Machine Learning Notes

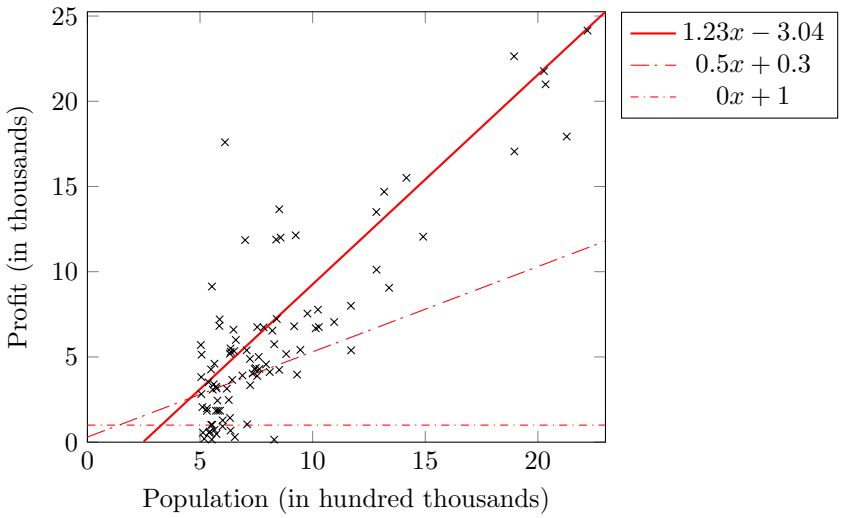
Andi Gu

April 2018

1 Linear Regression

The classic example of a machine learning algorithm is one that fits an equation to a set of data. For example, if we have some data on the population of cities and the profits made by some firm in these cities, we may try to fit an equation to these data points.

Figure 1: Profit vs. City Population



This technique is called linear regression, and works in a very simple way. We form a hypothesis about the nature of the line that fits the data. For this case, our hypothesis was:

$$y = \theta_0 + \theta_1 x$$

However, it is possible that the hypothesis takes on any range of forms – if we have more than one variable, we might see

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_2^2$$

If we want to write this in a more compact form, we might describe the define θ and x as

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \text{ and } x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

and now we can write our hypothesis $h_\theta(x)$ as the matrix product:

$$h_\theta(x) = \theta^T x$$

Now, we have n features and we introduce m data points. These data points are points for which both x and y are known. We can now put our data into a $m \times (n+1)$ matrix:

$$\begin{array}{c} \\ \text{Case } 1 \\ \text{Case } 2 \\ \\ \text{Case } m \end{array} \begin{array}{c} \begin{array}{ccccc} \text{Feature}^1\ 0 & \text{Feature } 1 & \text{Feature } 2 & \dots & \text{Feature } n \end{array} \\ \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \end{array} \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Now, we can apply our hypothesis h_θ to every case by simply multiplying X by θ. We define the vector $\hat{y}$ to be the result of this multiplication – it represents the hypothesized results for each test case.

1.1 Finding a Fit

We now wish to quantify exactly how good our hypothesis is in relation to the training data we are given. So we define the cost function J as a function of θ:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \Big(h_\theta(x^{(i)}) - y^{(i)} \Big)^2$$

We see that the farther $h_\theta(x^{(i)})$ drifts from the actual data $y^{(i)}$, the larger J becomes. Since our goal is to minimize J, we use a tool called **gradient descent**, whereby we gradually update each component of the vector θ depending on the derivative of J with respect to that component. More concretely, we update θ as follows:

$$\begin{aligned} \theta_j &= \theta_j - \alpha \frac{\partial J}{\partial \theta_j} \\ &= \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \Big[(h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \Big] \end{aligned}$$

Where α is a certain learning rate. It is important to set α carefully so that the cost function J decreases after every iteration (typical values are 0.001, 0.003, 0.01, 0.03, and so on).

However, these updates must all be conducted simultaneously. So, instead we can write this update in a 'vectorized' fashion, so that every element of θ is automatically updated simultaneously.

$$\theta = \theta - \frac{\alpha}{m} \big(X^T (X\theta - y) \big)$$

It can be shown that the curve J has a global minimum, so given that we set α carefully, θ will converge to a value so that J is minimized.

However, if we have a small number of features (typically $n < 1000$), it is practical to use an analytic solution[2] to θ. This is called the **normal equation**:

$$\theta = (X^T X)^{-1} X^T y$$

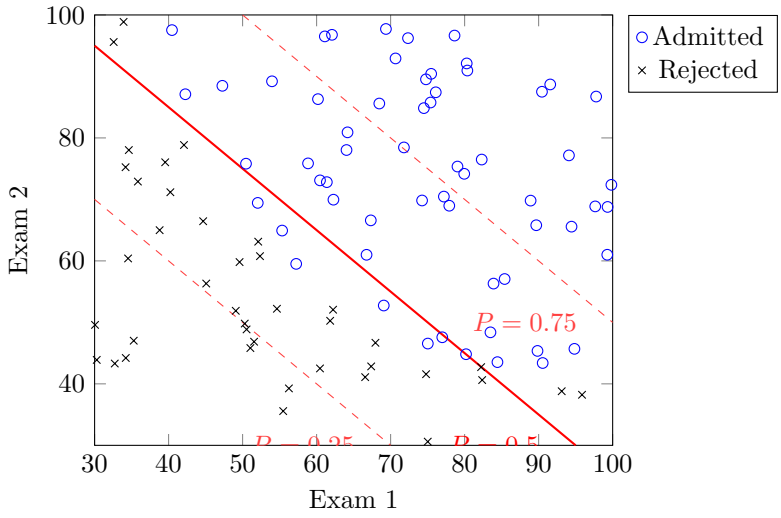
2 Logistic Regression

If we have two classes of data (e.g. failed students and passed students), we can instead have an algorithm that 'groups' these two classes of data.

[1] Typically this x_0 is 1, since we want our trendline to be $\theta_0 \cdot \mathbf{1} + \theta_1 x_1 + \ldots + \theta_n x_n$

[2] The inverse function $^{-1}$ is really the pseudoinverse, not the inverse.

Figure 2: Exam Scores



The goal of logistic regression is to find the probability that a given point belongs to a certain class of data given its features. The red line on the diagram is the 'decision boundary' and represents the locus of points where the data has a 50% chance of belonging to either class. We now create a model as follows. The probability h_θ that a test case with features x is 'positive' (in this case, accepted), is given by

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

We can now make an interesting observation. Say we want to find the locus of all points with a probability of P:

$$\begin{aligned} P &= \frac{1}{1 + e^{-\theta^T x}} \\ e^{-\theta^T x} &= \frac{1}{P} - 1 \\ \theta^T x &= \ln\left(\frac{P}{1 - P}\right) \end{aligned}$$

Since $\ln\left(\frac{p}{1-p}\right)$, if our $\theta^T x$ is in some linear form, the solution to this is a graph of a line. If it is something like $\theta_0 x_1^2 + \theta_1 x_2^2$, we will get an ellipse. So the 'shape' of the locus of points with identical probability depends on the way in which we design x – do we make it contain higher order terms, or keep it linear?

2.1 Finding a Fit

We must now design a new cost function. We define it as follows:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

where

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\ln(h_\theta(x)) & \text{if } y = 1 \\ -\ln(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

We can actually simplify this odd *Cost* function as follows:

$$\text{Cost}(h_\theta(x), y) = -y \ln(h_\theta(x)) - (1 - y) \ln(1 - h_\theta(x))$$

So now we have:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\Big[-y^{(i)}\ln\Big(h_\theta(x^{(i)})\Big) - (1-y^{(i)})\ln\Big(1-h_\theta(x^{(i)})\Big)\Big]$$

Interestingly enough, if we do gradient descent on this cost function, we get something very similar to the one we were doing in linear regression. Our update is:

$$\theta_j = \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}\Big[\Big(h_\theta(x^{(i)}) - y^{(i)}\Big)x_j^{(i)}\Big]$$

Now, if we define an operator g to take every element of a matrix and apply the function $\frac{1}{1+e^{-x}}$ to it, we can vectorize this implementation:

$$\theta = \theta - \frac{\alpha}{m}\big(X^T(g(X\theta) - y)\big)$$

2.2 Multiclass Classification

We now move on to the problem of more than one 'class' of data – for example, if we have weather that is sunny, rainy, or foggy. We can solve this by focusing on the i^{th} class of data and then regarding every single other piece of data as some 'other' class. We find the hypothesis $h_\theta^{(i)}(x)$ in this case. This i^{th} hypothesis represents the probability that a given data point is a member of the i^{th} class of data. We repeat this for every class of data, and then given a new unclassified data point, we plug it into every single hypothesis we have found. Then, we simply chose the class that maximizes this hypothesis output.

3 Feature Processing

We now introduce two processes to optimize our gradient descent.

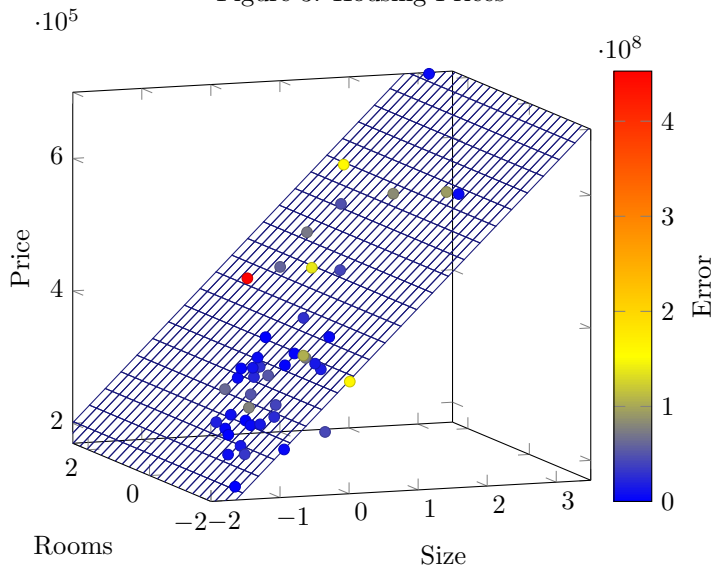
3.1 Feature Scaling

The first thing we do is to 'normalize' our data, so that every feature has a mean of 0 and a standard deviation of 1. We call this process feature scaling. We take a second look at our data matrix. We then calculate the mean $\overline{x_n}$ and standard deviation σ_n for each feature n (with the exception of the first column). We then transform the matrix as follows.

$$\begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & \frac{x_1^{(1)}-\overline{x_1}}{\sigma_1} & \frac{x_2^{(1)}-\overline{x_2}}{\sigma_2} & \dots & \frac{x_n^{(1)}-\overline{x_n}}{\sigma_n} \\ 1 & \frac{x_1^{(2)}-\overline{x_1}}{\sigma_1} & \frac{x_2^{(2)}-\overline{x_2}}{\sigma_2} & \dots & \frac{x_n^{(2)}-\overline{x_n}}{\sigma_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \frac{x_1^{(m)}-\overline{x_1}}{\sigma_1} & \frac{x_2^{(m)}-\overline{x_2}}{\sigma_2} & \dots & \frac{x_n^{(m)}-\overline{x_n}}{\sigma_n} \end{bmatrix}$$

In short for every test case m and every feature n, we transform the data by $x' = \frac{x_n^{(m)}-\overline{x_n}}{\sigma_n}$. The reason we do this is so that gradient descent runs faster – it turns out that if we have two feature sets that are orders of magnitude different, gradient descent will take a very slow path to the global minimum.

Figure 3: Housing Prices

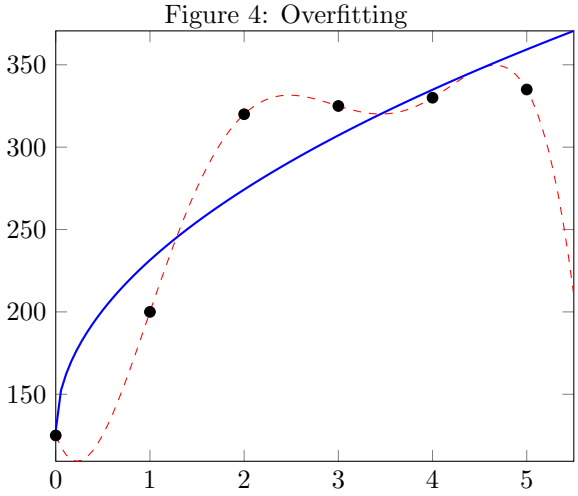


We see the data gets more 'centralized' or concentrated after normalization.

3.2 Feature Regularization

An issue we face is that it is actually possible to 'overfit' data. If we introduce several polynomial terms into our hypothesis, it is possible that the cost function shrinks extremely small, but the curve will then lose its predictive power. More concretely, say we have n data points. If we introduce a polynomial of order $n-1$, it is actually possible to achieve a cost function of 0 – that is, the curve perfectly matches up with every given data point. However, for new data points, the curve will have a very low probability of actually working. For example, we take a random data set of 5 points:

Figure 4: Overfitting



To counter this problem, we penalize the gradient descent algorithm as the values of the coefficients get larger. More concretely, we redefine J for linear regression as:

$$J(\theta) = \frac{1}{2m}\left(\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right)$$

Here, λ represents how strongly we want to punish high θ. For logistic regression, we do the same thing (simply adding a sum of all θ_j^2):

$$J(\theta) = \frac{1}{m}\left(\sum_{i=1}^{m}\left[-y^{(i)}\ln\Big(h_\theta(x^{(i)})\Big) - (1-y^{(i)})\ln\Big(1-h_\theta(x^{(i)})\Big)\right] + \lambda\sum_{j=1}^{n}\theta_j^2\right)$$

It's quite easy to show that the update rule for both takes on a new form that's nearly exactly the same as before.

$$\theta_j = \theta_j\Big(1-\alpha\frac{\gamma}{m}\Big) - \frac{\alpha}{m}\sum_{i=1}^{m}\left[\Big(h_\theta(x^{(i)}) - y^{(i)}\Big)x_j^{(i)}\right]$$

But for $j = 0$, we don't want to punish high θ_0. So our vectorized update rule looks like:

$$\theta = \theta - \begin{bmatrix} 0 \\ \alpha\frac{\gamma}{m} \\ \alpha\frac{\gamma}{m} \\ \vdots \\ \alpha\frac{\gamma}{m} \end{bmatrix} - \frac{\alpha}{m}\big(X^T(h_\theta(X) - y)\big)$$

Where $h_\theta(X)$ is the vector result of applying h_θ to every row in X.

Finally, we can develop the normal equation for linear regression with regularization. It is as follows:

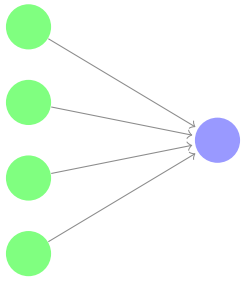
$$\theta = \big(X^TX + \gamma L\big)^{-1}X^Ty$$

where L is the $n \times n$ matrix:

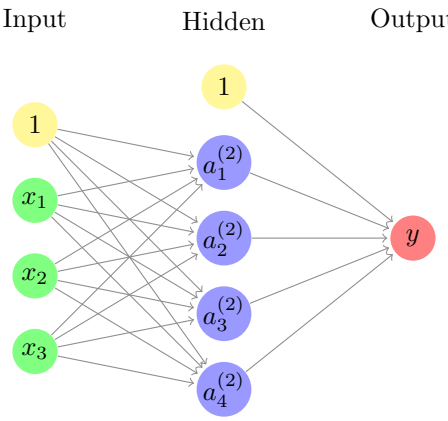
$$L = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

4 Neural Networks

The accepted model of a human brain has 100 billion neurons connected together, sending electrical impulses from one to the other. Since our goal is to mimic human intelligence, why not copy this model? So we create a structure known as a neural network, where we have neurons (in blue) accepting some numerical input from other neurons (in green). The blue neuron then does some simple transform on this data (perhaps summing some linear combination of the input data), and outputs it to the next neuron.



We now describe the complete neural network. It is composed of an input layer, any number of hidden layers, and one output layer.



We now introduce some notation. If we have data from a layer i that we call $a^{(i)}$, we will get the data for the next layer $a^{(i+1)}$ by transforming it with some Θ.

$$a^{(i)} = \begin{bmatrix} 1 \\ a_1^{(i)} \\ a_2^{(i)} \\ \vdots \end{bmatrix}$$

For convenience, we will refer to the j^{th} neuron in the i^{th} layer as $N_j^{(i)}$. We also index the matrix **starting from 0** for column-wise counting due to the presence of the 'bias' node, whose value is always 1.

$$\Theta^{(i)} = \begin{pmatrix} & \text{Bias} & N_1^{(i)} & N_2^{(i)} & \dots \\ N_1^{(i+1)} & \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & \dots \\ N_2^{(i+1)} & \Theta_{2,0} & \Theta_{2,1} & \Theta_{2,2} & \dots \\ & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

We can now say that:

$$a^{(i+1)} = \Theta^{(i)} a^{(i)}$$

However, we may sometimes want to apply a function to the data (perhaps the sigmoid function), so:

$$a^{(i+1)} = g\Big(\Theta^{(i)} a^{(i)}\Big)$$

Finally, we put all this together to see how we handle an input data set X with m test cases. To map X to the second layer $A^{(2)}$, we do $A^{(2)} = g\Big(X\big(\Theta^{(1)}\big)^T\Big)$, and add a column of 1's to the start of this matrix to represent the bias neurons.

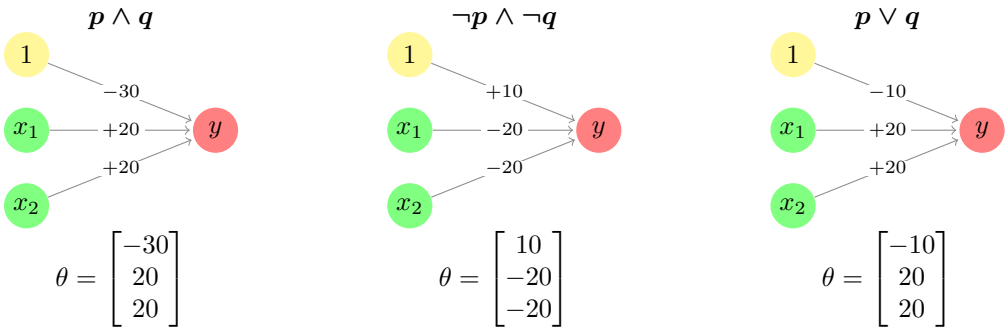
4.1 Intuition

The power of this method is not entirely clear at first. To perhaps understand it better, we will study the XOR function (represented by $\oplus$), whose truth table is as follows:

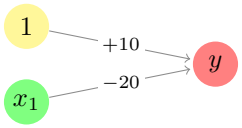
x_1	x_2	$y = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

And we design a neural network that will have this output, since $p \oplus q = \neg((p \wedge q) \vee (\neg p \wedge \neg q))$. There are simpler ways to write $p \oplus q$, but this one provides the best intuition.

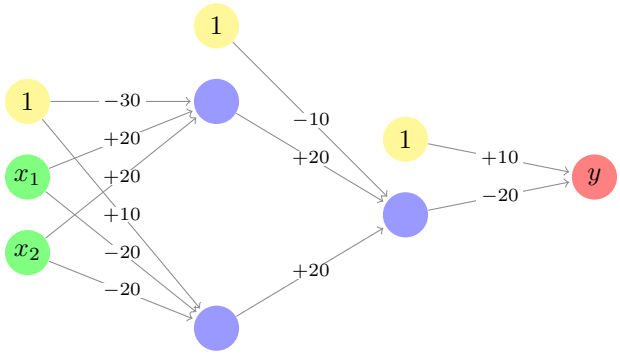
We begin by writing a neuron for each of the component functions $p \wedge q$, $\neg p \wedge \neg q$, and $p \vee q$:



Finally, we describe the neuron for $\neg p$:



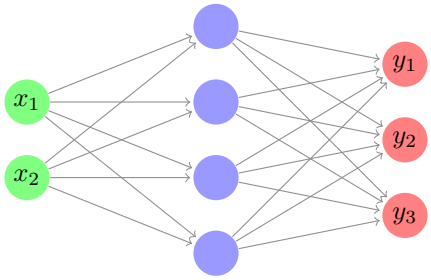
We can verify all of these by simply plugging in different pairings 0 and 1 for x_1 and x_2, running them through the neuron, and then applying the sigmoid function. It is clear that if $x\theta^T \leq -5$, $g(x\theta^T) \approx 0$. Conversely, if $x\theta^T \geq 5$, $g(x\theta^T) \approx 1$. We now combine all 4 to arrive at the following neural network:



So what neural networks allow is to do is to compose complicated functions, by making simple operations on each layer, and combining the results of these operations to get slightly more complex information on the next layer, and so on until we arrive at a useful, complex output.

4.2 Multiclass Classification

What happens when we wish to classify something as one of k groups, instead of just two? We simply change the neural network to have k outputs, where the i^{th} output represents the probability of the input representing the i^{th} class. The base nodes are not included in the below diagram for brevity.



So now our output is a vector y, with each element representing the probability that the input data was a certain class. We then classify the input as class i such that y_i is the largest probability in y.

4.3 Cost Function

The cost function for neural networks is similarly defined to the one used in logistic regression. The difference is now that we have K classes, so the cost now must be summed over each of the K classes so that:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} y_j^{(i)} \ln\Big(h_\Theta(x^{(i)})_j\Big) + (1 - y_j^{(i)}) \ln\Big(1 - h_\Theta(x^{(i)})_j\Big)$$

This can be vectorized if we convert y to a particular form, where each row is a row vector representing the desired output vector for a specific test case. For example, if we have K classes:

$$Y = \begin{pmatrix} & p(0) & p(1) & p(2) & & p(k) \\ \text{Case 1} & 1 & 0 & 0 & \dots & 0 \\ \text{Case 2} & 0 & 0 & 1 & \dots & 0 \\ \text{Case 3} & 0 & 1 & 0 & \dots & 0 \\ & \vdots & \vdots & \vdots & \ddots & \vdots \\ \text{Case } m & 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

If we write the hypothesis h_Θ in a similar form, we can form a new matrix with element-wise multiplication (written as $\circ$). For convenience, we introduce the *sum* function which finds the sum of every element in a matrix.

$$J = -\frac{1}{m} \cdot sum(Y \circ \ln(h_\Theta(X)) + (1 - Y) \circ \ln(1 - h_\Theta(X)))$$

If we wish to introduce regularization, we simply find the sum of the squares of every element in all Θ and add it to our original cost. The regularization term is thus:

$$\frac{\lambda}{2m}\sum_{i=1}^{L-1} sum\Big(\Theta^{(i)} \circ \Theta^{(i)}\Big)$$

4.4 Back-propagation

The process of forward propagation was how we got from our initial training data x to our hypothesis values h_Θ. We now introduce a process where we move 'backwards' through the neural network in order to find the partial derivative of the cost with respect to any particular element of any Θ: $\frac{\partial J}{\partial \Theta_{i,j}^{(l)}}$.

We introduce a $\delta^{(l)}$ term for every layer which is a vector with an entry for each node in the layer (including the bias unit, but we ignore it in the end). δ is defined as:

$$\delta^{(l)} = \Big(\Theta^{(l)}\Big)^T \delta^{(l+1)} \circ g'(z^{(l)})$$

Where g' is the derivative[3] of the sigmoid function. Perhaps a clearer element-wise definition is:

$$\delta_i^{(l)} = g'(z_i^{(l)}) \sum_{j=1}^{s^{(l+1)}} \Theta_{j,i}^{(l)} \delta_j^{(l+1)}$$

Where $s^{(l+1)}$ represents the number of units in the $l+1$ layer.

Finally, we can say

$$\frac{\partial J}{\partial \Theta_{i,j}^{(l)}} = \frac{1}{m}\delta_i^{(l+1)} a_j^{(l)} = \delta_i^{(l+1)} a_j^{(l)}$$

Where $a^{(l)}$ represents the data vector for a layer l with the sigmoid function applied.

The regularization term is relatively simple here, with the modification being:

$$\frac{\partial J}{\partial \Theta_{i,j}^{(l)}} = \left\{ \begin{array}{ll} \frac{1}{m}\Big(\delta_i^{(l+1)} a_i^{(l)} + \lambda \Theta_{i,j}^{(l)}\Big), & \text{for } j > 0 \\ \frac{1}{m}\delta_i^{(l+1)} a_i^{(l)}, & \text{for } j = 0 \end{array} \right\}$$

[3] It turns out if $g(z) = \frac{1}{1+e^{-z}}$, $g'(z) = g(z)(1 - g(z))$.

4.4.1 Proof of Correctness

It is possible to prove this back-propogation measure really does give us $\frac{\partial J}{\partial \Theta_{i,j}^{(l)}}$ by induction. First, we say the output of our neural network is H and its error function is J as defined before. Now, we introduce a term $z_i^{(l)}$ which represents the sum of all incoming data into the i^{th} neuron on layer l. Essentially, it represents the input to a certain neuron. To find the partial derivative of the cost J with respect to a given parameter $\Theta_{i,j}^{(l)}$, we use the chain rule and say:

$$\begin{aligned}
\frac{\partial J}{\partial \Theta_{i,j}^{(l)}} &= \frac{\partial J}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} \\
&= \frac{\partial J}{\partial z_i^{(l+1)}} \cdot \frac{\partial}{\partial \Theta_{i,j}^{(l)}} \left(\sum_{t=1}^{s^{(l)}} \Theta_{i,t}^{(l)} a_t^{(l)} \right) \\
&= \frac{\partial J}{\partial z_i^{(l+1)}} \cdot a_j^{(l)}
\end{aligned}$$

Now we define $\delta_i^{(l+1)}$ to be the $\frac{\partial J}{\partial z_i^{(l+1)}}$ term. So the difficulty is finding $\delta_i^{(l+1)}$. To do this, we study the effect of changing $z_i^{(l+1)}$ on layer $l+2$, and we also assume that we know the partial derivatives $\frac{\partial J}{\partial z_t^{(l+2)}}$ for $1 \leq t \leq s^{(l+2)}$.

$$\begin{aligned}
\Delta J &= \sum_{t=1}^{s^{(l+2)}} \frac{\partial J}{\partial z_t^{(l+2)}} \cdot \Delta z_t^{(l+2)} \\
&= \sum_{t=1}^{s^{(l+2)}} \frac{\partial J}{\partial z_t^{(l+2)}} \cdot \left(g\left(z^{(l+1)} + \Delta z_i^{(l+1)} \Theta_{t,i}^{(l+1)} \right) - g\left(z^{(l+1)} \right) \right)
\end{aligned}$$

We can now derive with respect to $z_i^{(l+1)}$ by observing the limit as $\Delta z_i^{(l+1)} \to 0$.

$$\begin{aligned}
\frac{\partial J}{\partial z_i^{(l+1)}} &= \sum_{t=1}^{s^{(l+2)}} \frac{\partial J}{\partial z_t^{(l+2)}} \cdot g'(z^{(l+1)}) \cdot \Theta_{t,i}^{(l+1)} \\
\delta_i^{(l+1)} &= g'(z^{(l+1)}) \cdot \sum_{t=1}^{s^{(l+2)}} \delta_t^{(l+2)} \cdot \Theta_{t,i}^{(l+1)}
\end{aligned}$$

We now have nearly all the tools necessary to find the derivative w.r.t $\Theta_{i,j}^{(l)}$ for and l, i, and j. All that's left is to find $\delta_i^{(L)}$ (for the output layer). Going back to the definition of J, we see that:

$$\begin{aligned}
J &= \sum_{k=1}^{K} \left[-y_k \ln(h_\theta(x))_k - (1 - y_k) \ln\left(1 - h_\theta(x^{(i)})_k \right) \right] \\
&= \sum_{k=1}^{K} \left[-y_k \ln\left(g(z_k^{(L)}) \right) - (1 - y_k) \ln\left(1 - g(z_k^{(L)}) \right) \right]
\end{aligned}$$

It can be shown that

$$\frac{\partial J}{\partial z_i^{(L)}} = \delta_i^{(L)} = y_i - a_i^{(L)}$$

All of this was done with only 1 test case – if we begin assuming m cases, we simply add a factor of $\frac{1}{m}$ to our final derivative $\frac{\partial J}{\partial \Theta_{i,j}^{(l)}}$ and sum each of our $\delta_i^{(l+1)} \cdot a_j^{(l)}$ terms over all m test cases.

5 Designing Machine Learning Systems

When we begin building a machine learning model, it may be difficult to choose a model. Two simple examples may be choosing the degree of the polynomial in a linear regression model, or choosing a λ with which to punish high θ values. To solve this issue, we separate our data into three randomized disjoint groups: one training group, one 'cross-validation' group, and one test group. A typical split may be 60-20-20. We then proceed as follows.

1. We train our various models on the training group. If we are considering different degrees d for our polynomial (in bivariate linear regression), we may test $d = 1, 2, 3, \ldots, 15$. In this step, we apply regularized costs.
2. For each of these trained models, we check the unregularized cost if we apply the model to our cross validation set ($\lambda = 0$).
3. We select the model with the lowest unregularized cost in the previous step, and test this against the test group.

5.1 Trouble shooting

To begin, we call the **unregularized** error for the training, cross-validation, and test groups J_{train}, J_{CV}, J_{test} respectively. We use J_{test} as a metric for how well our system is performing – our goal is to see a low J_{test} (not underfitting data) that closely matches the J_{train} (not overfitting, since this means the model generalizes well beyond just the training set).

If our model is underfitting (high bias) or overfitting (high variance), there is a variety of steps we can take:

Table 1: Methods of Troubleshooting

Underfitting	Overfitting
Increase degree of polynomial	Introduce more test data
Decrease λ	Increase λ
Introduce additional features	Try smaller sets of features

5.2 Skewed Data

There are times at which the data we get is largely skewed in favor of a certain class – for example, if we are building a spam filter, 1% of our data is actual spam mail and the other 99% is non-spam. This introduces a problem. If we build an algorithm that predicts non-spam all the time, we will achieve 99% accuracy – which at first glance seems good! However, we have missed the entire point of the spam filter. So we introduce two measures called precision and recall.

Table 2: Methods of Troubleshooting

	Actual 1	Actual 0
Predicted 1	True positive	False positive
Predicted 0	False negative	True negative

By convention, we call the rarer class of data '1'. We define precision p to be the ratio of number of true positives to number of predicted 1's. In other words, p measures the fraction of positive predictions that are correct. On the other hand, recall r measures the fraction of positive events that are correctly recognized. In logistic regression, there is a trade off between these two values if we vary the threshold for which the hypothesis must cross to be considered true. If we set the threshold to be ≥ 0.9, this means we want to be very sure of our prediction – thereby raising precision (since we will decrease number of false positives) and lowering recall (since number of false negatives will rise). The same goes for lowering the threshold to

something very small like ≤ 0.1, with a lowering precision and rising recall. To combine these two metrics into one, we develop a metric called the F-score:

$$F = 2\frac{p \cdot r}{p + r}$$

We see now that if either p or r is very small, the score will be low since the product $p \cdot r$ will be very small. In cases of skewed data, we often want to maximize this F-score.