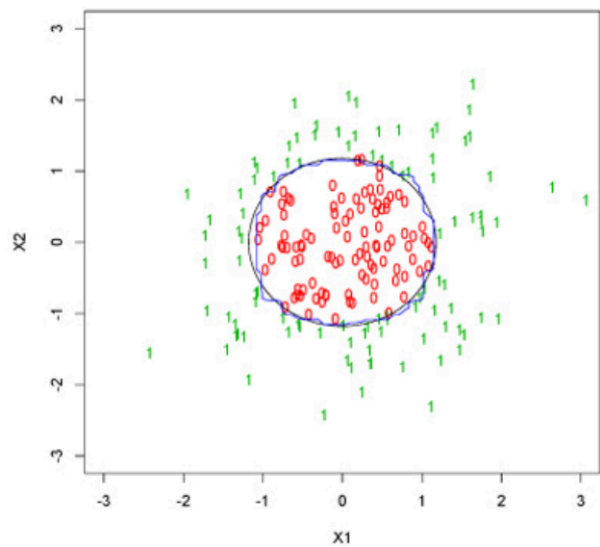


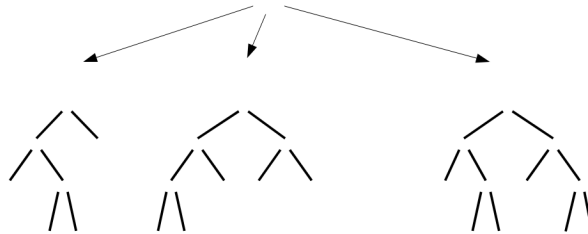
Figure 143: Bagging averages classification made by component classifiers, producing a smoother boundary overall.

9.2.4.2. Random Forests

A **random forest** is an ensemble of bagged decision trees with randomised feature selection. They are fast (scalable), accurate, simple to implement, and popular. Final decision is taken from all DTs.

- Bagging: randomly drawn subset of training data; typically each tree trained on 63.2% of data
- Randomised feature selection: randomly drawn subset of features

Typically features in subset $d' = \sqrt{d}$ or $\log_2(d)$ (round to nearest integer)



Note

If all the training samples are placed in a matrix, X (columns are samples, rows are features):

- Resampling creates a new matrix, X_k , with a subset of columns from X
- Randomised feature selection creates a new matrix, X_k , with a subset of rows from X

$$\mathbf{X} = \begin{pmatrix} 0.1 & 0.2 & 0.3 & 0.4 & 0.5 & 0.6 \\ 0 & -0.2 & -0.1 & 0.2 & 0.9 & 0.3 \\ 0.8 & 0.3 & 0.1 & 0.7 & 0.6 & 0.9 \\ 0.7 & 0.2 & 0.5 & 0.8 & 0.3 & 0.4 \\ -0.5 & 0.7 & 0.8 & 0.5 & 0.2 & 0.4 \end{pmatrix}$$
$$\mathbf{x}_k = \begin{pmatrix} 0 & -0.1 & 0.3 \\ -0.5 & 0.8 & 0.4 \end{pmatrix}$$

Remark: Resampling and randomised feature selection increase diversity among decision trees

We repeat the process specified below to grow the tree to some maximum depth:

Algorithm 2: Randomised Tree Learning procedures (to determine decision rule at each node) for the k -th decision tree using the sub-dataset $\mathbf{X}_k$

parameters: Choose r and d' ;

repeat

- select a feature, x_j , from the valid range of features, $j \in \{1, 2, \dots, d'\}$ (x_j is the j -th element of the feature vector);
- select a threshold, t , from the valid range of features, $t \in [\min(x_j), \max(x_j)]$;
- Measure how well this splits data into distinct classes;

until r times is reached;

Choose j and t that best splits data into distinct classes;

Remark: smaller r means more random tree; increasing diversity but decreasing performance of the component classifiers

10. Clustering

A **cluster** is a group of samples of similar properties.

Clustering is to put samples into one of the K clusters. The samples from the same cluster are similar to each other but dissimilar to samples in other clusters.

Note

Can be used for understanding: putting similar objects together can help us understand common characteristics. For utility: extracted info. can be used in different areas.

Clustering can be:

- **Partitional clustering**: divide samples into clusters such that each sample is in one cluster
- **Hierarchical clustering**: divide samples into nested clusters organised as a tree; each cluster is the union of its sub-clusters
 - **Agglomerative approach**: treat each sample as a single cluster then group into clusters in a bottom-up approach
 - **Divisive approach**: treats whole group of samples as a single cluster and then break them down into small clusters in a top-down approach

Important

Currently omitted a lot of content here because it's pointless if you've done modules covering basic clustering before.

10.1. k-Means Clustering

k-Means Clustering is, given c clusters/classes and n samples, assigning each sample to one of c clusters. Results are sensitive to initial location but works fine when clusters form well-separated, compact, clouds. It is fast and simple to implement.

We define a cost function as the sum of square Euclidean distances of each cluster from its mean:

$$J_e = \sum_{j=1}^c \sum_{x \in \mathcal{D}_j} \|x - m_j\|^2$$

The optimal partition is defined as one that minimizes J_e , also called **minimum variance partition**.

Algorithm: K-means clustering

begin initialise n, c ($K = c$), $\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_c$

do classify n samples according to the nearest $\mathbf{m}_j, j \in \{1, 2, \dots, c\}^\dagger$;

 recompute $\mathbf{m}_j^\ddagger$

until no change in $\mathbf{m}_j$

return $\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_c$

end

† minimise the Euclidean distance

‡ average of the samples in the j^{th} cluster

Note

Example included: 7ccsmpnn-wk10-kmeans-example.pdf

10.2. Fuzzy k-Means Clustering

Fuzzy k-Means is the same as normal k-Means, except:

- Sample can have graded/fuzzy membership of a cluster, defined by u_{ij} for a sample x_j in cluster i

$$u_{ij} \in [0, 1], \sum_{i=1}^c u_{ij} = 1$$

- Goal is to find the cluster centres, $\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_c$, given by:

$$\mathbf{m}_i = \frac{\sum_{j=1}^n \mu_{ij}^b \mathbf{x}_j}{\sum_{j=1}^n \mu_{ij}^b} \text{ where } b > 1$$

- An iterative algorithm (“fuzzifier”) adjusts the cluster centres, and membership values, to minimise the weighted sum of square Euclidean distances between centres and samples.

$$J_e = \sum_{i=1}^c \sum_{j=1}^n \mu_{ij}^b \|x_j - m_i\| \text{ where } b \geq 1$$

- Small b causes samples to be members of one/few clusters.
- Large b causes samples to be members of many clusters.

Algorithm: Fuzzy K-means clustering

begin initialise n, c ($K = c$), μ_{ij} ($i = 1, 2, \dots, c; j = 1, 2, \dots, n$), $\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_c$

normalize cluster memberships: $\mu_{ij} = \frac{\mu_{ij}}{\sum_{r=1}^c \mu_{rj}}$

do

update cluster centres: $\mathbf{m}_i = \frac{\sum_{j=1}^n \mu_{ij}^b \mathbf{x}_j}{\sum_{j=1}^n \mu_{ij}^b}$

update memberships: $\mu_{ij} = \frac{(1/\|\mathbf{x}_j - \mathbf{m}_i\|)^{\frac{2}{b-1}}}{\sum_{r=1}^c (1/\|\mathbf{x}_j - \mathbf{m}_r\|)^{\frac{2}{b-1}}}$

until no change in $\mathbf{m}_i$
return $\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_c$

end

$$S = \{ \underbrace{(-1, 3)}_{\mathbf{x}_1}, \underbrace{(1, 2)}_{\mathbf{x}_2}, \underbrace{(0, 1)}_{\mathbf{x}_3}, \underbrace{(4, 0)}_{\mathbf{x}_4}, \underbrace{(5, 4)}_{\mathbf{x}_5}, \underbrace{(3, 2)}_{\mathbf{x}_6} \}$$

$$\underbrace{(\mu_{11}, \mu_{21}) = (0.1, 0.9)}_{\text{for } \mathbf{x}_1}, \underbrace{(\mu_{12}, \mu_{22}) = (0.2, 0.8)}_{\text{for } \mathbf{x}_2}, \underbrace{(\mu_{13}, \mu_{23}) = (0.3, 0.7)}_{\text{for } \mathbf{x}_3},$$

$$\underbrace{(\mu_{14}, \mu_{24}) = (0.95, 0.05)}_{\text{for } \mathbf{x}_4}, \underbrace{(\mu_{15}, \mu_{25}) = (0.85, 0.15)}_{\text{for } \mathbf{x}_5}, \underbrace{(\mu_{16}, \mu_{26}) = (0.75, 0.25)}_{\text{for } \mathbf{x}_6}$$

Cluster centres:

$$\mathbf{m}_1 = \frac{\sum_{j=1}^n \mu_{1j}^b \mathbf{x}_j}{\sum_{j=1}^n \mu_{1j}^b} = \frac{\mu_{11}^b \mathbf{x}_1 + \mu_{12}^b \mathbf{x}_2 + \mu_{13}^b \mathbf{x}_3 + \mu_{14}^b \mathbf{x}_4 + \mu_{15}^b \mathbf{x}_5 + \mu_{16}^b \mathbf{x}_6}{\mu_{11}^b + \mu_{12}^b + \mu_{13}^b + \mu_{14}^b + \mu_{15}^b + \mu_{16}^b}$$

$$\mathbf{m}_2 = \frac{\sum_{j=1}^n \mu_{2j}^b \mathbf{x}_j}{\sum_{j=1}^n \mu_{2j}^b} = \frac{\mu_{21}^b \mathbf{x}_1 + \mu_{22}^b \mathbf{x}_2 + \mu_{23}^b \mathbf{x}_3 + \mu_{24}^b \mathbf{x}_4 + \mu_{25}^b \mathbf{x}_5 + \mu_{26}^b \mathbf{x}_6}{\mu_{21}^b + \mu_{22}^b + \mu_{23}^b + \mu_{24}^b + \mu_{25}^b + \mu_{26}^b}$$

Figure 147: Example assuming $K = 2$, two cluster centres denoted by m_k

10.3. Iterative Optimisation

Once a criterion function has been selected, clustering becomes a problem of *discrete optimisation*. As the sample set is finite there is a finite number of possible partitions, and the optimal one can be always found by exhaustive search.

More frequently, we use an iterative optimisation procedure to select the optimal partitions. Basic idea lies in starting from reasonable initial partition and ‘moving’ samples from one cluster to another to try to minimise the criterion function.
Guaranteed local, not global, optimisation.

Let us consider an iterative procedure to minimise the sum-of-squared-error criterion J_e ,

$$J_e = \sum_{j=1}^c J_j \text{ where } J_j = \sum_{\mathbf{x} \in \mathcal{D}_j} \|\mathbf{x} - \mathbf{m}_j\|^2 \text{ is the effective error per cluster}$$

It can be proved that if a sample $\hat{\mathbf{x}}$ currently in cluster $\mathcal{D}_i$ is tentatively moved into $\mathcal{D}_j$, then the change in the two errors is given by:

$$J_j^* = J_j + \frac{n_j}{n_j + 1} \|\hat{\mathbf{x}} - \mathbf{m}_j\|^2 \text{ and } J_i^* = J_i - \frac{n_j}{n_j - 1} \|\hat{\mathbf{x}} - \mathbf{m}_j\|^2$$

We select the right move by checking $\frac{n_j}{n_j - 1} \|\hat{\mathbf{x}} - \mathbf{m}_j\|^2 > \frac{n_j}{n_j + 1} \|\hat{\mathbf{x}} - \mathbf{m}_j\|^2$.

After moving, update $\mathbf{m}_i$ and $\mathbf{m}_j$.

Pseudo Code: Algorithm for Basic iterative minimum-squared-error clustering

```
begin initialise  $n, c, \mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_c$ 
do randomly select a sample  $\hat{\mathbf{x}}$ ;
   $i \leftarrow \arg \min_{i'} \|\hat{\mathbf{x}} - \mathbf{m}_{i'}\|$  (classify  $\hat{\mathbf{x}}$ )
  if  $n_i \neq 1$  (do not destroy a singleton cluster)
    compute  $\rho_j = \begin{cases} \frac{n_j}{n_j+1} \|\hat{\mathbf{x}} - \mathbf{m}_j\|^2 & j \neq i \\ \frac{n_j}{n_j-1} \|\hat{\mathbf{x}} - \mathbf{m}_j\|^2 & j = i \end{cases}, j = 1, 2, \dots, c$ 
     $k \leftarrow \arg \min_j \rho_j$  (find the minimum  $\rho_j$ )
    transfer  $\hat{\mathbf{x}}$  to  $\mathcal{D}_k$  from  $\mathcal{D}_i$ 
    recompute  $J_e, \mathbf{m}_i, \mathbf{m}_k$ 
  end
until no change in  $J_e$  in  $n$  attempts
return  $\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_c$ 
end
```

This is a **sequential version** of the K-means algorithm, with the difference being that K-mean waits until n samples have been reclassified before updating, whereas the latter updates each time a sample is reclassified.

10.4. Hierarchical Clustering

Agglomerative Hierarchical Clustering: given n samples and each sample represents a cluster, merge clusters one by one in each lever until c clusters are left.

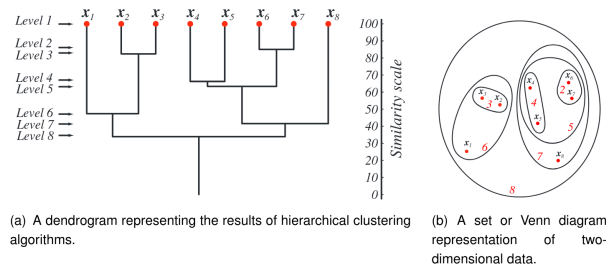


Figure 148: An example of Agglomerative Hierarchical Clustering

Pseudo Code: Algorithm for Agglomerative hierarchical clustering

```

begin initialise  $c, \hat{c} \leftarrow n, \mathcal{D}_i \leftarrow \{\mathbf{x}_i\}, i = 1, 2, \dots, n$ 
do  $\hat{c} = \hat{c} - 1$ 
    Find the nearest clusters, say,  $\mathcal{D}_i$  and  $\mathcal{D}_j$ 
    Merge  $\mathcal{D}_i$  and  $\mathcal{D}_j$ 
until  $c = \hat{c}$ 
return  $c$  clusters
end

```

c : target number of clusters; $\hat{c}$: current number of clusters.

Note

Example included: [7ccsmpnn-wk10-agglomerative-hierarchical-clustering-example.pdf](#)

Distance measures

- Single-link clustering: min. distance between samples

$$d_{\min}(\mathcal{D}_i, \mathcal{D}_j) = \min_{x \in \mathcal{D}_i, x' \in \mathcal{D}_j} \|x - x'\|$$

- Complete-link clustering: max. distance between samples

$$d_{\max}(\mathcal{D}_i, \mathcal{D}_j) = \max_{x \in \mathcal{D}_i, x' \in \mathcal{D}_j} \|x - x'\|$$

- Group-average clustering: average distance between all pairs of samples

$$d_{\text{avg}}(\mathcal{D}_i, \mathcal{D}_j) = \frac{1}{n_i n_j} \sum_{x \in \mathcal{D}_i} \sum_{x' \in \mathcal{D}_j} \|x - x'\|$$

- Centroid clustering: average of feature vectors in each cluster

$$d_{\text{mean}}(\mathcal{D}_i, \mathcal{D}_j) = \|\overline{\mathcal{D}_i} - \overline{\mathcal{D}_j}\|$$

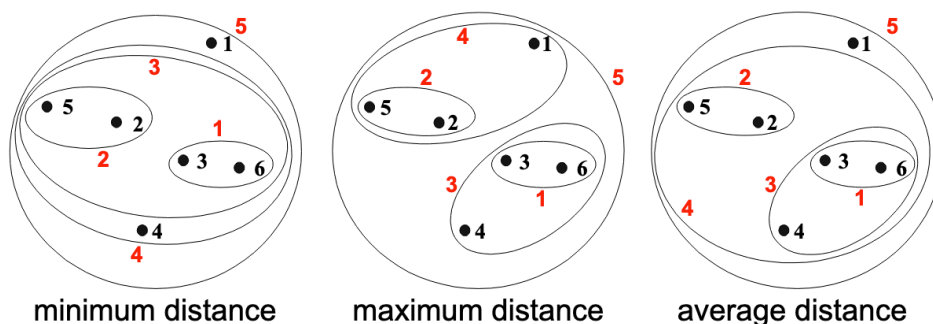


Figure 149: Results will vary depending on method used to calculate cluster similarity.

10.5. Competitive Learning

Competitive Learning is related to decision-directed versions of K-means based on neural network learning rules. Adjustment happens to the single cluster centre most similar to the pattern considered in each iteration.

Drawbacks

- Solution may not be a global optimal solution minimising a single cost or criterion function.
- Algorithm may not converge to a solution.

Competitive Learning w/o normalisation

No input augmentation needed

No input and weight normalisation are required

Similarity measures:

Euclidean distance criterion, i.e. $\|x - w_{j'}\|$

Update rule is $w_j \leftarrow w_j + \eta(x - w_{j'})$

```
Pseudo Code: Algorithm for Competitive learning without normalisation
begin initialise  $\eta (> 0)$ ,  $n$ ,  $c$ ,  $w_1, w_2, \dots, w_c$ 
do randomly select a pattern  $x$ ;
   $j \leftarrow \arg \min_j \|x - w_j\|$  (classify  $x$ )
   $w_j \leftarrow w_j + \eta(x - w_j)$  (weight update)
until no significant change in  $w_1, w_2, \dots, w_c$ 
return  $w_1, w_2, \dots, w_c$ 
end
```

Important Steps

- The initial weight vector (cluster centres) w_j are traditionally chosen from the dataset (with n samples), but not necessary.
- The weight vector is updated according to update rule given before. $\eta > 0$ is a pre-defined learning rate, x is randomly presented to the algorithm.
- The update rule will apply only to the most active cluster unit (winner-take-all), i.e., the one produces the smallest value of Euclidean distance.

Competitive Learning with normalisation

Input augmentation is required

Input and weight normalisation are required

Similarity measures:

Inner product criterion, i.e. $w_{j'}^T x$

Update rule is $w_j \leftarrow w_j + \eta x$

```
Pseudo Code: Algorithm for Competitive learning with normalisation
begin initialise  $\eta (> 0)$ ,  $n$ ,  $c$ ,  $w_1, w_2, \dots, w_c$ 
 $x_i \leftarrow [1; x_i], i = 1, \dots, n$  (augment all patterns)
 $\tilde{x}_i \leftarrow x_i / \|x_i\|, i = 1, \dots, n$  (normalise all patterns)
do randomly select a pattern  $x$ ;
   $j \leftarrow \arg \max_j w_j^T x$  (classify  $x$ )
   $w_j \leftarrow w_j + \eta x$  (weight update)
   $w_j \leftarrow w_j / \|w_j\|$  (weight normalisation)
until no significant change in  $w_1, w_2, \dots, w_c$ 
return  $w_1, w_2, \dots, w_c$ 
end
```

Important Steps

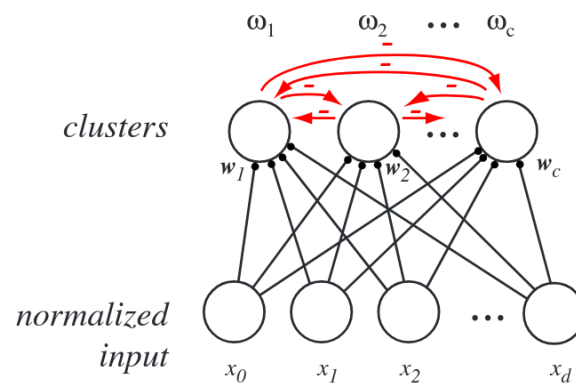
- Each input pattern must be augmented and then normalised to unit length.

Augmentation: $x' = \begin{bmatrix} 1 \\ x \end{bmatrix} = [1 \ x_1 \ x_2 \ \dots \ x_d]^T$

Normalisation: $x'' \leftarrow \frac{x'}{\|x'\|}$ such that $\|x''\| = 1$

- The initial weight vector (cluster centres) w_j are traditionally chosen from the dataset, but not necessary.
- Update rule is given as before, same parameters as version w/o normalisation.
- The update rule will apply only to the most active cluster unit, i.e., the one produces the largest value of inner product.
- Normalisation makes the classification/ clustering being done based on the position in the feature space rather than the magnitude of w ; otherwise, w could grow in magnitude and dominate the update process.

How to determine which cluster the presented $\mathbf{x}$ belongs to? The cluster with the weight producing the largest value of $net_j = \mathbf{w}_j^T \mathbf{x}$, i.e., $j \leftarrow \arg \max_{j'} \mathbf{w}_{j'}^T \mathbf{x}$.



10.6. Clustering for Unknown No. of Clusters

Typically, the number of clusters is assumed known, when it is not, there are a few ways to proceed:

1. Select number of clusters by minimising a criterion function for diff. number of clusters; common approach is to repeat clustering with $c = 1, 2, 3, \dots$

When J_e is the sum-of-squared-error, increasing c , always decreases J_e . However, typically when increasing c , J_e decreases faster up to a point then decreases more slowly.

2. Create a new cluster based on a threshold. For example, new cluster is created for coming sample if its 'distance' between existing clusters is larger than a pre-defined threshold. **Leader-follower clustering** is one algorithm used here.

Two versions exists: (1) without input/weight normalisation, (2) with input/weight normalisation

Pseudo Code: Algorithm for Basic leader-follower clustering **without normalisation**

```
begin initialise  $\eta (> 0)$ ,  $\theta \leftarrow$  threshold
   $\mathbf{m}_1 \leftarrow \mathbf{x}$  (create first cluster)
  do accept new  $\mathbf{x}$ ;
     $j \leftarrow \arg \min_{j'} \|\mathbf{x} - \mathbf{m}_{j'}\|$  (find the nearest cluster)
    if  $\|\mathbf{x} - \mathbf{m}_j\| < \theta$ 
       $\mathbf{m}_j \leftarrow \mathbf{m}_j + \eta(\mathbf{x} - \mathbf{m}_j)$  (update cluster centre)
    else
      add new  $\mathbf{m} \leftarrow \mathbf{x}$  (create a new cluster  $\mathbf{m}$ )
    end
  until no significant change in  $\mathbf{m}_1, \mathbf{m}_2, \dots$ 
  return  $\mathbf{m}_1, \mathbf{m}_2, \dots$ 
end
```

Pseudo Code: Algorithm for Basic leader-follower clustering **with normalisation**

```
begin initialise  $\eta (> 0)$ ,  $\theta \leftarrow$  threshold
   $\mathbf{x}_i \leftarrow [1; \mathbf{x}_i]$ ,  $i = 1, \dots, n$  (augment all patterns)
   $\mathbf{x}_i \leftarrow \mathbf{x}_i / \|\mathbf{x}_i\|$ ,  $i = 1, \dots, n$  (normalise all patterns)
   $\mathbf{m}_1 \leftarrow \mathbf{x}$  (create first cluster)
  do accept new  $\mathbf{x}$  (augmentation and normalisation have to be done)
     $j \leftarrow \arg \max_{j'} \mathbf{m}_j^T \mathbf{x}$  (find the nearest cluster)
    if  $\|\mathbf{x} - \mathbf{m}_{j'}\| < \theta$ 
       $\mathbf{m}_j \leftarrow \mathbf{m}_j + \eta \mathbf{x}$  (update cluster centre)
       $\mathbf{m}_j \leftarrow \mathbf{m}_j / \|\mathbf{m}_j\|$  (normalise weight)
    else
      add new  $\mathbf{m} \leftarrow \mathbf{x}$  (create a new cluster  $\mathbf{m}$ )
       $\mathbf{m} \leftarrow \mathbf{m} / \|\mathbf{m}\|$  (normalise weight)
    end
  until no significant change in  $\mathbf{m}_1, \mathbf{m}_2, \dots$ 
  return  $\mathbf{m}_1, \mathbf{m}_2, \dots$ 
end
```

11. Glossary