Machine Learning Notes

Sebastian Gaume

July 6, 2020

Contents

Chapter 1

Algorithms

This chapter will contain the overviews of various important algorithms, along with pseudocode and/or Octave implementations of them. It is intended to provide a quick reference to how we can implement the various algorithms and a reminder of what algorithmic options are available.

1.1 Gradient Descent

The most basic optimisation function, and the only one which is worth implementing yourself, is gradient descent. It works by evaluating the gradient of the cost function and then updating θ such that the cost function decreases with each iteration. It has one hyperparameter, α which is the 'learning rate' of the algorithm. This controls how much we change θ each iteration and it's important to tune it correctly so as to converge to the optimum quickly while also avoiding changing θ by too much and overshooting, which could cause the value of the cost function to explode to infinity.

Algorithm 1: The gradient descent algorithm

Input: The number of iterations to run, $nIters$, and the initial values for the parameter vector, $thetaInit$.
Output: The optimised parameter vector.
$n \leftarrow 0$;
$\theta \leftarrow thetaInit$;
while $n < nIters$ **do**
  $\theta \leftarrow \theta - \alpha \nabla J(\theta)$; // $J(\theta)$ is the cost function we are optimising against
  $n \leftarrow n + 1$;
end
return θ

1.2 Advanced Optimisation Algorithms

There are better optimisation algorithms than gradient descent, however these are mostly so complicated and optimised that a good implementation is difficult to write from scratch, and best left to specialists in numerical computing. As such, they should be imported in libraries or are available as built in functions in scientific programming languages such as Octave. For the most part, you ought to google the implementations of whichever libraries are the best for whatever language you're working in, however I will provide an overview for using one such function in Octave, as it is easy to use and demonstrates the general parameters you will need to pass to one of these advanced optimisation algorithms.

The Octave built-in optimisation function is 'fminunc', and it requires you to define a function which will return the cost J and the gradient grad for a given feature matrix X, target vector y and current parameter vector theta. It may also take other parameters, such as lambda for regularized cost functions.

```
function theta = optimiseTheta (X, y, lambda, maxIters)
    % Initialise theta with the correct dimensions as all zeros
    intialTheta = zeros(size(X, 2), 1);

    % Create a cost function object to pass to fminunc, costFunction most return
    % the cost, J, and the gradient, grad, in that order.
    % The @(t) allows fminunc to pass the current theta to costFunction each
    % iteration
```

```
    costFunctionObject = @(t) costFunction(X, y, t, lambda);

    % Set the options for fminunc
    options = optimset('MaxIter', maxIters, 'GradObj', 'on');

    % Optimise theta and then return it
    theta = fminunc(costFunctionObject, initialTheta, options);
end
```

This function will return the optimised theta for any given combination of X, y and lambda, assuming the feature matrix has been correctly constructed. These algorithms can always be substituted in for gradient descent and should be more efficient, especially as the number of features and training examples increase.

1.3 Forwards Propogation

Forwards propogation is the process by which a neural network calculates the hypothesis for a given feature vector $\boldsymbol{x}$. It's very simple to implement although it does require an implementation of the element-wise sigmoid function $g(\boldsymbol{z})$.

Algorithm 2: The forwards propogation algorithm

Input: The feature vector $\boldsymbol{x}$, and the set of weight matrices $\boldsymbol{\Theta^{(i)}}$ for $i \in \{1, 2, ..., L-1\}$ where L is the number of layers in the network.

Output: The hypothesis vector $\boldsymbol{h_\Theta}(\boldsymbol{x})$.

$\boldsymbol{a^{(1)}} \leftarrow \boldsymbol{x}$;

$i \leftarrow 2$;

while $i \leq L$ **do**

  $\boldsymbol{z^{(i)}} \leftarrow \boldsymbol{\Theta^{(i-1)}} \boldsymbol{a^{(i-1)}}$;

  $\boldsymbol{a^{(i)}} \leftarrow g(\boldsymbol{z^{(i)}})$; // $g(\boldsymbol{z})$ is the element-wise sigmoid function

  $i \leftarrow i + 1$;

end

return $\boldsymbol{a^{(L)}}$

1.4 Backwards Propogation

Backwards propogation is the process by which we calculate the partial derivatives of the cost function. It's difficult to understand, but not too difficult to implement once the derivative of the sigmoid function, $g'(\boldsymbol{z})$, has been implemented.

Algorithm 3: The backwards propogation algorithm

Input: The feature vectors $\boldsymbol{x^{(i)}}$, the target vectors $\boldsymbol{y^{(i)}}$ for $i \in \{1, 2, ..., m\}$, the regularization parameter λ and the $\boldsymbol{\Theta}^{(i)}$ weight matrices for $i \in \{1, 2, ..., L-1\}$.
Output: The parital derivative matrices $\boldsymbol{D^{(i)}}$ for $i \in \{1, 2, ..., L-1\}$.

$i \leftarrow 1$;
while $i < L$ **do**
  $\boldsymbol{\Delta^{(i)}} = \text{zeros}(s_{i+1},\, s_i + 1)$;
  $i \leftarrow i + 1$;
end
$i \leftarrow 1$;
while $i \leq m$ **do**
  $\boldsymbol{a}^{(1)} \leftarrow \boldsymbol{x}^{(i)}$;
  Carry out forwards propogation to compute $\boldsymbol{a}^{(l)}$ for $l \in \{2, 3, ..., L\}$; `// We also need to make sure to calculate the` $\boldsymbol{z}^{(l)}$ `values`
  $\boldsymbol{\delta}^{(L)} = \boldsymbol{a}^{(L)} - \boldsymbol{y}^{(i)}$;
  $l \leftarrow L - 1$;
  while $l > 0$ **do**
    $\boldsymbol{\delta}^{(l)} \leftarrow \left(\boldsymbol{\Theta}^{(l)}\right)^{\intercal} \boldsymbol{\delta}^{(l+1)} \mathbin{.*} g'(\boldsymbol{z}^{(l)})$; `//` $g'(\boldsymbol{z})$ `is the derivative of the element-wise sigmoid function`
    $\boldsymbol{\Delta}^{(l)} \leftarrow \boldsymbol{\Delta}^{(l)} + \boldsymbol{\delta}^{(l+1)} \left(\boldsymbol{a}^{(l)}\right)^{\intercal}$;
    $l \leftarrow l - 1$;
  end
  $i \leftarrow i + 1$
end
$l \leftarrow 1$;
while $l < L$ **do**
  $i \leftarrow 0$;
  while $i \leq s_{l+1}$ **do**
    $j \leftarrow 0$;
    while $j \leq s_l + 1$ **do**
      if $j = 0$ **then**
        $\boldsymbol{D}_{ij}^{(l)} = \frac{1}{m} \boldsymbol{\Delta}_{ij}^{(l)}$;
      end
      else
        $\boldsymbol{D}_{ij}^{(l)} = \frac{1}{m} \boldsymbol{\Delta}_{ij}^{(l)} + \lambda \boldsymbol{\Theta}_{ij}^{(l)}$;
      end
      $j \leftarrow j + 1$;
    end
    $i \leftarrow i + 1$;
  end
  $l \leftarrow l + 1$;
end
Unwrap the $\boldsymbol{D}$ matrices into dVec ;
return *dVec*

Chapter 2

Linear Regression

Linear regression models are used in regression problems of an arbitrary number of features and with features which can be functions of other features. In these problems, $\boldsymbol{y}^{(i)} \in \mathbb{R}$ and can represent any quantity. Our goal is to predict, based on features $\boldsymbol{x}^{(i)}$, what output value a datum corresponds to. For this model, the output of the hypothesis function is the predicted value of $\boldsymbol{y}^{(i)}$ given $\boldsymbol{x}^{(i)}$, parameterised by $\boldsymbol{\theta}$.

2.1 Hypothesis and Cost Function

In a linear regression model, our hypothesis is of the form

$$h_\theta(\boldsymbol{x}) = \boldsymbol{\theta}^\intercal \boldsymbol{x}, \quad h_\theta(\boldsymbol{x}) \in \mathbb{R} \tag{2.1}$$

or alternatively,

$$\boldsymbol{h_\theta}(\boldsymbol{X}) = \boldsymbol{X\theta}, \quad \boldsymbol{h_\theta}(\boldsymbol{X}) \in \mathbb{R}^m \tag{2.2}$$

and we use a mean square difference cost function of the form

$$J(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} \left[(h_\theta(\boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)})^2 \right] = \frac{1}{2m} (\boldsymbol{X\theta} - \boldsymbol{y})^\intercal (\boldsymbol{X\theta} - \boldsymbol{y}). \tag{2.3}$$

2.2 The Normal Equation

One way to minimise the cost function in linear regression is to do so analytically using the normal equation. This says that

$$\boldsymbol{\theta} = (\boldsymbol{X}^\intercal \boldsymbol{X})^{-1} \boldsymbol{X}^\intercal \boldsymbol{y} \tag{2.4}$$

gives us the $\boldsymbol{\theta}$ that minimises the cost function $J(\boldsymbol{\theta})$. This technique is useful up until $n >= 10^4$, as for these relatively low numbers of features it is more efficient than gradient descent and doesn't require us to choose a learning rate α.

2.3 Gradient Descent

We can also minimise the cost function using gradient descent. To do this, we need the gradient of $J(\boldsymbol{\theta})$ which is

$$\boldsymbol{\nabla}_j J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left[(h_\theta(\boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)}) \cdot \boldsymbol{x}_j^{(i)} \right] \tag{2.5}$$

$$\boldsymbol{\nabla} J(\boldsymbol{\theta}) = \frac{1}{m} \boldsymbol{X}^\intercal (\boldsymbol{X\theta} - \boldsymbol{y}) \tag{2.6}$$

and so we can just use this in our gradient descent algorithm and it will optimise $\boldsymbol{\theta}$. Note that all linear regression hypotheses are convex and so gradient descent will always converge with a correct learning rate and enough iterations to the global optimum.

2.4 Regularised Linear Regression

To avoid overfitting our training data by adding higher order terms to our hypothesis, we can use regularisation, which penalises larger weights for each term causing the algorithm to prefer lower valued weights and thus reducing

the problem of overfitting. To do this, we need to change our cost function to

$$J(\boldsymbol{\theta}) = \frac{1}{2m}(\boldsymbol{X\theta} - \boldsymbol{y})^\intercal(\boldsymbol{X\theta} - \boldsymbol{y}) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2. \tag{2.7}$$

The extra term penalises higher values of θ_j for all of $\boldsymbol{\theta}$ except θ_0 by convention. It also introduces a new hyper-parameter λ which controls how much the algorithm penalises the higher values. In general, we want $\lambda >= 1$ but not too large as if it's too large it will cause the model to underfit the training data.

2.4.1 The Normal Equation for Regularised Linear Regression

To minimise our new cost function for regularised linear regression, we have the same two options as before, but we need to modify them slightly. For the normal equation, this means we instead find that

$$\boldsymbol{\theta} = \left(\boldsymbol{X}^\intercal\boldsymbol{X} + \lambda \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}\right)^{-1} \boldsymbol{X}^\intercal\boldsymbol{y} \tag{2.8}$$

finds the $\boldsymbol{\theta}$ that minimises the cost function $J(\boldsymbol{\theta})$. Note that the matrix that is being multiplied by λ is the $\mathbb{R}^{n+1}$ identity matrix except the $I_{1,1}$ entry is 0 instead of 1.

2.4.2 Gradient Descent for Regularised Linear Regression

We also need to change the $\boldsymbol{\nabla}J(\boldsymbol{\theta})$ we use in our gradient descent algorithm if we choose to use regularised linear regression. The new gradient is given by

$$\boldsymbol{\nabla}_0 J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left[(h_\theta(\boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)})\right] \tag{2.9}$$

$$\boldsymbol{\nabla}_j J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left[(h_\theta(\boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)}) \cdot \boldsymbol{x}_j^{(i)}\right] + \frac{\lambda}{m}\boldsymbol{\theta}_j \qquad \text{for } j \neq 0 \tag{2.10}$$

$$\boldsymbol{\nabla}J(\boldsymbol{\theta}) = \frac{1}{m}\boldsymbol{X}^\intercal(\boldsymbol{X\theta} - \boldsymbol{y}) + \frac{\lambda}{m}\begin{bmatrix} 0 \\ \boldsymbol{\theta}_1 \\ \boldsymbol{\theta}_2 \\ \vdots \\ \boldsymbol{\theta}_n \end{bmatrix} \tag{2.11}$$

and we simply use these in our gradient descent algorithm instead of the previous versions.

Chapter 3

Logistic Regression

Logisitic regression models are used in binary classification problems, altough they can be extended to multi-class classification problems. In these problems, $\boldsymbol{y}^{(i)} \in \{0, 1\}$, where 0 represents the negative case and 1 represents the positive case, though these can then represent any given classes. Our goal is to predict, based on features $\boldsymbol{x}^{(i)}$, which class a datum is part of. For this model, the output of the hypothesis function is $P(\boldsymbol{y}^{(i)} = 1 | \boldsymbol{x}^{(i)}, \boldsymbol{\theta})$.

3.1 Hypothesis and Cost Function

In a logisitic regression model, our hypotheis is of the form

$$h_\theta(\boldsymbol{x}) = g(\boldsymbol{\theta}^\intercal \boldsymbol{x}), \quad h_\theta(\boldsymbol{x}) \in \mathbb{R} \tag{3.1}$$

$$g(\boldsymbol{z}) = \frac{1}{1 + e^{-\boldsymbol{z}}} \tag{3.2}$$

or equivalently

$$\boldsymbol{h_\theta}(\boldsymbol{X}) = g(\boldsymbol{X\theta}), \quad \boldsymbol{h_\theta}(\boldsymbol{X}) \in \mathbb{R}^m \tag{3.3}$$

as this ensures our hypothesis is between 0 and 1 for all possible inputs. We then use a cost logarithmic cost function which rewards $h_\theta(\boldsymbol{x}^{(i)})$ for being closer to $\boldsymbol{y}^{(i)}$ than the other possible value of $\boldsymbol{y}^{(i)}$ as it can only hold the values of 0 and 1. This cost function is

$$J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^{m} \left[\boldsymbol{y}^{(i)} \log \left(h_\theta(\boldsymbol{x}^{(i)}) \right) + \left(1 - \boldsymbol{y}^{(i)} \right) \log \left(1 - h_\theta(\boldsymbol{x}^{(i)}) \right) \right] \tag{3.4}$$

$$= \frac{1}{m} \left(-\boldsymbol{y}^\intercal \log \left(g(\boldsymbol{X\theta}) \right) - (\boldsymbol{1} - \boldsymbol{y})^\intercal \log \left(\boldsymbol{1} - g(\boldsymbol{X\theta}) \right) \right) \tag{3.5}$$

and it is, like that of linear regression, convex. This means we can be sure gradient descent will always converge to the global optimum when used to minimise the function.

3.2 Decision Boundaries

Because the output of the hypothesis function gives us $P(\boldsymbol{y}^{(i)} = 1 | \boldsymbol{x}^{(i)}, \boldsymbol{\theta})$, we should predict that $\boldsymbol{y}^{(i)} = 1 \iff h_\theta(\boldsymbol{x}^{(i)}) >= 0.5$. This means that if we want to visualise what our model will predict the class of a given datum is is to plot the decision boundary of the hypothesis function given the optimised $\boldsymbol{\theta}$. This decision boundary is given by

$$h_\theta(\boldsymbol{x}) = 0.5 \tag{3.6}$$

or equivalently

$$\boldsymbol{\theta}^\intercal \boldsymbol{x} = 0 \tag{3.7}$$

and it is the boundary that separates the regions where the model will predict $\boldsymbol{y}$ is 1 and where it will predict $\boldsymbol{y}$ is 0. For a small enough number of features this can be plotted and interpreted by a human to help give insight into what the model is doing.

3.3 Multi-Class Problems

We can also use logistic regression in problems where there are more than 2 classes. To do this, we use a 'one-vs-all' approach, whereby if we have n different classes, we train n different classifiers, one for each class. We then have n different hypotheses, $h_\theta^{(i)}(\boldsymbol{x})$ which correspond to the classes $i \in \{1, 2, 3, \cdots, n\}$. We train them such that $h_\theta^{(i)}(\boldsymbol{x})$

is a binary classifier using a logistic regression modifier with the positive case being that $\boldsymbol{x}$ is a member of class i and the negative case being that $\boldsymbol{x}$ is a member of any other class. As such, to find the class $\boldsymbol{x}$ is most likely to be a part of, we just compare the values of all the different $h_\theta^{(i)}(\boldsymbol{x})$, and see which is highest. If $h_\theta^{(j)}(\boldsymbol{x})$ outputs the highest value, we predict that $\boldsymbol{x}$ is a member of class j.

3.4 Gradient Descent

To fit our logistic regression model, we need to use an optimisation algorithm such as gradient descent. It is worth noting that their are other algorithms which do the same thing faster, however their implementations are best left to those who specialise in numerical methods, however they only require the cost function and its gradient to be written by the user, so we only need to know the same things as we do for gradient descent. For logistic regression, the gradient of the cost function turns out to be the same as that of linear regression's cost function, except the hypothesis function is switched out, ie

$$\boldsymbol{\nabla}_j J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left[(h_\theta(\boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)})\cdot \boldsymbol{x}_j^{(i)}\right] \tag{3.8}$$

$$\boldsymbol{\nabla} J(\boldsymbol{\theta}) = \frac{1}{m}\boldsymbol{X}^\intercal(g(\boldsymbol{X}\boldsymbol{\theta}) - \boldsymbol{y}). \tag{3.9}$$

3.5 Regularised Logistic Regression

Like with linear regression, we can include higher order features in our model to get a better fit model, however this can cause our model to overfit the training data. To solve this, we can once again use regularisation to penalise higher values for the weights of our parameters, to discourage over-fitting. For logisitic regression, this makes the cost function

$$J(\boldsymbol{\theta}) = \frac{1}{m}\left(-\boldsymbol{y}^\intercal \log\left(g(\boldsymbol{X}\boldsymbol{\theta})\right) - (\boldsymbol{1}-\boldsymbol{y})^\intercal \log\left(\boldsymbol{1} - g(\boldsymbol{X}\boldsymbol{\theta})\right)\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2. \tag{3.10}$$

3.5.1 Gradient Descent for Regularised Logistic Regression

To use gradient descent, or other similar optimisation algorithms, for regularised logistic regression, we need to find the gradient of our new cost function, which turns our to be

$$\boldsymbol{\nabla}_0 J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left[(h_\theta(\boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)})\right] \tag{3.11}$$

$$\boldsymbol{\nabla}_j J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left[(h_\theta(\boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)})\cdot \boldsymbol{x}_j^{(i)}\right] + \frac{\lambda}{m}\boldsymbol{\theta}_j \qquad \text{for } j \neq 0 \tag{3.12}$$

$$\boldsymbol{\nabla} J(\boldsymbol{\theta}) = \frac{1}{m}\boldsymbol{X}^\intercal(g(\boldsymbol{X}\boldsymbol{\theta}) - \boldsymbol{y}) + \frac{\lambda}{m}\begin{bmatrix}0\\ \boldsymbol{\theta}_1\\ \boldsymbol{\theta}_2\\ \vdots\\ \boldsymbol{\theta}_n\end{bmatrix}. \tag{3.13}$$

Chapter 4

Neural Networks

A neural network is a kind of machine learning model which is designed to imitate the connections between neurons in the human brain. It can be used in complex classification problems, as it takes in a set of features and then trains itself to recognise more features from these features, giving it access to complicated features we probably wouldn't have thought to design.

4.1 Neural Network Architecture

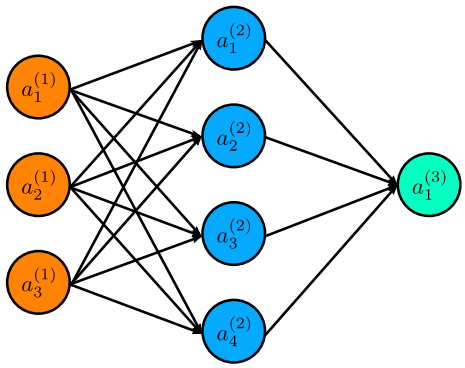

Figure 4.1: A basic neural network

A neural network is comprised of layers of activation units, which are connected to each other by a set of biases, which are the learned parameters of a neural network model. There are three main varieties of layers, the input layer, the hidden layers and the output layer. The input layer is effectively just another representation of the feature vector $\boldsymbol{x}$ and the ouput layer is just another representation of our hypothesis, $h_{\boldsymbol{\Theta}}(\boldsymbol{x})$. This leaves the hidden layers, which are effectively intermediary features which the model learns from the input features. A typical network might have one hidden layer, but you can add more to allow the network to learn more complex features.

The input layer is given the layer number 1, and the output layer is dependant on the number of hidden layers, but can be referred to as L. Furthermore, we can refer to the n^{th} activation unit in layer l with the notation $a_n^{(l)}$. Note that these then form the vector $\boldsymbol{a^{(l)}}$ which refers to all of the activation units in layer l. Each layer also has a bias unit, $a_0^{(l)}$ which is always equal to 1. This means that $\boldsymbol{a^{(1)}} = \boldsymbol{x}$, so the a notation is somewhat redundant for the input layer, though it does unify the notation. The notation s_l refers to the number of activation units in a given layer l, excluding the bias unit. Finally, we can refer to the matrix of weights that maps layer i to layer $i+1$ as $\boldsymbol{\Theta^{(i)}}$ where $\boldsymbol{\Theta^{(i)}} \in \mathbb{R}^{(s_{i+1})\times(s_i+1)}$. With this notation established, we now have all we need to allow us to make use of neural networks.

4.2 Forwards Propogation

The first thing we need to understand to use neural networks for machine learning is how a pre-trained neural network calculates values for the hypothesis given a feature vector $\boldsymbol{x}$. For each layer in a trained neural network, except the input layer, we will have a weights matrix $\boldsymbol{\Theta}$ which allows us to move from one layer to the next. As

such, we can't jump from the input layer to the output layer in one step, and we instead need to propogate values forwards between layers until we eventually reach the output layer. The general formula for moving from one layer to the next is

$$\boldsymbol{a^{(n)}} = g(\boldsymbol{\Theta^{(n-1)}a^{(n-1)}}) \tag{4.1}$$

which we can simplify to

$$\boldsymbol{a^{(n)}} = g(\boldsymbol{z^{(2)}}) \tag{4.2}$$

by defining

$$\boldsymbol{z^{(n)}} = \boldsymbol{\Theta^{(n-1)}a^{(n-1)}}. \tag{4.3}$$

Note that as before, $g(\boldsymbol{z})$ is the element-wise sigmoid function. A full algorithmic description of the process of forwards propogation is in Chapter 1, on algorithms, however this describes the maths behind the process, and you simply feed the values forwards until you reach the output layer, at which point the vector of activation units is the same as the vector of hypotheses for different classes - more notes on multi-class classification in a later section of this chapter.

4.3 Backwards Progpogation

Now that we know how to use a trained neural network, we need to learn how to train our own neural networks. Whenever we've trained a model before, we've determined an expression for the cost function, $J(\boldsymbol{\theta})$, and the gradient of the cost function, $\boldsymbol{\nabla} J(\boldsymbol{\theta})$, which we can compute for whatever our $\boldsymbol{\theta}$ currently is and then pass to our optimisation algorithm, however a neural network poses a new challenge in computing these values.

The cost function, $J(\boldsymbol{\Theta})$, is easy enough to compute, since we just need to have the value of our hypothesis for our given feature vector which we know how to calculate using forwards propogation. It can be written

$$\begin{aligned} J(\boldsymbol{\Theta}) = &-\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left(y_k^{(i)}\log\left(h_\Theta(x^{(i)})_k\right) + \left(1-y_k^{(i)}\right)\log\left(1-h_\Theta(x^{(i)})_k\right)\right) \\ &\ldots \quad + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ji}^{(l)}\right)^2 \end{aligned} \tag{4.4}$$

which includes regularization. Note that we don't regularize the weights that map from the bias units. The problem is computing the gradient of the cost function, $\boldsymbol{\nabla} J(\boldsymbol{\Theta})$, since we need to compute

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\boldsymbol{\Theta}) = D_{ij}^{(l)} \tag{4.5}$$

for every l, i and j. Note that $\Theta_{ij}^{(l)}$ is the weight going from $a_j^{(l-1)}$ to $a_i^{(l)}$. This means that we need to work out the partial derivatives of the cost function with respect to each and every connection between activation units. Furthermore, because each layer only acts on the output of the layer before it, once again we cant work out a lot of these partial derivatives directly, and we instead need to work our way backwards through the network, propogating the 'errors' of each layer backwards to work out our partial derivatives.

To work this out, we're going to define some new values $\delta_i^{(l)}$, which is the 'error' of node i in layer l and can be written as a vector $\boldsymbol{\delta^{(l)}}$. The first of these, δ^L is calculated fairly simply, by comparing the hypothesis to the target vector using the following equation:

$$\boldsymbol{\delta^{(L)}} = \boldsymbol{a^4} - \boldsymbol{y}. \tag{4.6}$$

The subsequent values are then calculated according to the same general formula

$$\boldsymbol{\delta^{(l)}} = \left(\boldsymbol{\Theta^{(l)}}\right)^\intercal \boldsymbol{\delta^{(l+1)}} \ .^* \ g'(\boldsymbol{z^{(l)}}) \tag{4.7}$$

where we define the $\cdot^*$ operator to be the element-wise multiplication of two vectors and $g'(\boldsymbol{z})$ is the derivative of the sigmoid function, which can be calculated using

$$g'(\boldsymbol{z^{(n)}}) = \boldsymbol{a^{(n)}} \ .^* \ \left(\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix} - \boldsymbol{a^{(n)}}\right). \tag{4.8}$$

Once the $\boldsymbol{\delta}$ vectors have been calculated, we can quite simply calculate the partial derivative terms by using

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\boldsymbol{\Theta}) = a_j^{(l)}\delta_i^{(l+1)} = D_{ij}^{(l)}. \tag{4.9}$$

An algorithmic approach to calculating these partial derivatives can be found in Chapter 1.

4.3.1 Matrix Unrolling

Most optimisation algorithms will expect $\boldsymbol{\Theta}$ and the partial derivatives to be in the form of vectors rather than matrices. To convert between the matrices that they exist as normally and vector representations, we can 'unroll' them. This means we write out each entry left to right first then top to bottom as a vector and do the inverse to convert back. In Octave, this can be done using the following example code:

```
% Unroll the Theta and D matrices into vectors
thetaVec = [Theta1(:); Theta2(:); Theta3(:)];
DVec = [D1(:); D2(:); D3(:)];

% Reshape the vector into matrices
Theta1 = reshape(thetaVec(1:110), 10, 11);
```

The documentation for the 'reshape' function is available through the help function in Octave.