Notes on Machine Learning from Andrew Ng's Coursera Course

Simon Zahn

September 5, 2016

Contents

Preface

This document started out as my notes while a student in Andrew Ng's Machine Learning course on Coursera. If this subject is of any interest to you, I highly recommend you check out the course here.

All credit for content contained herein goes to Andrew Ng and his course.

In creating these notes, there are some changes I have made to the content. Primarily, I switched from using the programming language used for the course, Octave, to Python. I believe Python is a more practical choice, and it is also my programming language of choice. Additionally, by using Python, I believe I am not violating any aspect of Coursera's honor code by posting some of my solutions to the exercises. The solutions look completely different in Python, and therefore won't really be of any help to those trying to complete the course. Whether you know Python or not, the notes on the mathematics and the course content should prove useful. I also include random tidbits of ML knowledge I've picked up from other places sporadically throughout.

These notes fill follow this format: each chapter will cover a week of the Coursera course. The theory that is covered in Professor Ng's videos will be the majority of these notes. The homework, coded in Python, comes next. It goes over my implementation of the methods discussed in the course. I'll include some of the homework text, but I may remove any Octave or MATLAB specifics, since they don't apply here. Finally, I'll end most sections with notes on 'Python Labs,' which will be using pre-built Python modules to implement these techniques.

The most recent version of these notes will be kept on my GitHub page at `https://github.com/Sz593/coursera_ml_notes`. If you have questions or comments, you can email me at simonzahn@gmail.com.

Chapter 1

Introduction

Two definitions of Machine Learning are offered at the start of the course:

Arthur Samuel 'The field of study that gives computers the ability to learn without being explicitely programmed.'

Tom Mitchell 'A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks T, as measured by P, improves with experience E.'

1.1 Supervised Learning

In supervised learning, we have a data set and already know what the correct output should be, knowing that there is an existing relationship between the input and output. There are two types of supervised learning: **classification**, and **regression**.

In a regression problem, we're trying to predict results with a continuous output; i.e. we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results with a discrete output. Let's look at some examples:

Regression If we're trying to predict the price of a house given data about the house such as its size, location, etc., this is a regression problem.

Regression Given a picture of a person, try to predict his/her age.

Classification Given a picture of a person, try to predict if he/she is of high school, college, or graduate age.

Classification As a bank, decide whether or not to give a loan to a potential borrower.

1.2 Unsupervised Learning

Unsupervised learning allows us to approach problems with little or no idea about what our results should look like. We can try and derive structure from data where we don't necessarily know the effect of the variables. We can derive this structure by clustering the data based on relationships among the variables in the data. With unsupervised learning, there is no feedback based on the prediction results.

One example of unsupervised learning is clustering. Imagine we take 1000 essays written by US college students. We can try and automatically group these essays into a smaller number that are somehow similar or related by different variables; word frequency, sentence length, page count, etc.

Here is an example of unsupervised learning that isn't clustering: the cocktail party algorithm. This is a way to find structure in messy data, such as the identification of individual voices and music from a mesh of sounds at a cocktail party.[1]

[1] For more information about auditory filtering, look at Wikipedia's Cocktail Party Effect article.

Chapter 2

Linear Regression

In regression problems, we take a variable (or multiple variables) as input, and try to fit the output to a continuous expected result function.

2.1 Univariate Linear Regression

In univariate linear regression, we want to predict a single output value $\hat{y}$ from a single input value x. Since this is supervised learning, we already have an idea about what the input/output relationship should look like.

2.1.1 The Hypothesis Function

Imagine we have a problem where the input is x and the output is y. In order to do machine learning, there should exist a relationship (a pattern) between the input and output variables. Let's say this function is $y = f(x)$. In this situation, f is known as the target function. However, this function f is unknown to us, so we need to try and guess what it is. To do that, we form a *hypothesis* function $h(x)$ that approximates the unknown $f(x)$.

For single variable linear regression, our hytothesis function takes two parameters: θ_0 and θ_1. As such, we often write it as $h_\theta(x)$, and it takes the form

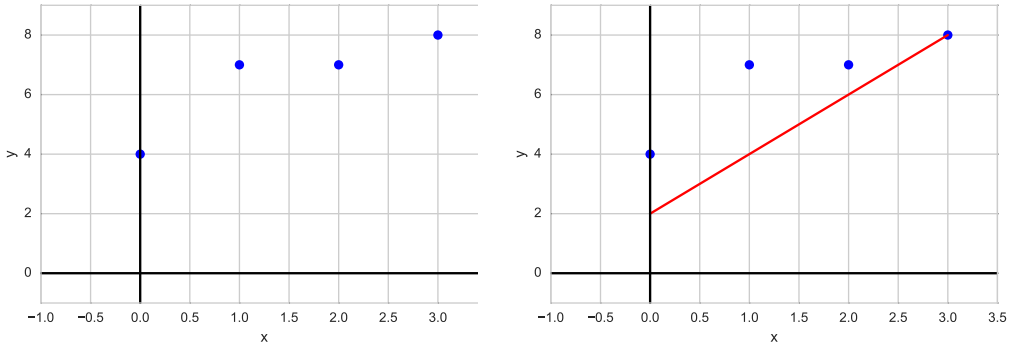
$$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x \tag{2.1}$$

Note that this is the equation of a straight line ($y = mx + b$). We're trying to find the values for θ_0 and θ_1 to get our estimated output $\hat{y}$. In other words, we're trying to determine the function h_θ that maps our input data (the x's) to our output data (the y's).

Suppose we have the following set of training data:

Input x	Output y
0	4
1	7
2	7
3	8

We can plot these points, as shown in Figure 2.1a. Let's make a random guess at our hypothesis function: $\theta_0 = 2$ and $\theta_1 = 2$, making our hypothesis function $h_\theta(x) = 2 + 2x$, as shown in Figure 2.1b.



(a) Plotting our example points.

(b) Plotting our example points.

Using this hypothesis function for $x = 1$, we have $\hat{y} = h_\theta(1) = 2 + 2 \cdot 1 = 4$. In this case, $\hat{y} = 4$, but $y = 7$, so mabe this isn't the best fit hypothesis.

2.1.2 The Cost Function

The cost function,[1] is a function used for parameter estimation, where the input to the cost function is some function of the difference between estimated and the true values for an instance of data. In this case, we can use the cost function to measure the accuracy of our hypothesis function.

The cost function looks at something similar to an average[2] of all the results of the hypothesis with inputs from the x's compared to the actual output y's. We define our cost function as follows:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2 \tag{2.2}$$

This is known as the **mean squared error**. If we set $\bar{x}$ equal to the mean of the squares all the $h_\theta(x_i) - y_i$, then the cost function is just the mean of $\bar{x}$. The term $\frac{1}{2m}$ is merely a convenience for the computation of gradient descent, which we'll see very shortly.

[1] The cost function can also be called the loss function.

[2] It's actually something a bit fancier than a standard average.

2.1.3 Gradient Descent

We now have our hypothesis function defined, as well as a way of measuring how well it fits the data. Now, we have to estimate the parameters in the hypothesis function, and that's where gradient descent comes in.

Let's graph our cost function as a function of the parameter estimates. This can be somewhat confusing, as we are moving up to a higher level of abstraction. We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting particular sets of parameters. We put θ_0 on the x-axis, and θ_1 on the y-axis, with the cost function on the vertical z-axis.

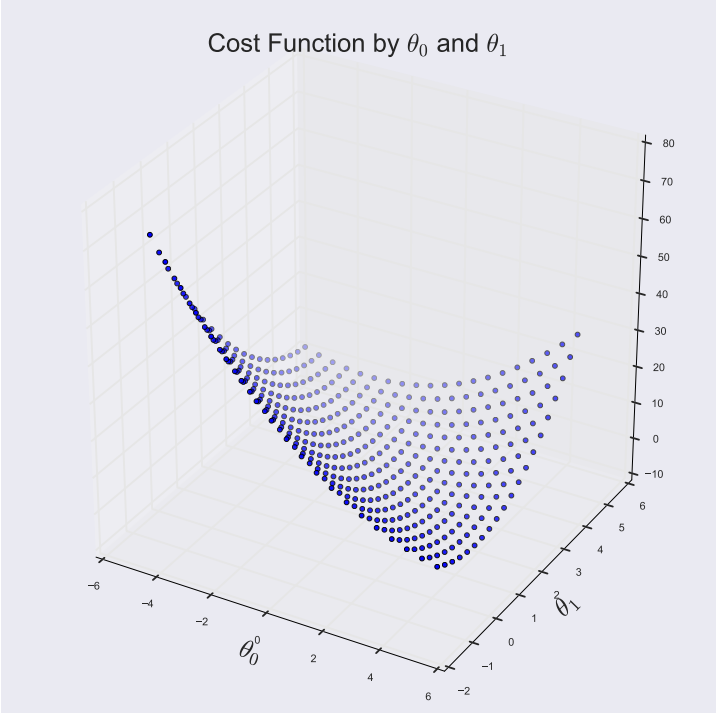


Figure 2.2: Plot of the cost function $J(\theta_0, \theta_1)$ using our hypothesis $h_\theta(x)$.

Our goal is to take the parameters θ_0 and θ_1 for when the cost function is at its minimum. We can calculate this value by taking the derivative of the cost function, which gives us direction of the steepest gradient to move towards. Take a step in that direction, and repeat.The step size is determined by the parameter α, which is called the **learning rate**. The gradient descent algorithm is:

Repeat until convergence:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \tag{2.3}$$

where $j = 0, 1$ represents the feature index number.

2.1.3.1 Gradient Descent for Linear Regression

When specifically applied to the case of univariate linear regression, we can derive another form of the gradient descent equation. If we substitute our actual hypothesis function and cost function, we can modify the equation to

Repeat until convergence: {

$$\begin{aligned} \theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i) \\ \theta_1 &:= \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} ((h_\theta(x_i) - y_i) x_i) \end{aligned} \tag{2.4}$$

}

where m is the size of the training set, θ_0 is a constant that will be changing simultaneously with θ_1, and x_1, y_i are values of the given training set. Note that we have separated out the two cases for θ_j into separate equations for θ_0 and θ_1, and that for θ_1 we are multiplying x_i at the end due to the derivative.

2.2 Multivariate Linear Regression

Let's start by looking at some sample housing data with multiple features.

Size (feet^2)	# of Bedrooms	# of Floors	Age (years)	Price (in 1000's of \$)
x_1	x_2	x_3	x_4	y
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
560	1	1	12	155

In this, we can introduce some notation:

- The variables x_1, x_2, etc. are the features.
- The variable y is the output variable.
- The number of input features is denoted n. In this example, $n = 4$.
- m specifies the number of training examples (rows). Here, $m = 5$.
- $x^{(i)}$ is the input (features) of the i^{th} training example. So $x^{(2)}$ is the column vector $[1416, 3, 2, 40, 232]$.

- $x_j^{(i)}$ is feature j in the i^{th} training example. Here, $x_1^{(4)} = 852$.

At this point, we can define the multivariable form of the hypothesis function for linear regression:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n \tag{2.5}$$

For convenience of notation, we will define $x_0 = 1$ for all feature vectors ($x_0^{(i)} = 1$). So now, if we include x_0, our hypothesis function takes the form:

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i x_i \tag{2.6}$$

Now, we can also write the x values and θ values as vectors:

$$\vec{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1} \qquad \text{and} \qquad \vec{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

In vector notation, this is

$$h_\theta(x) = \vec{\theta}^{\mathsf{T}} \vec{x} \tag{2.7}$$

where we transpose $\vec{\theta}$ into a row vector so we're able to take the inner product.

Now that we have our vector $\vec{\theta} \in \mathbb{R}^{n+1}$, the cost function is

$$J\left(\vec{\theta}\right) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 \tag{2.8}$$

2.2.1 Gradient Descent for Multiple Variables

Using our expanded hypothesis and cost functions, the gradient descent algorithm becomes:

Repeat until convergence: {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x_j^{(i)} \qquad \text{for } j = 0, 1, \cdots, n \tag{2.9}$$

}

2.2.2 Feature Scaling

When features are in very different ranges, it can slow down gradient descent dramatically (and also mess up our machine learning algorithms!), because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to

the minimum. The way to prevent this is to ensure that all the ranges are roughly the same, ideally:

$$-1 \leq x \leq 1$$

Two techniques to accomplish this are **feature scaling** and **mean normalization**. Feature scaling involved dividing the input values by the range (max value minus the min value) of the input variable, resulting in a new range of just 1.

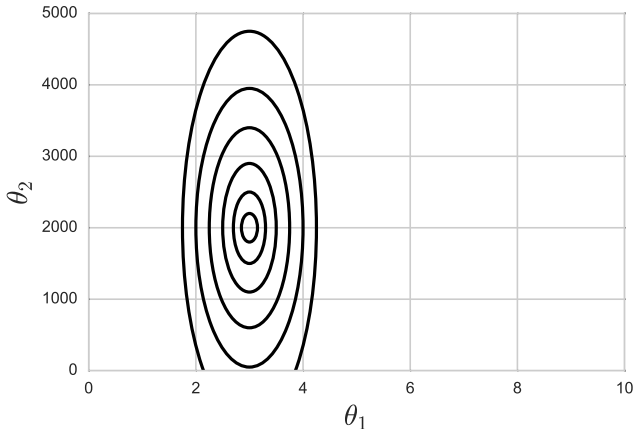


Figure 2.3: When one feature is on a much larger scale than the other, the plot of the cost function will be stretched out in the direction of the larger feature. Here, imagine that θ_1 is the number of bedrooms a house has, and θ_2 is the size in square feet.

Mean normalization involves subtracting the mean value for an input variable from the value for that input variable, resulting in a new mean of zero. To implement both of these simultaneously, use the following formula:

$$x_i := \frac{x_i - \mu_i}{s_i} \tag{2.10}$$

where μ_i is the average value of x_i, and s_i can either be the range ($x_{\max} - x_{\min}$) or the standard deviation.

2.2.3 Tips for Gradient Descent

Here are some of Professor Andrew Ng's tips on implementing gradient descent.

1. **Plot $J(\theta)$.** If you plot $J(\theta)$ as the ordinate and the number of iterations as the abscissa,[3] the graph should be steadily decreasing with increasing number of iterations. If $J(\theta)$ ever increases, then α is probably too large.

2. If $J(\theta)$ decreases by less then E in one iteration, where E is some very small number, such as 10^{-3}, then you can declare convergence.

[3] On a Cartesian coordinate plane, the ordinate is the y-axis and the abscissa is the x-axis.

3. For sufficiently small α, $J(\theta)$ should decrease with every iteration. To choose α, try a range of values for α with threefold increases, such as:

$$\cdots \to 0.001 \to 0.003 \to 0.01 \to 0.03 \to 0.1 \to 0.3 \to 1 \to 3 \to \cdots$$

4. Sometimes, it's better to define new features instead of using the ones given. For example, if we have a house with features frontage[4] and depth,[5] you can combine these into a new feature called area, which is how much land the house sits on.

2.2.4 Polynomial Regression

The form of the hypothesis doesn't necessarily need to be linear of that doesn't fit the data well. We can change the behavior or curve of our hypothesis function by making it quadratic, cubic, square root, or some other form.

For example, if our hypothesis function is $h_\theta(x) = \theta_0 + \theta_1 x_1$, we can create additional features based on x_1, to get the quadratic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$, or the cubic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$.

When thinking about nonlinear features, it is important to keep in mind that features scaling becomes even more essential than it was for linear regression. If x_1 has a range of 1 to 1000, the x_1^2 has a range of 1 to $1,000,000$.

2.3 Vectorized Equations

Let's revisit our housing example from §2.2. Recall that we looked at the the following example data, and we'll add an extra column for x_0 that always takes a value of one:

	Size (feet^2)	# Bedrooms	# Floors	Age (years)	Price (in \$1000's)
x_0	x_1	x_2	x_3	x_4	y
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1534	3	2	30	315
1	852	2	1	36	178
1	560	1	1	12	155

From this, we construct a matrix X that contains all of the features from the training data, and a vector $\vec{y}$ of all the output data.

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \\ 1 & 560 & 1 & 1 & 12 \end{bmatrix} \quad \text{and} \quad \vec{y} = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \\ 155 \end{bmatrix}$$

[4] The width of the land in the front of the house.
[5] The width of the land on the side of the house.

Here, X is a $m \times (n+1)$ matrix, and $\vec{y}$ is a m-dimensional vector.

Let's go through this again, but this time in full abstraction. Say we have m examples $\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right)$ and each $x^{(i)}$ has n features. Then, we have an $(n+1)$-dimensional feature vector:

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in \mathbb{R}^{n+1} \tag{2.11}$$

The matrix X, which is also called the **design matrix**, is constructed by taking the transpose of each vector $x^{(i)}$. Each feature vector $x^{(i)}$ becomes a row in the design matrix. Just as previously, the output vector $\vec{y}$ is obtained by taking all the labels and stacking them up into an m-dimensional vector, and the vector $\vec{\theta}$ is created from stacking all of the parameters for the hypothesis function.

$$X = \begin{bmatrix} -- & -- & \left(x^{(1)}\right)^\intercal & -- & -- \\ -- & -- & \left(x^{(2)}\right)^\intercal & -- & -- \\ -- & -- & \left(x^{(3)}\right)^\intercal & -- & -- \\ -- & -- & \left(x^{(4)}\right)^\intercal & -- & -- \\ -- & -- & \vdots & -- & -- \\ -- & -- & \left(x^{(m)}\right)^\intercal & -- & -- \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ y^{(4)} \\ \vdots \\ y^{(m)} \end{bmatrix} \qquad \vec{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_n \end{bmatrix} \tag{2.12}$$

Think back to the start of this section when we separated our table into the design matrix X and the output vector y. The design matrix is simply the data as stored in a table put into a matrix.

In our matrix notation for multivariate regression, the hypothesis function takes the form

$$h_\theta(X) = X\vec{\theta} \tag{2.13}$$

This will always work since X is an $m \times n$ matrix, and $\vec{\theta}$ is an $n \times 1$ vector. In a similar fashion, the cost function in matrix notation is

$$J\left(\vec{\theta}\right) = \frac{1}{2m}\left(X\vec{\theta} - \vec{y}\right)^\intercal \left(X\vec{\theta} - \vec{y}\right) \tag{2.14}$$

The gradient descent rule can be expressed as

$$\vec{\theta} := \vec{\theta} - \alpha \nabla J(\theta) \tag{2.15}$$

There ∇ is the gradient (vector derivative) operator. If we solve this out using our vectorized hypothesis function, we get

$$\vec{\theta} := \vec{\theta} - \frac{\alpha}{m} X^\intercal \left(X\vec{\theta} - \vec{y}\right) \tag{2.16}$$

2.4 The Normal Equation

The normal equation is a method of solving for the optimal θ analytically, that is, without iteration. From calculus, if we want to find the minimum of a quadratic equation, we set the derivative equal to zero, and solve. We can apply the same logic to the cost function. If we take the partial derivative $\partial/\partial\theta_j J(\theta)$ and set this equal to zero for all values of j, we'll analytically solve for the minimum.

The derivation of the normal equation is fairly involved from a linear algebra perspective, so at this point just take it as a fact:

$$\vec{\theta} = (X^\intercal X)^{-1} X^\intercal \vec{y} \tag{2.17}$$

When deciding whether to use gradient descent of the normal equation, consider the following:

Gradient Descent	**Normal Equation**
Need to choose α	No need to choose α
Needs many iterations	No need to iterate
$O\left(kn^2\right)$	$O\left(n^3\right)$, need to calculate $X^\intercal X$
Works well when n is large	Slow if n is very large

With the normal equation, computing the inverse has complexity $O\left(n^3\right)$. If we have a large number of features, this will cause the normal equation to perform slowly. In practice, when n exceeds $10,000$, it would probably be a good idea to use gradient descent.

2.4.1 Normal Equation Noninvertibility

When implementing the normal equation, sometimes the matrix $X^\intercal X$ is noninvertible. The common causes are:

- Redundant features, where two or more features are linearly dependent
- Too many features (i.e. $m \leq n$). In this case, delete some features or use regularization (which we'll get to later)

We typically avoid this problem by coding a pseudoinverse, instead of taking the actual inverse.

2.5 Homework

Introduction

In this exercise, you will implement linear regression and get to see it work on data. Before starting on this programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics. To get started with the exercise, you will need to download the starter code and unzip its contents to

the directory where you wish to complete the exercise. If needed, use the cd command in Octave/MATLAB to change to this directory before starting this exercise. You can also find instructions for installing Octave/MATLAB in the Environment Setup Instructions" of the course website.

To start our code, we'll need to import a few modules that we'll be using.

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline        # Use this if you're using Jupyter Notebooks
```

Simple Octave/MATLAB Function

The first part of ex1.m gives you practice with Octave/MATLAB syntax and the homework submission process. In the file warmUpExercise.m, you will find the outline of an Octave/MATLAB function. Modify it to return a 5 x 5 identity matrix.

```
A = np.eye(5)
print(A)
```

Linear Regression with One Variable

In this part of this exercise, you will implement linear regression with one variable to predict profits for a food truck. Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet. The chain already has trucks in various cities and you have data for profits and populations from the cities. You would like to use this data to help you select which city to expand to next.

Plotting the Data

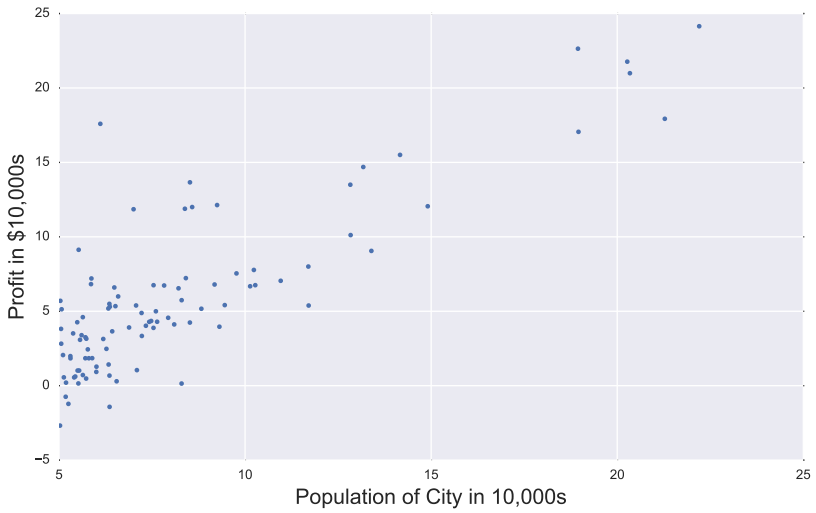
Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population). (Many other problems that you will encounter in real life are multi-dimensional and can't be plotted on a 2-d plot.)

We start by loading the data

```
datafile = 'ex1\\ex1data1.txt'
df = pd.read_csv(datafile, header=None, names=['Population', 'Profit'])
```

Then we define a function to plot the data and call this function.

```
def plot_data(x, y):
    plt.figure(figsize=(10, 6))
    plt.plot(x, y, '.', label='Training Data')
    plt.xlabel("Population of City in 10,000s")
    plt.ylabel("Profit in $10,000s")

plot_data(df['Population'], df['Profit'])
```



Gradient Descent

In this part, you will fit the linear regression parameters θ to our dataset using gradient descent.

Update Equations

The objective of linear regression is to minimize the cost function

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

where $h_\theta(x)$ is the hypothesis given by the linear model

$$h_\theta(x) = \theta^\intercal x = \theta_0 + \theta_1 x_1$$

Recall that the parameters of your model are the θ_j values. These are the values you will adjust to minimize cost $J(\theta)$. One way to do this is to use the batch gradient descent

algorithm. In batch gradient descent, each iteration performs the update

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)} \qquad \text{simultaneously update } \theta_j \text{ for all } j.$$

With each step of gradient descent, your parameters θ_j come closer to the optimal values that will achieve the lowest cost $J(\theta)$.

Implementation

In the following lines, we add another dimension to our data to accommodate the θ_0 intercept term.

```
# set the number of training examples
m = len(df['Population'])

# create an array from the dataframe (missing column for x_0 values)
X = df['Population'].values

# add in the first column of the array for x_0 values
X = X[:, np.newaxis]
X = np.insert(X, 0, 1, axis=1)

y = df['Profit'].values
y = y[:, np.newaxis]
```

Computing the Cost $J(\theta)$

Now, we can define our actual hypothesis function for linear regression with a single variable.

```
# define the hypothesis
def h(theta, X):
    """Takes the dot product of the matrix X and the vector theta,
    yielding a predicted result.
    """
    return np.dot(X, theta)


def compute_cost(X, y, theta):
    """Takes the design matrix X and output vector y, and computes the cost of
    the parameters stored in the vector theta.

    The dimensions must be as follows:
    - X must be m x n
```

```
    - y must be m x 1
    - theta must be n x 1

    """
    m = len(y)

    J = 1 / (2*m) * np.dot((np.dot(X, theta) - y).T, (np.dot(X, theta) - y))
    return J
```

Once you have completed the function, the next step once using θ initialized to zeros, and you will see the cost printed to the screen. You should expect to see a cost of 32.07.

```
# define column vector theta = [[0], [0]]
theta = np.zeros((2, 1))

# compute the cost function for our existing X and y, with our new theta vector
# verify that the cost for our theta of zeros is 32.07
compute_cost(X, y, theta)
```

This gives us our expected value of 32.07273388.

Gradient Descent

Now we'll actually implement the gradient descent algorithm. Keep in mind that the cost $J(\theta)$ is parameterized by the vector θ, not X and y. That is, we minimize $J(\theta)$ by changing θ. We initialize the initial parameters to 0 and the learning rate alpha to 0.01.

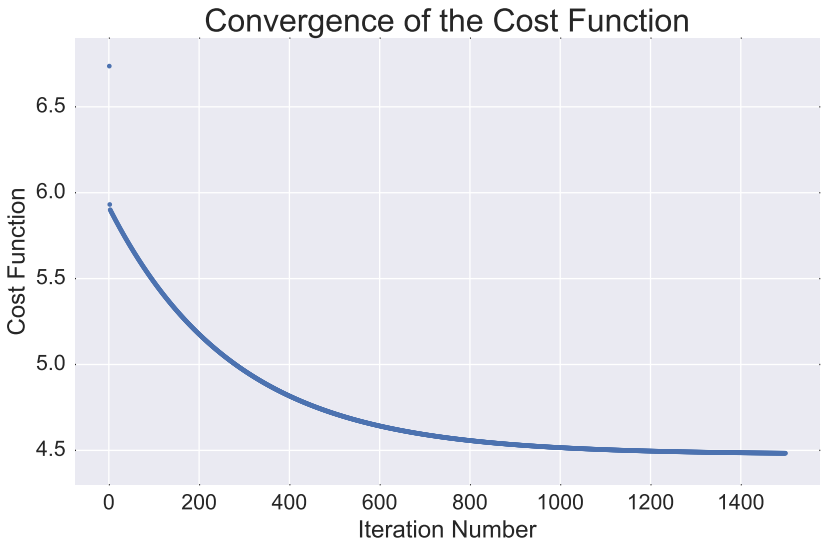
```
def gradient_descent(X, y, theta, alpha, num_iters):
    m = len(y)
    J_history = []
    theta_history = []

    for i in range(num_iters):
        J_history.append(float(compute_cost(X, y, theta)))
        theta_history.append(theta)
        theta = theta - (alpha / m) * np.dot(X.T, (np.dot(X, theta) - y))

    return theta, J_history, theta_history


# set up some initial parameters for gradient descent
theta_initial = np.zeros((2, 1))
iterations = 1500
alpha = 0.01
```

```
# run our gradient descent function
theta_final, J_hist, theta_hist = gradient_descent(X, y,
                                                   theta_initial,
                                                   alpha, iterations)
```

Visualizing $J(\theta)$

After running the batch gradient descent algorithm, we can plot the convergence of $J(\theta)$ over the number of iterations.

```
def plot_cost_convergence(J_history):
    abscissa = list(range(len(J_history)))
    ordinate = J_history

    plt.figure(figsize=(10, 6))
    plt.plot(abscissa, ordinate, '.')
    plt.title('Convergence of the Cost Function', fontsize=18)
    plt.xlabel('Iteration Number', fontsize=14)
    plt.ylabel('Cost Function', fontsize=14)
    plt.xlim(min(abscissa) - max(abscissa) * 0.05, 1.05 * max(abscissa))


plot_cost_convergence(J_hist)
plt.ylim(4.3, 6.9)
```

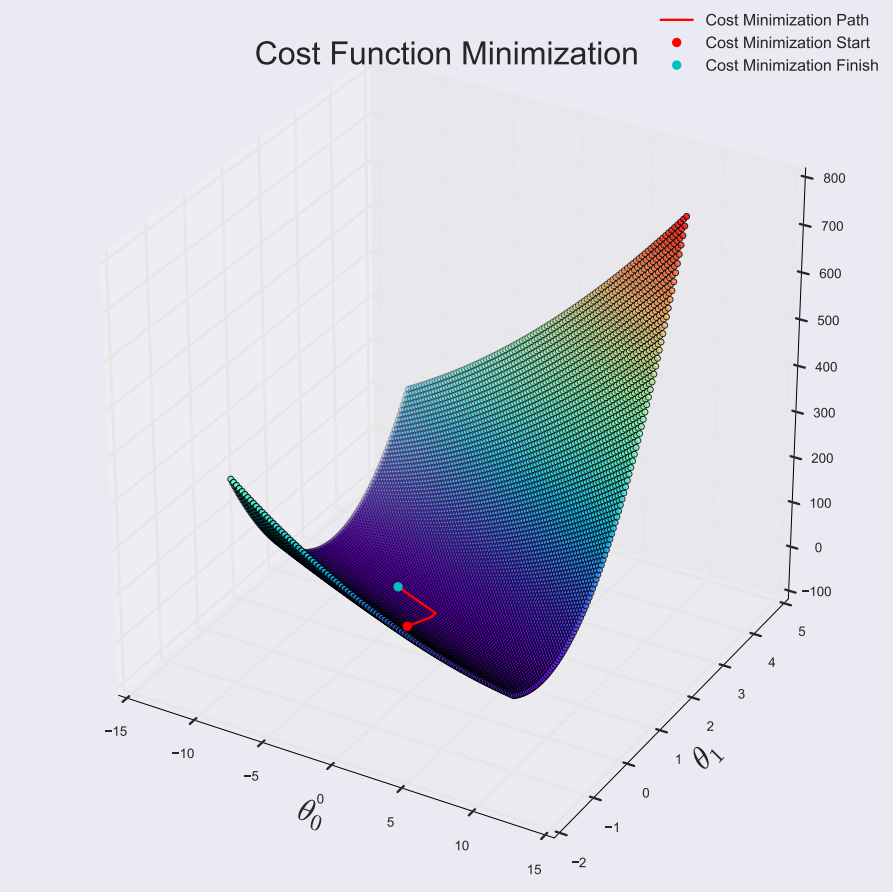


Now, let's plot the cost minimization on the surface of $J(\theta)$.

```
from mpl_toolkits.mplot3d import axes3d, Axes3D
from matplotlib import cm

theta_0_hist = [x[0] for x in theta_hist]
theta_1_hist = [x[1] for x in theta_hist]
theta_hist_end = len(theta_0_hist) - 1

fig = plt.figure(figsize=(12, 12))
ax = fig.gca(projection='3d')

theta_0_vals = np.linspace(-10, 10, 100)
theta_1_vals = np.linspace(-1, 4, 100)

theta1, theta2, cost = [], [], []

for t0 in theta_0_vals:
    for t1 in theta_1_vals:
        theta1.append(t0)
        theta2.append(t1)
        theta_array = np.array([[t0], [t1]])
        cost.append(compute_cost(X, y, theta_array))

scat = ax.scatter(theta1, theta2, cost,
                  c=np.abs(cost), cmap=plt.get_cmap('rainbow'))
```

```
plt.plot(theta_0_hist, theta_1_hist, J_hist, 'r',
         label='Cost Minimization Path')
plt.plot(theta_0_hist[0], theta_1_hist[0], J_hist[0], 'ro',
         label='Cost Minimization Start')
plt.plot(theta_0_hist[theta_hist_end],
         theta_1_hist[theta_hist_end],
         J_hist[theta_hist_end], 'co', label='Cost Minimization Finish')

plt.xlabel(r'$\theta_0$', fontsize=24)
plt.ylabel(r'$\theta_1$', fontsize=24)
plt.title(r'Cost Function Minimization', fontsize=24)
plt.legend()
```

2.6 Python Labs: Coding Linear Regression


Cost Function Minimization
Cost Minimization Path
Cost Minimization Start
Cost Minimization Finish
800
700
600
500
400
300
200
100
0
−100
−15
−10
−5
5
10
15
θ_0^0
−2
−1
0
1
2
3
4
5
θ_1

Chapter 3

Logistic Regression and Regularization

Now we turn away from regression to classification problems. Don't be confused by the name 'logistic regression,' it's actually just named for the mathematical function and it's a common approach to classification.

3.1 Binary Classification

In classification problems, instead of our output being continuous, we expect it to fall into discrete classes. We'll start with the simplest case: binary classification. Here, we have our output variable $y \in \{0, 1\}$. Typically, we take 0 as the negative class and 1 as the positive class.

Consider the following example: we have a sample of eight tumors, and we want to determine if they're malignant based on the tumor size. These are plotted in Figure 3.1a. One thing we can do is assume a linear relationship with hypothesis $h_\theta(\vec{x}) = \vec{\theta}^\intercal \vec{x}$. This shown in Figure 3.1b.

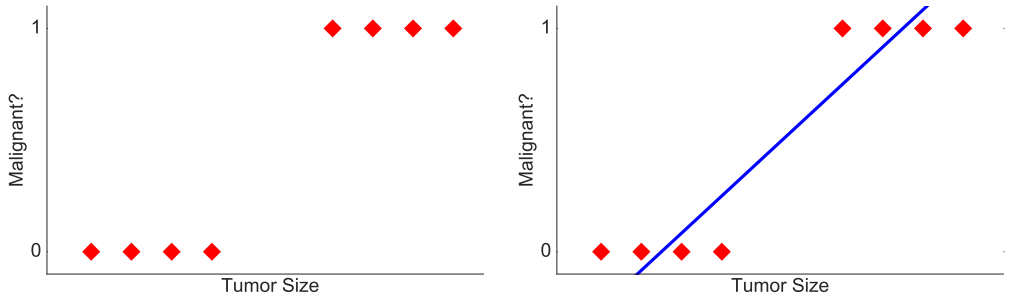
To try and make predictions, we can threshold the output at $h_\theta(x) = 0.5$, and then:

- If $h_\theta(x) \geq 0.5$, then predict $y = 1$
- If $h_\theta(x) < 0.5$, then predict $y = 0$

and you can see this in Figure 3.2.

In this example, it would seem like linear regression is a good classifier. However, what if we add a new data point for a large tumor. Suddenly, our results look like this

and now, we have a malignant tumor being misclassified as benign. Ergo, maybe linear regression isn't the best way to build a binary classifier.



(a) Sample tumor data.

(b) Plot of tumors by size with linear regression line $y = \frac{x-1}{6} - 0.25$.

Figure 3.1: Plots of tumors by size.

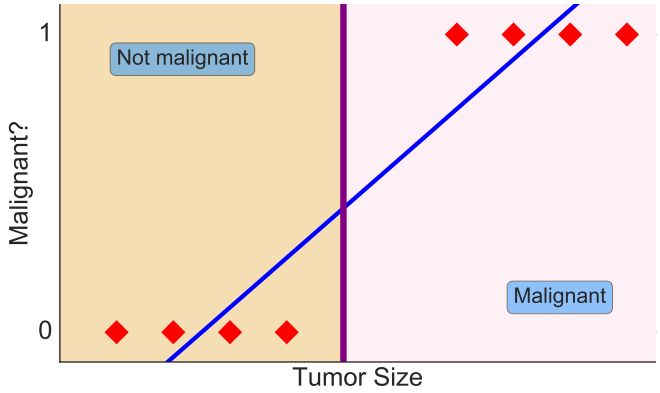


Figure 3.2: Linear regression plotted with classification regions.

3.2 Hypothesis Representation

In linear regression, our hypothesis was $h_\theta(\vec{x}) = \vec{\theta}^\intercal \vec{x}$ For logistic regression, we want our hypothesis to satisfy $0 \leq h_\theta(x) \leq 1$. To do this, we use the sigmoid function.[1] To make this work, we modify our hypothesis to be

$$h_\theta(x) = g\left(\vec{\theta}^\intercal \vec{x}\right) \tag{3.1}$$

where the function $g(z)$ is defined as

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

[1] This is also called the logistic function, and is the namesake for logistic regression.

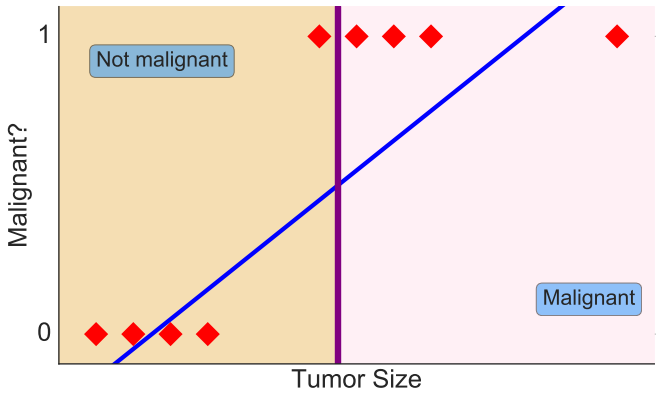


Figure 3.3: Linear regression plotted with classification regions after a new data point is added. Notice how one of the malignant tumors is now being misclassified as benign.

Thus, to get the hypothesis function using the sigmoid function, just set $z = \vec{\theta}^{\intercal}\vec{x}$.

The sigmoid function, shown in Figure 3.4, maps any real number onto the interval $(0, 1)$. This makes it immensely useful for transforming an arbitrary function for use with classification.

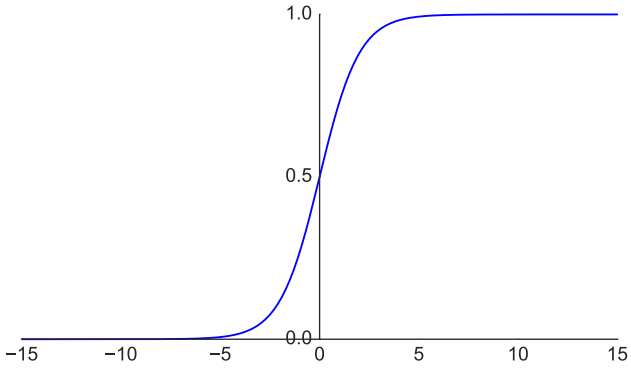


Figure 3.4: Sample plot of the sigmoid function.

3.2.1 Interpretation of the Logistic Hypothesis Function

When examining the hypothesis function output for logistic regression, we interpret $h_\theta(x)$ is the estimated probability that $y = 1$ on an input example x. For example, let's

revisit the tumor size question from above. We have

$$\vec{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ \text{tumor size} \end{bmatrix}$$

If our hypothesis $h_\theta(x) = 0.7$, then we can tell the patient that there is a 70% chance of the tumor being malignant.

Slightly more formally, we interpret $h_\theta(x)$ as:[2]

$$h_\theta(x) = P(y = 1|x;\theta) \tag{3.3}$$

Thus, by the rules of probability:

$$P(y = 0|x;\theta) + P(y = 1|x;\theta) = 1 \tag{3.4}$$
$$P(y = 0|x;\theta) = 1 - P(y = 1|x;\theta) \tag{3.5}$$

3.2.2 Fitting Logistic Regression to a Binary Classifier

Now, we need to fit our hypothesis function into a binary classfier: 0 or 1. Using our probabilistic interpretation of the logistic hypothesis function, we can make the following supposition:

$$y = 1 \text{ given that } h_\theta(x) \geq 0.5 \tag{3.6}$$
$$y = 0 \text{ given that } h_\theta(x) < 0.5 \tag{3.7}$$

Recall the plot of the sigmoid function in Figure 3.4. We see that $g(z) \geq 0.5$ when $z \geq 0$. In our case, if we're setting $z = \vec{\theta}^\intercal \vec{x}$, then we have:

$$h_\theta(x) = g\left(\vec{\theta}^\intercal \vec{x}\right) \geq 0.5 \quad \text{when} \quad \vec{\theta}^\intercal \vec{x} \geq 0 \tag{3.8}$$

From this, we can now state

$$\vec{\theta}^\intercal \vec{x} \geq 0 \implies y = 1 \tag{3.9}$$
$$\vec{\theta}^\intercal \vec{x} < 0 \implies y = 0 \tag{3.10}$$

When utilizing the sigmoid function, keep the following in mind:

- When $z = 0$, then $e^0 = 1$ so $g(x) = \frac{1}{2}$
- When z goes to ∞, we have $e^{-\infty} \to 0$, and this implies $g(x) = 1$
- As $z \to -\infty$, $e^\infty \to \infty \implies g(z) = 0$

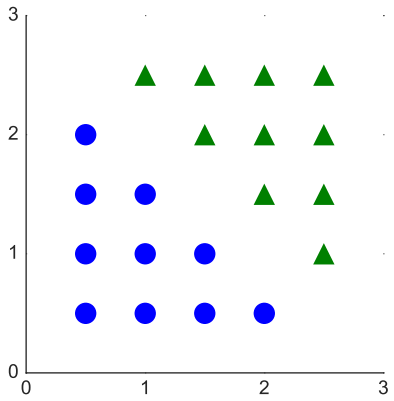


Figure 3.5: Sample data.

3.3 Decision Boundary

Consider the data plotted in Figure 3.5. Suppose our hypothesis is given by

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

We haven't yet discussed how to fit the parameters of this model (that's coming up next), but suppose we choose the following values for the parameters

$$\vec{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} -3 \\ 1 \\ 1 \end{bmatrix}$$

Given this choice of parameters, let's figure out where $y = 1$ and where $y = 0$. From §3.2.2, recall that we predict $y = 1$ when $\vec{\theta}^\intercal \vec{x} \geq 0$, so here, we predict $y = 1$ if $-3 + x_1 + x_2 \geq 0$. If we solve this for $x_1 + x_2$ we get

$$x_1 + x_2 \geq 3 \implies y = 1$$

If we change this to a pure equality, $x_1 + x_2 = 3$, we have the equation of a straight line (shown on Figure 3.6). The line drawn, is called the **decision boundary**. The decision boundary is the line created by the hypothesis function that separates the area where we classify $y = 0$ and where $y = 1$.

To be clean, the decision boundary is a property of the hypothesis function, and not a property of the dataset. We fit the parameters of the hypothesis based on the training data, but once those parameters are set, the decision boundary is a property solely of the hypothesis function.

[2]This is read as "the probability that $y = 1$, given x, parameterized by θ.

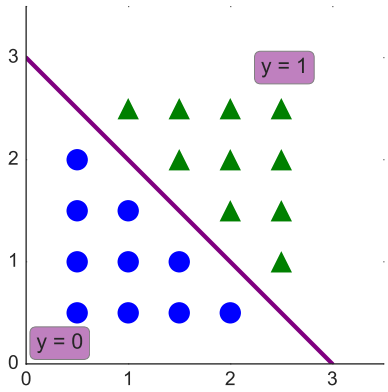


Figure 3.6: Some sample binary data with a plotted decision boundary. Here, the blue circles represent $y = 0$, the green triangles $y = 1$ and the purple line is the decision boundary.

Now, suppose we have data as shown below in Figure 3.7a. It's fairly obvious that no straight line decision boundary will work for this data. Again, we don't know how to fit the parameters for this model yet, but say our hypothesis function looks like this

$$h_\theta(x) = g\left(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2\right)$$

Imagine we fit the parameters appropriately, and we get

$$\vec{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

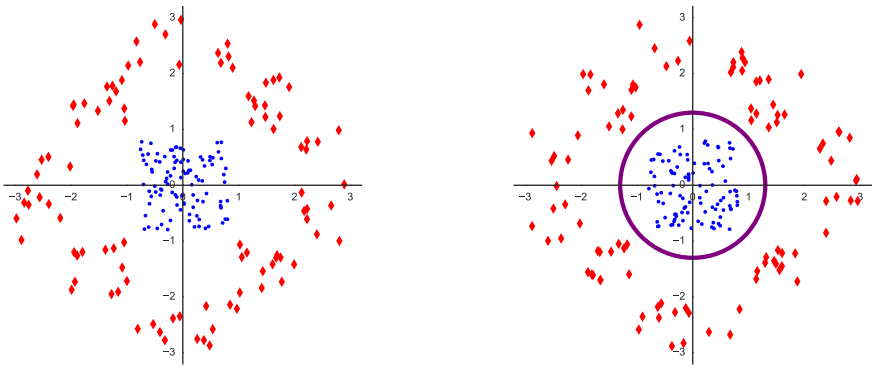
Then, our hypothesis predicts that $y = 1$ when $x_1^2 + x_2^2 \geq 1$. This is the equation for a circle of radius 1, centered at the origin (see Figure 3.7b). In this case, we predict $y = 1$ everywhere outside the purple circle, and $y = 0$ everywhere inside the circle.

With even higher order polynomials, we can get even more complicated decision boundaries.

3.4 Cost Function

Imagine we have a training set of data with m examples

$$\left\{\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \cdots, \left(x^{(m)}, y^{(m)}\right)\right\}$$



(a) This is a sample of data with no clear linear decision boundary.

(b) By altering our hypothesis function to include polynomial terms, we can have a non-linear decision boundary.

Figure 3.7: Data that can't be fit with a linear decision line.

and n features, represented by an $n+1$-dimensional feature vector

$$x \in \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

where $x_0 = 1$ and our output $y \in \{0, 1\}$. Our hypothesis is given by

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^\intercal x}} \tag{3.11}$$

How do we choose the parameters for this model? For linear regression, we had the following cost function (adjusted slightly, we moved the $\frac{1}{2}$ to the inside of the summation)

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2 \tag{3.12}$$

but we're going to change how we write this function a little bit. Instead, we'll write

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost} \left(h_\theta \left(x^{(i)} \right), y^{(i)} \right) \tag{3.13}$$

where we'll define the cost to be

$$\text{Cost} \left(h_\theta \left(x^{(i)} \right), y^{(i)} \right) = \frac{1}{2} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2 \tag{3.14}$$

This allows us to see more clearly that the cost function is really the sum over the cost term. To simplify even further, we'll remove the superscripts (i)

$$\text{Cost}\left(h_\theta\left(x\right), y\right) = \frac{1}{2}\left(h_\theta\left(x\right) - y\right)^2 \tag{3.15}$$

If we try to minimize this function called Cost, it turns out to be a non-convex function. That means that the may be several local minima, which would prevent our gradient descent algorithm from working well. You can see a sample non-convex function in Figure 3.8. What we want instead, is a convex function (like a parabola) that only has a single minimum that is the global minimum. The sigmoid function is a non-linear signal function, so $J\left(\theta\right)$ ends up being non-convex.

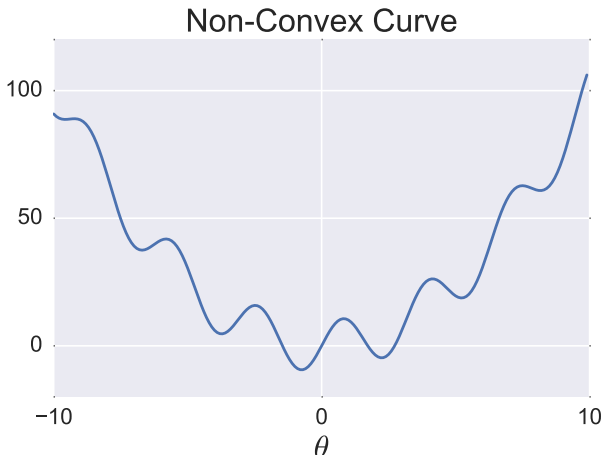


Figure 3.8: A non-convex curve. Notice all of the local minima.

We need to define a new (convex) function that will allow us to determine the parameters in our hypothesis. For logistic regression, we use the following function

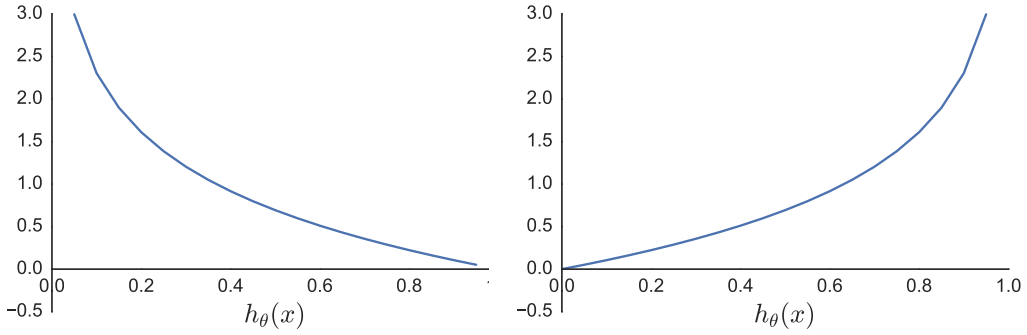
$$\text{Cost}\left(h_\theta\left(x\right), y\right) = \begin{cases} -\log\left(h_\theta\left(x\right)\right) & \text{if } y = 1 \\ -\log\left(1 - h_\theta\left(x\right)\right) & \text{if } y = 0 \end{cases} \tag{3.16}$$

We plot this function below in Figure 3.9.

The shape of the curve comes from standard plot of $\log\left(x\right)$, and we just use a negative to flip it upside-down. This function, has some very desirable properties for us right now.

- If $y = 1$, then $h_\theta\left(x\right) = 1$ and the cost is zero. However, as $h_\theta\left(x\right) \to 0$ then cost $\to \infty$. This captures the intuition that if $h_\theta\left(x\right) = 0$, but $y = 1$, we'll penalize the learning algorithm by a very large cost.

- For $y = 0$, this is reversed. If we have $y = 0$ and $h_\theta\left(x\right) = 0$, then the cost is 0. If $y = 0$, the cost grows very large as $h_\theta\left(x\right)$ increases towards 1.



(a) $y = 1$.

(b) $y = 0$

Figure 3.9: The piecewise function used for the logistic regression cost function.

3.4.1 Simplified Cost Function

Recall our cost function

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right) \tag{3.17}$$

where the cost is

$$\text{Cost}\left(h_\theta(x), y\right) = \begin{cases} -\log\left(h_\theta(x)\right) & \text{if } y = 1 \\ -\log\left(1 - h_\theta(x)\right) & \text{if } y = 0 \end{cases} \tag{3.18}$$

and $y \in \{0, 1\}$. Since y is always either 0 or 1, we can take advantage of this to write a simplified version of our cost function:

$$\text{Cost}\left(h_\theta(x), y\right) = -y \log\left(h_\theta(x)\right) - (1 - y)\log\left(1 - h_\theta(x)\right) \tag{3.19}$$

For either value of y, one of the terms will be multiplied by zero and disappear. So now, our cost function is[3]

$$\begin{aligned} J(\theta) &= \frac{1}{m}\sum_{i=1}^{m} \text{Cost}\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right) \\ &= -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right] \end{aligned} \tag{3.20}$$

We want to minimize the cost function$J(\theta)$ to fit the parameters θ, so we can make our predictions using the hypothesis function. Again, we determine θ by calculating

$$\min_\theta J(\theta)$$

[3]Just as an FYI, this cost function is derived statistically using maximum likelihood estimation

and predict using

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

using the calculated parameters. Now, we just need to determine how to minimize $J(\theta)$.

3.4.2 Gradient Descent for Logistic Regression

We again return to gradient descent, of the form

Repeat until convergence:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \tag{3.21}$$

where we simultaneously update all θ_j. If we calculate the partial derivative $\frac{\partial}{\partial \theta_j} J(\theta)$, we find

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)} \tag{3.22}$$

Plugging this back into the formula for gradient descent, we get

Repeat until convergence:

$$\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)} \tag{3.23}$$

But wait! This looks exactly like the formula for linear regression! Is this actually a different algorithm? Yes! It is! The difference here is that the hypothesis function $h_\theta(x)$ is a different function.

Remember that for gradient descent, we often need to apply feature scaling to make the algorithm run faster.

3.5 Vectorized Equations

In equation 3.20, we derived a simplified version of the cost function. We can do the same thing with a vectorized implementation. First, we state the vectorized hypothesis function as

$$h = g\left(X\vec{\theta} \right) \tag{3.24}$$

and then we can write the simplified cost function as

$$J(\theta) = \frac{1}{m} \cdot \left(-\vec{y}^\intercal \log(h) - (1 - \vec{y})^\intercal \log(1 - h) \right) \tag{3.25}$$

Finally, we write out the vectorized form for gradient descent

$$\vec{\theta} := \vec{\theta} - \frac{\alpha}{m} X^\intercal \left(g\left(X\vec{\theta} \right) - \vec{y} \right) \tag{3.26}$$

3.6 Advanced Optimization

There are other, more sophisticated algorithms that are able to more quickly optimize θ. These algorithms are a little more complicated, so you shouldn't try to write them yourself unless you're an expert in numerical computing.

In particular, there are three algorithms that we'll mention:

- Conjugate gradient algorithm
- Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm
- L-BFGS-B

In Python, these algorithms (and several others) are available in the `scipy.optimize` package. The algorithm is chosen using the `method='X'` flag, where X can be CG, BFGS, or L-BFGS-B, respectively, to use any of the above algorithms.

```
from scipy.optimize import minimize
```

We'll go through an example or two in the coding section.

3.7 Multiclass Classification: One-vs-All

What is a multiclass classification problem? Here are some examples:

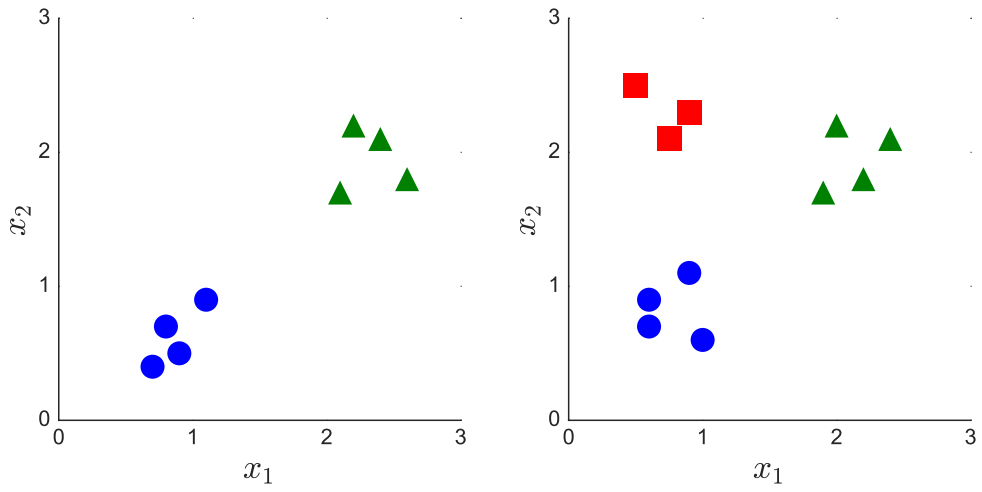
- You want to build an algorithm to automatically tag your email with different categories: work, friends, family, and hobby. Here, you have a classification problem for four classes.
- For a medical visit, you might want to classify patients into not ill, having a cold, or having the flu.
- To build an algorithm that classifies the weather into sunny, cloudy, rain, and snow.

Previously, for a binary classification problem, our data looked like the data in Figure 3.10a, with multiclass classification, the data looks more like Figure 3.10b.

We know how to perform binary classification, but how to we make this work with more classes? We can use the idea of one-vs-all classification.[4] Here's how it works; let's say we have a training set of three classes (such as in Figure 3.10b). What we can do is turn this into three separate binary classification problems.[5] Start by picking a class, say the blue circles, and make it our positive class; then lump all the other data into the negative class (see Figure 3.11).

[4] This can also be called one-vs-rest classification.

[5] If we have n different classes, our problem splits into $n+1$ different binary classification problems.This is because the vector $\vec{y}$ starts at index 0, so $||\vec{y}|| = n + 1$.



(a) Binary Classification (b) Multiclass Classification

Figure 3.10: Examples of binary and multiclass classification.

We fit a classifier to this, called $h_\theta^{(1)}(x)$. We then do this for the two other classes, and fit them to logistic regression classifiers $h_\theta^{(2)}(x)$ and $h_\theta^{(3)}(x)$. Here, we've fit three classifiers

$$h_\theta^{(i)}(x) = P(y = i|x; 0) \qquad \text{for } i = \{1, 2, 3\} \tag{3.27}$$

that are trying to estimate the probability that y is equal to class i, given x and parameterized by θ. So $h_\theta^{(i)}(x)$ is trying to estimate the probability that a data point is of class i.

For a new input x, to make a prediction, we pick the class i that maximizes

$$\max_i h_\theta^{(i)}(x) \tag{3.28}$$

and this tells us which class to assign the new input to.

3.8 Regularization

For the two machine learning algorithms we've seen so far, they tend to work pretty well. But when applied to specific datasets, they can run into a problem called overfitting, and this can cause them to perform very poorly. We're going to discuss a little more detail about this problem, and then go into ways to ameliorate it and increase our algorithm performance. Let's plot some housing data, and then take a look at three potential regressions.

Figure 3.12a is a linear regression on this data. This is the simplest regression, but looking at the data, it seems pretty clear that this isn't really a good fit. As the size of the house increases, the housing prices seem to plateau after a certain point, whereas the

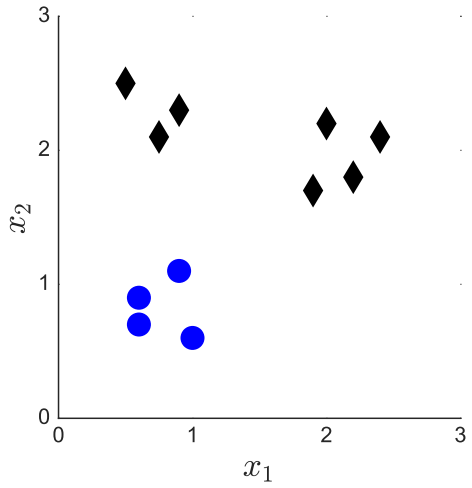


Figure 3.11: One-vs-all classification. We're selecting one class to be our positive class, and the rest all become the negative class.

linear regression line keeps increasing. This is known as **underfitting**, or **high bias**.[6] Both of these roughly mean that the regression just isn't fitting to the training data very well. In Figure 3.12b, we can fit a quadratic function to the data. This seems to work pretty well, and looks like a better fit than the linear regression. Figure 3.12c shows what happens when we **overfit** the line. This is also known as **high variance**. The curve fits the training set very well, but it particular to the specific training set, so it doesn't work well for any other data set.

Let's formally define these terms now:

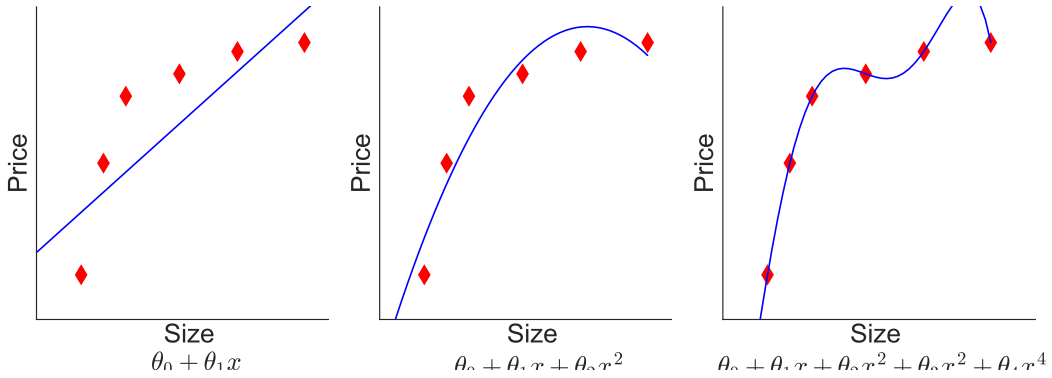
Overfitting If we have too many features, the learned hypothesis may fit the training set very well ($J(\theta) \approx 0$), but fails to generalize to new examples. This is also called high variance.

Underfitting This occurs when the form of our hypothesis function maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features. This is also called high bias.

This apply to both linear and logistic regression.

So how do we prevent our models from overfitting to the data? For the simple examples where we have one or two features, we can plot the data and determine it that way; but when we start working with data sets that have hundreds of features, this no longer works. We have two main options to address overfitting:

[6]The term *bias* is somewhat or a historical or technical term. It carries with it the idea that if fitting a straight line to the data, it's as if the algorithm has a preconception (or bias) that the housing prices should vary linearly with their size, despite evidence in the data that this is not the case. Irrespective of the evidence to the contrary, the algorithm fits the data to a straight line, and it ends up not being a very good fit.



(a) Linear Regression (b) Quadratic Regression (c) Quadratic Regression with Additional Terms

Figure 3.12: Plots of linear and quadratic regressions with varying amounts of terms (features).

1. Reduce the number of features
   - Manually select which features to keep, or
   - Use a model selection algorithm (which we'll see later)
2. Regularization
   - Keep all the features, but reduce the parameters θ_j

Regularization works well when we have a lot of slightly-useful features.

3.8.1 Cost Function

If we suspect overfitting in our hypothesis function, we can reduce the weight of some of the terms in our function by increasing their cost. Say we have a hypothesis of the form

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

To reduce the effect of $\theta_3 x^3$ and $\theta_4 x^4$ without actually getting rid of those features, we instead modify our cost function to something like this

$$\min_\theta \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right)^2 + 1000\,(\theta_3)^2 + 1000\,(\theta_4)^2$$

where 1000 is an arbitrary large number that we chose. This inflates the cost of both θ_3 and θ_4, so to reduce our cost function, we end up reducing the values of θ_3 and θ_4 to near zero. This will greatly reduce the values of $\theta_3 x^3$ and $\theta_4 x^4$ in our hypothesis function. This usually yields a simpler hypothesis that is less prone to overfitting.

In a more general sense, having small values for *all* parameters typically decreases the likelihood that we'll overfit our regression.

To implement regularization, we modify the cost function by adding a regularization term at the end to shrink every parameter, where λ is called the regularization parameter.

$$J_\theta = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{3.29}$$

By convention, these summations start at 1, so we don't regularize θ_0. In this equation, the regularization parameter λ controls the trade-off between two different goals: fitting the training data well (the first term of the equation), and keeping the parameters small (the regularization term).

Conceptually, it can be somewhat difficult to see why keeping the parameters small reduces overfitting, but if you program this yourself, you'll see this effect firsthand. What actually happens is that by using the above cost function with regularization, the resulting output of the hypothesis function is smoothed out to reduce overfitting. However, if λ is chosen to be too large, it may smooth out the function too much and underfit. For example, if $\lambda \sim 10^{10}$ for our housing data problem, then $\theta_j \to 0$ for $j \neq 0$.This leaves us with a regression that looks like $h_\theta(x) = \theta_0$, meaning our hypothesis is just a horizontal line.

3.8.2 Regularized Linear Regression

We've previously looked at two algorithms for linear regression: one based on gradient descent, and one based on the normal equation. Now, we'll take these two algorithms and generalize them to regularized linear regression. Here is our cost function

$$J_\theta = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{3.30}$$

and our goal is to get

$$\min_\theta J(\theta) \tag{3.31}$$

3.8.2.1 Gradient Descent

Our standard gradient descent algorithms looks something like this:

Repeat {

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} \qquad \text{for } j = 0, 1, 2, \ldots, n$$

}

To make this easier, we're just going to write the update for θ_0 separately.

Repeat {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_0^{(i)} \tag{3.32a}$$

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)} \qquad \text{for } j = 1, 2, \ldots, n \tag{3.32b}$$

}

Now, we want to modify these equations to include our new regularization parameter. The θ_0 equation stays the same since we don't regularize θ_0, but equation 3.32b becomes

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_0^{(i)} \tag{3.33a}$$

$$\theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \qquad \text{for } j = 1, 2, \ldots, n \tag{3.33b}$$

If you do the calculus out, you can prove that the large term in brackets in equation 3.33b is the partial derivative of the new $J(\theta)$ with the regularization term.

$$\frac{\partial}{\partial \theta_j} J(\theta)$$

With some manipulation, we can rewrite our update rule as

$$\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m} \right) - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)} \tag{3.34}$$

where in the first term in the equation, the value $1 - \alpha \frac{\lambda}{m}$ will always be less than one. Intuitively, the term that's always less than one will serve to reduce θ_j by a small amount, and the other term is actually exactly the same as our previous gradient descent algorithm before we introduced regularization.

3.8.2.2 The Normal Equation

Gradient descent was just one of the two algorithms we've explored as a solution to linear regression; the second algorithm was based on the normal equation. With the normal equation, we crafted the design matrix X where each row corresponds to a separate training example, and we have an m-dimensional vector $\vec{y}$ that contains our labels.

$$X = \begin{bmatrix} \left(x^{(1)} \right)^\intercal \\ \left(x^{(2)} \right)^\intercal \\ \vdots \\ \left(x^{(m)} \right)^\intercal \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

where we calculated the vector $\vec{\theta}$ as

$$\vec{\theta} = \left(X^\intercal X\right)^{-1} X^\intercal \vec{y}$$

This minimized the unregularized cost function $J(\theta)$. To add in regularization, we add another term inside the parentheses

$$\vec{\theta} = \left(X^\intercal X + \lambda L\right)^{-1} X^\intercal \vec{y} \tag{3.35}$$

where the matrix L is a diagonal matrix[7] given by:

$$L = \begin{bmatrix} 0 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix} \tag{3.36}$$

The matrix L has 0 in it's top left spot, then 1's down the diagonal and zeros everywhere else. It has dimensions $(n+1) \times (n+1)$. Intuitively, this is the identity matrix (sans x_0), multiplied by a single number $\lambda \in \mathbb{R}$.

Recall that if the number of examples $m \leq$ the number of features n, then $X^\intercal X$ is non-invertible.[8] We got around this programatically by using the pseudoinverse. Fortunately, regularization also takes care of this for us as well. As long as $\lambda \in \mathbb{R}_{>0}$ is strictly greater than zero, then $X^\intercal X + \lambda L$ will be strictly invertible.

3.8.3 Regularized Logistic Regression

Logistic regression, just like linear regression, is prone to overfitting if you overfit with higher order polynomials. As we saw in equation 3.20, our logistic regression cost function is given by

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

To morph this into a regularized equation, we need to add a regularization term at the end, resulting in

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2 \tag{3.37}$$

Note that in the second summation, $j \in [1, n]$, so this excludes θ_0.

[7] In linear algebra, a diagonal matrix is a matrix (usually a square matrix) in which the off-diagonal elements are all zero. The main diagonal entries themselves may or may not be zero.

[8] A non-invertible square matrix is also called singular or degenerate.

In a similar fashion to what we did with linear regression, we're going to make the same changes to gradient descent for logistic regression. We start with our standard gradient descent equation (equation 3.21), and separate out the θ_0 term into its own equation. This gives us the same initial result as linear regression.

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_0^{(i)} \tag{3.38}$$

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)} \qquad \text{for } j = 1, 2, \ldots, n \tag{3.39}$$

Then, we again modify the second update rule

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_0^{(i)} \tag{3.40a}$$

$$\theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \qquad \text{for } j = 1, 2, \ldots, n \tag{3.40b}$$

Cosmetically, this is the same formula as for linear regression, but the hypothesis is different, so the equations end up being different. Again, the bracketed term is the new partial derivative of $J(\theta)$ with the regularized cost function.

3.9 Python Labs: Coding Logistic Regression in Python

3.10 Homework

Chapter 4

Neural Networks: Representation

Let's start by discussing the motivation for neural networks. We already have seen and coded two powerful machine learning algorithms, so we do we need another?

4.1 Non-linear Hypotheses

If we have a fairly messy dataset with three terms, x_1, x_2, and x_3, we can classify them using logistic regression, but we'll probably need to introduce polynomial terms to get an accurate classifier. This would give us a hypothesis in the following form:

$$h_\theta(x) = g\left(\theta_0, \theta_1 x_1^2 + \theta_2 x_1 x_2 + \theta_3 x_1 x_3 + \theta_4 x_2^2 + \theta_5 x_2 x_3 + \theta_6 x_3^2\right)$$

Simply by including quadratic terms, we created six features. We can determine this number of features mathematically from combinatorics, and we can model it after sampling with replacement:

$$\text{num. quadratic features } = \binom{n+k-1}{k} = \frac{(n+k-1)!}{k!\,(n-1)!} = \frac{(3+2-1)!}{2!\cdot(3-1)!} = \frac{4!}{4} = 6 \tag{4.1}$$

If we think back to our housing example, and want to perform classification instead of regression using 100 features, that would give us 5050 polynomial terms to include, in addition to the 100 linear terms. We can approximate the growth of the number of new features we get with all quadratic terms with $\mathcal{O}\left(n^2/2\right)$. If we wanted to include cubic terms in our hypothesis too, the features would grow asymptotically as $\mathcal{O}\left(n^3\right)$. Since the number of features grows so rapidly, the number of quadratic and cubic features very quickly becomes impractical.

Consider a collection of 50×50 pixel black-and-white photograph, where we want to determine which photographs are of cars. Then the length of our feature vector is 2500[1], since we have 2500 individual pixels. Each features here represents the brightness of the pixel. Now if we want to include quadratic features, we have approximately $2500^2/2 = 3,125,000$ features.

[1] If we were using RGB values, this would be 7500.

4.2 Neurons and the Brain

Neural networks originated when people thought to build algorithms to mimic how the human brain learns. They were popular in the 1980s, but somewhat fell out of use in the 90s; however, there has been a pretty big surge in neural network use lately due to the massive advances in computer hardware and processing speed.

While it might seem like the human brain learns different things in different brain regions, there is a hypothesis that the brain only uses a single learning algorithm for all its different functions. This was motivated by an experiment where scientists rewired the optical nerve to the auditory cortex in animals, and the auditory cortex actually learned to see. This was repeated with other areas of the brain as well. The principle behind this is called neuroplasticity.

4.3 Model Representation I

Neural networks were developed to simulate neurons and networks of neurons in the brain. Very simplistically, the neuron takes inputs via the dendrites as electrical inputs (called spikes) and then channels the output via the axon.

For an artificial neural network, we'll use a very simple model of what a neuron does, we'll model a neuron as a logistic unit. In our model, our inputs are the input features x_1, x_2, etc. and the output is the result of our hypothesis function. Just like with logistic regression, we have two parameter vectors $\vec{x}$ and $\vec{\theta}$

$$\vec{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \vec{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

where $x_0 = 1$ is the bias term. When representing neural networks, we always have the θ_0 bias term, but we sometimes omit it for notational convenience. Additionally, when representing neural networks, we'll typically use 3 features, though in reality the number of features is a parameter of the problem.

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow [\ ] \rightarrow h_\theta(x)$$

We use the same logistic function in the hypothesis as logistic regression. However, in neural networks, it is often called the sigmoid function, or the sigmoid/logistic activation function.

$$\frac{1}{1 + e^{-\theta^\intercal x}} \tag{4.2}$$

We sometimes call the θ parameters **weights** for neural networks, as is traditional in the neural network literature, so now we might refer to $\vec{\theta}$ as either parameters or weights.

Now, let's look at a very simple model of a neural network. The first layer, $\vec{x}$, is called the **input layer**. The output of the hypothesis function is called the **output layer**, which gives the our final value for the hypothesis. In between the input layer and the final layer, there are one or more hidden layers. The hidden layer nodes are labeled $a_1^{(2)}$,

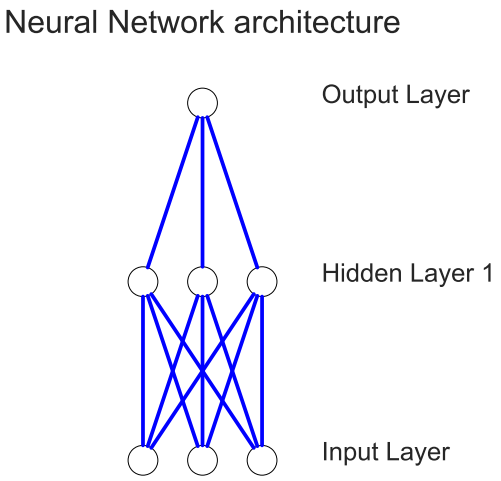


Figure 4.1: A sample artificial neural network, with three inputs and one hidden layer.

$a_2^{(2)}$, etc. and called activation units, where $a_i^{(j)}$ is the activation of unit i in layer j. The matrix $\Theta^{(j)}$ is the matrix of weights controlling the function mapping from layer j to layer $j+1$. Mathematically, we might represent this as

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} \rightarrow h_\Theta(x)$$

Now, let's break out the computations that are represented by this diagram

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3\right) \tag{4.3a}$$

$$a_2^{(2)} = g\left(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3\right) \tag{4.3b}$$

$$a_3^{(2)} = g\left(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3\right) \tag{4.3c}$$

$$h_\Theta(x) = a_1^{(3)} = g\left(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}\right) \tag{4.3d}$$

This is saying that we compute our activation nodes using a 3×4 matrix or parameters. We apply each row of parameters to our inputs to obtain the value for one activation

node. Our hypothesis output is the sigmoid function applied to the sum of the values from the activation nodes, which have been multiplied by yet another parameter matrix, $\Theta^{(2)}$, containing the weights for our second layer of nodes.

More generally, the dimension of the matrix of weights $\Theta^{(j)}$ is given by the following: if a network has s_j units in layer j, and s_{j+1} units in later $j+1$, then $\Theta^{(j)}$ will have dimensions

$$\|\Theta^{(j)}\| = (s_{j+1}) \times (s_j + 1) \tag{4.4}$$

The $+1$ for layer j comes from the bias nodes, x_0 and $\Theta_0^{(j)}$, and it's only applied to the input nodes since the output of a layer doesn't include a bias node.

For example, if layer one has 2 input nodes and layer two has 4 activation nodes, then $\Theta^{(1)}$ will be a 4×3 matrix, since $s_j = s_1 = 2$ and $s_{j+1} = s_2 = 4$.

4.4 Model Representation II

Now, we're going to go through the neural network model again, but this time with a vectorized implementation. We begin by defining a new variable $z_k^{(j)}$ that encompasses the parameters inside of our sigmoid function g. As such, we can now rewrite equations 4.3 as

$$\begin{aligned}
a_1^{(2)} &= g\left(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\right) \qquad \Longrightarrow \qquad a_1^{(2)} = g\left(z_1^{(2)}\right) \\
a_2^{(2)} &= g\left(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\right) \qquad \Longrightarrow \qquad a_2^{(2)} = g\left(z_1^{(2)}\right) \\
a_3^{(2)} &= g\left(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\right) \qquad \Longrightarrow \qquad a_3^{(2)} = g\left(z_1^{(2)}\right)
\end{aligned}$$

So the z values are just a weighted linear combination of the input values x_0, x_1, etc. going to a particular neuron. In other words, for layer $j = 2$ and node k,

$$z_k^{(2)} = \Theta_{k,0}^{(1)}x_0 + \Theta_{k,1}^{(1)}x_1 + \cdots + \Theta_{k,n}^{(1)}x_n \tag{4.5}$$

The vector representations of x and $z^{(j)}$ are

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad\qquad z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ z_3^{(j)} \end{bmatrix}$$

From these vectors, we have $z^{(2)} = \Theta^{(1)}x$ and $a^{(2)} = g\left(z^{(2)}\right)$, where $z^{(2)} \in \mathbb{R}^3$ and $a^{(2)} \in \mathbb{R}^3$. We define x to be $a^{(1)}$, which makes sense because x is our input vector and $a^{(1)}$ implies that we're looking at our first layer, which is the input layer. Then, we can write the general definition for z as

$$z^{(j)} = \Theta^{(j-1)}a^{(j-1)} \tag{4.6}$$

Here, we are multiplying our matrix $\Theta^{(j-1)}$, with dimensions $s_j \times (n+1)$, by our vector a^{j-1}, with length $(n+1)$. This yields our vector $z^{(j)}$ with length s_j.[2]

From this, we can create a vector of our activation nodes for layer j as

$$a^{(j)} = g\left(z^{(j)}\right) \tag{4.7}$$

where the sigmoid function g is applied element-wise to $z^{(j)}$. Next, to get our hypothesis, we need to add a bias term to the layer $j = 2$, $a_0^{(2)} = 1$. In fact, we can generalize going forward that we will need to add bias terms, and they'll all equal one $a_0^{(j)} = 1$. Now, we have $a^{(2)} \in \mathbb{R}^4$, since we just added the bias term to the previously length-three vector. Now, we can compute

$$z^{(3)} = \Theta^{(2)} a^{(2)}$$

and

$$h_\Theta(x) = a^{(3)} = g\left(z^{(3)}\right)$$

This process for computing $h_\Theta(x)$ is called **forward propagation**, because we start off with the activations of the input units, then forward propagate that to compute the activations of the hidden layer, then again forward propagate that to compute the activations of the output layer.

Let's step back for a minute. What we're doing here is very similar to logistic regression, though it might not seem like it. Previously, we have the input feed directly into the logistic regression; now instead, we have the nodes from layer $j = 2$ (the hidden layer) feeding into the logistic regression. However, those nodes $a_k^{(2)}$ are themselves learned from the input data

We've been specifically talking about the neural network architecture described in Figure 4.1, but there can be other neural network architectures too. Consider the example shown in Figure 4.2. Here, we have the same input layer, but there are two hidden layers. The first hidden layer has three hidden units, which are computed as some complex function of the input layer. The second hidden layer can take the first hidden layer's features and compute even more complex features, so the output layer can have very complex features.

4.5 Examples and Intuitions

Let's say we have inputs $x_1, x_2 \in \{0, 1\}$. In this case, our target label $y = x_1$ AND x_2. This is a logical *and*. Can we make a neural network that can recreate this *and* operator? The graph of our function will look something like this

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \left[g\left(z^{(2)}\right)\right] \rightarrow h_\Theta(z)$$

[2] Recall that s_j is the number of activation nodes.

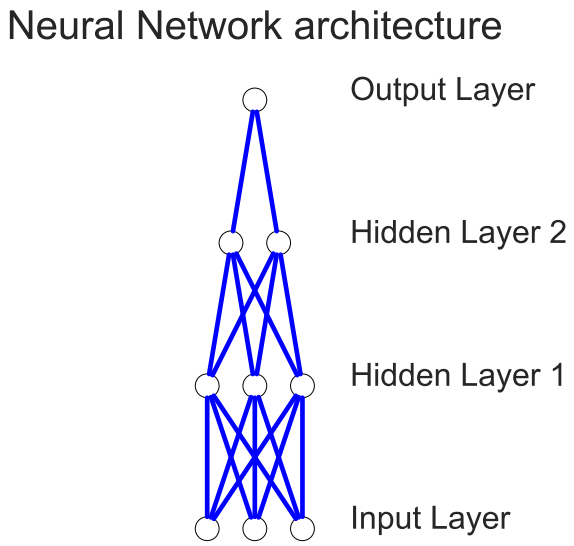


Figure 4.2: A sample artificial neural network, with three inputs and two hidden layer.

where $x_0 = 1$ is our bias variable. For this example, let's define our first $\Theta^{(1)}$ matrix as

$$\Theta^{(1)} = \begin{bmatrix} \Theta^{(1)}_{1,0} & \Theta^{(1)}_{1,1} & \Theta^{(1)}_{1,2} \end{bmatrix} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix}$$

This means our hypothesis is given by

$$h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$$

Let's figure out what our hypothesis evaluates to for different combinations of x_1 and x_2[3]

x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

This is exactly the truth table for the logical *and*, so $h_\Theta(x) \approx x_1$ AND x_2. Using a small neural network, we have just constructed one of the most fundamental operations in computing: the *and* gate.

[3]Keep in mind that the sigmoid function evaluates to about 0.99 for an input of 4.6, and about 0.01 for an input value of -4.6.

4.5.1 Building Logical Gates Using Neural Networks

We are also able to build neural networks to simulate all other logical gates. Let's start with a super-simple example. If we have a single input variable x_1, let's use a neural network to build the logical *not* gate. We could do this with

$$\Theta^{(1)} = \begin{bmatrix} \Theta_{1,0}^{(1)} & \Theta_{1,1}^{(1)} \end{bmatrix} = \begin{bmatrix} 10 & -20 \end{bmatrix}$$

giving us a hypothesis of

$$h_\Theta(x) = g(10 - 20x)$$

If we fill out the table of values for this, we get

x_1	$h_\Theta(x)$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

As a reminder, here is a truth table for some additional logic gates.

Input		**Output**					
A	B	A and B	A or B	A nand B	A nor B	A xor B	A xnor B
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1

We can similarly construct Θ for the *or* gate as

$$\Theta^{(1)} = \begin{bmatrix} \Theta_{1,0}^{(1)} & \Theta_{1,1}^{(1)} & \Theta_{1,2}^{(1)} \end{bmatrix} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix}$$

and Θ for the *nor* gate as

$$\Theta^{(1)} = \begin{bmatrix} \Theta_{1,0}^{(1)} & \Theta_{1,1}^{(1)} & \Theta_{1,2}^{(1)} \end{bmatrix} = \begin{bmatrix} 10 & -20 & -20 \end{bmatrix}$$

4.5.2 Logical XNOR Gate

Having defined the *not*, *and*, *or*, and *nor* gates, let's try and build a logical *xnor* gate. We'll start by building a hidden layer with two nodes, one built with the *and* gate and the other with the *nor* gate. Using

$$\Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \\ 10 & -20 & -20 \end{bmatrix}$$

we can build $a_1^{(2)}$ from the *and* gate and build $a_2^{(2)}$ from the *or* gate. This gives us the following

x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\Theta(x)$
0	0	0	1	
0	1	0	0	
1	0	0	0	
1	1	1	0	

Now, to finish our *xnor* gate, we can use the *or* gate between our two existing nodes on the second layer.

$$\Theta^{(2)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix}$$

Writing this out formally, we find

$$a^{(2)} = g\left(\Theta^{(1)}x\right)$$
$$h_\Theta(x) = a^{(3)} = g\left(\Theta^{(2)}a^{(2)}\right)$$

Filling in the rest of our table, we find we've built the *xnor* gate!

x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\Theta(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

4.6 Multiclass Classification

Similar to logistic regression, we can do multiclass classification with neural networks, and the way we do it is essentially an extension of the one-vs-all method. Let's say we have a computer vision example, where we're trying to classify an image into a pedestrian, a car, a motorcycle, or a truck. We would do this buy building a neural network with an output of four numbers, meaning the output h_Θ will actually be a 4-vector. In our example, when we have a pedestrian or car, we'd want our output to be

$$\text{(pedestrian) } h_\Theta(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{(car)} h_\Theta(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Our training set will look similar

$$\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \left(x^{(3)}, y^{(3)}\right), \cdots, \left(x^{(m)}, y^{(m)}\right)$$

but instead of representing $y \in \{1, 2, 3, 4\}$, we'll represent $y^{(i)}$ as one of the following:

$$y^{(i)} \in \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right\}$$

where both $h_\Theta(x)$ and $\vec{y}$ will be in $\mathbb{R}^4$.

Let's write this out a bit. Sticking with the image classification problem with four output classes, our artificial neural network can be represented by

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(3)} \\ a_1^{(3)} \\ a_2^{(3)} \\ a_3^{(3)} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} h_\Theta(x)_1 \\ h_\Theta(x)_2 \\ h_\Theta(x)_3 \\ h_\Theta(x)_1 \end{bmatrix}$$

The final hidden layer of nodes, when multiplied by its Θ matrix, will result in another vector, on which we can apply the sigmoid function g to get a vector of hypothesis values, which will asymptotically be similar to one of the four $y^{(i)}$ vectors.

Appendix A

Notation

$\hat{}$	predicted output
$\hat{y}$	predicted output of the variable y

$\mathbb{Z}$	the set of integers; zahlen is the German word for numbers
$\bigoplus$	the earth
$\odot$	the sun
c	speed of light
G	Newton′s gravitational constant
C	circumference
S	distance

Appendix B

Linear Algebra Review