Introduction to Neural Networks : Revision Lectures

1. Module Aims and Learning Outcomes
2. Biological and Artificial Neural Networks
3. Training Methods for Multi Layer Perceptrons
4. Bias + Variance and Improving Generalization
5. Applications of Multi-Layer Perceptrons
6. Radial Basis Function Networks
7. Committee Machines
8. Self Organizing Maps and Learning Vector Quantization

L1 : Module Aims and Learning Outcomes

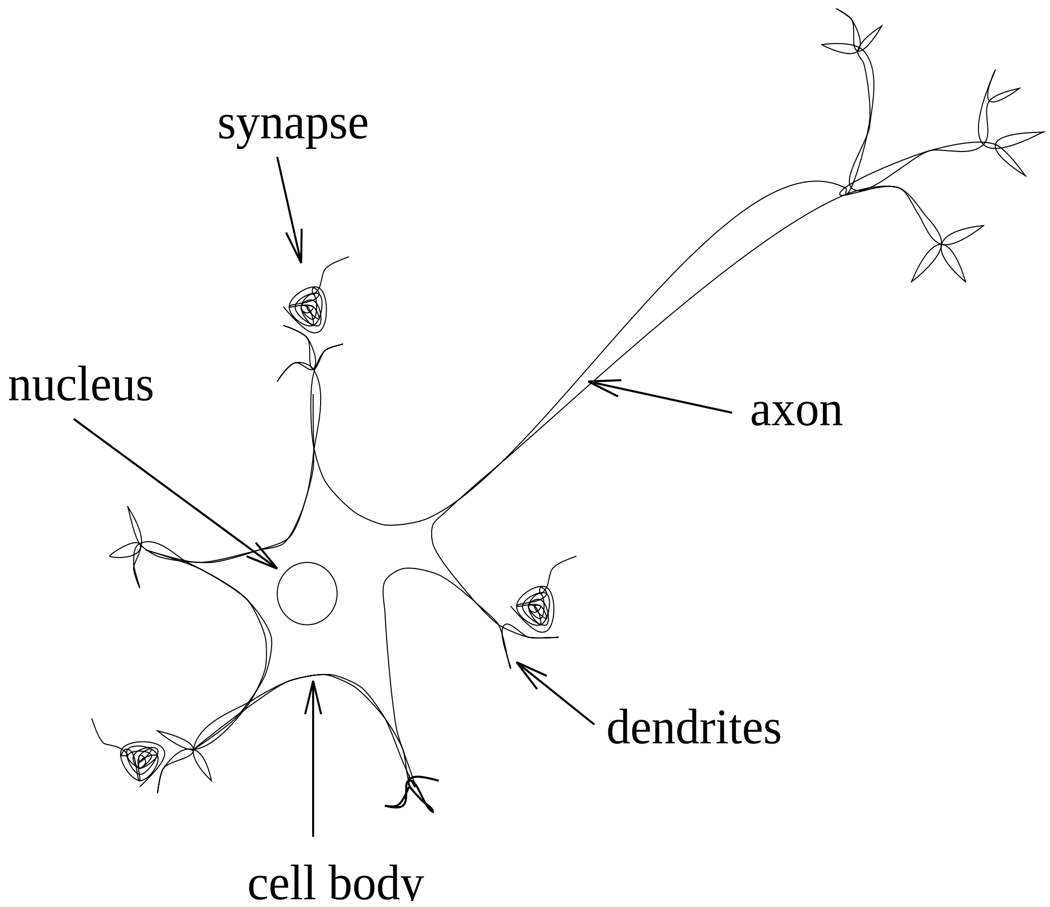
Aims

1. Introduce some of the fundamental techniques and principles of neural network systems.
2. Investigate some common models and their applications.

Learning Outcomes

1. Understand the relation between real brains and simple artificial neural network models.
2. Describe and explain the most common architectures and learning algorithms for Multi-Layer Perceptrons, Radial-Basis Function Networks, Committee Machines, and Kohonen Self-Organising Maps.
3. Explain the learning and generalisation aspects of neural network systems.
4. Demonstrate an understanding of the implementational issues for common neural network systems.
5. Demonstrate an understanding of the practical considerations in applying neural networks to real classification, recognition and approximation problems.

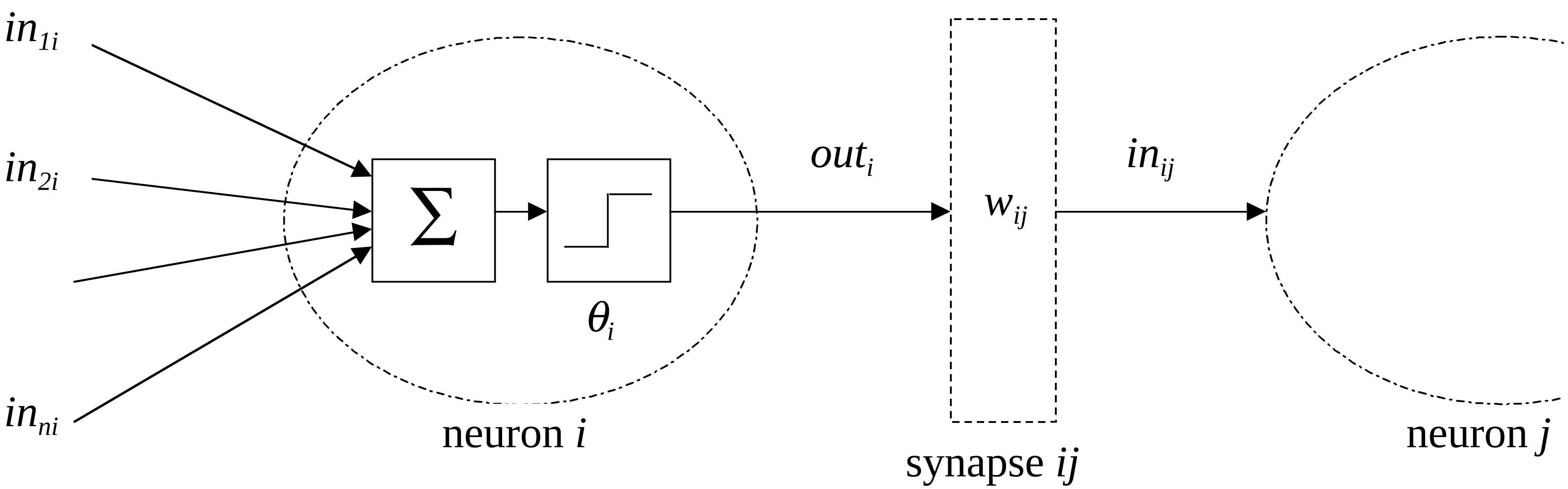
L2 : Biological Neural Networks



1. The majority of ***neurons*** encode their outputs or activations as a series of brief electical pulses (i.e. spikes or action potentials).
2. ***Dendrites*** are the receptive zones that receive activation from other neurons.
3. The ***cell body (soma)*** of the neuron's processes the incoming activations and converts them into output activations.
4. ***Axons*** are transmission lines that send activation to other neurons.
5. ***Synapses*** allow weighted transmission of signals (using ***neurotransmitters***) between axons and dendrites to build up large neural networks.

L3 : Networks of McCulloch-Pitts Neurons

Artificial neurons have the same basic components as biological neurons. The simplest ANNs consist of a set of ***McCulloch-Pitts neurons*** labelled by indices k, i, j and activation flows between them via synapses with strengths $w_{ki,}$ w_{ij}:



$$in_{ki} = out_k w_{ki} \qquad out_i = \mathrm{sgn}\Big(\sum_{k=1}^{n} in_{ki} - \theta_i\Big) \qquad in_{ij} = out_i w_{ij}$$

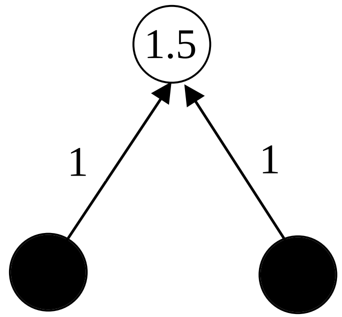
Implementation of Simple Logic Gates

We have inputs in_i and output $out = \mathrm{sgn}(w_1\, in_1 + w_2\, in_2 - \theta)$ and need to solve for w_1 and θ:

AND

in_1	in_2	out
0	0	0
0	1	0
1	0	0
1	1	1

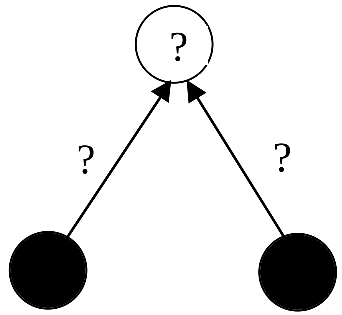
$$w_1\, 0 + w_2\, 0 - \theta < 0$$
$$w_1\, 0 + w_2\, 1 - \theta < 0$$
$$w_1\, 1 + w_2\, 0 - \theta < 0$$
$$w_1\, 1 + w_2\, 1 - \theta > 0$$
$$\boldsymbol{\theta > 0 \;,\; w_1 , w_2 < \theta \;,\; w_1 + w_2 > \theta}$$



XOR

in_1	in_2	out
0	0	0
0	1	1
1	0	1
1	1	0

$$w_1\, 0 + w_2\, 0 - \theta < 0$$
$$w_1\, 0 + w_2\, 1 - \theta > 0$$
$$w_1\, 1 + w_2\, 0 - \theta > 0$$
$$w_1\, 1 + w_2\, 1 - \theta < 0$$
$$\boldsymbol{\theta > 0 \;,\; w_1 , w_2 > \theta \;,\; w_1 + w_2 < \theta}$$



Solutions only exist for ***linearly separable*** problems, but since the simple gates (AND, OR, NOT) can be linked together to solve arbitrarily complex mappings, they are very powerful.

Building an Artificial Neural Network

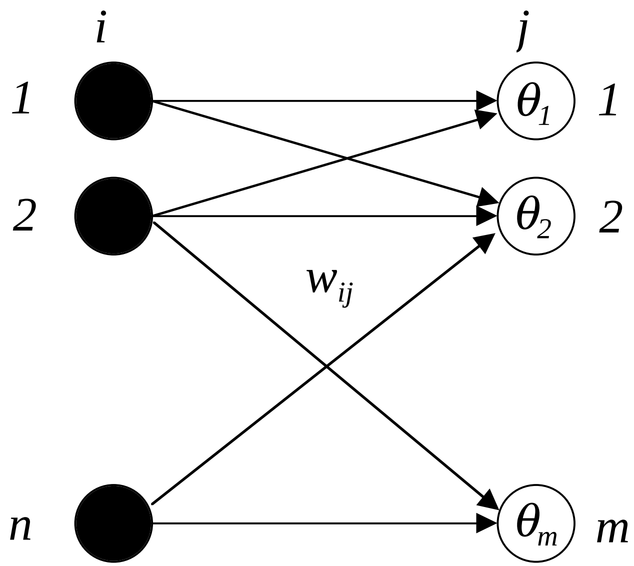
Using artificial neural networks to solve real problems is a multi-stage process:

1. Understand and specify the problem in terms of inputs and required outputs.
2. Take the simplest form of network that might be able to solve the problem.
3. Try to find appropriate connection weights and neuron thresholds so that the network produces appropriate outputs for each input in its training data.
4. Test the network on its training data, and also on new (validation/testing) data.
5. If the network doesn't perform well enough, go back to stage 3 and work harder.
6. If the network still doesn't perform well enough, go back to stage 2 and work harder.
7. If the network still doesn't perform well enough, go back to stage 1 and work harder.
8. Problem solved – move on to next problem.

After training, the network is usually expected to ***generalize*** well, i.e. produce appropriate outputs for ***test patterns*** it has never seen before.

L4 : The Perceptron and the Perceptron Learning Rule

An arrangement of one input layer of activations feeding forward to one output layer of McCulloch-Pitts neurons is known as a simple ***Perceptron***:



Network Activations:

$$out_j = \text{sgn}\left(\sum_{i=1}^{n} in_i w_{ij} - \theta_j\right)$$

Perceptron Learning Rule:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

$$\Delta w_{ij} = \eta.(targ_j - out_j).in_i$$

The ***Perceptron Learning Rule*** iteratively shifts around the weights w_{ij} and hence the decision boundaries to give the target outputs for each input. If the problem is ***linearly separable***, the required weights will be found in a finite number of iterations.

L5 : Learning by Gradient Descent Error Minimisation

The Perceptron learning rule is an algorithm that adjusts the network weights w_{ij} to minimise the difference between the actual outputs out_j and the target outputs $targ_j^p$. We can quantify this difference by defining the ***Sum Squared Error*** function, summed over all output units j and all training patterns p:

$$E(w_{mn}) = \tfrac{1}{2}\sum_p \sum_j \left(targ_j^p - out_j(in_i^p)\right)^2$$

It is the general aim of network ***learning*** to minimise this error by adjusting the weights w_{mn}. Typically we make a series of small adjustments to the weights $w_{mn} \rightarrow w_{mn} + \Delta w_{mn}$ until the error $E(w_{mn})$ is 'small enough'. We can determine which direction to change the weights in by looking at the gradients (i.e. partial derivatives) of E with respect to each weight w_{mn}. Then the ***gradient descent update equation*** (with positive learning rate η) is

$$\Delta w_{kl} = -\eta \frac{\partial E(w_{mn})}{\partial w_{kl}}$$

which can be applied iteratively to minimise the error.

L6 : Practical Considerations for Gradient Descent Learning

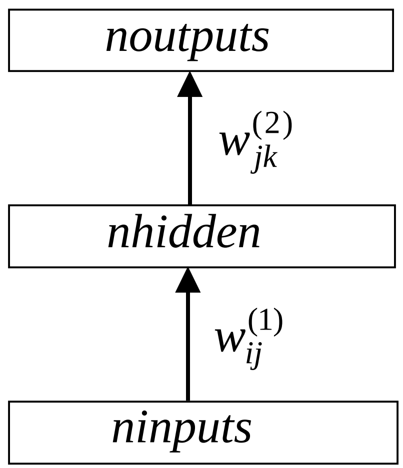
There a number of important practical/implementational considerations that must be taken into account when training neural networks:

1. Do we need to pre-process the training data? If so, how?
2. How may hidden units do we need?
3. Are some activation functions better than others?
4. How do we choose the initial weights from which we start the training?
5. Should we have different learning rates for the different layers?
6. How do we choose the learning rates?
7. Do we change the weights after each training pattern, or after the whole set?
8. How do we avoid flat spots in the error function?
9. How do we avoid local minima in the error function?
10. When do we stop training?

In general, the answers to these questions are highly problem dependent.

L7 : Multi-Layer Perceptrons (MLPs)

To deal with non-linearly separable problems (such as XOR) we can use non-monotonic activation functions. More conveniently, we can instead extend the simple Perceptron to a ***Multi-Layer Perceptron***, which includes a least one hidden layer of neurons with ***non-linear*** activations functions *f(x)* (such as sigmoids):



$$out_k^{(2)} = f\Big(\sum_j out_j^{(1)} w_{jk}^{(2)}\Big)$$

$$out_j^{(1)} = f\Big(\sum_i out_i^{(0)} w_{ij}^{(1)}\Big)$$

$$out_i^{(0)} = in_i$$

Note that if the activation on the hidden layer were linear, the network would be equivalent to a single layer network, and wouldn't be able to cope with non-linearly separable problems.

The Back-Propagation Learning Algorithm

By computing the necessary partial derivatives using the ***chain rule*****,** we obtain the gradient descent weight update equation for an N layer MLP:

$$\Delta w_{hl}^{(n)} = -\eta \partial E(w_{jk}^{(n)}) \big/ \partial w_{jk}^{(n)} = \eta \sum_{p} delta_l^{(n)}.out_h^{(n-1)}$$

in which the error signal $delta_k^{(N)}$ at the output layer N is simply the difference between the target and actual outputs times the derivative of the output activation function:

$$delta_k^{(N)} = \left(targ_k - out_k^{(N)}\right).f'\left(\sum_j out_j^{(1)} w_{jk}^{(N)}\right) = \left(targ_k - out_k^{(N)}\right).out_k^{(N)}.\left(1 - out_k^{(N)}\right)$$

and these error signals ***propagate back*** to give the *delta*s at earlier layers n:

$$delta_k^{(n)} = \left(\sum_k delta_k^{(n+1)}.w_{lk}^{(n+1)}\right).f'\left(\sum_j out_j^{(n-1)} w_{jk}^{(n)}\right) = \left(\sum_k delta_k^{(n+1)}.w_{lk}^{(n+1)}\right).out_k^{(n)}.\left(1 - out_k^{(n)}\right)$$

This is the famous ***Back-Propagation*** learning algorithm for MLPs.

Training a Two-Layer MLP Network

The procedure for training a two layer MLP is now quite straight-forward:

1. Take the set of training (input – output) patterns the network is required to learn
   $\{in_i^p, out_j^p : i = 1 \ldots ninputs, j = 1 \ldots noutputs, p = 1 \ldots npatterns\}$.
2. Set up a network with *ninputs* input units fully connected to *nhidden* hidden units via connections with weights $w_{ij}^{(1)}$, which in turn are fully connected to *noutputs* output units via connections with weights $w_{jk}^{(2)}$.
3. Generate random initial connection weights, e.g. from the range [–*smwt*, +*smwt*]
4. Select an appropriate error function $E(w_{jk}^{(n)})$ and learning rate η.
5. Apply the gradient descent weight update equation $\Delta w_{jk}^{(n)} = -\eta \partial E(w_{jk}^{(n)}) / \partial w_{jk}^{(n)}$ to each weight $w_{jk}^{(n)}$ for each training pattern *p*. One set of updates of all the weights for all the training patterns is called one *epoch* of training.
6. Repeat step 5 until the network error function is ‘small enough’.

The extension to networks with more hidden layers is straightforward.

L8 : Improvements Over Back-Propagation

We can smooth out back-propagation updates by adding a ***momentum*** term $\alpha.\Delta w_{hl}^{(n)}(t-1)$ so

$$\Delta w_{hl}^{(n)}(t) = \eta \sum_{p} delta_l^{(n)}(t).out_h^{(n-1)}(t) + \alpha.\Delta w_{hl}^{(n)}(t-1) \ .$$

Another way to speed up learning is to compute good step sizes at each step of gradient descent by doing a ***line search*** along the gradient direction to give the best step *size(t)*, so

$$\Delta w_{hl}^{(n)}(t) = size(t).dir_{hl}^{(n)}(t)$$

There are efficient parabolic interpolation methods for doing the line searches.

A problem with using line searches on true gradient descent directions is that the subsequent steps are orthogonal, and this can cause unnecessary zig-zagging through weight space. The ***Conjugate Gradients*** learning algorithm computes better directions $dir_{hl}^{(n)}(t)$ than true gradients and then steps along them by amounts determined by line searches. This is probably the best general purpose approach to MLP training.

L9 : Bias and Variance

If we define the expectation or average operator $\mathcal{E}_D$ which takes the ***ensemble average*** over all possible training sets *D*, then some rather messy algebra allows us to show that:

$$\mathcal{E}_D\Big[\big(\mathcal{E}[y\,|\,x_i] - net(x_i, W, D)\big)^2\Big]$$

$$= \big(\mathcal{E}_D\big[net(x_i, W, D)\big] - \mathcal{E}[y\,|\,x_i]\big)^2 + \mathcal{E}_D\Big[\big(net(x_i, W, D) - \mathcal{E}_D\big[net(x_i, W, D)\big]\big)^2\Big]$$

$$= \qquad (\text{bias})^2 \qquad + \qquad (\text{variance})$$

This error function consists of two positive components:

(bias)^2 : the difference between the average network output $\mathcal{E}_D[net(x_i, W, D)]$ and the regression function $g(x_i) = \mathcal{E}[y\,|\,x_i]$. This can be viewed as the *approximation error*.

(variance) : the variance of the approximating function $net(x_i, W, D)$ over all the training sets *D*. It represents the *sensitivity* of the results on the particular choice of data *D*.

In practice there will always be a trade-off to get the best generalization.

L10 : Improving Generalization

For networks to generalize well they need to avoid both under-fitting of the training data (high statistical bias) and over-fitting of the training data (high statistical variance).

There are a number of approaches to ***improving generalization*** – we can:

1. Arrange to have the optimum number of free parameters (independent connection weights) in the network (e.g. by fixing the number of hidden units, or weight sharing).
2. Stop the gradient descent training process just before over-fitting starts.
3. Add a regularization term $\lambda\Omega$ to the error function to smooth out the mappings that are learnt (e.g. the regularizer $\Omega = -\,{}^{1}/_{2} \sum (w_{ij})^{2}$ which corresponds to weight decay).
4. Add noise (or jitter) to the training patterns to smooth out the data points.

We can use a ***validation set*** or ***cross-validation*** as a way of estimating the generalization using only the available training data. This provides a way of optimizing any of the above procedures (e.g. the regularization parameter λ) to improve generalization.

L11: Applications of Multi-Layer Perceptrons

Neural network applications fall into two basic types:

Brain modelling The scientific goal of building models of how real brains work. This can potentially help us understand the nature of human intelligence, formulate better teaching strategies, or better remedial actions for brain damaged patients.

Artificial System Building The engineering goal of building efficient systems for real world applications. This may make machines more powerful, relieve humans of tedious tasks, and may even improve upon human performance.

We often use exactly the same networks and techniques for both. Frequently progress is made when the two approaches are allowed to feed into each other. There are fundamental differences though, e.g. the need for biological plausibility in brain modelling, and the need for computational efficiency in artificial system building. Simple neural networks (MLPs) are surprisingly effective for both. Brain models need to cover Development, Adult Performance, and Brain Damage. Real world applications include: Data Compression, Time Series Prediction, Speech Recognition, Pattern Recognition and Computer Vision.

L12 : Radial Basis Function (RBF) Mappings

Consider a set of N data points in a multi-dimensional space with D dimensional inputs $\mathbf{x}^p = \{x_i^p : i = 1,...,D\}$ and corresponding K dimensional target outputs $\mathbf{t}^p = \{t_k^p : k = 1,...,K\}$. That output data will generally be generated by some underlying functions $g_k(\mathbf{x})$ plus random noise. The goal here is to approximate the $g_k(\mathbf{x})$ with functions $y_k(\mathbf{x})$ of the form

$$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x})$$

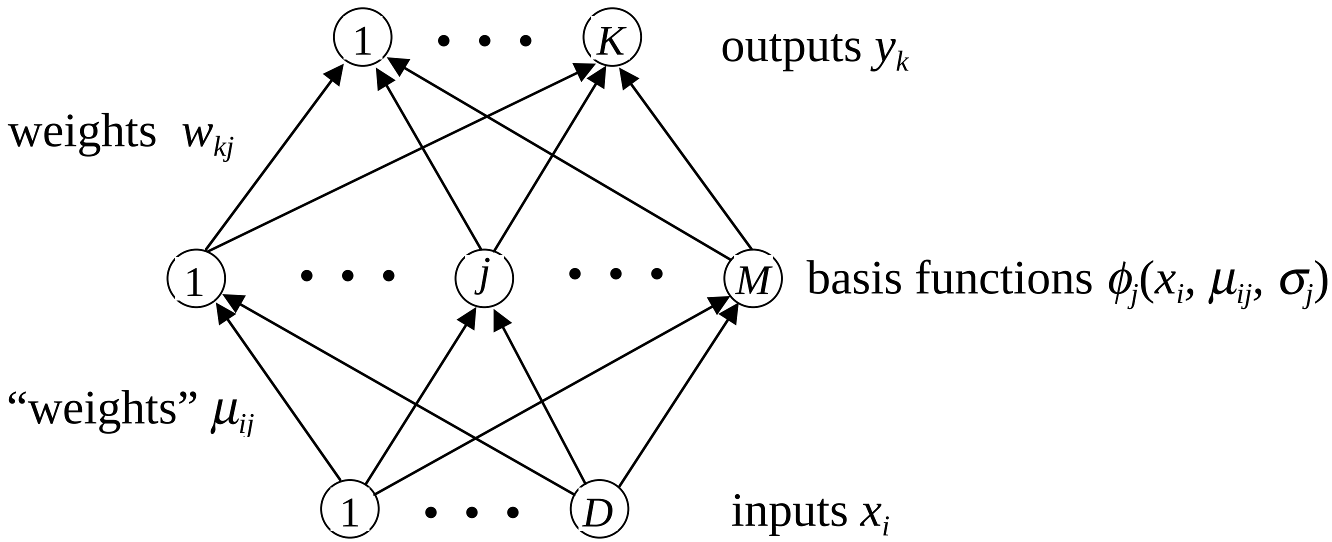
There are good computational reasons to use Gaussian basis functions

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{\left\|\mathbf{x} - \boldsymbol{\mu}_j\right\|^2}{2\sigma_j^{\,2}}\right)$$

in which we have basis centres $\{\boldsymbol{\mu}_j\}$ and widths $\{\sigma_j\}$. If $M = N$ we can use matrix inversion techniques to perform ***exact interpolation***. But this would be computationally inefficient and not give good generalization. It is better to take a different approach with $M << N$.

L13,14 : RBF Networks and Their Training

We can cast the RBF mapping into a form that looks like a neural network:



First the basis centres $\{\boldsymbol{\mu}_j\}$ and widths $\{\sigma_j\}$ can be obtained by unsupervised methods (e.g. centres at random training points with widths to match). The output weights $\{w_{kj}\}$ can then be found analytically by solving a set of linear equations. This makes the training very quick, with no difficult to optimise learning parameters, which is a major advantage over MLPs.

L15 : Committee Machines

Committee machines are combinations of two or more neural networks that can be made to perform better than individual networks. There are two major categories:

1. Static Structures

The outputs of several constituent networks (experts) are combined by a mechanism that does not involve the input signal, hence the designation *static*. Examples include

- ***Ensemble averaging***, where the constituent outputs are linearly combined.
- ***Boosting***, where weak learners are combined to give a strong learner.

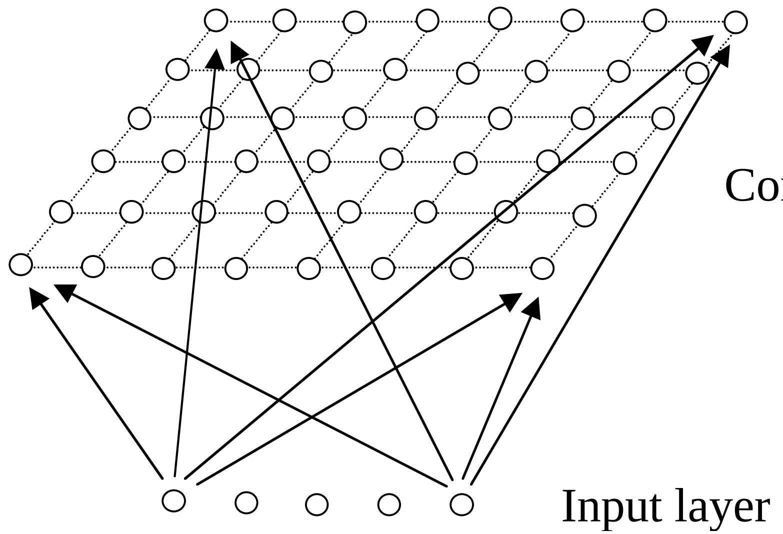
2. Dynamic structures

The input signal is directly involved in actuating the mechanism that integrates/combines the constituent outputs, hence the designation *dynamic*. The main example is

- ***Mixtures of experts***, where the constituent outputs are non-linearly combined by some form of gating system (which may itself be a neural network).

L16,17 : The Kohonen Self Organizing Map (SOM)

The ***SOM*** is an ***unsupervised training*** system based on ***competitive learning***. The aim is to learn a ***feature map*** from a spatially ***continuous input space***, in which our input vectors live, to a low dimensional spatially ***discrete output space*** formed by arranging the computational neurons into a grid that is fully connected to all the input layer neurons.



This provides an approximation of the input space with dimensional reduction, topological ordering, density matching, and feature selection.

Components of Self Organization

The self-organization process has four major components:

***Initialization*:** All the connection weights are initialized with small random values.

***Competition*:** For each input pattern, each output nodes compute their respective values of a *discriminant function* which provides the basis for competition. Simple Euclidean distance between the input vector and the weight vector for each output node is suitable. The particular neuron with the smallest distance is declared the *winner*.

***Cooperation*:** The winning neuron determines the spatial location of a *topological neighbourhood* of excited neurons, thereby providing the basis for cooperation among neighbouring neurons.

***Adaptation*:** The excited neurons increase their individual values of the discriminant function in relation to the input pattern through suitable adjustment to the associated connection weights, such that the response of the winning neuron to the subsequent application of a similar input pattern is enhanced.

The SOM Algorithm

The self organising process is implemented in the SOM algorithm:

1. ***Initialization*** – Choose random values for the initial weight vectors $\mathbf{w}_j$.

2. ***Sampling*** – Draw a sample training input vector $\mathbf{x}$ from the input space.

3. ***Matching*** – Find the winning neuron $I(\mathbf{x})$ that has weight vector closest to the input vector, i.e. the minimum value of the discriminant function $d_j(\mathbf{x}) = \sum_{i=1}^{D} (x_i - w_{ji})^2$.

4. ***Updating*** – Apply the weight update equation $\Delta w_{ji} = \eta(t)\, T_{j,I(\mathbf{x})}(t)\, (x_i - w_{ji})$ where $T_{j,I(\mathbf{x})}(t) = \exp(-S_{j,I(\mathbf{x})} / 2\sigma^2(t))$ is the Gaussian topological neighbourhood around the winning node $I(\mathbf{x})$ defined by the distance $S_{j,I(\mathbf{x})}$ between nodes j and $I(\mathbf{x})$ on the output grid. $\sigma(t)$ is the Gaussian's width and $\eta(t)$ is the learning rate, both of which generally decrease with time (e.g. exponentially).

5. ***Continuation*** – keep returning to step 2 until the feature map stops changing.

L18 : Learning Vector Quantization (LVQ)

The ***LVQ algorithm*** is a supervised process which starts from a trained SOM with input vectors $\{\mathbf{x}\}$ and weights (i.e. *Voronoi vectors*) $\{\mathbf{w}_j\}$. The classification labels of the inputs give the best classification for the nearest neighbour cell (i.e. *Voronoi cell*) for each $\mathbf{w}_j$,. It is unlikely that the cell boundaries (i.e. *Voronoi Tesselation*) will coincide with the classification boundaries. The LVQ algorithm attempts to correct this by shifting the boundaries:

1. If the input $\mathbf{x}$ and the associated Voronoi vector $\mathbf{w}_{I(\mathbf{x})}$ (i.e. the weight of the winning output node $I(\mathbf{x})$) have the same class label, then move them closer together by $\Delta\mathbf{w}_{I(\mathbf{x})}(t) = \beta(t)(\mathbf{x} - \mathbf{w}_{I(\mathbf{x})}(t))$ as in the SOM algorithm.
2. If the input $\mathbf{x}$ and associated Voronoi vector $\mathbf{w}_{I(\mathbf{x})}$ have the different class labels, then move them apart by $\Delta\mathbf{w}_{I(\mathbf{x})}(t) = -\beta(t)(\mathbf{x} - \mathbf{w}_{I(\mathbf{x})}(t))$.
3. Voronoi vectors $\mathbf{w}_j$ corresponding to other input regions are left unchanged with $\Delta\mathbf{w}_j(t) = 0$.

where $\beta(t)$ is a learning rate that decreases with the number of iterations/epochs of training. In this way we end up with better classification than by the SOM alone.

Overview and Reading

1. The module appears to have achieved its aims and learning outcomes.
2. We began by seeing how we could take simplified versions of the neural networks found in real brains to produce powerful computational devices.
3. We have seen how Multi-layer Perceptrons, Radial Basis Function Networks, Committee Machines, and Kohonen Self Organizing Maps can be set up and trained.
4. We have studied the issues underlying learning and generalization in neural networks, and how we can improve them both.
5. Along the way we have considered the various implementational and practical issues that might complicate our endeavours.

Reading

1. Your lecture notes!