Machine Learning - Andrew Ng - Stanford

- Arthur Samuel: The field of study that gives computers the ability to learn without being explicitly programed to do so.

- Tom Mitchel: "A computer program is said to learn from experience "E" with respect of some class T and performance measure "P", if it's performance at tasks in "T", as measured by "P", improves with experience "E" "

Example playing checkers:

E = the experience of playing many times checkers 10K +

T = the task of playing checkers

P = the probability that the program will win the next game

In general, any machine learning can be assigned to one of two broad classifications:

- Supervised learning: we give answers for examples
- Unsupervised learning

- Supervised learning

In supervised learning, we are given a dataset and already know what our correct output should look like, having the idea that there is a ~~relationship~~ relationship between the input and the output.

Supervised learning problems are categorized into "regression" and "classification" problems. In a regression problem, we are trying to predict results within a continuous output, meaning that we are trying to map input values into a continuous function. In classification problem, we are trying to predict results in a discrete output.

Example 1: Given the size of houses on the real estate market, try to predict their price. The price as a function of size is a continuous output (numerical value)

We can turn the example into a classification problem, if we try to predict if a house will sell for more or less than the asking price we are getting two discrete (label) values, 'yes' or 'not'.

Example 2: Given a picture of a person, we have to predict
a) Regression: their age on the basis of the given picture

b) Classification: Given a patient with a tumor, we have to predict whether the tumor is malignant or benign.

Unsupervised learning:

Unsupervised Learning allow us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables.

We can derive this structure by clustering the data based on relationships among the variables in the data.

Example clustering: Taking a group of 1000,000 ~~genes~~ genes, and find a way to automatically group these genes into groups that are somehow similar or related to different variables such as lifespan, location, roles, etc.

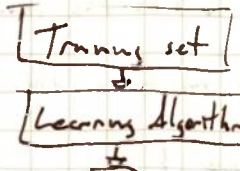
Example Non Clustering: The 'cocktail party algorithm', allows us to find structure in a chaotic environment, for ~~example~~ example, identifying individual voices and music from a mesh of sounds at a cocktail party.

Linear Regression

Through the course superscripts will be used to denote the "i th" input variable.

We'll use $x^{(i)}$ to denote the input variables and $y^{(i)}$ the output or target variable that we are trying to predict; a pair $(x^{(i)}, y^{(i)})$ is called a training example and 'm' is used to denote the training set

h = hypothesis



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

θ_i's = Parameters

If the target variable is continuous, we call this learning problem a regression problem. Classification is when we want to predict discrete values.

We can measure the accuracy of our hypothesis function by using a cost function. This takes an average difference (actually a fancier version of average) of all the results of the hypothesis with inputs from X's and the actual output Y's.

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

In cost function, not in gradient descent

to break it apart, it is $\frac{1}{2} \bar{x}$ where $\bar{x}$ is the mean of the squares of $h_\theta(x_i) - y_i$, or the difference between the predicted value and the actual value.

This function is otherwise called the "Squared error function" or "mean squared error". The mean is halved ($\frac{1}{2}$) as a convenience for the computation of the gradient ~~descent~~ descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term.

Cost Function – Intuition I

If we try to think of it in visual terms, our training data set is scattered on the x-y plane. So we are trying to make a straight line (defined by $h_\theta(x)$) which passes through these scattered data points.

The objective is to get the best possible line. The best line will be such that the average squared vertical distances of the scattered points from the line will be the least.

Gradient Descent – to minimize the cost function J

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{d}{d\theta_j} J(\theta_0, \theta_1) \quad (\text{for } j = 0 \text{ and } j = 1)$$

}

α – learning rate

Correct simultaneous update

$$\text{temp}0 := \theta_0 - \alpha \frac{d}{d\theta_0} J(\theta_0, \theta_1)$$
$$\text{temp}1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_0, \theta_1)$$

$\theta_0 := \text{Temp}1$
$\theta_1 := \text{temp}2$

The learning rate can be set big or small, having small 'baby steps' will take longer but will avoid overshooting, thing that happens if α is too big.

The intuition behind the convergence is that $\frac{d}{d\theta} J(\theta_1)$ will approach to $\emptyset$ as we approach the bottom of our convex function. Once at the minimum, the derivative will be always $\emptyset$ (zero), thus we get

$$\theta_1 := \theta_1 - \alpha \cdot 0 \quad \text{which is}$$
$$\theta_1 := \theta_1 - 0 \quad \therefore \quad \theta_1 = \theta_1$$

Gradient Descent for Linear Regression

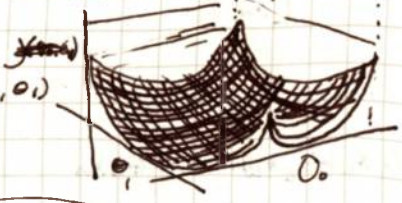
repeat until convergence {

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

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

} → Simultaneously

Cost function for linear regression will be always a convex function (bowl shape)



Multivariate Linear Regression

Notation:

n = number of features m = sample size

$x^{(i)}$ = input (features) of 'i th' training example: $x^{(3)}$
- not exponent but superscript

$x_j^{(i)}$ = value of feature 'j' in 'i th' training example

Size (feet²)	Number of bedrooms	Number of floors	Age of Home (years)	Price (k$) "y"
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

m = 47 samples

$x^{(2)} = \begin{bmatrix} 1416 \\ 3 \\ 2 \\ 40 \end{bmatrix}$ → $x_3^{(2)} = 2$

The regression formula for hypothesis now becomes

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

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

For convenience of notation, we define x_0 as 1

$x_0 = 1$

$$\theta = \underbrace{[\theta_0, \theta_1, \theta_2, \cdots \theta_n]}_{\theta^T} \qquad \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$(n+1) \times 1$ matrix

$$\boxed{\theta^T x = h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n}$$

For gradient descent with more than one feature, the formula changes:

repeat until convergence {

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

(simultaneously update θ_j for $j = 0 \ldots j = n$)

}

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

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_2^{(i)}$$

...

...

...

Feature normalization / Scaling

Put all the variables between reasonable small limits

Mean normalization (don't apply it to the bias)

Examples

$$x_i := \frac{\text{value} - \text{average}}{\max - \min}$$

$$x_1 := \frac{\text{size} - 1000}{2000}$$

$$x_2 := \frac{\text{bedrooms} - 2}{5}$$

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

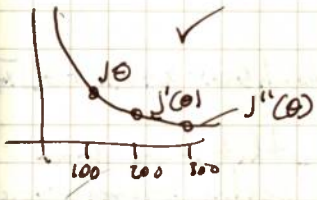
μ_i ← mean

s_i ← max − min or standard deviation.

Choose the right learning rate

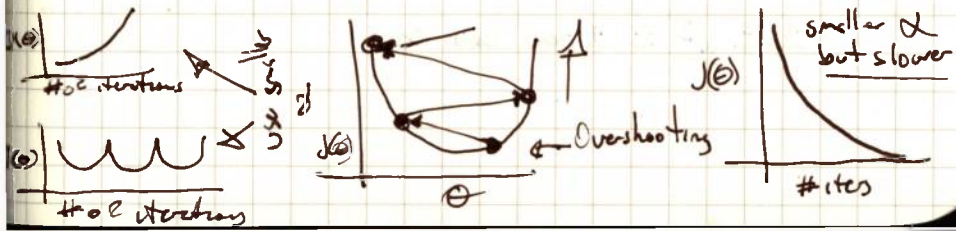
The best approach to test and choose the right learning rate, is to debug it.

By plotting the cost function across each iteration: if the cost function $J(\theta)$ ever increases, then the learning rate should be decreased.



A way to declare convergence is if $J(\theta)$ decreases less than 'ε' in one iteration, where 'ε' is some small value such as 10^{-3} (0.001). However is difficult in practice to choose this threshold value.

It has been mathematically proved that if learning rate α is sufficiently small, then $J(\theta)$ will decrease on every iteration



Features and Polynomial Regression

We can improve our features and the form of our hypothesis function in a couple different ways.

We can combine different features into one, for example, we can combine x_1 and x_2 into a new feature x_3 by taking $x_1 \cdot x_2$

- Polynomial regression

~~Our hypothetic hypothesis example~~
Our hypothesis function need not to be linear (straight line) if that does not fit the data well.

We can change the behavior of the curve of our hypothesis function by making it a quadratic, cubic or square ~~function~~ root function (or any other form)

For example, if our hypothesis function is $h_\theta(x) = \theta_0 + \theta_1 x_1$, then we can create additional features based on x_1, to get the quadratic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$ or the cubic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$

In the cubic version we've made new features x_2 and x_3 where $x_2 = x_1^2$ and $x_3 = x_1^3$

To make a square root function, we could do: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 \sqrt{x_1}$

One important thing to keep in mind is that if we choose our features this way, then feature scaling becomes crucial.

eg, if x_1 has a range of $1 \leq x_1 \leq 1000$, then

$$1 \leq x^2 \leq 1\,000\,000 \text{ and}$$

$$1 \leq x^3 \leq 1\,000\,000\,000$$

Normal equation

Gradient descent gives a way of minimizing J. But there is a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In the normal equation method, we will minimize J by explicitly taking it's derivatives with respect to the θ_j's, and setting them to zero. This allows us to find the optimum theta without iteration.

The normal equation formula is given below:

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

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

Comparison between gradient descent and the normal equation.

Gradient Descent	Normal Equation
(-) Need to choose α Needs many iterations	(+) No need to choose α No need to iterate
(+) $O(kn^2)$ Works well when 'n' is large	$O(n^3)$, need to calculate inverse of $X^T X$ (-) Slow if 'n' is very large Is cube of the dimension of the matrix

In practice, a large set of features will imply problems in current computing; if $n < 10000$ still is viable, but if $n > 10000$, then iterative process with gradient descent.

This do not work on logistic regression or classification.

Normal Equation Noninvertibility

What if $X^T X$ is non-invertible? What to do if happens?

- Redundant features:

  x_1 = size in $feet^2$
  x_2 = size in m^2

- Too many features ($m \leq n$)
  - not a good idea
  - delete some features, or use regularization

Week 3 – Logistic Regression

Logistic regression is a method for classifying data into discrete outcomes. For example, to classify transactions as valid or fraud, email as spam or not, etc.

In this module we will introduce the notion of classification, the cost function for logistic regression and the application of logistic regression to multi-class classification.

We we also covering regularization. Machine Learning models need to generalise well to new examples that the model has not seen in practice. This will be used to prevent overfitting the training data.

Classification

Linear regression is not a suggested algorithm, it fails to encompass all the features properly and ~~also~~ miss out, because it's not a problem for a linear function.

A classification problem is similar to a regression problem but the values we want now to predict are a small number of discrete values. For now we will focus on binary classification which can take only 2 values, 0 and 1.

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

0 is usually called the negative class and denotes absence of something

1 is usually called the positive class and denotes presence of something

They may be also denoted by symbols '+' and '-'. Given the $x^{(i)}$, the corresponding $y^{(i)}$ is also called the label for the training example.

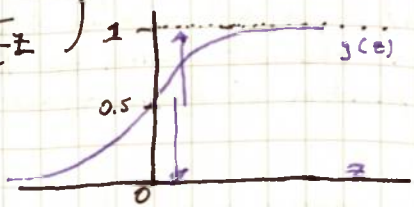
Hypothesis representation

Logistic Regression model: Want $0 \le h_\theta(x) \le 1$

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

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

↳ Sigmoid function.
↳ Logistic function



parameters θ

$h_\theta(x)$ = estimated probability that $y=1$ on input x $P(y=1|x)$

$x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ \text{tumor size} \end{bmatrix}$ $h_\theta(x) = 0.7$ → 70% chance

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

As $y \in \{0,1\}$

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

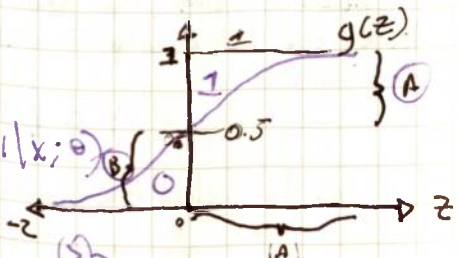
$\therefore P(y=0|x;\theta) = 1 - P(y=1|x;\theta)$ just a subtraction

Decision Boundary

We've got the formulas

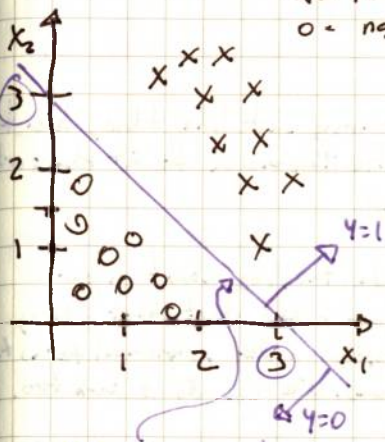
$h_\theta(x) = g(\theta^T x) = P(y=1|x;\theta)$

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



suppose predict $y=1$ if $h_\theta(x) \ge 0.5$ (A)
predict $y=0$ if $h_\theta(x) < 0.5$ (B)

x = positive example
o = negative example



$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

test: -3 1 1

Predict "$y = 1$" if $-3 + x_1 + x_2 \geq 0$

$$\theta = \begin{bmatrix} -3 \\ 1 \\ 1 \end{bmatrix} \quad \theta^T = [-3 \;\; 1 \;\; 1]$$

$y = 1$ if $\theta^T \geq 0$

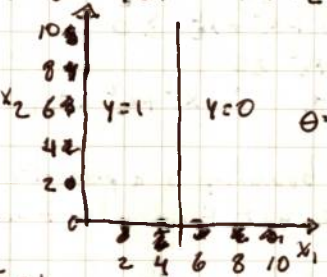
$-3 + x_1 + x_2 \geq 0$

$x_1 + x_2 \geq 3$

$x_1 + x_2 = 3$

Quiz:

Consider logistic regression with two features x_1 and x_2. Suppose $\theta_0 = 5$, $\theta_1 = -1$ and $\theta_2 = 0$, so that $h_\theta(x) = g(5 - x_i)$.



Less than 5 = 1
more than 5 = 0

$5 + (-1)x_1 + 0x_2 \geq 0$

$\theta = \begin{bmatrix} 5 \\ -1 \\ 0 \end{bmatrix}$

$5 - x_1 \geq 0 = x_1 \leq 5$

$-x_1 \geq -5$

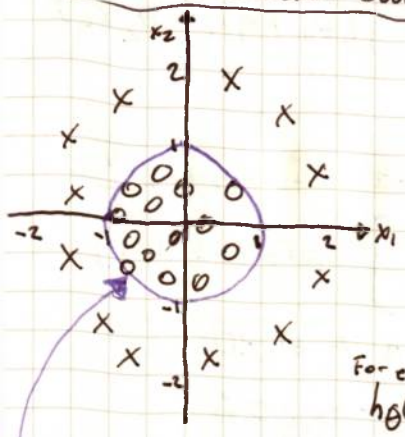
(* -1) multiply by -1

$x_1 \leq 5 = 1$

$\therefore x = 0$

$x > 5 = 0$

Non linear decision boundaries



Given a training set like this, how to get Logistic Regression to fit this sort of data?

Earlier while talking about linear regression we talked about about adding extra higher order polynomial terms to the features. We can do it here too.

For example

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$$

For example: in the figure above, it was the same test/example

hypothesis: $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$

sample values $\Rightarrow$ -1 0 0 1 1

$\theta = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$ Predict "$y=1$" if $-1 + 0 \times x_1 + 0 \times x_2 + 1 \times x_1^2 + 1 \times x_2^2 \geq 0$

$= -1 + 0 + 0 + x_1^2 + x_2^2 \geq 0$

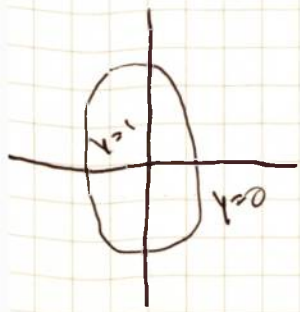
$= -1 + x_1^2 + x_2^2 \geq 0$

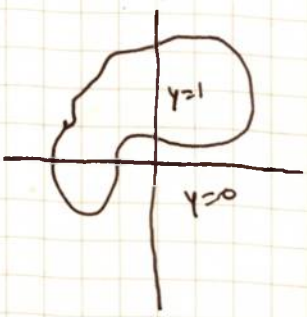
$= x_1^2 + x_2^2 \geq 1$ $\therefore$ $x_1^2 + x_2^2 = 1$ Decision Boundary

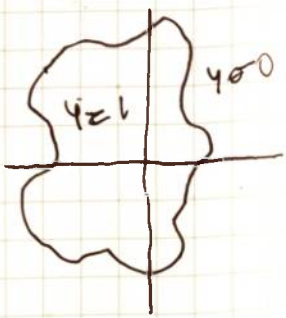
Important is to say, the decision boundary is a property of the hypothesis and the parameters, not from the training set.

The training set can be used to fit the parameters θ.

Given that, even more complex formulas may be used, which would have a representation as an ellipse or another shape on the hyperplane. ie.:







Logistic Regression : Cost function

Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), \cdots, (x^{(m)}, y^{(m)})\}$

m examples

$x \in \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \cdots \\ x_n \end{bmatrix}$ Vector

$x_0 = 1$, $y \in \{0, 1\}$

x_0: constant

$x_0 = 1$ always

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

How to choose parameters for θ?

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

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

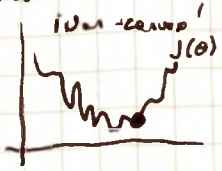
$$\text{cost}\left(h_\theta(x^{(i)}, y^{(i)}\right)$$

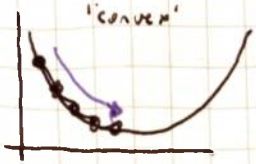
$$\rightarrow \text{Cost}\left(h_\theta(x^{(i)}), y^{(i)}\right) = \frac{1}{2} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$ to make it looks simpler

Cost function adopted but NOT OPTIMAL

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

This cost function works fine for linear regression, but here we are trying to do logistic regression, to find labels, not predict values. So if we could minimize the function cost for J, that would work ok, but if we use this function, it would be a 'non' convex function $J(\theta)$.

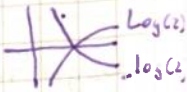




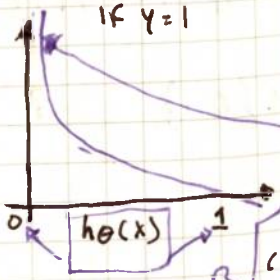
Non convex: it can fall in local minima but not guaranteed on global minima

Convex : the bowl shaped, it's guaranteed to find global minima

Logistic regression cost function



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

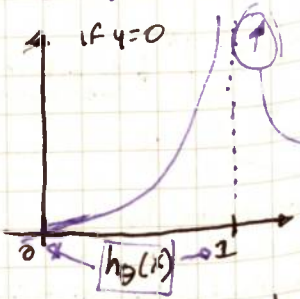


Cost is $\emptyset$ if $y=1$ and $h_\theta(x)=1$ (A)
But as $h_\theta(x) \to \emptyset$
Cost $\to \infty$
] because of the logarithmic nature

(B) Thus captures the intuition that if $h_\theta(x)=0$, (predict $P(y=1|x;\theta)=0$) but $y=1$, we'll penalize the learning algorithm by a very large cost.

(A) As $h_\theta(x) \geq 0.5 = 1$ (due to sigmoid), and the expected value matches exactly (1), then the cost is $\emptyset$; therefore the text in "B" explains why would be impossible to say that if $y=1$, that $h_\theta(x)$ would be $\emptyset$ or ≤ 0.5 on sigmoids

So that's why the cost approaches infinity, if $y=1$ but $h_\theta(x)=0$, because it's wrong, very very wrong.



What this curve does, it goes up, and goes to plus infinity as $h_\theta(x)$ goes to 1 because as said, that if $y=0$ but we predicted $y=1$ with almost certainty, it would have a great cost.

And controversely, if $h_\theta(x)=\emptyset$ and $y=0$, at this point the cost function is going to be zero. The hypotheses melted.

Simplified cost function and Gradient Descent

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

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

Note: $y = 1$ or $y = 0$ always

Then it can be written as:

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

Demonstration:

$1 - 1 = 0$ ∴

if $y = 1$: $Cost(h_\theta(x), y) = -1\log(h_\theta(x)) - (1-1)\log(1-h_\theta(x))$

$= -1\log(h_\theta(x))$

if $y = 0$: $Cost(h_\theta(x), y) = -0\log(h_\theta(x)) - (1-0)\log(1-h_\theta(x))$

$= -1\log(1-h_\theta(x))$

Logistic Regression Cost function:

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

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

This is a function or cost function coming from statistics, using the principle of maximum likelyhood estimation. Which is an idea in statistics for how to efficiently find parameters data for different models. Also it has a nice property, that it is convex.

Next step is to fit parameters to θ, so we need to minimize these cost function and then make a prediction given new x:

fit parameters θ: $\min_\theta J(\theta)$

To make a prediction given new x:

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

Gradient Descent

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

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

repeat {

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

(simultaneously update all θ_j)

}

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

$\therefore$

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

repeat {

FORMULA

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

(simultaneously update all θ)

}

And the algorithm looks identical to linear regression!

- For linear regression, $h_\theta(x)$ was $\theta^T x$
- For logistic regression, $h_\theta(x)$ is $\frac{1}{1+e^{-\theta^T x}}$

Quiz: Suppose you are running gradient descent to fit a logistic regression model with parameter $\theta \in \mathbb{R}^{n+1}$. Which of the following is a reasonable way to make sure the learning rate α is set properly and the gradient descent is running correctly.

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

While running logistic regression with gradient descent, we have all of the different parameter values $\theta_0 \cdots \theta_n$; we could use a for loop to update their values, but also it's possible to use vectorization to update all the parameters in one step:

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

Advanced Optimization

As we know, it's possible to use gradient descent.

There are other algorithms, more advanced and sophisticated ones, that if we provide them a way to compute $J(\theta)$ and $\frac{\partial}{\partial \theta} J(\theta)$, then these algorithms will compute the cost function for us.

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

Advantages:
- no need to manually pick α
- often faster than gradient descent

Disadvantages:
- more complex

These are algorithms that are able to tune up automatically the learning rate on each iteration, and tend to converge faster.

How to use this algorithms?

Example $\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$ $J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$

$\frac{\partial}{\partial \theta_1} = 2(\theta_1 - 5)$

$\frac{\partial}{\partial \theta_2} = 2(\theta_2 - 5)$

```
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);
```

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

then:

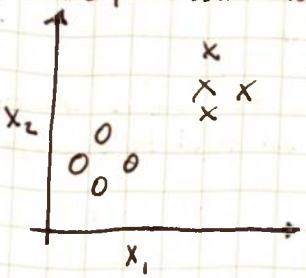
```
options = optimset('GradObj', 'On', 'MaxIter', '100');
initialTheta = zeros(2,1)
[optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta, options);
```

Multiclass Classification: One vs All classification

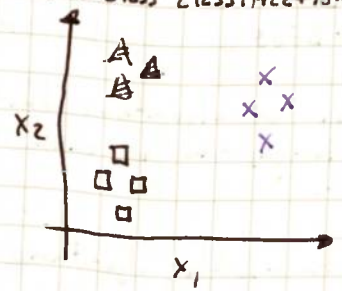
If we want to classify several options such as:

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

Binary Classification

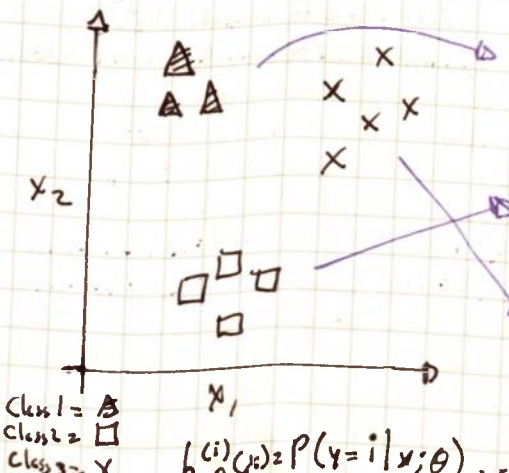


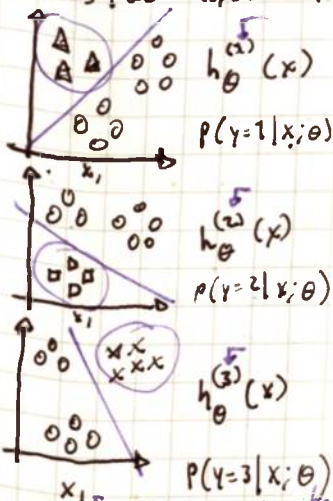
Multi-class classification



On this model, the challenge is to determine the class where an hypothesis belongs. For that, what we do w to do, in the example, is to do 3 different classifications, in other words, make 3 different binary classifications.

We create three different binary classifications, we'll separate only 2 classes, one at a time:





$h_\theta^{(i)}(x) = P(y=i|x;\theta)$ $i = 1, 2, 3$

For a new sample, choose the class i that maximizes the one with higher probability

[Solving the problem of overfitting]

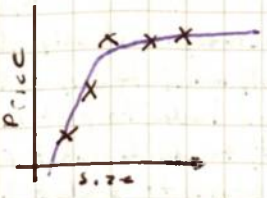
Overfitting and it's problem

Overfitting describes a situation where the model learns to describe random error or noise ε instead of 'real' meaningful relationships between features and the output. In other words, when it fits the training sample "too exactly, too accurate", and new samples, unseen to the model, can't and won't fit.

Example with **linear** regression (Housing prices)

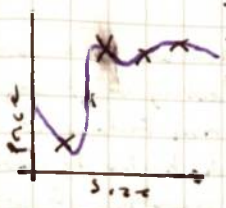
$\theta_0 + \theta_1 x$

"Underfitting" or "high bias", doesn't fit very well



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

"Just right"

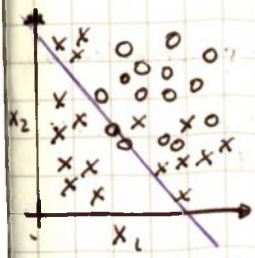


$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

"Overfit", "high variance"

Thus, if we have too many features, the learned hypothesis may fit the training set very well ($J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \approx 0$), but fail to generalize to new examples

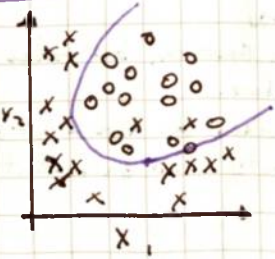
Example with **logistic** regression



$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$

g = sigmoid

"Underfit"



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

"Just right"

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

"Overfit"

Addressing Overfitting

In practice, we will have sometimes several attributes/features

x_1 = size of house
x_2 = no. of bedrooms
x_3 = no of floors
x_4 = age of the house
x_5 = average income in neighborhood
x_6 = kitchen size
x_{100}

When we have several features, is not only about a matter of just selecting what degree polynomial to use, and in fact, when we have so many features it becomes harder to plot the data and it becomes much harder to visualize it, to decide what features to keep or not. So concretly, if we are trying to predict housing prices, sometimes we can have lots of different features, and they are seen kind of useful, but if we have a lot of features and very little training data, then overfitting can become a problem.

however, throwing away features is like throwing away information we have about the problem...

Options:

1) Reduce number of features
   - Manually select which features to keep
   - Model selection algorithm (later in course)

2) Regularization
   - Keep all the features, but reduce the magnitude/values of θ parameters.
   - Works well when we have a lot of features, each of which contributes a bit on predicting y

Cost function

If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost.

Say we wanted to make the following function more quadratic:

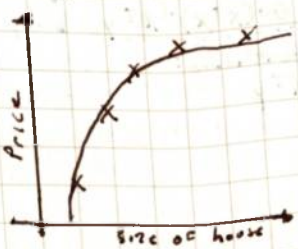
$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

We will want to reduce (almost eliminate) the influence of $\theta_3 x^3$ and $\theta_4 x^4$ without actually getting rid of these features or changing the form of our hypothesis, then we can instead modify our cost function:

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

We've added two extra terms at the end to inflate the cost of θ_3 and θ_4. Now in order to get close to zero, we will have to reduce the values of θ_3 and θ_4 near to zero. This will in turn greatly reduce the values of $\theta_3 x^3$ and $\theta_4 x^4$ in our hypothesis function. As a result, we see that the new hypothesis (depicted by the blue curve) looks like a quadratic function but fits the data better due the extra small terms $\theta_3 x^3$ and $\theta_4 x^4$.

Intuition:



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

$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

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

$\theta_3 \approx 0,\ \theta_4 \approx 0$

We also could regularize all of our ~~data~~ theta parameters in a single summation as:

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

The λ or lambda, is the regularization parameter, and determines how much the costs of θ are inflated. With that, we can smooth the output of our hypothesis function to reduce overfitting. If λ is chosen to be too large, it may smooth too much and cause underfitting.

[Regularized Linear Regression]

We can apply regularization to both linear regression and logistic regression. On the case of linear regression:

Gradient descent

We will modify our gradient descent function to separate out θ_0 from the rest of the parameters because we don't want to penalize θ_0.

repeat {

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

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

} $\quad j \in \{1, 2, \ldots, n\}$

The term $\frac{\lambda}{m}\theta_j$ performs our regularization. With some manipulation our update rule can also be represented as:

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

The first term in the above equation, $1 - \alpha \frac{\lambda}{m}$ will always be less than 1 (< 1). Intuitively can be seen as reducing the value of θ_j on every update. Notice the second term is exactly the same as it was before.

Normal Equation

Using the alternate method of the non-interactive normal equation, we use the same formula as before: $\theta = (X^T X)^{-1} X^T y$ but, now we add another term inside the parenthesis:

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

where $L = \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}$ or $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \vdots & & & \ddots & \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$ $\quad (n+1) \times (n+1)$

$m \leq n$ (#examples, #features)

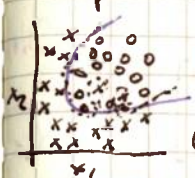
L is a matrix with $\emptyset$ at the top left and 1's down the diagonal with $\emptyset$'s everywhere else. It should have the dimension $(n+1) \times (n+1)$.

Intuitively, this is an identity matrix (though we are not including x_0), multiplied with a single real number λ.

Important to recall that if $m < n$ (examples are less than features), then $X^T X$ is non-invertible. However when we add the term λL, then $X^T X + \lambda \cdot L$ becomes invertible.

[Regularized Logistic Regression]

Similar as regularized linear regression, now we will adapt our functions to optimize the or regularize logistic regression.



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

.....: Dotted line = overfitted
——: Blue line = regularized

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

$+ \left[\frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2\right]$ ← Regularization term

$\theta_1, \theta_2 \dots \theta_n$

That's our cost function for logistic regression and we've modified it to use regularization, we add $\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$. Which also has the effect of penalizing parameters θ_j for being too large.

Gradient descent:

repeat {

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

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

$(j = 1, 2, 3, \dots, n)$

$\theta_1 \dots \theta_n$

$\}$ $\frac{d}{d\theta} J(\theta)$

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

Advanced optimization

Function [jVal, gradient] = costFunction(theta)

jVal = [code to compute $J(\theta)$];

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

$\theta = \begin{bmatrix}\theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_n\end{bmatrix}$ index starts at 1 in Octave; theta1, theta2, theta3, theta n+1

index=2 gradient(1) = [code to compute $\frac{d}{d\theta_0} J(\theta)$]; ← zero ← leave θ_0 intact

% $\frac{1}{m}\sum (h_\theta(x^{(i)}) - y^{(i)})\, x_0^{(i)}$ ← zero

gradient(2) = [code to compute $\frac{d}{d\theta} J(\theta)$];

% $\frac{1}{m}\sum (h_\theta(x^{(i)}) - y^{(i)})\, x_1^{(i)} + \frac{\lambda}{m}\theta_1$ ← theta 1

gradient(3) = [code to compute $\frac{\partial}{\partial \theta_2} J(\theta)$];

% $\frac{1}{m}\sum (h_\theta(x^{(i)}) - y^{(i)})\, x_2^{(i)} + \frac{\lambda}{m}\theta_2$ ← theta 2

gradient(n+1) = [code for $\frac{\partial}{\partial \theta_n} J(\theta)$];

% $\frac{1}{m}\sum (h_\theta(x^{(i)}) - y^{(i)})\, x_n^{(i)} + \frac{\lambda}{m}\theta_n$

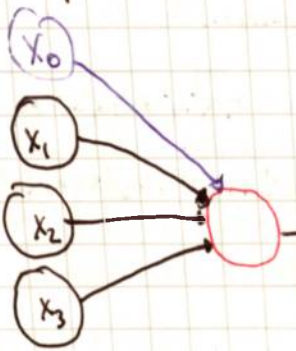
A vectorized implementation is:

$h = g(X\theta)$

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

Week 4 - Neural Networks

Model representation, Logistic unit, a neuron



- Neuron
- Inputs
- Bias

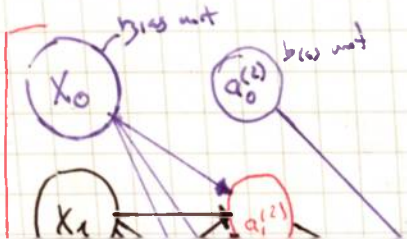
$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

Weights or parameters

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

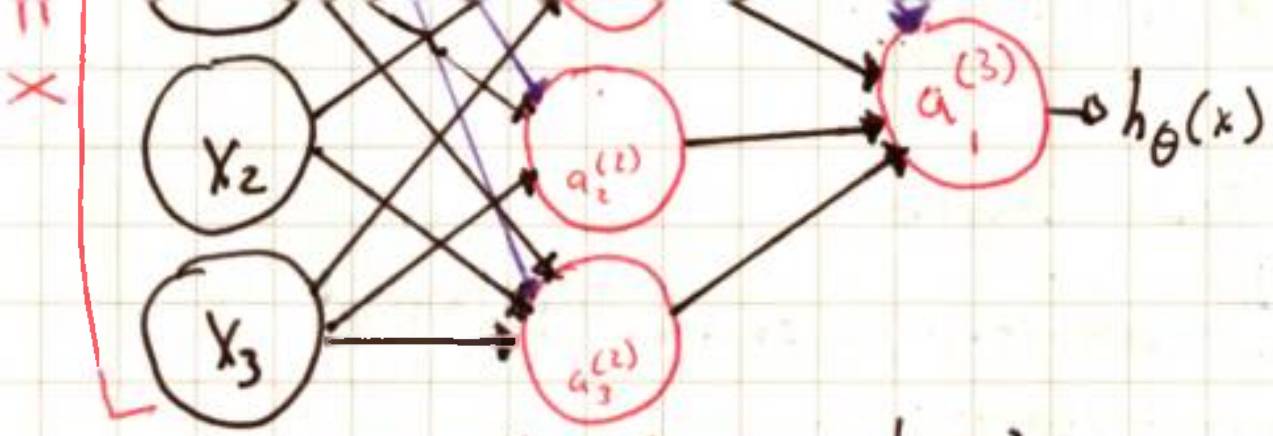
This is a neuron with a sigmoid or logistic activation f.

Neural Networks



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

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



Layer 1
3 input units

Layer 2
(Hidden)
3 hidden units

Layer 3
(output)

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

3×4 matrix

$$\Theta^{(1)} \in \mathbb{R}^{3 \times (3+1) = 3 \times 4}$$

dimensional matrix

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

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

$$s_{j+1} \times (s_j + 1)$$

j = layer

output

input

The size of the matrix is outputs × inputs, where "outputs" is the number of units in the next layer and "inputs" is the number of units in previous layer including the bias unit, without the bias unit

Model representation 2 : Vectorized implementation

Previously we said that the sequence of steps we ~~need to~~ that we need in order to compute the output of a hypothesis are the equations given before on the previous page; compute the activation values of the three hidden units, and use those to compute the final output of our hypothesis $h_\Theta(x)$. To simplify the notation we'll introduce new terms:

$z_1^{(2)} = \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3$ ∴ $a_1^{(2)} = g(z_1^{(2)})$

$z_2^{(2)} = \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3$

$z_3^{(2)} = \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3$ ∴ $a_3^{(2)} = g(z_3^{(2)})$

The dotted area resembles a matrix vector operation, of $\Theta^{(1)} x$, so we can write a vectorized computation:

$x = a^{(1)} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$ $z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$ ~ 3d vector (3×1)

$z^{(2)} = \Theta^{(1)} a^{(1)}$; $a^{(2)} = g(z^{(2)})$ ~ $z^{(2)}$ ∈ $\mathbb{R}^3$

And to account for the $b_{(i)}$; $a_0^{(2)} = 1$, so $a^{(2)}$ (second layer) becomes $a^{(2)} \in \mathbb{R}^4$ ($n+1$), and to compute the final value, on the third layer, of our hyp.

$z^{(3)} = \Theta^{(2)} a^{(2)}$ ∴

$h_\Theta(x) = a^{(3)} = g(z^{(3)})$

Alejandro: note that the superscripts are being assigned forward. Because of this, it's called "Forward propagation"

Repeating:

$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$ $z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$ → $z^{(2)} = \Theta^{(1)} a^{(1)}$; $a^{(2)} = g(z^{(2)})$ → $z^{(3)} = \Theta^{(2)} a^{(2)}$; $a^{(3)} = g(z^{(3)})$ → $h_\Theta(x)$

The neural network looks as logistic regression, first layer (input) takes x to perform calculations, but subsequent layers uses already computed or "learned" parameters from the previous layer, as input to the second one.

Examples and Intuitions (1)

Predicting the **AND** function: 0 0 = 0, 0 1 = 0, 1 0 = 0, 1 1 = 1, the graph of our functions will look like $\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow g(z^{(2)}) \Rightarrow h_\theta(x)$

x_0 is our bias unit, always = 1

Assuming weights of -30 for x_0, 20 for x_1 and 20 for x_2, we have:

$\Theta^{(1)} = [-30 \quad 20 \quad 20]$

This will cause the output of our hypothesis to be 1 only if both x_1 and x_2 are 1:

$h_\theta(x) = g(-30x_0 + 20x_1 + 20x_2)$

$h_\theta(x) = g(-30 + 20x_1 + 20x_2)$ ∴

$x_1 = 0$	$x_2 = 0$	then	$g(-30) \approx 0$
$x_1 = 0$	$x_2 = 1$	then	$g(-10) \approx 0$
$x_1 = 1$	$x_2 = 0$	then	$g(-10) \approx 0$
$x_1 = 1$	$x_2 = 1$	then	$g(10) \approx 1$

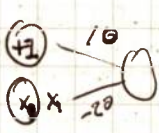
OR function $\Theta^{(1)} = [-10 \quad 20 \quad 20] \Rightarrow h_\theta(x) = g(-10x_0 + 20x_1 + 20x_2)$

OR

$-10(x_0)$	$(20)x_1$	$(20)x_2$	∴		
-10	20(0)	20(0)	∴	≈	$g(-10) \approx 0$
-10	20(0)	20(1)	∴	≈	$g(10) \approx 1$
-10	20(1)	20(0)	∴	≈	$g(10) \approx 1$
-10	20(1)	20(1)	∴	≈	$g(30) \approx 1$

Intuition 2

NOT

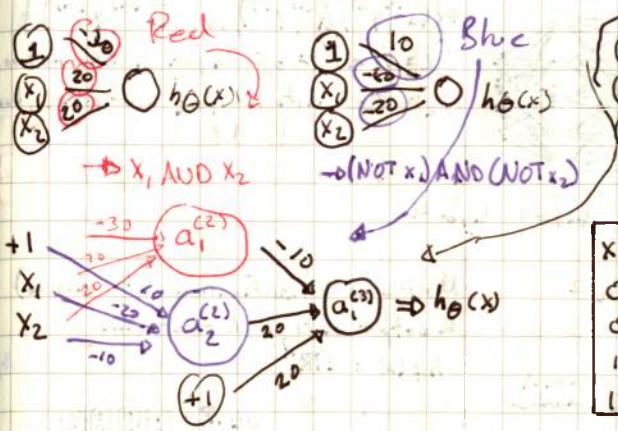


x_1	$h_\theta(x)$	
0	$g(10-0)$	≈ 1
1	$g(10-20)$	≈ 0

$g(10)$ →

$h_\theta(x) = g(10 - 20x_1)$ ∴ ↗

Putting them altogether: X_1 XNOR X_2



x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\Theta(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	1	1

Multiclass Classification

To classify data into multiple classes, we let our hypothesis function to return a vector of values, say if we want to distinguish pictures from a pedestrian, a car, a motorcycle and a truck, what we would have is for example:

$h_\Theta(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ pedestrian $\quad h_\Theta(x) \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$ car $\quad h_\Theta(x) \approx \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$ motorc. $\quad h_\Theta(x) \approx \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$ truck

We can define our set of resulting classes as $y =$

$$y^{(i)} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

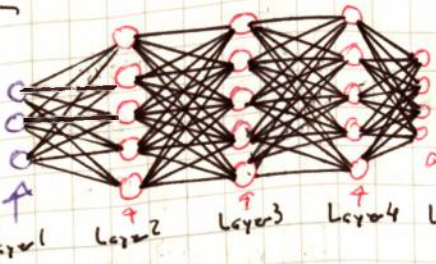
Each $y^{(i)}$ represents a different image corresponding to either a car, pedestrian, truck or motorcycle. The inner layers, each provide us with some new information which leads to our final hypothesis function. The setup looks like:

$$\begin{bmatrix}x_0\\x_1\\x_2\\\cdots\\x_n\end{bmatrix} \rightarrow \begin{bmatrix}a_0^{(2)}\\a_1^{(2)}\\a_2^{(2)}\\\cdots\end{bmatrix} \rightarrow \begin{bmatrix}a_0^{(3)}\\a_1^{(3)}\\a_1^{(3)}\\\cdots\end{bmatrix} \rightarrow \begin{bmatrix}a_0^{(4)}\\a_0^{(4)}\\a_2^{(4)}\\\cdots\end{bmatrix} \rightarrow \cdots \rightarrow \begin{bmatrix}h_\Theta(x)_1\\h_\Theta(x)_2\\h_\Theta(x)_3\\h_\Theta(x)_4\end{bmatrix}$$

Our resulting hypothesis for one set of inputs may look like

$$h_\Theta(x) = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$$ which in our example would represent a motorcycle

[Week 5 Neural Networks: Learning]



$\rightarrow \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

L = total # of layers in the network

s_l = # of units in layer l (not counting bias)

$\therefore L = 4$

$s_1 = 3, s_2 = 5, s_3 = 5, s_4 = 4 = s_L = 4$

s_L = the final layer

Binary Classification:

labels, $y = 0$ or 1

one output unit

$h_\Theta(x) \in \mathbb{R}$ A real number

$K = 1$

$s_L = 1$

one output but with two possible values, {1, 0}

K = the number of units in the output layer

$K \geq 1$

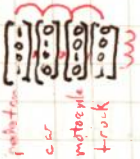
Multiclass classification (K classes)

K output units, K distinct classes

$h_\Theta(x) \in \mathbb{R}^K$ e.g. K dimensional output

$s_L = K$

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



$K \geq 3$ because for 2 classes we would use a binary classification with one output unit

Cost Function:

The cost function for a neural network will be a generalization of the one used for logistic regression used to minimize the function of $J(\Theta)$, with it's regularization function (still without regularizing the bias).

Now instead of having one basically one logistic regression output unit, we may instead have K of them.

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

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

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

Sum of the K output units on the final layer

which of the K output we're looking at

$h_\Theta(x)_k$

y_K

not from 0 or 1

Looks complicated but what it's doing is summing over terms $\Theta_{ji}^{(l)}$ for all values of i, j, l, except that we don't sum over the terms corresponding to this bias values like we had for logistic regression. Concretely, we don't sum over the terms corresponding to where $i = 0$, that is the bias unit, and we should not.

Notes:

- Double sum simply adds the logistic regression costs calculated for each cell in L_2
- The triple sum simply adds up the squares of all the individual Θs in the entire network.
- The i in the triple sum does NOT refer to training example i.

because it would yield into regularizing the $\Theta_{i0}^{(l)} x_0$ term

x_0 $\Theta_{i0}^{(l)} x_0 + \Theta_{i1}^{(l)} + \cdots$

Back propagation: algorithm to try to minimize $J(\Theta)$

We need to have code to compute:

$J(\Theta)$ → Using the formula on the previous page

$\frac{d}{d\Theta_{ij}^{(l)}} J(\Theta)$ → the partial derivative

Gradient computation

To simplify, imagine we have only one training example (x, y), no subscripts, so (x, y), only one pair

Forward Propagation

$a^{(1)} = x$

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

$a^{(2)} = g(z^{(2)})$ (add $a_0^{(2)}$)

$z^{(3)} = \Theta^{(2)} a^{(2)}$

$a^{(3)} = g(z^{(3)})$ (add $a_0^{(3)}$)

$z^{(4)} = \Theta^{(3)} a^{(3)}$

$a^{(4)} = g(z^{(4)}) = h_\Theta(x)$

Back Propagation

$a_j^{(l)}$ = activation of unit j in layer l

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

On the above and last page sample, $L = 4$

$\Delta_j^{(4)} = a_j^{(4)} - y_j$; activation of unit j in output layer $(h_\Theta(x))_j$

$\delta^4 = a^{(4)} - y$ (vectors)

$\Delta^{(3)} = (\Theta^{(3)})^T \Delta^{(4)} .* g'(z^{(3)})$

$\Delta^{(2)} = (\Theta^{(2)})^T \Delta^{(3)} .* g'(z^{(2)})$

$g'(z^{(3)}) = a^{(3)} .* (1 - a^{(3)})$

$g'(z^{(2)}) = a^{(2)} .* (1 - a^{(2)})$

- No $\Delta^{(1)}$, because is the input layer

Back propagation is a neural network terminology for minimizing our cost function, just as we were doing with gradient descent in logistic and linear regression. Our goal is to compute $\min_\Theta J(\Theta)$

That is, we want to minimize our cost function J using an optimal set of parameters in theta. In this section we'll look at the equation we use to compute the partial derivative of $J(\Theta)$: $\frac{d}{d\Theta_{ij}^{(l)}} J(\Theta)$

Given the training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

- Set $\Delta_{ij}^{(l)} := 0$ for all i, j, l; hence has as a matrix full of zeroes.

For training example $t = 1$ to m: $(x^{(i)}, y^{(i)})$ training example

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

2 Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$ (all layers) — $h_\Theta(x)$, output

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

4 Compute $\delta^{(L-1)}, \delta^{(L-2)}, \delta^{(L-3)} \ldots$ Descend down to $\delta^{(2)}$

5 $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ — vectorized: $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

End for

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

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

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

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

Where L is our total number of layers and $a^{(L)}$ is the vector of outputs of the activations units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct results in y. To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left: (4)

(4) Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) .* a^{(l)} .* (1 - a^{(l)})$

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

The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l.

Then we multiply 'element wise' that with a function called g', or g prime, which is the derivative of the activation function g evaluated with the input values given by $z^{(l)}$. The g-prime derivative terms, can also be written out as:

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

(5) $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$, or with vectorization: $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

Hence we update our new matrix Δ:

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

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

the capital Delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

Implementation note: unrolling parameters

With neural networks, we are working with sets of matrices:

$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}, \ldots$

$D^{(1)}, D^{(2)}, D^{(3)}, \ldots$

Optimization routines tend to need vectors as an input, that's why we will want to 'unroll' all the elements and put them into a long vector:

```
thetaVector = [theta1(:); theta2(:); theta3(:)]
deltaVector = [D1(:); D2(:); D3(:)]
```

We use (:)

- If the dimension of theta1 is 10x11, theta2 is 10x11, and theta3 is 1x11, we will get a vector with length of 231 elements
- To get back a set of matrices from a vector, we use the reshape command, so we get the unrolled version of it:

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

reshape(vector(from:to), new rows, new cols)

Learning algorithm:

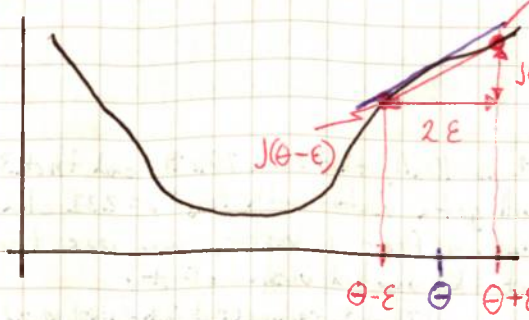
- Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
- Unroll to get initialTheta to pass to
  fminunc(@costFunction, initialTheta, options)

Function [jval, gradientVec] = costFunction(thetaVec)
- From thetaVec, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$
- Use forward propagation and backward propagation to get $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$
- Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get gradientVec.

Gradient checking → "Debug forward and back prop"

While implementing the algorithm, it might look that the cost function indeed gets reduced, but with fairly complex algorithms, it might happen that by a small bug in the implementation, we end with a neural network that has a big error rate, and to avoid that, it's recommended to test it, like with the following procedure.

Numerical estimation of gradients:



Note: $\theta \in \mathbb{R}$ (θ is a real number)

$$\frac{d}{d\theta} J(\theta) \approx \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$$

$\varepsilon = 10^{-4}$ (suggested value)

Implemat: gradApprox = (J(theta+EPSILON) − J(theta−EPSILON))/2*EPSILON (derivative)

A way to estimate the gradient, is to choose a value of theta and place two theta +/− EPSILON. So we can numerically approximate.
Then we connect those points with a straight line J(θ+ε) and J(θ−ε), also then draw a line perpendicular line down from J(θ+ε) and horizontal line from J(θ−ε), forming an imaginary triangle; then, similar to the calculation of the slope of the line, we do so, with $\frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$.
Usually a small value works fine like $\varepsilon = 10^{-4}$, but choosing smaller values can cause numerical problems.

Quiz: Let $J(\theta) = \theta^3$. Furthermore, let $\theta = 1$ and $\varepsilon = 0.01$, what's the approximate derivative

$\theta = 1$

$J(\theta) = \theta^3$

$$\frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon} = \frac{(1+0.01)^3 - (1-0.01)^3}{2 \times 0.01} = \frac{(1.01)^3 - (0.99)^3}{0.02}$$

$$= \frac{1.0303 - 0.9703}{0.02} = \frac{0.0600}{0.02} = 3.001$$

Now, on the previous page we did consider $\theta \in \mathbb{R}$, now we will look a more general case where θ is $\mathbb{R}^n$ (vector) (θ is an 'unrolled' version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$, the parameters of our NN, so θ is a vector with n elements, from $[\theta_1 \dots \theta_n] = \theta$; we can then use a similar idea to compute the approximate partial derivatives:

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

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

$$\vdots$$

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

In code:

```
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;
```

$\begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_i + \varepsilon \\ \theta_n \end{bmatrix}$ $\theta - \varepsilon$

And then, check that gradApprox $\approx$ Dvec ; should be pretty close.
(Dvec: derivatives from back prop. (vector))

Implementation note:
- Implement back prop to compute Dvec (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
- Implement numerical gradient check to compute gradApprox
- Make sure they give similar values
- Turn off gradient checking and use back propagation for learning.

Important: Be sure to disable gradient checking code before training the classifier. If numerical gradient computation is run on every iteration of gradient descent (or in inner loop of costFunction), the code will be very slow.

[Random Initialization]

Initializing all theta weights to zero does not work with neural networks. When we back propagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our Θ matrices using random values.

The random values should be symmetrically distributed, meaning that the range of possible values have the same 'distance'; to achieve that we expect to have initialized:

$\Theta_{ij}^{(l)}$ to a random value in $[-\varepsilon, \varepsilon]$
($i, e.i.$ $-\varepsilon \leq \Theta_{ij}^{(l)} \leq \varepsilon$)

Note that ε here doesn't relate to the previous value used on numerical reduction for gradient checking.

In code, for example:

```
theta1 = rand(10, 11) * (2*EPSILON) - EPSILON
theta2 = rand(1, 11) * (2*EPSILON) - EPSILON
```

Putting it together: Pick the network architecture

- Number of input units: # of dimensions of features $x^{(i)}$
- Number of output units: # of classes
- Number of hidden units per layer: Usually the more the better, must balance with computation costs.
- Defaults: 1 hidden layer; if we have more than 1, then it's recommended to have the same number of units in every hidden layer.

Training a NN:

1: Randomly initialize the weights
2: Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3: Implement the cost function
4: Implement back propagation to compute partial derivatives
5: Use gradient checking to confirm that your backprop works, then disable it
6: Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

For $i=1:m$
Perform forward propagation and back propagation using example $(x^{(i)}, y^{(i)})$
Get activations $a^{(l)}$ and delta terms $d^{(l)}$ for $l=2, 3, \ldots, L$

[Evaluating the hypothesis]

As usual we should choose the parameters that minimize the errors. It can happen that we overfit.

Learn parameter Θ from training data (minimizing training error $J(\Theta)$) on 70% of the samples

Once we've done some troubleshooting for errors in our predictions, we can try to:

- get more training examples
- try additional features
- increase or decrease λ
- try a smaller set of features
- try polynomial features

Ideally don't chose them randomly, we'll explore diagnostic techniques that best of above's solution following on.

Evaluating an hypothesis

An hypothesis may have a low error for the training examples but still be inaccurate (because of overfitting). Thus to evaluate a hypothesis, given a dataset of training examples, we can split up the data into two sets: training set and test set. Typically it's chosen at a ratio 70-30%.

The new procedure can't now on:

1: Learning Θ and minimize $J_{train}(\Theta)$ using the training set

2: Compute the test error $J_{Test}(\Theta)$

The test set error

1 For linear regression: $J_{test}(\Theta) = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \left(h_\Theta(x_{test}^{(i)}) - y_{test}^{(i)}\right)^2$

2 For classification ~ misclassification error (0/1 misclassification error)

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

this gives us a binary 0 or 1 result based on a misclassification.

The average test error for the test set is:

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

this gives us the proportion of the test data that was misclassified.

[Diagnosing Bias vs Variance]

cv = cross validation

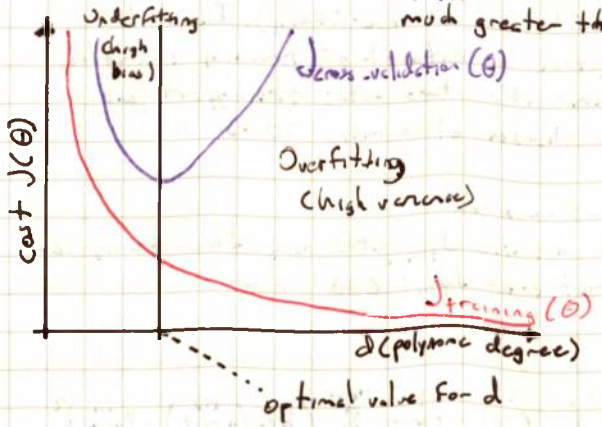
60-20-20 %.

In this section we examine the relationship between the degree of the polynomial 'd' and the underfitting or overfitting of our hypothesis.

- We need to distinguish whether bias or variance is the problem contributing to bad decisions.
- High bias is underfitting and high variance is overfitting. Ideally we need to find a 'golden mean' between these two.

The train error will tend to decrease as we increase 'd' (the polynomial degree) up. At the same time, the cross validation error will tend to decrease as we increase d up to a point, and then it will increase as d is increased, forming a convex curve.

- High Bias (underfitting): both $J_{train}(\theta)$ and $J_{cv}(\theta)$ will be high. Also $J_{cv}(\theta) \approx J_{train}(\theta)$
- High Variance (overfitting): $J_{train}(\theta)$ will be low and $J_{cv}(\theta)$ will be much greater than $J_{train}(\theta)$



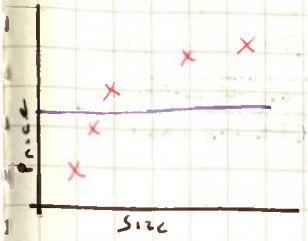
[Regularization and Bias / Variance]

Linear regression with regularization

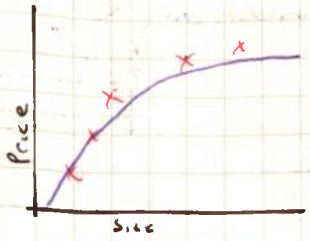
For the cost function $J(\theta)$ we add the regularization term to adapt/soften/or indeed regularize the cost. Now with $J_{cv}(\theta)$ or $J_{test}(\theta)$, we still use it but only for $J(\theta)$; the regularization is not used on $J_{cv}(\theta)$ neither on $J_{test}(\theta)$.

Model: $h_\theta(x) = \theta_0 + \theta_1 x^1 + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

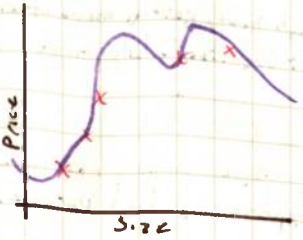
$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{n}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \boxed{\frac{\lambda}{2m}\sum_{j=1}^{m}\theta^2}$$



Large λ
High bias (underfit)
$\lambda > 10000$. $\theta_1 \approx 0, \theta_2 \approx 0 \ldots$
$h_\theta(x) \approx \theta_0$



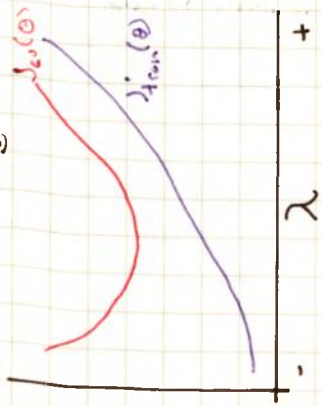
Intermediate λ
"Just right"



Small λ
High Variance (overfit)
$\lambda \to 0$, $\lambda = 0$

In the figures above, we see that as λ increases, our fit becomes more rigid. On the other hand, as λ approaches to 0, we tend to overfit the data. So, how do we choose our parameter λ to get it "just right"? In order to choose the model and λ:

1. Create a list of λ's, i.e. $\lambda \in \{0, 0.01, 0.02, 0.04, 0.08 \ldots\}$
2. Create a set of models with different degrees or any other variants
3. Iterate through λs and for each λ, go thru all models to learn some θ
4. Compute CV error using the learned θ (computed with λ) on the $J_{cv}(\theta)$ without regularization or $\lambda = 0$
5. Select the best combo that produces the lowest error on the CV set
6. Using the best combo θ and λ, apply it on $J_{test}(\theta)$ to see if it has a good generalization of the problem



↳

Learning Curves

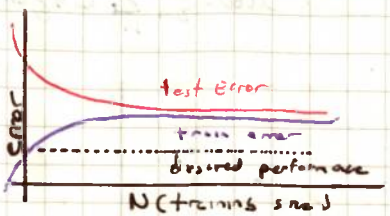
Training an algorithm on a very few number of datapoints (such as 1, 2 or 3) will easily have 0 errors because we can always find a quadratic curve that touches exactly those points. Therefore:

- As the training set gets larger, the error for a quadratic function increases
- The error will 'plateau' out after a certain m, or training set size.

Experiencing high bias:

- **Low training set size:** causes $J_{train}(\theta)$ to be low and $J_{cv}(\theta)$ to be high.
- **Large training set size:** causes both $J_{train}(\theta)$ and $J_{cv}(\theta)$ to be high, with $J_{train}(\theta) \approx J_{cv}(\theta)$

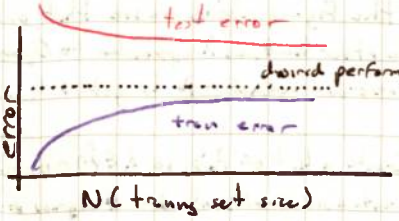
If a learning algorithm is suffering from high bias, getting more training data will not (by itself) help much.



Experiencing high variance:

- **Low training set size:** $J_{train}(\theta)$ will be low and $J_{cv}(\theta)$ will be high
- **Large training size:** $J_{train}(\theta)$ increases with training set size and $J_{cv}(\theta)$ continues to decrease without leveling off.
  Also, $J_{train}(\theta) < J_{cv}(\theta)$, but the difference between them remains significant.

If a learning algorithm is suffering from high variance, getting more training data is likely to help.



Deciding what to do next (revisited)

Our decision process can be broken down as follows:

- Getting more training examples: Fixes high variance
- Trying smaller set of features: Fixes high variance
- Adding features: Fixes high bias
- Adding polynomial features: Fixes high bias
- Decreasing λ: Fixes high bias
- Increasing λ: Fixes high variance

Diagnosing Neural Networks:

- A neural network with fewer parameters is prone to underfitting. It's also computationally cheaper.
- A large neural network with more parameters is prone to overfitting. It's also computationally expensive. In this case, using regularization (increase λ) will address the overfitting.

Using a single layer ($L=3$) is a good starting default. You can train your neural network on a number of hidden layers using your cross validation set. Then you select the one that performs the best.

Model Complexity Effects:

- Lower order polynomials (low model complexity) have high bias and low variance. In this case the model fits poorly consistently.
- Higher-order polynomials (high model complexity) fit the training data extremely well and the test data extremely poorly. These have low bias on the training data, but very high variance.
- In reality, we would want to choose a model in between, that generalizes well, but also fits the data reasonably well.

[Error metrics for skewed classes]

When we have samples which consist mainly on one specific classification (1 or 0), and our logistic regression model is highly oriented into predicting ~~that~~ that class type. Sort of influenced.

For example in the cancer detection problem, $h_\theta(x)$. ($y=1$ if cancer, $y=0$ otherwise) we find that we've got 1% error on the test set, but only 0.5 of patients have cancer.

When we get this skewed distributions, we need a different error metric or error evaluation metric called Precision / Recall.

Predicted Class \ Actual Class	1	0
1	true positive	false positive
0	false Negative	true negative

P Precision: Of all patients that were predicted $y=1$, what fraction actually had cancer.

$$\frac{\text{true positives}}{\#\text{predicted +}} = \frac{\text{true positives}}{\#\text{true pos + false pos}}$$

R Recall: Of all patients that actually have cancer, what fraction of them did we correctly detect as having?

$$\frac{\text{true positives}}{\#\text{actual positives}} = \frac{\text{true positives}}{\text{true pos + false neg}}$$

Tradeoff

For some application we want to control the tradeoff between precision and recall, to do it, we increment the threshold (or reduce it), as we want to provide confident information, and we do this in our sigmoid or logistic function, by predicting 1 if $h_\theta(x) \geq 0.7$ and 0 if $h_\theta(x) < 0.7$. This will have higher precision but lower recall. Likewise if we would like to be more 'flexible' into classifying more samples as positive, then the threshold should be lowered. This depends on the purpose, if need to correct Flag to avoid problems. This will have less precision and higher recall.

[F1 score]: How to compare precision and recall numbers?

	Precision (P)	Recall (R)	~~Average~~	F1 Score	
Algorithm 1	0.5	0.4	0.45	0.444	the higher
Algorithm 2	0.7	0.1	0.4	0.175	the better on
Algorithm 3	0.02	1.0	0.51	0.0392	the Cross Validation set

~~Average = $\frac{P+R}{2}$~~

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

$P=0$ or $R=0$ $\Rightarrow$ F-score $= 0$

$P=1$ and $R=1$ $\Rightarrow$ F-score $= 1$ perfect F score

Support Vector Machine Optimization Objective

Week 7

The Support Vector Machine (SVM) is yet another type of supervised machine learning algorithm. It is sometimes cleaner and more powerful.

Recalling that in logistic regression, we used the following rules:

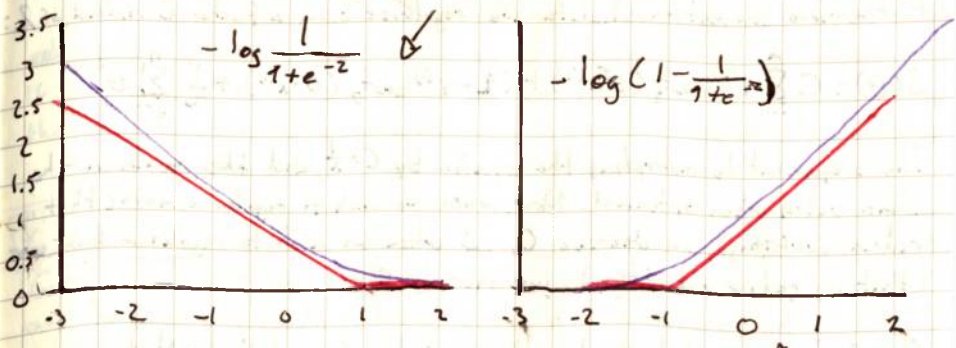
if $y=1$, then $h_\theta(x) \approx 1$ and $\theta^T x >> 0$
if $y=0$, then $h_\theta(x) \approx 0$ and $\theta^T x << 0$

And the cost function for unregularized logistic regression

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

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

To make a support vector machine, we will modify the first term of the cost function: $-\log(h_\theta(x)) = -\log\left(\frac{1}{1+e^{-\theta^T x}}\right)$, so that when $\theta^T x$ (from now on, we shall refer to this as z) is greater than 1... it outputs 0. Furthermore, for values of z less than one, we shall use a straight decreasing line instead of the sigmoid curve. In the literature this is called a "~~Hanga~~ Hinge loss function"



Similarly, we modify the second term of the cost function $-\log(1-h_\theta(x))$ $= -\log\left(1-\frac{1}{1+e^{-\theta^T x}}\right)$, so that when z is less than ~~0~~ -1, it outputs 0. We also modify it so that for values of z greater than -1, we use a straight increasing line instead of the sigmoid curve.

We shall denote these as $cost_1(z)$ and $cost_0(z)$ (respectively, note that $cost_1(z)$ is the cost for classifying when $y=1$ and $cost_0(z)$ is the cost for classifying when $y=0$), and we define them as follows (where k is an arbitrary constant defining the magnitude of the slope of the line):

$$z = \theta^T x \qquad cost_0(z) = \max(0, k(1+z))$$
$$cost_1(z) = \max(0, k(1-z))$$

Recalling the cost function from (unregularized) logistic regression:

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

Note that the negative sign has been distributed into the sum of the above equation. We may transform this into the cost function for SVM by substituting $cost_0(z)$ and $cost_1(z)$:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}\, cost_1(\theta^T x^{(i)}) + (1-y^{(i)})\, cost_0(\theta^T x^{(i)}) + \frac{1}{2n}\sum_{j=1}^{n}\theta_j^2$$

This can be optimized a bit by multiplying all by "m" (thus removing it from the denominators). Note that this does not affect our optimization since it's multiplied by a positive constant:

$$J(\theta) = \sum_{i=1}^{m} y^{(i)}\, cost_1(\theta^T x^{(i)}) + (1-y^{(i)})\, cost_0(\theta^T x^{(i)}) + \frac{\lambda}{2}\sum_{j=1}^{n}\theta_j^2$$

Furthermore convention dictates we regularize using a factor C instead of λ:

$$J(\theta) = C\sum_{i=1}^{m} y^{(i)}\, cost_1(\theta^T x^{(i)}) + (1-y^{(i)})\, cost_0(\theta^T x^{(i)}) + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$

This is equivalent to multiplying the equation by $C=\frac{1}{\lambda}$, and thus results in the same values when optimized. Now, when we wish to regularize more (that is, reduce overfitting), we decrease C, and when we wish to regularize less (that is, reduce underfitting), we increase C.

One final note is that the hypothesis of the Support Vector Machine is not interpreted as the probability of y being 1 or 0 (as is the hypothesis of logistic regression). Instead, it outputs either 1 or 0 straight (it's a discriminant function):

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

[Large Margin Intuition]

A useful way to think about Support Vector Machines, is to think about them as Large Margin Classifiers..

If $y=1$, we want $\theta^T x \geq 1$ (not just ≥ 0)
If $y=0$, we want $\theta^T x \leq -1$ (not just < 0)

Now when we set our constant C to a very large value (eg 100,000), our optimizing function will constrain Θ such as the first part of the equation equals zero. We impose the following constraints on Θ:

$\theta^T x \geq 1$ if $y=1$ and $\theta^T x \leq -1$ if $y=0$

If C is very large, we must choose Θ parameters such that:

$\sum_{i=1}^{m} y^{(i)} \text{cost}_1(\theta^T x) + (1-y)\text{cost}_0(\theta^T x) = 0$

And this reduces our cost function to:

$$J(\theta) = C \cdot 0 + \frac{1}{2}\sum_{j=1}^{n} \theta_j^2 = \frac{1}{2}\sum_{j=1}^{n} \theta_j^2$$

Recalling the decision boundary from logistic regression (the line separating positive and negative examples). In SVM, the decision boundary has the special property that is as far away as possible, from both the positive and negative values.

The distance of the decision boundary to the nearest example is called the margin and this, a large margin is only achieved when C is very large.. Data is linearly separable when a straight line can separate the positive and negative examples. If we have outlier examples that we don't want to affect the decision boundary, then we can reduce C. Increasing and decreasing C is similar to respectively decreasing and increasing λ, and can simplify our decision boundary.

[Kernels 1]

Kernels allow us to make complex, non linear classifiers using SVMs. Given 'x', compute new feature depending on proximity to landmarks $l^{(1)}, l^{(2)}, l^{(3)}$ and to do so, we should find the "similarity" of 'x' and some landmark $l^{(i)}$.

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

, the similarity func. is called a "gaussian Kernel". It is a specific example of a kernel. The function can also be written as follows;)

$$f_i = \text{similarity}(x, l^{(i)}) = \exp\left(-\frac{\sum_{j=1}^{n}(x_j - l_j^{(i)})^2}{2\sigma^2}\right)$$

There are a couple properties of the similarity function:

if $x \approx l^{(i)}$, then $f_i = \exp\left(-\frac{\approx 0^2}{2\sigma^2}\right) \approx 1$ and if x is far from l then :)

$f_i = \exp\left(-\frac{(\text{large number})^2}{2\sigma^2}\right) \approx 0$

In other words, if x and the landmark are close, the similarity will be close to 1, but and if x and the landmark are far away from eachother, the similarity will be close to 0.

Each landmark gives us the features in our hypothesis: $l^{(1)} = f_1$, $l^{(2)} = f_2$, $l^{(3)} = f_3$ | $h_\Theta(x) = \Theta_1 f_1 + \Theta_2 f_2 + \Theta_3 f_3 + \cdots$

σ^2 is a parameter of the Gaussian Kernel, and it can be modified to increase or decrease the dropoff for our feature f_i. Combined with looking at the values inside Θ, we can choose these landmarks to get the general shape of the decision boundary.

[Kernels II] One way to get the landmarks is to put them in the exact same location as all the training examples. This gives us "m" landmarks, with one landmark per training example.

Given example x: $f_1 = \text{similarity}(x, l^{(1)})$, $f_2 = \text{similarity}(x, l^{(2)})$, $f_3 = \text{similarity}(x, l^{(3)})$, and so on.

This gives us a 'feature vector', $f^{(i)}$ of all our features for the example $x^{(i)}$. We may also set $f_0 = 1$ to correspond with Θ_0. Thus given training example $x^{(i)}$:

$$x^{(i)} = \begin{bmatrix} f_1^{(i)} = \text{similarity}(x^{(i)}, l^{(1)}) \\ f_2^{(i)} = \text{similarity}(x^{(i)}, l^{(2)}) \\ \vdots \\ f_m^{(i)} = \text{similarity}(x^{(i)}, l^{(m)}) \end{bmatrix}$$

Now to get the parameters Θ we can use the SVM minimization algorithm but with $f^{(i)}$ substituted in for $x^{(i)}$:

$$\min_\Theta C \sum_{i=1}^{m} y^{(i)} \text{cost}_1(\Theta^T f^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\Theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2$$

Using Kernels to generate $f^{(i)}$ is not exclusive for SVMs and may also be applied to logistic regression. However, because of computational optimizations on SVMs, kernels combined with SVMs is much faster than with other algorithms, so Kernels are almost always found combined only with SVMs.

[Choosing SVM parameters]

Choosing C ($C = \frac{1}{\lambda}$)

- If C is large, then we get higher variance and lower bias
- If C is small, then we get lower variance and higher bias

Choosing σ^2

- Large σ^2: Features f_i vary more smoothly, causing higher bias and lower variance
- Small σ^2: Features vary less smoothly, causing lower bias and higher variance.

Using a SVM

There are lots of good SVM libraries out there, such as 'liblinear' or 'libsvm'. And in real life is much better to use them rather than reinventing the wheel. In real life there are choices to be made, such as:

- Choice of parameter C
- Choice of kernel (the similarity function)
  - No kernel ("linear" kernel) gives standard linear classif.
  - Choose when n is large and m is small
  - Gaussian kernel - need to choose σ^2 - choose when n is small and m is large

* the library may ask to provide the kernel function.

Notes:
- DO perform feature scaling before using Gaussian kernel
- Not all similarity functions are valid kernels, they must satisfy the 'Mercer's' theorem. (which worries SVM package optimizations run efficiently and won't diverge.

Multi Class Classification:

Many SVM's have multiclass classification built in. It is possible to use as well one-vs-all method just as we did for logistic regression, where $y \in 1,2,3,..k$ with $\Theta^{(1)}, \Theta^{(2)}, \ldots, \Theta^{(k)}$. We pick the class i with the largest $(\Theta^{(i)})^T x$.

Logistic Regression vs. SVMs

- If n is large (relative to m), then use logistic regression, or SVM without kernel (linear)
- If n is small and m is intermediate, then use SVM with Gaussian kernel.
- If n is small and m is large, then manually create/add more features, then use logistic regression or SVMs without kernel

In the first case, we don't have enough examples to need a complicated polynomial hypothesis

In the second example, we have enough examples, that we might need a complex non-linear hypothesis

In the third example, we want to increase our features so that logistic regression becomes applicable.

NOTE: A neural network is likely to work well for any of these situations but may be slower to train.

[Week 8 - Clustering]

[Unsupervised learning: Introduction]

Unsupervised learning is contrasted from supervised learning because it uses an unlabeled training set rather than a labeled one. In other words, we don't have the vector Y of expected results, we only have a dataset of features where we can find structure.

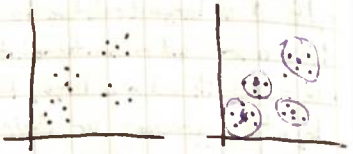
Clustering is good for:

- market segmentation
- social network analysis
- Organizing computer clusters
- Astronomical data analysis

[K-means algorithm]

The K-Means algorithm is the most popular and widely used algorithm for automatically grouping data into coherent subsets. It works as follows:

1: Randomly initialize 2 (K) points in the dataset, called the cluster centroids.
2: Cluster assignment: assign all examples into one of 2 (K) groups based on which cluster centroid the example is closest to.
3: Move centroid: Compute the averages of all the points inside each of the two(K) cluster centroid groups, then move the centroids to those averages.
4: Re run #2 and #3: Until we've found our clusters.



Our main variables are:

- K (number of clusters)
- Training set $x^{(1)}, x^{(2)}, x^{(3)}, \ldots, x^{(m)}$
- Where $x^{(i)} \in \mathbb{R}^n$ - n dimensional

} Note we won't use the $x_0 = 1$ convention.

The algorithm:

```
1  Randomly initialize K cluster centroids mu(1), mu(2), ..., mu(K)
2  Repeat:
3      for i = 1 to K:
4          c(i) := index (from 1 to K) of cluster centroid closest to x(i)
5      for K = 1 to K:
6          mu(K) := average (mean) of points assigned to cluster K
```

the first for-loop (lines 3 and 4) is the 'cluster assignment' step. We make a vector c where $c(i)$ represents the centroid assigned to example $x^{(i)}$. We can write the operation of cluster assignment more mathematically as follows: $c^{(i)} = \arg\min_k \|x^{(i)} - \mu_k\|^2$, that is each $c^{(i)}$ contains the index of the centroid that has minimal distance to $x^{(i)}$.

By convention, we square the right-hand side, which makes the function we are trying to minimize "more sharply increasing". It is mostly just a convention. But a convention that helps reduce the computational load because the Euclidean distance requires a square root but it's cancelled.

- Without the square: $\|x^{(i)} - \mu_k\| = \sqrt{(x_1^{(i)} - \mu_{1(k)})^2 + (x_2^{(i)} - \mu_{2(k)})^2 + (x_3^{(i)} - \mu_{3(k)})^2 + \dots}$
- With the square: $\|x^{(i)} - \mu_k\|^2 = (x_1^{(i)} - \mu_{1(k)})^2 + (x_2^{(i)} - \mu_{2(k)})^2 + (x_3^{(i)} - \mu_{3(k)})^2 + \dots$

so squaring convention serves two purposes: minimize sharply and less computation

The second for-loop is the "move centroid" step where we move each centroid to the average of it's group. More formally, the equation is as follows:

$$\mu_k = \frac{1}{n}\left[x^{(k_1)} + x^{(k_2)} + \dots + x^{(k_n)}\right] \in \mathbb{R}^n$$

Where each of $x^{(k_1)}, x^{(k_2)}, x^{(k_3)}, \dots, x^{(k_n)}$ are the training examples assigned to group μ_k. If we happen to have a cluster centroid with 0 points assigned to it, we can randomly re-initialize that centroid to a new point, or simply eliminate that cluster group.

After a number of iterations, the algorithm will converge where new iterations won't affect the clusters.

[Optimization Objective]

Recalling some of the parameters we've used in our algorithm:

- $c^{(i)}$ = index of cluster $(1, 2, \dots, k)$ to which example $x^{(i)}$ is assigned currently
- μ_k = cluster centroid k ($\mu_k \in \mathbb{R}^n$)
- $\mu_{c(i)}$ = cluster centroid of cluster to which example $x^{(i)}$ has been assigned

Using these variables, we can define our cost function:

$$J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k) = \frac{1}{m}\sum_{i=1}^{m} \|x^{(i)} - \mu_{c(i)}\|^2$$

Our optimization objective is to minimize all of our parameters using the cost function. That is, we are finding all values in sets c, representing all of our clusters, and μ, representing all of our centroids, that will minimize the average of the distances of every training example to it's corresponding centroid cluster centroid. This cost function is also called sometimes "the "distortion" of the training examples.

With K-means, it is not possible for the cost function to increase, it should always decrease.

[Random Initialization]

There is one particular recommended method for randomly initializing c

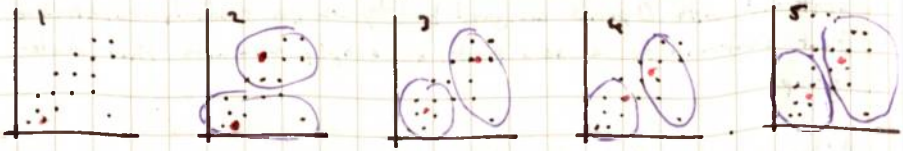
1- have $K < n$ ~ Make sure there are less clusters than training samples

2- Randomly pick K training examples ~ be sure they are unique.

3- Set $\mu_1, \ldots, \mu_K$ equal to these K examples.

K means can get stuck on local optima. To decrease the chance of this happening you can run the algorithm on many different random initializations. In cases where $K < 10$ it's strongly recommended to run a loop of random initialization

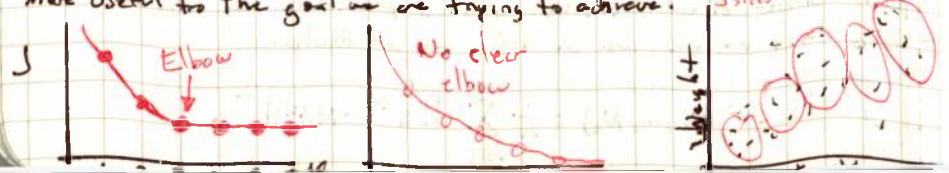
1. for i=1 to 100:
2. randomly initialize K-means
3. run K-means to get 'c' and 'm'
4. compute the cost function (distortion) $J(c,m)$
5. pick the clustering that gave us the lowest cost



[Choosing the Number of Clusters]

Choosing K can be quite arbitrary and ambiguous.

- The elbow method: Plot J and # of clusters K; J should reduce as K increases; However fairly often, the curve is very gradual, so there is no clear elbow, and also J will always decrease as K is increased. Only exception is if K gets stuck in local optimum
- The downstream purpose: Assign it arbitrarily, choose K that proves to be more useful to the goal we are trying to achieve.

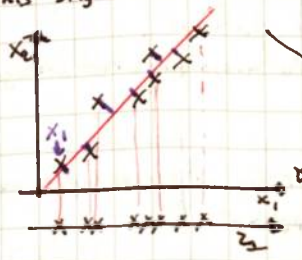


[Dimensionality Reduction]

There are a couple of reasons why we would like to do Dimensionality Reduction, whether is to do data compression, which will use less computer memory, but also speed up our learning algorithms.

[Motivation 1]: Data Compression:

- We may want to reduce the dimension of our features if we have redundant data.
- To do this we find highly correlated features, plot them, and make a new line that seems to describe both features accurately. We place new features on this single line.



Reducing 2D data into 1D

$x_i \in \mathbb{R}^2 \Rightarrow z^{(i)} \in \mathbb{R}^1$

Similarly, we can reduce from 3D to 2D or $\mathbb{R}^{1000}$ to $\mathbb{R}^{100}$

this will reduce the total data we have to store in computer memory and will speed up our learning algorithm.

Note: In dimensionality reduction, we are reducing the number of features rather than our number of examples. Our variable m will stay exactly the same, however the number of n will be reduced.

[Motivation 2]: Visualization

It is not easy to visualize data that is more than 3D, we can then reduce the dimensions of data to 3 or less in order to plot it.

One example is: Having hundreds of features related to a country's economic system, which may be combined into one feature that you can call "Economic Activity"

Principal Component Analysis - Problem Formulation

The most popular dimensionality reduction algorithm is Principal Component Analysis (PCA)

Problem Formulation: Given two features X_1 and X_2, we want to find a single line that effectively describes both features at once. We then map our old features onto this new line to get a new single feature.

The goal of PCA is to reduce the average of all the distances of every feature to the projection line, this is: the projection error.

Reduce from 2d to 1d: Find a direction (a vector $u^{(1)} \in \mathbb{R}^n$) onto which project the data so as to minimize the projection error.

A more general case:

- Reduce from n-dimensional to k-dimension: Find k vectors $u^{(1)}, u^{(2)}, u^{(3)}, u^{(k)}$ onto which to project the data so as to minimize the projection error. If we are converting from 3d into 2d, we will project our data onto two directions (a plane), so k will be 2.

[PCA is not linear regression]:

- In linear regression, we are minimizing the squared error from every point to our predictor line. These are vertical distances
- In PCA, we are minimizing the shortest distance, or shortest orthogonal distances, to our datapoints

More generally, in linear regression we are taking all our examples in X and applying the parameters in Θ to predict Y

In PCA, we are taking a number of features $x_1, x_2, \ldots, x_n$, and finding a closest common datatset among them. We aren't trying to predict any result and we aren't applying any theta weights to the features.

[Principal Component Analysis Algorithm]

Before we can apply PCA, there is a data-preprocessing we must perform

- Data Preprocessing:
  - Given a training set: $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$
  - Preprocess (feature scale / mean normalization):

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

  - Replace each $x_j^{(i)}$ with $x_j^{(i)} - \mu_j$
  - If there are different features on different scales (eg x_1 = size of the house and x_2 = number of bedrooms), scale features to have comparable range of values.

3 Compute Covariance Matrix

Above we first subtract the mean of each feature from the original feature and then we can scale all the features $x_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{s_j}$

We can define what it means specifically to reduce from 2D to 1D as:

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

Vectorized implementation = Sigma = 1/m * X' * X;

We denote the covariance matrix with a capital sigma (which happens to be the same symbol for summation, confusingly - they represent entirely different things).

Note that $x^{(i)}$ is a $n \times 1$ vector, $x^{(i)T}$ is a $1 \times n$ vector and X is a $m \times n$ matrix (row-wise stored examples). The product of these will be an $n * n$ matrix, which are the dimensions of Σ

(2) Compute 'eigenvectors' of covariance matrix Σ

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

"SVD" is the "singular value decomposition", a built in function.

What we actually want out of svd() is the "U" matrix of the Sigma covariance matrix: $U \in \mathbb{R}^{n \times n}$. U contains $u^{(1)}, \ldots, u^{(n)}$, which is what we want.

(3) Take the first K columns of the U matrix to compute z

We'll assign the first K columns of U to a variable called 'UReduce', this will be a $n * K$ matrix. We compute z with:

$$z^{(i)} = UReduce^T \cdot x^{(i)}$$

$UReduce^T \cdot x^{(i)}$ will have dimensions $K * n$ while $x^{(i)}$ will have dimensions $n \times 1$. The product $UReduce^T \cdot x^{(i)}$ will have dimension $K \times 1$

To summarize, the whole algorithm is:

```
Sigma = (1/m) * X' * X;
[U,S,V] = svd(Sigma);
Ureduce = U(:, 1:k);
z = x * Ureduce;
```

Reconstruction From Compressed Representation

If PCA can be used to compress, also can be used to get an approximation of the original data and some original amount of features. (eg $z \in \mathbb{R} \rightarrow z \in \mathbb{R}^2$)

This can be done with the equation $x_{approx}^{(1)} = Ureduce * z^{(1)}$

Turns out that the U matrix has the special property that is an UnitaryMatrix and one of that matrix properties is that $U^{-1} = U^*$ (* Conjugate transpose). Since we are dealing here worth real numbers, this is equivalent to: $U^{-1} = U^T$ so we could compute the inverse and use that, but it would be a waste of energy and cycles.

[Choosing the Number of Principal Components]

How do we choose k, also called the 'number of principal components'? Recall that k is the dimension we are reducing to.

One way to choose k is by using the following formula:

- Given the squared projection error $\frac{1}{m}\sum_{i=1}^{m} \|x^{(i)} - x_{approx}^{(i)}\|^2$
- Also given the total variation of data $\frac{1}{m}\sum_{i=1}^{m} \|x^{(i)}\|^2$
- Choose k to be the smallest value such that: $\frac{\frac{1}{m}\sum_{i=1}^{m} \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m}\sum_{i=1}^{m} \|x^{(i)}\|^2} \leq 0.01$

In other words: the squared projection error divided by the total variation should be less than one percent, so that 99% of the variance is retained.

Algorithm for choosing k

1. Try PCA with $k = 1, 2, \ldots$
2. Compute U_{reduce}, z, x
3. Check the formula complete, if not k + 1 and repeat

the prodedure is very inefficient, but we can use svd(Sigma) and use the matrix 'S'. We can check for 99% with the S matrix as:

$$\frac{\sum_{i=1}^{k} s_{ii}}{\sum_{i=1}^{n} s_{ii}} \geq 0.99$$

[Advice for Applying PCA]

- The most common use of PCA is to speed up supervised learning. Given a large training set with a large number of features (eg. $x^{(1)} \ldots x^{(m)} \in \mathbb{R}^{10000}$) we can use PCA to reduce the number of features in each example of the training set, so $z^{(1)}, \ldots, z^{(m)} \in \mathbb{R}^{10000} \rightarrow z^{(1)}, \ldots, z^{(m)} \in \mathbb{R}^{1000}$
- Note that we should define the PCA reduction from $x^{(i)}$ to $z^{(i)}$ only on training set, and not on Cross Validation or Test sets. We can then apply the mapping $z^{(i)}$ to the C.V. or Test sets, after it is defined on the training set.
- Applications:
  - Compression: Reduce space of data. Speed up algorithms
  - Visualization of data: Choose $k = 2$ or 3

Bad use of PCA: Trying to prevent overfitting. We may think that reducing the number of features with PCA would be a good way to address overfitting. It might work but using regularization is preffered because PCA doesn't take into account $Y^{(i)}$. Try first the full algorithm and then attempt PCA

Week 9: Anomaly Detection

As in any other machine learning problem, we are given a dataset $x^{(1)} \dots x^{(m)}$ and given an example x_{test} we want to know if this example is normal or anomalous.

We define a "model" $P(x)$ that tells us the probability that the example is not anomalous. We also use a threshold ε (epsilon) as a dividing line so we can say which examples are anomalous and which are not.

A very common application of anomaly detection is detecting a fraud:
- $x^{(i)}$ = features of user's i's activities
- model $P(x)$ from the data
- identify unusual users by checking which have $P(x) < \varepsilon$

If our anomaly detector is flagging too many anomalous examples, then we need to decrease our threshold ε.

Gaussian Distribution

The Gaussian Distribution is a familiar bell-shaped curve that can be described by a function $\mathcal{N}(\mu, \sigma^2)$.

Let $x \in \mathbb{R}$ if the probability distribution of x is Gaussian with mean μ, variance σ^2, then: $x \sim \mathcal{N}(\mu, \sigma^2)$ | the tilde $\sim$ means "distributed as"

The Gaussian Distribution is parameterized by a mean and a variance, μ or $\mathcal{N}$ describes the center of the curve, called the mean. The width of the curve is described by sigma or σ, called the standard deviation.

The full function is as follows: $P(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$

We can estimate μ from a given dataset by taking the average of samples

$\mu = \frac{1}{n}\sum_{i=1}^{n} x^{(i)}$ and for σ^2, with our familiar squared error formula: $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)^2$

Algorithm

Given a training set of examples, $\{x^{(1)}, \ldots, x^{(m)}\}$ where each example is a vector $x \in \mathbb{R}^n$:

$$p(x) = p(x_1; \mu_1, \sigma_1^2) \cdot p(x_2; \mu_2, \sigma^2) \cdot \ldots \cdot p(\mu_n; \mu_n, \sigma_n^2)$$

In statistics this is called "independence assumption" on the values of the features inside training example x. More compactly, the above expression can be written as follows:

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$$

Steps

- Choose features x_j that I think might be indicative of anomalous examples
- Fit parameters $\mu_1, \ldots \mu_n$; $\sigma_1^2 \ldots \sigma_n^2$ $\rightarrow \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \mu_n \end{bmatrix}$
- Calculate $\mu_j = (\sum_{i=1}^{m} x_j^{(i)})/m$
- Calculate $\sigma_j^2 = 1/n \cdot \sum_{i=1}^{m} (x_j^{(i)} - \mu_j)^2$
- Given a new example x, compute $p(x)$:

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$

- Anomaly if $p(x) < \varepsilon$

Developing and Evaluating an Anomaly Detection System

To evaluate our learning algorithm, we take some labeled data, categorized into anomalous and non-anomalous examples ($y = 0$ if normal, $y = 1$ if anomalous)

Among that data, take a large proportion of GOOD, non-anomalous data for the training set on which to train $p(x)$. Then take a smaller proportion of mixed anomalous and non-anomalous data examples (you will usually have many more non-anomalous examples) for your cross validation and test sets.

For example, we may have a set where 0.2% of the data is anomalous. We take 60% of those examples, all of which are good ($y=0$) for the training set. We then take 20% of the examples for the cross-validation set (with 0.1% of the anomalous examples) and another 20% from the test set (with another 0.1% of anomalous)

In other words, we split the data in 60/20/20, training/CV/test, and then split the anomalous examples 50/50 between the CV and test sets.

Algorithm evaluation:

- Fit model $p(x)$ on training set $\{x^{(1)}, \ldots, x^{(m)}\}$
- On a cross validation / test example x, predict:
  - if $p(x) < \varepsilon$, then $y = 1$
  - if $p(x) \geq \varepsilon$, then $y = 0$

Possible evaluation metrics:
- True positive / false positive / false negative / true negative
- precission / recall
- F_1 score

* Note that we use the CV set to choose ε

[Anomaly Detection vs Supervised Learning]

When do we use anomaly detection and when do we use supervised learning?

Use Anomaly Detection when ...
- We have very small number of positive examples ($y = 1$ ... 0-20 examples is common) and a large number of negative ($y = 0$) examples
- We have many different 'types' of anomalies and it's hard for any algo. to ~~get~~ learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've look so far.

Use Supervised Learning When ...
- We have a large number of both positive and negative examples. In other words, the training set isn't skewed, it is more evenly divided into classes.
- We have enough positive examples for the algorithm to get a sense of what new positive examples look like. the future positive examples are likely similar to the ones in the training set.

Choosing what features to use

I.E: create new features like $\frac{CPU}{\text{network usage}}$ or $\frac{CPU^2}{\text{Network}}$

The features will greatly affect how well our anomaly detection algorithm works; we can check that our features are gaussian by plotting a histogram of our data and checking for the bell shaped curve. Some transforms we can try on an example feature x that does not have a bell shaped curve:
- $\log(x)$
- $\log(x+1)$
- $\log(x+c)$ for some constant
- $\sqrt{x} = x^{1/2}$
- $x^{1/3}$

We can play with each of these to try and achieve the gaussian shape in our data

There is an error analysis procedure for anomaly detection that is very similar to the one in supervised learning. Our goal ~~to~~ for $p(x)$ to be large for normal examples and small for anomalous. One way is to examine the anomalous and come out with new features that will better distinguish the data. In gen. choose features that will be unusually large or small in the event of anomaly

[Multivariate Gaussian Distribution]

Multivariate Gaussian Distribution is an extension of anomaly detection and may or may not catch more anomalies.

Instead of modeling $p(x_1), p(x_2)$... separately, we will model $p(x)$ all in one go. Our parameters will be $\mu \in \mathbb{R}^n$ and $\Sigma \in \mathbb{R}^{n \times n}$.

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

The important effect is that we can model along gaussian contours, allowing us to better fit data that might not fit into the normal circular contours. Varying Σ changes the shape, width and orientation of the contours, changing μ will move the center of the distribution.

[Anomaly Detection using the Multivariate Gaussian Distribution]

When doing anomaly detection with multivariate gaussian distribution, we compute μ and Σ normally. We then compute $p(x)$ using the new formula in the previous section and flag an anomaly if $p(x) < \varepsilon$.

The original model for $p(x)$ corresponds to a multivariate Gaussian where the contours of $p(x; \mu, \Sigma)$ are axis-aligned.

The original model for maintains some advantages: it is computationally cheaper as there is no matrix to invert, which is costly for a large number of features... and it performs well even with small training set size. (in multivariate Gaussian model, it should be greater than the number of features for Σ to be invertible).

The multivariate gaussian model can automatically capture correlations between different features of x.