

# ML - Machine Learning

## Misc informations

- There are two classes on ML problems: Linear Regression and Classification
- Deep NN: a NN with more than one hidden layer
- AI requires big data
- Data sets: It is important that the training data is split from the test data. It is essential to make sure that what we have learned actually generalises
- Example of ML: Web page ranking, Anti spam algorithms, Face recognition if fb photo, Autonomous vehicles, Handwritten recognition, Speech recognition, Computer Vision, Brain modelling, customise Netflix product recommendations
- Example of database mining: medical records, computational biology, engineering
- "ML is the field of study that gives computers the ability to learn without being explicitly programmed" (Arthur Samuel, 1959)
- "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E" (Tom Mitchell, 1998)
- Supervised Learning: a training set of data with exact labels is given
- Unsupervised Learnings: find clusters or hidden structure from unlabeled data. Example: market segmentation
- SVD (Singular Value Decomposition): factorisation of a real or complex matrix in the form  $USV^*$ , where U and V are unitary and S is diagonal with real non-negative numbers on the diagonal
- For ML prototyping engineers use one of several languages, Octave, Matlab, R, Python, NumPy. Octave is open source and very easy. Matlab is expensive. Python and NumPy syntax is more complex
- Vectorization: it is the process of converting loops of scalar operations into vector operations leveraging specific libraries, or frameworks. Vectorization improves greatly computation efficiency and code readability. For example, instead of performing this

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

or, in C++, this

```
double prediction = 0.0;
for (int j = 0; j <= n; j++)
    prediction += theta[j] * x[j];
```

perform this

$$\theta^T x$$

or this

```
double prediction
    = theta.transpose() * x;
```

## Machine Learning courses and books

- Intro to Machine Learning by A. Ng (<http://bit.ly/1IXp8Lg>)
- Exercises: <http://cs229.stanford.edu/materials.html>
- Machine Learning by J. W. Paisley, video course (available at edX <http://bit.ly/2s3pjmG>)
- Machine Learning A-Z™ by K. Eremenko, video course on Udemy (<http://bit.ly/2s33KTx>)
- Machine Learning for Hackers by D. Conway and J. M. White, book R language (<http://amzn.to/2t2Plv2>)
- Machine Learning from for the Web by A. Isoni, book Python language (<http://amzn.to/2sLsgKw>)
- Machine Learning with Python by S. Raschka, book Python language (<http://amzn.to/2rJTSi3>)
- Foundations of Deep Learning by Hugo Larochelle, Twitter ([https://youtu.be/zij\\_FTbJHsk](https://youtu.be/zij_FTbJHsk))
- Deep Learning for Computer Vision by Andrej Karpathy, OpenAI (<https://youtu.be/u6aEYuemt0M>)
- Deep Learning for Natural Language Processing by Richard Socher, Salesforce (<https://youtu.be/oGk1v1jQITw>)
- TensorFlow Tutorial by Sherry Moore, Google Brain ([https://youtu.be/Ejec3ID\\_h0w](https://youtu.be/Ejec3ID_h0w))
- Foundations of Unsupervised Deep Learning by Ruslan Salakhutdinov, CMU (<https://youtu.be/rK6bchqeaN8>)
- Nuts and Bolts of Applying Deep Learning by Andrew Ng (<https://youtu.be/F1ka6a13S9I>)
- Deep Reinforcement Learning by John Schulman, OpenAI (<https://youtu.be/PtAlh9KSnjo>)
- Theano Tutorial by Pascal Lamblin, MILA (<https://youtu.be/OU8I1oJ9HhI>)
- Deep Learning for Speech Recognition by Adam Coates, Baidu (<https://>

[youtu.be/g-sndkf7mCs](https://youtu.be/g-sndkf7mCs))

- Torch Tutorial by Alex Wiltschko, Twitter (<https://youtu.be/L1sHcj3qDNc>)
- Sequence to Sequence Deep Learning by Quoc Le, Google ([https://youtu.be/G5RY\\_SUJih4](https://youtu.be/G5RY_SUJih4))
- Foundations and Challenges of Deep Learning by Yoshua Bengio ([https://youtu.be/11rsu\\_WwZTc](https://youtu.be/11rsu_WwZTc))

Full Day Live Streams:

Day 1: <https://youtu.be/eyovmAtouX0>

Day 2: <https://youtu.be/9dXiAecyJrY>

## Supervised Models

Estimating the relationships among one or more input features and the output variable

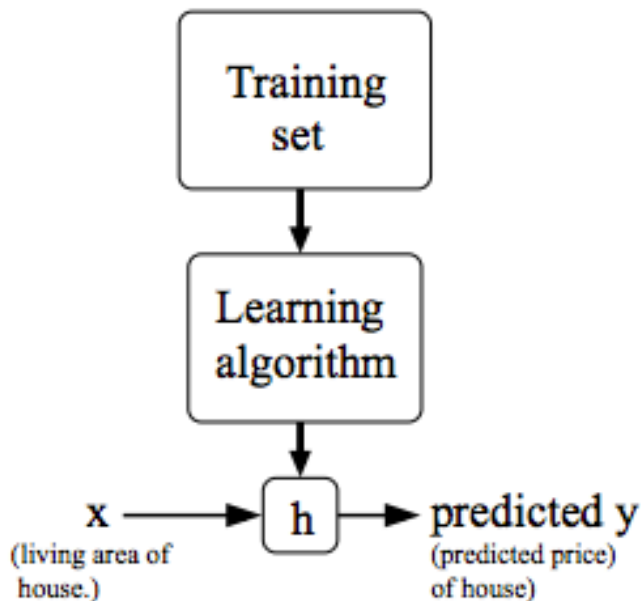
$x^{(i)}$  in  $X$ , the input features

$y^{(i)}$  in  $Y$ , the target variable that we are trying to predict

$(x^{(i)}, y^{(i)})$ , training example

$(x^{(i)}, y^{(i)})$ ;  $i=1, \dots, m$ , training dataset

$h: X \rightarrow Y$ , the hypothesis function



- Regression problem:  $Y$  is continuous. Example: estimate the age of a person on the basis of a given picture of that person
- Classification problem:  $Y$  is discrete. Example: predict whether the tumor is benign or malign on the basis of the tumor size and the patient's age
- Cost Function: measures the accuracy of the hypothesis

## Univariate Linear Regression

The model is univariate when the hypothesis depends on one feature  $x$  ( $n=1$ )

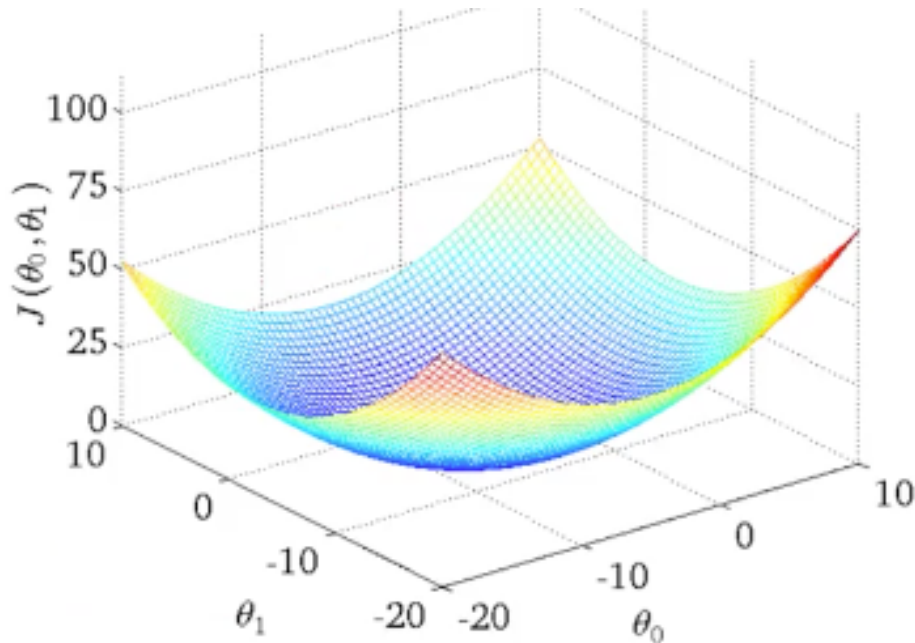
**Hypothesis:** 
$$h_{\theta}(x) = \theta_0 + \theta_1 x$$

**Parameters:**  $\theta_0, \theta_1$

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

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

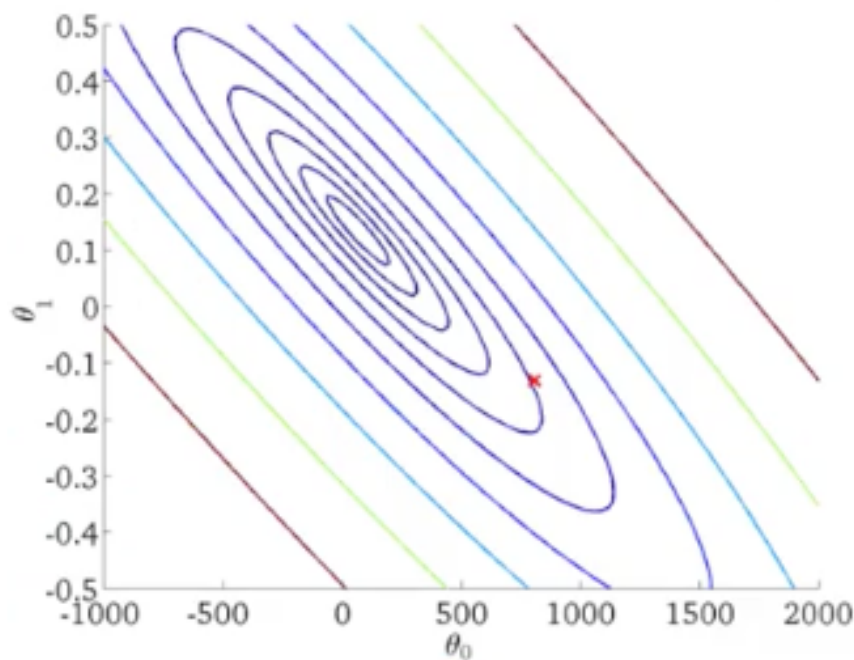
The Cost Function is a hyperboloid



Contour plot for  $J$

$$J(\theta_0, \theta_1)$$

(function of the parameters  $\theta_0, \theta_1$ )



- *Gradient Descent* algorithm: We start with some parameters value, then we keep changing them to reduce  $J$  until we hopefully end up at a minimum. Repeat the following iteration over all parameters until convergence

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

- Warning: you have to update all the parameters simultaneously! So you have to calculate first all the right-hand expressions, then make the assignments
- Alpha is the *Learning Rate*. If alpha is too small, the convergence takes too long. If alpha is too large, the process may not converge or even diverge
- For the Linear Regression the Gradient Descent is

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

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

- *Batch* Gradient Descent: because each step of the algorithm uses all the training examples
- For the Linear Regression  $J$  is a convex quadratic function, thus there is only one global minimum and the algorithm always converges, assuming the learning rate is not too large

## Linear Algebra

- 1-indexed vs 0-indexed vectors

$$y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \quad y = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

In a linear algebra course you might have 1-indexed notation

In a ML low level implementation you might have the 0-indexed notation

## Multivariate Linear Regression

The model is multivariate when the hypothesis depends on more than one feature ( $n > 1$ )

- By adding a dummy feature always equal to one

$$x_0^{(i)} = 1 \text{ for } (i \in 1, \dots, m)$$

we can express the hypothesis as a vector multiplication

$$\text{Hypothesis: } h_{\theta}(x) = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

$$\text{Parameters: } \theta_0, \theta_1, \dots, \theta_n$$

Cost function:

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

- Now the *Gradient Descent* algorithm is to repeat the following iterations over all parameters (simultaneously) until convergence

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

- For the Linear Regression the Gradient Descent is

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

Then this is the general case

$$\begin{aligned}
 &\text{repeat until convergence: } \{ \\
 &\quad \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)} \\
 &\quad \theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)} \\
 &\quad \theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)} \\
 &\quad \dots \\
 &\quad \}
 \end{aligned}$$

- Feature Scaling: we can speed up gradient descent by having each of our input values in roughly the same range. This is because  $\theta$  will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven. The way to prevent this is to modify the ranges of our input variables so that they are all roughly the same. Ideally  $-1 \leq x(i) \leq 1$ . This is not an exact requirements; we are only trying to speed things up. The goal is to get all input variables into roughly one of these ranges, give or take a few.

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

where  $\mu$  is the mean and  $s$  is the max-min or standard deviation of  $x$ .

- How to know that the Gradient Descent is working ? Just plot the cost function vs number of iterations. It should decrease after every iteration
- When to stop ? We can declare convergence if  $J$  decreases by less than 0.001 in one iteration
- How to choose the correct alpha ? choose the maximum alpha such that  $J$  decreases on every iteration. Start with 0.001, then 0.01, 0.1, ..

## Polynomial Regression

The model need not to be linear if it does not fit the data well. You can build up new features which are nonlinear function of the original feature. If you do this, then the feature scaling becomes very important.

## Normal Equation

Gradient descent gives one way of minimizing  $J$ . Normal Equation is a method to solve it analytically.

If  $J$  is a quadratic function, then

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

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

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

There is no need to do feature scaling with the normal equation

Examples:  $m = 4$ .

	Size (feet <sup>2</sup> )	Number of bedrooms	Number of floors	Age of home (years)	Price (\$1000)
$x_0$	$x_1$	$x_2$	$x_3$	$x_4$	$y$
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1534	3	2	30	315
1	852	2	1	36	178

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

The exact solution is given by

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

In Octave the instruction to do this is

**`pinv(X' * X) * X' * y`**

This method is advantageous when  $n$  is not very large, because  $n \times n$  matrix inversion cost is  $O(n^3)$ , while for the Gradient Descent the cost is  $O(kn^2)$ . So if  $n = 10^4$  it is better to use the Gradient Descent. The Normal Equation method does not work for different ML algorithms

- What if  $X^T X$  is non-invertible? (singular/degenerate). This case can happen if two features are linearly dependent, or when you have too many features ( $m \leq n$ ). In this case you can delete some features, or use regularisation.
- In Octave we have `pinv()` and `inv()`. The first one gives you a value of  $\theta$  even if  $X^T X$  is not invertible.

## Classification problems



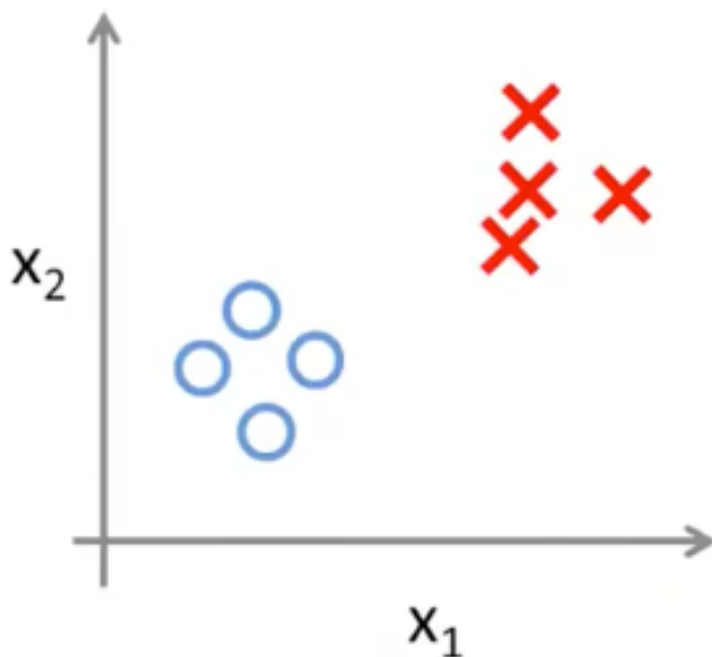
- Examples: decide whether
  - a received email is Spam or Not Spam
  - an online transaction is fraudulent or not
  - a tumor is malignant or benign
- The output variable is a discrete variable, in the simplest case 0 or 1 (Binary Classification Problems)
- In all of these problems the variable that we're trying to predict is a variable  $y$  that we can think of as taking on two values either 0 or 1 (also known as the negative and the positive classes)
- The linear regression method doesn't work with a classification task because classification is not actually a linear function

## Logistic Regression models

### Binary classification

- In binary classification we have

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



- We'll develop an algorithm called logistic regression, which is one of the most popular and most widely used learning algorithms today, for which

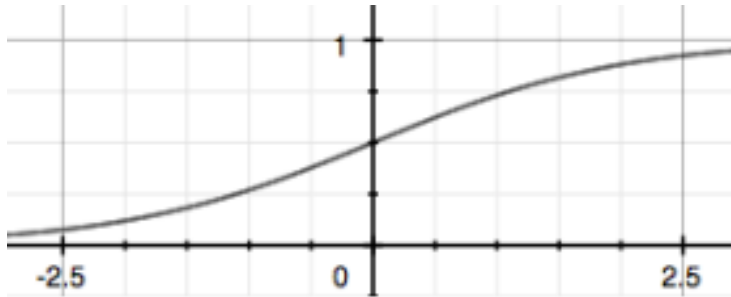
$$0 \leq h_{\theta}(x) \leq 1$$

- The classification estimator takes the form of the Sigmoid (or Logistic) Function

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

$$z = \theta^T x$$

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



- $h(x)$  is interpreted as the estimated probability that  $y=1$  on input  $x$ , or  $h(x, \theta) = P(y=1|x;\theta)$
- Let's set the decision threshold to 0.5

$$h_{\theta}(x) \geq 0.5 \rightarrow y = 1$$

$$h_{\theta}(x) < 0.5 \rightarrow y = 0$$

- Decision boundary is the hyperplane in the input space  $x$  which divides the region  $y=0$  by the region where  $y=1$

$$\theta^T x \geq 0 \Rightarrow y = 1$$

$$\theta^T x < 0 \Rightarrow y = 0$$

- Again, the input to the sigmoid function  $g(z)$  doesn't need to be linear, and could be a function that describes a circle or any shape to fit our data
- Now, the problem is to find an automatic method to find the  $\theta$  parameters fitting the training set
- We need a convex Cost Function (in order to make the gradient descent convergent to the global minimum). Then we define the convex Logistic Cost Function as follows

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

where

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

$\text{Cost}(h_{\theta}(x), y) = 0$  if  $h_{\theta}(x) = y$   
 $\text{Cost}(h_{\theta}(x), y) \rightarrow \infty$  if  $y = 0$  and  $h_{\theta}(x) \rightarrow 1$   
 $\text{Cost}(h_{\theta}(x), y) \rightarrow \infty$  if  $y = 1$  and  $h_{\theta}(x) \rightarrow 0$

The cost function above can be written

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

which brings to the following expression for the total cost

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

A vectorised implementation is

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

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

In Octave

```
function [J, grad] = costFunction(theta, X, y)
m = length(y); % number of training examples
J = y'*log(sigmoid(X*theta)) + (1 - y)'*log(1 - sigmoid(X*theta));
J = -J/m;
```

The Gradient Descent algorithms looks like

$$\text{Repeat } \{$$

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

$$\}$$

Or, more explicitly,

$$\text{Repeat } \{$$

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

$$\}$$

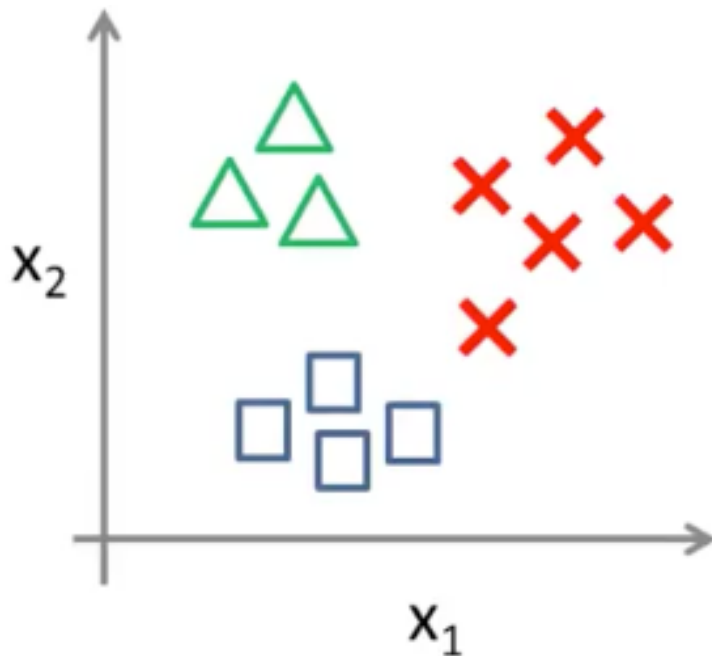
A vectorized version is

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

## Multi-class classification

- In Multi-class classification we have K output units

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



- One-vs-all classification algorithm

Example: an algorithm that puts an email in one of several folders, like Work, Friends, Family, Hobby

- The idea is to train a Logistic regression classifier for each class  $i$  to predict the probability that  $y = i$ .

Then, to make a prediction on a new input  $x$ , get the class  $i$  that maximises

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

$$y \in \{0, 1, \dots, n\}$$

$$h_{\theta}^{(0)}(x) = P(y = 0|x; \theta)$$

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

...

$$h_{\theta}^{(n)}(x) = P(y = n|x; \theta)$$

$$\text{prediction} = \max_i (h_{\theta}^{(i)}(x))$$

- We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic

regression to each case, and then use the hypothesis that returned the highest value as our prediction.

## Advanced Optimisation functions in Octave

- Gradient Descent is not the only algorithm to optimise the cost function. There are more complex, and faster, ways
  - Conjugate Gradient
  - BFGS
  - L-BFGS

The aforementioned algorithms don't need to manually set alpha: at every iteration an optimal alpha is selected and used.

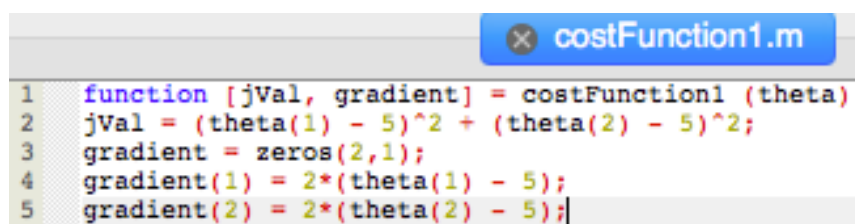
- Octave provides a number of complex library functions to deal with optimisation problems (like Linear and Logistic regressions), i.e. find a local minimum of some functional. They do the job much more efficiently than the Gradient Descent.
- Consider the optimisation of the following cost function

$$J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$$

We know the solution is  $\theta = (5, 5)$ , but our purpose is to find it using `optimset` and `fminunc`.

`fminunc` solves an unconstrained optimization problem defined by a given function `J`. This function should accept a parameters vector `theta` (the unknown variables) and return `J` and the gradient of `J` evaluated in `theta`.

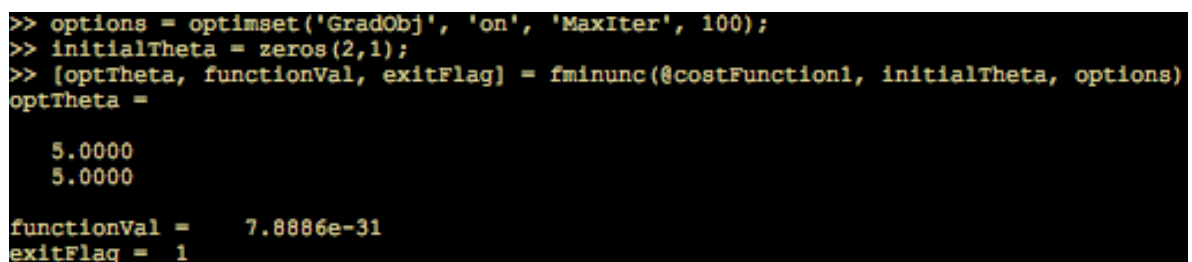
In our example



```
1 function [jVal, gradient] = costFunction1(theta)
2 jVal = (theta(1) - 5)^2 + (theta(2) - 5)^2;
3 gradient = zeros(2,1);
4 gradient(1) = 2*(theta(1) - 5);
5 gradient(2) = 2*(theta(2) - 5);
```

From the console we solve very easily

where `optTheta` is the solution of the problem, `functionVal` is the reached local minimum, `exitFlag` tells whether the process converged or not



```
>> options = optimset('GradObj', 'on', 'MaxIter', 100);
>> initialTheta = zeros(2,1);
>> [optTheta, functionVal, exitFlag] = fminunc(@costFunction1, initialTheta, options)
optTheta =

    5.0000
    5.0000

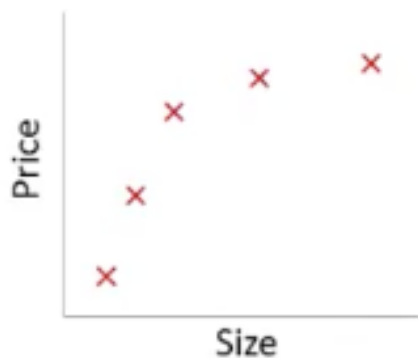
functionVal =    7.8886e-31
exitFlag =    1
```

- `fminunc` means "unconstrained minimum of function". Constraints in optimization often refer to constraints on the parameters, for example,

constraints that bound the possible values  $\theta$  can take (e.g.,  $\theta \leq 1$ ). Logistic regression does not have such constraints since  $\theta$  is allowed to take any real value.

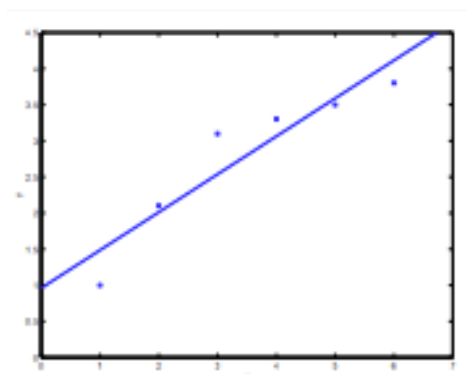
## Underfitting and Overfitting

- *Underfitting*: If the number of nonlinear features is insufficient the model performs poorly. We say that the it has “high bias”. This term is kind of a historical or technical one, but the idea is that if I you insist to fit a nonlinear  $x \rightarrow y$  relationship with a straight line, then it's as if the algorithm has a very strong preconception, or a very strong bias that  $y$  is going to vary linearly with  $x$  despite the data to the contrary
- *Overfitting*: If the number of nonlinear features is too high, the trained model performs very well on the training points ( $J$  almost 0), but it generalizes poorly in between. We say that the mode has “high variance”.
- For example, suppose you want to predict the price of a house as a function of the linear size of the front view. This is a linear regression problem.



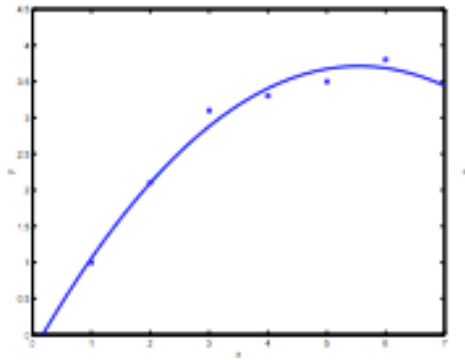
The chance is that the univariate linear regression underfits, especially for larger  $x$

$$y = \theta_0 + \theta_1 x$$



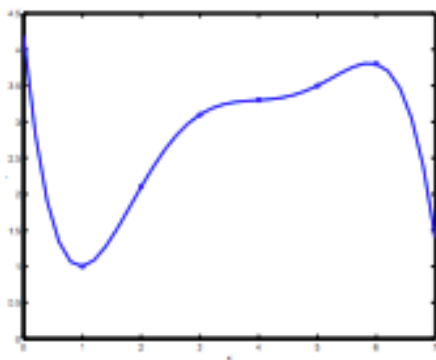
The quadratic model is the best

$$y = \theta_0 + \theta_1 x + \theta_2 x^2$$



The quartic model overfits

$$y = \sum_{j=0}^5 \theta_j x^j$$



- The overfitting is likely to occur also when we have a lot of features, but little training data
- To avoid overfitting can reduce the number of features, either manually or using a selection algorithm

## Regularization

- Another method is Regularization: keep all features, because all of them contribute more or less to predicting  $y$ , but smooth the output of our hypothesis function by reducing the magnitude of some (or all) model parameters. This can be achieved by adding suitable terms to the cost function
- Regularisation can apply to both linear regression and logistic regression.
- Example1. Lambda ( $>0$ ) is the *regularisation parameter*. If lambda is too large you fall into underfitting, because only  $\theta_0$  survives (h almost constant)

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

and, by separating the update function for  $\theta_0$ , the update function for  $j=1..n$  looks like this

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

- Example2: Suppose we want to penalise and make theta\_3,4 very small

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

- Example3: In the Normal Equation case, if we have m<n then X'X is non invertible, and may be non-invertible if m=n . The regularized solution takes the form

$$\theta = (X^T X + \lambda \cdot L)^{-1} X^T y$$

where L is the identity matrix (n+1)x(n+1), and X'X+lambda\*L is invertible

- Example4: For the Logistic Regression, the regularised cost function is

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

Note that the last term start from j=1, because we don't want regularise the first element theta\_0.

In a Octave implementation we achieve that with

```
2 m = length(y); % number of training examples
3 h = sigmoid(X*theta);
4 J = -(1/m)*[y'*log(h) + (1-y)'\log(1-h)];
5
6 temp = theta;
7 temp(1) = 0;
8 J = J + (lambda/(2*m))*temp'*temp;
```

## Neural Networks

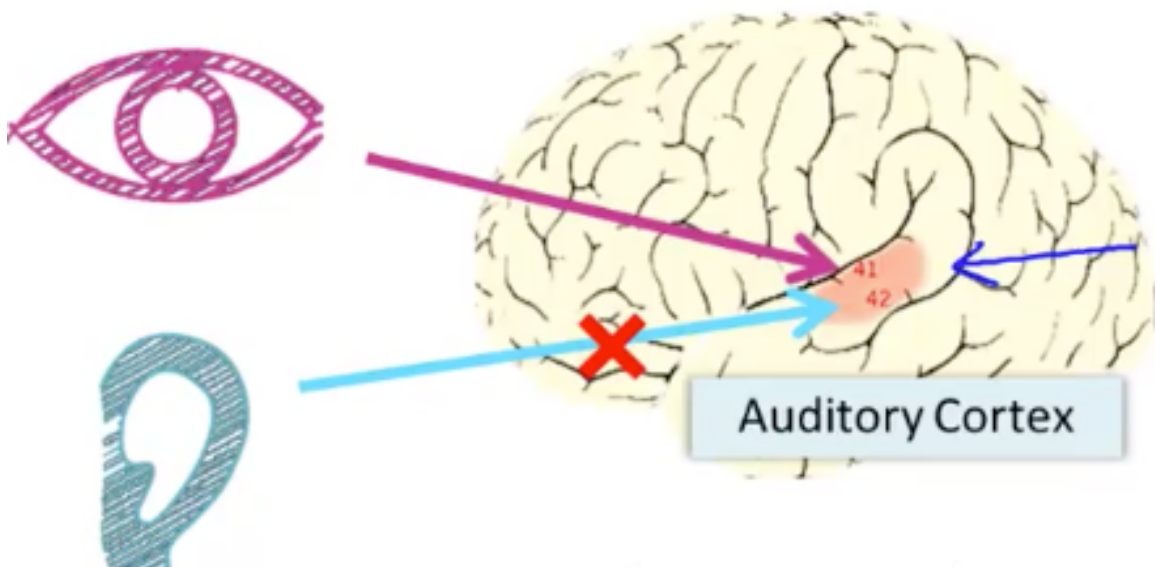
- Now we know how to train Linear Regression and Logistic Regression models, either with linear or nonlinear hypothesis, to provide optimal prediction. So why to introduce another paradigm ? The problem arises when the number of features n is high and you need higher order polynomial terms (N) to the hypotesis to fit complex datasets.



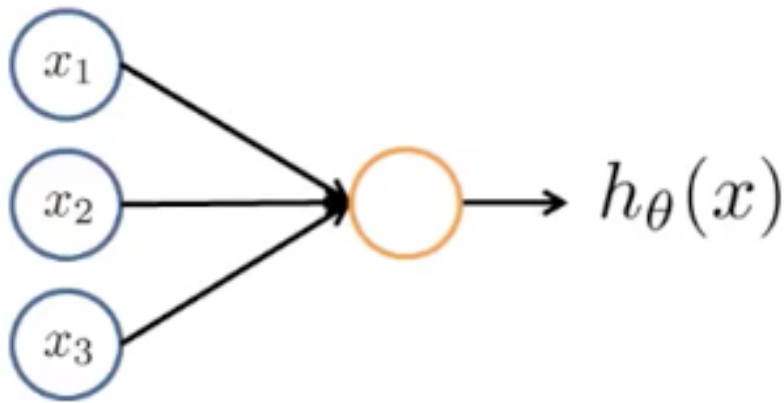
$$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^3 x_2 + \theta_6 x_1 x_2^2 + \dots)$$

Then the number of terms is  $O(n^N)$ , which becomes impractical in many cases.

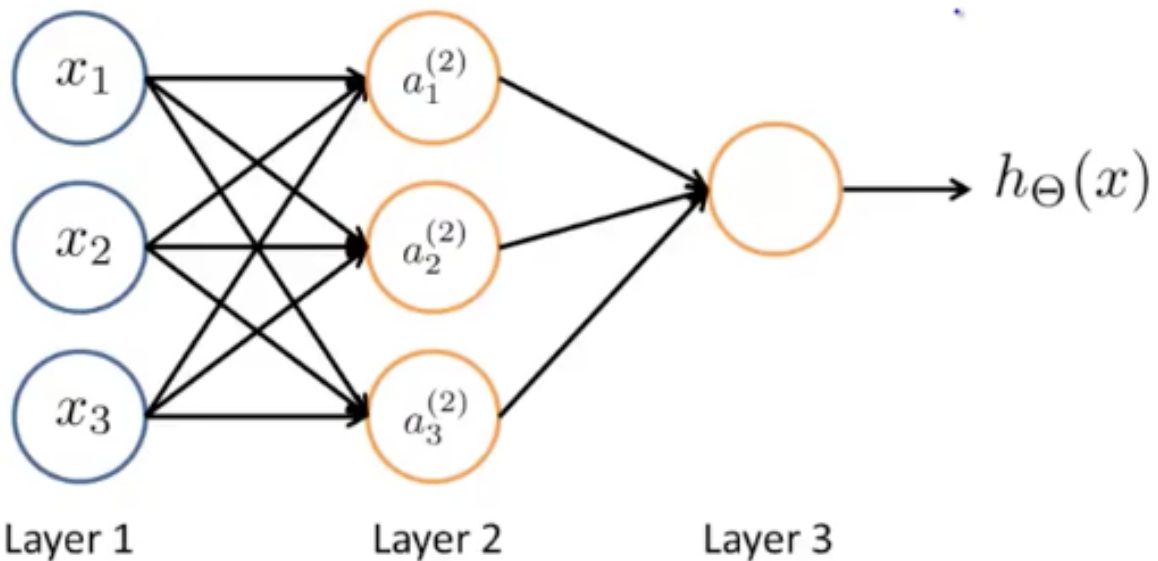
- The "one learning algorithm" hypothesis: Neuroscientists have replaced auditory with visual afferent signals to the auditory cortex of animals. Then the animal started to learn to see ! The same happens with the somatosensory cortex as well (Metin & Frost, 1989)



- Other studies on brain neuroplasticity confirm the non-specific nature of the primary sensory cortex (BrainPort; Welsh & Blasch, 1997; Nagel et al., 2005; Constantine-Paton & Law, 2009)
- Neuron model: this is a Logistic unit, where theta parameters are called "weights" and the sigmoid (logistic) function  $g$  is called "activation function". Sometimes an extra node  $x_0$  is added, which is known as the "bias unit", which is usually equal to 1.



- Neural Network: an example is



where

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

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

Also hidden layers may have an additional bias neuron. Then we have

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

- In supervised learning, only the input and the output layers are observables.
- Each layer gets its own matrix of weights and the dimensions of these matrices of weights is determined as follows:

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

- The +1 comes from the addition of the bias nodes
- The output nodes will not include the bias nodes while the inputs and the hidden layers will
- we can define vector  $x$  as  $a^{(1)}$
- We introduce  $z$  vectors to express the scalar products fed into the activation functions. For example, we compute the first hidden layer values in two steps

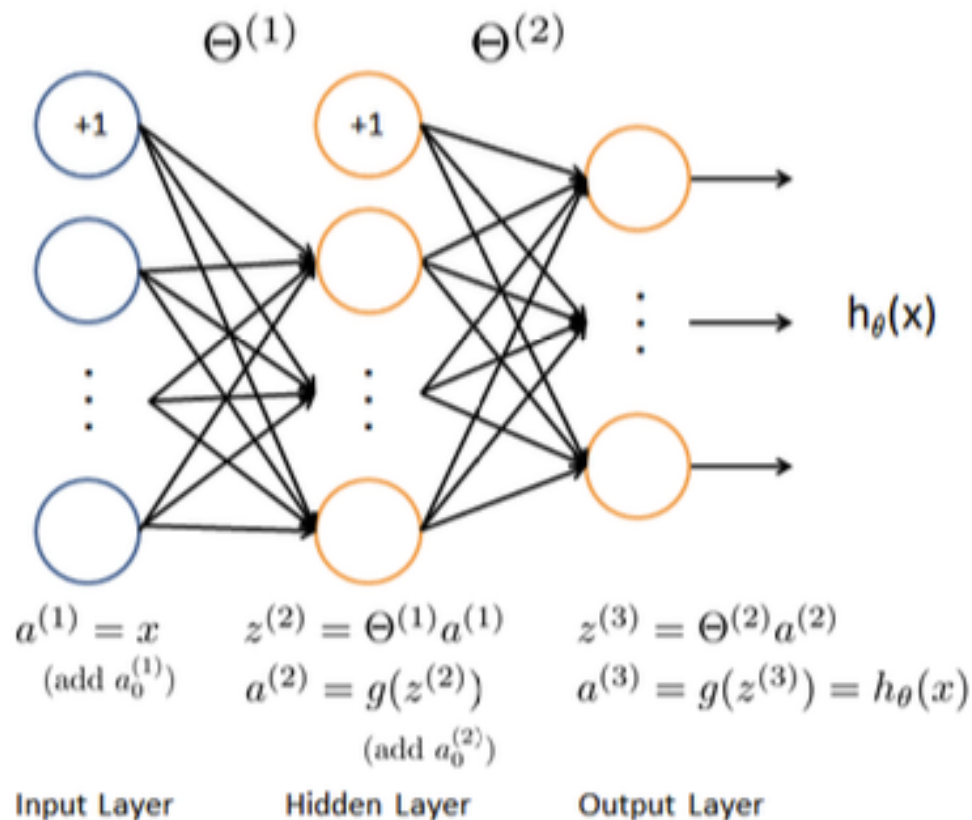
$$z^{(2)} = \Theta^{(1)} a^{(1)}; \quad a^{(2)} = g(z^{(2)})$$

In general we have  $L$  layers (including the input one), with  $s_j$  number of units (not counting the bias unit) in the  $j$ -th layer

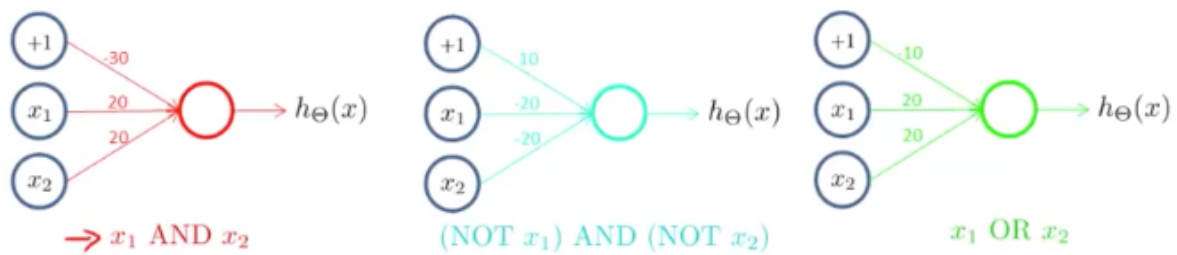
$$z_k^{(2)} = \Theta_{k,0}^{(1)} x_0 + \Theta_{k,1}^{(1)} x_1 + \dots + \Theta_{k,n}^{(1)} x_n$$

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

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



- Example: we can easily implement simple logic functions



It turns out that in order to implement either XOR or XNOR we have to add an hidden layer.

The general intuition is that each new layer we add to the network provides a more complex function than the previous layers. By adding many layers we can estimate arbitrary complex nonlinear hypothesis.

- Multiple class classification: One-vs-all



Pedestrian



Car



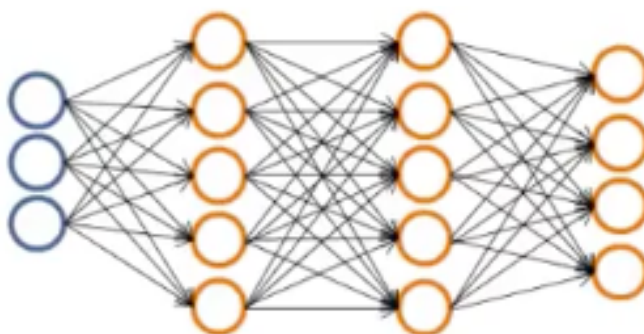
Motorcycle



Truck



Want  $h_{\Theta}(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ ,  $h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ ,  $h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ , etc.  
 when pedestrian      when car      when motorcycle



## NN Training

Suppose we have a  $m$  element training set.  
 We have two kind of NN Classification problems

### Binary classification (K=1)

- In binary classification we have 1 output binary unit ( $K=1$ )

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

## Multi-class classification ( $K>1$ )

- In Multi-class classification we have  $K$  output binary units

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

The cost function is a generalisation of the Logistic regression cost function

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

$$J(\Theta) = -\frac{1}{m} \left[ \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_{\Theta}(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_{\Theta}(x^{(i)}))_k) \right]$$

$$+ \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

As you see, we follow the convention of not including the bias terms into the summation of the regularisation ( $i=0$ ).

The number of columns in our current theta matrix is equal to the number of nodes in our

current layer (including the bias unit). The number of rows in our current theta matrix is

equal to the number of nodes in the next layer (excluding the bias unit).

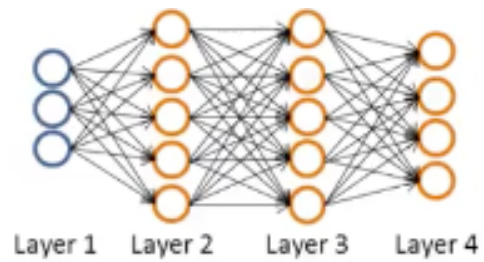
Note that the  $i$  in the triple sum does not refer to training example  $i$ .

## Backpropagation algorithm

"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression

With just one training example and  $L=4$  we have

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



The algorithm consists in calculating the error from the output layer back to the second layer.

Consider now  $m > 1$

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

After a preliminary initialization

$$\Delta_{ij}^{(l)} := 0 \text{ for all } (l, i, j)$$

for each example we have to run through a loop

**For**  $i = 1$  to  $m$   
**Set**  $a^{(1)} = x^{(i)}$   
**Perform forward propagation to compute**  $a^{(l)}$  **for**  $l = 2, 3, \dots, L$   
**Using**  $y^{(i)}$ , **compute**  $\delta^{(L)} = a^{(L)} - y^{(i)}$   
**Compute**  $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$   
 $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

The delta terms are given by

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

where we use element-wise multiplications.

The delta term of a node measures how much that node was “responsible” for any errors in our output.

For an output node, we can directly measure the difference between the network’s activation and the true target value.

For the hidden units, you will compute  $\delta^{(l)}$  based on a weighted average of the error terms of the nodes in layer  $(l + 1)$ .

The last operation in the for loop can be vectorised as follows

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



The partial derivatives of the cost function take the form

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

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

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

The term  $j=0$  does not contain the regularisation term, according to our convention.

Remember that you have to perform forward propagation and backward propagation using the first example, then you have to perform forward propagation and backward propagation using the second example, and so forth.

## Unrolling parameters for advanced optimisation

When training NNs we deal with weights parameters and deltas all of which are set of matrices.

However the costFunction (whose pointer we pass to optimising functions such as *fminunc*), accepts only vectors as arguments and return output.

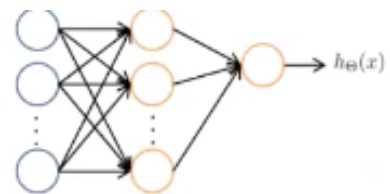
Then we need to "unroll" all the parameter matrices into a single vector.

The procedure is very simple, as shown by the following 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(:) ] ;
```

Once the D matrices have been computed throughout the forward prop/back prop, we have to unroll them in a single vector to return.

The reverse operation or unrolling can be achieved using the reshape function. For example

```

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

```

## Gradient checking

It is important to introduce a way to check the gradient function implementation.

An estimation for the partial derivatives of J are

$\theta \in \mathbb{R}^n$  (E.g.  $\theta$  is “unrolled” version of  $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ )

$\theta = \theta_1, \theta_2, \theta_3, \dots, \theta_n$

$$\frac{\partial}{\partial \Theta_j} J(\Theta) \approx \frac{J(\Theta_1, \dots, \Theta_j + \epsilon, \dots, \Theta_n) - J(\Theta_1, \dots, \Theta_j - \epsilon, \dots, \Theta_n)}{2\epsilon}$$

with epsilon about 0.0001

Then in the Octave/MATLAB implementation

```

epsilon = 1e-4;
for i = 1:n,
    thetaPlus = theta;
    thetaPlus(i) += epsilon;
    thetaMinus = theta;
    thetaMinus(i) -= epsilon;
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;

```

Now we have to check that gradApprox is pretty close to DVec obtained from back propagation. It turns out that the gradient checking is computationally intensive, so you have to turn off it before to using back propagation code to train your classifier. Otherwise the code will be very slow.

## Random Initialization

We need the initial value for the theta parameters.

With linear regression we used 0.

However it can be shown that following this strategy also for NN is not appropriate because after each update of the gradient descent, weights corresponding to inputs going into each of the hidden units are identical. In order to avoid this redundancy (or perform the so-called symmetry breaking) we need random initialisation to some suitable interval (-epsilon,epsilon).



Example:

If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11

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

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

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

rand(x,y) is just a function in octave that will initialize a matrix of random real numbers between 0 and 1.

For Theta\_l a good choice for epsilon is

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

where  $L_{in}$  is  $s_l$  and  $L_{out}$  is  $s_{l+1}$ , the number of units in the layers adjacent to Theta\_l

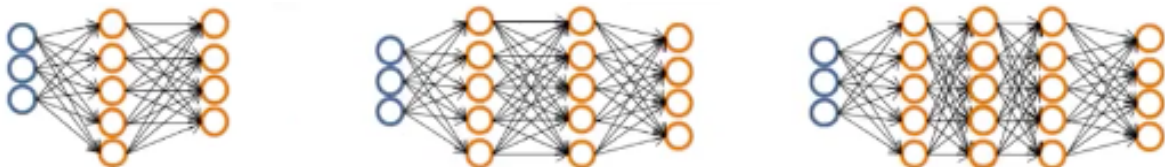
## Select the NN architecture

How to choose the architecture of the NN ?

The number of units in the input layer is equal to the number of features  $x_i$

The number of units in the output layer is equal to the number of classes (K)

The number of the hidden layers depends. Typically 1 is a good choice in most cases. But if you decide to have more than one, use the same number of units for all of them (usually more than the number of input features)



## Training the NN

### Training a neural network

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

1. Randomly initialize the weights
2. Implement forward propagation to get  $h(x)$  for any  $x$
3. Implement the cost function  $J$

4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

When we perform forward and back propagation, we loop on every training example:

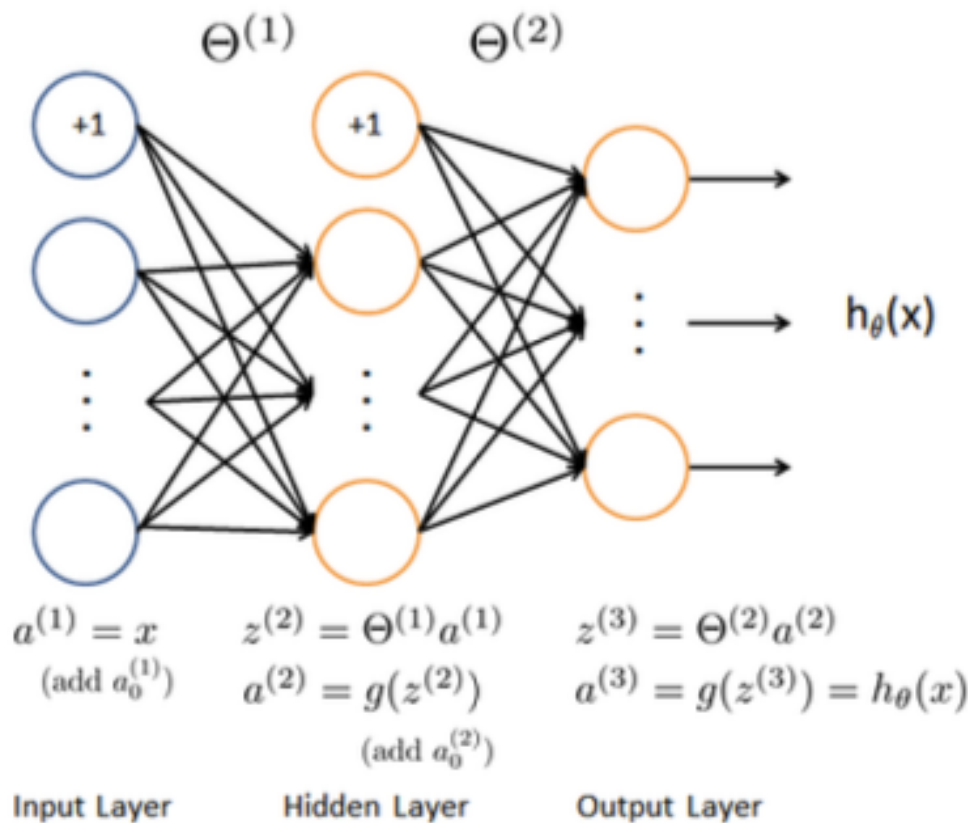
## Exercise 4

We want to implement the back propagation algorithm to train a NN to predict handwritten digits. This is a 10-class classifier ( $K=10$ ).

The dataset is taken from MNIST.

Each example is a 28x28 pixel image of a 0 to 9 digit.

The NN is made out of 3 layers ( $L=3$ ), a 400 unit input layer ( $s_1=400$ ), a 20 units hidden layer ( $s_2=20$ ) and a 10 units output layer ( $s_3=10$ ).



Let's start in Octave from a directory with required files.

```
cd /Users/gabrielefilosofi/OctaveProjects/machine-learning-ex4/ex4
```

- There are 5000 training examples in ex4data1.mat, where each training example is a 28x28 pixel grayscale image of the digit. Each pixel is represented by a floating point number indicating the grayscale pixel intensity. The 28x28 grid of pixels is "unrolled" into a 784-dimensional vector. Each of these training examples becomes a single row in our data matrix  $X$ . This gives us a 5000 by 784 matrix  $X$  where every row is a

training example for a handwritten digit image

$$X = \begin{bmatrix} \text{---} (x^{(1)})^T \text{---} \\ \text{---} (x^{(2)})^T \text{---} \\ \vdots \\ \text{---} (x^{(m)})^T \text{---} \end{bmatrix}$$

After having added a initial column of all 1s, each training example  $X(i,:)$  is a  $n+1$  vector  $x^{(i)}$ .

The second part of the training set is a 5000-dimensional vector  $y$  that contains labels for the training set. "0" digit is labeled as 10, while the digits "1" to "9" are labeled as 1 to 9. This has to be recoded as 10-vectors containing all 0s except of a 1 at the index corresponding to the label.

The matrices Theta1 and Theta2 contain the parameters for each unit in rows. The first row of Theta1 corresponds to the first hidden unit in the second layer.

- ex4.m is the main script of the exercise.

The exercise consists in editing the functions

- sigmoidGradient.m
- randInitializeWeights.m
- nnCostFunction.m

- clear all loaded variables and plots

```
clear ; close all; clc
```

- Initialise the architecture parameters

```
input_layer_size = 400; % 20x20 Input Images of Digits
```

```
hidden_layer_size = 25; % 25 hidden units
```

```
num_labels = 10; % 10 labels, from 1 to 10 (10 is for '0')
```

- Print something and load in workspace the training set matrix

```
printf('Loading and Visualizing Data ...\n')
```

```
load('ex4data1.mat'); % we load X (5000x400) and y (5000x1) matrices
```

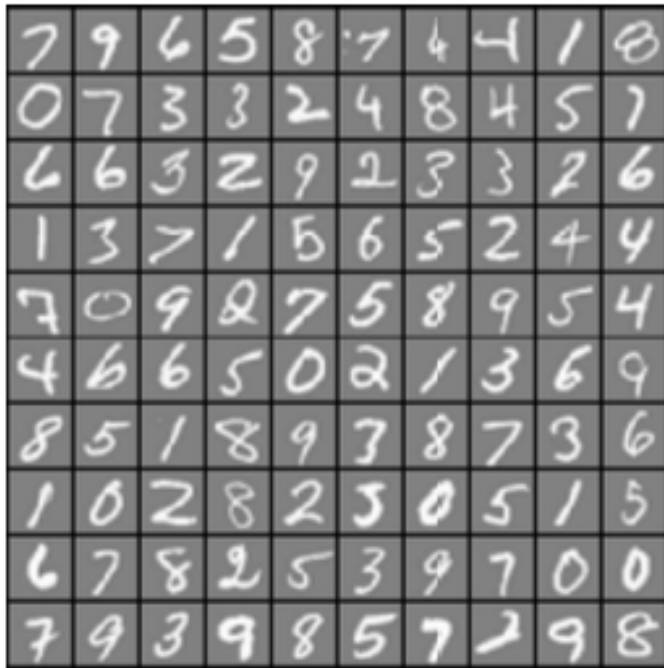
```
m = size(X, 1); % gives 5000, the number of examples or X's rows
```

- Randomly select 100 data points from X and display them in a 10x10 grid

```
sel = randperm(m); % generate a vector with random permutation of 1..m
```

```
sel = sel(1:100); % keep only 100 of them
```

```
displayData(X(sel, :)); % display the selected training examples in a 2D grid
```



- Load the weights into variables Theta1 and Theta2  
`load("ex4weights.mat");`

- Unroll the weights into a single long vector (10285x1)  
`nn_params = [Theta1(:) ; Theta2(:)];`

- Weight regularization parameter (we set this to 0 here).  
`lambda = 0;`

```
J = nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels,
X, y, lambda);
```

- Open the nnCostFunction and implement the feedforward cost without regularization first (so that it will be easier for you to debug)

- Reconstruct the original matrices from the unrolled vector  
`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));
```

- Remap the output label vector y to a one-vs-all matrix Y  
`m = size(X, 1);`  
`I = eye(num_labels);`  
`Y = zeros(m, num_labels);`  
`for i = 1:m`

```
Y(i,:) = l(y(i,:),:);
end
```

- Feedforward computation of nodes activation

```
a1 = [ones(m,1), X];
z2 = a1*Theta1';
a2 = sigmoid(z2);
a2 = [ones(size(z2,1),1) a2];
z3 = a2*Theta2';
a3 = sigmoid(z3);
h = a3;
```

- Implement the unregularized cost function

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K \left[ -y_k^{(i)} \log((h_{\theta}(x^{(i)}))_k) - (1 - y_k^{(i)}) \log(1 - (h_{\theta}(x^{(i)}))_k) \right]$$

```
J = 0;
J = sum(sum((-Y).*log(h) - (1-Y).*log(1-h), 2))/m;
```

- Implement the regularisation term and add it to J

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

$$J(\Theta) = -\frac{1}{m} \left[ \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_{\Theta}(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_{\Theta}(x^{(i)}))_k) \right]$$

$$+ \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

```
reg = sum(sum(Theta1(:, 2:end).^2, 2))+sum(sum(Theta2(:, 2:end).^2, 2));
reg = lambda*reg/(2*m);
J = J + reg;
```

- Now we need to complete the nnCostFunction.m so that it returns an appropriate value for grad J. Once we have computed the gradient, we will be able to train the NN by minimizing  $J(\Theta)$  using an advanced optimizer such as *fmincg*.
- implement the sigmoid gradient function

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1 - g(z))$$

```
function g = sigmoidGradient(z)
g = zeros(size(z));
g = sigmoid(z).*(1 - sigmoid(z));
```

To check it works you should get 0.25 when executing `sigmoidGradient(0)`

- Let's edit the function `randomInitialization.m`

```
function W = randInitializeWeights(L_in, L_out)
W = zeros(L_out, 1 + L_in);
epsilon_init = 0.12;
W = rand(L_out, 1 + L_in)*2*epsilon_init - epsilon_init;
end
```

Note that `W` should be set to a matrix of size `(L_out, 1 + L_in)` as the first column of `W` handles the "bias" terms

- Calculate sigmas

```
sigma3 = a3.-Y;
sigma2 = (sigma3*Theta2).*sigmoidGradient([ones(size(z2, 1), 1) z2]);
sigma2 = sigma2(:, 2:end);
```

- Accumulate gradients

```
delta_1 = (sigma2'*a1);
delta_2 = (sigma3'*a2);
```

- Calculate unregularized gradient

```
Theta1_grad = delta_1./m;
Theta2_grad = delta_2./m;
```

- Calculate regularization terms and add to the gradient

```
reg1 = (lambda/m)*[zeros(size(Theta1, 1), 1) Theta1(:, 2:end)];
reg2 = (lambda/m)*[zeros(size(Theta2, 1), 1) Theta2(:, 2:end)];
Theta1_grad += reg1;
Theta2_grad += reg2;
```

## Visualize the activation of the hidden units

One way to understand what the NN is learning is to visualize what the representations captured by the hidden units.

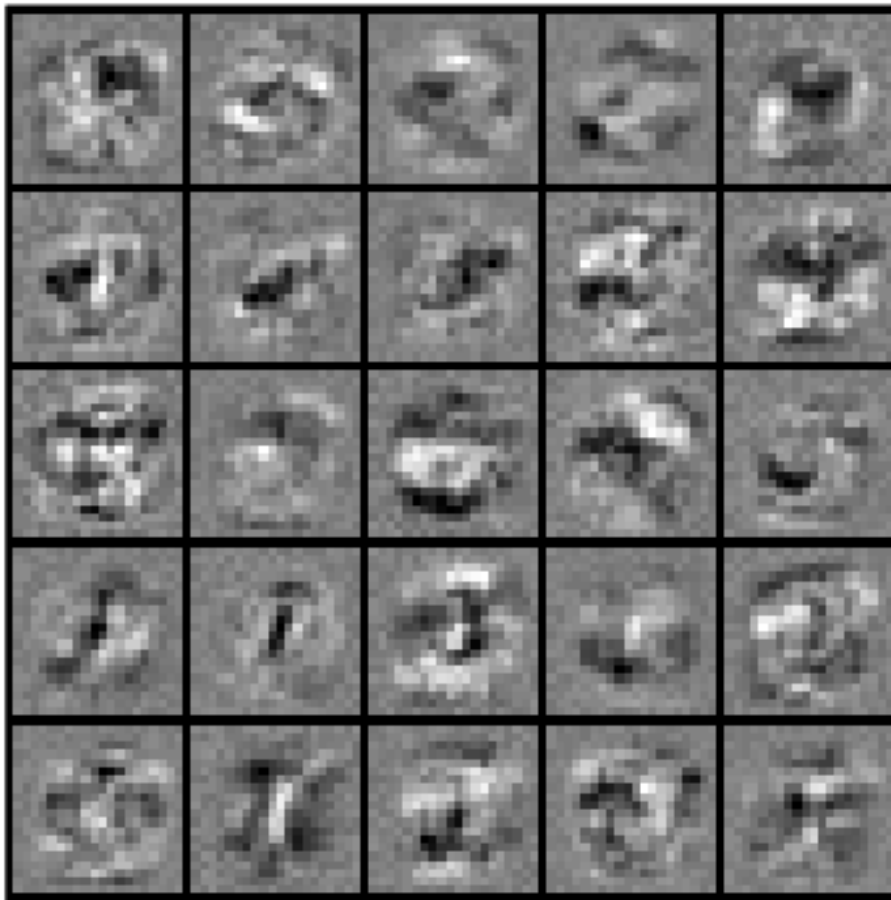
Given a particular hidden unit, one way to visualize what it computes is to find an

input  $x$  that will cause it to activate (that is, to have an activation value close to 1).

For the NN we trained, notice that the  $i$ -th row of  $\Theta(1)$  is a 401-dimensional vector that represents the parameter for the  $i$ -th hidden unit. If we discard the bias term, we get a 400 dimensional vector that represents the weights from each input pixel to the hidden unit. Thus, one way to visualize the "representation" captured by the hidden unit is to reshape this 400 dimensional

vector into a 20x20 image and display it.

It turns out that this is equivalent to finding the input that gives the highest activation for the hidden unit, given a “norm” constraint on the input.



In our trained network, the hidden units correspond roughly to detectors that look for strokes and other patterns in the input.

## Overfitting

The performance of the NN varies with the regularization parameter  $\lambda$  and the number of training steps (MaxIter option when using fmincg).

Without regularization, it is possible for a NN to “overfit” a training set so that it obtains close to 100% accuracy on the training set but does not as well on new examples that it has not seen before.

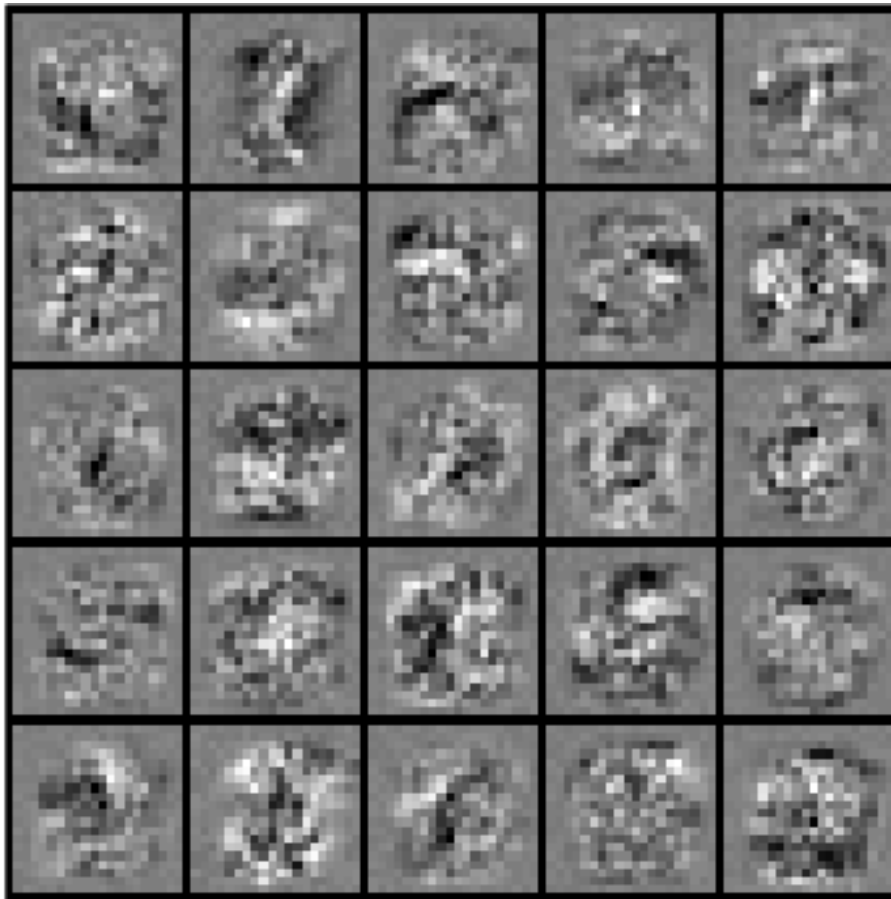
We can set the regularization  $\lambda$  to a smaller value and the MaxIter parameter to a higher number of iterations to see the effects.

Example:

changing  $\lambda$  from 1 to 0.3 and maxIter from 50 to 150

We achieve 99.9% accuracy over the training set.

The activation of the units belonging to the second layer looks like this



## Improving a learning algorithm

In general the error of your hypothesis as measured on the data set with which you trained the parameters will be lower than the error on any other data set. It may happen that you train a prediction model over a data set obtaining a high accuracy, and then moving to a new data set you may find out the accuracy is not so good.

There are a number of possible options you may apply trying to improve the model

- get more training examples (usually fixes high variance problems)
- try a smaller set of feature (usually fixes high variance problems)
- try getting additional features (usually fixes high bias problems)
- try adding polynomial features (usually fixes high bias problems)
- try decreasing  $\lambda$  (usually fixes high bias problems)
- try increasing  $\lambda$  (usually fixes high variance problems)

For a NN you can

- try a smaller architecture (cheaper but more prone to underfitting)
- try a larger architecture (computationally expensive and prone to overfitting if not regularized)



Many people spend months following one of the avenues above without a good rationale.

Fortunately there are some diagnostic tests to guide you

## Evaluating the hypothesis

Just because a learning algorithm fits a training set well, that does not mean it is a good hypothesis.

To evaluate a hypothesis given a dataset of training examples, we should split up the dataset into two sets:

- Training set (70% of the examples)
- Test set (30% of the examples)

The new procedure using these two sets is then

1. Train the model and find theta parameters using the Training set
2. Compute the test set error  $J$  using the Test set examples

## Model selection

You might want to determine the best polynomial degree ( $d$ ) or the best  $\lambda$  for the hypothesis. Now  $d$  (or  $\lambda$ ) represents an additional parameter. Then you have to split the dataset this way

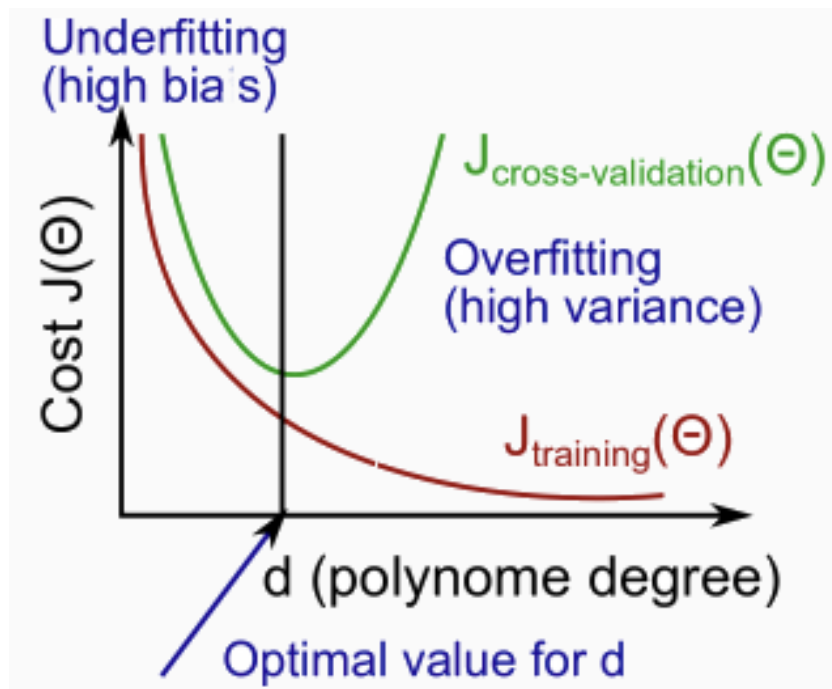
- Training set (60%)
- Cross Validation set (20%)
- Test set (20%)

It is important to understand that the example data over which you select your model, the example data over which you train the selected model and the example data over which you test the generalization accuracy of your model MUST BE DIFFERENT!!!

## Diagnose Bias vs Variance problems

When you plot the errors as a function of  $d$ :

- High bias or underfitting:  $J_{cv}$  and  $J_{train}$  are both high and with similar values
- High variance or overfitting:  $J_{cv} \gg J_{train}$



When you plot the errors as a function of  $\lambda$ :

- High bias or underfitting:  $J_{\text{cv}} \gg J_{\text{train}}$
- High variance or overfitting:  $J_{\text{cv}}$  and  $J_{\text{train}}$  are both high and with similar values

Note that the CV error is computed without regularisation.

Typically the selection process for  $\lambda$  uses the following guess values

0, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 1.28, 2.56, 5.12, 10.24

## Learning Curves

We plot  $J_{\text{train}}$  or  $J_{\text{cv}}$  as a function of  $m$  (training set size).

For small  $m$   $J_{\text{train}}$  are close or identical to 0.

$J_{\text{train}}$  increases with increasing  $m$

$J_{\text{cv}}$  decreases with increasing  $m$

- High bias or underfitting:  $J_{\text{cv}}$  and  $J_{\text{train}}$  quickly converge to the same (this is because increasing  $m$  will not help much)
- High variance or overfitting:  $J_{\text{cv}}$  and  $J_{\text{train}}$  converge slowly (getting more training data is likely to help)

## Other

### The IDX file format

The IDX file format is a simple format to represent multidimensional matrices.

The format is

*magic number (uint32)*

*size in dim 0 (uint32)*  
*size in dim 1 (uint32)*  
..  
*size in dim N (uint32)*  
*data*

The magic number tells the type of data (0x08: uchar, .., 0x0E: double) and the number of dimensions (N). All numbers are represented in Big Endian.

## Troubleshooting

ex3 gives:

*Testing lrCostFunction() with regularization*  
*Cost: 3.734819*  
*Expected cost:2.534819*

But, I can't figure out what's wrong with my implementation for the regularised cost function below

$h = \text{sigmoid}(X * \theta);$   
 $J = y' * \log(h) + (1 - y)' * \log(1 - h) - \lambda * \theta' * \theta / 2;$   
 $J = -J / m;$

Please give me a hint