Learning in Multi-Layer Perceptrons - Back-Propagation

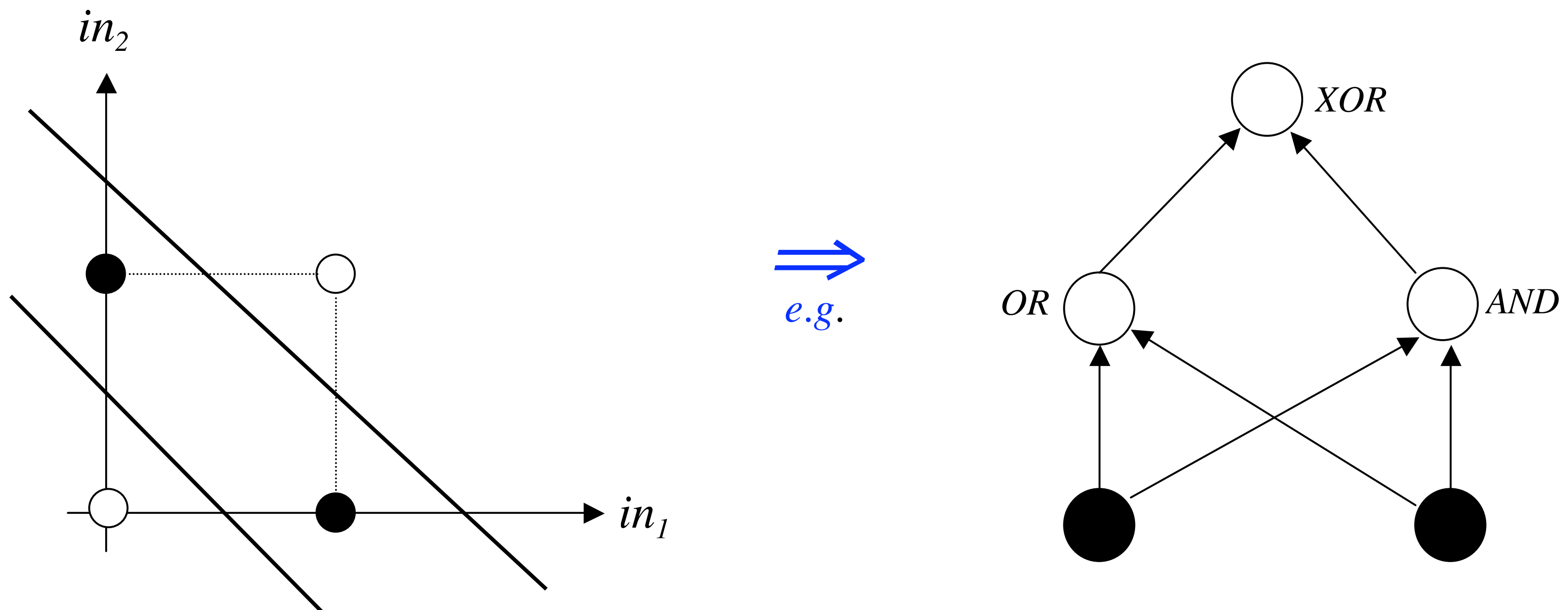
Neural Computation : Lecture 7

© John A. Bullinaria, 2015

1. Linear Separability and the Need for More Layers
2. Notation for Multi-Layer Networks
3. Multi-Layer Perceptrons (MLPs)
4. Learning in Multi-Layer Perceptrons
5. Choosing Appropriate Activation and Cost Functions
6. Deriving the Back-Propagation Algorithm
7. Further Practical Considerations for Training MLPs
   - (8) How Many Hidden Layers and Hidden Units?
   - (9) Different Learning Rates for Different Layers?

Linear Separability and the Need for More Layers

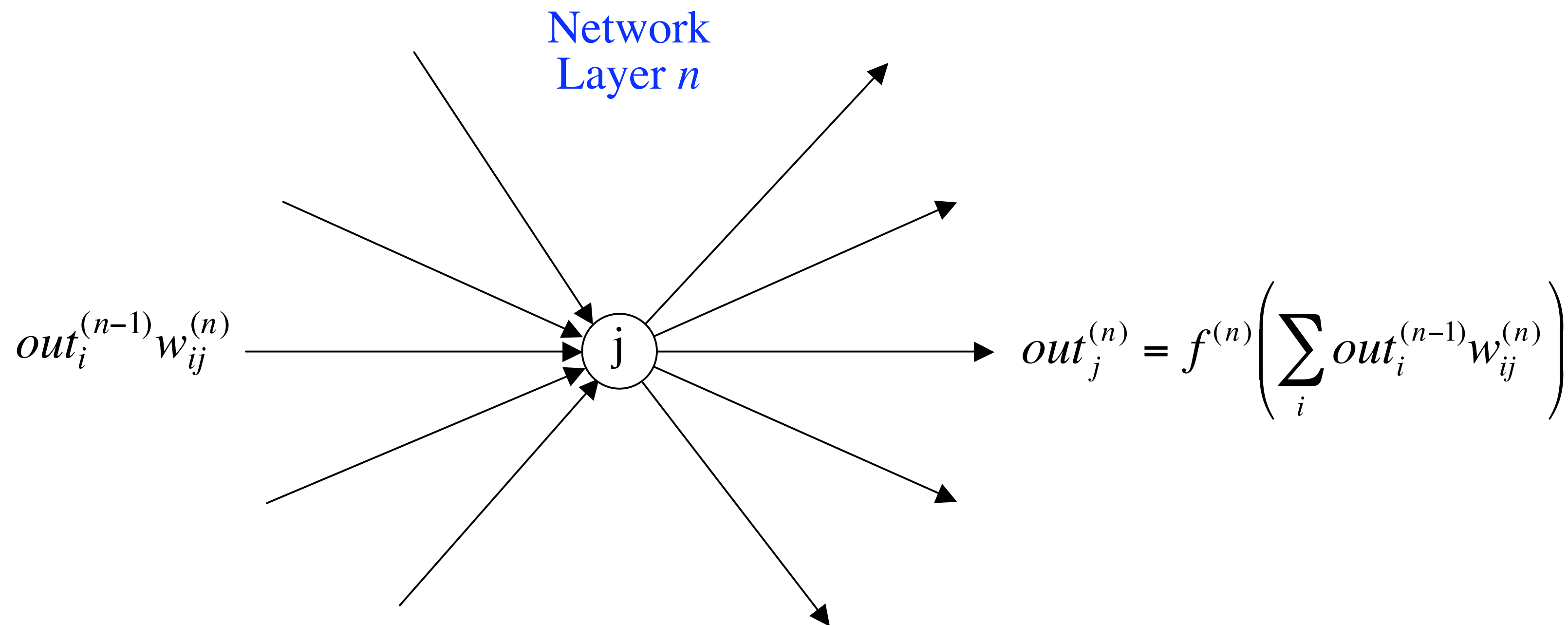
We have already shown that it is not possible to find weights which enable Single Layer Perceptrons to deal with non-linearly separable problems like XOR:



However, Multi-Layer Perceptrons (MLPs) are able to cope with non-linearly separable problems. Historically, the problem was that there were no known learning algorithms for training MLPs. Fortunately, it is now known to be quite straightforward.

Notation for Multi-Layer Networks

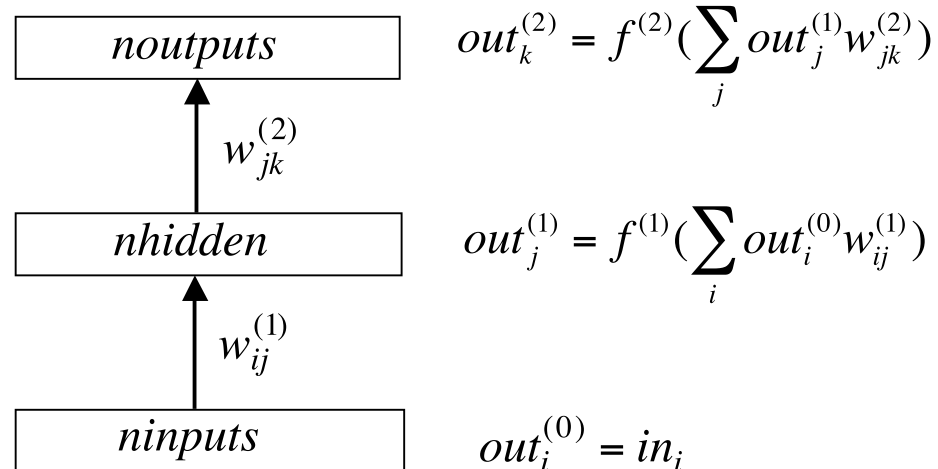
Dealing with multi-layer networks is easy if a sensible notation is adopted. We simply need another label *(n)* to tell us which layer in the network we are dealing with:



Each unit j in layer n receives activations $out_i^{(n-1)} w_{ij}^{(n)}$ from the previous layer of processing units and sends activations $out_j^{(n)}$ to the next layer of units.

Multi-Layer Perceptrons (MLPs)

Conventionally, the input layer is layer 0, and when we talk of an *N* layer network we mean there are *N* layers of weights and *N* non-input layers of processing units. Thus a two layer Multi-Layer Perceptron takes the form:



It is clear how we can add in further layers, though for most practical purposes two layers will be sufficient. Note that there is nothing stopping us from having different activation functions $f^{(n)}(x)$ for different layers, or even different units within a layer.

The Need For Non-Linearity

We have noted before that if we have a regression problem with non-binary network outputs, then it is appropriate to have a linear output activation function. So why not simply use linear activation functions on the hidden layers as well?

With activation functions $f^{(n)}(x)$ at layer n, the outputs of a two-layer MLP are

$$out_k^{(2)} = f^{(2)}\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right) = f^{(2)}\left(\sum_j f^{(1)}\left(\sum_i in_i w_{ij}^{(1)}\right).w_{jk}^{(2)}\right)$$

so if the hidden layer activations are linear, i.e. $f^{(1)}(x) = x$, this simplifies to

$$out_k^{(2)} = f^{(2)}\left(\sum_i in_i.\left(\sum_j w_{ij}^{(1)} w_{jk}^{(2)}\right)\right)$$

But this is equivalent to a single layer network with weights $w_{ik} = \sum_j w_{ij}^{(1)} w_{jk}^{(2)}$ and we know that such a network cannot deal with non-linearly separable problems.

Non-linear Activation/Transfer Functions

We have seen that the standard logistic sigmoid function is a convenient differentiable non-linear activation/transfer function that performs a smooth thresholding suitable for artificial neural networks, but are there other possibilities?

The standard logistic sigmoid function ranges from 0 to 1. There is some empirical evidence that an anti-symmetric threshold function, i.e. one that satisfies $f(-x) = -f(x)$, enables the gradient descent algorithm to learn faster. The hyperbolic tangent is simply related to the standard sigmoid and has that property:

$$f(x) = \tanh(x) = 2\,\mathrm{Sigmoid}(2x) - 1 \qquad\qquad f(-x) = -f(x)$$

and, like the standard sigmoid function, has a particularly simple derivative:

$$f'(x) = 1 - f(x)^2$$

Obviously, this will not be appropriate for outputs that are required to be probabilities, but it is a useful alternative to use for hidden unit activations.

Learning in Multi-Layer Perceptrons

Training N-layer neural networks follows the same ideas as for single layer networks. The network weights $w_{ij}^{(n)}$ are adjusted to minimize an output cost function, e.g.

$$E_{SSE} = \tfrac{1}{2} \sum_p \sum_j \left(targ_j^p - out_j^{(N)p} \right)^2$$

or

$$E_{CE} = -\sum_p \sum_j \left[targ_j^p . \log\left(out_j^{(N)p} \right) + (1 - targ_j^p) . \log\left(1 - out_j^{(N)p} \right) \right]$$

and again we can do this by a series of gradient descent weight updates

$$\Delta w_{kl}^{(m)} = -\eta \frac{\partial E(\{w_{ij}^{(n)}\})}{\partial w_{kl}^{(m)}}$$

Note that it is only the outputs $out_j^{(N)}$ of the final layer that appear in the output error function E. However, the final layer outputs will depend on all the earlier layers of weights, and the learning algorithm will adjust all of them too. The learning algorithm automatically adjusts the outputs $out_j^{(n)}$ of the earlier (hidden) layers so that they form appropriate intermediate (hidden) representations.

Training a Multi-Layer Perceptron

Training for multi-layer networks is similar to that for single layer networks:

1. Take the set of training patterns you wish the network to learn

   $$\{in_i^p, targ_j^p : i = 1 \ldots ninputs, j = 1 \ldots noutputs, p = 1 \ldots npatterns\} .$$

2. Set up the network with *ninputs* input units, *N-1* hidden layers of $nhidden^{(n)}$ non-linear hidden units, and *noutputs* output units in layer *N*. Fully connect each layer (n) to the previous layer (n-1) with connection weights $w_{ij}^{(n)}$.

3. Generate random initial 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 weight update $\Delta w_{jk}^{(n)} = -\eta \partial E(w_{jk}^{(n)}) \big/ \partial w_{jk}^{(n)}$ 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”.

To be practical, algebraic expressions need to be derived for the weight updates.

Choosing Appropriate Activation and Cost Functions

We already know from the maximum-likelihood consideration of single layer networks what output activation and cost functions should be used for particular problem types. We have also seen that non-linear hidden unit activations are needed, such as sigmoids. So we can summarize the required network properties:

Regression/ Function Approximation Problems

SSE cost function, linear output activations, sigmoid hidden activations

Classification Problems (2 classes, 1 output)

CE cost function, sigmoid output and hidden activations

Classification Problems (multiple-classes, 1 output per class)

CE cost function, softmax outputs, sigmoid hidden activations

In each case, application of the gradient descent learning algorithm (by computing the partial derivatives) leads to appropriate back-propagation weight update equations.

Computing the Partial Derivatives for Regression

We use SSE and for a two layer network the linear final outputs can be written:

$$out_k^{(2)} = \sum_j out_j^{(1)}.w_{jk}^{(2)} = \sum_j f\left(\sum_i in_i w_{ij}^{(1)}\right).w_{jk}^{(2)}$$

We can then use the chain rules for derivatives, as for the Single Layer Perceptron, to give the derivatives with respect to the two sets of weights $w_{hl}^{(1)}$ and $w_{hl}^{(2)}$:

$$\frac{\partial E_{SSE}(\{w_{ij}^{(n)}\})}{\partial w_{hl}^{(m)}} = -\sum_p \sum_k \left(targ_k - out_k^{(2)}\right).\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(m)}}$$

$$\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(2)}} = \sum_j out_j^{(1)} \frac{\partial w_{jk}^{(2)}}{\partial w_{hl}^{(2)}} = \sum_j out_j^{(1)}.\delta_{jh}.\delta_{kl} = out_h^{(1)}.\delta_{kl}$$

$$\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(1)}} = \sum_j f'\left(\sum_i in_i w_{ij}^{(1)}\right).\left(\sum_m in_m \frac{\partial w_{mj}^{(1)}}{\partial w_{hl}^{(1)}}\right).w_{jk}^{(2)} = f'\left(\sum_i in_i w_{il}^{(1)}\right).in_h.w_{lk}^{(2)}$$

Deriving the Back Propagation Algorithm for Regression

All we now have to do is substitute our derivatives into the weight update equations

$$\Delta w_{hl}^{(2)} = \eta \sum_p \sum_k \left(targ_k - out_k^{(2)} \right).out_h^{(1)}.\delta_{kl} = \eta \sum_p \left(targ_l - out_l^{(2)} \right) out_h^{(1)}$$

$$\Delta w_{hl}^{(1)} = \eta \sum_p \sum_k \left(targ_k - out_k^{(2)} \right).f'\left(\sum_i in_i w_{il}^{(1)} \right).w_{lk}^{(2)}.in_h$$

Then if the transfer function $f(x)$ is a Sigmoid we can use $f'(x) = f(x).(1 - f(x))$ to give

$$\Delta w_{hl}^{(2)} = \eta \sum_p \left(targ_l - out_l^{(2)} \right) out_h^{(1)}$$

$$\Delta w_{hl}^{(1)} = \eta \sum_p \sum_k \left(targ_k - out_k^{(2)} \right).w_{lk}^{(2)}.out_l^{(1)}.\left(1 - out_l^{(1)} \right).in_h$$

These equations constitute the Back-Propagation Learning Algorithm for Regression.

Computing the Partial Derivatives for Classification

Here we use CE and for a two layer network the sigmoidal final outputs can be written:

$$out_k^{(2)} = f\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right) = f\left(\sum_j f\left(\sum_i in_i w_{ij}^{(1)}\right).w_{jk}^{(2)}\right)$$

We can then use the chain rules for derivatives, as for the Single Layer Perceptron, to give the derivatives with respect to the two sets of weights $w_{hl}^{(1)}$ and $w_{hl}^{(2)}$:

$$\frac{\partial E_{CE}(\{w_{ij}^{(n)}\})}{\partial w_{hl}^{(m)}} = -\sum_p \sum_j \sum_k \frac{\partial}{\partial out_k^{(2)}}\left[targ_j.\log\left(out_j^{(2)}\right) + (1 - targ_j).\log\left(1 - out_j^{(2)}\right)\right].\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(m)}}$$

$$= -\sum_p \sum_k \left[\frac{targ_k}{out_k^{(2)}} - \frac{1 - targ_k}{1 - out_k^{(2)}}\right].\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(m)}}$$

$$= -\sum_p \sum_k \left[\frac{targ_k - out_k^{(2)}}{out_k^{(2)}.(1 - out_k^{(2)})}\right].\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(m)}}$$

and the derivatives of $out_k^{(2)}$ are as in the SSE case with an extra f' at the output layer.

Deriving the Back Propagation Algorithm for Classification

All we now have to do is substitute our derivatives into the weight update equations

$$\Delta w_{hl}^{(2)} = \eta \sum_p \left[\frac{targ_l - out_l^{(2)}}{out_l^{(2)}.(1 - out_l^{(2)})} \right] .f'\left(\sum_j out_j^{(1)} w_{jl}^{(2)} \right) .out_h^{(1)}$$

$$\Delta w_{hl}^{(1)} = \eta \sum_p \sum_k \left[\frac{targ_k - out_k^{(2)}}{out_k^{(2)}.(1 - out_k^{(2)})} \right] .f'\left(\sum_j out_j^{(1)} w_{jk}^{(2)} \right) .f'\left(\sum_i in_i w_{il}^{(1)} \right) .w_{lk}^{(2)}.in_h$$

Then if the transfer function $f(x)$ is a Sigmoid we can use $f'(x) = f(x).(1 - f(x))$ to give

$$\Delta w_{hl}^{(2)} = \eta \sum_p \left(targ_l - out_l^{(2)} \right) out_h^{(1)}$$

$$\Delta w_{hl}^{(1)} = \eta \sum_p \sum_k \left(targ_k - out_k^{(2)} \right) .w_{lk}^{(2)}.out_l^{(1)}.\left(1 - out_l^{(1)} \right) .in_h$$

These equations constitute the Back-Propagation Learning Algorithm for Classification.
For multiple-class CE with Softmax outputs we get exactly the same equations.

Simplifying the Computation

So we get exactly the same weight update equations for regression and classification. When implementing the Back-Propagation algorithm it is convenient to define

$$delta_l^{(2)} = \left(targ_l - out_l^{(2)}\right)$$

which is the output error. We can then write the weight update rules as

$$\Delta w_{hl}^{(2)} = \eta \sum_p delta_l^{(2)} . out_h^{(1)}$$

$$\Delta w_{hl}^{(1)} = \eta \sum_p \left[\left(\sum_k delta_k^{(2)} . w_{lk}^{(2)} \right) . out_l^{(1)} . \left(1 - out_l^{(1)}\right) \right] in_h$$

So the weight $w_{hl}^{(2)}$ between units h and l is changed in proportion to the output of unit h and the *delta* of unit l. The weight changes at the first layer now take on the same form as the final layer, but the "error" *delta* at each unit l is ***back-propagated*** from each of the output units k via the weights $w_{lk}^{(2)}$.

Networks With Any Number of Hidden Layers

It is now becoming clear that, with the right notation, it is easy to extend the gradient descent algorithm to work for any number of hidden layers. For both classification and regression, if we use appropriate matching cost and activation functions we can define

$$delta_l^{(N)} = \left(targ_l - out_l^{(N)} \right)$$

as the delta for the output layer, and then back-propagate the deltas to earlier layers using

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

Then each weight update equation can be written as:

$$\Delta w_{hl}^{(n)} = \eta \sum_p delta_l^{(n)} . out_h^{(n-1)}$$

Suddenly the Back-Propagation Algorithm looks very simple and easily programmable!

Keeping Track of the Learning

The weight update equations don't involve the output error/cost function, so there is no necessity to compute it at each stage of training, though it might be helpful to use it to keep track of how well the learning is progressing.

For regression problems, the SSE cost function itself is often the most useful measure of performance. Sometimes it is helpful to divide that by the number of patterns and take the square root to give the Root-Mean-Squared Error (RMSE). It might also help to normalize the outputs by dividing by the target mean or standard-deviation.

For classification problems, it is usually more useful to know the percentage of training patterns for which the network is predicting the correct class than what the CE is. For multiple-class cases, one can just check how often the right output unit has the highest activation, and for single output cases, one just needs to check how many outputs are on the right side of 0.5. Alternatively, one can check what percentage of outputs are within a particular tolerance (e.g., 0.1 or 0.2) of their targets.

Practical Considerations for Back-Propagation Learning

Most of the practical considerations necessary for general Back-Propagation learning were already covered when we talked about training single layer Perceptrons:

1. Do we need to pre-process the training data? If so, how?
2. How do we choose the initial weights from which the training is started?
3. How do we choose an appropriate learning rate η?
4. Should we change the weights after each training pattern, or after the whole set?
5. How can we avoid local minima in the error function?
6. How can we avoid flat spots in the error function?
7. How do we know when we should stop the training?

However, there are now two more important issues that were not covered before:

8. How many hidden layers with how many hidden units do we need?
9. Should we have different learning rates for the different layers?

How Many Hidden Layers and Hidden Units?

The best number of hidden layers and hidden units depends on many factors, including:

1. The numbers of input and output units
2. The complexity of the function or classification to be learned
3. The amount of noise in the training data
4. The number and distribution of training data patterns
5. The type of hidden unit connectivity and activation functions
6. The training algorithm used

Too few hidden units will generally leave high training and generalisation errors due to under-fitting. Too many hidden units will result in low training errors, but will make the training unnecessarily slow, and will often result in poor generalisation unless some other technique (such as *regularization*) is used to prevent over-fitting.

Virtually all "rules of thumb" you might hear about are actually nonsense. The sensible strategy is to try a range of numbers of hidden units and see which works best.

Different Learning Rates for Different Layers?

The gradient descent approach leads to the same learning rate η for each component of the network, but there is empirical evidence that it helps to have different learning rates η_X for the thresholds/biases and the real connection weights, and also for the different layers. There are a number of factors that may affect the choices:

1. The required total weighted activations feeding into each node depend on the target activations and also on their activation functions.
2. The weight changes are proportional to the activation magnitudes passing along each connection and these will vary throughout the network.
3. The later network layers (nearer the outputs) tend to have larger local errors (*deltas*) than the earlier layers (nearer the inputs).

In practice, it is often quicker to just use the same low learning rate for all the weights and thresholds, rather than spending time trying to determine appropriate differences. A very powerful approach is to use evolutionary algorithms to find the best learning rates, and that frequently does result in massive differences across network components.

Overview and Reading

1. We started by revisiting the concept of linear separability and the need for multi-layered non-linear neural networks.
2. We then saw how the Back-Propagation Learning Algorithm for multi-layered networks could be derived "easily" from the standard gradient descent approach for both regression and classification problems.
3. We ended by looking at some practical issues that didn't arise for the single layer networks.

Reading

1. Gurney: Sections 6.1, 6.2, 6.3, 6.4
2. Haykin-2009: Sections 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8
3. Beale & Jackson: Sections 4.1, 4.2, 4.3, 4.4, 4.5
4. Bishop: Sections 4.1, 4.8
5. Callan: Sections 2.2, 2.3, 2.4, 2.5