Machine Learning

Coursera Notes

by

Noam Levi

Contents

1 Dictionary

- Supervied Learning - The algorithm predicts the right answer (i.e. the output) based on the assumption of there being a relationship in the training set between the input and the output. Correct estimation of data is given ("right answer" is known).
- Regression - Predicts **continuous-valued** output (price, temperature, etc.).
- Classification - Predicts **discrete-valued** output (binary, multiclass).
- Feature/Attribute - Variables by which it is possible to analyze data.
- Unsupervised Learning - Interpretation or correct classification of data is not given, there are no known "correct answers", allows the algorithm to find structure in the data by itself. Sometimes known as "clustering".

2 Supervised Learning

2.1 Univariate Linear Regression - Week 1

Notation and Definitions:

- Training set - Data with "correct" or desired output, on which one trains the ML algorithm.
- m = number of training examples.
- x = "input" variable/feature.
- y = "output" variable/target.
- (x, y) = Training vector/data set example.
- $(x^{(i)}, y^{(i)}), i = 1, ..., m = {}^{i}$th row of data.
- The learning algorithm takes a training set and outputs h.
- Hypothesis function $\hat{y}$ represented as $h_\theta(x) = \theta_0 + \theta_1 x$,
- θ_i: parameters
- θ_0: y-intersect
- $\theta_1 x$: gradient
- α: learning rate (greater = more aggressive)

2.1.1 Basics of Linear Regression

The basic ML algorithm performs the following scheme:
Training set $\rightarrow$ Learning Algorithm $\rightarrow$ Hypothesis Function.

The Hypothesis function seeks to map new input variables/features to “correct” output variables, therefore predicting/estimating the correct behaviour of the data $H : x \to y$.

The first ML algorithm will be that of Linear Regression. Here, the hypothesis function will take the form

$$h_\theta(x) = \theta_0 + \theta_1 x, \tag{1}$$

where we have only one feature x, and the θ's are constants known as parameters. This hypothesis is used for univariate (one variable) linear regression.

How do we choose the θ parameters? An optimization/fitting problem. We must define an error weight/Cost function, such that a price is paid when the estimation is dissimilar to the measurements.

Problem: Minimize θ_0, θ_1 such that the error between prediction and actuall data is small.

Standard error - sum of squared distances:

$$error = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \equiv J(\theta_0, \theta_1). \tag{2}$$

We would like to have the optimal estimator/hypothesis. In order to achieve our goal, we must minimize our cost function $J(\theta)$ using some algorithm. This algorithm should perform the following steps:

- Start with some initial valued (θ_0, θ_1).
- Keep changing (θ_0, θ_1) to reduce $J(\theta)$ until reaching a global minimum.

Note that theoretically we should be concerned with choosing our initial conditions properly, otherwise we may find a local minimum and not the global one (if there exist more than one minimum). However, in this case our cost function is convex, ensuring we only have one minimum and therefore the initial conditions are irrelevant.

2.1.2 Gradient Descent Algorithm

One iterative algorithm which fulfills the requirements for minimizing $J(\theta)$ is called Gradient Descent. It is used extremely frequently in ML optimization problems. The algorithm is as follows:

Repeat until convergence

$$\{ \quad \theta_j := \theta_j - \alpha \tfrac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad \},$$

where $j = 0, 1$, and α is a constant which determines the step size at each iteration, and is known as the **learning rate**.

This implementation of this algorithm **simultaneously** updates all of the parameters. An implementation which uses all the training examples in each step is called "batch" gradient descent. In pseudo code:

$$\begin{aligned} temp0 &:= \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1); \\ temp1 &:= \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1); \\ \theta_0 &:= temp0; \\ \theta_1 &:= temp1; \end{aligned}$$

Gradient descent aims to find the quickest path to the local optimum.

Translates to:

j=0 $\quad \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$

j=1 $\quad \theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} ((h_\theta(x^{(i)}) - y^{(i)}) x^{(i)})$

The derivative refers to the direction taken by the gradient descent (i.e. positive or negative slope direction).

Need to simultaneously update θ_0, θ_1..., θ_n by assigning the computed gradiant descents for j=0,j=1...,j=n to each variable at the same time. If θ_0 is updated before θ_1, the second computation will be using the newly assigned value of θ_0.

- If α is too small, convergence will be very slow.
- if α is too large, may always overshoot and never converge to the minimum.

As we approach the minimum, gradient descent will decrease the step size since $\partial_\theta J \to 0$, so there's no need to change the value of α to ensure convergence.

2.2 Multivariate Linear Regression - Week 2

In this scenario, there is more than one feature (input variable), but still only one target feature (output). $n =$ number of features.
$x_1, ...x_n$ denote variables.
$x^{(i)}$: features of i^{th} training example, shown as vector (e.g. the i^{th} row).
$x_j^{(i)}$: value of feature j in i^{th} training example. n: number of features x.
m: training examples.
y: ouptut variable.
The hypothesis is:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n. \tag{3}$$

We can define the following vectors

$$x = \begin{bmatrix} x_0 \\ x_1 \\ ... \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1}, \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ ... \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}. \tag{4}$$

For convenience, let $x_0 = 1$. So we may write in vector notation

$$h_\theta^{(x)} = \theta_0 x_0 + ...\theta_n x_n = \theta^T x, \tag{5}$$

where $\theta^T \in \mathbb{R}^{1\times(n+1)}$ matrix

We can think about θ_0 as the basic price of a house, θ_1 as the price per square meter, θ_2 as the price per floor, etc. x_1 will be the number of square meters in the house, x^2 the number of floors, etc. (and x_0 is 1).

2.3 Gradient Descent

Parameters: θ, an n+1-dimensional vector.

Repeat until convergence: $\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$for j := 0...n

Simultaneous update applies.

Feature scaling: variables should be similarly scaled by dividing x_i by the range (or standard deviation), else contours will skew, making it harder to reach global minimum. Get every feature into an approximate $-1 \leq x_i \leq 1$ range.

Mean normalisation: $x_i := \frac{x_i - \mu_i}{s_1}$, where μ is the mean and s_1 the range or standard deviation.

Look at a convergence test plot to see if gradient descent is working correctly. Change the α until the global minimum is reached. $J(\theta)$ should be decreasing on every iteration. Try 3-fold or 10-fold increases in α for instance.

2.4 Features and Polynomial Regression

By combining features, e.g. x_1 is frontage and x_2 is depth, which can be multiplied to create a new feature, x (area), the hypothesis may be easier to calculate.

Polynomial Regression: For a potentially better fit use:

- *quadratic function* $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$
- *cubic function* $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$
- *square root function* $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2\sqrt{x_1}$

where x is a feature such as the size of a house.

2.5 Normal Equation

Normal equation is a method to solve for θ analytically.

Take x_i training sets and place them in an X matrix. Add the y training set to a y matrix. The X matrix should be m x (n+1) and the vector m-dimensional.

i.e. design matrix $X = \begin{bmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix}$

Remember that $x_0 = 1$.

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

$(X^T X)^{-1}$is the inverse of matrix$X^T X$.

In Octave: `pinv(X'*X)*X'*y`

Normal Equation Rules:

1. No need to use feature scaling.
2. No need to choose α.
3. No need to iterate.

However:

- Need to compute $X^T X)^{(-1)}$
- Slow if n is very large (consider using gradient descent if $n >$ 10,000).

Noninvertibility:

Not every matrix $X^T X$ is invertible. These are called noninvertible/ singular/degererate matrices.

Noninvertibility occurs when matrices contain:

- Redundant features (linearly dependent i.e. different units for the same feature, such as size in metres and square feet).
- Too many features (i.e. if $m \leq n$).

3 Classification and Representation

$y \in \{0,1\}$ where 0 = negative, 1 = positive.

Binary and multiclass classification problems exist.

Using linear regression is unreliable as outliers can skew hypothesis and $h_\theta(x)$ can scale beyond 0 and 1.

Logistic Regression: $0 \leq h_\theta(x) \leq 1$

Sigmoid function: $h_\theta(x) = g(\theta^T x),$ where $g(z) = \frac{1}{1+e^{-z}}$ and $z : \mathbb{R}$

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

Interpretation: $h_\theta(x) = 0.7 = 70\%$ chance of positive outcome.

$h_\theta(x) = P(y=1|x;\theta)$ i.e. the probability that y=1, given x parameterised by θ.

$P(y=0|x;\theta) + P(y=1|x;\theta) = 1$

Predicts $y = 1$ when $h_\theta(x) \geq 0.5$, or when $\theta^T x \geq 0$

Decision boundaries can be linear and non-linear. They are defined by the chosen theta.

3.1 Cost Function

Using logistic regression cost function returns a non-convex function: not guaranteed to converge to global minimum.

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

$$\text{Cost}(h_\theta(x^{(i)}, y^{(i)}) = \begin{cases} -\log(h_\theta(x)) \text{ if } y = 1 \\ -\log(1 - h_\theta(x)) \qquad\quad \text{if } y = 0 \end{cases}$$

If $y = 1, h_\theta(x) \to 0$ pays a higher cost.
If $y = 0, h_\theta(x) \to 1$ pays a higher cost.

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

If $y = 1$, the second half of the cost function is ignored.
If $y = 0$, the first half of the cost function is ignored.

This method is derived from maximum likelihood estimation.

3.2 Gradient Descent

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

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

```
Repeat{
    $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$
}    (simultaneously update all $\theta_j$)
```

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

The difference between linear and logistic regression gradient descent is that the definition of $h_\theta^{(x)}$ is now equal to $\frac{1}{1+e^{-\theta^T x}}$.

3.3 Optimisation

Algorithms:

- Gradient descent
- **Conjugate gradient**
- **BFGS**
- **L-BFGS**

Advantages: automatically pick α; faster than gradient descent. Disadvantages: more complex

Octave has a built in library containing implementations of these algorithms. Be careful with third party libraries as they vary in optimisation.

```
% cost function J(theta)
% 5 used as example expected theta
function [jVal, gradient] = costFunction(theta)

jVal = (theta(1) - 5) ^2 + (theta(2) - 5) ^2;     % compute J(theta)
gradient = zeros(2,1);
gradient(1) = 2 * (theta(1) - 5);                 % derivative for
gradient(2) = 2 * (theta(2) - 5);                 % theta_0 and theta_1

% optimisation
options = optimset('GradObj', 'on', 'MaxIter', '100');
initialTheta = zeros(2,1);
[optTheta, functionVal, exitFlag] = ...
        fminunc(@costFunction, initialTheta, options);
```

3.4 Multi-class Classification

$y \in \{0, 1, 2, \ldots, n\}$ or $y \in \{1, 2, 3, \ldots, n\}$

n symbols used for plotting on graph.

One-vs-all classification:

Each y is assigned a class, n, which is referred to as $h_\theta^{(i)}(x)$.

Logistic regression is run n times, with the class i a postive value and the remaining classes a negative value.

$h_\theta^{(i)} = P(y = i|x; \theta) \quad (i = 1, 2, \ldots, n)$

To make a prediction, pick $\max_i(h_\theta^{(i)}(x)$

4 Regularisation

Underfitting: High bias; doesn't fit data well enough.
Overfitting: High variance: fits the data well but not with new examples.

Applies to both linear and logistic regression.

Addressing overfitting:

1. Reduce number of features (manually or by using an algorithm).
2. Regularisation
   - Keep all features but reduce values of parameters θ_j.
   - Works well with lots of features, each contributing to predicting y.

4.1 Cost Function

By awarding small values for parameters $\theta_0, \theta_1, \ldots, \theta_n$ it's possible to get a simpler hypothesis which is less prone to overfitting.

e.g. if x_3, x_4 are very high, θ_3 and θ_4 can be valued at close to 0.

If there's no way of knowing, the cost function can be adjusted using regularisation to reduce all $\theta^{(x)}$ values:

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

$min_\theta J(\theta)$

If λ is set too high, $\theta^{(x)}$ values will be penalised to hard and $h_\theta(x)$ will be represented primarily by θ_0. This will result in an underfitting of the data.

4.2 Linear Regression

Separate the function $min_\theta J(\theta)$ into θ_0 and θ_j because θ_0 is not being penalised.

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

While θ_0 will not be regularised.

Can also be thought of as $\theta_j := \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$

where $1 < \alpha\frac{\lambda}{m} < 1$

4.2.1 Normal Equation

Minimise cost function by using:

$$\theta = (X^TX + \lambda \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix})^{-1}X^Ty$$

The matrix is (n+1)x(n+1), in this case n=2.

4.2.2 Non-invertibility

If $m \leq n$ the normal equation is non-invertible. Using `pinv` will be unreliable.

Regularisation using the equation above for normal equation also takes care of non-invertibility.

4.3 Logistic Regression

Separate θ_0 and θ_j as with linear regression.

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

4.3.1 Advanced Optimisation

```
% cost function passed to fminunc
jVal =          % code to compute J_theta + regularisation
gradient(1) =   % code to compute derivative J_theta
gradient(2) =   % code to compute derivative J_theta + regularisation
gradient(n+1) =  % code to compute derivative J_theta + regularisation
```

5 Neural Networks

5.1 Representation

A neural network is an algorithm that can be used for learning non-linear hypotheses. They are preferred if there are lots of features. Using logistic regression can take thousands/millions of polynomial features to compute if there are many features.

Single neuron equivalent to a linear regression function. A network is comprised of a group of neurons and contains 3 layers: the input layer, the hidden layer and the output layer.

$a_i^{(j)}$ = activation of unit i in layer j.

$\Theta^{(j)}$ = matrix of weights (parameters) controlling function mapping from layer j to layer $j+1$.

If network has s_j units in layer j and s_{j+1} units in layer s_{j+1}, then $\Theta^{(j)}$ will be of dimension $s_{j+1}x(s_j+1)$.

Within the hidden layer, for 3 features there will be 3 activations:

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

$a_2^{(2)} = \dots$

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

The output layer then calculates:

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

5.1.1 Forward propogation: Vectorised implementation

$a_1^{(2)} = g(z_1^{(2)})$

$a_2^{(2)} = \dots$

$a_3^{(2)} = g(z_3^{(2)})$

So: $a^{(2)} = g(z^{(2)})$, where $z^{(2)} = \Theta^{(1)}x$.

a and z are both n-dimensional vectors so the sigmoid is applied element-wise as in logistic regression.

The bias unit also needs to be added, which remains 1. This makes a a $n+1$-dimensional vector.

Output layer: $h_\Theta(x) = a^{(3)} = g(z^{(3)})$ where $z^{(3)} = \Theta^{(2)}a^{(2)}$

Therefore, for any activation calculation:

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

$a^{(j)} = g(z^{(j)})$

And for any output calculation:

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

$h_\Theta^{(x)} = a^{(j+1)} = g(z^{(j+1)})$

5.1.2 Multiple output units: one-vs-all

$h_\Theta(x) \in \mathbb{R}^4$

If there are 4 output units, first outcome $h_\Theta \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ and so on.

So while $y \in \{1, 2, 3, 4\}$ previously, now $y^{(i)}$ one of $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, and so on.

5.2 Learning

5.2.1 Cost Function

- L = total number of layers (also used to denote output layer)
- s_l = number of units (not counting bias) in layer l
- K = number of units in output layer

Classification:
Binary = $y = 0$ or $1, K = 1$
Multi-class = $K \geq 3$

$h_\Theta(x) \in \mathbb{R}^K$

$$J(\Theta) = -\tfrac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\Theta_{ji}^{(l)})^2$$

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

Note:

the double sum simply adds up the logistic regression costs calculated for each cell in the output layer the triple sum simply adds up the squares of all the individual s in the entire network. the i in the triple sum does not refer to training example i

5.2.2 Backpropagation

To minimise $J(\Theta)$ a backpropagation algorithm is used. The term refers to the fact that the output layer is calculated first, proceeding backwards.

$\delta_j^{(l)}$ = error of activation node j in layer $l, (a_j^{(l)})$

For the output layer, calculate $\delta_j^{(L)} = y^{(i)} - a_j^{(L)}$

For the hidden layers, calculate $\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)}. * g'(z^{(l)})$

g' (g prime) $= a^{(l)}. * (1 - a^{(l)})$

Procedure:

For $i = 1$ to m

- set $a^{(1)} = x^{(i)}$
- perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, ..., L$
- Using $y^{(i)}$, compute $\delta(L)$

- Compute $\delta^{(L-1)}, \ldots, \delta^{(2)}$
- $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Then compute the partial derivative (accounting for bias) gradient matrices:

$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)} if j \neq 0$

$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} if = 0$

5.2.3 In Practice

Advanced Optimisation:

```
function [jVal, gradient] = costFunction(theta)
...
optTheta = fminunc(@costFunction, initialTheta, options)

% the above works for logistic regression, however with multiple
% Theta and gradients, they'll need to be unrolled into one vector:

thetaVec = [Theta1(:); Theta2(:); Theta3(:)];
DVec = [D1(:); D2(:); D3(:)];

% to revert, use for e.g. s1=10, s2=10, s3=1:

Theta1 = reshape(thetaVec(1:110), 10, 11);
Theta2 = reshape(thetaVec(111:220), 10, 11);
Theta3 = reshape(thetaVec(221:231), 1, 11);

% where s is the number of units in a layer and n = s * s+1

% thetaVec can then be used to calculate the cost function

function [jval, gradientVec] = costFunction(thetaVec)

% reshape to get individual Theta
% use forward/backwards propagation to compute D1,D2,D3 and J(Theta)
% unroll D1,D2,D3 to get gradientVec
```

Gradient Checking:

Gradient checking is used to identify subtle bugs in neural network algorithms.

$\frac{\delta}{\delta\theta}J(\theta) \approx \frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$

In octave:

```
gradApprox = (J(theta+EPSILON) - J(theta-EPSILON)) / (2*EPSILON)
```

With each partial derivative calculation (θ_i), an epsilon is added or subtracted to/from θ_i:

```
epsilon = 1e-4      % for example - should be small but not too small.
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) = thetaPlus(i) + EPSILON;
  thetaMinus = theta;
  thetaMinus(i) = thetaMinus(i) - EPSILON;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus)) / (2*EPSILON);
end;
```

Then check that `gradApprox` $\approx$ `DVec`. Once it does gradient checking needs to be turned off as it's computationally expensive to run on every iteration of gradient descent or within costFunction.

Random Initialisation:

Initialising theta as a matrix of zeros doesn't work in neural networks as after each update, parameters for each hidden unit are identical.

To perform symmetry breaking, use random initialisation:

$-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$

```
Theta1 = rand(10,11) * (2*initEpsilon) - initEpsilon;
Theat2 = rand(1,11) * (2*initEpsilon) - initEpsilon;
```

5.2.4 Implementation

Network architecture

A reasonable default is to use a single hidden layer. If more than one hidden layer is used, the same number of units should be used for each.

Generally speaking, the more hidden units in a hidden layer the better, however it will be more computationally expensive. It should be at least comparable to the number of input units, however, hidden units can often be 2, 3 or 4 times the size of input units.

Training a Neural Network

1. Randomly initialise weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$
3. Implement code to compute cost function $J(\Theta)$
4. Implement backward propagation to compute partial derivatives $\frac{\delta}{\delta\Theta^{(l)}_{jk}}J(\Theta)$
5. Use gradient checking to compare partial derivatives computed using backwards propagation against numerical estimate of gradient of $J(\Theta)$, then disable gradient checking
6. Use gradient descent or advanced optimization method with backpropagation to try to minimise $J(\Theta)$ as a function of parameters Θ

$J(\Theta)$ is non-convex, therefore can get stuck at a local minimum, however this is should not be a problem as it should still get close to the global minimum.

6 Diagnostics and Applications

6.1 Running Diagnostics

Debugging:

- Get more training examples
- Try larger/smaller set of features
- Try adding polynomial features
- try increasing/decreaing lambda

Evaluate a hypothesis:

Place random 30% of data from the training set into a test set. Learn theta from training data (remaining 70%).

Linear regression:

$J_{test}(\theta) = (1/(2 * m_{test}))$ * sum of errors squared.

Logistic regression:

$J_{test}(\theta) =$ logistic regression of test set.

Misclassification errror:

test error $= 1/m_{test}$ sum of err$(h(x)_{test}, y_{test})$

Choosing degree of polynomial:

Set $d =$ degree of polynomial

Calculate theta1, 2, ... 10 (i.e. find theta using different degrees of polynomials).

The problem is d is set to fit the test set, so it might not fit new examples well.

To address this problem, divide data into 3: training set (60%), cross validation set (20%), test set (20%).

After minimising theta for each order, apply theta to the cross validation error function.The order with the lowest error is chosen.

Problems may be due to bias or variance:

High bias (underfit): $J_{train}(\theta)$ and $J_{cv}(\theta)$ are high.

High variance (overfit): $J_{cv}(\theta) \gg J_{train}(\theta)$.

High lambda may underfit data, low lambda may overfit data.

How to choose regularisation parameter lambda:

Try lambda = 0, 0.01, 0.02, ..., 10 and min J(theta) to theta1, theta2, theta_n respectively. Then apply each theta to CV error function and pick the lowest. Then apply the lowest theta to the test error function.

If a learning algorithm is suffering from high bias, more training data will not help (i.e. won't fit a straight line any better). Conversely more data will help a learning algorithm with high variance.

Learning curves will give a better sense of whether there is a bias or variance problem with the learning algorithm.

Fix high bias: increase number of features, try more polynomial features,decrease lambda.

Fix high variance: increase number of training examples, reduce number of features, increase lambda.

Neural Networks:

Small neural network: fewer parameters, more prone to underfitting but computationally cheaper.

Large neural network: more parameters, more prone to overfitting and computationally more expensive. Using regularisation can address overfitting, so the larger the number of neurons, the better (potentially).

Number of hidden layers: To optimise, choose hidden layers=1, 2, 3 and see which performs best in the cross validation error function.

Same bias/variance problems as shown in learning curves apply to problems in neural networks.

6.2 Applications

Email spam filter: repeatedly used words in spam emails are the features. Within each training example, a word appearing is given 1, not appearing 0. y is whether or not the email is spam (or what type of spam it is).

Recommended approach:

- Implement simple algorith quickly and test on cross-validation data.
- Plot learning curves to decide on how it can be improved.
- Error analysis: manually examine the examples that the algorithm made errors on. See whether there is a systematic trend in the type of examples it's making errors on.

Stemming software (e.g. Porter Stemmer) can be used for natural language processing to treat stems of words as the same, such as discount, discounted. It can make mistakes however, such as universe, university. Therefore numerical evaluation (CV error) should be used to determine the error with and without stemming.

Skewed classes: ratio of positive/negative examples is biased.

Precision = true positives / number of predicted positives (true + false +ve)

Recall = true positives / actual positives (true +ve + false -ve)

It is possible to trade off precision and recall. If the threshold of y=1 is increased (to only predict a postive outcome if very confident) then the result will be higher precision, lower recall (avoid false +ves).

Lower threshold = higher recall, lower precison (avoid false -ves).

The F score is a forumla to determine the best combination of precision and recall between any number of algorithms.

F Score = $2\frac{PR}{P+R}$

There are a number of types of F scores however. Taking the product of P an R means that both values need to be high to attain a high F score.

How much data?

Useful test: given the input features x, can a human expert confidently predict y?

Ask whether a large number of training examples can be obtained.

7 Support Vector Machines

Optimisation objective function:

$$\min_0 C \sum_{i=1}^{m} y^{(i)} \text{cost}_1(\theta^T x^{(i)} + (1 - y^{(i)}) \text{cost}_0(\theta^T x^{(i)} + \frac{1}{2} \sum_{(i=1)}^{n} \theta_j^2$$

The logistic regression optimisation function $A + \lambda B$ is taken and changed to $CA + B$. C is equal to $\frac{1}{\lambda}$.

Hypothesis $h_\theta(x) = 1$ if $\theta^T x \geq 0, 0$ otherwise.

SVMs have a larger decision margin than that of linear/logistic regression.

If C is very large, the decision margin is more likely to take outliers into account (if data is not linearly separable).

7.1 Kernels

Kernels are used to define boundaries of more complex non-linear functions.

Can think of a decision boundary as theta0 + theta1 * f1 + theta2 * f2 ... where f1 = x1.

But as higher order polynomial terms can be more computationally expensive, landmarks are used.

fi becomes the similarity of x and l(i). The similarity function is called the Guassian Kernel.

$f1 = \exp(-\frac{||x - l^{(1)}||^2}{2\sigma^2})$

If $x \approx l(1), f1 \approx 1$.

If x far from l(1), f1 ≈ 0.

Higher sigma squared means a higher f can be obtained more easily.

Choosing landmarks:

$x^{(1)} = l^{(1)}$

$x^{(m)} = l^{(m)}$

$f_0 = 1$
$f_1 = similarity(x, l^{(1)})$
$f_2 = similarity(x, l^{(2)})$
$f_m = similarity(x, l^{(m)})$

Hypothesis: given x, compute features $f \in \mathbb{R}^{m+1}$

Predict y=1 if $(\theta^{(i)})^T f^{(i)} \geq 0$

Using the SVM optimisation objective function, swap $\theta^T x^{(i)}$ for $\theta^T f^{(i)}$.

Choosing SVM Parameters:

Large C: Lower bias, higher variance.
Small C: Higher bias, lower variance.

Large sigma squared: features fi vary more smoothly - higher bias, lower variance.

7.2 Implementing SVM

Use a library such as liblinear and libsvm to solve parameters theta. A choice of parameter C and the kernel (similarity function) is still needed however.

Might use no kernel (a linear kernel) if there are many features but few training examples.

If using a Guassian kernel, then sigma squared will need to be chosen too. This may be used if the number of features is small and the number of examples is large.

```
function f = kernel(x1, x2)       % where x1 is x(i) and x2 is l(j) or x(j
    )

% kernel function provided by lib

return
```

Important to use feature scaling before using the Guassian kernel.

Other kernels include:

- Polynomial kernel
- String kernel
- Chi-square kernel
- Histogram intersection kernel

As with other machine learning algorithms, test on the cross validation sample to determine which kernel to use.

Multi-class classification: Most SVM packages have multi-class functionality. Otherwise use the one-vs.-all method (train K SVMs).

Use logistic regression if n $\geq$ m, n=10,000, m=10-1000.

Use SVM with Guassian kernel if m $>$ n, n=1-1000, m=10-10,000.

If m $\gg$ n, (n=1-1000, m=50,000+) create/add more features then use logistic regression or SVM without a kernel.

A neural network is likely to work well for most of these settings, however it may be slower to train.

8 Unsupervised Learning

For unlabelled data: no y.

Uses: market segmentation, social network analysis, organising computing computing clusters, astronomical data anlaysis.

8.1 Clustering

K-means: n cluster centroids created, where n is number of clusters wanting to be created. Each point in the dataset will be evaluated in relation to how close they are to either centroid and are then assigned to that centroid. After that, a new set of centroids are created using the newly adjusted means of the clusters. These steps are repeated until k-means have converged.

K: number of clusters
k: index of cluster
$x^{(1)} - x^{(m)}$: training set

$x^{(i)} \in \mathbb{R}^n$ so $x_0 = 1$ convention is dropped.

Randomly initialise K cluster centroids $\mu_1, \mu_2, ..., \mu_K$

Repeat for i=1 to m (cluster assignment step),

- $c^{(i)}$:= index(1-K) of cluster centroid closest to $x^{(i)}$ by minimising $||x^{(i)} - \mu_k||$. The value of $c^{(i)}$ is the cluster a point has been assigned to.

Repeat for k=1 to K (move centroid step),

- μ_k := mean of points assigned to cluster k.

If a centroid hasn't been assigned a point, it can be deleted or randomly re-initialised.

Optimisation:

$\mu c^{(i)}$: cluster centroid of cluster to which $x^{(i)}$ has been assigned.

$J(c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K) = \frac{1}{m} \sum ||x(i) - muc(i)||^2$

Cluster assignment: $minJ(c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K)$

The move centroid step is choosing the values of $c^{(i)}$ and μ_k to minimise J.

Initialising K-means:

Random initialisation: should have $K < m$, randomly pick K training examples, set $\mu_1, ..., \mu_K$ equal to these K examples, i.e. $\mu_1 = x^{(i)}, \mu_2 = x^{(j)}$ etc.

K-means can end up at different local optima, so it is run a number of times to find the best local optima.

For i=1 to 100,

- Randomly initialise K-means.
- Run K-means to get $c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K$.
- Compute cost function (distortion) $J(c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K)$

Then pick clustering that gave lowest cost.

Choosing K:

Elbow method: run sequential values of K and plot K against cost function J. The 'elbow' in the curve should be picked. If the elbow is ambiguous then this method isn't the best.

Better method is to evaluate K-means based on a metric for how well it performs for a later/downstream purpose.

9 Dimensionality Reduction

In case two features are the same (e.g. lenth in cm and length in inches, or pilot skill and enjoyment which can reduce to pilot aptitude), data can be compressed to get only one feature. This speeds up the learning algorithm, reduces memory/disk needed to store data and is able to visualise high-dimensional data.

$x^{(i)}$ represented by 2 numbers becomes $z^{(i)}$, a real number which refers to the proximity to the projection (PCA is not linear regression).

Similarly 3-dimensional data can be reduced to 2-dimensional data. z_1 and z_2 represent a the axis of a 2-dimensional plane within which the data fits.

Even 50-dimensional data can be reduced to 10D or even 2D data. There are no ways to visualise more than 3D data however.

Principle Component Analysis: Finds a surface to project data onto so that it can minimise the least squared error. This surface/direction/line is a vector $u^{(1)} \in \mathbb{R}^n$. To reduce from n-dimension to k-dimension, find k vectors $u^{(1)}$ to $u^{(k)}$.

Preprocessing: perform feature scaling/mean normalisation.

Compute covariance matrix: $\Sigma = \frac{1}{m} \sum_{i=1}^{n} (x^{(i)})(x^{(i)})^T$

Vectorised implementation: `Sigma = (1/m) * X' * X;`

Compute eigenvectors of matrix Sigma (singular value decomposition):

```
[U,S,V] = svd(Sigma);
```

Sigma = n x n matrix.
U = n x n matrix.

Take first k parts of U matrix and assign to a Ureduce n x k matrix. The transpose of the Ureduce matrix is the k x n z matrix.

```
Ureduce = U(:,1:k);
z = Ureduce' * x;
```

Reconstruction: To retrieve the original feature vector n, calculate Xapprox = Ureduce * z(1).

Choosing k:

Typically choose k to be the smallest value so that the avg sq projection error / total variation in the data is less than or equal to 0.01, meaning that 99% of variance is retained.

$\frac{\frac{1}{m}\sum_{(i=1)}^{m}||x^{(i)}-x_{\approx}^{(i)}||^2}{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2}$

Try PCA with k=1
Compute Ureduce, $z^{(1)}$ to $z^{(m)}$, $x_{approx}^{(1)}$ to $x_{approx}^{(m)}$
Check variance.

Using `[U,S,V] = svd(Sigma)` and for a given value of k, the variance can be calculated using $1 - \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}}$ i.e. sum Sk / sum Sn.

Using supervised learning, $(x^{(i)}, y^{(i)})$ becomes $(z^{(i)}, y^{(i)})$, which can then be used to find $h_{\theta}^{(z)}$. Mapping $X^{(i)}$ to $z^{(i)}$ should only be done on the training set. That mapping can then be applied to the CV and test sets.

PCA should not be used to prevent overfitting, regularisation should be used instead. Furthermore, not using PCA should be considered first. If the raw data does not achieve the desired result or if the algorithm is particularly slow, then consider using PCA.

10 Anomaly Detection

Density estimation: if $p(x_{test}) < epsilon$, flag as anomaly.

Gaussian (normal) distribution: $x\ N(\mu, \sigma^2)$, $p(x; \mu, \sigma^2)$.

μ = mean

σ^2 = variance, $\frac{1}{m} * \sum(x - \mu)^2$

p(x) = sum of product of each p($x^{(i)}$), assuming gaussian distribution and a given mu(i)/sigma(i).

p(x) = $\pi_{j=i}^{n} p(x_j; \mu_j, \sigma_j^2)$

Π = sum of products

Algorithm:

Choose features xi that might be anomalous

Fit parameters $\mu_1 - \mu_n, \sigma_1^2 - \sigma_n^2$.

$\mu_j = \frac{1}{m} \sum x_j^{(i)}$.

$\sigma_j^2 = \frac{1}{m} \sum (x_j^{(i)} - \mu_j)^2$

Given new example x, compute p(x) using above $\Pi p(x_j)$ algorithm.

Anomaly if p(x) ¡ epsilon.

Assuming labeled data, y=0 if normal, y=1 if anomalous.

If using a cross validation, data set will be much more skewed towards y=0, with a very high accuracy. Instead, evaluating an anomaly detection algorithm can be done through true/false positive/negatives, precision/recall or F-score.

A cross validation set can be used to choose epsilon.

Anomaly detection vs. supervised learning:

AD: very small num positive examples (0-20), large num of negative. If there are many different types of anomalies - hard to learn from positive examples what anomalies look like. E.g. fraud detection, manufacturing, monitoring machines in data center.

SL: large num of positive/negative examples. Enough positive examples to learn what future positive examples will look like. E.g. spam or health classification, weather prediction.

```
hist(x);
hist(x,50);
hist(x.\^0.5,50);
% etc to obtain a good Gaussian %distribution
```

Error analysis for anomaly detection:

Most common problem: p(x) is comparable for normal and anomalous examples.

Try to create a new feature which x1 can be mapped against.

Try to create new features that combine existing features e.g. cpu load (optionally squared to capture high cpu load) / network traffic.

This method can potentially capture the different types of anomalies.

Multivariate Gaussian Distribution: Doesn't model p(x1),p(x2) separately, models p(x) all in one go.

$p(x) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} exp(-\frac{1}{2}(x-\mu)^T\Sigma{-1}(x-\mu))$

mu determines peak of distribution, Sigma the variance. Sigma can also determine the correlation of data.

Parameter fitting: mu = mean, sigma = same as PCA.

1. fit model p(x) by setting mu and Sigma.

2. given a new example x, compute p(x) using Gaussian distribution algorithm.

Then flag anomaly if p(x) < epsilon.

When to use original or multivariate models:

Original: manually create features to capture anomalies where x1,x2 take unusual combinations of values. Computationally cheaper, scales better to large n). Can be used with small m.

Multivariate: automatically caputres correlation between features. Computationally more expensive. Must have m > n, or else Sigma is non-invertible. Sigma also non-invertible if features are redundant.

11 Recommender Systems

These systems learn which features to use as opposed to features being manually selected.

$x^{(i)}$ parameters for movie. Contains vector with $x_0 = 1$ and parameters $x_1 - x_m$ (such as how much of an action or romance film it is).

$\theta^{(i)}$ parameters for user.

$(\theta^{(i)})^T x^{(i)}$ to determine rating for a movie.

$r(i, j) = 1$ if user j has rated movie i (0 otherwise).

$y(i, j) =$ rating by user j for movie i (if defined.

$n_u =$ num users
$n_m =$ num movies
$m^{(j)} =$ num movies rated by user j

To learn $\theta^{(j)}$ use cost and gradient descent similar to linear regression.

Collaborative filtering: difficult to determine value of the content features. Based on movie ratings, make predictions for parameters $x^{(i)}$. Can also continue to estimate theta and x using previously found values.

To estimate theta given x, sum over all j (all movies for user). To estimate x given theta, sum over all i (all users for movie).

1. Initialise x, theta to small random values.

2. Min $J(x^{(1)} - x^{n_m}, theta^1 - theta^{n_m})$ using gradient descent.

3. For a user with parameters theta and a movie with learned features x, predict a star

rating thetaTx

Y = matrix of (i,j) dimensions.

Predicted rating (i,j) = $\theta^{(j)})^T x^{(i)}$.

X = matrix of $(x^{(i)})^T$. Theta = matrix of $(\theta^{(i)})^T$.

Multiply to obtain the predicted ratings (named low rank matrix factorisation).

To find related movies find small $||x^{(i)} - j^{(j)}||$.

Mean normalisation solves problem of unrated movies.

mu vector of average rating for each movie is subtracted from each movie rating to calculate Y matrix, used to learn $\theta^{(j)}, x^{(i)}$.

For user j, on movie i predict: $(\theta^{(i)})^T(x^{(i)}) + \mu$. Therefore the predicted movie rating for a user is the average rating.

12 Large-scale Machine Learning

Plot $J_{cv}(\theta)$ against $J_{train}(\theta)$ to see the variance of the data. More data should decrease the variance.

Can be very expensive to sum over 1,000,000+ data entries, needed just to compute one step of gradient descent.

Stochastic gradient descent can be used to calculate gradient descent on large datasets.

$J_{train}(\theta) = \frac{1}{m}\sum cost(\theta, (x^{(i)}, y^{(i)}))$ where,

$cost(\theta, (x, y)) = \frac{1}{2}(h(\theta(x^{(i)}) - y^{(i)})^2$

First randomly shuffle dataset to speed up convergence. The algorithm will change parameters for each training example, rather than changing the parameters for every training example.

Stochastic gradient descent can also be used for logistic regression and neural networks.

Mini-batch gradient descent: use b examples in each iteration where b is the mini-batch size. Like Stochastic GD, no need to iterate over all examples, so it is much faster. b = 2-100 commonly used.

Stochastic vs. mini-batch: mini-batch likely to outperform Stochastic only if a vectorised implementation is used. The extra parameter b could also increase time taken to compute as finding the optimal b is required.

To check for convergence in Stochastic GD, plot avg cost every (say) 1000 iterations (so cost/iterations). Can then change learning rate or number of iterations accordingly. Learning rate can be slowly decreased over time using const1/(iterations + const2). This can take time to optimise however. Typically not done as keeping alpha constant will converge close enough to the global minimum.

Online learning: using a continuous stream of data (e.g. from a continuous stream of users) to train an algorithm. Each training example is discarded after use. This method also adapts to changing user preferences.

Repeat forever:

- Get (x,y) corresponding to user.
- Update theta using (x,y).

Map-reduce approach: divide data so that machines available (each computer) have the same ratio of data. Each temp thetaj will be combined afterwards on a master server. This is the same as batch gradient descent but is much quicker.

Map-reduce can also be applied on a single computer with mutiple cores.

13 Application Case Study

Machine Learning pipeline: distinct stages of application development.

Image $\rightarrow$ text detection $\rightarrow$ character segementation $\rightarrow$ character recognition.

Sliding window detection: for new image, an nxm image patch is taken based on the aspect ratio of y=1 and ran through the classifier. After that size patch has run through the entire image, a larger patch is chosen, which is scaled to the original nxm image patch and ran through the classifier, and so on. The step size determines the position of each subsequent image patch.

For text detection, text is found individually using the above, with the output an image where white=text, black=no text. Then an expansion operator is used which expands these singular white pixels based on whether there exists another white pixel in the near vicinity, so full words/text is found. Bounding boxes are then drawn around text with the correct aspect ratio (i.e. greater width than height).

Character segmentation finds blank space between two characters. y=1 will be sliding windows with the split and y=0 will be examples with whole characters or entirely blank space.

Once each character has been found, it can then be classified using a machine learning algorithm.

Artificial data synthesis: there are cases where data can be created 'artificially'. One example is in text recognition, where synthetic data can be created from taking characters from fonts, for a potentially unlimited supply of labelled data for a supervised learning algorithm.

Similarly distortions can be added to the original training examples to artificially increase the amount of labelled data. This can be applied to other problem, such as speech recognition where a voice can be overlayed on different background noise. The distortion should be representative of the type of distortions that may be seen in the test set.

Make sure to have a low bias classifier before synthesising more data, else the new examples won't help to optimise the algorithm.

Consider how long it will take to get 10x as much data, i.e. calculate how long it takes to get one example x how much you want. Can also use 'crowd sourcing' like Amazon Mechanical Turk.

Ceiling Analysis: what part of the pipeline should the most time be spent improving?

Find the accuracy of the overall system. Then manually determine the ground truth/classification of each example in the training set for the first part of the pipeline. Find the accuracy of the overall system again but with the 100% accuracy of the first pipeline stagee. Repeat for other parts of the pipeline. Using this method it is possible to see which areas of the pipeline can be improved the most.

14 Appendix A: Linear Algebra (Matrices and Vectors)

Matrix: 2-dimensional array of numbers.

Dimensions: rows x columns, $\mathbb{R}$ 3x2

A_{ij} denotes the entry in the i^{th} row, j^{th} column.

Vector: An n x 1 matrix.

A 4-dimensional vector contains 4 columns.

y_i refers to the i^{th} element (could be 0 or 1-indexed).

14.1 Operations

To add/subtact two matrices, they must be the same dimension.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a+w & b+x \\ c+y & d+z \end{bmatrix}$$

Scalar multiplication:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * x = \begin{bmatrix} a*x & b*x \\ c*x & d*x \end{bmatrix}$$

Matrix-vector multiplication: a **m x n matrix** multiplied by a **n x 1 vector** becomes a **m x 1 vector**.

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a*x+b*y \\ c*x+d*y \\ e*x+f*y \end{bmatrix}$$

Matrix-matrix multiplication: A **m x n matrix** mutliplied by a **n x o matrix** results in a **m x o matrix**.

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a*w+b*y & a*x+b*z \\ c*w+d*y & c*x+d*z \\ e*w+f*y & e*x+f*z \end{bmatrix}$$

14.2 Properties

- Not commutative, A x B != B x A.
- Is associative, A x B x C calculated as A x (B x C) and (A x B) x C.
- Identity Matrix (1 is the identity, denoted I). Is commutative!

14.3 Inverse and Transpose

Matrix Inverse:

If A is an m x m matrix, and has an inverse, $A(A^{-1}) = A^{-1}A = I$.

Only square matrices can have an inverse, i.e. 2x2 x inverse = I_{2x2}

I_{2x2} being $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Matrix Transpose:

$B_{ij} = A_{ji}$

e.g. $A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 5 & 9 \end{bmatrix}$ $A^T = \begin{bmatrix} 1 & 3 \\ 2 & 5 \\ 0 & 9 \end{bmatrix}$

15 Appendix B: Octave

Misc operators and functions:

```
~=                          % Not True
&&                          % AND
||                          % OR (Can also use xor(n, m))



disp()                      % print function
sprintf()                   % make string (sep=', quote='')
format long, short          % show floating point decimals

% semicolon suppresses output, comma chains commands
```

15.1 Vectors and Matrices:

```
A = [1 2; 3 4; 5 6]         % return a 3x2 matrix
B = [1; 2; 3]               % return a 1x3 matrix
V = [1 2 3]                 % return a 3-Dimensional vector

V = 1:0.1:2                 % return a vector starting at 1, ending
                            % at 2, increments of 0.1 inbetween

ones(2,3)                   % return a 2x3 matrix containing all 1
                            % multiply by 2 for all 2
zeros(n, m)                 % all 0

rand(n, m)                  % all random between 0-1
randn(n, m)                 % all random from Guassian distribution

eye(n)                      % return an n x n identity matrix

hist(A, n)                  % plot a histogram for matrix with
                            % n bins
```

```
VAR(n:m)                % show vector elements n to m
                        % show matrix element (n, m)
                        % (n,:) to show all in n row
                        % (:,n) to show all in n column
                        % use ([n m]) to show multiple

A(:,n) = [x, y, z]      % assignment
A = [A, [x, y, z]]
    or  [x; y; z]

C = [A B]               % concatenate horizontally
    [A; B]              % vertically

A(:)                    % put all matrix elements into vector

size(A, n)              % return size of matrix in a matrix
                        % n=1 for rows, 2 for columns
length(V)               % return size of vector
```

15.2 Managing Data

```
load FILE               % load data in file to Octave

who                     % show variables in current scope
whos                    % detailed info about variables

clear VAR               % remove variable (no arg=all)

save FILE VAR           % save VAR to FILE (.txt or .mat)
                        % -ascii option at end if .txt

addpath('dir')          % assert Octave search path
```

15.3 Computational Operations

```
A * B                   % multiply
A .* B                  % multiply corresponding elements
A .^ 2                  % square each element
1 ./ A                  % reciprical for each element

log(A)                  % element-wise logorithm
exp(A)                  % element-wise exponential
abs(A)                  % element-wise absolute value

-A                      % negative elements
A + 1                   % add 1 to each element

A'                      % transpose

max(V)                  % return max value
                        % returns max value row for matrix
[val, ind] = max(V)     % assign max value and its index
max(A,[],n)             % n=1 returns max column, 2 max row
                        % for matrices
max(max(A))             % return max value of max row
max(A(:))               % convert to vector to find abs max

A < n                   % returns True/False for each element
find(A < n)             % returns real value of True elements

[x,y] = find(A > n)     % assign index to x and y variables

magic(n)                % generate n x n matrix where each
                        % row/column adds up to same value


sum(A, n)               % return sum of all elements
                        % n=1 returns column sum, 2 row sum
prod(A)                 % return product (multiplication)
floor(A)                % round down
ceil(A)                 % round up
```

```
pinv(A)                    % return inverse
```

15.4 Plotting Data

```
plot(x, y)              % plot x against y
hold on                 % continue using current plot
xlabel(), ylabel()      % label x and y-axes ‘string’
legend()                % provide legend
title()                 % provide title

print -dpng ‘name’      % save as png

clf                     % clear figure
close                   % close figure

figure(n): plot()       % assign a plot to figure n

subplot (n, m, x)       % divide n x m grid, access element x
                        % then use plot() to plot in element

axis([x1 x2 y1 y2])     % set x and y ranges

imagesc(A)              % plot matrix as coloured grid
colorbar                % show colourbar legend
colormap COLOUR         % change colour of map
```

15.5 Control Statements

```
% for loops
% n:m is the range
for i=n:m,
  statement;
end;

% while loops
i = n
while true,
  statement;
  if i == n,
    break;      % break, continue available
  end;
end;

% if, elsif and else statements
if v(i) == n,
  statement;
elseif v(i) == m,
  statement;
else
  statement;
end;

% functions
% write in function_name.m file and cd into dir
function y = function_name(x)        % args = x

y = x + 1;

% multiple outputs
function [y1, y2] = function_name(x)

y1 = x + 1;
y2 = x + 2;

% call in interpreter
>> function_name(5)
ans = 6

% cost function
function J = costFunctionJ(X, y, theta)   % X is design matrix
                                          % y is class labels

m = size(X, 1);                           % training examples
predictions = X*theta;                    % predictions of h on all
```

```
                                          % m examples
sqrErrors = (predictions-y) .^ 2;         % squared errors

J = 1/(2*m) * sum(sqrErrors);             % final cost function
```

15.6 Vectorisation

Think of $h_\theta(x)$ as $\theta^T x$.

```
% unvectorised implementation
prediction = 0.0;
for j = 1:n+1,
  prediction = prediction + theta(j) * x(j)
end;

% vectorised implementation
prediction = theta' * x;
```

For simultaneously updated gradient descent, $\theta_0, \ldots \theta_n$ (as seen on page 3) can be thought of as:

$\theta := \theta - \alpha\delta$

where $\delta = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x^{(i)}$

$\theta = \mathbb{R}^{n+1}$

$\alpha = \mathbb{R}$

$\delta = \mathbb{R}^{n+1}$

$\alpha\delta = \mathbb{R}^{n+1}$

$(h_\theta(x^{(i)}) - y^{(i)}) = \mathbb{R}$
$x^{(i)} = \mathbb{R}^{n+1}$

$u(j) = 2v(j) + 5w(j)$ (for all j) vectorises to: $u = 2v + 5w$.

15.7 Linear Regression

```
% remember to add 1s to the first column of the design matrix

% compute cost function
function J = computeCost(X, y, theta)

m = length(y);
h = X * theta;
errors = h - y;
sqrErr = error .^ 2;

J = (1/2*m)) * sum(sqrErr);

% gradient descent
function theta = gradientDescent(X, y, theta, alpha, iterations)

m = length(y);
for iter = 1:iterations
  h = X * theta;
  errors = h - y;
  theta_change = alpha * (1/m) * (X' * errors);

  theta = theta - theta_change;

% as the above functions are vectorised they will also work for
% multivariate linear regression

% feature normalisation
function [X_norm, mu, sigma] = featureNormalise(X)

m = length(X);
mu = mean(X);
sigma = std(X);
mu_matrix = ones(m, 1) .* mu;
sigma_matrix = ones(m, 1) .* sigma;
```

```
X_norm = (X - mu_matrix) ./ sigma_matrix;

% prediction
prediction = [1, x1, x2, ...] * theta

% see docs/ML/ex1/ for info on how to plot graphs

% normal equation
function [theta] = normalEqn(X, y)

theta = pinv(X' * X) * X' * y;
```