Machine Learning

Notes from the Course by Stanford University's Andrew Ng

Joshua Brown

Contents

Chapter 1

Week 1

1.1 Model and Cost Functions

1.1.1 Notation

Some notation will be somewhat unique for this course, and some will be unique just to my notes

Notation	Meaning
X	The space of input variables
$x^{(i)}$	The i^{th} value of an input variable
Y	The space of outputs
$y^{(i)}$	The i^{th} value of the output variable
θ_j	The j^{th} model parameter
$J(\theta_0, \theta_1)$	The cost function
$h_{\theta_0,\theta_1}(x)$	The hypothesis function

1.1.2 Cost Function

Common cost function "Mean squared error"

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

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

Our goal is to minimize the value of the cost function: that is,

$$\underset{\theta_0,\theta_1}{\text{minimize}}\, J(\theta_0, \theta_1)$$

1.2 Parameter Learning

1.2.1 Gradient Descent

The gradient descent update is given by:

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

This needs to be a simultaneous update. That is,

```
temp0 = update(theta0)
temp1 = update(theta1)
theta0 = temp0
theta1 = temp1
```

A small value of α can lead to slow convergence. A large value of α can lead to no convergence, or even divergence. Because the derivative will shrink as you approach a local minimum, the step size will automatically get smaller.

1.2.2 Gradient Descent for Linear Regression

To apply gradient descent to linear regression, we need to calculate the derivative term:

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$= \frac{\partial}{\partial \theta_j} \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2$$

A little calculus yields:

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h(x^{(i)}) - y^{(i)} \right)$$

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h(x^{(i)}) - y^{(i)} \right) x^{(i)}$$

The cost function for linear regression is convex, which is great because it means we do not have to worry about multiple local minima.

"Batch" Gradient Descent: Each iteration uses all of the training examples.

An exact solution exists for linear regression, but gradient descent scales better to larger data sets.

1.3 Linear Algebra Review

Instead, let's just practice typesetting matrices.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$A_{12} = 2$$

$$\mathbf{x} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

$$x_3 = 3$$

Professor Ng refers to vectors as being synonymous with $n \times 1$ matrices. This is wrong (imprecise), but pretty common, so we'll go with it since it is compatible with the MatLab/Octave representation.

Neat trick: represent competing hypotheses using matrix-matrix multiplication:

$$sizes = \begin{pmatrix} 2104 \\ 1416 \\ 1534 \\ 852 \end{pmatrix}$$

Three hypotheses: $h_\theta(x) = -40 + 0.25x$, and $h_\theta(x) = 200 + 0.1x$, and $h_\theta(x) = -150 + 0.4x$

$$\begin{pmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1534 \\ 1 & 852 \end{pmatrix} \begin{pmatrix} -40 & 200 & -150 \\ 0.25 & 0.1 & 0.4 \end{pmatrix} = \begin{pmatrix} 486 & 410 & 692 \\ 314 & 342 & 416 \\ 344 & 353 & 464 \\ 173 & 285 & 191 \end{pmatrix}$$

Each column on the right side is a set of predictions for each of the three hypothesis functions!

Chapter 2

Week 2

2.1 Multivariate Linear Regression

2.1.1 Notation

Notation revisited—we have to introduce a little bit of new notation to deal with multiple input features.

Notation	Meaning
m	The number of training examples
n	The number of variables
X	The space of input variables
$x^{(i)}$	The input features of the i^{th} training example
$x_j^{(i)}$	The value of the j^{th} feature in the i^{th} training example
Y	The space of outputs
$y^{(i)}$	The i^{th} value of the output variable
θ_j	The j^{th} model parameter
$J(\theta_0, \theta_1)$	The cost function
$h_{\theta_0,\theta_1}(x)$	The hypothesis function

2.1.2 Multiple Features

We also need a new form for the hypothesis function:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n = \theta_0 + \sum_{j=1}^{n} \theta_j x_j$$

Alternately, define $x_0 = 1$. So,

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$$

Or even more simply,

$$h_\theta(x) = \theta \cdot \mathbf{x} = \theta^T \mathbf{x}$$

2.1.3 Gradient Descent

The only thing new is the partial derivative term:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=0}^{m} \Big(h_\theta(x) - y^{(i)} \Big) x_j^{(i)}$$

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=0}^{m} \Big(\theta^T \mathbf{x}^{(i)} - y^{(i)} \Big) x_j^{(i)}$$

So our gradient descent algorithm becomes

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

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=0}^{m} \Big(\theta^T \mathbf{x}^{(i)} - y^{(i)} \Big) x_j^{(i)}$$

2.1.4 Feature Scaling

Gradient descent converges more quickly when the features have a similar scale. When the scales are very different, the contour ellipsoids are very narrow, so the gradient descent can oscillate. Concretely, just get each variable to a $-1 \leq x \leq 1$ range. Roughly. You can also center the data.

$$x_j := \frac{x_j - \mu_j}{\sigma_j}$$

2.1.5 Learning Rate

For a good value of α, the cost J should decrease with every iteration. If it diverges or oscillates, then α is too large. If it converges too slowly, then α is too small. Ideally, find one value that is too large and one value that is too small, and choose something in between.

2.1.6 Features and Polynomial Regression

You can also feature engineer.

- Combine features together to form new features
- Operate on a single feature for form a new feature (square it, etc.)

When you use polynomial features, feature scaling becomes especially important.

2.1.7 Computing Parameters Analytically

An alternative to gradient descent is using a closed-form solution which identifies the best fit. When $X\theta = y$ is overdetermined, the best-fit solution is given by the Normal Equation:

$$X^T X \theta = X^T y$$

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

This is obtained by solving $\frac{\partial}{\partial \theta_j} J_\theta = 0$ for all j. In this context, X is sometimes called the Design Matrix, and it is $m \times (n+1)$.

Octave:

```
theta = pinv(X' * X) * X' * y
% pinv will give the best theta even if X' * X is not invertible!
```

Advantages of normal equation:

- No need to choose α
- No need to iterate

Disadvantages of normal equation:

- Does not work as well for large n (much bigger than 1000) because $X^T X$ becomes an expensive calculation ($\mathcal{O}(n^3)$)

2.2 Octave

```
% Range (inclusive)
r = 1:6
% [1 2 3 4 5 6]

v = 1:0.1:2
% [1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0]

% Print a histogram
normal = -6 + 10 * randn(1, 10000);
bincount = 50;
hist(normal, bincount)

% Dimensions
A = [1 2; 3 4; 5 6];
length(A) % 3 - length gives the longest dimension
size(A) % [3 2];

% Load data
load('datafile.dat')

% Indexing
y = randn(1, 100000)
v = y(1:10) % v is the first ten elements of y

A(3,2) % 6
A(2,:) % [3 4]
A(:,2) % [2;4;6]
A([1 3], :) % [1 2; 5 6]
A(:) % [1; 2; 3; 4; 5; 6]
B = [A A] % [1 2 1 2; 3 4 3 4; 5 6 5 6]


% Save variable to a file
save filename.mat v
clear
```

```
% Restore v from the file
load filename.mat

% Use a cleartext format instead of binary
save filename.txt v -ascii

% Multiplication
C = A * B

% Elementwise Multiplication
C = A .* B

% Elementwise exponentiation
C = A .^ 2

% Elementwise division
C = 1 ./ A

% log, exp, abs, +, -, <, > are elementwise as well

% Transpose
AT = A'

v = [1 3 2]
A = [1 3 2; 4 2 5]

[val, ind] = max(v) % [[3, 2]]
ind = find(v < 3) % [1 3]
[r, c] = find(A < 3) % [1 1 2], [1 3 2]

% Heatmap of Matrix
imagesc(A)

%
% Control flow
%
v = zeros(1, 10)
for i = 1:10
    v(1,i) = i^2;
end


indices = 1:10
for i = indices
    v(1,i) = i^3
end

j = 100
n = 0
while j > 1
    n = n + j
    j = j - 1
    if j == 53
        break
```

```
    elseif j == 63
        j = j - 4
    end
end
```

2.2.1 Function files

```
% In squareAndCube.m
function [y1, y2] = squareAndCube(x)
y1 = x * x;
y2 = y1 * x;
end
```

To make it accessible, you can either change to its directory, or use `addpath('/path/to/file/')`

2.2.2 Vectorization

```
x = [1, 2, 3, 4, 5];
theta = [2, 3, 4, 5, 6];

% Slowly...
val = 0;
for i = 1:length(x)
    val = val + x(i) * theta(i)
end

% Alternately
val = theta' * x
```

$$J = \frac{1}{2m} \sum (X\theta - y)^T (X\theta - y) + \frac{\lambda}{2m} \sum \theta \odot \theta$$

$$\frac{\partial}{\partial \theta} J = \frac{1}{m} X^T (X\theta - y) + \frac{\lambda}{m} \theta$$

Chapter 3

Week 3

3.1 Logistic Regression

3.1.1 Classification and Representation

Predicting the value of $y \in \{0, 1\}$ to start. Later, $y \in \{0, 1, 2, 3\}$

Naïve solution: just apply linear regression, and set a threshold. But this produces bad outputs in many cases; the accuracy is very sensitive to the threshold value. Also, we could predict values > 1 or < 0, which is nonsense.

Hypothesis Representation

Instead of $h_\theta(x) = \theta^T x$, use a modified hypothesis function:

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

Here, $g(z)$ is the Sigmoid function (or logistic function). It maps $\mathbb{R} \to (0, 1)$. We will treat the output of the hypothesis function as the probability of an event. More formally,

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

Read this as "The probability that $y = 1$, given x, parameterized by θ"

Decision Boundary

Suppose we set a threshold of 0.5, so we predict $y = 1$ if $h_\theta(x) \geq 0.5$. By inspecting a plot of g, we can see that $g(z) \geq 0.5$ whenever $z \geq 0$. So we predict an event whenever $\theta^T x \geq 0$.

This effectively draws a decision boundary (hyperplane) through the data. We can produce a nonlinear decision boundary by adding polynomial terms. (Of course, it is still a hyperplane when these plynomial terms are just considered to be separate dimensions.)

3.1.2 Logistic Regression Model

Cost Function

A slightly rewritten version of the linear regression cost function:

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

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

For logistic regression, this is not ideal. h includes $g(z)$, which is nonlinear. This means J would not be convex, so gradient descent would not be guaranteed to converge to a global minimum. Instead, try:

$$\text{cost}(h, y) = \begin{cases} -\log(h) & \text{if } y = 1 \\ -\log(1 - h) & \text{if } y = 0 \end{cases}$$

If the probability (h) is 1 and the value is 1 then the cost is 0. If the probability is 0 and the value is 1, then the cost goes to ∞. If the probability is 0 and the value is 0 then the cost is 0. If the probability is 1 and the value is 0, then the cost goes to ∞.

Equivalently,

$$\text{cost}(h, y) = -y\log(h) - (1 - y)\log(1 - h)$$

So,

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

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

This can actually be derived statistically from maximum likelihood estimation.

Vectorizing some of these calculations yields:

$$h = g(X\theta)$$

$$J(\theta) = -\frac{1}{m}\big(y^T\log(h) + (1 - y)^T\log(1 - h)\big)$$

Gradient Descent

Usual template: find $\min_\theta J(\theta)$ by iterating $\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta)$.

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

And we're done. This is exactly what we have for linear regression, except the hypothesis function is different.

Also, feature scaling is useful here, for the same reason that it was useful for linear regression.

Vectorizing this yields:

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

Advanced Optimization

For gradient descent, to find $\min_\theta J(\theta)$, we need to have code to compute $J(\theta)$ and $\frac{\partial}{\partial\theta_j}J(\theta)$ for all j. Technically you only need to compute the partials, but if you're monitoring J then you need to compute it.

Other algorithms have exactly the same requirements:

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

For the latter three, there is no need to manually choose a learning rate α, and they are often faster than gradient descent, although more complex.

Implementing these can be complicated, but you can still use them by implementing functions to calculate J and $\frac{\partial}{\partial\theta_j}J$

```
function [jVal, gradient] = costFunction(theta)
  gradient = % implement gradient calculation
  jVal = % implement cost calculation
end

options = optimset('GradObj', 'on', 'MaxIter', '100');
initialTheta = zeros(2,1);

% fminunc -> function minimization unconstrained
% @costFunction -> function pointer
[optimalTheta, functionVal, exitFlag] = fminunc(@costFunction,
   initialTheta, options);
```

3.1.3 Multiclass Classification: One-vs-All

For a three-class classification $y \in \{1, 2, 3\}$, split it into three problems: 1-vs-not-1, 2-vs-not-2, and 3-vs-not-3. The prediction is the class that has the highest probability.

3.2 Regularization

3.2.1 Solving the Problem of Overfitting

"Underfitting" or "high bias" means the model is not fitting the training data well. "Overfitting" means the model is fitting the training data too well, and does not generalize well.

Options to address this:

- Reduce the number of features, either by manual selection or a model selection algorithm
- Regularization: keeps all the features but reduces the magnitudes of the parameters θ_j. This works well when there are a lot of features, and all of them actually contributes to y

3.2.2 Cost Function

We can penalize particular terms by adding terms to the cost function. For polynomial regression $h = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$, we can penalize overfitting functions by ensuring that θ_3 and θ_4 are small.

$$J(\theta) := J(\theta) + 1000\theta_3^2 + 1000\theta_4^2$$

In general, smaller parameter values correspond to "simpler" hypotheses. Instead of the specific example above, we can introduce a general $l2$ regularization term:

$$\lambda \sum_{j=1}^{m} \theta_j^2$$

By convention, we do not penalize θ_0 so notice the sum above begins at 1. λ is called the regularization parameter. It controls the tradeoff between fitting the data well and keeping the parameters small. If λ is too large, it may result in underfitting. If λ is too small, it may result in overfitting.

3.2.3 Regularized Linear Regression

Updated cost function:

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

Updated gradient descent based on new :

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

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

The term $(1-\alpha\frac{\lambda}{m})$ is always less than 1 (and usually pretty close to 1). So in each gradient descent iteration, θ_j is automatically scaled down a little.

To use regularization with the normal equation, $(X^TX)^{-1}X^Ty$ becomes

$$\left(X^TX + \lambda\begin{bmatrix}0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1\end{bmatrix}\right)^{-1} X^Ty$$

If $m \leq n$ then X^TX is singular, which is a problem! But if $\lambda > 0$ then that fixes the invertibility problem.

3.2.4 Regularized Logistic Regression

Updated gradient descent:

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

3.2.5 Vectorized Implementation

$$J = \frac{1}{2m} \sum (g(X\theta) - y)^T (g(X\theta) - y) + \frac{\lambda}{2m} \sum \theta \odot \theta$$

$$\frac{\partial}{\partial \theta} J = \frac{1}{m} X^T (g(X\theta) - y) + \frac{\lambda}{m} \theta$$

Chapter 4

Week 4

4.1 Nonlinear Hypotheses

We can produce nonlinear hypotheses in logistic regression by adding polynomial terms; but this only well for small n because the combinations of features are easy to enumerate. With a large n, this would end up as a combinatorial explosion of generated features.

Image recognition is a classic case where n is necessarily large.

4.2 Neural Networks

Each input (x_1, x_2, x_3) to a neuron produces some output $h_\theta(x) = \frac{1}{1+e^{\theta^T x}}$. This example would be a neuron with a sigmoid activation function. θ_j are the weights of x_j.

The first layer is the "input layer". The final layer is the "output layer". Intermediate layers are the "hidden layers". Each neuron in layer j is connected to each neuron in layer $j+1$.

Some notation:

Notation	Meaning
$a_i^{(j)}$	Activation of unit i in layer j
$\Theta^{(j)}$	Matrix of weights controlling the function mapping layer j to layer $j+1$
$\Theta_{12}^{(3)}$	Entry $(1,2)$ in the matrix connecting layers 3 and 4
s_j	The number of units in layer j

So given a network with three inputs, three neurons in the hidden layer, and one output, we have:

$$\begin{aligned} a_1^{(2)} &= g\Big(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\Big) \\ a_2^{(2)} &= g\Big(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\Big) \\ a_3^{(2)} &= g\Big(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\Big) \\ h_\Theta(x) = a_1^{(3)} &= g\Big(\Theta_{10}^{(2)}x_0 + \Theta_{11}^{(2)}x_1 + \Theta_{12}^{(2)}x_2 + \Theta_{13}^{(2)}x_3\Big) \end{aligned}$$

If layer j has s_j units and layer $j+1$ has s_{j+1} units, then $\Theta^{(j)}$ has size $s_{j+1} \times (s_j + 1)$. The $+1$ comes from the bias node which always sends a value of 1.

Define $z_i^{(j)}$ such that $a_i^{(j)} = g(z_i^{(j)})$. For consistency, write $a^{(1)} = x$.

Let $a^{(1)} = x = \begin{pmatrix} x_0 \\ \vdots \\ x_3 \end{pmatrix}$ and $z^{(2)} = \begin{pmatrix} z_1^{(2)} \\ \vdots \\ z_3^{(2)} \end{pmatrix}$. Then $z^{(2)} = \Theta^{(2)} a^{(1)}$ and $a^{(2)} = g(z_2^{())}$. Succinctly,

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

This process of calculating h_Θ from the inputs is called "forward propagation". Fully vectorizing this, we have:

$$A^{(1)} = X \qquad A^{(j+1)} = g\Big(\begin{bmatrix} 1 & A^{(j)} \end{bmatrix} \Theta^{(j)T}\Big)$$

Effectively, the last step is equivalent to logistic regression, except instead of being constrained to use x_j as the inputs features, the neural network is free to create its own input features from x_j.

"Network architectures" define the shapes of neural networks—that is, the number of nodes in each layer, and how the layers are connected.

4.2.1 Examples

Individual neurons can be used to compute logical functions like and, or, xor, etc.

$$\begin{aligned} \Theta_{and} &= \begin{pmatrix} -30 & 20 & 20 \end{pmatrix} \\ \Theta_{or} &= \begin{pmatrix} -10 & 20 & 20 \end{pmatrix} \\ \Theta_{nor} &= \begin{pmatrix} 10 & -20 & -20 \end{pmatrix} \\ \Theta_{not} &= \begin{pmatrix} 10 & -20 \end{pmatrix} \end{aligned}$$

And these can be combined.

4.2.2 Multiclass Classification

Similar to one-vs-all method for logistic regression. For k classes, have k output nodes. The prediction is the output node with the largest value. Each training example is $y \in \mathbb{R}^k$.

Chapter 5

Week 5

5.1 Neural Network Cost Function

Let L be the number of layers in the network, of sizes s_l for $1 \leq l \leq L$. Let K be the number of units in the output layer. That is, $K = s_L$.

The cost function is generalized from the logistic regression cost function. Recall, for logistic regression:

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

For neural networks, we have

$$h_\Theta(x) \in \mathbb{R}^K \qquad (h_\Theta(x))_i = i^{th}\text{output}$$

And we can generalize the cost function to:

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

The extra sum inside the first term is a sum over all classes, so our cost function considers the error for each possible output. The regularization term at the end is just the sum over all layers l of the sum of all the Θ entries squared (except the sum does not include the entries with subscript 0, so it ignores bias units).

Note: in the triple sum, i is not training example i, but it is node i within layer l, and j is node j within layer $l+1$.

5.2 Backpropagation

Backpropagation is an algorithm to minimize $J(\Theta)$ for a neural network. Using gradient descent or another optimization algorithm, we would need to compute $J(\Theta)$ and $\frac{\partial}{\partial\Theta_{ij}^{(l)}}$ for all i, k, and l.

Consider the case with a single training example (x, y), and a network with $s_1 = 3$, $s_2 = 5$, $s_3 = 5$, and $s_4 = 4$. Forward propagation looks like this:

$$
\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g\Big(z^{(2)}\Big) \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g\Big(z^{(3)}\Big) \\
z^{(4)} &= \Theta^{(3)} a^{(3)} \\
a^{(4)} &= g\Big(z^{(4)}\Big) \\
h_\Theta(x) &= a^{(4)}
\end{aligned}
$$

Where, recall, $a_0^{(l)}$ must be added to the matrices at each level.

For backpropagation, we will compute $\delta_j^{(l)}$, which is the "error" of node j in layer l. For the last layer, this is straightforward:

$$\delta_j^{(4)} = a_j^{(4)} - y_j = h_\Theta(x)_j - y_j$$

Or vectorized, $\delta^{(4)} = a^{(4)} - y$. Applying the chain rule, we can calculate δs for previous layers:

$$
\begin{aligned}
\delta^{(3)} &= (\Theta^{(3)})^T \delta^{(4)} \odot g'(z^{(3)}) \\
\delta^{(2)} &= (\Theta^{(2)})^T \delta^{(3)} \odot g'(z^{(2)})
\end{aligned}
$$

Formally, $g'(z^{(3)})$ is the derivative of the sigmoid function evaluated at the values $z^{(3)}$. In practice, we can evaluate it as $a^{(3)} \odot (1 - a^{(3)})$. Note: $\delta^{(1)}$ does not exist, because the first layer are the input features. So:

$$\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} \odot a^{(l)} \odot (1 - a^{(l)})$$

Ignoring λ (or assuming $\lambda = 0$), it can be shown that:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_k^{(l)} \delta_i^{(l+1)}$$

Now we have enough background to write a backpropagation algorithm.

1. Training set $\big\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\big\}$
2. Set $\Delta_{ij}^{(l)} = 0$ for all l, i, j
3. For all training examples $i \in 1, \ldots, m$:
   (a) Set $a^{(1)} = x^{(i)}$
   (b) Perform forward propagation to compute $a^{(l)}$ for $l \in 2, \ldots, L$
   (c) Compute $\delta^{(L)} = a^{(L)} - y^{(i)}$ (the error for the outputs for this training example)
   (d) Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
   (e) Set $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

   i. Vectorized: $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} \times a^{(l)}$

4. Set $D_{ij}^{(l)} := \frac{1}{m}\left(\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)}\right)$ if $j \neq 0$ (not the bias term)

5. Set $D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)}$ if $j = 0$ (the unregularized bias term)

Then we have:

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

So we do FP followed by BP for each training example in turn. For reference, in the algorithm above,

- i is the index of the current training example in $1, 2, \ldots, m$
- j is the index of the current output value in $1, 2, \ldots, K$
- l is the index of the current layer in $1, 2, \ldots, L$

5.2.1 Intuition

For unit j in layer l,

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}}\text{cost}(i)$$

$$\text{cost}(i) = y^{(i)}\log\left(h_\Theta(x^{(i)})\right) + (1 - y^{(i)})\log\left(1 - h_\Theta(x^{(i)})\right)$$

In other words, $\delta_j^{(l)}$ is the amount we would like to change $z_j^{(l)}$ in order to reduce the cost for that node.

For forward propagation, we had:

$$z_1^{(l+1)} = \Theta_{10}^{(l)}1 + \Theta_{11}^{(l)}a_1^{(l)} + \cdots + \Theta_{1s_l}^{(l)}a_{s_l}^{(l)}$$

That is, the sum over all nodes feeding into node 1 in layer $l+1$ of the inputs to that node multiplied by the weight of the connection. For backward propagation, we have:

$$\delta_2^{(l-1)} = \Theta_{12}^{(l)}\delta_1^{(l)} + \cdots + \Theta_{s_l 2}^{(l)}\delta_{s_l}^{(l)}$$

That is, the sum over all nodes getting input from node 2 of layer $l-1$ of the errors of those nodes multiplied by the weight of the connection.

5.3 Backpropagation in Practice

5.3.1 Unrolling Parameters

```
function [J, gradient] = costFunction(theta)
  gradient = % implement gradient calculation
  J = % implement cost calculation
end

[optimalTheta, functionVal, exitFlag] = fminunc(@costFunction,
   initialTheta, options);
```

The `fminunc` function in Octave assumes that `theta` is a vector being used to compute a `gradient` which is also a vector. But for our cost function and its gradient, we use matrices $\Theta^{(1)}$ through $\Theta^{(L)}$ to compute deltas $D^{(1)}$ through $D^{(L)}$.

So we must "unroll" our matrices into vectors. For example, assuming:

$$s_1 = 10 \qquad s_2 = 10 \qquad s_3 = 1$$

We have:

$$\Theta^{(1)}, D^{(1)} \in \mathbb{R}^{10\times 11} \qquad \Theta^{(2)}, D^{(2)} \in \mathbb{R}^{10\times 11} \qquad \Theta^{(3)}, D^{(3)} \in \mathbb{R}^{1\times 11}$$

We can unroll these into vectors with:

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

And we can roll them back into matrices with:

```
% Note: the indices are inclusive
Theta1 = reshape(thetaVec(1:110),   10, 11);
Theta2 = reshape(thetaVec(111:220), 10, 11);
Theta3 = reshape(thetaVec(221:231), 1,  11);
D1 = reshape(DVec(1:110),   10, 11);
D2 = reshape(DVec(111:220), 10, 11);
D3 = reshape(DVec(221:231), 1,  11);
```

5.3.2 Gradient Checking

It's possible to have J always decreasing even with a buggy implementation! We can use gradient checking to try to find these subtle bugs. Check our calculated gradients against an approximation for the partial derivatives:

$$\frac{\partial}{\partial theta_i} J(\theta) \approx \frac{J(\theta_1 + \cdots + (\theta_i + \epsilon) + \cdots + \theta_n) - J(\theta_1 + \cdots + (\theta_i - \epsilon) + \cdots + \theta_n)}{2\epsilon}$$

Be sure to turn off gradient checking before training the model!

5.3.3 Random Initialization

Initializing $\Theta_{ij}^{(l)} = 0 \forall i, j, l$ does not work. This is because the hidden units will all calculate the same feature. Instead, initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$.

```
Theta1 = rand(10,11) * (2*INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(10,11) * (2*INIT_EPSILON) - INIT_EPSILON;
Theta3 = rand(1,11) * (2*INIT_EPSILON) - INIT_EPSILON;
```

5.3.4 Putting it Together

Choose an architecture

First and last layers are constrained by the number of input values and the number of output classes. A reasonable default is to use one hidden layer. If more than one hidden layer, use the same number of nodes in each layer. Usually, the more hidden units the better.

Train the neural network

1. Randomly initialize weights
2. Implement forward propagation to compute $h_\Theta(x^{(i)})$
3. Implement code to compute cost function $J(\Theta)$
4. Implement back propagation to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$
5. Use gradient checking to compare partial derivatives to numerical computation
6. Use gradient descent or other optimization method to minimize $J(\Theta)$

Tip: use a for-loop for the first implementation. Use forward prop and back prop for a single $x^{(i)}$ at a time.

Note: $J(\Theta)$ is non-convex so you can get stuck in local optima. This is usually not a big deal.

Chapter 6

Week 6

6.1 Advice for Applying Machine Learning

6.1.1 Deciding what to try next

- Get more training examples
- Try smaller sets of features
- Try getting additional features
- Try changing the regularization parameter λ

Decide which option using machine learning diagnostics.

6.1.2 Evaluating a hypothesis

Split data into training set and test set. Train the data based on the training set and evaluate the model using the test set. If a model is overfit, then

$$J_{test}(\theta) > J(\theta)$$

Another metric is misclassification error ("0/1 misclassification error").

$$\text{err}(h, y) = \begin{cases} 1 \text{ if } h \geq 0.5 \text{ and } y = 0 \text{ or } h < 0.5 \text{ and } y = 1 \\ 0 \text{ if } h < 0.5 \text{ and } y = 0 \text{ or } h \geq 0.5 \text{ and } y = 1 \end{cases}$$

$$\text{Misclassification Error } = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \text{err}(h_\theta(x_{test}^{(i)}), y^{(i)})$$

In other words, the misclassificaiton error is the percentage of the test set that was classified incorrectly.

6.1.3 Model Selection

Consider a hyperparameter d that defines the degree of polynomial used in a regression problem. The simple approach is to compare $J_{test}(\theta^{(i)})$ for all d_i and choose the one with the lowest error. But this might not necessarily generalize well; d has been fit to the test set.

Instead, split the data into three pieces: a training set, a cross-validation set, and a test set. (Try 60/20/20 ratio.) Find which value of d applied to the cross-validation set produces the lowest J. Then, applying this model to the test set yields a good estimate for how well the model will generalize.

6.1.4 Diagnosing bias vs. variance

High bias is equivalent to underfitting; high variance is equivalent to overfitting.

Using the same example, increasing d decreases the training error J_{train}, probably monotonically. On the other hand, J_{cv} will be optimized for some value of d; it will decrease then increase as d increases.

So assume you have a model that is performing badly; J_{cv} or J_{test} is high. How can you differentiate between high bias and high variance? Based on the observation above, if J_{train} and J_{cv} are both high, then it is probably a high-bias (underfitting) scenario. If J_{train} is low and J_{cv} is high, then it is probably a high-variance (overfitting) scenario.

6.1.5 Regularization and bias/variance

Large λ leads to high bias (underfitting). Think a straight horizontal line.

Small λ leads to high variance (overfitting). Think a high-degree polynomial.

Try a grid of values for λ from a very small value to a very large value, and find the value that minimizes J.

6.1.6 Learning Curves

Plot J_{train} and J_{cv} as a function of the training set size m. As m increases, it makes sense that J_{train} should start low and increase; it is easy to fit a small data set perfectly. Likewise, J_{cv} should start high and decrease; small data sets do not generalize well.

In a high-bias case, J_{cv} will decrease and stabilize at a certain value, while J_{train} will increase and stabilize just below J_{cv}. The value at which they stabilize will be pretty high. In this case, getting more training data will not help.

In a high-variance case, J_{train} will still increase and J_{cv} will still decrease, but the gap between them will remain large; the model will continue to perform much better on the training data. In this case, getting more data is likely to help.

6.1.7 What to do next, revisited

- Get more training examples (fixes high variance)
- Try fewer features (fixes high variance)
- Try more features (fixes high bias)
- Try adding polynomial features (fixes high bias)
- Try decreasing λ (fixes high bias)
- Try increasing λ (fixes high variance)

Small neural networks have fewer features. They are computationally cheaper, but more prone to bias. Large neural networks are computationally more expensive, and more prone to overfitting. But usually using regularization is more effective than using a smaller network, unless computation time is an issue.

6.2 Machine Learning System Design

6.2.1 Error analysis

Manually examine the examples your model misclassifies. This might inspire you to create new features, for example.

Having a numerical measurement of model performance is important, because you can use that metric to make decisions about what to do in your model.

For skewed classes, the metric is more difficult to come up with. Accuracy, for example, is a particularly bad choice. Precision/Recall is better.

	Actual 1	Actual 0
Predicted 1	true positive	false positive
Predicted 0	false negative	true negative

$$\text{Precision} = P(\text{event}|\text{predicted event}) = \frac{\text{true positives}}{\text{predicted positives}} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

$$\text{Recall} = P(\text{predicted event}|\text{event}) = \frac{\text{true positives}}{\text{actual positives}} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

Sometimes we need to control the tradeoff between precision and recall.

Suppose we want to predict $y = 1$ only if we're very confident. In other words, set the cutoff threshold to something higher than 0.5. Such a model will have higher precision, but lower recall.

Suppose we want to avoid missing too many positives, so we set a lower cutoff threshold. Such a model will have higher recall but lower precision.

F_1 score allows you to have a single number for evaluating precision and recall. Note: $P + R/2$ is not a good choice for both of the reasons above.

$$F_1 = 2\frac{PR}{P + R}$$

6.2.2 Data for Machine Learning

Under some circumstances, getting a lot of data is ideal. The results of many algorithms increase monotonically with m.

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

Given assumptions:

1. Features are sufficient for prediction
2. Algorithm has many parameters ($\dim(\theta)$ is large)

Then we can say these are "low-bias" algorithms, and J_{train} ought to be small. If we have a very large training set, then we are unlikely to overfit (i.e. $J_{train} \approx J_{test}$). Therefore, the test set error should be small.

So in these circumstances, more data is always better!

Chapter 7

Week 7

7.1 Support Vector Machines

7.1.1 Alternate view of logistic regression

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

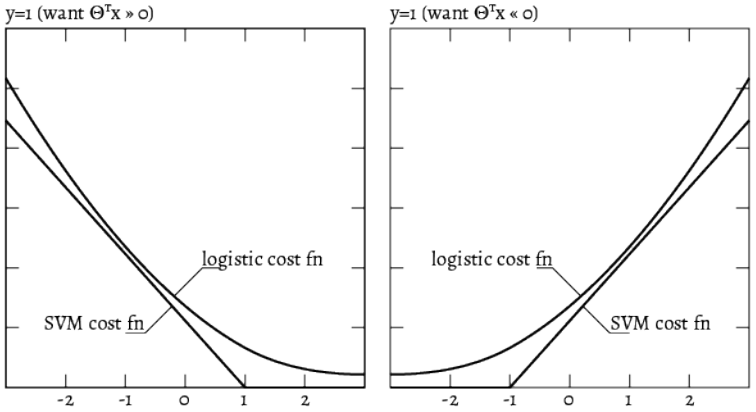
Ideally, if $y = 1$, we would like $h_\theta(x) \approx 1$, so we want $\theta^T x \gg 0$. Likewise, if $y = 0$, we want $\theta^T x \ll 0$.

Each example contributes the following cost:

$$-(y \log(h_\theta(x)) + (1 - y) \log(1 - h_\theta(x)))$$

The first term only applies when $y = 1$ and the second term only applies when $y = 0$.

For a SVM, we modify this cost term to be flat at 0 (no cost) when the decision is correct, and to grow linearly the more the decision is incorrect. (The slope doesn't matter much.)



Notation: these cost functions are called $\text{cost}_1(z)$ for the $y = 1$ case, and $\text{cost}_0(z)$ for the $y = 0$ case. So a Support Vector Machine has a cost function:

$$\min_\theta C \sum_{i=1}^{m} \left[y^{(i)} \, \text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \, \text{cost}_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{j=0}^{n} \theta_j^2$$

By convention, we exclude the factor of $\frac{1}{m}$ from the cost function, and we use C as the regularization parameter. That is, we minimize $CA + B$ rather than $A + \lambda B$. So a small C weighs regularization heavily, while a large C weighs fitting the training set heavily. (Think of $C \approx \frac{1}{\lambda}$).

Unlike logistic regression, an SVM does not output a probability.

$$h_\theta(x) = \begin{cases} 1 \text{ if } \theta^T x \geq 0 \\ 0 \text{ if } \theta^T x < 0 \end{cases}$$

7.1.2 Large Margin Intuition

$\text{cost}_1(z)$ is 0 only when $z \geq 1$, even though we have classified correctly whenever $z \geq 0$. Likewise, $\text{cost}_0(z)$ is 0 only when $z \leq -1$, even though we have classified correctly whenever $z < 0$. Consequently, we want not just to barely get the examples right, but to have a significant margin of correctness.

Imagine C is extremely large, so we weigh fitting the data very highly. So whenever $y^{(i)} = 1$ we want $\theta^T x \geq 1$, and whenever $y^{(i)} = 0$ we want $\theta^T x \leq -1$. Since C is so large, we will probably reject any hypothesis which incorrectly classifies any examples, so we are left minimizing only the regularization term.

For a linearly separable case, this will draw a decision boundary which has the largest possible margin between the boundary and the closest points.

For very large C, the SVM is very sensitive to outliers, and might drastically change the boundary to accommodate a single point. Regularization prevents this.

7.1.3 Mathematics behind large margin classification

Consider $u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$ and $v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$. Then the inner product $u^T v$ is $u_1 v_1 + u_2 v_2$. And $\|u\| = \sqrt{u_1^2 + u_2^2}$. We also have $u^T v = \text{proj}_u(v) \cdot \|u\| = \text{proj}_v(u) \cdot \|v\|$.

For the SVM decision boundary, we are effectively minimizing

$$\min_\theta \frac{1}{2} \sum_{j=1}^{n} \theta_j^2 \quad \text{such that} \quad \theta^T x^{(i)} \geq 1 \text{ if } y^{(i)} = 1 \quad \text{and} \quad \theta^T x^{(i)} \leq -1 \text{ if } y^{(i)} = 0$$

Assume $\theta_0 = 0$. In the case where $n = 2$, we are minimizing $\theta_1^2 + \theta_2^2$, or equivalently, $\left(\sqrt{\theta_1^2 + \theta_2^2}\right)^2$, which is $\|\theta\|^2$.

Now consider $\theta^T x^{(i)}$. This can be thought of as the projection of $x^{(i)}$ onto the parameter vector θ times $\|\theta\|$. So we are really finding:

$$\min_\theta \frac{1}{2} \|\theta\|^2 \quad \text{such that} \quad \text{proj}_\theta x^{(i)} \cdot \|\theta\| \geq 1 \text{ if } y^{(i)} = 1 \quad \text{and} \quad \text{proj}_\theta x^{(i)} \cdot \|\theta\| \leq -1 \text{ if } y^{(i)} = 0$$

Simplifying notation: let $p^{(i)} = \text{proj}_\theta x^{(i)}$

This all makes sense if θ is a vector orthogonal to the decision boundary. The projections of $x^{(i)}$ onto θ are the distances to the decision boundary, and we require these all to be above 1.

If the boundary is bad (small margins), then $p^{(i)}$ is small, so by our condition above, $\|\theta\|$ must be large (in order for the product to be greater than 1). Since we're trying to minimize $\|\theta\|^2$, our algorithm will not choose such a boundary.

Removing the assumption that $\theta_0 = 0$ just allows us to have decision boundaries that do not pass through the origin. It is just somewhat less obvious to show that this leads to a large margin classifier.

7.2 Kernels

Kernels adapt SVMs to support non-linear models. One way is to include polynomial features. This would look like:

$$\text{Predict } y = 1 \text{ if } \quad \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \cdots \geq 0$$

Instead we can write this more generally as

$$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 + \theta_4 f_4 + \cdots \geq 0$$

$$\theta^T f \geq 0$$

Where f is the new set of features. But f doesn't have to come from polynomial features.

Example: compute new features based on proximity to landmark points $l^{(1)}, l^{(2)}, l^{(3)}$. So $f_j = \text{similarity}\big(x, l^{(j)}\big)$. We can use a similarity metric like:

$$\text{similarity}\Big(x, l^{(j)}\Big) = \exp\left(-\frac{\big\|x - l^{(j)}\big\|^2}{2\sigma^2}\right)$$

The similarity function is called a kernel, and this particular function is called a Gaussian kernel. It has the feature that $f \approx 1$ when $x \approx l$, and $f \approx 0$ when x is far from l. When θ_j has a positive value, the classifier will predict 1 for points sufficiently close to the landmark point $l^{(j)}$.

But how do we choose the landmark points l? And what other similarity functions work?

One choice of how to choose l is to use $l^{(i)} = x^{(i)}$ for i from 1 to m. So you would have as many features as training examples (plus 1 for θ_0), and the data matrix would become a similarity matrix. So $X_{ij} = \text{similarity}\big(x^{(i)}, x^{(j)}\big)$.

This is a lot of features, but it's way fewer than you would get by adding many polynomial features.

An additional detail: the regularization term $\frac{1}{2}\sum_{i=1}^{m} \theta_j^2 = \theta^T\theta$ is typically replaced by $\theta^T M \theta$ where M is some matrix that depends on the kernel. This is done mostly for computational efficiency.

Kernels could be used for other learning methods, but it turns out to be more efficient for SVMs than for something like logistic regression.

7.3 SVM hyperparameters

Choosing C trades off between bias and variance

Choosing σ^2 causes features f_i to vary more smoothly, resulting in higher bias and lower variance.

7.4 Using an SVM

Use a SVM software package to solve for parameters θ

Need to specify C and the kernal. No kernel is called a "linear kernel". This is okay if n is large, or m is small. If you choose a Gaussian kernel, you must also specify σ^2.

You should perform feature scaling before using the Gaussian kernel; otherwise, larger-valued features will dominate in $\|x - l\|$.

Any kernel must satisfy a condition called "Mercer's Theorem". A second example is the polynomial kernel: $k(x, l) = (x^T l + \text{constant})^{\text{degree}}$, but this generally performs worse than the Gaussian kernel. However, sometimes it makes sense for non-negative data. More rare:

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

7.4.1 Multi-class Classification

One option is the one-vs-all method (train K SVMs for K classes, get $\theta^{(1)} \dots \theta^{(K)}$, and pick the class with the largest $\theta^T x$).

The other option is just to use the built-in multi-class utility in the software package you're using.

7.4.2 Logistic Regression vs. SVM

- If $n \gg m$ use logistic regression or a SVM with a linear kernel
- If n is small and m is intermediate (10-10,000), use a SVM with a Gaussian kernel
- If n is small and m is large (50,000+), add more features and use logistic regression or an SVM with a linear kernel

Neural networks will probably work well for many of these scenarios, but may be slower to train.

SVMs have the benefit of convexity, so they will always find a global minimum.

Chapter 8

Week 8

8.1 Clustering

A type of unsupervised learning problem in which we identify distinct groups among our training examples.

8.1.1 K-Means Algorithm

Inputs: the number of clusters K and the data X ($\{x^{(1)}, \ldots, x^{(m)}\}$).

1. Randomly initialize K cluster centroids $\mu_1, \ldots, \mu_K$
2. Cluster assignment step: assign each data point to the closest centroid. $c^{(i)}$:= index of closest cluster centroid.
3. Move centroid step: set each new centroid at the mean value of the data points assigned to it.
4. Repeat steps 2 and 3 until the centroids no longer move in step 3.

If a centroid ends up with no points assigned to it, you can either re-initialize it, or drop it (to end up with $K-1$ clusters).

Example for non-separated clusters: t-shirt sizing.

8.1.2 Optimization Objective

Reminder: two sets of variables to keep track of:

$c^{(i)} \in \{1, 2, \ldots, K\}$ = the index of the cluster to which $x^{(i)}$ is assigned.

$\mu_k \in \mathbb{R}^n$ = the centroid of cluster k.

Additional notation: $\mu_{c^{(i)}}$ is the centroid of the cluster to which $x^{(i)}$ has been assigned.

So now we can write the cost function:

$$J\Big(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K\Big) = \frac{1}{m} \sum_{i=1}^{m} \left\| x^{(i)} - \mu_{c^{(i)}} \right\|^2$$

And we want to find

$$\min_{c,\mu} J(c, \mu)$$

J is sometimes called the "Distortion" of the K-means algorithm.

8.1.3 Random Initialization

One initialization technique is to randomly choose K actual training examples. This is popular and effective.

If you are unlucky with your initialization, K-means can get stuck in local optima. This can be resolved by running the entire thing multiple times with different initializations. Running it 50-1000 times would be pretty typical. Then pick the clustering that has the lowest distortion J.

8.1.4 Choosing the number of clusters

There is not a good way to automate this; doing it by hand is generally accepted as the best.

One "automatic" method is the elbow method. Plot J as a function of K, and choose the value for K at which the cost starts going down more slowly. The problem is, there often is not a clear elbow.

Another choice is to choose pragmatically based on how well it serves some downstream purpose. If you are selling t-shirts, $K = 3$ (small, medium, large) or $K = 5$ (add XS and XL) might be good.

8.2 Dimensionality Reduction

Convert data X where $x \in \mathbb{R}^n$ to Z where $z \in \mathbb{R}^k$, and $k < n$. This helps for compressing data; 100-dimensional data takes up much less disk space than 1000-dimensional data. It also helps for visualization; you cannot visualize 50-dimensional data, but you can visualize 3 or 2-dimensional data!

8.2.1 Principal Component Analysis

We want to find the $k < n$ vectors $u^{(1)}, \ldots, u^{(k)} \in \mathbb{R}^n$ which minimize the projection error. By convention, since we just need directions, we will get normalized vectors.

1. Perform mean normalization (always) and feature scaling (usually). That is, replace each $x_j^{(i)}$ with $\frac{x_j - \mu_j}{\sigma_j}$.
2. Compute the covariance matrix $\Sigma = \frac{1}{m} X^T X = \frac{1}{m} \sum_{i=1}^{n} \left(x^{(i)}\right)\left(x^{(i)}\right)^T$
3. Compute the "eigenvectors" (really the singular values) of Σ: `[U,S,V] = svd(Sigma);`. Since Σ is symmetric positive-semidefinite, this will give the same result as `eig(Sigma)`, but `svd` is more numerically stable. $Sigma \in \mathbb{R}^{n \times n}$, and $U \in \mathbb{R}^{n \times n}$.
4. The columns of U are the directions u that we want; take the first k columns $U_{\text{reduce}} \in \mathbb{R}^{n \times k}$.
5. The data $z \in \mathbb{R}^k$ can be found by $z = U_{\text{reduce}}^T x$

Implementation:

```
% Principal Component
    Analysis
Sigma = (1/m) * X' * X;
[U,S,V] = svd(Sigma);
Ureduce = U(:, 1:k);

Z = X * Ureduce;
z = Ureduce' * x;
```

Dimensions:

- $X \in \mathbb{R}^{m \times n}$
- $\Sigma \in \mathbb{R}^{n \times n}$
- $U \in \mathbb{R}^{n \times n}$
- $U_{\text{reduce}} \in \mathbb{R}^{n \times k}$
- $Z \in \mathbb{R}^{m \times k}$
- $x \in \mathbb{R}^n$
- $z \in \mathbb{R}^k$

8.2.2 Reconstruction from Compressed Representation

Go from $z \in \mathbb{R}^k \to x \in \mathbb{R}^n$ with $x \approx x_{\text{approx}} = U_{\text{reduce}} z$

8.2.3 Choosing the Number of Principal Components

The number K is a parameter of the PCA algorithm. Given the average squared projection error is given by:

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

And the total variation in the data is given by:

$$\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)}\right\|^2$$

We can choose k such that 99% of variance is retained. That is, choose the smallest k which satisfies:

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - x_{\text{approx}^{(i)}}\right\|^2}{\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)}\right\|^2} \leq 0.01$$

So really, rather than choosing k, it is more common to choose the amount of variance we want to retain.

One way to satisfy this is to keep increasing k until it no longer satisfies the requirement. Yikes!

A better way is to examine the diagonal of S (the singular values of the covariance matrix). The trace of S is the total amount of variance. Increase k until the first k diagonal entries over the total is greater than the threshold.

$$\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{m} S_{ii}} \geq 0.99$$

8.2.4 Advice for applying PCA

Speeding up supervised learning:

Rather than training X against y, train Z against y. To make predictions, use $z_{test} = U_{reduce}^{T} x_{test}$, and run the model on z_{test}. This should be faster.

Bad use: using PCA to reduce overfitting. It works okay, but is much less effective than regularization. PCA does not account for the labels y, so when used this way, it might throw away useful information!

Another bad use: reducing dimension without trying the same problem on the original data. If it's not necessary, don't do it!

Chapter 9

Week 9

9.1 Anomaly Detection

Given $\left\{x^{(1)}, \ldots, x^{(m)}\right\}$, is x_{test} anomalous? Flag as an anomaly if $p(x_{test}) < \epsilon$.

Examples: fraud detection, manufacturing, monitoring servers in a data center

9.1.1 Gaussian Distribution

Say $x \in \mathbb{R}$. If x has a Gaussian distribution with mean μ and variance σ^2, then we say $x \sim N(\mu, \sigma^2)$.

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Given a dataset, find μ and σ^2:

$$\mu \approx \bar{x} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \quad \sigma^2 \approx s = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu\right)^2$$

(really the coefficient should be $\frac{1}{m-1}$ in the latter formula, but for large m it makes little difference)

9.1.2 Anomaly Detection Algorithm

Given a data set $\left\{x^{(1)}, \ldots, x^{(m)}\right\}$ where each $x \in \mathbb{R}^n$,

$$p(x) = p(x_1; \mu_1, \sigma_1^2)p(x_2; \mu_2, \sigma_2^2)\ldots p(x_n; \mu_n, \sigma_n^2) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$$

We assume $x_j \sim N(\mu_j, \sigma_j^2)$, and that all x_j are independent. They are almost definitely not independent, but the algorithm works okay anyway.

Algorithm:

1. Choose features $x_i \in \mathbb{R}^n$
2. Fit parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$ using maximum likelihood estimators
3. Given a new example x, compute $p(x)$ using the product above
4. Classify as anomalous if $p(x) < \epsilon$

Since we are just using the value of the probability density function, this doesn't have a whole lot of meaning. But it works.

9.1.3 Anomaly Detection System

In order to evaluate the algorithm (for choosing features, etc.), we want a real-valued metric. Assume we have labeled data, with $y = 1$ indicating an anomaly. Assume the training set is "normal". Assume also that the cross-validation and tests sets have some normal data and some anomalies. Ensure that the anomalies are split between the cross-validation and test sets.

Possible evaluation metrics:

- True positive, false positive, false negative, true negative
- Precision/Recall
- F_1-score

You can choose ϵ to maximize your metric on the cross-validation set. This should be monotonic, so a binary search should work well.

9.1.4 Anomaly Detection vs. Supervised Learning

If we have labeled data, why not just use supervised learning?

Anomaly detection:

- Very small number of positive Examples
- Large number of negative example
- Many different "types" of anomalies; maybe future examples would be different from ones we've already seen

Supervised learning:

- Large number of positive and negative examples
- Enough positive examples for the algorithm to get a sense of what positive examples are like in general

9.1.5 Choosing What Features to Use

Plot histograms to check if a feature is at least vaguely gaussian, or can be transformed to be gaussian.

Example transformations:

- $x \to \log(x + c)$
- $x \to x^{1/c}$

Error analysis: run the algorithm; examine the incorrect classifications; use that examination as inspiration to develop new features.

In particular, choose features that might take on unusually large values if something unusual is going on.

9.1.6 The Multivariate Gaussian Distribution

This has some advantages relative to the version with the independence assumption, and some disadvantages. In particular, it handles colinear features much better by accounting for covariance.

Parameters: a mean vector $\mu \in \mathbb{R}^n$, and a covariance matrix $\Sigma \in \mathbb{R}^{n \times n}$.

$$p(x;\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}\det(\Sigma)^{1/2}}\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

where

$$\mu = \frac{1}{m}\sum_{i=1}^{m}x^{(i)} \quad \Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)}-\mu)(x^{(i)}-\mu)^T$$

This is the same as the original model, given the constraint that Σ is diagonal.

Disadvantaves of the multivariate model: computationally more expensive than the original model, and does not scale as well to large n (because of the matrix inverse). Furthermore, for the multivariate model you must have $m > n$ (ideally $m > 10n$)— otherwise Σ is not invertible, where the original model has no such constraint.

9.2 Recommender Systems

Consider a set of users who watched a set of movies:

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)
Love at Last	5	5	0	0
Romance Forever	5	-	-	0
Cute Puppies of Love	-	4	0	-
Nonstop Car Chases	0	0	5	4
Swords vs. Karate	0	0	5	-

So Alice and Bob appear to like romance, while Carol and Dave like action.

Notation:

- n_u = number of users
- n_m = number of movies
- $r(i,j) = 1$ if user j has rated movie i
- $y^{(i,j)}$ = rating given by user j to movie i

Given this incomplete dataset, predict the missing values.

9.2.1 Content-based Recommendation

Suppose you have a set of n features describing each movie in addition to the ratings:

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1	x_2
Love at Last	5	5	0	0	0.90	0.00
Romance Forever	5	-	-	0	1.00	0.01
Cute Puppies of Love	-	4	0	-	0.99	0.00
Nonstop Car Chases	0	0	5	4	0.10	1.00
Swords vs. Karate	0	0	5	-	0.00	0.90

One option is to treat predicting ratings for each user as a separate linear regression problem. For each user j, learn a set of parameters $\theta^{(j)} \in \mathbb{R}^{n+1}$. Predict user j as rating movie i with $(\theta^{(j)})^T x^{(i)}$ stars.

In other words, (with $m^{(j)}$ = number of movies rated by person j), we learn $\theta^{(j)}$ with:

$$\min_{\theta^{(j)}}\frac{1}{2}\sum_{i:r(i,j)=1}\left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{k=1}^{n}\left(\theta_k^{(j)}\right)^2$$

And to learn all of $\theta^{(1)}, \ldots, \theta^{(n_u)}$ we apply a simple variation where our total cost includes the cost for each user:

$$\min_{\theta^{(1)},\ldots,\theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(\theta_k^{(j)} \right)^2$$

The gradient descent update for this looks like:

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right) x_k^{(i)} + \lambda \theta_k^{(j)} \right)$$

(where $\theta_0 = 0$ so we do not regularize the constant term)

9.2.2 Collaborative Filtering

The previous example assumed that you are provided with x_1 (romance) and x_2 (action). Now, assume instead that each user (i) has rated their own enjoyment of romance and action movies $(\theta^{(i)})$

$$\theta^{(1)} = \begin{pmatrix} 0 \\ 5 \\ 0 \end{pmatrix} \quad \theta^{(2)} = \begin{pmatrix} 0 \\ 5 \\ 0 \end{pmatrix} \quad \theta^{(3)} = \begin{pmatrix} 0 \\ 0 \\ 5 \end{pmatrix} \quad \theta^{(4)} = \begin{pmatrix} 0 \\ 0 \\ 5 \end{pmatrix}$$

So again, Alice and Bob enjoy romance while Carol and Dave enjoy action. From here, it is possible to infer the values of x_1 and x_2 for a given movie $x^{(i)}$:

$$\text{Find } x^{(1)} \text{ such that } \left(\theta^{(1)} \right)^T x^{(1)} \approx 5$$

More formally, given $\theta^{(1)}, \ldots, \theta^{(n_u)}$ to learn $x^{(i)}$:

$$\min_{x^{(i)}} \frac{1}{2} \sum_{j:r(i,j)=1} \left(\left(\theta^{(j)} \right)^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} \left(x_k^{(i)} \right)^2$$

So we minimize the sum over all users j that have rated movie i of the squared error from the actual rating $y^{(i,j)}$.

More generally, given $\theta^{(1)}, \ldots, \theta^{(n_u)}$ to learn $x^{(1)}, \ldots, x^{(n_m)}$:

$$\min_{x^{(i)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} \left(\left(\theta^{(j)} \right)^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} \left(x_k^{(i)} \right)^2$$

This is a simple change: both terms in the cost now sum over all movies.

We can make predicitons in both directions: given $x^{(1)}, \ldots, x^{(n_m)}$ we can estimate $\theta^{(1)}, \ldots, \theta^{(n_u)}$, and likewise given $\theta^{(1)}, \ldots, \theta^{(n_u)}$ we can estimate $x^{(1)}, \ldots, x^{(n_m)}$. Now, we can take an iterative approach. Given an initial guess θ, can can predict:

$$\theta \to x \to \theta \to x \to \theta \to x \to \ldots$$

And this actually works, but it is not yet ideal.

9.2.3 Collaborative Filtering Algorithm

Instead of alternating between predicting θ and x, we can throw them into the same objective function.

$$J\Big(x^{(1)},\dots,x^{(n_m)},\theta^{(1)},\dots,\theta^{(n_u)}\Big)=\frac{1}{2}\sum_{(i,j):r(i,j)=1}\Big((\theta^{(j)})^Tx^{(i)}-y^{(i,j)}\Big)^2+\frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}\Big(x_k^{(i)}\Big)^2+\frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\Big(\theta_k^{(i)}\Big)^2$$

And now, we minimize J over all x and θ. If you hold x constant then you are solving the first problem where we were given features for each movie. If you hold θ constant then you are solving the second problem where we were given parameters for each user.

By convention, in this case, we will dispense with the constant terms, so $x,\theta\in\mathbb{R}^n$. If the algorithm wants a constant feature, it can learn one.

The gradient will have terms for both x and θ.

For this recommendation algorithm, initialize x and θ to small random values to break symmetry and ensure that the algorithm learns $x^{(i)}$ that are different from one another.

Our gradient descent steps will look like this:

$$x_k^{(i)}:=x_k^{(i)}-\alpha\left(\sum_{j:r(i,j)=1}\Big((\theta^{(j)})^Tx^{(i)}-y^{(i,j)}\Big)\theta_k^{(j)}+\lambda x_k^{(i)}\right)$$

$$\theta_k^{(i)}:=x_k^{(i)}-\alpha\left(\sum_{j:r(i,j)=1}\Big((\theta^{(j)})^Tx^{(i)}-y^{(i,j)}\Big)x_k^{(j)}+\lambda\theta_k^{(i)}\right)$$

Where the terms in parentheses are $\frac{\partial}{\partial x_k^{(i)}}J(x,\theta)$ and $\frac{\partial}{\partial\theta_k^{(i)}}J(x,\theta)$

For a user j with parameters θ and a movie i with features x, predict a rating of θ^Tx (assuming $r(i,j)=0$, otherwise we already know the answer!).

9.2.4 Low-Rank Matrix Factorization

Again, given the movie ratings dataset:

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)
Love at Last	5	5	0	0
Romance Forever	5	-	-	0
Cute Puppies of Love	-	4	0	-
Nonstop Car Chases	0	0	5	4
Swords vs. Karate	0	0	5	-

This can be rewritten as a matrix $Y\in\mathbb{R}^{n_m\times n_u}$:

$$Y=\begin{pmatrix}5&5&0&0\\5&?&?&0\\?&4&0&?\\0&0&5&4\\0&0&5&0\end{pmatrix}$$

Meanwhile our predictions can be rewritten as a matrix where each entry is given by $(\theta^{(j)})^T(x^{(i)})$

$$\begin{pmatrix} (\theta^{(1)})^T x^{(1)} & (\theta^{(1)})^T x^{(1)} & \cdots & (\theta^{(n_u)})^T x^{(1)} \\ (\theta^{(1)})^T x^{(2)} & (\theta^{(1)})^T x^{(2)} & \cdots & (\theta^{(n_u)})^T x^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ (\theta^{(1)})^T x^{(n_m)} & (\theta^{(1)})^T x^{(n_m)} & \cdots & (\theta^{(n_u)})^T x^{(n_m)} \end{pmatrix}$$

If we let X be the matrix whose rows are the feature vectors for each movie and let Θ be the matrix whose rows are the parameter vectors for each user, then

$$X = \begin{pmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(n_m)})^T \end{pmatrix} \quad \Theta = \begin{pmatrix} (\theta^{(1)})^T \\ \vdots \\ (\theta^{(n_u)})^T \end{pmatrix}$$

$$\text{Predicted Ratings} = X\Theta^T$$

The algorithm we've dealt with is called low-rank matrix factorization. For each product/movie i, we learn a feature vector $x^{(i)} \in \mathbb{R}^n$. How do we find movies j related to movie i? Well, two movies are similar if $\left\|x^{(j)} - x^{(i)}\right\|$ is small.

9.2.5 Mean Normalization

Consider a user who has not rated any movies:

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	Eve (5)
Love at Last	5	5	0	0	-
Romance Forever	5	-	-	0	-
Cute Puppies of Love	-	4	0	-	-
Nonstop Car Chases	0	0	5	4	-
Swords vs. Karate	0	0	5	-	-

And recall our optimization objective:

$$J(x,\theta) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} \left(x_k^{(i)}\right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(\theta_k^{(i)}\right)^2$$

Then the first term in this cost function will be 0 for Eve, since she has not rated any movies. Only the last regularization term (the sum over θ) will contribute, so θ will be set to 0. Consequently, we will predict that Eve will rate every movie as a 0.

Let μ be the vector of means for each movie:

$$Y = \begin{pmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{pmatrix} \quad \mu = \begin{pmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{pmatrix}$$

And instead of working with Y, work with $Y := Y - 1^T\mu$

$$Y = \begin{pmatrix} 2.5 & 2.5 & -2.5 & -2.5 & ? \\ 2.5 & ? & ? & -2.5 & ? \\ ? & 2 & -2 & ? & ? \\ -2.25 & -2.25 & 2.75 & 1.75 & ? \\ -1.25 & -1.25 & 3.75 & -1.25 & ? \end{pmatrix}$$

So instead of predicting the ratings, we will predict how far far above its mean rating user j will rate movie i. And now, to get back to real ratings, we will predict

$$(\theta^{(j)})^T(x^{(i)}) + \mu_i$$

For Eve, $(\theta^{(j)})^T(x^{(i)})$ is still 0, so we predict that she will rate every movie with its mean rating.

You can vary this to account for movies with no ratings (by doing mean normalization on columns), but this is less important; you probably do not want to recommend such a movie.

9.2.6 Vectorized Implementations

Let P be the predicted ratings and let R be the matrix which is 1 only if the rating exists.

$$P = X\Theta^T$$

$$J = \frac{1}{2}\sum\left((P \odot R - Y)^T(P \odot R - Y)\right) + \frac{\lambda}{2}\sum X \odot X + \frac{\lambda}{2}\sum \Theta \odot \Theta$$

$$\frac{\partial}{\partial X} = (P \odot R - Y)\Theta + \lambda X$$

$$\frac{\partial}{\partial \Theta} = (P \odot R - Y)^T X + \lambda \Theta$$

Chapter 10

Week 10

10.1 Gradient Descent with Large Datasets

One way to build a strong classifier is to train a low-bias learning algorithm and train it on a large amount of data. But gradient descent typically involves summation over m entries, so when m is very large, this becomes computationally very expensive. Sanity check: you can also start by training on a subset of the data. Learning curves (J vs m) can tell you whether training on the full data will improve performance.

10.1.1 Stochastic Gradient Descent

Batch Gradient Descent: each update uses all m training examples. Using linear regression as the example:

$$h_\theta(x^{(i)}) = \theta^T x \qquad J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \qquad \theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

Now, write the cost for an individual data point:

$$\text{cost}(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}\Big(h_\theta(x^{(i)}) - y^{(i)}\Big)^2 \qquad J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\text{cost}(\theta, (x^{(i)}, y^{(i)})) \qquad \frac{\partial}{\partial\theta_j}\text{cost}(\theta, x, y) = (h_\theta(x) - u)x$$

For Stochastic Gradient Descent:

1. Randomly shuffle the dataset (just in case there is an ordering)
2. Repeat:
   (a) for $i \in 1, \ldots, m$:
      i. $\theta_j = \frac{\partial}{\partial\theta_j}\text{cost}(\theta, x^{(i)}, y^{(i)})$ for $j \in 1, \ldots, n$

In other words, compute the gradient for the first data point, and update θ. Then compute the gradient for the second data point, and update θ, and so on. Stochastic gradient descent does not take as direct a path toward the global minimum; it tends to wander. But each update is much faster.

10.1.2 Mini-Batch Gradient Descent

Mini-batch gradient descent falls in between the batch and stochastic algorithms. Rather than using all m examples or only 1 example per update, we will use b examples, with $1 < b < m$. Typical values for b range from 2–100. We still shuffle the dataset, and use b examples at a time, going through the entire data set.

This can perform better than stochastic gradient descent, because good vectorization can make the summations run more efficiently.

10.1.3 Convergence

For batch gradient descent, we can verify it is working by ensuring that J decreases on each iteration.

For stochastic gradient descent, we can do the same verification by ensuring that the cost for the training example is lower after the update compared to before. Plotting $\text{cost}(\theta, x, y)$ averaged over the last 1000 (or so) examples should show a plot that generally appears to converge. Changing this number can make the trend more apparent. If the plot appears to diverge, use a smaller α.

If we want θ to be more certain to converge, you can decrease α over time:

$$\alpha = \frac{\text{const1}}{\text{iterationNumber} + \text{const2}}$$

But this is often not done because fiddling with the constants takes time.

10.2 Online Learning

Online learning handles the scenario where there is a continuous stream of data coming in which should refine the algorithm's predictions.

1. Repeat forever:
   (a) Get (x, y) corresponding to a single new data point
   (b) Update θ using (x, y)
      i. $\theta_j := \theta_j - \alpha(h_\theta(x) - y)x \forall j \in 1, \ldots, n$

In this case, there is no fixed training set - just one example at a time. If the nature of the data changes over time, the algorithm will adapt.

This applies well to predicted click-through-rates, showing special offers on websites, etc.

10.3 Map Reduce and Data Parallelism

Each machine, in parallel, trains on a fraction of the data to compute a gradient descent update. Each calculates its own value for the gradient, and then they are averaged on a master machine. If there were no network latency, this could speed up processing by as much as N times, where N is the number of machines. Whenever the bulk of a learning algorithm can be expressed as a sum over a large data set, then map-reduce can be a good way to speed up the work.

Exploiting multiple cores on a single machine can yield similar benefits. Some numerical linear algebra libraries will do this automatically.