1 Linear algorithms

1.1 Definitions

def. Regression problem - our goal is to predict a **continuous** valued output.
def. Classification problem - our goal is to predict a **discrete** valued output.
def. Supervised learning - we have data set, which matches some response to the input data. (Having the idea that there is a relationship between the input and the output.)
def. Unsupervised learning - we have a data set and we don't know what to do with it. Our unsupervised learning should highlight some structure, **clusters**. So this is called a **clustering algorithm**. As an example of the application of this algorithm, we can consider the separation of conversation from music and music from conversation, this is called the **cocktail party problem algorithm**.
It can be written in one line:

$$[W, s, v] = \text{svd}((\text{repmat}(\text{sum}(x. * x, 1), \text{size}(x, 1), 1). * x) * x');$$

1.2 What is machine learning ?

Example: playing checkers.
E = the **experience** of playing many games of checkers
T = the **task** of playing checkers
P = the **probability** that the program will win the next game

1.3 What about development environment?

You need to use Octave or Matlab, because you can write code in one line in them, as shown in paragraph 1.1.

1.4 Some notations

1. m - number of training examples
2. x's - input features
3. y's - output target variable

(x, y) - one training example

$(x^{(i)}, y^{(i)}) - \text{i}^{\text{th}}$ training example

$h_\Theta(x) = h(x) = \Theta_0 + \Theta_1 \cdot x$ - **linear regression** or **univariate linear regression**
h is called a hypothesis, $h : X \to Y$

1.5 Cost function (linear regression)

Idea: Choose Θ_0, Θ_1 so that $h_\Theta(x)$ is close to y for our training examples (x, y)
Squared error (function):

$$\frac{1}{2m} \cdot \sum_{i=1}^{m} \left(h_\Theta(x^{(i)}) - y^{(i)} \right)^2 = \mathbb{J}(\Theta_0, \Theta_1) \xrightarrow[\Theta_0\Theta_1]{} min, h_\Theta(x^{(i)}) = \widehat{y^{(i)}} \tag{1}$$

Why $\frac{1}{2}$? The mean is halved as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the 0.5 term.
We can simplify it and set $\Theta_0 = 0$, then $\mathbb{J}$ is a function of one variable.

Then let's take a data set of the following pairs $(1,1);(2,2);(3,3)$. We will take different values $\Theta_1 = 0, 0.5, 1$ in our simplified formula. We will substitute them into the function for h, and then for $\mathbb{J}$. As a result, we will know the values of $\mathbb{J}$ and will be able to plot $\mathbb{J}$. From the constructed graph, the minimum value is visible, and therefore the answer.

def. A **contour plot** is a graph that contains many contour lines. A contour line of a two variable function has a constant value at all points of the same line.

If we have both parameters involved, then this graph can be drawn in **3D**:

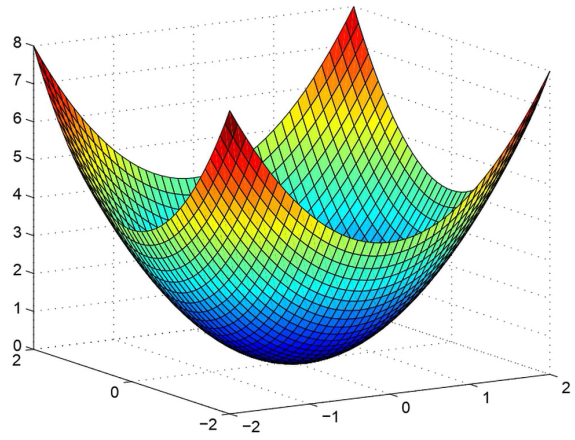

Figure 1: Cost function in 3D

1.6 Gradient Descent (algorithm)

In the previous paragraph, we drew a lot of graphs and built a lot of functions. Now we want to understand how to quickly find these parameters in which the value of the function $\mathbb{J}$ reaches its minimum. Then we can build a perfect straight line to find the answer.

A brief idea of the algorithm:

- Start with some Θ_0, Θ_1(maybe $= 0$).
- Keep changing Θ_0, Θ_1 to reduce $\mathbb{J}(\Theta_0, \Theta_1)$ until we hopefully end up at a minimum. (or local minimum)

Now maths, repeat until convergence:

$$\theta_j := \theta_j - \alpha \cdot \frac{\partial}{\partial \theta_j} \cdot \mathbb{J}(\theta_0, \theta_1), j = 0..1 \tag{2}$$

For example (simultaneous update):

$$\text{temp0} := \theta_0 - \alpha \cdot \frac{\partial}{\partial \theta_0} \cdot \mathbb{J}(\theta_0, \theta_1)$$
$$\text{temp1} := \theta_1 - \alpha \cdot \frac{\partial}{\partial \theta_1} \cdot \mathbb{J}(\theta_0, \theta_1)$$
$$\theta_0 := \text{temp0} \text{ and } \theta_1 := \text{temp1}$$

Minor clarification:
α - this is **learning rate**, determines the step size of our algorithm, if α is very large, then gradient descent is very aggressive.
For example, you can again consider a function from one variable and draw a parabola. If we look at the right branch, then our derivative is > 0 and we move to the left with step α, otherwise vice versa. The α parameter is very important, because if it is very small, then the algorithm will be very slow, if it is large, then we will jump over the minimum or even begin to **diverge**.
And as gradient descent runs, you will automatically take smaller and smaller steps, therefore, it is not necessary to reduce the α.

1.7 Gradient Descent for linear regression

Let's combine gradient descent and regression together:

$$\frac{\partial}{\partial\theta_j}\mathbb{J}(\theta_0,\theta_1) = \frac{\partial}{\partial\theta_j}\cdot\frac{1}{2m}\sum_{i=1}^{m}\left(h_\Theta(x^{(i)}) - y^{(i)}\right)^2 = \frac{\partial}{\partial\theta_j}\cdot\frac{1}{2m}\sum_{i=1}^{m}\left(\Theta_0 + \Theta_1 x^{(i)} - y^{(i)}\right)^2 \tag{3}$$

Take partial derivatives:

$$j = 0 : \frac{\partial}{\partial\theta_0}\mathbb{J}(\theta_0,\theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\Theta(x^{(i)}) - y^{(i)}\right)$$

$$j = 1 : \frac{\partial}{\partial\theta_1}\mathbb{J}(\theta_0,\theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\Theta(x^{(i)}) - y^{(i)}\right)\cdot x^{(i)}$$

In total, we have the transition of our first algorithm:

$$\theta_0 := \theta_0 - \alpha\cdot\frac{1}{m}\sum_{i=1}^{m}\left(h_\Theta(x^{(i)}) - y^{(i)}\right) \tag{4}$$

$$\theta_1 := \theta_1 - \alpha\cdot\frac{1}{m}\sum_{i=1}^{m}\left(h_\Theta(x^{(i)}) - y^{(i)}\right)\cdot x^{(i)} \tag{5}$$

Function for linear regression is always **convex** function. Another name for this algorithm is "Batch" Gradient Descent. **"Batch"**: each step of gradient descent uses all the training examples. Now we know how to implement gradient descent for linear regression.

1.8 Linear Algebra Review

How to use linear algebra in Matlab:

```
A = [1, 2, 3; 4, 5, 6; 7, 8, 9; 10, 11, 12]
v = [1;2;3]
[m,n] = size(A)
dim_A = size(A)
dim_v = size(v)
A_23 = A(2,3)
```

```
s = 2
add_AB = A + B
sub_AB = A - B
mult_As = A * s
div_As = A / s
add_As = A + s
```

Matrices in machine learning: $\text{A} = \begin{bmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1534 \\ 1 & 852 \end{bmatrix}$, $h_\theta(x) = -40 + 0.25 \cdot x \Rightarrow$ result is $A \cdot \begin{bmatrix} -40 \\ 0.25 \end{bmatrix}$

Identity matrix in Matlab, transposed and inversed matrix:

```
A = [2, 4; 5, 6]
% Initialize a 2 by 2 identity matrix
I = eye(2)
% The above notation is the same as I = [1,0;0,1]
% Inversed matrix
pinv(A)
inv(A)
% Transposed matrix
transpose(A)
A'
```

2 Linear Regression with Multiple Variables, Octave, Matlab

The basic skills of using Matlab can be found here: `https://www.coursera.org/learn/machine-learning/supplement/Mlf3e/more-octave-matlab-resources`
Here you can find Matlab online: `https://matlab.mathworks.com/`

2.1 Multivariate Linear Regression

Notations:
n = number of features
$x^{(i)}$ = input (features) of i^{th} training example
$x_j^{(i)}$ = value of feature j in i^{th} training example

Hypothesis:
$h_\Theta(x) = \Theta_0 + \Theta_1 x_1 + \cdots + \Theta_n x_n$

For **convenience** of notation, define $x_0 = 1$. This is an additional zero feature for each training example: $x_0^{(i)} = 1$.

Matrix notation:

$$\mathbb{X} = \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1}, \ \Theta = \begin{bmatrix} \Theta_0 \\ \Theta_1 \\ \cdots \\ \Theta_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

Now you can write like this:

$$h_\Theta(x) = \Theta_0 x_0 + \Theta_1 x_1 + \cdots + \Theta_n x_n = \Theta^T \mathbb{X} \tag{6}$$

2.2 Gradient Descent For Multiple Variables

The algorithm is very similar to the one we have already passed, so I'll just write down the transition formula:

$$\text{repeat until convergence: } \theta_j := \theta_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \left(h_\Theta(x^{(i)}) - y^{(i)} \right) \cdot x_j^{(i)} \quad \text{for } j := 0 \dots n$$

2.3 What can be done to make the algorithm work faster?

Idea: Let's make sure that all the features belong to the same range, for example $-1 \leq x_i \leq 1$, then the algorithm will converge to the global minimum much faster. Our elongated contour ellipses will become circles.

Mean normalization: Replace x_i with $x_i - \mu_i$ to make features have approximately zero mean (Do not apply to $x_0 = 1$). For example, all features: $-0.5 \leq x_i \leq 0.5$. General formula: $x_i = \frac{x_i - \mu_i}{s_i}$, where μ_i - average value of x_i in training set, s_i - range, max - min (or standard deviation).

Debugging: How to make sure that the descent works correctly? Let's plot $\mathbb{J}(\theta)$ from the number of iterations (100-400). It should decrease like hyperbole. From the graphics, you can also understand when it has become close to the horizontal line, so you can correctly choose the number of iterations to convergence.

Automatic convergence test: Declare convergence if $\mathbb{J}(\theta)$ decreases by less than $\epsilon = 10^{-3}$ in one iteration.

2.4 Polynomial regression

Perhaps a quadratic or cubic function is better suited than just a straight line, so you can modify the multivariate linear regression to a polynomial.

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 = \theta_0 + \theta_1 (var) + \theta_2 (var)^2 + \theta_3 (var)^3$$

And just in this algorithm, it is very important to adjust the values to the same range. Perhaps it is better not to switch to the cubic function, but to replace the quadratic term with the square root: $\sqrt{var}$.

2.5 Computing Parameters Analytically

Normal equation: Method to solve for θ analytically.

$$\theta \in \mathbb{R}^{n+1} \quad \mathbb{J}(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \cdot \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

$$\frac{\partial}{\partial \theta_j} \mathbb{J}(\theta) = \dots = 0 \text{ (for every j)}$$

Solve for $\theta_0, \theta_1, \dots, \theta_n$

Let's solve:

$$\mathbb{X}_{m \times (n+1)} = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix} \quad y_{m \times 1} = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

Solution: $\theta = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T y$, $\mathbb{X}$-design matrix, $\mathbb{X}_i = (x^{(i)})^T$.
In this method, there is **no need** to use features scaling.
Problem: what if the inverse matrix doesn't exist? There are two functions in Matlab: *inv* and *pinv*. The latter will work even if the matrix is not reversible. There are two possible reasons: **redundant features** (linear dependent) or too **many features** ($m \leq n$) $\Rightarrow$ delete some features or use regularization (later in this course).

3 Implementation

3.1 Implementation of the gradient descent algorithm with one variable

```
% load data (comma separated)
data = load('ex1data1.txt');
X = data(:, 1);
y = data(:, 2);

% plot
plotData(X,y);

% initialization
m = length(X);
X = [ones(m,1),data(:,1)];
theta = zeros(2, 1);
iterations = 1500;
alpha = 0.01;

% Compute theta, run gradient descent
theta = gradientDescent(X, y, theta, alpha, iterations);

% Print theta to screen
fprintf('Theta computed from gradient descent:\n%f,\n%f\n',theta(1),theta(2));

fprintf('J-function value: %f\n', computeCost(X, y, theta));

hold on; % keep previous plot visible
plot(X(:,2), X*theta, '-')
legend('Training data', 'Linear regression')
hold off % don't overlay any more plots on this figure

% Predict value for population size of 70,000
predict = [1, 7] * theta;
fprintf('For population = 70,000, we predict a profit of %f\n', predict*10000);

% drawing a graph of our data
function plotData(x, y)
figure;
plot(x, y, 'rx', 'MarkerSize', 5);
ylabel('Profit in $10,000s');
xlabel('Population of City in 10,000s');
end

% Gradient Descent function
function [theta, J_history] = gradientDescent(X, y, theta, alpha, num_iters)
m = length(y);
J_history = zeros(num_iters, 1);

for iter = 1 : num_iters
    sum_theta1 = 0;
    sum_theta2 = 0;
    for i = 1 : m
```

```
        sum_theta1 = sum_theta1 + (theta' * X(i, :)' - y(i)) * X(i, 1);
        sum_theta2 = sum_theta2 + (theta' * X(i, :)' - y(i)) * X(i, 2);
    end
    theta1 = theta(1) - alpha / m * sum_theta1;
    theta2 = theta(2) - alpha / m * sum_theta2;
    theta = [theta1; theta2];
end
end

% J-function
function J = computeCost(X, y, theta)
m = length(y);
J = 0;
for i = 1 : m
    J = J + (theta' * X(i, :)' - y(i)) ^ 2;
end
J = J / (2 * m);
end
```

3.2 Visualizing $\mathbb{J}(\theta)$

```
% Grid over which we will calculate J
theta0_vals = linspace(-10, 10, 100);
theta1_vals = linspace(-1, 4, 100);

% Initialize J_vals to a matrix of 0's
J_vals = zeros(length(theta0_vals), length(theta1_vals));

% Fill out J_vals
for i = 1 : length(theta0_vals)
    for j = 1 : length(theta1_vals)
      t = [theta0_vals(i); theta1_vals(j)];
      J_vals(i, j) = computeCost(X, y, t);
    end
end

% Surface plot
J_vals = J_vals';
figure;
surf(theta0_vals, theta1_vals, J_vals)
xlabel('\theta_0'); ylabel('\theta_1');

% Contour plot
figure;
% Plot J_vals as 15 contours spaced logarithmically between 0.01 and 100
contour(theta0_vals, theta1_vals, J_vals, logspace(-2, 3, 20))
xlabel('\theta_0'); ylabel('\theta_1');
hold on;
plot(theta(1), theta(2), 'rx', 'MarkerSize', 10, 'LineWidth', 2);
hold off;
```

3.3 Implementation of the gradient descent algorithm with two variable

```
% Load Data
data = load('ex1data2.txt');
X = data(:, 1:2);
y = data(:, 3);
m = length(y);
```

```
6
7  % First 10 examples from the dataset
8  fprintf(' x = [%.0f %.0f], y = %.0f \n', [X(1:10,:) y(1:10,:)]');
9  % Scale features and set them to zero mean
10 [X, mu, sigma] = featureNormalize(X);
11
12 % Add intercept term to X
13 X = [ones(m, 1) X];
14
15 % Feature Normalize
16 function [X_norm, mu, sigma] = featureNormalize(X)
17 X_norm = X;
18 mu = zeros(1, size(X, 2));
19 sigma = zeros(1, size(X, 2));
20 cols = size(X, 2);
21 for col = 1:cols
22     cur_mean = mean(X(:, col));
23     cur_std = std(X(:, col));
24     X_norm(:, col) = (X(:, col) - cur_mean) / cur_std;
25     mu(1, col) = cur_mean;
26     sigma(1, col) = cur_std;
27 end
28 end
29
30 % J-function multi
31 function J = computeCostMulti(X, y, theta)
32 m = length(y);
33 sqr_term = (X * theta - y);
34 J = sqr_term' * sqr_term / (2 * m);
35 end
```

$$\mathbb{J}(\theta) = \frac{1}{2m} \cdot (X\theta - \overrightarrow{y})^T \cdot (X\theta - \overrightarrow{y}), \text{ vectorized form of cost function} \tag{7}$$

```
1  % Gradient Descent
2  function [theta, J_history] = gradientDescentMulti(X, y, theta, alpha, num_iters)
3  m = length(y);
4  J_history = zeros(num_iters, 1);
5  cols = size(X, 2);
6
7  for iter = 1 : num_iters
8      sum_theta = zeros(cols, 1);
9      for i = 1 : m
10         for col = 1 : cols
11             sum_theta(col, 1) = sum_theta(col, 1) +
12             (theta' * X(i, :)' - y(i)) * X(i, col);
13         end
14     end
15     for col = 1 : cols
16         theta(col, 1) = theta(col, 1) - alpha / m * sum_theta(col, 1);
17     end
18     J_history(iter) = computeCostMulti(X, y, theta);
19 end
20 end
21
22 % Run gradient descent
23 alpha = 0.1;
24 num_iters = 400;
25
26 % Init Theta and Run Gradient Descent
27 theta = zeros(3, 1);
28 [theta, ~] = gradientDescentMulti(X, y, theta, alpha, num_iters);
29
```

```
% Display gradient descent's result
fprintf('Theta computed from gradient descent:\n%f\n%f\n%f',
theta(1),theta(2),theta(3))
% Estimate the price of a 1650 sq-ft, 3 br house
size_of_flat = (1650 - mu(1)) / sigma(1);
amount_of_rooms = (3 - mu(2)) / sigma(2);
price = theta' * [1; size_of_flat; amount_of_rooms];
fprintf('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent):
\n $%f', price);
```

3.4 Implementation of Normal Equations

```
% Solve with normal equations:
% Load Data
data = csvread('ex1data2.txt');
X = data(:, 1:2);
y = data(:, 3);
m = length(y);

% Add intercept term to X
X = [ones(m, 1) X];

% Calculate the parameters from the normal equation
theta = normalEqn(X, y);

% Display normal equation's result
fprintf('Theta computed from the normal equations:\n%f\n%f\n%f',
theta(1),theta(2),theta(3));

% Estimate the price of a 1650 sq-ft, 3 br house
price = theta' * [1; 1650; 3];
fprintf('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent):
\n $%f', price);

% Normal Equations function
function [theta] = normalEqn(X, y)
theta = pinv(X' * X) * X' * y;
end
```

3.5 Basic Matlab features

```
xor(1, 0) % xor
1 ~= 2 % not equals
PS1('>> ') % Octave function to change prompt style
a = 3; % semicolon supressing output

a = pi;
disp(a) % just print
disp(sprintf('%0.8f', a)) % C-style output = 3.14159265

% set 3.141592653589793 or 3.1416 by default output
format long;
format short;

% start from 1, end with 10, step is 0.5
v = 1:0.5:10
% generates vector with step 1
v_step_one = 1:10
```

```
m = ones(4, 5) * 4 % matrix of 4 (zeros - of 0)

% random matrix of random numbers: 0 <= x <= 1
m = rand(3, 3)

% Gaussian distribution with mean zero and std equal to one
g = randn(2, 2)

% Example of Gaussian distribution
w = -6 + sqrt(10) * (randn(1, 10000));
hist(w, 100)
```

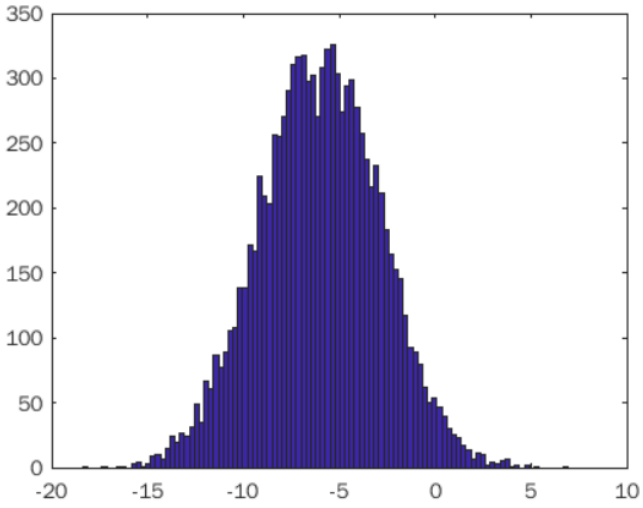

Figure 2: Gaussian distribution

3.6 Working with data

```
% size of matrix
rows = size(x, 1)
cols = size(x, 2)

length(x) % the biggest dimension of x

% loading data
load ex1data1.txt
% or <=>
load('ex1data1.txt')

who % prints all variables in our workspace
whos % detail info

clear var % delete variable var

% saving data
v = rand(1, 5)
% compressed binary format
save res.mat v;

```

```
% format for human, you can read it
save res.txt v -ascii;

% select some rows and some cols
A([1 3], [4 5])

% append column [1; 2; 3] to the end of matrix A
A = [A, [1; 2; 3]];

A(:) % convert matrix into vector of matrix's elements

% pow corresponding elements
% C(1, 1) = A(1, 1) ^ B(1, 1)
C = A .^ B

% similarly we can do
X = 1 ./ A
% A .* X = ones
% similarly log(A), exp(A), abs(A), -A

% max, min
v = rand(1, 5) * 4
[val, ind] = max(v)
% or just max(v)

b = v < 10 % element-wise operation
find(b) % returns indexes which satisfy condition

res = magic(4)
% returns a matrix where sums over all columns, rows, diagonals are equal
% to one number
[r, c] = find(res >= 7)

a = [1, 2, 3]
% sum(a), prod(a), floor(a), ceil(a)

% element-wise maximum in two random matrices
max(rand(3, 3), rand(3, 3))

max(A, [], 1) % columns-wise max
max(A, [], 2) % rows-wise max

max(max(A)) % max(matrix) or
max(A(:))

sum(A, 1) % columns sums
sum(A, 2) % rows sums

sum(sum(A .* eye(3))) % main diag sum
sum(sum(A .* flipud(eye(3)))) % secondary diag sum
flipud(A) % turns the matrix upside down
```

3.7 Plotting data

```
t = [0:0.01:0.98];
y1 = sin(2*pi*4*t);
plot(t, y1);

y2 = cos(2*pi*4*t);
plot(t, y1);
hold on;
plot(t, y2, 'r'); % 'r' - red color
```

```
xlabel('time');
ylabel('value');
legend('sin', 'cos');
title('sin and cos');
print -dpng 'sin_and_cos.png'; % save your plot
```

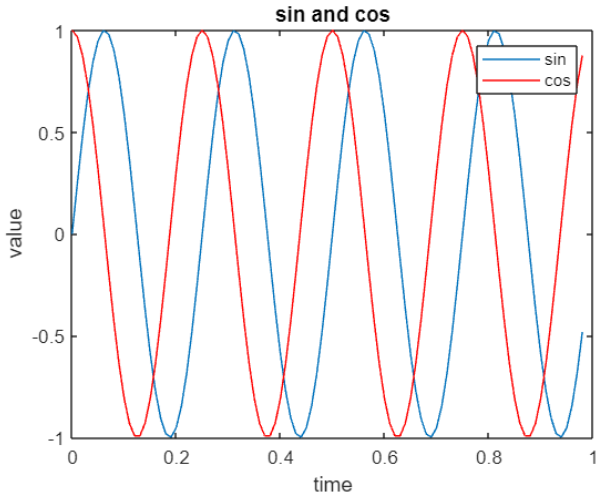

Figure 3: Plotting data

```
close; % closes all opened figures

% If you want two different figures:
figure(1); plot(t, y1);
figure(2); plot(t, y2);

% What if I want many plots on one image?
% I can divide my screen into grid!
subplot(1, 3, 1); % divide 1x3 grid and access first
plot(t, y1);
subplot(1, 3, 2); % access second
plot(t, y2);
subplot(1, 3, 3); % access third
plot(t, y1);

% How to change axis?
axis([-10 10 -1 1]);
% set x from -10 to 10
% set y from -1 to 1

clf; % clear figure, not delete

% I want to print my matrix
% Where for different elements different colors
A = [1 2 3; 3 2 1; 4 5 1];
% imagesc(A);
imagesc(A), colorbar, colormap autumn;
% second is more complicated command
% it plots matrix, sets color map to autumn
% and prints color bar on the right of our plot
% here is an example:
```

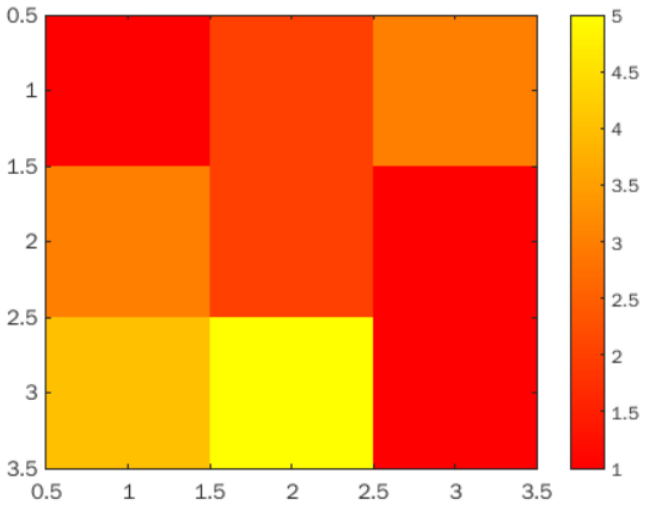

Figure 4: Plotting matrix

3.8 Control Statements

```
% `for` loop
vals=1:2:7;
for i=vals
    disp(i);
end

% `while` loop
% `break`, `continue` statements
i = 1;
while i <= 10
    disp(i);
    i = i + 1;
    if i == 6
        disp("end")
        break
    end
end

% `if`-`else` statement
x=1;
if x==1, disp("one")
elseif x==2, disp("two")
else, disp("another")
end

pwd, cd, addpath % Octave functions
% current dir, move to another dir, add path to PATH

% definition of functions
% all data, that returns function
[ans1, ans2] = SquareAndCubeFunction(5);
fprintf('%d, %d\n', ans1, ans2);
% only first variable, that returns function
fprintf('%d\n', SquareAndCubeFunction(5));

```

```
% function that returns square and cube
function [square, cube] = SquareAndCubeFunction(x)
square = x ^ 2;
cube = x ^ 3;
end
```

3.9 Vectorization

Vectorization is the idea of using functions from standard libraries. This reduces the code, increases its readability. These functions are highly optimized and can use parallelism, so the code will run much faster and more efficiently.

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

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) * x_j^{(i)} \Leftrightarrow$$

$$\Theta := \Theta - \alpha \frac{1}{m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)}) * x^{(i)}$$

4 Logistic Regression

4.1 Classification and Representation

4.1.1 Classification

Email: Spam / Not Spam ?
Online transactions: Fraudulent (Yes / No) ?
Tumor: Malignant / Benign ?
$y \in \{0, 1\}$
0: "Negative Class" (benign tumor)
1: "Positive Class" (malignant tumor)

Threshold classifier output $h_\theta(x)$ at 0.5: (linear regression)
if $h_\theta(x) \geq 0.5$, predict "y = 1"
if $h_\theta(x) < 0.5$, predict "y = 0"
This method can work on a specific example, but in most cases it will work extremely poorly.
It may even happen that $h_\theta(x)$ can be > 1 or < 0.
Therefore, we will develop a **logistic regression** algorithm: $0 \leq h_\theta(x) \leq 1$.

4.1.2 Hypothesis Representation

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

$g(z) = \dfrac{1}{1 + e^{-z}}$ - sigmoid (logistic) function

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

$h_\theta(x)$ = estimated probability that y = 1 on input x, or in the language of mathematics: $h_\theta(x) = \mathbb{P}(y = 1|x; \theta)$.

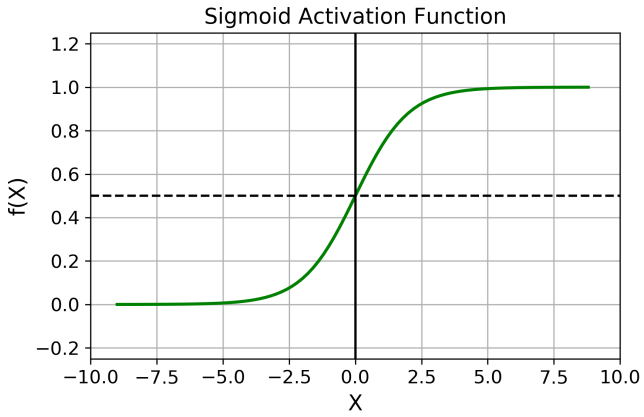

Figure 5: Sigmoid function

The relationship of probabilities:

$$\mathbb{P}(y = 0|x; \theta) + \mathbb{P}(y = 1|x; \theta) = 1 \Leftrightarrow$$
$$\mathbb{P}(y = 0|x; \theta) = 1 - \mathbb{P}(y = 1|x; \theta)$$

4.1.3 Decision Boundary

How to understand what we predicted 0 or 1: $g(z) \geq 0.5 \Leftrightarrow z \geq 0 \Leftrightarrow \theta^T x \geq 0$. Accordingly, if we reverse all the signs, then we predict 0. By writing $\theta^T x$, we can build a figure on the graph that separates our sets. (line, parabola, circle) This equation is called **decision boundary**.

4.2 Logistic Regression Model

4.2.1 Cost Function

If we leave our cost function as it was in linear regression, then given that our hypothesis is very complex, we get a non-convex function. This function will have many local optima.

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if y} = 1 \\ -\log(1 - h_\theta(x)) & \text{if y} = 0 \end{cases} \tag{8}$$

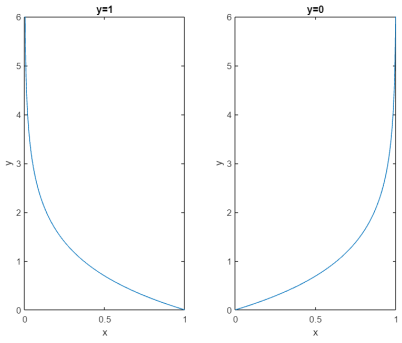

Figure 6: Cost Function for Logistic Regression

4.2.2 Simplified Cost Function and Gradient Descent

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

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

Vector form of writing our formulas:

$$h = g(\mathbb{X}\theta)$$
$$\mathbb{J}(\theta) = -\frac{1}{m} \cdot \left(y^T \log(h) + (1-y)^T \log(1-h)\right)$$
$$\theta := \theta - \frac{\alpha}{m} \cdot \mathbb{X}^T \left(g(\mathbb{X}\theta) - \overrightarrow{y}\right)$$

To simplify the cost function, we merged this system into one formula, so it's easier to count and calculate the derivative. It turned out to be a very interesting coincidence: the gradient descent formulas coincided for linear regression and logistic regression. Vector formulas are obvious and do not require explanation.

4.2.3 Advanced Optimization

Optimization algorithms:

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

Advantages:

- No need to manually pick α. In fact, they have a clever inter-loop called a **line search algorithm** that automatically tries out different values for the learning rate α and automatically picks a good learning rate α so that it can even pick a different learning rate for every iteration. And so then you don't need to choose it yourself.
- Often faster than gradient descent

Disadvantages:

- More complex

To use these advanced algorithms, you need to implement a special function:

```
function [jVal, gradient] = costFunction(theta)
jVal = (theta(1)-5)^2+(theta(2)-5)^2;
gradient = zeros(2, 1);
gradient(1) = 2*(theta(1)-5);
gradient(2) = 2*(theta(2)-5);
```

Now we can call this function: (exitFlag = did the algorithm converge?!)

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

4.3 Multiclass Classification

4.3.1 Multiclass Classification: One-vs-all

Email foldering/tagging: Work, Friends, Family, Hobby
Medical diagrams: Not ill, Cold, Flu
Weather: Sunny (y=1), Cloudy (y=2), Rain (y=3), Snow (y=4)

How will the algorithm work? We sort through our sets in order. We take the current set and denote it as a positive class, and everything else as a negative class. We run logistic regression on this data and repeat the same with another set. Each logistic regression classifier we denote as $h_\theta^{(i)}(x)$. We fit n classifiers: $h_\theta^{(i)}(x) = \mathbb{P}(y = i|x; \theta)$.

On a new input x, to make a prediction, pick the class i that maximizes $\max_i \left(h_\theta^{(i)}(x)\right)$.

5 Regularization

5.1 Solving the Problem of Overfitting

5.1.1 The Problem of Overfitting

Let's say we have data that can be depicted on a plane as part of a function x^4. We can train a linear model $\theta_0 + \theta_1 x$, but then our predictions will not correspond to reality at all accurately. This problem is called **underfitting**, sometimes it is said that the algorithm has **high bias**. If we train a quadratic model, then it will perfectly fit our data. If we take more parameters, for example, a function of the fourth degree, then even if our curve passes through all the given ones, it will still be **wavy**. They say about it like this: algorithm has **high variance** or problem **overfitting** (too many features). Even if the value of the $\mathbb{J}$ function is approximately 0, then everything will be bad in the new examples.

Options:

- Reduce number of features.
  - Manually select which features to keep.
  - Model selection algorithm (later in course).
- Regularization.
  - Keep all the features, but reduce magnitude / values of parameters θ_j.
  - Works well when we have a lot of features, each of which contributes a bit to predicting y.

5.1.2 Cost Function

Suppose we penalize and make θ_3, θ_4 really small:

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

There may be a situation when it is not clear which parameters we want to compress. Then we have to compress all the parameters, for this we will add another term to our cost function.

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

This is regularization term, and λ is regularization parameter. The function is smoother.

5.1.3 Regularized Linear Regression

It is important to remember that we compress all θ values **except** θ_0.

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

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

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

The term $1 - \alpha\frac{\lambda}{m} \approx 0.99$, so it shrinks θ_j on each iteration.
Normal equation:

$$\Theta = \left(\mathbb{X}^T\mathbb{X} + \lambda\cdot\begin{bmatrix}0 & 0 & \dots & 0\\ 0 & 1 & \dots & 0\\ 0 & 0 & 1 & \dots\\ 0 & \dots & 0 & 1\end{bmatrix}_{(n+1)\times(n+1)}\right)^{-1}\cdot\mathbb{X}^T y$$

Interesting fact: if $\lambda > 0$, then matrix is **invertible** (this can be proven). What is written in square brackets is the corresponding partial derivative.

5.1.4 Regularized Logistic Regression

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

The transition formulas for gradient descent are the same as for linear regression.

6 Implementation

6.1 Required Functions

6.1.1 Cost Function, Gradient

```
function [J, grad] = costFunction(theta, X, y)
m = length(y);
h = sigmoid(X * theta);
J = -1 / m * (y' * log(h) + (1 - y)' * log(1 - h));
grad = X' * (h - y) / m;
end
```

6.1.2 Plotting Data

```
function plotData(X, y)
figure;
hold on;
pos = find(y == 1);
neg = find(y == 0);
plot(X(pos, 1), X(pos, 2), 'k+','LineWidth', 2, 'MarkerSize', 7);
```

```
plot(X(neg, 1), X(neg, 2), 'ko', 'MarkerFaceColor', 'y','MarkerSize', 7);
hold off;
end
```

6.1.3 Prediction

```
function p = predict(theta, X)
p = X * theta >= 0;
end
```

6.1.4 Sigmoid

```
function g = sigmoid(z)
g = 1 ./ (1 + exp(-z));
end
```

6.1.5 Map Feature

```
function out = mapFeature(X1, X2)
degree = 6;
out = ones(size(X1(:,1)));
for i = 1:degree
    for j = 0:i
        out(:, end + 1) = (X1 .^ (i - j)) .* (X2 .^ j);
    end
end
end
```

6.1.6 Plotting Decision Boundary

```
function plotDecisionBoundary(theta, X, y)
plotData(X(:, 2:3), y);
hold on
if size(X, 2) <= 3
    plot_x = [min(X(:, 2)),  max(X(:, 2))];
    plot_y = -(theta(2) * plot_x + theta(1)) / theta(3);
    plot(plot_x, plot_y)
    legend('Admitted', 'Not admitted', 'Decision Boundary')
    axis([30, 100, 30, 100])
else
    u = linspace(-1, 1.5, 50);
    v = linspace(-1, 1.5, 50);
    z = zeros(length(u), length(v));
    for i = 1:length(u)
        for j = 1:length(v)
            z(i,j) = mapFeature(u(i), v(j)) * theta;
        end
    end
    z = z';
    contour(u, v, z, [0, 0], 'LineWidth', 2)
end
hold off
```

```
23 end
```

6.1.7 Regularized Cost Function, Gradient

```
1  function [J, grad] = costFunctionReg(theta, X, y, lambda)
2  m = length(y);
3  n = size(theta, 1);
4  h = sigmoid(X * theta);
5  s = 0;
6  for i=2:n
7      s = s + theta(i) ^ 2;
8  end
9  add = lambda / (2 * m) * s;
10 J = -1 / m * (y' * log(h) + (1 - y)' * log(1 - h)) + add;
11 grad = X' * (h - y) / m;
12 shift = lambda / m * theta;
13 grad(2:end) = grad(2:end) + shift(2:end);
14 end
```

6.2 Launching

6.2.1 First part

```
1  data = load('ex2data1.txt');
2  X = data(:, [1, 2]);
3  y = data(:, 3);
4
5  plotData(X, y);
6  xlabel('Exam 1 score')
7  ylabel('Exam 2 score')
8  legend('Admitted', 'Not admitted')
9
10 [m, n] = size(X);
11 X = [ones(m, 1) X];
12 initial_theta = zeros(n + 1, 1);
13
14 options = optimoptions(@fminunc,'Algorithm','Quasi-Newton','GradObj', 'on',
15 'MaxIter', 400);
16 [theta, cost] = fminunc(@(t)(costFunction(t, X, y)), initial_theta, options);
17
18 plotDecisionBoundary(theta, X, y);
19 hold on;
20 xlabel('Exam 1 score')
21 ylabel('Exam 2 score')
22 legend('Admitted', 'Not admitted')
23 hold off;
24
25 prob = sigmoid([1 45 85] * theta);
26 fprintf('For a student with scores 45 and 85, we predict an admission probability
27 of %f\n\n', prob);
28
29 p = predict(theta, X);
30 fprintf('Train Accuracy: %f\n', mean(double(p == y)) * 100);
```

6.2.2 Second part

```
data = load('ex2data2.txt');
X = data(:, [1, 2]); y = data(:, 3);

plotData(X, y);
hold on;
xlabel('Microchip Test 1')
ylabel('Microchip Test 2')
legend('y = 1', 'y = 0')
hold off;

X = mapFeature(X(:, 1), X(:, 2));

initial_theta = zeros(size(X, 2), 1);
lambda = 1;
[cost, grad] = costFunctionReg(initial_theta, X, y, lambda);

initial_theta = zeros(size(X, 2), 1);
lambda = 0; % try: 0, 1, 10, 100
options = optimoptions(@fminunc, 'Algorithm', 'Quasi-Newton', 'GradObj', 'on',
'MaxIter', 1000);
[theta, J, exit_flag] = fminunc(@(t)(costFunctionReg(t, X, y, lambda)),
initial_theta, options);

plotDecisionBoundary(theta, X, y);
hold on;
title(sprintf('lambda = %g', lambda))
xlabel('Microchip Test 1')
ylabel('Microchip Test 2')
legend('y = 1', 'y = 0', 'Decision boundary')
hold off;

p = predict(theta, X);
fprintf('Train Accuracy: %f\n', mean(double(p == y)) * 100);
```

7 Neural Networks: Representation

7.1 Motivations

7.1.1 Non-linear Hypotheses

Why do we need neural networks, because we have linear and logistic regression? Let's say we have 100 parameters, then in order to include all the quadratic terms in the hypothesis, we will need about ≈ 5000 features. This is bad for two reasons. Firstly, it is expensive in memory and time, and secondly, most likely we will retrain. As a solution to the problem, we can leave only a subset, but then we reduce our capabilities.

In order to train the classifier, we need to show him a lot of photos with our object and without our object. To recognize photos, we need a nonlinear model, because the dependencies are quite complex. If we have a 50×50 image, that we need 2500 pixels, if we have RGB, then we have 7500 features, which means in total we have 3 million parameters. That's a lot.

7.1.2 Neurons and the Brain

One part of the brain can hear and see and feel, which means that there is some kind of specialized algorithm that can do everything. This is the motivation for neural networks. You can add various sensors and devices to a person that perform some function, then the brain will learn to perform it very quickly.

7.2 Neural Networks

7.2.1 Model Representation 1

Simplified neuron model (logistic unit): $(x_0 = 1), x_1, x_2, \ldots, x_n \longrightarrow \bigcirc \longrightarrow h_\theta(x)$. $x_0 = 1$ - "bias" unit. Hypotheses: $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$. θ - weights. $h_\theta(x)$ - sigmoid (logistic) activation function.

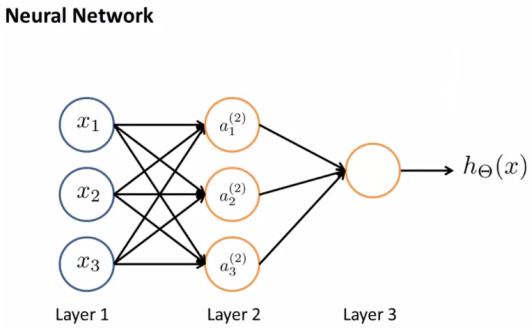

Figure 7: Neural Network Model

Layer 1 - input layer, Layer 3 - output layer, Layer 2 - hidden layer.
$a_i^{(j)}$ = activation of unit i in layer j
$\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer j + 1

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$

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

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

7.2.2 Model Representation 2

Forward propagation: Vectorized implementation
Designations: $a_1^{(2)} = g(z_1^{(2)})$, $a_2^{(2)} = g(z_2^{(2)})$, $a_3^{(2)} = g(z_3^{(2)})$.

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

$$z^{(2)} = \Theta^{(1)} x \quad \text{and} \quad a^{(2)} = g(z^{(2)})$$

If we put $a^{(1)} = x$, then: $z^{(2)} = \Theta^{(1)} a^{(1)}$.
Add $a_0^{(2)} = 1$, $z^{(3)} = \Theta^{(2)} a^{(2)}$, $h_\Theta(x) = a^{(3)} = g(z^{(3)})$.

7.3 Applications

7.3.1 Multiclass Classification

$$h_\Theta(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad h_\Theta(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad h_\Theta(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad \text{etc.}$$

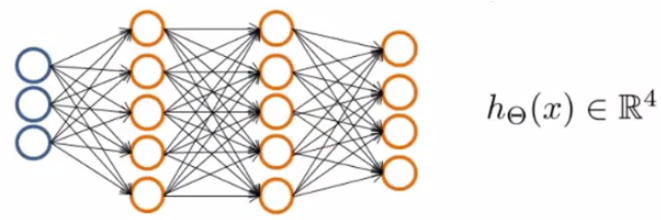

Figure 8: Multiclass Classification Problem

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} \longrightarrow \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ \cdots \end{bmatrix} \longrightarrow \begin{bmatrix} a_0^{(3)} \\ a_1^{(3)} \\ a_2^{(3)} \\ \cdots \end{bmatrix} \longrightarrow \cdots \longrightarrow \begin{bmatrix} h_\Theta(x)_1 \\ h_\Theta(x)_2 \\ h_\Theta(x)_3 \\ h_\Theta(x)_4 \end{bmatrix}$$

7.4 Implementation

7.4.1 All Functions

```
% Function #1
function [h, display_array] = displayData(X, example_width)
if ~exist('example_width', 'var') || isempty(example_width)
    example_width = round(sqrt(size(X, 2)));
end
```

```
colormap(gray);
[m, n] = size(X);
example_height = (n / example_width);
display_rows = floor(sqrt(m));
display_cols = ceil(m / display_rows);
pad = 1;
display_array = - ones(pad + display_rows * (example_height + pad), ...
    pad + display_cols * (example_width + pad));
curr_ex = 1;
for j = 1:display_rows
    for i = 1:display_cols
        if curr_ex > m, break;
        end
        max_val = max(abs(X(curr_ex, :)));
        display_array(pad + (j - 1) * (example_height + pad) + (1:example_height), ...
                      pad + (i - 1) * (example_width + pad) + (1:example_width)) = ...
                        reshape(X(curr_ex, :), example_height, example_width) / max_val;
        curr_ex = curr_ex + 1;
    end
    if curr_ex > m, break;
    end
end
h = imagesc(display_array, [-1 1]);
axis image off
drawnow;
end

% Function #2
function [J, grad] = lrCostFunction(theta, X, y, lambda)
m = length(y);
h = sigmoid(X * theta);
J = -1 / m * (y' * log(h) + (1 - y)' * log(1 - h)) + lambda / (2 * m) * ...
    (theta' * theta - theta(1) ^ 2);
grad = X' * (h - y) / m;
shift = lambda / m * theta;
grad(2:end) = grad(2:end) + shift(2:end);
end

% Function #3
function [all_theta] = oneVsAll(X, y, num_labels, lambda)
m = size(X, 1);
n = size(X, 2);
all_theta = zeros(num_labels, n + 1);
X = [ones(m, 1) X];
for c=1:num_labels
    initial_theta = zeros(n + 1, 1);
    options = optimset('GradObj', 'on', 'MaxIter', 50);
    [theta] = fmincg(@(t)(lrCostFunction(t, X, (y == c), lambda)), ...
        initial_theta, options);
    all_theta(c, :) = theta';
end
end

% Function #4
function p = predict(Theta1, Theta2, X)
m = size(X, 1);
p = zeros(size(X, 1), 1);
X = [ones(m, 1) X];
for i=1:m
    [~, pos] = max(sigmoid(Theta2 * [1; sigmoid(Theta1 * X(i, :)')]));
    p(i) = pos;
end
end

% Function #5
```

```
function p = predictOneVsAll(all_theta, X)
m = size(X, 1);
p = zeros(size(X, 1), 1);
X = [ones(m, 1) X];
for i=1:m
    [~, pos] = max(all_theta * X(i, :)');
    p(i) = pos;
end
end
```

7.4.2 Launch

```
load('ex3data1.mat');

m = size(X, 1);
rand_indices = randperm(m);
sel = X(rand_indices(1:100), :);
displayData(sel);

theta_t = [-2; -1; 1; 2];
X_t = [ones(5,1) reshape(1:15,5,3)/10];
y_t = ([1;0;1;0;1] >= 0.5);
lambda_t = 3;
[J, grad] = lrCostFunction(theta_t, X_t, y_t, lambda_t);
fprintf('Cost: %f | Expected cost: 2.534819\n',J);
fprintf('Gradients:\n'); fprintf('%f\n',grad);
fprintf('Expected gradients:\n 0.146561\n -0.548558\n 0.724722\n 1.398003');

num_labels = 10; % 10 labels, from 1 to 10
lambda = 0.1;
[all_theta] = oneVsAll(X, y, num_labels, lambda);

pred = predictOneVsAll(all_theta, X);
fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);


load('ex3data1.mat');
m = size(X, 1);

sel = randperm(size(X, 1));
sel = sel(1:100);
displayData(X(sel, :));

load('ex3weights.mat');
pred = predict(Theta1, Theta2, X);
fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);

rp = randi(m);
pred = predict(Theta1, Theta2, X(rp,:));
fprintf('\nNeural Network Prediction: %d (digit %d)\n', pred, mod(pred, 10));

displayData(X(rp, :));
```

8 Neural Networks: Learning

8.1 Cost Function and Backpropagation

8.1.1 Cost Function

Designations:

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$$
L = total num. of layers in network
s_l = num. of units (not counting bias unit) in layer l
K = number of classes

Cost function:

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

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

8.1.2 Backpropagation Algorithm

Gradient computation:
We need to compute $\mathbb{J}(\Theta)$ and $\frac{\partial}{\partial \Theta_{ij}^{(l)}}\mathbb{J}(\Theta)$.

Given one training example (x, y):
Forward propagation:

$$\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \ \text{(add } a_0^{(2)}\text{)} \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g(z^{(3)}) \ \text{(add } a_0^{(3)}\text{)} \\
z^{(4)} &= \Theta^{(3)} a^{(3)} \\
a^{(4)} &= h_\Theta(x) = g(z^{(4)})
\end{aligned}$$

Gradient computation: Backpropagation algorithm
Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.
For each output unit (layer L): $\delta_j^{(L)} = a_j^{(L)} - y_j \Leftrightarrow \delta^{(L)} = a^{(L)} - y$.

$$\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} .* g'(z^{(l)}) = (\Theta^{(l)})^T \delta^{(l+1)} .* a^{(l)} .* (1 - a^{(l)})$$

For the first level, we do not consider the error function!

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}}\mathbb{J}(\Theta) = a_j^{(l)} \delta_i^{(l+1)} \ \text{(ignoring regularization)}$$

Backpropagation algorithm:
Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j).
For i = 1 to m:
1. Set $a^{(1)} = x^{(i)}$
2. Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$
3. Using $y(i)$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
4. Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
5. $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)} \iff \Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

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

8.1.3 Backpropagation Intuition

Focusing on a single example $x^{(i)}, y^{(i)}$, the case of 1 output unit, and ignoring regularization ($\lambda = 0$),

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

(Think of $\text{cost}(i) \approx (h_\Theta(x^{(i)}) - y^{(i)})^2$)
Formally (for $j \geq 0$):

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

Example:

$$\delta_2^{(2)} = \Theta_{12}^{(2)} \delta_1^{(3)} + \Theta_{22}^{(2)} \delta_2^{(3)}$$
$$\delta_2^{(3)} = \Theta_{12}^{(3)} \delta_1^{(4)}$$

8.2 Backpropagation in Practice

8.2.1 Implementation Note: Unrolling Parameters

Advanced optimization

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

Neural Network (L=4):

- $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (Theta1, Theta2, Theta3)
- $D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (D1, D2, D3)

"Unroll" into vectors

Example:

$$s_1 = 10, s_2 = 10, s_3 = 1$$
$$\Theta^{(1)} \in \mathbb{R}^{10 \times 11}, \Theta^{(2)} \in \mathbb{R}^{10 \times 11}, \Theta^{(3)} \in \mathbb{R}^{1 \times 11}$$
$$D^{(1)} \in \mathbb{R}^{10 \times 11}, D^{(2)} \in \mathbb{R}^{10 \times 11}, D^{(3)} \in \mathbb{R}^{1 \times 11}$$

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

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

Learning Algorithm
Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
Unroll to get initialTheta to pass to fminunc.

function [jVal, gradientVec] = costFunction(thetaVec)
From thetaVec, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $\mathbb{J}(\Theta)$.
Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get gradientVec.

8.2.2 Gradient Checking

Numerical estimation of gradients

$$\frac{d}{d\Theta}\mathbb{J}(\Theta) \approx \frac{\mathbb{J}(\Theta+\epsilon) - \mathbb{J}(\Theta-\epsilon)}{2\epsilon}$$

Parameter vector θ

$$\begin{aligned}
&\theta \in \mathbb{R}^n \text{ ("unrolled")} \\
&\theta = \theta_1, \theta_2, \ldots, \theta_n \\
&\frac{\partial}{\partial\theta_1}\mathbb{J}(\theta) \approx \frac{\mathbb{J}(\theta_1+\epsilon, \theta_2, \ldots, \theta_n) - \mathbb{J}(\theta_1-\epsilon, \theta_2, \ldots, \theta_n)}{2\epsilon} \\
&\frac{\partial}{\partial\theta_2}\mathbb{J}(\theta) \approx \frac{\mathbb{J}(\theta_1, \theta_2+\epsilon, \ldots, \theta_n) - \mathbb{J}(\theta_1, \theta_2-\epsilon, \ldots, \theta_n)}{2\epsilon} \\
&\ldots \\
&\frac{\partial}{\partial\theta_n}\mathbb{J}(\theta) \approx \frac{\mathbb{J}(\theta_1, \theta_2, \ldots, \theta_n+\epsilon) - \mathbb{J}(\theta_1, \theta_2, \ldots, \theta_n-\epsilon)}{2\epsilon}
\end{aligned}$$

Check that gradApprox ≈ DVec. Once you have verified once that your backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.

8.2.3 Random Initialization

If we initialize $\Theta_{ij}^{(l)} = 0$, then there will be a lot of identical values in our neural network that change by the same amount. After each update, parameters corresponding to inputs going into each of two hidden units are identical.

Random initialization: Symmetry breaking
Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$ (.i.e $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)
Theta1 = rand(10, 11) * (2 * initEpsilon) - initEpsilon
Theta2 = rand(1, 11) * (2 * initEpsilon) - initEpsilon
rand - between 0 and 1

8.2.4 Putting It Together

If there are several hidden layers in our architecture, then it is desirable that they are the same size. This size should be comparable to the number of input and output elements. $\mathbb{J}(\Theta)$ - non-convex function.
One effective strategy for choosing e_{init} is to base it on the number of units in the network. A good choice of e_{init} is $e_{\text{init}} = \frac{\sqrt{6}}{\sqrt{L_{\text{in}}+L_{\text{out}}}}$ where $L_{\text{in}} = s_l$ and $L_{\text{out}} = s_{l+1}$ are the number of units in the layers adjacent to $\Theta^{(l)}$. This range of values ensures that the parameters are kept small and makes the learning more efficient.

8.3 Implementation

8.3.1 Functions

```
% Function #1: Numerical Gradient
function numgrad = computeNumericalGradient(J, theta)
numgrad = zeros(size(theta));
perturb = zeros(size(theta));
e = 1e-4;
for p = 1:numel(theta)
    perturb(p) = e;
    loss1 = J(theta - perturb);
    loss2 = J(theta + perturb);
    numgrad(p) = (loss2 - loss1) / (2*e);
    perturb(p) = 0;
end
end

% Function #2: Regularized Cost Function and Gradient
function [J, grad] = nnCostFunction(nn_params, ...
                                    input_layer_size, ...
                                    hidden_layer_size, ...
                                    num_labels, ...
                                    X, y, lambda)
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
                 hidden_layer_size, (input_layer_size + 1));

Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end), ...
                 num_labels, (hidden_layer_size + 1));
m = size(X, 1);
J = 0;
Theta1_grad = zeros(size(Theta1));
Theta2_grad = zeros(size(Theta2));
for i=1:m
    a1 = [1 X(i, :)]';
    a2 = [1; sigmoid(Theta1 * a1)];
    h = sigmoid(Theta2 * a2);
    vy = size(num_labels, 1);
    for j=1:num_labels
        vy(j, 1) = j;
    end
    vy = (vy == y(i, 1));
    delta_last = h - vy;
    delta_hidden = Theta2' * delta_last .* a2 .* (1 - a2);
    delta_hidden = delta_hidden(2:end);
    Theta1_grad = Theta1_grad + delta_hidden * (a1)';
    Theta2_grad = Theta2_grad + delta_last * (a2)';
    J = J + sum(-1 * vy .* log(h) - (1 - vy) .* log(1 - h));
end
Theta1_grad = Theta1_grad / m;
Theta2_grad = Theta2_grad / m;
```

```
for i=1:size(Theta1_grad, 1)
    for j=2:size(Theta1_grad, 2)
        Theta1_grad(i, j) = Theta1_grad(i, j) + Theta1(i, j) * lambda / m;
    end
end
for i=1:size(Theta2_grad, 1)
    for j=2:size(Theta2_grad, 2)
        Theta2_grad(i, j) = Theta2_grad(i, j) + Theta2(i, j) * lambda / m;
    end
end
J = J / m;
reg = 0;
for i=1:size(Theta1, 1)
    for j=2:size(Theta1, 2)
        reg = reg + Theta1(i, j) ^ 2;
    end
end
for i=1:size(Theta2, 1)
    for j=2:size(Theta2, 2)
        reg = reg + Theta2(i, j) ^ 2;
    end
end
J = J + lambda / (2 * m) * reg;
grad = [Theta1_grad(:) ; Theta2_grad(:)];
end

% Function #3: Random Initial Weights
function W = randInitializeWeights(L_in, L_out)
epsilon_init = 0.12;
W = rand(L_out, 1 + L_in) * 2 * epsilon_init - epsilon_init;
end

% Function #4: Sigmoid Derivate
function g = sigmoidGradient(z)
g = sigmoid(z) .* (1 - sigmoid(z));
end
```

8.3.2 Launching

```
% Part #1: Display Data
load('ex4data1.mat');
m = size(X, 1);
sel = randperm(size(X, 1));
sel = sel(1:100);
displayData(X(sel, :));

% Part #2: Load Weights
load('ex4weights.mat');

% Part #3: Unregularized Cost Function
input_layer_size  = 400;
hidden_layer_size = 25;
num_labels = 10;
nn_params = [Theta1(:) ; Theta2(:)];
lambda = 0;
J = nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels,
X, y, lambda);
fprintf('Cost at parameters (loaded from ex4weights): %f', J);

% Part #4: Regularized Cost Function
lambda = 1;
J = nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels,
```

```
X, y, lambda);
fprintf('Cost at parameters (loaded from ex4weights): %f', J);

% Part #5: Random Initialization
initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size);
initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels);
initial_nn_params = [initial_Theta1(:) ; initial_Theta2(:)];

% Part #6: Learning
options = optimset('MaxIter', 500);
lambda = 1;
costFunction = @(p) nnCostFunction(p, input_layer_size, hidden_layer_size,
num_labels, X, y, lambda);
[nn_params, ~] = fmincg(costFunction, initial_nn_params, options);
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)),
hidden_layer_size, (input_layer_size + 1));
Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end),
num_labels, (hidden_layer_size + 1));
pred = predict(Theta1, Theta2, X);
fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);
```

9 Advice for Applying Machine Learning

9.1 Evaluating a Learning Algorithm

9.1.1 Deciding What to Try Next

Debugging a learning algorithm
Suppose you have implemented regularized linear regression to predict housing prices. However, when you test your hypothesis on a new set of houses, you find that it makes unacceptably large errors in its predictions. What should you try next?

- Get more training examples
- Try smaller sets of features (avoid overfitting)
- Try getting additional features
- Try adding polynomial features (x_1^2, x_2^2, x_1x_2, etc.)
- Try decreasing λ
- Try increasing λ

Machine learning diagnostic:
Diagnostic: A test that you can run to gain insight what is/isn't working with a learning algorithm, and gain guidance as to how best to improve its performance. Diagnostic can take time to implement, but doing so can be a very good use of your time.

9.1.2 Evaluating a Hypotheses

You can take our dataset and divide it into training and test parts, in a ratio of 70% to 30%.

- m_{test} = no. of test examples
- $(x_{\text{test}}^{(i)}, y_{\text{test}}^{(i)})$ - one test

Before separating the data, they need to be randomly mixed.
Training/testing procedure

- Learn parameter θ from training data
- Compute test set error $\mathbb{J}_{\text{test}}(\theta)$
- Misclassification error (0/1 misclassification error)

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

$$\text{Test Error } = \frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \text{err}(h_\Theta(x_{\text{test}}^{(i)}), y_{\text{test}}^{(i)})$$

9.1.3 Model Selection and Train/Validation/Test Sets

Model selection - this is the process of selecting features, the regularization parameter, the degree of the polynomial, etc. Let's denote d = degree of polynomial. Now let's train our models for different d and calculate our error function on them, choose the degree that has the least error. Despite the fact that we are looking at the error on the test set, the model can still generalize poorly.
Solution. Let's divide our dataset into 3 parts: Training set, **Cross Validation set (CV)**, Test set. Now we will choose our degree based on validation data, and evaluate the generalization error based on test data. Ratio: 60% : 20% : 20%.

9.2 Bias vs. Variance

9.2.1 Diagnosing Bias vs. Variance

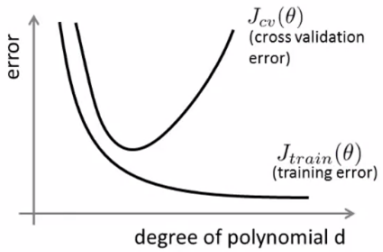

Figure 9: Bias vs. Variance

Is it a bias problem or a variance problem?
Bias (underfit, leftward): train and cv errors are high, and they are approximately equal.
Variance (overfit, on the right): train error is low, but in contrast, cv error is high.
The **optimal** value is in the **middle**.

9.2.2 Regularization and Bias/Variance

Large λ, high bias: $\lambda = 10000, \theta_1 \approx 0, \theta_2 \approx 0, \ldots, h_\theta(x) \approx \theta_0$.
Intermediate λ: "Just right".
Small λ, high variance: $\lambda = 0$.
Choosing the **regularization** parameter λ **automatically**:

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

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

$$\mathbb{J}_{train,cv,test}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Try $\lambda = 0, 0.01, 0.02, 0.04, 0.08, \ldots, 10$.

9.2.3 Learning Curves

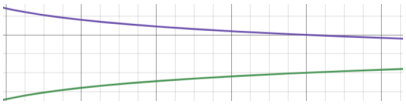

Figure 10: Learning Curves

Let's build such a graph - the dependence of the error on the number of examples. On top - an error on validation data, on the bottom - on test examples. If the curves converge, the difference between the straight lines is small, but the error is big - it's a lack of training. If the difference between the straight lines is large, there is a small error on the training data, and a large one on the validation data, then this is retraining. If you add a lot of test cases, then the lack of training will not be corrected, but retraining will become much better. Similarly with neural networks: few layers - under-training, many - retraining. Also, 2 is computationally more difficult.

9.3 Implementation

9.3.1 Functions

```
% #1
function [J, grad] = linearRegCostFunction(X, y, theta, lambda)
m = length(y);

J = (X * theta - y)' * (X * theta - y) / (2 * m) + ...
    lambda / (2 * m) * (theta' * theta - theta(1, 1) ^ 2);

grad = zeros(size(theta));
for j=1:size(theta, 1)
    sum = 0;
    for i=1:m
        sum = sum + (X(i, :) * theta - y(i, 1)) * X(i, j);
    end
    if j >= 2
        sum = sum + lambda * theta(j, 1);
    end
    grad(j, 1) = sum / m;
end
grad = grad(:);
end

% #2
function [error_train, error_val] = ...
    learningCurve(X, y, Xval, yval, lambda)
m = size(X, 1);
error_train = zeros(m, 1);
error_val   = zeros(m, 1);
for i=1:m
    [theta] = trainLinearReg(X(1:i,:), y(1:i), lambda);
    error_train(i) = linearRegCostFunction(X(1:i,:), y(1:i), theta, 0);
    error_val(i) = linearRegCostFunction(Xval, yval, theta, 0);
end
end

% #3
function [X_poly] = polyFeatures(X, p)
X_poly = zeros(numel(X), p);
for i=1:p
    X_poly(:, i) = X(:, 1) .^ i;
end
end

% #4
function [lambda_vec, error_train, error_val] = ...
    validationCurve(X, y, Xval, yval)
lambda_vec = [0 0.001 0.003 0.01 0.03 0.1 0.3 1 3 10]';
error_train = zeros(length(lambda_vec), 1);
error_val = zeros(length(lambda_vec), 1);
for i=1:length(lambda_vec)
    [theta] = trainLinearReg(X, y, lambda_vec(i, 1));
    error_train(i) = linearRegCostFunction(X, y, theta, 0);
    error_val(i) = linearRegCostFunction(Xval, yval, theta, 0);
end
end
```

9.3.2 Code

```
load ('ex5data1.mat');
m = size(X, 1);
figure;
plot(X, y, 'rx', 'MarkerSize', 10, 'LineWidth', 1.5);
xlabel('Change in water level (x)');
ylabel('Water flowing out of the dam (y)');

theta = [1 ; 1];
J = linearRegCostFunction([ones(m, 1) X], y, theta, 1);
fprintf('Cost at theta = [1 ; 1]: %f', J);

[J, grad] = linearRegCostFunction([ones(m, 1) X], y, theta, 1);
fprintf('Gradient at theta = [1 ; 1]:  [%f; %f] \n',grad(1), grad(2));

lambda = 0;
[theta] = trainLinearReg([ones(m, 1) X], y, lambda);

figure;
plot(X, y, 'rx', 'MarkerSize', 10, 'LineWidth', 1.5);
xlabel('Change in water level (x)');
ylabel('Water flowing out of the dam (y)');
hold on;
plot(X, [ones(m, 1) X]*theta, '--', 'LineWidth', 2)
hold off;

lambda = 0;
[error_train, error_val] = learningCurve([ones(m, 1) X], y,
[ones(size(Xval, 1), 1) Xval], yval, lambda);

plot(1:m, error_train, 1:m, error_val);
title('Learning curve for linear regression')
legend('Train', 'Cross Validation')
xlabel('Number of training examples')
ylabel('Error')
axis([0 13 0 150])

p = 8;
X_poly = polyFeatures(X, p);
[X_poly, mu, sigma] = featureNormalize(X_poly);
X_poly = [ones(m, 1), X_poly];

X_poly_test = polyFeatures(Xtest, p);
X_poly_test = X_poly_test-mu;
X_poly_test = X_poly_test./sigma;
X_poly_test = [ones(size(X_poly_test, 1), 1), X_poly_test];

X_poly_val = polyFeatures(Xval, p);
X_poly_val = X_poly_val-mu;
X_poly_val = X_poly_val./sigma;
X_poly_val = [ones(size(X_poly_val, 1), 1), X_poly_val];

lambda = 0;
[theta] = trainLinearReg(X_poly, y, lambda);

plot(X, y, 'rx', 'MarkerSize', 10, 'LineWidth', 1.5);
plotFit(min(X), max(X), mu, sigma, theta, p);
xlabel('Change in water level (x)');
ylabel('Water flowing out of the dam (y)');
title (sprintf('Polynomial Regression Fit (lambda = %f)', lambda));

[error_train, error_val] = learningCurve(X_poly, y, X_poly_val, yval, lambda);
plot(1:m, error_train, 1:m, error_val);
title(sprintf('Polynomial Regression Learning Curve (lambda = %f)', lambda));
xlabel('Number of training examples')
```

```
ylabel('Error')
axis([0 13 0 100])
legend('Train', 'Cross Validation')

[lambda_vec, error_train, error_val] = validationCurve(X_poly, y, X_poly_val,
yval);
plot(lambda_vec, error_train, lambda_vec, error_val);
legend('Train', 'Cross Validation');
xlabel('lambda');
ylabel('Error');
```

10 Machine Learning System Design

10.1 Building a Spam Classifier

Let's create a list of the most common words in our dataset. (10000-50000 words) We will put 1 if the word occurs and 0 if not. So we made up x and y. It is recommended to write a simple quick solution, and then analyze the result with your eyes and hands, this can help you decide what to do next. Do I need to distinguish between large and small letters in words, do I need to highlight the common root of words? It is very difficult to answer right away, so there is a need for one number that will rate the algorithm.

10.2 Handling Skewed Data

Let's say our test system detects 99 percent of cases of the disease. That is, it has 1% errors. At the same time, only 0.5% of all people actually have the disease. Then our system is actually very bad. Therefore, a different error metric is needed.

Precision/Recall

$y = 1$ in presence of rare class that we want to detect

Actual class | Predicted class

1 | 1 - True Positive

1 | 0 - False Negative

0 | 1 - False Positive

0 | 0 - True Negative

Precision

(Of all patients where we predicted $y = 1$, what fraction actually has cancer?)

$$\text{Precision} = \frac{\text{True pos.}}{\#\text{predicted pos.}} = \frac{\text{True pos.}}{\text{True pos. + False pos.}}$$

Recall

(Of all patients that actually have cancer, what fraction did we correctly detect as having cancer?)

$$\text{Recall} = \frac{\text{True pos.}}{\#\text{actual pos.}} = \frac{\text{True pos.}}{\text{True pos. + False neg.}}$$

Good classifier: Recall and Precision $\to 1$.

How to estimate the algorithm?

Average: $\frac{P+R}{2}$.

F_1Score(F score): $2 \cdot \frac{PR}{P+R}$.

Accuracy $= \frac{\text{True pos. + True neg.}}{\#\text{total number}}$.

11 Support Vector Machines

11.1 Large Margin Classification

11.1.1 Optimization Objective

Alternative view of logistic regression
In logistic regression, we had some kind of curve function for the cost function. Let's simplify it a bit and divide it into two parts: an inclined straight line and a horizontal straight line - a polyline. If $y = 1$, then we denote it $cost_1(z)$, otherwise $cost_0(z)$.

$$min(\theta) \ : \ C \cdot \sum_{i=1}^{m} \left[y^{(i)} cost_1(\theta^T x^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

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

11.1.2 Large Margin Intuition

If $y = 1$, we want $\theta^T x \geq 1$.
If $y = 0$, we want $\theta^T x \leq -1$.
SVM Decision Boundary: Linearly separable case
Other algorithms will separate the two clusters with a line so that the clusters are on different sides of the line. The SVM should divide so that the clusters are at the same distance from the line, that is, the straight line will be better circumscribed. Therefore, this algorithm is sometimes called "Large margin classifier".
Large margin classifier **in presence of outliers**
If we have two well-defined clusters and one outlier, then our algorithm can be very sensitive to this outlier. It doesn't seem right. If C is large, then the algorithm is sensitive, if C is small, then the border will most likely not change. $C \approx \frac{1}{\lambda}$.

11.1.3 Mathematics

Vector Inner Product

$$u^T v = v^T u = p * ||u|| = u_1 v_1 + u_2 v_2$$
$$p = \text{length of projection of v onto u}$$

SVM Decision Boundary
Simplification: $\theta_0 = 0, n = 2$.

$$min \ \frac{1}{2} \sum_{j=1}^{n} \theta_j^2 = \frac{1}{2}(\theta_1^2 + \theta_2^2) = \frac{1}{2} \left(\sqrt{\theta_1^2 + \theta_2^2} \right)^2 = \frac{1}{2} \left(||\theta|| \right)^2$$

$$\theta^T x^{(i)} = p^{(i)} \cdot ||\theta|| = \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)}$$

where $p^{(i)}$ is the projection of $x^{(i)}$ onto the vector θ.
Consider the **first case**. Let our boundary be drawn poorly and practically touches the clusters, then it can be shown that the θ vector will be perpendicular to our boundary, which means that the projections of the tangent clusters on the perpendicular will be very small, then in order for the equation to be fulfilled $p^{(i)} \cdot ||\theta|| \geq 1$, a very large theta is needed, which contradicts the minimized function.
Consider the **second case**. We have drawn a good boundary, and similarly we can show that the θ vector is perpendicular to the boundary. Then the projections are quite large, and therefore θ is quite small, so we will choose exactly such a boundary.

11.2 Kernels

11.2.1 Kernels 1

$$\Theta_0 + \Theta_1 f_1 + \Theta_2 f_2 + \Theta_3 f_3 + \dots$$
$$f_1 = x_1, f_2 = x_2, f_3 = x_1 x_2, f_4 = x_1^2, \dots$$

Given x, compute new feature depending on proximity to landmarks $l^{(1)}, l^{(2)}, l^{(3)}$.
Given x:

$$f_1 = similarity(x, l^{(1)}) = \exp\left(-\frac{||x - l^{(1)}||^2}{2\sigma^2}\right)$$
$$f_2 = similarity(x, l^{(2)}) = \exp\left(-\frac{||x - l^{(2)}||^2}{2\sigma^2}\right)$$
$$f_3 = similarity(x, l^{(3)}) = \exp\left(-\frac{||x - l^{(3)}||^2}{2\sigma^2}\right)$$
$$similarity \Leftrightarrow k$$
$$f_1 = \exp\left(-\frac{\sum_{j=1}^{n}(x_j - l_j^{(1)})^2}{2\sigma^2}\right)$$

k - Kernel (Gaussian Kernel) function
If $x \approx l^{(1)}$: $f_1 \approx \exp(0) \approx 1$.
If x is far from $l^{(1)}$: $f_1 \approx 0$. If the sigma value is small ($\sigma^2 = 0.5$), then our graph will be narrow,

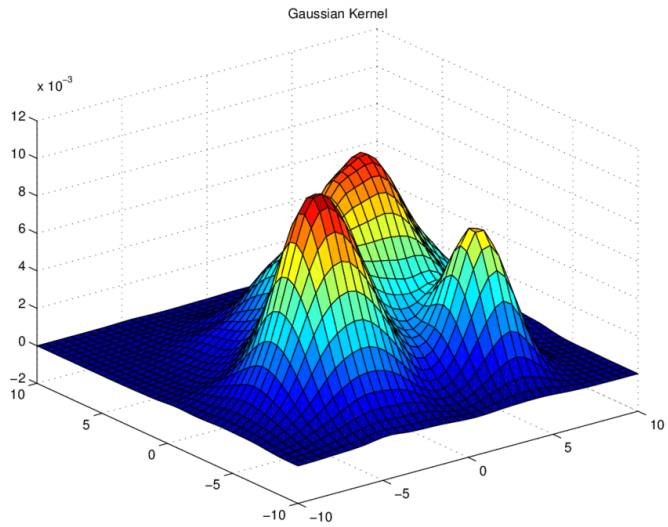

Figure 11: Kernel function

sharp descents. If it is bigger ($\sigma^2 = 3$), then the graph will become wider, gentle descents. At the same time, the maximum of 1 will remain. We will predict "1" when

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

11.2.2 Kernels 2

Given $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$, choose $l^{(1)} = x^{(1)}, \ldots, l^{(m)} = x^{(m)}$.
Given example x:
$f_i = similarity(x, l^{(i)})$

$$f = \begin{bmatrix} f_0 = 1 \\ f_1 \\ \ldots \\ f_m \end{bmatrix}$$

$x^{(i)} \in \mathbb{R}^{n+1} \Rightarrow f^{(i)} \in \mathbb{R}^{m+1}$ and $\Theta \in \mathbb{R}^{m+1}$
Predict "y=1" if $\theta^T f \geq 0$
Training:

$$min(\theta) \ : \ C \cdot \sum_{i=1}^{m} \left[y^{(i)} cost_1(\theta^T f^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T f^{(i)}) \right] + \frac{1}{2} \sum_{j=1}^{m} \theta_j^2$$

If the number of examples is very large (≈ 10000), then in real conditions this formula is used for regularization $\Theta^T \cdot M \cdot \Theta$ instead of this one $\Theta^T \cdot \Theta$. Support vector machine and kernels - very fast and good idea which optimizes well. But logistic regression and kernels is bad idea because it will be computationally expensive. How to optimize this cost function? It's not important, because there are already a lot of perfect software with some kinds of tricks.
SVM parameters:
$C(= \frac{1}{\lambda})$: Large C - lower bias, high variance. Small C - higher bias, low variance.
σ^2 : Large σ^2 - features f_i vary more smoothly. Higher bias, lower variance. Small σ^2 - features f_i vary less smoothly. Lower bias, higher variance (derivatives).

11.3 SVMs in Practice

11.3.1 Using An SVM

Use SVM software package (e.g. liblinear, libsvm, ...) to solve for parameters θ.
Need to specify:

- Choice of parameter C.
- Choice of kernel (similarity function):
  - E.g. No kernel ("linear kernel"). Predict "y=1" if $\theta^T x \geq 0$. n - large, m - small.
  - Gaussian kernel. Need to choose σ^2. n - small, m - large. Complex decision boundary.

Kernel (similarity) functions:
function f = kernel($x1$, $x2$)

$$f = \exp\left(-\frac{||x1 - x2||^2}{2\sigma^2} \right)$$

return
Note: Do perform feature scaling before using the Gaussian kernel.
Other choices of kernel:
Note: Not all similarity functions $similarity(x, l)$ make valid kernels. (Need to satisfy technical condition called "Mercer's Theorem" to make sure SVM packages's optimizations run correctly, and do not diverge).
Many off-the-shelf kernels available:

- Polynomial kernel: $k(x, l) = (x^T l + constant)^{degree}$.
- More esoteric: String, chi-square kernels, ...

If n is large (relative to m) - use logistic regression, or SVM without a kernel (linear).
If n in small, m is intermediate - use SVM with Gaussian kernel.
If n is small, m is large - create/add more features, then use logistic regression or SVM without a kernel.
Neural network likely to work well for most of these settings, but may be slower to train.