Notes

Machine Learning by Andrew Ng on Coursera

Sparsh Jain

October 13, 2020

Contents

Chapter 1

Introduction

Machine learning (task, experience, performance) can be classified into *Supervised* and *Unsupervised* learning.

1.1 Supervised Learning

Supervised learning can be basically classified into *Regression* and *Classification* problems.

1.1.1 Regression Problem

Regression problems work loosely on continuous range of outputs.

1.1.2 Classification Problems

Classification problems work loosely on discrete range of outputs.

1.2 Unsupervised Learning

An example is *Clustering Problem*.

Check Lecture1.pdf for more details.

Chapter 2

Linear Regression with One Variable

2.1 Notations

$$
\begin{aligned}
m &= \text{number of training examples} \\
x\text{'s} &= \text{`input' variables / features} \\
y\text{'s} &= \text{`output' variables / `target' variables} \\
(x, y) &= \text{single training example} \\
(x^{(i)}, y^{(i)}) &= i^{th} \text{ example}
\end{aligned}
$$

2.2 Supervised Learning

We have a data set (*Training Set*).

Training Set $\rightarrow$ Learning Algorithm $\rightarrow$ h (*hypothesis*, a function $X \rightarrow Y$)

To Represent h

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Cost

$$\underset{\theta_0,\ \theta_1}{\text{minimize}} \frac{1}{2m} \sum_{1}^{m} (h_\theta(x) - y)^2$$

Cost Function

Squared Error Cost Function

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

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

2.3 Gradient Descent

Finds local optimum:

1. Start with some value
2. Get closer to optimum

Algorithm

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

where α = learning rate

Important!

Simultaneous Update!

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

2.4 Gradient Descent for Linear Regression

Cost function for linear regression is convex!

Batch Gradient Descent: Each step of gradient descent uses all training examples.

Check Lecture2.pdf for more details.

Chapter 3

Linear Algebra

3.1 Matrix

Rectangular array of numbers:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Dimension of the matrix: #rows x #cols (2 x 3)

Elements of the matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$A_{ij} = \text{“}i, j \text{ entry” in the } i^{th} \text{ row, } j^{th} \text{ col}$$

3.2 Vector

An $n \times 1$ matrix.

$$y = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$y_i = i^{th} \text{ element}$$

Note: Uppercase for matrices, lowercase for vectors.

3.3 Addition and Scalar Multiplication

Add/Subtract (element by element) matrices of same dimention only!

Multiply/Divide (all elements) a matrix by scalar!

3.4 Matrix Matrix Multiplication

$m \times n$ matrix multiplied by $n \times o$ matrix gives a $m \times o$ matrix.

Properties

1. Matrix Multiplication is *not* Commutative.
2. Matrix Multiplication is Associative.
3. *Identity Matrix (I):* 1's along diagonal, 0's everywhere else in an $n \times n$ matrix. $AI = IA = A$.

3.5 Inverse and Transpose

Inverse

Only square ($n \times n$) matrices *may* have an inverse.

$$AA^{-1} = A^{-1}A = I$$

.

Matrices that don't have an inverse are *singular* or *degenerate* matrices.

Transpose

Let A be an $m \times n$ matrix and let $B = A^T$, then

$$B_{ij} = A_{ji}$$

Example:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$B = A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Check Lecture3.pdf for more details.

Chapter 4

Linear Regression with Multiple Variables

4.1 Notations

$$\begin{aligned}
n &= \text{number of features} \\
x^{(i)} &= \text{input (features) of } i^{th} \text{ training example} \\
x_j^{(i)} &= \text{value of feature } j \text{ of } i^{th} \text{ training example}
\end{aligned}$$

4.2 Hypothesis

Previously:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Now:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

For convinience, define $x_0 = 1$. So

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

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

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

4.3 Gradient Descent

$$\begin{aligned} \text{Hypothesis} : h_\theta(x) = \theta^T x & \qquad = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n \\ \text{Parameters} : \theta & \qquad = \theta_0, \theta_1, \ldots, \theta_n \\ \text{Cost Function} : J(\theta) = J(\theta_0, \theta_1, \ldots, \theta_n) & \qquad = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \end{aligned}$$

Gradient Descent :

$$\begin{aligned} &\text{Repeat}\{ \\ \theta_j &:= \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \\ &= \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1, \ldots, \theta_n) \\ &= \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)} - y^{(i)}) x_j^{(i)}) \\ &\}(\text{simultaneously update } \forall j = 0, 1, \ldots, n) \end{aligned}$$

4.3.1 Feature Scaling

Idea: Make sure features are on a similar scale.
Get every feature into approximately a $-1 \leq x_i \leq 1$ range.

4.3.2 Mean Normalization

Replace x_i with $x_i - \mu_i$ to make features have approximately zero mean (Do not apply to $x_0 = 1$).

General Rule

$$x_i \leftarrow \frac{x_i - \mu_i}{S_i}$$

where

$$\begin{aligned}
\mu_i &= \text{average value of } x_i \\
S_i &= \text{range (max - min)} \qquad\qquad\qquad\qquad \textit{or} \\
&= \sigma(\text{standard deviation})
\end{aligned}$$

4.3.3 Learning Rate

$J(\theta)$ should decrease after every iteration. #iterations vary a lot.

Example *Automatic Convergence Test:* Declare convergence if $J(\theta)$ decreases by less than ϵ (say 10^{-3}) in one iteration.

If $J(\theta)$ increases, use smaller α. Too small α means slow convergence.

To choose α, try $\ldots,\ 0.001,\ 0.003,\ 0.01,\ 0.03,\ 0.1,\ 0.3,\ 1,\ \ldots$

4.4 Features and Polynomial Regression

4.4.1 Features

Get an insight in your problem and choose better features (may even combine/separate features).

Ex: size = length $\rightarrow$ breadth.

4.4.2 Polynomial Regression

Ex:

$$\begin{aligned}
x_1 &= size \\
x_2 &= size^2 \\
x_3 &= size^3
\end{aligned}$$

4.5 Normal Equation

Solve for θ analytically!

$$
\begin{aligned}
x^{(i)} &= \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} && \in \mathbb{R}^{n+1} \\
X &= \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix} && \in \mathbb{R}^{m\times(n+1)} \\
&= \begin{bmatrix} x_0 & x_1 & \dots & x_n \end{bmatrix} && \in \mathbb{R}^{m\times(n+1)} \\
y &= \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} && \in \mathbb{R}^{m} \\
\theta &= (X^T X)^{-1} X^T y
\end{aligned}
$$

Inverse of a matrix grows as $O(n^3)$, use wisely.

4.5.1 Non Invertibility of $X^T X$

Use 'pinv' function in Octave (pseudo-inverse) instead of 'inv' function (inverse).

If $X^T X$ is non-invertible, common causes are

1. Redundant features (linearly dependent)
2. Too many features ($m \leq n$). In this case, delete some features or use *regularization*

Check Lecture4.pdf for more details.

Chapter 5

Octave Tutorial

Check Lecture5.pdf for more details.

Chapter 6

Classification

Classify into categories (binary or multiple).

6.1 Logistic Regression

$$
\begin{aligned}
0 \leq h_\theta(x) &\leq 1 \\
h_\theta(x) &= g(\theta^T x) \\
g(z) &= \frac{1}{1+e^{-z}} \qquad g \text{ is called a } \textit{sigmoid} \text{ function or a } \textit{logistic} \text{ function.} \\
h_\theta(x) &= \frac{1}{1+e^{\theta^T x}}
\end{aligned}
$$

Interpretation of Hypothesis Output

$h_\theta(x) =$ estimated probability that $y = 1$ on input x
$h_\theta(x) = P(y = 1|x; \theta) =$ probability that $y = 1$, given x, parameterized by θ

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

6.2 Decision Boundary

$$\begin{aligned}
\text{Predict: } y = 1 && \text{if } h_\theta(x) \geq 0.5 \\
&& (\theta^T x \geq 0) \\
\text{Predict: } y = 0 && \text{if } h_\theta(x) < 0.5 \\
&& (\theta^T x < 0) \\
\theta^T x = 0 && \text{is the } \textit{decision boundary.}
\end{aligned}$$

Non-linear Decision Boundaries

Use same technique as polynomial regression for features.

6.3 Cost Function

$$\begin{aligned}
\text{Training Set : } &\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ./dots, (x^{(m)}, y^{(m)})\} \\
x = &\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1} \\
x_0 = &\ 1 \\
y \in &\ \{0, 1\} \\
h_\theta(x) = &\ \frac{1}{1 + e^{-\theta^T x}}
\end{aligned}$$

How to choose parameter θ?

Linear Regression:

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

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

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

Logistic Regression:

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

$$\text{Cost}(h_\theta(x), y) = 0 \text{ if } h_\theta(x) = y$$

$$\text{Cost}(h_\theta(x), y) \to \text{inf} \;\text{ if } y = 0 \text{ and } h_\theta(x) \to 1$$

$$\text{Cost}(h_\theta(x), y) \to \text{inf} \;\text{ if } y = 1 \text{ and } h_\theta(x) \to 0$$

Note: $y = 0$ or 1 always.

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

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

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

To fit parameters θ:

$$\underset{\theta}{\text{minimize}}\, J(\theta)$$

To make a prediction given a new x:

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

Gradient Descent:

Simultaneously update all θ_j

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

Plug in the derivative

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

Don't forget feature scaling!

6.4 Advanced Optimization:

Something better than gradient descent:

1. Conjugate Gradient
2. BFGS
3. L-BFGS

Advantages:

1. No need to manually pick α
2. Often faster than gradient descent

Disadvantages:

1. More complex

Use libraries! Beware of bad implementations!

How to use: We first need to provide a function that evaluates the following two functions for a given input value of θ.

1. $J(\theta)$
2. $\frac{\partial}{\partial \theta_j} J(\theta)$

```
% We can write a single function that can return both of these:
function [jVal, gradient] = costFunction(theta)
        jVal = [...code to compute J(theta)...];
        gradient = [...code to compute derivative of J(theta)...];
end
```

Then we can use octave's *fminunc()* optimization algorithm along with the *optimset()* function that creates an object containing the options we want to send to *fminunc()*.

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

6.5 Multi-Class Classification

6.5.1 One vs All

Build a separate binary classifier $h_\theta^{(i)}(x)$ for each class against all other classes.

$$h\theta^{(i)} = P(y = i|x;\theta)\ \forall i$$

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

Check Lecture6.pdf for more details.

Chapter 7

Regularization

7.1 Problem of Overfitting

1. Underfitting (High Bias)
2. Right Fit
3. Overfitting (High Variance)

Overfitting: If we have too many features, the learned hypothesis may fit the training set very well $\left(J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h\theta(x^{(i)}) - y^{(i)})^2 \approx 0\right)$, but fail to generalize to new examples (predict prices on new examples).

7.2 Addressing Overfitting

Options:

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

7.3 Cost Function

Say our overfitting hypothesis is $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$. Suppose, we penalize and make θ_3, θ_4 very small

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

Regularization

Small values for parameters $\theta_0, \theta_1, \ldots, \theta_n$.

- *Simpler* hypothesis
- Less prone to overfitting

Which parameters to penalize?

- Features: $x_1, x_2, \ldots, x_{100}$
- Parameters: $\theta_0, \theta_1, \theta_2, \ldots, \theta_{100}$

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

Note: The convention, we don't regularize θ_0 but it doesn't make very much difference.

λ here is *regularization parameter*.

What if λ is set too high (say 10^{10})? Underfitting!

7.4 Regularized Linear Regression

Updated $J(\theta)$:

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

$$\underset{\theta}{\text{minimize}}\, J(\theta)$$

Gradient Descent:

Repeat{

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

$$\theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left((h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right] \quad \forall j \in \{1, 2, \ldots, n\}$$

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

}

Normal Equation:

$$X = \begin{bmatrix} \left(x^{(1)}\right)^T \\ \left(x^{(2)}\right)^T \\ \vdots \\ \left(x^{(m)}\right)^T \end{bmatrix} \qquad \in \mathbb{R}^{m \times (n+1)}$$

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \qquad \in \mathbb{R}^{m}$$

$$\theta = \left(X^T X + \lambda \begin{bmatrix} 0 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \left(\in \mathbb{R}^{(n+1) \times (n+1)} \right) \right)^{-1} - X^T y \quad \in \mathbb{R}^{n+1}$$

Non-Invertibility

Suppose $m \leq n$,

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

If $\lambda > 0$,

$$\theta = \left(X^T X + \lambda \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix} \right)^{-1} - X^T y$$

Not a problem!

7.5 Regularized Logistic Regression

Cost Function:

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

Gradient Descent:

$$\begin{aligned}
&\text{Repeat}\{ \\
&\qquad \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\
&\qquad \theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left((h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right] \quad \forall j \in \{1, 2, \ldots, n\} \\
&\quad \equiv \theta_j := \theta_j (1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \\
&\qquad \}
\end{aligned}$$

Advanced Optimization:

```
function [jVal, gradient] = costFunction(theta)
        jVal = [...code to compute J(theta)...];
        gradient = [...code to compute derivative of J(theta)...];
```

Check Lecture7.pdf for more details.

Chapter 8

Neural Networks

8.1 Non-Linear Hypothesis

If #features is high, hypothesis could have extremely high #terms. Hence the need for non-linear hypothesis.

8.2 Neural Networks

- *Origins:* Algorithms that try to mimic brain.
- Widely used in 80s and early 90s; diminished in late 90s.
- Resurgance: State-of-the-art technique for many applications.

8.3 Model Representation

Neuron Model: Logistic Unit

1. input layer
2. hidden layer / computation layer
3. output layer
4. activation function ($g()$)

parameters $\equiv$ weights

8.4 Notations

$$\begin{aligned}
a_i^{(j)} &= \textit{activation} \text{ of unit } i \text{ in layer } j \\
\Theta^{(j)} &= \text{matrix of weights controlling} \\
&\quad\ \text{function mapping from layer } j \text{ to} \\
&\quad\ \text{layer } j+1
\end{aligned}$$

Example:

$$\begin{aligned}
&\text{layer 1} : \{x_1, x_2, x_3\} \\
&\text{layer 2} : \{a_1^{(2)}, a_2^{(2)}, a_3^{(2)}\} \\
&\text{layer 3} : \{a_1^{(3)}\}
\end{aligned}$$

$$\begin{aligned}
a_1^{(2)} &= g\left(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3\right) \\
a_2^{(2)} &= g\left(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3\right) \\
a_3^{(2)} &= g\left(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3\right) \\
h_\Theta(x) = a_1^{(3)} &= g\left(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(1)} a_3^{(2)}\right)
\end{aligned}$$

Note: If network has s_j units in layer j, and s_{j++1} units in layer $j+1$, then

$$\Theta^{(j)} \in \mathbb{R}^{s_{j+1} \times (s_j+1)}$$

8.5 Forward Propagation

$$
\begin{aligned}
z_1^{(2)} &= \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3 \\
z_2^{(2)} &= \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3 \\
z_3^{(2)} &= \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3 \\
a_1^{(2)} &= g(z_1^{(2)}) \\
a_2^{(2)} &= g(z_2^{(2)}) \\
a_3^{(2)} &= g(z_3^{(2)})
\end{aligned}
$$

Vectorized Implementation

$$
x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 3 \end{bmatrix},\ x_0 = 1
$$

$$
z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}
$$

$$
\begin{aligned}
a^{(1)} &= x && \in \mathbb{R}^4 \\
z^{(2)} &= \Theta^{(1)} a^{(1)} && \in \mathbb{R}^3 \\
a^{(2)} &= g(z^{(2)}) && \in \mathbb{R}^3 \\
\textbf{Add } a_0^{(2)} &= 1 && \to a^{(2)} \in \mathbb{R}^4 \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
h_\Theta(x) = a^{(3)} &= g(z^{(3)})
\end{aligned}
$$

Other network architechtures are possible!

Non-linear classification example: XOR/XNOR
x_1, x_2 are binary (0 or 1)

$$\begin{aligned} y &= x_1 \text{ XOR } x_2 & \textit{or} \\ &= x_1 \text{ XNOR } x_2 & \equiv \text{NOT}(x_1 \text{ XOR } x_2) \end{aligned}$$

[1] Simple example: AND

$$\begin{aligned} x_0 &= 0 \\ x_1, x_2 &\in \{0,1\} \\ y &= x_1 \,\text{AND}\, x_2 \\ \Theta^{(1)}_{10} &= -30 \\ \Theta^{(1)}_{11} &= 20 \\ \Theta^{(1)}_{12} &= 20 \\ h_\Theta(x) &= g(-30 + 20x_1 + 20x_2) \end{aligned}$$

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

We can combine AND, OR, NOT, (NOT x_1) AND (NOT x_2) to get XNOR, XOR, etc.

8.6 Multi-Class Classification

Extension of One-vs-All Method!

Example: Say 4 classes, so 4 output units. So, $y \in \mathbb{R}^4$ with 1 in corresponding class and 0 elsewhere.

[1] Work out more examples (OR, NOT, (NOT x_1) AND (NOT x_2)!
Check Lecture8.pdf for more details.

Chapter 9

Back Propagation

9.1 Notations

$$
\begin{aligned}
L &= \text{total no. of layers in the network} \\
s_l &= \text{no. of units (not counting bias unit) in layer } l
\end{aligned}
$$

Binary Classification

$$y = 0 \text{ or } 1$$

1 output unit

$$
\begin{aligned}
h_\Theta(x) &\in \mathbb{R} \\
s_L &= 1 \\
K &= 1 && \text{(for simplification)}
\end{aligned}
$$

Multi-class Classification (K classes)

$$y \in \mathbb{R}^K$$

Example: $K = 4$

$$y \in \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right\}$$

K output units

$$h_\Theta(x) \in \mathbb{R}^K$$
$$s_L = K$$
$$K \geq 3$$

9.2 Cost Function

Logistic Regression

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

Neural Network

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

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

9.3 Backpropagation Algorithm

9.3.1 Gradient Computation

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

$\min_\Theta J(\Theta)$

Need code to compute:

- $J(\Theta)$
- $\dfrac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

Given one training example (x, y):

Forward Propagation:

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

Gradient Computation: Back Propagation

Intuition: $\delta_j^{(l)} =$ "error" of node j in layer l

For each output unit $(L = 4)$

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

Vectorize:

$$
\begin{aligned}
\delta^{(4)} &= a^{(4)} - y \\
\delta^{(3)} &= (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)}) \\
g'(z^{(3)}) &= a^{(3)} .* (1 - a^{(3)}) \\
\delta^{(2)} &= (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)}) \\
g'(z^{(2)}) &= a^{(2)} .* (1 - a^{(2)})
\end{aligned}
$$

No $\delta^{(1)}$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)} \qquad \text{(ignoring } \lambda\text{; if } \lambda = 0\text{)}$$

Backpropagation Algorithm

Training set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j) $\rightarrow$ accumulators to compute $\dfrac{\partial}{\partial\Theta_{ij}^{(l)} J(\Theta)}$

For $i = 1$ to m

Set $a^{(1)} = x^{(1)}$

Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)}\delta_i^{(l+1)}$

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

$$D_{ij}^{(l)} := \begin{cases} \dfrac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \\ \dfrac{1}{m}\Delta_{ij}^{(l)} & \text{if } j = 0 \end{cases}$$

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

9.4 Implementation Note

9.4.1 Unrolling Parameters

Advanced optimization functions like ‘fminunc’ assume that inputs ‘theta’, ‘gradient’, etc. are vectors. These are matrices in Neural Networks, so we *unroll* them into vectors.

Example: Neural Network with 3 layers (1 input, 1 hidden, 1 output) and $s_1 = 10$, $s_2 = 10$, $s_3 = 1$.

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

To unroll:

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

To go back to matrices:

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

9.4.2 Learning Algorithm

Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$

Unroll to get `initialTheta` to pass to

`fminunc(@costFunction, initialTheta, options)`

`function [jVal, gradientVec] = costFunction(thetaVec)`

From `thetaVec`, get $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$ using `reshape`

Use forward/back propagation to compute $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ and $J(\Theta)$

Unroll $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ to get `gradientVec`

9.5 Gradient Checking

An unfortunate problem is that it might seem like it's working even when there's a bug in the backpropagation algorithm (implementation). Subtle bugs can hence cause higher level of error and perform worse than a bug-free implementation. Here, we try to make sure that our implementation is 100% correct.

9.5.1 Numerical Estimation of gradients

To estimate derivative of $J(\theta)$, we calculate $J(\theta+\epsilon)$ and $J(\theta-\epsilon)$ and take the slope of the line connecting these two points.

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

Use pretty small ϵ ($\approx 10^{-4}$).

The above formula is called the two sided difference. Another formula called the one sided difference is

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

But the two sided difference usually gives a slightly more accurate approximation.

Implement:

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

Parameter *vector* θ

$$\theta \in \mathbb{R}^n$$

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

$$\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1 + \epsilon, \theta_2, \theta_3, \dots, \theta_n) - J(\theta_1 - \epsilon, \theta_2, \theta_3, \dots, \theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2 + \epsilon, \theta_3, \dots, \theta_n) - J(\theta_1, \theta_2 - \epsilon, \theta_3, \dots, \theta_n)}{2\epsilon}$$

$$\vdots$$

$$\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \dots, \theta_n + \epsilon) - J(\theta_1, \theta_2, \theta_3, \dots, \theta_n - \epsilon)}{2\epsilon}$$

Implement:

```
for i = 1:n
        thetaPlus = theta;
        thetaPlus(i) = thetaPlus(i) + EPSILON;
        thetaMinus = theta;
        thetaMinus(i) = thetaMinus(i) - EPSILON;
        gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*EPSILON);
end;
```

Check that `gradApprox` $\approx$ `DVec` (from backpropagation).

9.5.2 Implementation Note:

- Implement backpropagation to compute `Dvec` (unrolled $D^{(1)}$, $D^{(2)}$, $\dots$)
- Implement numerical gradient check to compute `gradApprox`
- Make sure they give similar values
- Turn off gradient checking (much slower) and use backpropagation for learning (much faster)

Important:

- Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computaion on every iteration of gradient descent (or in the inner loop of `costFunction(...)`) your code will be very slow.

9.6 Random Initialization

Initial value of Θ

For gradient descent and advanced optimization methods, we need initial value of Θ.

Consider gradient descent:
Set `initialTheta = zeroes(n, 1)` ?

Works fine for linear/logistic regression, but not when we're training a neural network. Why?

$$\begin{aligned} \Theta_{ij}^{(l)} &= 0 \text{ for all } i, j, l \\ a_1^{(2)} &= a_2^{(2)} \\ \delta_1^{(2)} &= \delta_2^{(2)} \\ \frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) &= \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta) \end{aligned}$$

Meaning, after each gradient descent update, parameters corresponding to inputs going into each unit of the next layer are identical. Thus, all the units of the next layer are still computing the same function of the input. The values may change, but will always be the same, preventing the neural network from learning something interesting since it is extremely redundant.

Random Initialization: Symmetry Breaking

To get around this (*symmetric weights*), we do random initialization. Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$ (i.e. $-epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$).

Example:

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

A good choice of ϵ is:

$$\epsilon = \frac{\sqrt{6}}{\sqrt{L_{in} + L_{out}}}$$

Where L_{in} and L_{out} are number of units in the layers adjacent to $\Theta^{(l)}$.

9.7 Putting it all together

Pick a network architecture (connectivity pattern between neurons), like number of layers, number of units in each layer, etc. For example, $[3 \to 5 \to 4]$ vs $[3 \to 5 \to 5 \to 4]$ vs $[3 \to 5 \to 5 \to 5 \to 4]$.

$$\begin{aligned} \text{No. of input units} &: \text{Dimension of features } x^{(i)} \\ \text{No. of output units} &: \text{Number of classes} \end{aligned}$$

Remember:

$$y \in \{1, 2, \ldots, 10\} \text{ becomes}$$
$$y \in \left\{ \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \right\}$$

For Hidden Layers?

Reasonable Default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden units in every layer (usually, the more the better, just that it may get computationally expensive). Usually, comparable to the number of features.

Training a neural network:

1. Randomly initialize weights
2. Implement forward propagation to get $h_\Theta(x)$ for any x
3. Implement code to compute cost function $J(\Theta)$
4. Implement backpropagation to compute partial derivatives $\dfrac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

```
for i = 1:m
        Perform forward propagation ...
                and backpropagation ...
                using example (x^(i), y^(i))

        Get activations a^(l) ...
                and delta terms δ^(l) ...
                for l = 2,...,L

        Δ^(l) := Δ^(l) + δ^(l+1) (a^(l))^T
        ⋮
end
⋮
compute ∂/∂Θ_jk^(l) J(Θ)
```

5. (a) Use gradient checking to compare $J'(\Theta)$ computed using backpropagation vs using numerical estimate.
   (b) Then disable gradient checking code.
6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$

$J(\Theta)$ in general for a neural network is non-convex, and hence susceptible to local optima. However, it turns out, in practice, this is not usually a huge problem.

Check Lecture9.pdf for more details.