1. Hebbian Learning
2. Learning by Error Minimisation
3. Gradient Descent Learning
4. Deriving the Delta Rule
5. Delta Rule vs. Perceptron Learning Rule
6. Single Layer Classification Networks
7. Single Layer Regression Networks
Hebbian Learning

The neuropsychologist Donald Hebb postulated in 1949 how biological neurons learn:

“When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place on one or both cells such that A’s efficiency as one of the cells firing B, is increased.”

In more familiar terminology, that can be stated as the Hebbian Learning rule:

1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e. synchronously), then the strength of that synapse is selectively increased.

Then, in the notation used for Perceptrons, that can be written as the weight update:

$$\Delta w_{ij} = \eta . out_j . in_i$$

There is strong physiological evidence that this type of learning does take place in the region of the brain known as the hippocampus.
Modified Hebbian Learning

An obvious problem with the above rule is that it is unstable – chance coincidences will build up the connection strengths, and all the weights will tend to increase indefinitely. Consequently, the basic learning rule (1) is often supplemented by:

2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.

Another way to stop the weights increasing indefinitely involves normalizing them so they are constrained to lie between 0 and 1. This is preserved by the weight update

$$\Delta w_{ij} = \frac{w_{ij} + \eta \cdot \text{out}_j \cdot \text{in}_i}{\left( \sum_k (w_{kj} + \eta \cdot \text{out}_j \cdot \text{in}_k)^2 \right)^{1/2}} - w_{ij}$$

which, using a small $\eta$ and linear neuron approximation, leads to *Oja’s Learning Rule*

$$\Delta w_{ij} = \eta \cdot \text{out}_j \cdot \text{in}_i - \eta \cdot \text{out}_j \cdot w_{ij} \cdot \text{out}_j$$

which is a useful stable form of Hebbian Learning.
Hebbian versus Perceptron Learning

It is instructive to compare the Hebbian and Oja learning rules with the Perceptron learning weight update rule we derived previously, namely:

$$\Delta w_{ij} = \eta \cdot (\text{targ}_j - \text{out}_j) \cdot \text{in}_i$$

There is clearly some similarity, but the absence of the target outputs $\text{targ}_j$ means that Hebbian learning is never going to get a Perceptron to learn a set of training data.

There exist variations of Hebbian learning, such as Contrastive Hebbian Learning, that do provide powerful supervised learning for biologically plausible networks.

However, it has been shown that, for many relevant cases, much simpler non-biologically plausible algorithms end up producing the same functionality as these biologically plausible Hebbian-type learning algorithms.

For the purposes of this module, we shall therefore pursue simpler non-Hebbian approaches for formulating learning algorithms for our artificial neural networks.
Learning by Error Minimisation

The general requirement for learning is an algorithm that adjusts the network weights $w_{ij}$ to minimise the difference between the actual outputs $out_j$ and the desired outputs $targ_j$.

It is natural to define an Error Function $E$ to quantify this difference, for example:

$$E_{SSE}(w_{ij}) = \frac{1}{2} \sum_p \sum_j \left( targ_j - out_j \right)^2$$

For obvious reasons this is known as the Sum Squared Error (SSE) function. It is the total squared error summed over all output units $j$ and all training patterns $p$.

The aim of the learning algorithm is to minimise such an error measure by making appropriate adjustments to the weights $w_{ij}$. Typically we apply a series of small updates to the weights $w_{ij} \rightarrow w_{ij} + \Delta w_{ij}$ until the error $E(w_{ij})$ is “small enough”.

A systematic procedure for doing this requires the knowledge of how the error $E(w_{ij})$ varies as we change the weights $w_{ij}$, i.e. the gradient of $E$ with respect to $w_{ij}$. 
Computing Gradients and Derivatives

The branch of mathematics concerned with computing gradients is called \textit{Differential Calculus}. The relevant general idea is straightforward. Consider a function \( y = f(x) \):

\[
\Delta y / \Delta x \quad \text{for small } \Delta x.
\]

It can be written exactly as

\[
\lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}
\]

which is known as the \textit{partial derivative} of \( f(x) \) with respect to \( x \).
Examples of Computing Derivatives Analytically

Some simple examples illustrate how derivatives can be computed:

\[ f(x) = a \cdot x + b \quad \Rightarrow \quad \frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{[a \cdot (x + \Delta x) + b] - [a \cdot x + b]}{\Delta x} = a \]

\[ f(x) = a \cdot x^2 \quad \Rightarrow \quad \frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{[a \cdot (x + \Delta x)^2] - [a \cdot x^2]}{\Delta x} = 2ax \]

\[ f(x) = g(x) + h(x) \quad \Rightarrow \quad \frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{(g(x + \Delta x) + h(x + \Delta x)) - (g(x) + h(x))}{\Delta x} = \frac{\partial g(x)}{\partial x} + \frac{\partial h(x)}{\partial x} \]

Other derivatives can be found in the same way. Some particularly useful ones are:

\[ f(x) = a \cdot x^n \quad \Rightarrow \quad \frac{df(x)}{dx} = nax^{n-1} \quad f(x) = \log_e(x) \quad \Rightarrow \quad \frac{df(x)}{dx} = \frac{1}{x} \]

\[ f(x) = e^{ax} \quad \Rightarrow \quad \frac{df(x)}{dx} = ae^{ax} \quad f(x) = \sin(x) \quad \Rightarrow \quad \frac{df(x)}{dx} = \cos(x) \]
Gradient Descent Minimisation

Suppose we have a function $f(x)$ and we want to change the value of $x$ to minimise $f(x)$. What we need to do depends on the gradient of $f(x)$. There are three cases to consider:

If $\frac{df}{dx} > 0$ then $f(x)$ increases as $x$ increases so we should decrease $x$

If $\frac{df}{dx} < 0$ then $f(x)$ decreases as $x$ increases so we should increase $x$

If $\frac{df}{dx} = 0$ then $f(x)$ is at a maximum or minimum so we should not change $x$

In summary, we can decrease $f(x)$ by changing $x$ by the amount:

$$\Delta x = x_{\text{new}} - x_{\text{old}} = -\eta \frac{df}{dx}$$

where $\eta$ is a small positive constant specifying how much we change $x$ by, and the derivative $\frac{df}{dx}$ tells us which direction to go in. If we repeatedly use this equation, $f(x)$ will (assuming $\eta$ is sufficiently small) keep descending towards its minimum, and hence this procedure is known as gradient descent minimisation.
Gradients in More Than One Dimension

It might not be obvious that one needs the gradient/derivative itself in the weight update equation, rather than just the sign of the gradient. So, consider the two dimensional function shown as a contour plot with its minimum inside the smallest ellipse:

A few representative gradient vectors are shown. By definition, they will always be perpendicular to the contours, and the closer the contours, the larger the vectors. It is now clear that we need to take the relative magnitudes of the $x_1$ and $x_2$ components of the gradient vectors into account if we are to head towards the minimum efficiently.
Training a Single Layer Feed-forward Network

Now we understand how gradient descent weight update rules can lead to minimisation of a neural network’s output errors, it is straightforward to train any network:

1. Take the set of training patterns you wish the network to learn
   \( \{ \text{inp}_i^p, \text{targ}_j^p : i = 1 \ldots \text{ninputs}, j = 1 \ldots \text{noutputs}, p = 1 \ldots \text{npatterns} \} \)

2. Set up the network with \( \text{ninputs} \) input units fully connected to \( \text{noutputs} \) output units via connections with weights \( w_{ij} \)

3. Generate random initial weights, e.g. from the range \([-\text{smwt}, +\text{smwt}]\)

4. Select an appropriate error function \( E(w_{ij}) \) and learning rate \( \eta \)

5. Apply the weight update \( \Delta w_{ij} = -\eta \partial E(w_{ij})/\partial w_{ij} \) to each weight \( w_{ij} \) for each training pattern \( p \). One set of updates of all the weights for all the training patterns is called one \textit{epoch} of training.

6. Repeat step 5 until the network error function is “small enough”.

This will produce a trained neural network, but steps 4 and 5 can still be difficult…
Gradient Descent Error Minimisation

We will look at how to choose the error function $E$ next lecture. Suppose, for now, that we want to train a neural network by adjusting its weights $w_{ij}$ to minimise the SSE:

$$E(w_{ij}) = \frac{1}{2} \sum_p \sum_j \left( \text{targ}_j - \text{out}_j \right)^2$$

We have seen that we can do this by making a series of gradient descent weight updates:

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

If the transfer function for the output neurons is $f(x)$, and the activations of the previous layer of neurons are $\text{in}_i$, then the outputs are $\text{out}_j = f(\sum_i \text{in}_i w_{ij})$, and

$$\Delta w_{kl} = -\eta \frac{\partial}{\partial w_{kl}} \left[ \frac{1}{2} \sum_p \sum_j \left( \text{targ}_j - f(\sum_i \text{in}_i w_{ij}) \right)^2 \right]$$

Dealing with equations like this is easy if we use the chain rules for derivatives.
Chain Rules for Computing Derivatives

Computing complex derivatives can be done in stages. First, suppose $f(x) = g(x).h(x)$

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{g(x + \Delta x).h(x + \Delta x) - g(x).h(x)}{\Delta x} = \lim_{\Delta x \to 0} \left(\frac{g(x)}{h(x)} + \frac{\partial g(x)}{\partial x} \frac{\Delta x}{\Delta x}\right) \left(h(x) + \frac{\partial h(x)}{\partial x} \Delta x\right) - g(x).h(x)$$

We can similarly deal with nested functions. Suppose $f(x) = g(h(x))$

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{g(h(x) + \Delta x)) - g(h(x))}{\Delta x} = \lim_{\Delta x \to 0} \frac{g(h(x)) + \frac{\partial g(h(x))}{\partial h(x)} \frac{\Delta h(x)}{\Delta x} - g(h(x))}{\Delta x}$$

$$\frac{df(x)}{dx} = \frac{\partial g(h(x))}{\partial h(x)} \cdot \frac{\partial h(x)}{\partial x}$$
Using the Chain Rule on the Weight Update Equation

The algebra gets rather messy, but after repeated application of the chain rule, and some tidying up, we end up with a very simple weight update equation:

\[
\Delta w_{kl} = -\eta \frac{\partial}{\partial w_{kl}} \left[ \frac{1}{2} \sum_{p} \sum_{j} \left( \text{targ}_j - f \left( \sum_{i} \text{in}_i w_{ij} \right) \right)^2 \right]
\]

\[
\Delta w_{kl} = -\eta \left[ \frac{1}{2} \sum_{p} \sum_{j} \frac{\partial}{\partial w_{kl}} \left( \text{targ}_j - f \left( \sum_{i} \text{in}_i w_{ij} \right) \right)^2 \right]
\]

\[
\Delta w_{kl} = -\eta \left[ \frac{1}{2} \sum_{p} \sum_{j} 2 \left( \text{targ}_j - f \left( \sum_{i} \text{in}_i w_{ij} \right) \right) \left( -\frac{\partial}{\partial w_{kl}} f \left( \sum_{m} \text{in}_m w_{mj} \right) \right) \right]
\]

\[
\Delta w_{kl} = \eta \left[ \sum_{p} \sum_{j} \left( \text{targ}_j - f \left( \sum_{i} \text{in}_i w_{ij} \right) \right) \left( f' \left( \sum_{n} \text{in}_n w_{nj} \right) \frac{\partial}{\partial w_{kl}} \left( \sum_{m} \text{in}_m w_{mj} \right) \right) \right]
\]
\[ \Delta w_{kl} = \eta \left[ \sum_p \sum_j \left( \text{targ}_j - f \left( \sum_i \text{in}_i w_{ij} \right) \right) \left( f' \left( \sum_n \text{in}_n w_{nj} \right) \left( \sum_m \text{in}_m \frac{\partial w_{mj}}{\partial w_{kl}} \right) \right) \right] \]

\[ \Delta w_{kl} = \eta \left[ \sum_p \sum_j \left( \text{targ}_j - f \left( \sum_i \text{in}_i w_{ij} \right) \right) \left( f' \left( \sum_n \text{in}_n w_{nj} \right) \left( \sum_m \delta_{mk} \delta_{jl} \right) \right) \right] \]

\[ \Delta w_{kl} = \eta \left[ \sum_p \sum_j \left( \text{targ}_l - f \left( \sum_i \text{in}_i w_{il} \right) \right) \left( f' \left( \sum_n \text{in}_n w_{nl} \right) \left( \text{in}_k \right) \right) \right] \]

\[ \Delta w_{kl} = \eta \sum_p \left( \text{targ}_l - \text{out}_l \right) f' \left( \sum_n \text{in}_n w_{nl} \right) \text{in}_k \]

The **prime notation** is defined such that \( f' \) is the derivative of \( f \). We have also used the **Kronecker Delta** symbol \( \delta_{ij} \) defined such that \( \delta_{ij} = 1 \) when \( i = j \) and \( \delta_{ij} = 0 \) when \( i \neq j \).
The Delta Rule

We now have the gradient descent learning algorithm for single layer SSE networks:

$$\Delta w_{kl} = \eta \sum_p (targ_l - out_l) f'(\sum_n in_n w_{nl}) \cdot in_k$$

Notice that the updates involve the derivative $f'(x)$ of the transfer function $f(x)$. This will clearly be problematic for simple Perceptrons that use the step function $\text{step}(x)$ as their threshold function, because this has zero derivative everywhere except at $x = 0$ where it is infinite. However, for a simple linear activation function $f(x) = x$ we have

$$\Delta w_{kl} = \eta \sum_p (targ_l - out_l) \cdot in_k$$

which is often known as the **Delta Rule** because each weight update $\Delta w_{kl}$ is simply proportional to the relevant input $in_k$ and the corresponding output discrepancy

$$\text{delta}_l = targ_l - out_l$$

This update rule is exactly the same as the Perceptron Learning Rule we saw earlier.
Delta Rule vs. Perceptron Learning Rule

We have seen that the Delta Rule and the Perceptron Learning Rule for training Single Layer Perceptrons have exactly the same weight update equation.

However, the two algorithms were obtained from very different theoretical starting points. The Perceptron Learning Rule was derived from a consideration of how we should shift around the decision hyper-planes for step function outputs, while the Delta Rule emerged from a gradient descent minimisation of the Sum Squared Error for a linear output activation function.

The Perceptron Learning Rule will converge to zero error and no weight changes in a finite number of steps if the problem is linearly separable, but otherwise the weights will keep oscillating. On the other hand, the Delta Rule will (for sufficiently small $\eta$) always converge to a set of weights for which the error is a minimum, though the convergence to the precise target values will generally proceed at an ever decreasing rate proportional to the output discrepancies $\Delta l$. 
Single Layer Classification Networks

We have seen that the activation function \( f \) must be differentiable for gradient descent learning to work. A simple linear activation function with the SSE cost function leads to the simple delta rule, but the most appropriate choice of activation function and cost function actually depends on what type of problem is being studied.

For *classification* problems, it is natural to look for simple smooth (i.e. differentiable) versions of the step threshold function we used for the Simple Perceptron. The standard sigmoid (a.k.a. logistic function) is a particularly convenient replacement:

\[
f(x) = \text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}
\]

One can attempt to use this with the SSE cost function, but we shall see next lecture that there are better cost functions to use for classification problems.
Single Layer Regression Networks

In special cases there are simpler ways than gradient descent to find network weights $w_{ij}$ that minimise the difference between the actual outputs $out_j$ and desired outputs $targ_j$.

For regression (a.k.a. function approximation) problems, the output activation function should be linear, rather than the step() or Sigmoid() required for classification outputs. This means the outputs of a single layer regression network take the simple form:

$$out_j = \sum_{i=1}^{n} in_i w_{ij} - \theta_j = \sum_{i=0}^{n} in_i w_{ij}$$

and for the minimum of the Sum Squared Error on the outputs we have:

$$\frac{\partial}{\partial w_{kl}} \left[ \frac{1}{2} \sum_{p} \sum_{j} \left( targ_j - \sum_{i} in_i w_{ij} \right)^2 \right] = 0$$

This is a set of simultaneous linear equations, one for each training pattern, and finding the weights in this case reduces to solving that set of equations for the $w_{ij}$. 
Learning by Simple Matrix Inversion

If we introduce explicit training pattern labels $p$ and compute the derivative we have

$$\sum_p \left( \text{targ}_{pl} - \sum_i \text{in}_{pl}w_{il} \right) \text{in}_{pk} = 0$$

and this set of equations can be written in conventional matrix form as

$$\text{in}^T (\text{targ} - \text{in} \ w) = 0$$

which has a formal solution for the weights in terms of the input **pseudo-inverse** $\text{in}^\dagger$

$$w = \text{in}^\dagger \ \text{targ} = (\text{in}^T \ \text{in})^{-1} \ \text{in}^T \ \text{targ}$$

Thus, it is possible to compute the required network weights directly from the inputs and targets using standard matrix pseudo-inversion techniques.

Unfortunately, this simple matrix inversion approach will only work for this particularly simple case. In general, one has to apply the full iterative gradient descent procedure.
Overview and Reading

1. We began with a brief look at Hebbian Learning and Oja’s Rule.
2. We then considered how neural network weight learning could be put into the form of minimising an appropriate output error function.
3. Then we saw how to compute the gradients/derivatives that enable us to formulate efficient error minimisation algorithms, and how they could be used to derive the Delta Rule for training Simple Perceptrons.
4. Finally, we looked at how simple matrix pseudo-inversion techniques could be used to find the weights for single layer regression networks.

Reading

1. Bishop: Sections 3.1, 3.2, 3.3, 3.4, 3.5
2. Haykin-1999: Sections 2.2, 2.4, 3.3, 3.4, 3.5
3. Gurney: Sections 5.1, 5.2, 5.3
4. Beale & Jackson: Section 4.4
5. Callan: Sections 2.1, 2.2