



Information Technology

FIT5186 Intelligent Systems

Lecture 4

Supervised Learning - Backpropagation Learning Rule

Learning Objectives

- Understand
 - supervised learning and the backpropagation learning rule for multilayered feedforward neural networks
 - the training issues and parameter selection for multilayered feedforward neural networks
 - the advantages of neural networks
 - the practical issues for building neural networks
- Be able to
 - Build and train a neural network
 - build a neural network application using a neural network software system

Preliminaries

net: the total input to a neuron

$$net = \sum_{i=1}^n w_i x_i$$

- Recall from Lecture 2, the general equation for learning (weight adaptation)

$$w(t+1) = w(t) + c r(\mathbf{w}, \mathbf{x}, \mathbf{d}) \mathbf{x}(t)$$

\mathbf{w} : weight vector

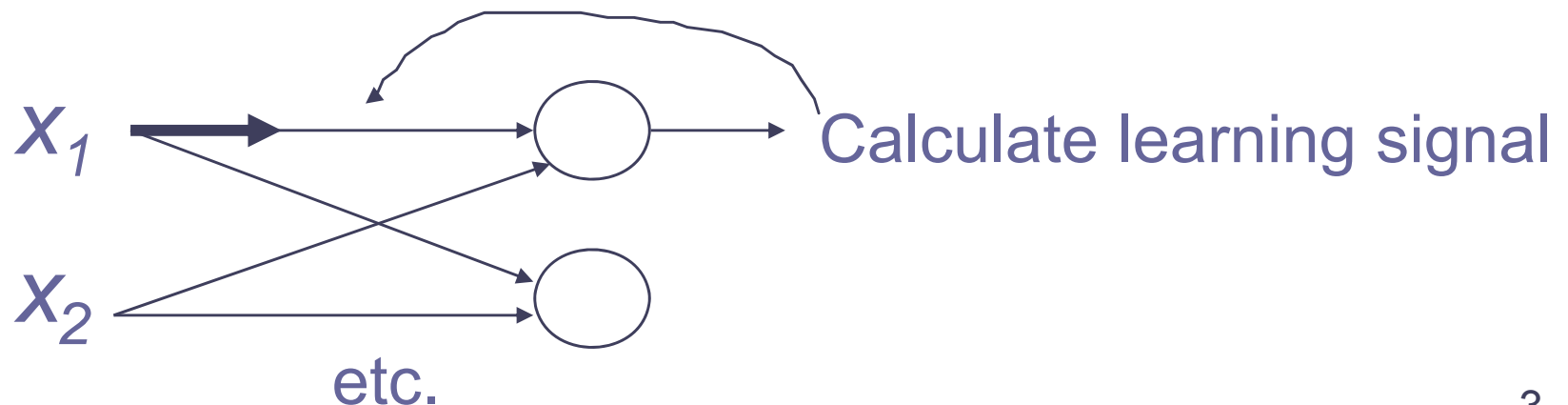
\mathbf{x} : input signal

\mathbf{d} : desired output

Learning constant
(determines the
rate of learning)

Learning signal
(determines the
type of learning)

Input to the output
(destination) neuron



net: the total input to a neuron

$$net = \sum_{i=1}^n w_i x_i$$

Preliminaries (continued)

- Learning signals are based on
 - Outputs (for unsupervised learning) or
 - Errors = desired outputs - actual (observed) outputs ($d - o$) (for supervised learning)
 - Learning signals are used to modify the weight so that the output of the network is correct.

- Backpropagation learning rule is based on the Delta learning rule:

$$r = (d - o) \frac{df(net)}{d(net)}$$

- What is $\frac{df(net)}{d(net)}$?

Derivative of $f(net)$

Differentiating the activation function

$$f(net) = \frac{1}{1 + e^{-\lambda net}}$$

$$\frac{df(net)}{d(net)} = \frac{\lambda e^{-\lambda net}}{(1 + e^{-\lambda net})^2}$$

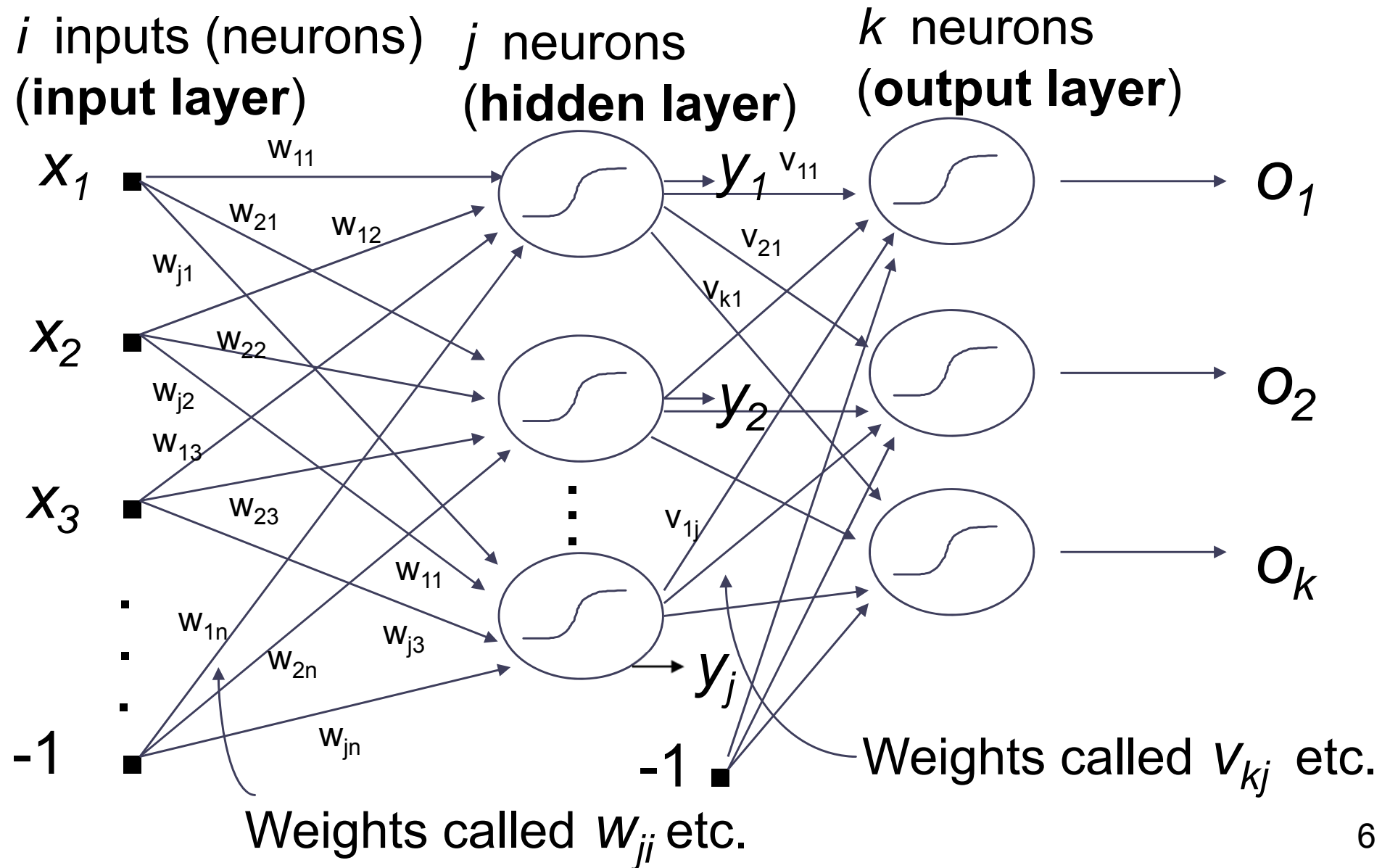
Preliminaries (continued)

- Now,
$$\frac{\lambda e^{-\lambda net}}{(1 + e^{-\lambda net})^2} = \frac{1}{1 + e^{-\lambda net}} \cdot \frac{\lambda e^{-\lambda net}}{1 + e^{-\lambda net}}$$
$$= f(net) \cdot f(net) \cdot \lambda \left(\frac{1}{f(net)} - 1 \right)$$
$$= f(net) \lambda (1 - f(net))$$

- So,
$$\frac{df(net)}{d(net)} = \lambda o (1 - o)$$

The learning signal $r = (d - o) \frac{df(net)}{d(net)} = \lambda (d - o) o (1 - o)$

Multilayered Feedforward Neural Network (MFNN)



Multilayered Feedforward Neural Network

- The architecture of MFNN shown in the previous slide is most used in business applications.
- The number of input neurons is the number of variables or factors being considered in the classification or prediction.
- The number of output neurons is the number of classes, or number of quantities being predicted.
 - These are often straightforward and are determined by the nature of the classification or prediction problem.
- The number of hidden neurons is a more experimental issue.
 - Too many hidden neurons will cause the MFNN to simply memorise the data, while too few hidden neurons may prevent the MFNN from learning any relationships among the variables.
 - To be discussed later.
- The learning rule for MFNNs is known as **the generalised delta rule** or **backpropagation learning rule**.

Delta Rule

- The Delta rule only works for a single layer of neurons.

$$r = (d - o) \frac{df(net)}{d(net)}$$

- We can use the Delta rule to calculate the weight changes for the weights in the output layer (using the outputs of the hidden layer (y_j) as the inputs to the output layer).

$$v_{kj} \leftarrow v_{kj} + \Delta v_{kj}$$

$$\begin{aligned} \text{where } \Delta v_{kj} &= c r_k^o y_j = c (d_k - o_k) \frac{df(net_k^o)}{d(net_k^o)} y_j \\ &= \lambda c o_k (d_k - o_k) (1 - o_k) y_j \end{aligned}$$

where r_k^o is the learning (error) signal for neuron k in the output layer.

Delta Rule - Backpropagation Learning

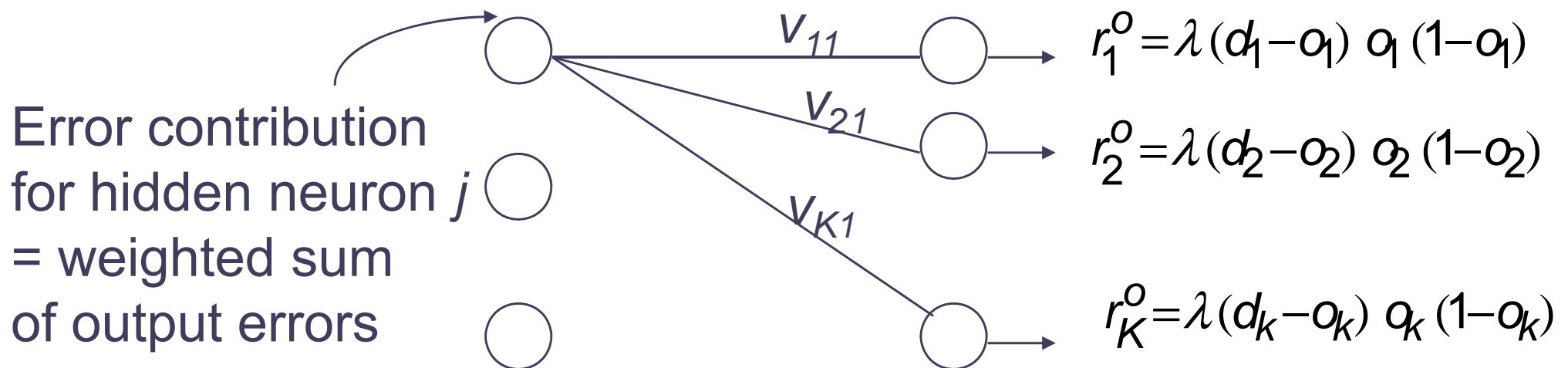
- To calculate the weight changes for *hidden layers*, we need to know the desired outputs of hidden neurons (and we don't).
- **Solution:** Since we know the final error, we should be able to propagate that error backwards to calculate the error at the hidden neurons - hence the name “**backpropagation**”.
- For the neurons in the hidden layer:
$$\Delta w_{ji} = c \cdot r_j^h \cdot x_i$$
 where r_j^h is the learning signal for neuron j in the hidden layer.

Delta Rule - Backpropagation Learning

- Using the Delta rule, we *should* have

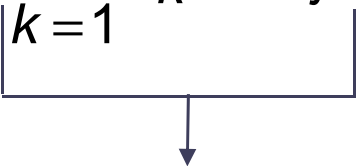
$$r_j^h = \lambda (d_j - y_j) y_j (1 - y_j)$$

- But how do we calculate r_j^h when we don't know d_j ?
 - Consider that each error $(d_j - y_j)$ contributes to the final error $(d_k - o_k)$



Delta Rule - Backpropagation Learning

- So the learning signal for the hidden neurons can be calculated as

$$r_j^h = \lambda \left(\sum_{k=1}^K r_k^o v_{kj} \right) y_j (1 - y_j)$$


instead of $(d_j - y_j)$

and

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

$$\text{where } \Delta w_{ji} = c r_j^h x_i$$

$$= c \lambda \left(\sum_{k=1}^K r_k^o v_{kj} \right) y_j (1 - y_j) x_i$$

Backpropagation Learning Algorithm (I)

- Step 1: Present an N dimensional input vector (pattern) \mathbf{x} to the input layer, including an extra input of -1 for the threshold
- Step 2: Calculate the net inputs to the hidden layer neurons

$$\text{net } h_j = \sum_{i=1}^{N+1} w_{ji} x_i$$

- Step 3: Calculate the outputs of the hidden layer neurons

$$y_j = f(\text{net } h_j) = \frac{1}{1 + e^{-\lambda \text{net } h_j}}$$

- Step 4: Calculate the net inputs to the output layer neurons

$$\text{net } o_k = \sum_{j=1}^{J+1} v_{kj} y_j$$

Backpropagation Learning Algorithm (II)

- Step 5: Calculate the outputs of the output layer neurons

$$o_k = f(\text{net}_k^o) = \frac{1}{1 + e^{-\lambda \text{net}_k^o}}$$

- Step 6: Calculate the learning signals for the output neurons

$$r_k^o = \lambda(d_k - o_k)o_k(1 - o_k)$$

- Step 7: Calculate the learning signals for the hidden neurons

$$r_j^h = \lambda\left(\sum_{k=1}^K r_k^o v_{kj}\right)y_j(1 - y_j)$$

Backpropagation Learning Algorithm (III)

- Step 8: Update the weights in the output neurons

$$v_k(t+1) = v_{kj}(t) + cr_k^o y_j(t) = v_{kj}(t) + c\lambda(d_k - o_k)o_k(1 - o_k)y_j(t)$$

- Step 9: Update the weights in the hidden neurons

$$w_{ji}(t+1) = w_{ji}(t) + cr_j^h x_i(t) = w_{ji}(t) + c\lambda\left(\sum_{k=1}^K r_k^o v_{kj}\right)y_j(1 - y_j)x_i(t)$$

- Step 10: Update the error $E \leftarrow E + \sum_{k=1}^K (r_k^o)^2$
for this epoch,
- Repeat from Step 1 with the next input vector \mathbf{x} until all input vectors (patterns) presented (1 epoch).
- At the end of each epoch, reset $E = 0$.

Repeat until $E < E_{max}$ (a pre-defined tolerance level, say 0.000001)

Backpropagation Learning for the XOR Problem

- As a major breakthrough in neural networks, the backpropagation learning algorithm enables the relationships between any sets of input patterns and desired outputs to be modelled.
- See **additional material for Lecture 4** for an example of learning the weights for the XOR problem.
- Make sure you understand the algorithm.
- Each step in the algorithm needs to be repeated for each input vector (pattern) in the training data set.
- Once all patterns have been presented (called the end of one **epoch**), the total error of the network across all patterns is measured.
- This error will reduce at each epoch until an acceptable small error is reached.
- Typically this takes thousands of epochs.

Training the Neural Network

- As an example, using the commercial NN software (NeuroShell 2,) it takes 6,810 epochs to train the network to solve the XOR problem (using $E_{max} = 0.00001$, $c = 1$, $\lambda = 1$).
 - This speed can be increased by varying the choice of parameters.
- **Training** is attempting to minimise the total average error for a complete presentation of the input vectors.
- Average error (over 1 epoch)

$$E_{av} = \frac{1}{P} \sum_{p=1}^P \sum_{k=1}^K (d_k(p) - o_k(p))^2$$

for input
pattern p



Training the Neural Network (continued)

- This average error reduces each epoch until it stabilises at a minimum (hopefully near 0), or training terminates because $E < E_{max}$.
- Analysing the weight adaptations and the effect they have on E_{ave} , shows that the backpropagation algorithm is a method of *steepest descent* with a step size determined by c .
- The steepest descent algorithm starts at some random point on the error surface of hills and valleys in multidimensional weigh space (given by the initial weights) and rolls straight down the steepest hill until the bottom of the valley is found.

Training the Neural Network (continued)

- Every step in the weight space (change in weights) results in a lower point on the error surface, until eventually no steps lead downhill.
- The valley found may not be the deepest valley, since it depends on where the algorithm starts.
- This could be either a local minimum $E > 0$ or (hopefully) a global minimum $E = 0$.
- A local minimum means that the network will always generate some error, and so the learning has not been perfect.
- In the XOR problem, a local minimum is encountered in approximately 1% of random starts.

Training Issues and Parameter Selection

- **Number of hidden neurons**

- The number of hidden neurons affects how well the network is able to separate the data.
- Local minima corresponds to 2 or more classes being categorised as the same.
- Poor internal representation of data: adding more hidden neurons helps to separate data.
- The network with too many hidden neurons may learn to correctly classify or predict the data it has been trained on, but its performance on new data (or its generalisability) may be poor.

Training Issues and Parameter Selection

- **Number of hidden neurons (continued)**

- How many hidden neurons are best?
 - Often a trial and error approach is needed.
 - Try to start with a modest number of hidden neurons and gradually increase the number if the network fails to reduce its error.
 - Remember: if the network fails to converge, try adding more hidden neurons; if it does converge try removing some (the network will run faster).

- The default formula used by NeuroShell 2 is

$$\frac{1}{2} (N + K) + \sqrt{P}$$

where N: the number of input dimensions

K: the number of output neurons

P: the number of patterns in the training set

- How many layers are best (3 or more)?

Training Issues and Parameter Selection

- **Number of hidden neurons (continued)**
 - Some commonly used rules to determine the best number of hidden neurons J

$$J = \frac{1}{2}(N + K) \quad \leftarrow \text{the most widely used}$$

$$J = \sqrt{N * K}$$

$$J = N + K$$

$$J = 2(N + K)$$

where N : the number of input dimensions

K : the number of output neurons

Nelson, M. and Illingworth, W.T. (1991). *A Practical Guide to Neural Nets*. Addison-Wesley, New York.

Training Issues and Parameter Selection

- **Choice of initial weights**

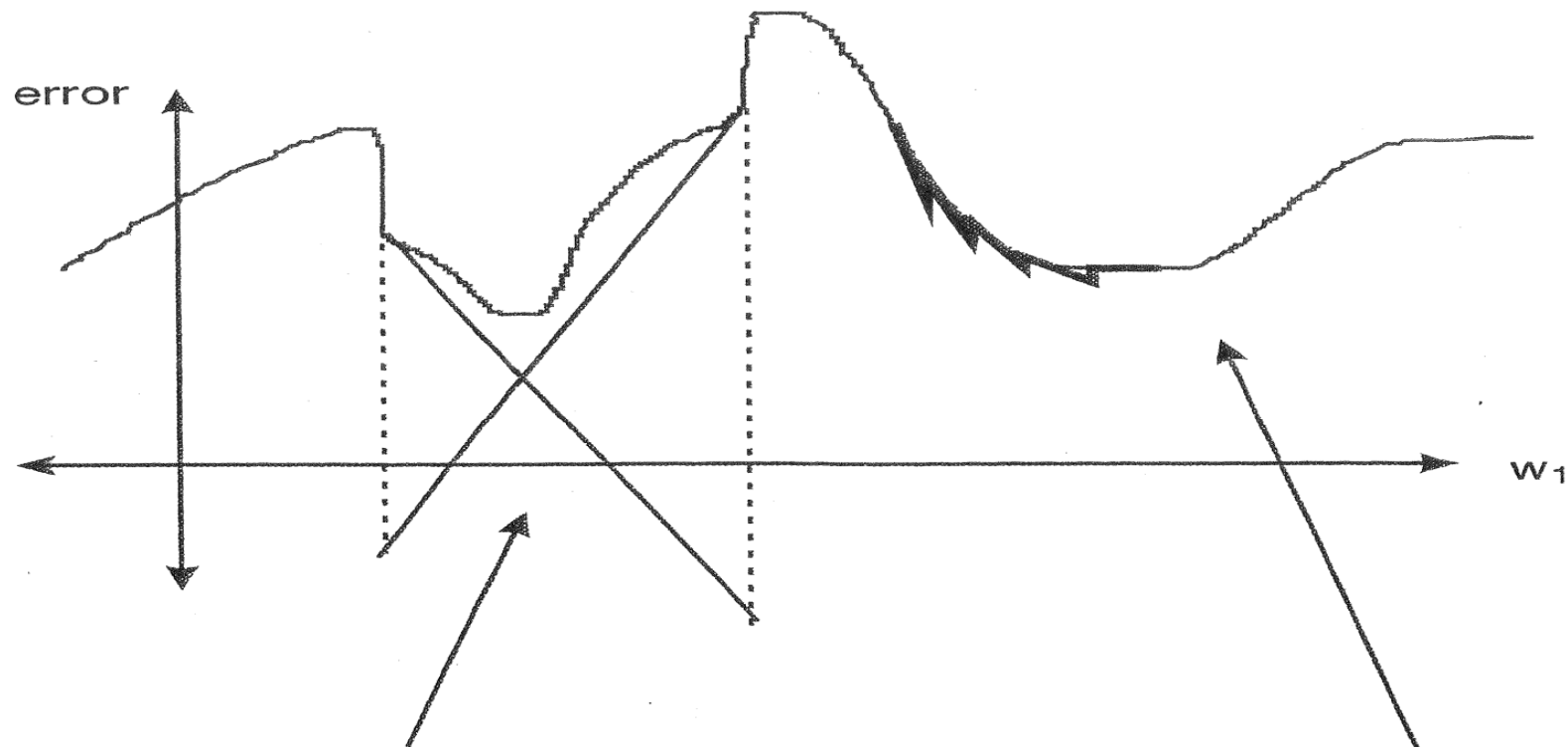
- Typically weights are initially small random values.
- Initialisation strongly affects the final solution.
- Try several different starting weight values to see if the network results are improved.

- **Choice of learning rate (constant) c**

- c is the step size in the steepest descent algorithm.
- Affects effectiveness and convergence of learning.
- Optimal choice depends on the problem.
- The default values are around 0.1.
- Somewhere in the range 0.001 to 10.

Training Issues and Parameter Selection

- Large c : quick convergence but may overshoot global minimum, or fail to converge at all.
- Small c : slow convergence but true steepest descent (will definitely converge).



a) large learning rate

b) small learning rate

Choice of Activation Function Parameter λ

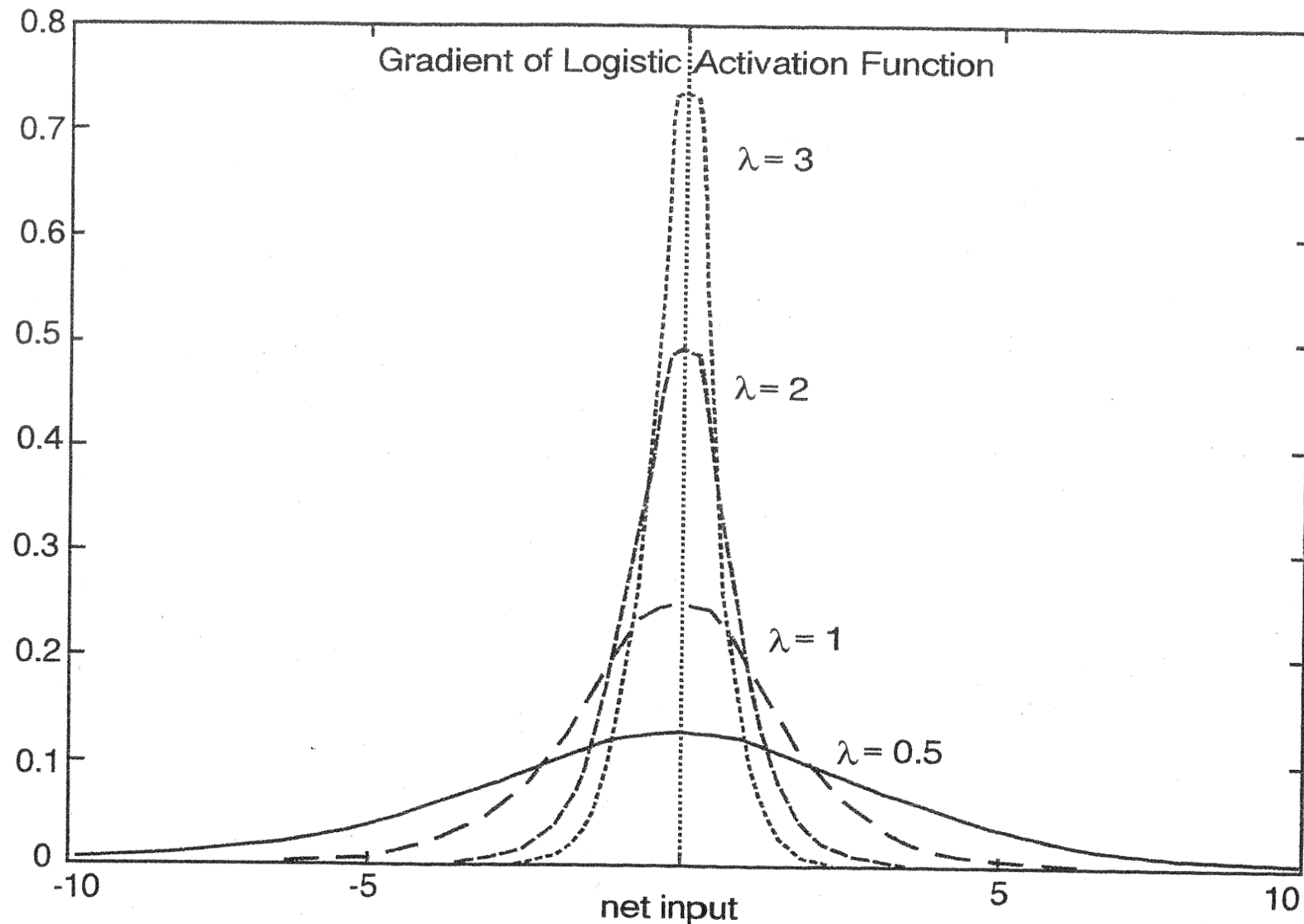
- Choice of activation function parameter λ

$$f(net) = \frac{1}{1 + e^{-\lambda net}}$$

- λ affects both output and error calculations, as it controls the steepness of the activation function.
- Recall that the learning signal is a function of the error ($d - o$) multiplied by the derivative or gradient of the activation function $df(net)/d(net)$.
- A larger λ value produces a larger gradient, thus giving a weight a greater amount of learning.
- See the next slide for its effect on learning.
- The most learning occurs where the net input is zero, with less learning occurring for large net inputs.

Choice of Activation Function Parameter λ

- Effect of the activation function parameter λ on learning



Add a Momentum Term

- **Add a momentum term**
 - Steepest descent tends to cause “zig-zagging” down the error surface.
 - Weight changes can be given some “momentum” by introducing an extra term to the weight adaptation rule.
 - If changes are currently large, then the next change will also be large (gains momentum).
 - If changes are currently small, then the next change will also be small (loses momentum).
 - The network will be less likely to get stuck in local minima early on, since momentum will push the changes over small hills.
 - Increases convergence speed.

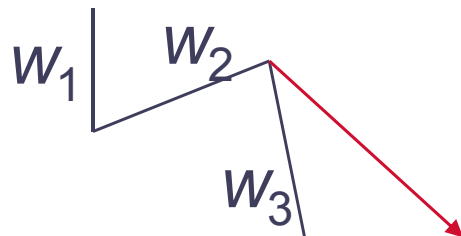
Add a Momentum Term (continued)

- New weight adaptation rules (at Steps 8 and 9 of the algorithm):

$$v_{kj}(t+1) = v_{kj}(t) + cr_k^o y_j + \alpha(v_{kj}(t) - v_{kj}(t-1))$$

$$w_{ji}(t+1) = w_{ji}(t) + cr_j^h x_i + \alpha(w_{ji}(t) - w_{ji}(t-1))$$

- α is the momentum factor where $0 < \alpha < 1$.
- A default value of around 0.1 is commonly used.
- Keep descent going in the same general direction (instead of zig-zagging).



w_3 with momentum

Advantages of Neural Networks

- **Generalisability**

Neural networks can learn the patterns and relationships amongst data (the training set), and then generalise these relationships to data they have never seen before (the test set).

- Certain features of the inputs are detected and coded into the weights of the network.
- If the network correctly classifies the test set data, then it has generalised its learning.

Advantages of Neural Networks (continued)

- **Fault Tolerance**

Neural networks are basically parallel processing elements:

- Should the information contained in a neuron or weight be damaged, the network can adapt itself to still function properly (much like the human brain).
 - Tolerant to “noisy” data: able to generalise noisy data to “clean” data classifications.
- So neural networks are ideally suited to problems such as pattern recognition, classification, prediction, and forecasting.

Practical Issues

- **Determine what is to be forecast, predicted or classified.**
 - Needs to be directly relevant to business decisions.
 - e.g. if you are trying to make a profit buying then selling stock, you may want to try to predict the future value of the stock.
 - Alternatively, you may just want to predict whether the value goes up or down (much easier).

Practical Issues (continued)

- **Collect data having a relationship to what is being forecasted or classified**
 - This information will be used to create forecast.
 - Today's temperature will have little bearing on tomorrow's stock price, but today's stock price is directly relevant to the forecast.
 - What else might be relevant?
 - How far back does historical data need to go?
- **Preprocess the data to make it more useful**
 - Combine data to save space (change in value rather than initial and final values).
 - Normalise data (removes bias).

Practical Issues (continued)

- **Extract test set data**

- Data set is divided into training set and test set, so that once the network has learnt the patterns in the training data, it can be tested on data it has never seen before.
- Usually around 10% of the data is extracted at random for the test set.
- After the network is trained with the training data, the network is presented with the test set to see how it performs (the generalisation of the network).
- If the network produces a large error on the test data, it has not learnt to generalise and it has simply memorised the training data. The network should be re-trained with modified parameter values to achieve generalisation.

Practical Issues (continued)

- **Select suitable network architecture**
 - Number of hidden layers, and number of neurons in each layer?
 - About 85% of all problems are trained on a backpropagation 3-layer network.
- **Select suitable training parameters**
 - Choose parameters c and λ .
 - Momentum method or straight?
 - Stopping criterion?
 - There are other variations of backpropagation.

Practical Issues (continued)

- **Train the network**
 - Present the training data as inputs.
 - Overtraining issues?
- **Test the network**
 - Present the test data as inputs.
 - If error is satisfactory, then the network is ready to be applied to create a forecast!

Does It Really Work?

- Often, the initial results are disappointing.
- The designer needs to rethink what is to be forecasted, the choice of inputs, and the pre-processing.
- Many software packages let us see which inputs contribute little to the forecast result.
 - Assists in rethinking.

Success Factors

- A higher success rate comes from:
 - A studied selection of the target output.
 - Selection of inputs having a predictive relationship to the target output.
 - Selection of an optimal neural network architecture.
 - Selection of an optimal training algorithm and parameters.

Week 4 Tutorial

NeuroShell 2 on-line Tutorial Example 1

- You will be using this software during the next 3 tutorials, as well as your assignment.
- The best way to explain how to use NeuroShell 2 is to build a neural network application.
- The application - how to create a network which will predict the electricity cost per day in a home.
- Work through the application step by step to learn to use the software.

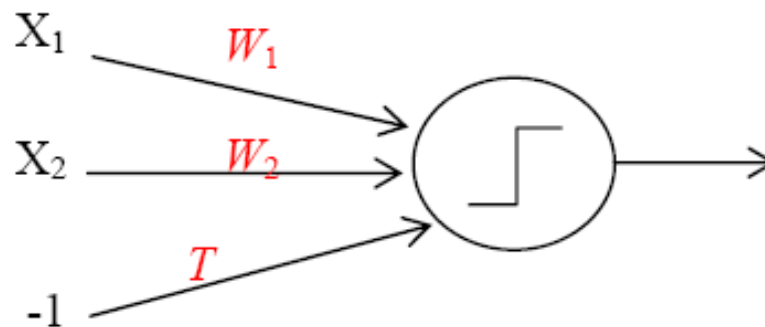
Summary of Lectures 2 – 4

Linear Separable Problems

- **Dichotomiser Perceptrons**

- One Single layer Perceptron with a discrete activation function

$$W_1X_1 + W_2X_2 - T = 0$$



- Use an additional input with a value of -1 and a weight of T in order to convert the firing threshold value of every neuron to be at zero.

Summary of Lectures 2 – 4

Linear Separable Problems (continued)

- **Multicategory Perceptrons (R-Category Classifier)**
 - R single layer Perceptrons with a discrete activation function
 - For example:
 - 3-category classifier using three discrete Perceptrons with desired outputs of (1 0 0), (0 1 0) and (0 0 1) respectively.
 - Only one output will be 1, the rest will be 0.

$$W_{11}X_1 + W_{12}X_2 - T_1 = 0$$

$$W_{21}X_1 + W_{22}X_2 - T_2 = 0$$

$$W_{31}X_1 + W_{32}X_2 - T_3 = 0$$

- Indecision regions
 - More than one perceptron outputs “1” or no perceptrons output “1”
- Single layer Perceptrons cannot solve linear non-separable problems.

Summary of Lectures 2 – 4

Linear Non-Separable Problems

- **Multilayered Perceptrons with a discrete activation function**
 - Neurons in the hidden layer are used to transfer the linear non-separable data to make it linearly separable.
 - **The credit assignment problem**

With the use of a discrete activation function, the Perceptrons in the output layer do not know what the effect of the hidden layer weights is, thus cannot assign credit or blame to these weights (in order to determine how they should be modified).
 - The solution is to use a continuous activation function rather than a discrete activation function.

Summary of Lectures 2 – 4

Linear Non-Separable Problems (continued)

- **Multilayered Feedforward Neural Networks (MFNN)**
 - Use a continuous activation function and the Delta learning rule.
 - The Delta rule cannot be directly applied to the hidden layer weights, because we don't know the desired outputs of the hidden neurons.
 - The solution is to use **the backpropagation learning algorithm**, which propagates the error backwards (from the output neurons) to calculate the error at the hidden neurons.
 - Training issues with the architecture and parameter selection.