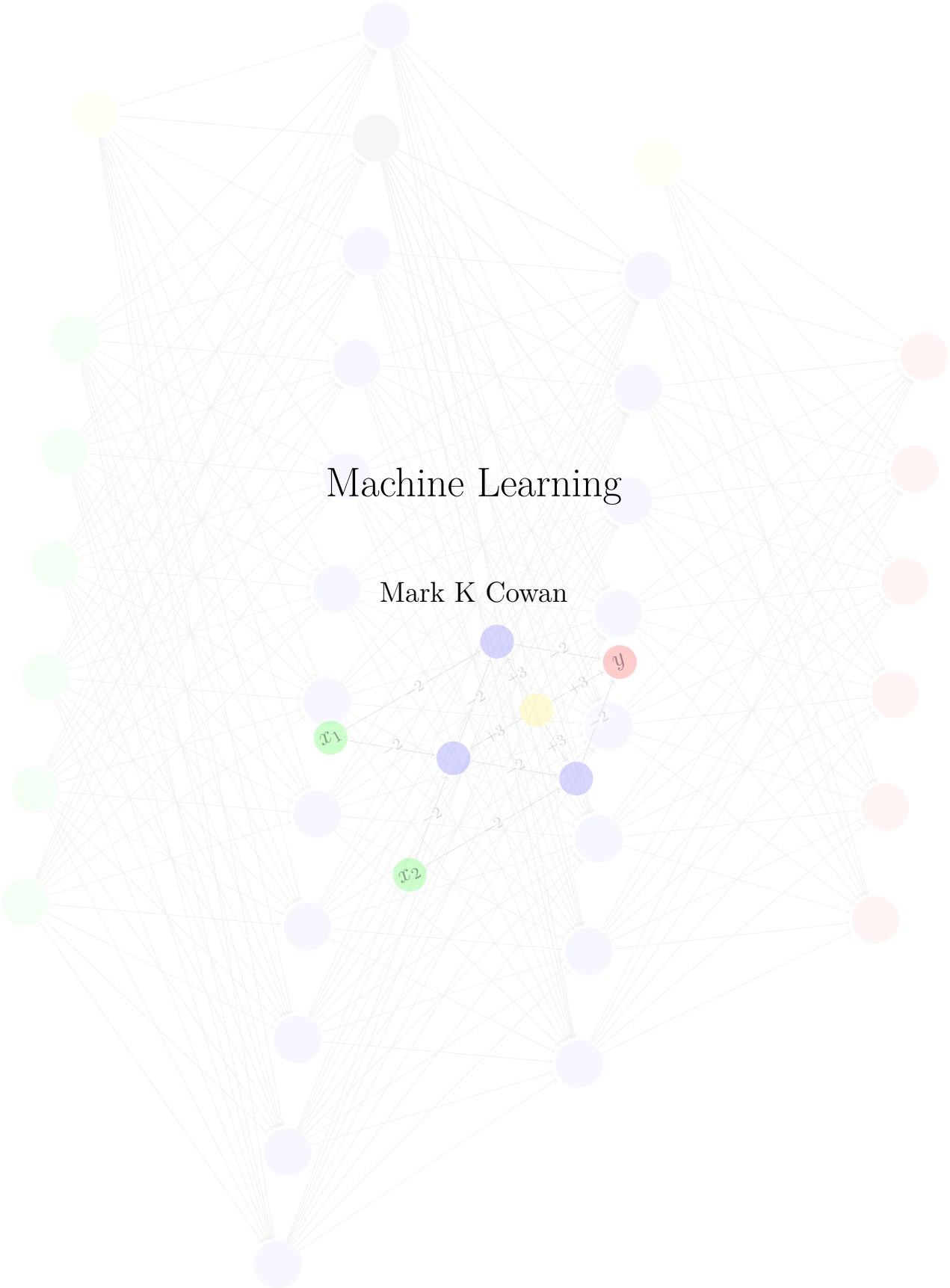
Machine Learning

Mark K Cowan



This book is based partly on content from the 2013 session of the on-line *Machine Learning* course run by Andrew Ng (Stanford University). The on-line course is provided for free via the Coursera platform (www.coursera.org). The author is no way affiliated with Coursera, Stanford University or Andrew Ng.

DRAFT

14/08/2013

Contents

Viewing

This ebook is designed to be viewed like a book, with two (facing) pages at a time and a separate cover page.

Notation

Matrices are in **bold**-face, vectors are under arr$\vec{\text{o}}$ws. Vectors which could be expanded into matrices to process several data at once are shown in **bold**-face.

Functions of matrices/vectors are applied element-wise, with a result that is of the same type as the parameter. The Hadamard matrix product is denoted by $\mathbf{A} \circ \mathbf{B}$. The norm of a matrix $\left\|\mathbf{A}\right\|$ is the norm of the vector produced by unrolling the matrix, i.e.

$$\left\|\mathbf{A}\right\|^2 = \sum_{i,j} \mathbf{A}_{ij}^2$$

Preface

TODO

1 Regression

Regression is a simple, yet powerful technique for identifying trends in datasets and predicting properties of new data points. It forms a building block of many other more complex and more powerful techniques[1].

Objective:
Create a model:
$\boldsymbol{\Theta} \in \mathbb{R}^{p\times q}$, $\mathrm{g}\colon \mathbb{R} \to \mathbb{R}$
that predicts output:
$\vec{y} \in \mathbb{R}^q$
given input:
$\vec{x} \in \mathbb{R}^p$.

Usage:
The output $\vec{y}$ can be a continuous value, a discrete value (e.g. a boolean), or a vector of such values. The input $\vec{x}$ is a vector, and can also be either discrete or continuous. The individual properties of one data row are called *features*. Typically, many inputs are to be processed at once so the feature vectors are augmented to form matrices, and the model can be rewritten for m data rows as:

$$\begin{aligned} \text{Model:} &\quad \boldsymbol{\Theta} \in \mathbb{R}^{p\times q},\ \mathrm{g}\colon \mathbb{R} \to \mathbb{R} \\ \text{Predicts:} &\quad \mathbf{Y} \in \mathbb{R}^{m\times q} \\ \text{For input:} &\quad \mathbf{X} \in \mathbb{R}^{m\times p} \end{aligned}$$

Discrete:
For discrete problems, threshold value(s) is/are chosen between the various "levels" of output. The phase-space surface defined by these levels is called the *decision boundary*. Categorisation problems are slightly more complicated, as the categories do not necessarily have a logical order (with respect to the $<$ and $>$ comparison operators). The index of the predicted category c is given by $c = \mathrm{argmax}_i(\vec{y}_i)$, and $\vec{y}$ is a vector with length equal to the number of categories.

1.1 Hypothesis

The prediction is done via a *hypothesis* function $\mathrm{h}_\Theta(\vec{x})$. This function is typically of the form:

$$\mathrm{h}_\Theta\left(\vec{x}\right) = \mathrm{g}\left(\vec{x}\boldsymbol{\Theta}\right)$$

where Θ is a matrix defining a linear transformation of the input parameters, x is a row-vector of input values, and $\mathrm{g}(k)$ is an optional non-linear transfer function. Common non-linear

[1] For example, artificial neural networks (chapter 2)

choices for the transfer function are[2] the *sigmoid* function and the *inverse hyperbolic tangent* function.

$$\mathrm{g}(k) = \mathrm{sigmoid}(k) = \frac{1}{1+e^{-k}} \qquad 0 \leq \mathrm{g}(k) \leq +1$$
$$\mathrm{g}(k) = \mathrm{artanh}(k) \quad \propto \ln\tfrac{1+k}{1-k} \qquad -1 \leq \mathrm{g}(k) \leq +1$$

Where y is continuous, a linear transfer function is used to provide *linear regression*. In cases where the output is discrete (i.e. classification problems and decision making), the sigmoid transfer function is typically used, providing *logistic regression*. Where the parameter k of g is a vector or a matrix, g is applied element-wise to the parameter, i.e. for $g = \mathrm{g}(k)$, $g_{i,j} = \mathrm{g}(k_{i,j})$.

1.2 Learning

The "learning" is driven by a set of m examples, i.e. m values of $\vec{x}$ for which the corresponding y values are known. The set of $\vec{x}$ vectors can therefore be grouped into a matrix $\mathbf{X}$, and the corresponding y values/vectors may also be grouped into a vector/matrix $\mathbf{Y}$. The hypothesis becomes $\mathrm{h}_\Theta(\mathbf{X}) = \mathrm{g}(\mathbf{X\Theta})$ (g is applied element-wise) and the learning stage may now be defined as a process which minimises the *cost-function*:

$$\mathrm{J}_\Theta(\varepsilon), \text{using } \varepsilon = \left\|\mathrm{h}_\Theta(\mathbf{X}) - \mathbf{Y}\right\|^2$$

This non-negative valued function gives some indication of the accuracy of the current hypothesis h_Θ on the learning dataset $[\mathbf{Y}, \mathbf{X}]$, and the objective of the learning stage is to minimise the value of this J_Θ.

1.2.1 Normal equation

In the unlikely event that the $\mathbf{X}$ matrix is square and non-singular, the inverse can be used to find $\mathbf{\Theta}$. For other cases, the left pseudo-inverse may be used:

$$\mathbf{Y} = \mathbf{X\Theta} \Longrightarrow \mathbf{\Theta} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{Y}$$

This minimises $\left\|\mathbf{X\Theta} - \mathbf{Y}\right\|^2$ (i.e. assumes that $\mathrm{h}_\Theta(\mathbf{X}) = \mathbf{X\Theta}$). Since the matrix to be inverted is symmetric and typically positive semi-definite, Cholesky decomposition can be used to rapidly invert it in a stable manner. Singular value decomposition (svd) is also practical.

1.2.2 Gradient descent

Successive approximations of the minimum of the cost-function may be obtained by "walking" down the landscape defined by the cost-function. This is achieved by the following iteration:

$$\mathbf{\Theta}_{i+1} = \mathbf{\Theta}_i - \alpha\vec{\nabla}\mathrm{J}_\Theta$$

[2] See figure 1.1 on page 5

where α is the *learning rate*, and defines the speed at which the learning algorithm converges. A small value of α will result in slow learning, while a large value of α will result in oscillations about a minimum (and will therefore prevent convergence). A steadily decreasing function may be used in place of a constant for α in order to provide fast convergence towards a minimum, but to also reduce the final distance from the minimum if the algorithm oscillates. The initial estimate, $\mathbf{\Theta}_0$ is usually initialised to normally distributed random values with mean $\mu = 0$. Note that the gradient descent method will not always find the global minimum of the cost function, it can get trapped in local minima.

1.3 Data normalisation

If different features are on considerably different magnitude scales, then *normalisation* may be necessary first. Otherwise, the features on larger scales will dominate the cost-function and prevent the other features from being "learnt". One simple way to transform all features to a similar scale is:

$$x_{normalised} = \frac{x_{raw} - \mu}{\sigma}$$

where the mean and the standard deviation of a list x containing N values are respectively:

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

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N} (x_i - \mu)^2}$$

TODO
Graphical examples to show why this works

TODO
Mention Tikhonov regularisation and use PhD examples to demonstrate the power of image/fourier operators. Maybe a mention of the cool stuff that can occur when other AI techniques such as evolutionary algorithms are added to the mix

1.4 Regularisation

The previous solvers will to minimise the residual $\left\|\mathrm{h}_\Theta(\mathbf{X}) - \mathbf{Y}\right\|$, but consequently may attempt to fit the noise and errors in the learning data too. In order to prevent this, other

terms can be added into the minimisation process. A simple extra term to minimise is the magnitude of the learning values $\boldsymbol{\Theta}$, by modifying the residual as follows:

$$\left\|\mathrm{h}_{\Theta}(\mathbf{X}) - \mathbf{Y}\right\|^2 + \lambda\left\|\boldsymbol{\Theta}\right\|^2$$

The non-negative variable λ is the *regularisation parameter*, and it determines whether the learning process primarily minimises the magnitude of the parameter vector ($\boldsymbol{\Theta}$), which results in *under-fitting* and may be fixed by decreasing λ, or whether the learning process *over-fits* the training data $(\mathbf{X}, \mathbf{Y})$ which may be remedied by increasing λ.

If one of the parameters ($\boldsymbol{\Theta}_1$ in $\boldsymbol{\Theta}$) represents a constant offset (e.g. $\mathbf{X}_1 \stackrel{\text{def}}{=} \vec{1}$) then this parameter (and any associated weightings) are excluded from the regularisation[3], i.e:

$$\left\|\mathrm{h}_{\Theta}(\mathbf{X}) - \mathbf{Y}\right\|^2 + \lambda\left\|\boldsymbol{\Theta}_{2..N}\right\|^2$$

TODO
Justify why/when bias features are required, and why they often escape regularisation

Regularisation of the normal equation:

$$\boldsymbol{\Theta} = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{Y}$$

* Remember to set $\mathbf{I}_1$ to zero if $\boldsymbol{\Theta}_1$ is an offset.

TODO
Prove it

Regularisation of the gradient descent method:

$$\mathrm{J}_{\Theta} = \left\|\mathrm{h}_{\Theta}\left(\mathbf{X}\right) - \mathbf{Y}\right\|^2 + \lambda\left\|\boldsymbol{\Theta}\right\|^2$$

* Remember to exclude $\boldsymbol{\Theta}_1$ from the norm calculation if $\boldsymbol{\Theta}_1$ is an offset.

TODO
Prove it

[3]Note that the bias nodes of a neural network (chapter 2) are extra features, not constant offsets

1.5 Linear regression

By taking the transfer function to be $g(k) = k$, the model simply represents a linear combination of the input values, parametrised by the $\mathbf{\Theta}$ matrix. While this results in a linear hypothesis function h_Θ (with respect to $\mathbf{X}$ and $\mathbf{Y}$), the output need not necessarily be linear with respect to the parameters of the actual underlying problem – the regression parameters (x values) may be non-linear functions of the problem parameters. For example, given a problem involving two physical parameters a and b, we can construct a set of *features* for the regression that are non-linear with respect to a and b, e.g:

$$\vec{x} = \left(a^2,\ ab,\ b^2,\ a^3/b\right)$$

1.6 Logistic regression

A non-linear transfer function (e.g. the sigmoid function: $g(k) = \frac{1}{1+e^{-k}}$) can be used to produce a non-linear hypothesis. This is particularly useful for problems with discrete output such as categorisation and decision problems, as these kind of problems imply non-linear hypotheses. The output may then be passed through a threshold function in order to produce a discrete value[4], for example:

$$\text{threshold}\,(y) = \begin{cases} \text{true} & \text{if } y \geq 0.5, \\ \text{false} & \text{if } y < 0.5 \end{cases}$$

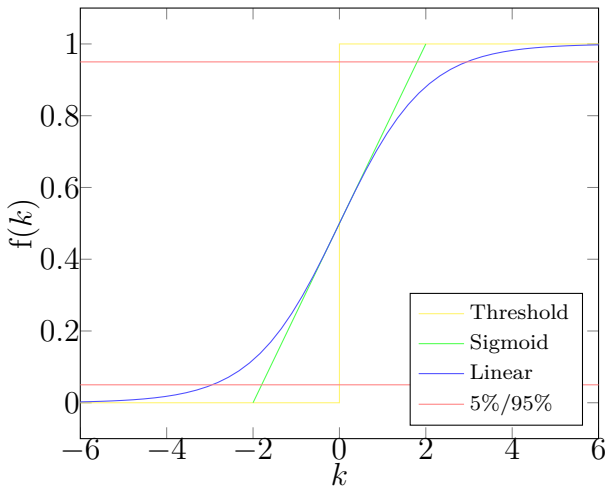


Figure 1.1: Plot of the sigmoid transfer function

The sigmoid function has a range of $[0, 1]$, which is is ideal for binary problems with yes/no outputs such as categorisation. For ternary problems with positive/neutral/negative outputs, a function with a range of $[-1, +1]$ may be more applicable. One such function is the artanh function, although the sigmoid function can also be scaled to this new range.

[4]For example, yes/no

1.7 Multi-class classification

In order to classify input vectors $\vec{x}$ into some category c, a separate classifier can be trained for each category. Each individual classifier C_i predicts whether or not the object o (represented by input vector $\vec{x}$) is in category c_i, i.e. each individual classifier predicts the result of the boolean expression $C_i(\vec{x}) \rightarrow o \in c_i$. Since each individual classifier predicts only whether the object is in the associated category or is not in it, this approach is often called *one vs all classification*. The cost-function of a multi-class classifier is the sum of the cost-functions for each individual one-vs-all classifier. Consequently, the gradient of the cost-function is the sum of all the gradients of the cost-functions for the individual classifiers.

TODO
Simplify and shorten, add examples

1.8 Formulae

Linear regression:

$$\begin{aligned}
\mathrm{g}(k) &= k \\
\mathrm{J}_\Theta &= \tfrac{1}{2m}\left\|\mathrm{h}_\Theta(\mathbf{X}) - \mathbf{Y}\right\|^2 + \tfrac{\lambda}{2m}\left\|\boldsymbol{\Theta}\right\|^2 \\
\vec{\nabla}\mathrm{J}_\Theta &= \tfrac{1}{m}\mathbf{X}^T\Big(\mathrm{h}_\Theta(\mathbf{X}) - \mathbf{Y}\Big) + \tfrac{\lambda}{m}\boldsymbol{\Theta}
\end{aligned}$$

Logistic regression:

$$\begin{aligned}
\mathrm{g}(k) &= \frac{1}{1+e^{-k}} \text{ for yes/no} \\
\text{or} &= \tanh^{-1}(k) \text{ for positive/neutral/negative} \\
\mathrm{J}_\Theta &= -\tfrac{1}{m}\Big(\vec{y}\cdot\log\big(\mathrm{h}_\Theta(\vec{x})\big) + (1-\vec{y})\cdot\log\big(1-\mathrm{h}_\Theta(\vec{x})\big)\Big) + \tfrac{\lambda}{2m}\left\|\boldsymbol{\Theta}\right\|^2 \\
\vec{\nabla}\mathrm{J}_\Theta &= \frac{1}{m}\mathbf{X}^T\Big(\mathrm{h}_\Theta(\mathbf{X}) - \mathbf{Y}\Big) + \frac{\lambda}{m}\boldsymbol{\Theta}
\end{aligned}$$

1.9 Fine-tuning the learning process

The result of iterative learning processes such as gradient descent can be assessed through a variety of techniques, which provides information that can be used to make intelligent adjustments to the learning rate and to the regularisation parameter.

Problems arising from a poor choice of the learning rate α may be identified by looking at the *learning curves*. Bad values for the regularisation parameter λ may be diagnosed by *cross-validation*, which identifies *bias* and *variance*.

1.9.1 α: Learning, convergence and oscillation

As the learning process iterates, the cost-function can be plotted against the iteration number to produce *learning curves*. The evolution of the cost-function during the learning process will indicate whether the learning rate is too high, or too slow. The cost-function should ideally converge to some value when the learning process is complete (figure 1.2c), resulting from the function reaching a local minimum. A lack of convergence indicates that the learning process is far from complete (figure 1.2a), and oscillation indicates that the learning process is unlikely to complete (figure 1.2b).

(a) No convergence
Increase α

(b) Oscillation
Decrease α

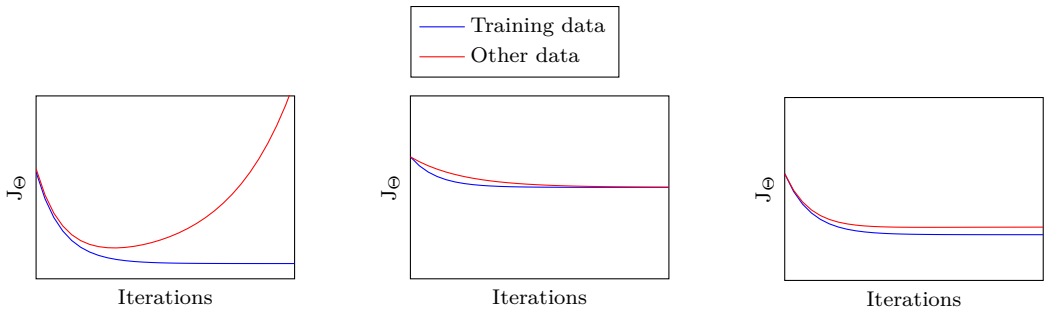
(c) Convergence
Good α

Figure 1.2: Learning curves for various learning rates

A simple optimisation is to start with a high value of α, then decrease it with each oscillation. This provides rapid convergence initially with a high chance of oscillations as the cost-function minimum is approached, but with the oscillations decreasing as the learning rate is decreased, resulting in convergence close to the cost-function minimum.

1.9.2 λ: Bias and variance

The dataset used to train the machine learning algorithm does not completely represent the problem that the algorithm is intended to solve, so it is possible for the algorithm to *over-fit* the data, by learning the imperfections in the dataset in addition to the desired trends. In this case the cost-function will be very low when evaluated on the training data, since the algorithm fits the training data well. When the algorithm is applied to other data though, the error can be large (figure 1.3a). Over-fitting can be prevented by regularisation, however this leads to another problem, *under-fitting*, where the cost-function will be high for both the training dataset and for other data as a result of the algorithm not learning the trend in enough detail (figure 1.3b). Ideally, the algorithm should perform with a low but similar error value on both the training data and on other data (figure 1.3c).



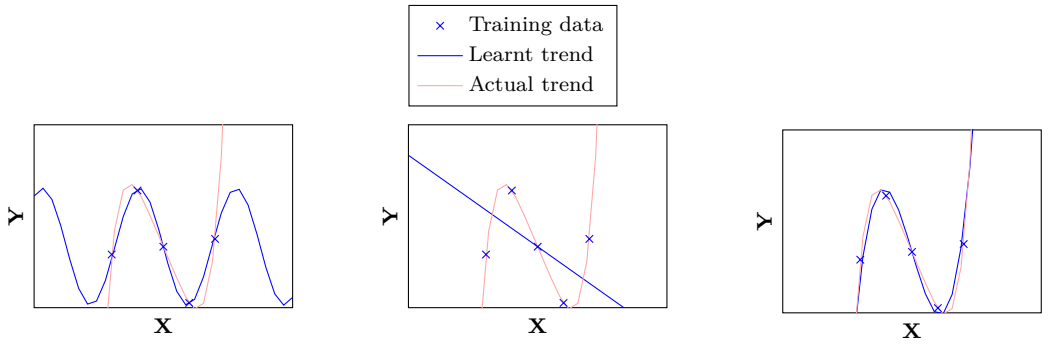
(a) Over-fit/Variance
Increase λ

(b) Under-fit/Bias
Decrease λ

(c) Good

Figure 1.3: Learning curves for various learning rates

Example:
Taking a fifth-order polynomial as an example, we train a linear regression system (using gradient descent) by using four[5] known points in the polynomial. Training with many features (for example, monomials up to the tenth order), we can find an exact or near-exact match for the training data as shown in figure 1.4a, but this learnt trend may be very different to the actual trend. Conversely, training with too few features or with high regularisation produces a learnt trend that is equally bad at matching either the training dataset or future data, shown in figure 1.4b. The good fit in figure 1.4c doesn't perfectly follow the trend, since there is insufficient information in just four points to describe a fifth-order polynomial - but it is a reasonable match. Of course, if the actual trend was a sinusoid then the "good" fit would have high error instead. When the training data contains insufficient information to describe the trend, then some properties of the trend must be known beforehand when designing the features, for example are we to use use sinusoid features, monomials features, exponential features or some combination of them all?



(a) Variance:
many features and
low regularisation

(b) Bias:
few features or
high regularisation

(c) Good

Figure 1.4: Examples of over-fitting, under-fitting and a good fit

Cross-validation:
If the actual trend were known, then there would be no need for machine learning. In

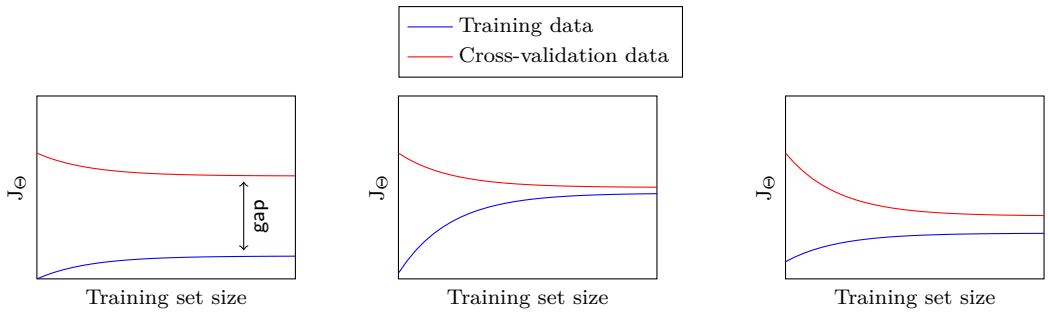
[5]This is two less than would be required to uniquely describe the polynomial

practice, the learnt trend cannot be directly compared against the actual trend, and for many real-world problems there is no "actual trend". Instead, the training dataset may be split into two subsets, the *training* set and the *cross-validation* set. The learning algorithm may then be applied to the training set, and the regularisation parameter varied in order to minimize the error[6] in the cross-validation set (see below). The amount of regularisation can therefore be determined intelligently, rather then by guesswork. In figure 1.3, the "other data" series becomes the cost-function curve for the cross-validation dataset.

$$\boldsymbol{\Theta}_{good} = \underset{\Theta}{\operatorname{argmin}}\ \mathrm{J}_{\Theta}\left(\lambda, \mathbf{X}_{train}\right)$$
$$\lambda_{good} = \underset{\lambda}{\operatorname{argmin}}\ \mathrm{J}_{\Theta}\left(\lambda, \mathbf{X}_{cross}\right)$$

TODO
Validation dataset and example

Growing the training set:
Another technique to diagnose bias/variance is to plot the cost-function for the training set and a fixed-cross-validation set while varying the size of the training set. Adding more training data will not reduce error caused bias, but will reduce error caused by variance. Therefore, the errors in the training set and in the cross-validation set will both converge to some high value if the system is suffering from high bias (Figure 1.5a). If the system is suffering from high variance instead then the errors may appear to converge to two separate values with a large gap in between (Figure 1.5b), but will converge eventually if a large and diverse enough training set is used. Over-fitting can therefore be eliminated by getting more training data when possible, instead of increasing the regularisation parameter (λ).



(a) Variance (b) Bias (c) Good

Figure 1.5: Identifying bias and variance by varying the size of the training set

Summary:
Over-fitting can be interpreted as the system trying to infer too much from too little training data, whereas under-fitting can be interpreted as the system making inefficient use of the training data, or there not being sufficient training data for the system to create a useful

[6]The value of the cost-function, J_{Θ}

hypothesis. Therefore, both can be fixed by either adjusting the amount of regularisation or by altering the amount of training data available. In order to increase the amount of training data, new data/features[7] can be collected or new features can be produced by applying non-linear operations to the existing elements, such as exponentiation ($\vec{x}_7 = \vec{x}_1^2$), multiplication or division ($\vec{x}_8 = \vec{x}_4\vec{x}_5/\vec{x}_6$).

High bias	**High variance**
Decrease λ	Increase λ
Get more features	Use less features
Create more features	Grow the training dataset

Table 1.1: Cheat-sheet for handling bias and variance

1.9.3 Accuracy, precision, F-values

TODO
This section is high priority

[7]Recall that features are properties of a data item

2 Neural networks

Neural networks are complex non-linear models, built from components that individually behave similarly to a regression model. They can be visualised as graphs[1], and some sub-graphs may exist with behaviour similar to that of logic gates[2]. Although the structure of a neural network is explicitly designed beforehand, the processing that the network does in order to produce a hypothesis (and therefore, the various logic gates and other processing structures within the network) evolves during the learning process[3]. This allows a neural network to be used as a solver that "programs itself", in contrast to typical algorithms that must be designed and coded explicitly.

Evaluating the hypothesis defined by a neural network may be achieved via *feed-forward*, which amounts to setting the input nodes, then propagating the values through the connections in the network until all output nodes have been calculated completely. The learning can be accomplished by using gradient descent, where the error in the output nodes is pushed back through the network via *back-propagation*, in order to estimate the error in the *hidden nodes*, which allows calculation of the gradient of the cost-function.

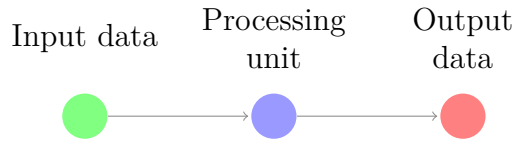


Figure 2.1: An information processing unit, visualised as a graph

2.1 Regression visualised as a building block

Linear regression may be visualised as a graph. The output is simply the weighted sum of the inputs:

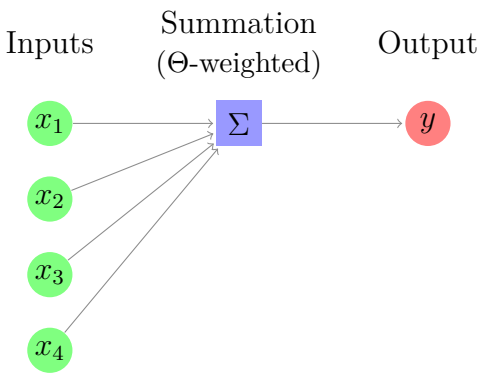


Figure 2.2: Linear regression

[1] For example, figure 2.6
[2] For example, figure 2.10
[3] Analogous to biological neural networks, from which the concept of artificial neural networks is derived

Similarly, logistic regression may be visualised as a graph, with one extra node to represent the transfer function. A logistic regression element may also be described using a linear regression element and a transfer node, by recognising that the first two stages of a logistic regression element form a linear regression element.

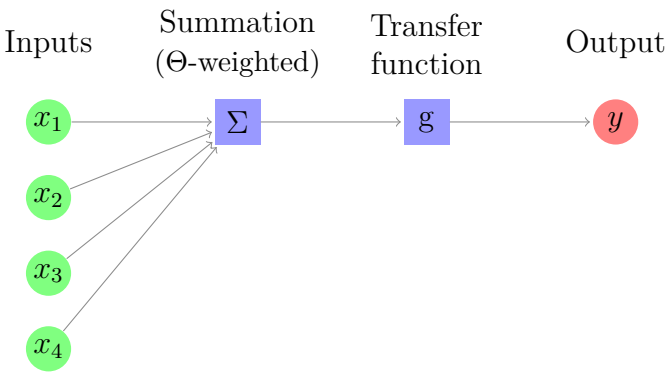


Figure 2.3: Logistic regression

Since the last three stages of the pipeline are dependent only on the first stage, we will condense them into one non-linear mixing operation at the output:

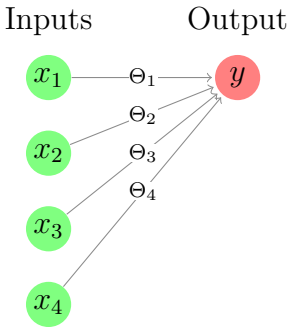


Figure 2.4: Simplified anatomy of a logistic regression process

Using this notation, a network that performs classification via several one-vs-all classifiers has the following form, where the parameter vectors $\boldsymbol{\Theta}$ have been combined to form a parameter matrix $\boldsymbol{\Theta}$, with a separate column to produce each column of the output vector:

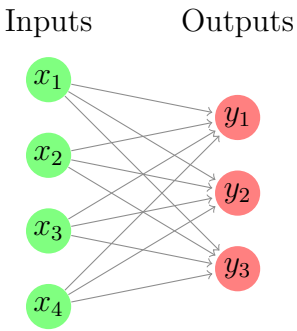


Figure 2.5: Simplified anatomy of a multi-class classification network

2.2 Hidden layers

Logistic regression is a powerful tool but it can only form simple hypotheses, since it operates on a linear combination of the input values (albeit applying a non-linear function as soon as possible). Neural networks are constructed from layers of such non-linear mixing elements, allowing development of more complex hypotheses. This is achieved by stacking[4] logistic regression networks to produce more complex behaviour. The inclusion of extra non-linear mixing stages between the input and the output nodes can increase the complexity of the network, allowing it to develop more advanced hypotheses. This is relatively simple:

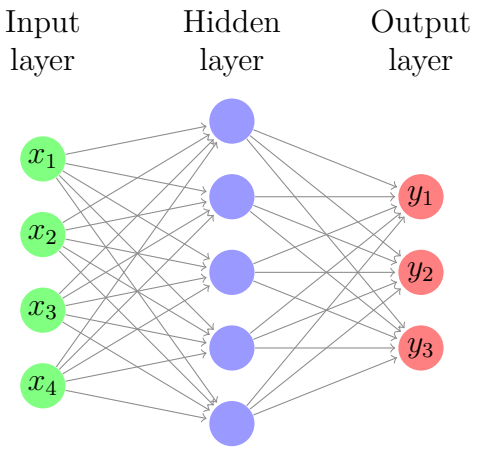


Figure 2.6: A simple neural network with one hidden layer

Although layers of linear regression nodes could be used in the network there is no point since each logistic regression element transforms a linear combination of the inputs, and a linear combination of a linear combination is itself a linear combination[5].

2.3 Notation for neural network elements

The input value to the j'th node (or *neuron*) of the l'th layer in a network with L layers is denoted z^l_j, and the output value (or *activation*) of the node is denoted $a^l_j = \mathrm{g}(z^l_j)$. The parameter matrix for the l'th layer (which produces z^l from a^{l-1}) is denoted $\mathbf{\Theta}^{l-1}$. The activation of the first (or *input*) layer is given by the network input values: $a^1 = \vec{x}$. The activation of the last (or *output*) layer is the output of the network: $a^L = \vec{y}$.

[4]Neural networks can also contain feedback loops and other features that are not possible by stacking alone

[5]i.e. Multiple linear combination layers can be combined to give one single linear combination element

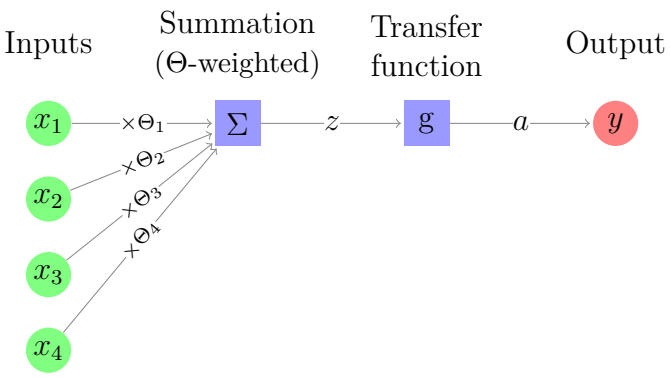


Figure 2.7: Θ, g, z and a in a neural network element

2.4 Bias nodes

Typically, each layer contains an offset term which is set to some constant value (e.g. 1). For convenience, this will be given index 0, such that $a_0^l = 1$. There is a separate parameter vector for each layer, so we now have a set of $\boldsymbol{\Theta}$ matrices. A biased network with several hidden layers is shown below to illustrate the structure and notation for such a network:

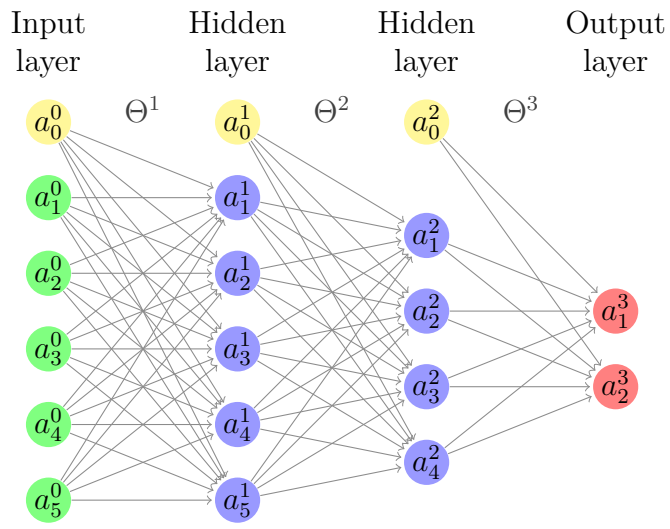
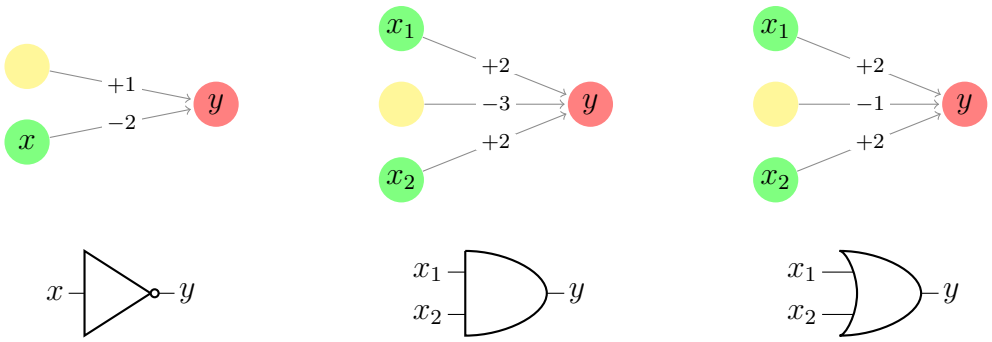


Figure 2.8: Example of the notation used to number the nodes of a neural network

2.5 Logic gates

The presence of multiple layers can be used to construct all the elementary logic gates. This in turn allows construction of advanced digital processing logic in neural networks – and this construction occurs automatically during the learning stage. Some examples are shown below, which take inputs of 0/1 and which return a positive output for *true* and a non-positive output for $false$:



(a) Logical *NOT* gate (b) Logical *AND* gate (c) Logical *OR* gate

Figure 2.9: Elementary logic gates as neural networks

From these, it becomes trivial to construct other gates. Negating the Θ values produces the inverted gates, and these can be used to construct more complex gates. Thus, neural networks may be understood as "self-designing microchips", capable of both digital and analogue processing.

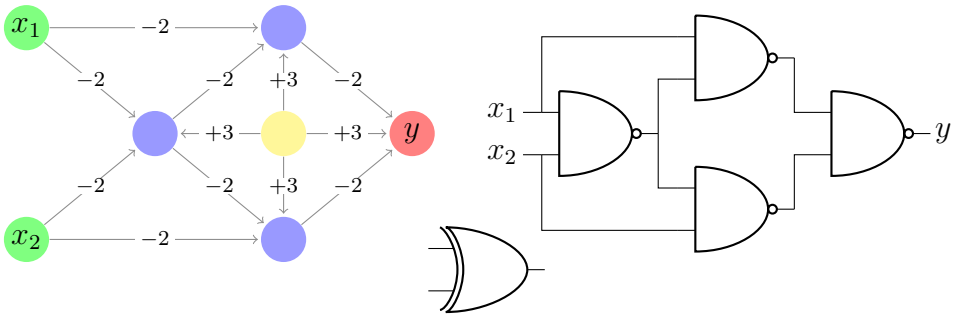


Figure 2.10: Logical *XOR* gate, constructed from four *NAND* gates

2.6 Feed-forward

To evaluate the hypothesis $\mathrm{h}_\Theta(\vec{x})$ for some input data $\vec{x}$, the data is fed forward through the layers:

$$\begin{aligned}
\vec{z}^{l+1} &= \vec{a}^{l}\boldsymbol{\Theta}^{n} && \text{— input value of neuron} \\
\vec{a}^{l+1} &= \mathrm{g}\left(\vec{z}^{l+1}\right) && \text{— activation of neuron} \\
\vec{a}^{1} &= \vec{x} && \text{— network input} \\
\vec{a}^{L} &= \mathrm{h}_\Theta\left(\vec{x}\right) && \text{— network output}
\end{aligned}$$

2.7 Cost-function

The cost-function remains the same as that for logistic regression:

$$\mathrm{J}_\Theta = -\tfrac{1}{m}\Big(\vec{y}\cdot\log\left(\mathrm{h}_\Theta\left(\vec{x}\right)\right) + \left(1-\vec{y}\right)\cdot\log\left(1-\mathrm{h}_\Theta\left(\vec{x}\right)\right)\Big) + \tfrac{\lambda}{2m}\sum_{l=1}^{L}\left\|\boldsymbol{\Theta}^{l}\right\|^{2}$$

Note the dot-product in the expression for the cost-function; this represents a summation of the cost-functions for each individual one-vs-all classifier. Since neural networks have hidden layers between the input and output nodes, the learning process is slightly more complex than that of the logistic regression multi-class network.

2.8 Gradient via back-propagation

The gradients of the cost-function (with respect to the various parameter matrices) are calculated by propagating the error in the output back through the network.

$$\begin{aligned} e^L &= \mathrm{h}_\Theta(\mathbf{X}) - \mathbf{Y} && \text{— error at the final layer} \\ e^l &= \left(e^{l+1}\,\boldsymbol{\Theta}^{l^T}\right) \circ \mathrm{g}'(z^l) && \text{— error at each prior layer} \\ \mathrm{g}(k) &= \tfrac{1}{1+e^{-k}} \rightarrow \mathrm{g}'(k) = \tfrac{\mathrm{g}(k)}{1-\mathrm{g}(k)} && \text{— transfer function and derivative} \end{aligned}$$

With the error at each node calculated, the amount that each node contributes to the over-all cost can be estimated, leading to the gradient:

$$\begin{aligned} \Delta^n &= a^{n^T} e^{n+1} && \text{— error contributed to the next layer} \\ \vec{\nabla}_{\Theta^n} \mathrm{J}_\Theta &= \tfrac{1}{m}\Delta^n + \tfrac{\lambda}{m}\boldsymbol{\Theta}^n && \text{— gradient } \tfrac{\partial \mathrm{J}_\Theta}{\partial \Theta^n} \end{aligned}$$

Note that the gradient shown here includes the regularisation term. Also, although the bias nodes have constant values for their activations, the error contributed by bias nodes is not necessarily zero[6], since the weights of these nodes ($\Theta^l_{0,j}$) are not constant. Therefore the weights of the bias nodes should be excluded from regularisation, as described in section 1.4 except for rare cases where it may be useful to minimise the amount of bias.

TODO
Real-world examples

[6] Therefore the gradient of the cost-function with respect to the weights of these nodes may also be non-zero

List of Figures