Machine Learning

https://www.youtube.com/watch?v=PPLop4L2eGk&list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN Andrew NG course

Supervised learning

- Regression problem: predict continuous valued output
- Classification problem: predict discrete valued output

Unsupervised learning

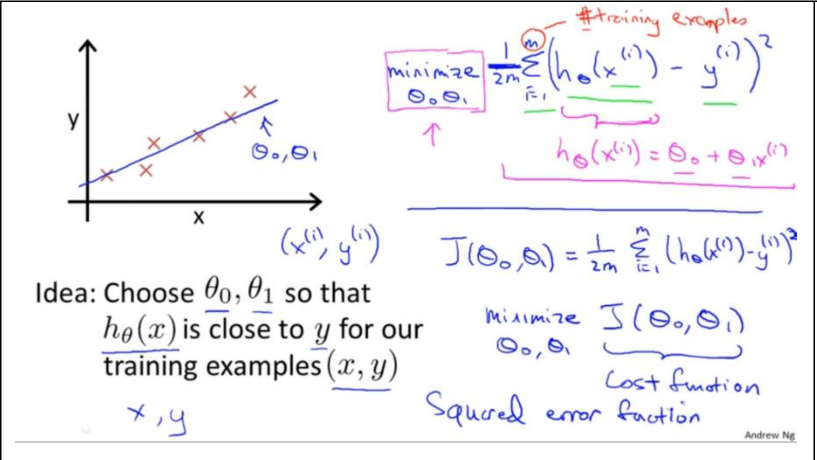
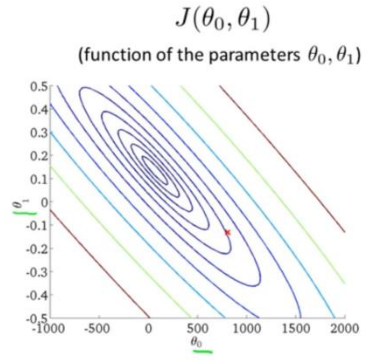
- Clustering problem: Here is the data, can you find some structure on them. A clustering algorithm can cluster the data in different clusters.

---> Supervised Learning <---

Linear Regression

Univariate linear regression

Univariate (one variable) linear regression. For example find a linear function that maps the footage of houses (x) to their price (y). The official terminology for the function is "h" from hypothesis. m is the number of training examples. $H_\theta(x)=\theta0+\theta1*x$. θ0 and θ1 are the parameters of the model. The formalization of this problem is to Minimize over θ0 and Θ1 the sum of squared differences between predictions and real values [We multiply the sum by 1/2m, minimizing the (half of) average error in order to simplify the math along the way]. minimize over θ0 and θ1 means find the value Θ0 and θ1 that minimize this expression which is a function of θ0 and θ1. this expression J is called the cost function which is the objective function we want to minimize.

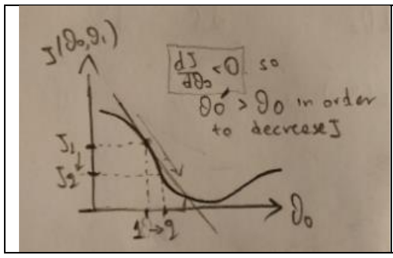
	Contour plots: represent a 3d graph in a 2d graph where continuous lines have the same value for the 3rd coordinate. 

linear regression's cost function is always a convex function which means that it doesn't have local optima but only one global optimum.
m: number of data points (training examples)
n: number of parameters of the hypothesis function (number of features) that we want to fit to the data

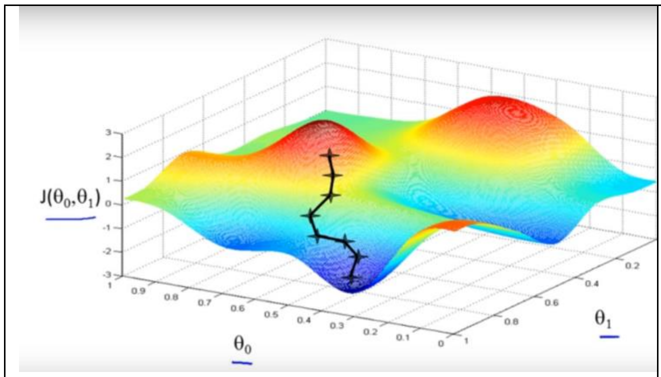
Gradient descent

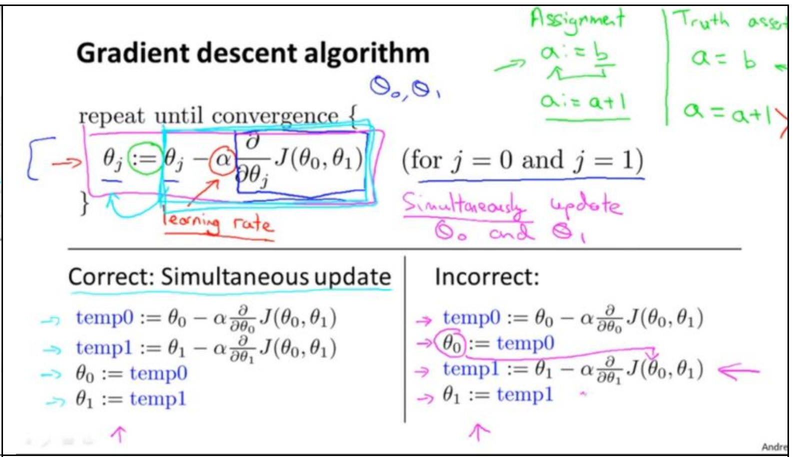
So how you minimize the cost function? One way is using the gradient descent algorithm

In order to find the hypothesis that best fits the data, we need to find the hypothesis parameters that minimize the cost function. Gradient descent is one method that can be used to minimize a function, a function of the form J(θ0, θ1, ... , θn).

	Each point of the surface corresponds to a specific hypothesis function, which has its own cost value. You begin by selecting a random point in the surface. You consider one parameter as constant which means that you get a slice of the surface, which gives you a 2d function of the remaining variable. You want to move on that surface to the direction that makes the y value (the cost) smaller. The easiest way to do this is to calculate the derivative in the point in which you currently are. The derivative can be calculated by the data and the current hypothesis predictions.

If the derivative is negative you have to move to the direction that increases the variable (and decreases the cost). Then you do the same for the other variable. The two movements, one in each direction gives you your total move and your new position on the surface. When you reach to a minimum the derivatives there would be zero so the algorithm will converge, meaning that the new values for the variables would be equal to the previous ones.





Gradient descent algorithm

θ_0, θ_1

repeat until convergence {

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

}

learning rate

Simultaneously update θ_0 and θ_1

Assignment

$a := b$

$a := a+1$

Truth asse

$a = b$

$a = a+1$

Correct: Simultaneous update

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

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

$\theta_0 := \text{temp0}$

$\theta_1 := \text{temp1}$

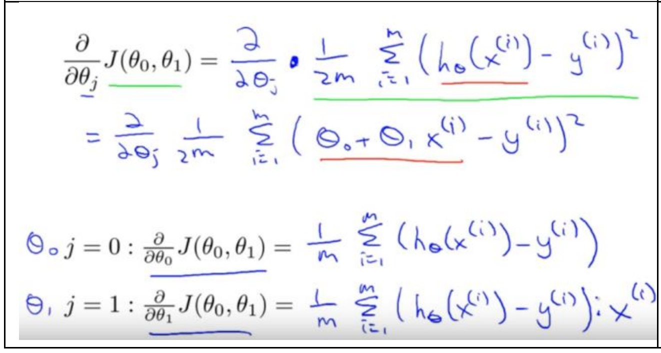
Incorrect:

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

$\theta_0 := \text{temp0}$

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

$\theta_1 := \text{temp1}$



$$\frac{\partial}{\partial\theta_j} J(\theta_0,\theta_1) = \frac{\partial}{\partial\theta_j} \cdot \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

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

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

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

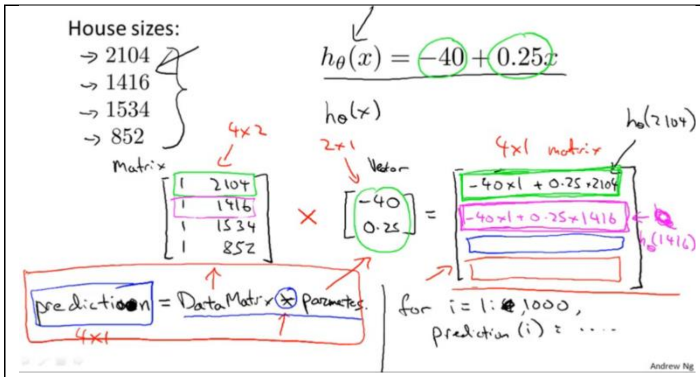
The parameters of the cost function (θ0 and θ1 here) must be updated simultaneously meaning that we first find the new θ0 and then we find the new θ1 using the old θ0. Then we update θ0 and θ1 to get the new values. These describe a new hypothesis function, with a new cost value which is depicted as a point in the cost function surface. Then we repeat. As we approach a local minimum gradient descent will automatically take smaller steps since the derivative starts to approach 0 so it becomes smaller and smaller. So there is no need to decrease the learning rate (α) over time.
You can calculate the formulas for the derivative terms and then use them to run gradient descent.

Notice that if you try to minimize a random function that has many local optima, then **gradient descent would converge to one of those local optima without searching for the global optimum**. But what is important with **linear regression is that its cost function is always a convex function which means that it doesn't have local optima but only one global optimum**.

Batch vs stochastic gradient descent
In a linear regression problem, at each step of gradient descent the gradient descent algorithm uses all the training examples. It calculates the next values for its parameters by minimizing the average cost (the squared distance from all training examples). This is called Batch gradient descent.
I guess that in stochastic gradient descent you randomly select some training examples at each step, so you actually create a new cost function (a new surface) at each step which is an approximation of the real one and minimize there. As I saw most real world cases use stochastic gradient descent because it converges faster.

The normal equation method
Have in mind that in linear regression you can analytically calculate the minimum of the cost function without using an iterative process like gradient descent. That method is called the normal equation method. It has pros and cons, one advantage is that you don't need to care about setting a proper learning rate since it doesn't use such a thing. But it turns out that gradient descent scales much better for large datasets.

Vectorization
The partial derivatives of the cost function with respect to the parameters of the hypothesis are calculated in each time step and they include the calculation of the hypothesis for all the training data. We can just iteratively evaluate for each data point or we can reform the problem in a way that it can be calculated more efficiently.



House sizes:

2104
1416
1534
852

$h_\theta(x) = -40 + 0.25x$

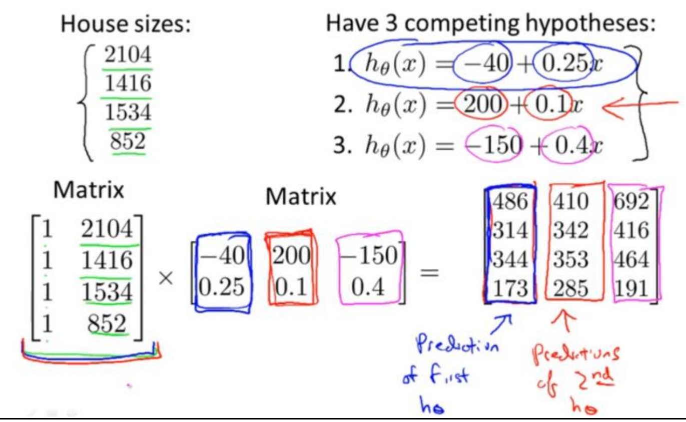
$h_\theta(x)$

$$\underset{4\times 2\ \text{Matrix}}{\begin{bmatrix}1 & 2104\\ 1 & 1416\\ 1 & 1534\\ 1 & 852\end{bmatrix}} \times \underset{2\times 1\ \text{Vector}}{\begin{bmatrix}-40\\ 0.25\end{bmatrix}} = \underset{4\times 1\ \text{matrix}}{\begin{bmatrix}-40\times 1 + 0.25\times 2104\\ -40\times 1 + 0.25\times 1416\\ \\ \end{bmatrix}}$$

$\text{prediction} = \text{DataMatrix} \times \text{Parameters}$

4x1

```
for i = 1:1000,
    prediction (i) = ....
```



House sizes:

2104
1416
1534
852

Have 3 competing hypotheses:

1. $h_\theta(x) = -40 + 0.25x$
2. $h_\theta(x) = 200 + 0.1x$
3. $h_\theta(x) = -150 + 0.4x$

Matrix

Matrix

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

You just must create a Data matrix by adding a first column of ones. The vectorized solution is much faster and more compact. ***Prediction = Data Matrix * Parameters Vector***

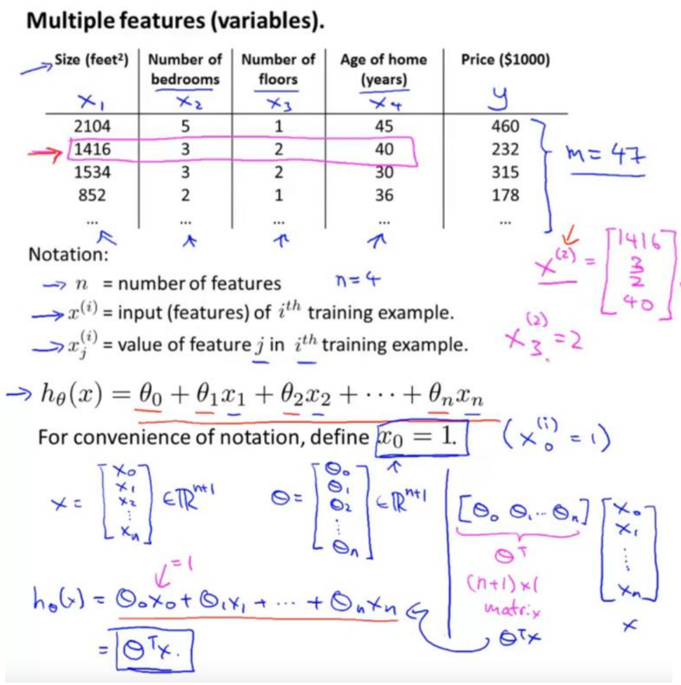
Also, if you have more than one competing hypothesis and you want to calculate the price predictions for each one of them, you can vectorize it using matrix to matrix multiplication.

Have in mind that Matrix multiplication is associative (A*B)*C=A*(B*C) but not commutative A*B != B*A

Multivariate Linear regression

(with multiple variables)
The variables are also called features.

Notations



Multiple features (variables).

Size (feet²) x_1	Number of bedrooms x_2	Number of floors x_3	Age of home (years) x_4	Price ($1000) y
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

Notation:

- n = number of features $n=4$
- $x^{(i)}$ = input (features) of i^{th} training example.
- $x_j^{(i)}$ = value of feature j in i^{th} training example.

$m = 47$

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

$x_3^{(2)} = 2$

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

For convenience of notation, define $\boxed{x_0 = 1.}$ $(x_0^{(i)} = 1)$

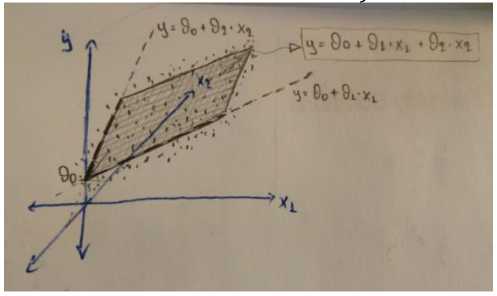
$x = \begin{bmatrix}x_0\\ x_1\\ x_2\\ \vdots\\ x_n\end{bmatrix} \in \mathbb{R}^{n+1}$ $\quad \Theta = \begin{bmatrix}\Theta_0\\ \Theta_1\\ \Theta_2\\ \vdots\\ \Theta_n\end{bmatrix} \in \mathbb{R}^{n+1}$

$\underbrace{[\Theta_0\ \Theta_1 \cdots \Theta_n]}_{\Theta^T \atop (n+1)\times 1\ \text{matrix}} \begin{bmatrix}x_0\\ x_1\\ \vdots\\ x_n\end{bmatrix}_{x}$ $\quad \Theta^T x$

$h_\theta(x) = \Theta_0 \overset{=1}{x_0} + \Theta_1 x_1 + \cdots + \Theta_n x_n$

$= \boxed{\Theta^T x.}$

Now the hypothesis function that best fits to the data describes a multidimensional object (in case of two variables a two d linear surface).

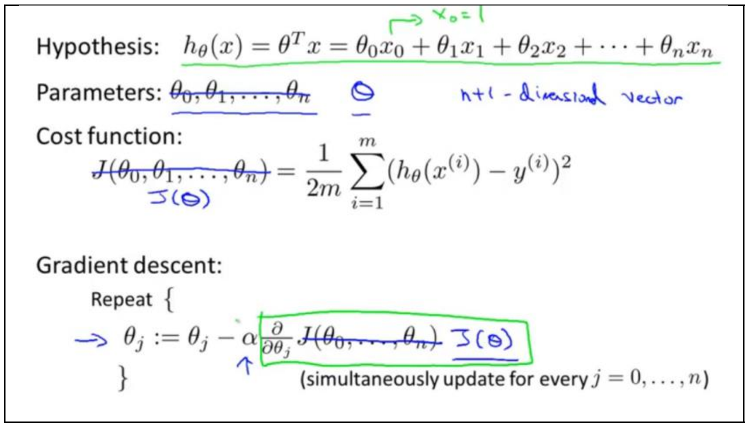


y
$y = \theta_0 + \theta_2 \cdot x_2$
$y = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2$
$y = \theta_0 + \theta_1 \cdot x_1$
x_2
θ_0
x_1

In order to express the hypothesis formulation in a vectorized form, we define an additional feature (feature zero x_0) that has value of one. Using it we can write the hypothesis function as dot product of two vectors, the transpose of the parameters vector and the features vector.

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

Minimizing the cost function in multivariate linear regression with Gradient descent



Hypothesis: $h_\theta(x) = \theta^T x = \overset{x_0=1}{\theta_0 x_0} + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$

Parameters: ~~$\theta_0, \theta_1, \ldots, \theta_n$~~ $\underline{\theta}$ n+1 - dimensional vector

Cost function:

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

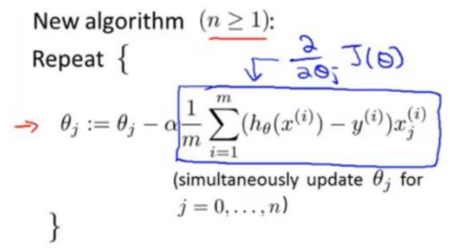
Gradient descent:

Repeat {

$$\theta_j := \theta_j - \alpha \boxed{\frac{\partial}{\partial \theta_j} \cancel{J(\theta_0, \ldots, \theta_n)} \; J(\theta)}$$

} (simultaneously update for every $j = 0, \ldots, n$)

The partial derivative of the cost function with respect to any parameter of the cost function has a common formula.



New algorithm $(n \geq 1)$:

Repeat {

$$\frac{\partial}{\partial \theta_j} J(\theta)$$

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

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

}

So in multivariate gradient descent we just update more parameters in each step, not just two as in linear regression of one variable.

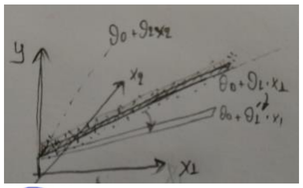
Have in mind that the gradient of a function gives you the direction of steepest ascend. So, to decrease it you move to the opposite direction. The length of that vector gives an indication of how steep that curve is.

Gradient descent

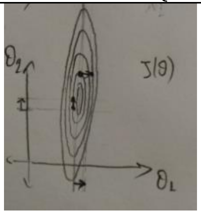
Gradient descent tricks

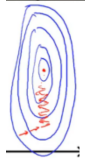
Feature scaling

If the scale of the variables/features are similar (the range of values is similar) then gradient descent converges faster.



$\theta_0 + \theta_2 x_2$
y
x_2
$\theta_0 + \theta_1 \cdot x_1$
$\theta_0 + \theta_1' \cdot x_1$
x_1

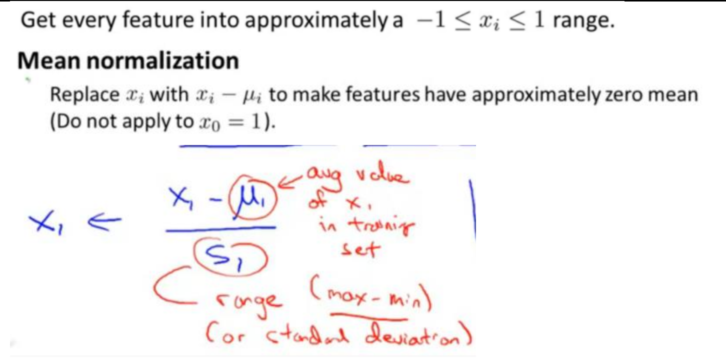




If the scale is not similar, for example x1 (0-2000) and x2 (1-5) then if you select a good θ1 then θ2 has little effect on the cost. You can have a bad θ2 and still have a low total cost (since θ1 is far more important). On the other hand a small change in θ1 would have a big impact on the cost.

This situation causes slow convergence to gradient descent. This happens since initially you get a big step towards the correct θ1 since the gradient of the cost function is big in the θ1 direction and within a couple of steps you have reached the minimum in the θ1 dimension. Then you must move along the θ2 direction which has a very small gradient (the slope in the vertical direction is very small) so the steps would be too small and since you have one learning coefficient you can't increase it since this would cause a big step in the θ1 direction which would increase the cost.

Maybe we can use two different learning coefficients one for each parameter? The common approach is to scale the data appropriately so that the cost contours are uniform.



Get every feature into approximately a $-1 \leq x_i \leq 1$ range.

Mean normalization

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

$$x_1 \leftarrow \frac{x_1 - \mu_1}{s_1}$$

avg value of x_1 in training set

range (max - min) (or standard deviation)

Feature Scaling

The trick is to make them approximately to -1 to 1 range. This can be done by dividing each value of a feature with its range.

You can also apply mean normalization to them, subtracting from each value of a feature its average value and divide with the range (or the standard deviation)

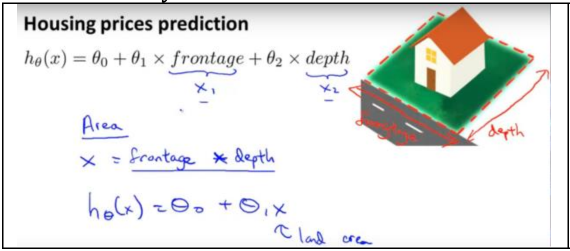
These two steps define the general rule *x=x-μ/s* where s could be the standard deviation of the feature or its range.

Feature choosing

With Choosing/designing features you can achieve:

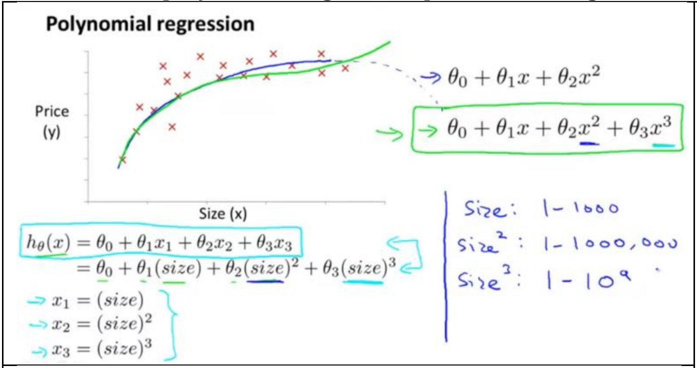
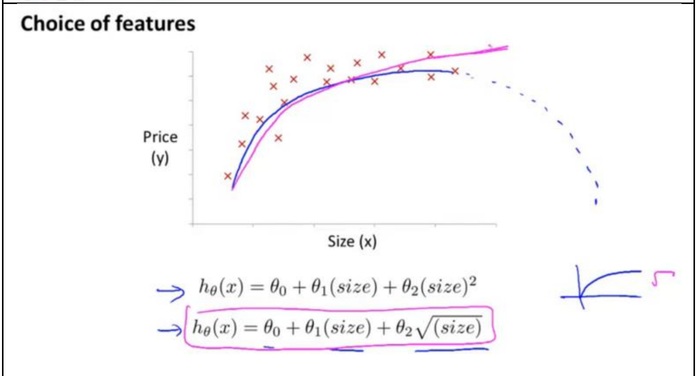
- Feature dimensionality reduction
- Polynomial regression (fitting more complex shapes to your data, not just simple straight lines)

Dimensionality reduction

 $h_\theta(x) = \theta_0 + \theta_1 \times frontage + \theta_2 \times depth$	If you have some insights about the problem you are studying, then you can use them to combine existing dependent features into a new one. This way you reduce the dimensions of the problem which makes it simpler to solve.

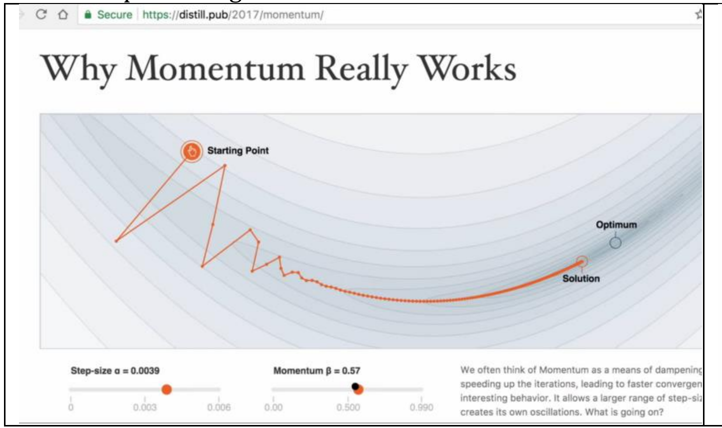
Polynomial regression

You can solve polynomial regression problems using linear regression mechanics, by appropriately redesigning the features.

 $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 = \theta_0 + \theta_1(size) + \theta_2(size)^2 + \theta_3(size)^3$; $x_1 = (size)$, $x_2 = (size)^2$, $x_3 = (size)^3$  $h_\theta(x) = \theta_0 + \theta_1(size) + \theta_2(size)^2$; $h_\theta(x) = \theta_0 + \theta_1(size) + \theta_2\sqrt{(size)}$	There are cases in which a polynomial function would be a better fit to the data than a linear function. For example assume you have a one variable problem (the size of the house). Assume that you know the size and the price of houses form a chart like this. In this case a cubic polynomial would fit well to the data. A quadratic polynomial would fit well to some values but it comes back down again for large values of x so we don't want it. A cubic function would be better. The trick to use linear regression mechanics to solve polynomial regression problem is to redesign the features (feature choosing). in this case we can create two new features where the second and third are the size of the house squared and cubed respectively. But what is also important now, is to do a proper feature scaling. Then we can solve a linear regression problem to identify the correct parameters of the polynomial. Notice that **if you just see the hypothesis function you would assume that it describes a 3 features problem, for which each feature affects linearly the price**, but in reality **it is a one feature problem (the size of the house) that affects cubically the price**. Notice that you could also use another polynomial with a square root. Depending on your insights about the data you can design the features appropriately.

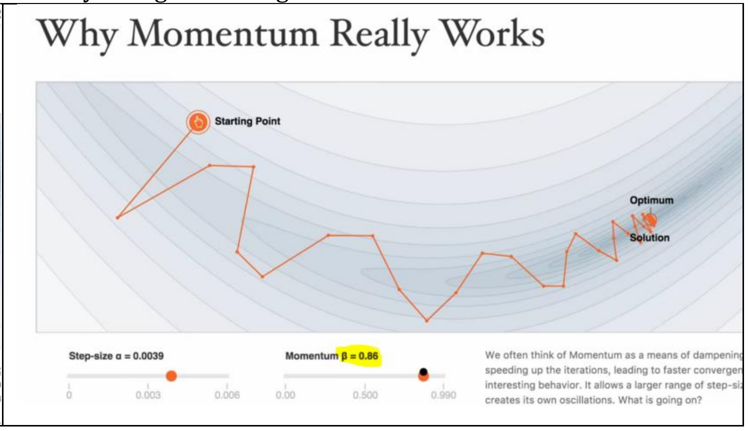
Momentum

(from another source) the model keeps traces of the past gradient directions and doesn't recompute the gradient in each step but uses a previous gradient. It does this for some iterations. This way it might converge faster.



Why Momentum Really Works

We often think of Momentum as a means of dampening ... speeding up the iterations, leading to faster converge... interesting behavior. It allows a larger range of step-si... creates its own oscillations. What is going on?

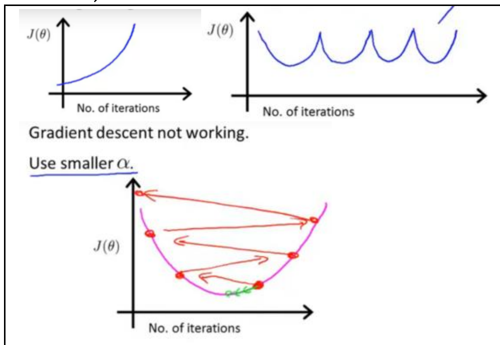


Why Momentum Really Works

We often think of Momentum as a means of dampening ... speeding up the iterations, leading to faster converge... interesting behavior. It allows a larger range of step-si... creates its own oscillations. What is going on?

Convergence

The best way to understand if gradient descent has converged is to make a plot of the cost function relative to the number of iterations. When the plot starts to become parallel to the x axis gradient we can say that gradient descent has converged. Another way is to use automatic convergence which means to set a threshold ε and if the cost decreases by less than ε in one iteration then accept convergence. But it is not very easy to select the proper ε. 10^{-3} might be enough for one application but not for another,



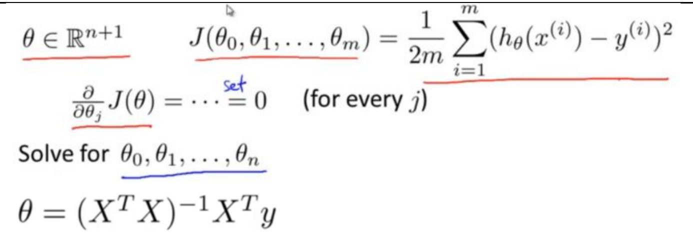
The cost must decrease in every iteration of gradient descent. If instead it increases you should probably decrease α.

- For sufficiently small α, $J(\theta)$ should decrease on every iteration.
- But if α is too small, gradient descent can be slow to converge.

Practically you should try different α values to see how they behave. Testing a range for example 0.001 to 1 by increasing α by a factor of 3 each time. Select the biggest possible that converges.

Normal equation method

Linear regression problems can be solved analytically too, using the normal equation method. You have a multivariate linear regression problem. You add the zero feature. You have a cost function. You can find the extreme values of a function by equating its derivative with 0. This is the main idea.



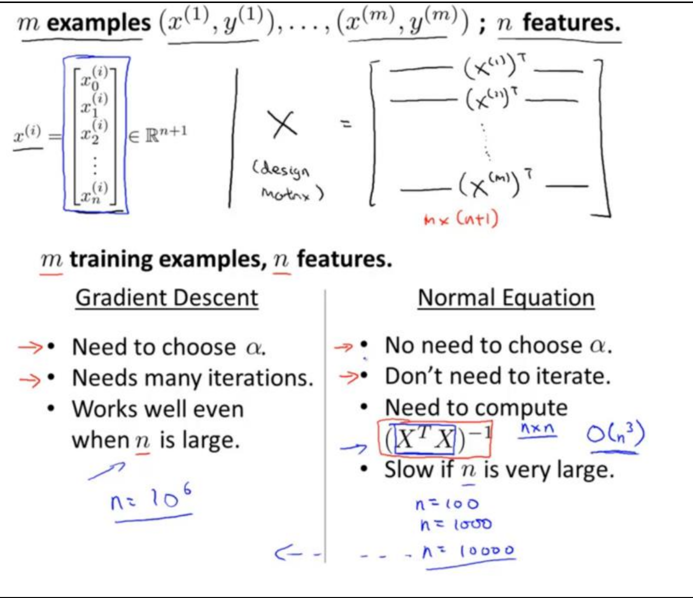
$\theta \in \mathbb{R}^{n+1}$ $\quad J(\theta_0,\theta_1,\ldots,\theta_m) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$

$\frac{\partial}{\partial\theta_j}J(\theta) = \cdots \overset{\text{set}}{=} 0$ (for every j)

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

$\theta = (X^TX)^{-1}X^Ty$

You have to equate with 0, all the partial derivatives of the cost function and then solve a system of equations for the value of the parameters. These values are the values that eliminate the partial derivatives and minimize the function (which is convex so we know that it has a global minimum). This would lead to a function that you have to solve: $\theta = (X^TX)^{-1}X^Ty$



m **examples** $(x^{(1)},y^{(1)}),\ldots,(x^{(m)},y^{(m)})$; n **features.**

$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in \mathbb{R}^{n+1}$ $\quad \underset{\text{(design matrix)}}{X} = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}$ $m \times (n+1)$

m **training examples,** n **features.**

Gradient Descent	Normal Equation
• Need to choose α.	• No need to choose α.
• Needs many iterations.	• Don't need to iterate.
• Works well even when n is large.	• Need to compute $(\boxed{X^TX})^{-1}$ $n\times n$ $O(n^3)$
$n = 10^6$	• Slow if n is very large.
	$n=100$
	$n=1000$
	$n=10000$

X is called <u>the design matrix</u> and its rows are the transposed feature vectors. y is the training data vector.

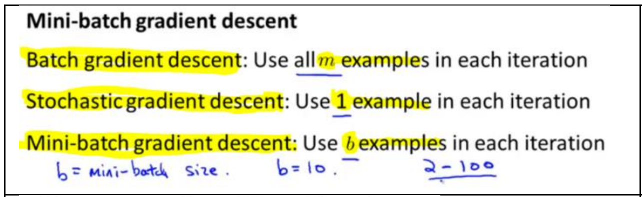
So this means that you have to compute the X^TX matrix and then invert it. This matrix would be a n by n matrix and the <u>inverting operation has an efficiency of O(n^3)</u>. this means that if the number of features is very big this operation would be very expensive which would make gradient descent far more efficient.

Have in mind that some rare cases X^TX might be non-invertible. <u>If a matrix has rows or columns linearly dependent then the matrix is not invertible, (matrix determinant = 0)</u>. Some reasons for that could be 1. that some features are linearly dependent or 2. there are too many features (m<n). Notice that there are libraries that can handle non-invertibility and still solve the problem (pseudo-invertibility)

Notice that you don't need to perform feature scaling for the normal equation method.

Batch, Mini batch and Stochastic gradient descent

When you have a huge number of training examples (m is huge) you must sum over all these examples to calculate the partial derivative used in the gradient descent algorithm. You calculate which direction decreases the cost function for all examples. In the version of stochastic gradient descent, you calculate the error based on only one example. In mini batch you calculate the error based on b examples where usually 2<b<100. So you will make more gradient descent steps to reach to an acceptable solution but each step will be much faster. The steps will not go directly to the direction of biggest decrease, but they will wonder around the cost function somewhat randomly. Stochastic and mini batch gradient descent doesn't find the global optimum, but it oscillates around it.



Mini-batch gradient descent

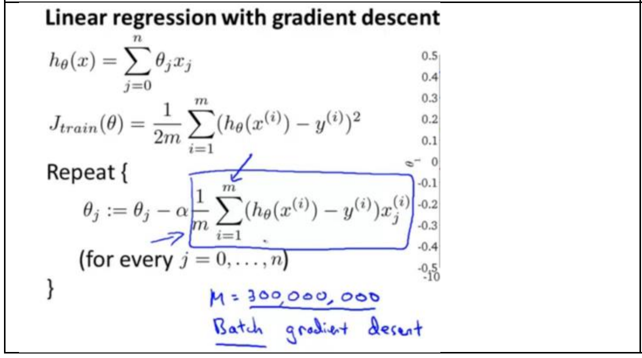
Batch gradient descent: Use all m examples in each iteration

Stochastic gradient descent: Use 1 example in each iteration

Mini-batch gradient descent: Use b examples in each iteration

b = mini-batch size. $b = 10$. 2 - 100

Minibatch might be more efficient than stochastic gradient descent if you have a good library for vectorized calculations. The disadvantage is that you have an additional hyperparameter, the batch size, to tune.



Linear regression with gradient descent

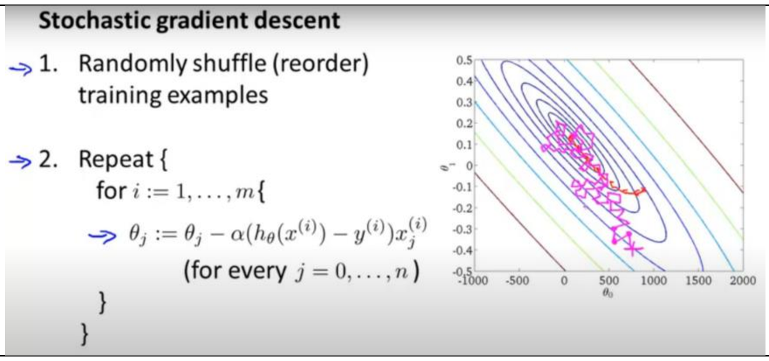
$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$$

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

```
Repeat {
    $\theta_j := \theta_j - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
        (for every $j = 0, \dots, n$)
}
```

$M = 300{,}000{,}000$

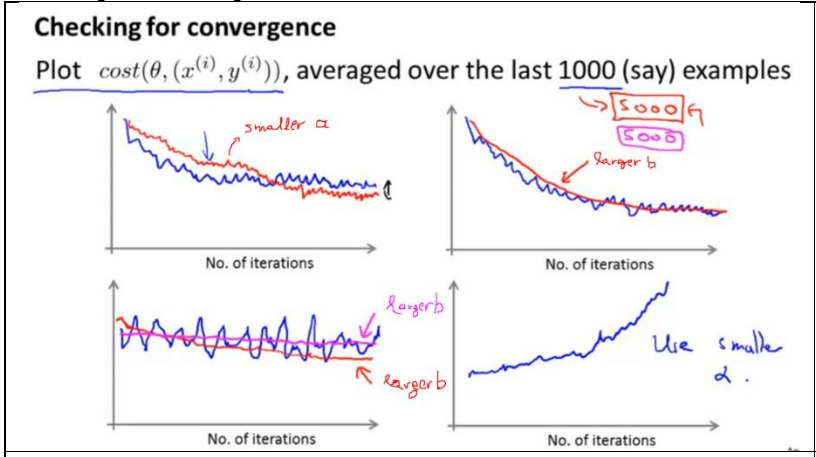
Batch gradient descent



Stochastic gradient descent

```
1. Randomly shuffle (reorder) training examples
2. Repeat {
       for $i := 1, \dots, m${
           $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
               (for every $j = 0, \dots, n$)
       }
   }
```

Checking for convergence

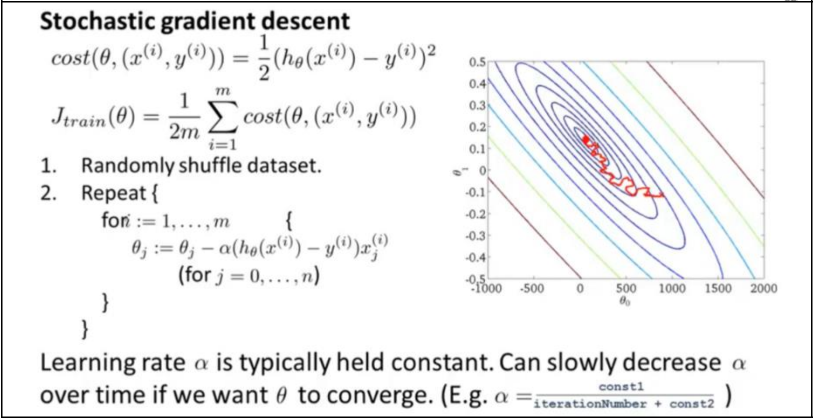


Checking for convergence

Plot $cost(\theta, (x^{(i)}, y^{(i)}))$, averaged over the last 1000 (say) examples

Stochastic and mini batch gradient descent doesn't find the global optimum, but it oscillates around it.

To check convergence you plot the learning curve. These are some cases. In the case where it is flat, the algorithm doesn't learn and you might try to find other features or more data etc.
If it diverges use smaller learning rate a.



Stochastic gradient descent

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

$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

```
1. Randomly shuffle dataset.
2. Repeat {
       for $i := 1, \dots, m$ {
           $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
               (for $j = 0, \dots, n$)
       }
   }
```

Learning rate α is typically held constant. Can slowly decrease α over time if we want θ to converge. (E.g. $\alpha = \frac{\texttt{const1}}{\texttt{iterationNumber + const2}}$)

In most cases we are ok with the result of stochastic gradient descent. But if we want a solution closer to the optimum we can try to decrease a as the iterations progress. Although this isn't used often, because we have two additional hyperparameters to tune.

Misc

Minibatch

- The only reason to use minibatch vs stochastic is the parallelization.
- Its good to have in the minibatch samples that are different with each other.
- Minibatch size determined by the hardware you have. For large ANNs and a gpu let's say 16-64
- If you increase the batch size too much (in a big cluster for example) you accelerate the calculation due to increased parallelization but you decrease the speed of convergence. So there is a limit until which it has meaning to increase the batch size.
- The size must not be larger than the number of classes

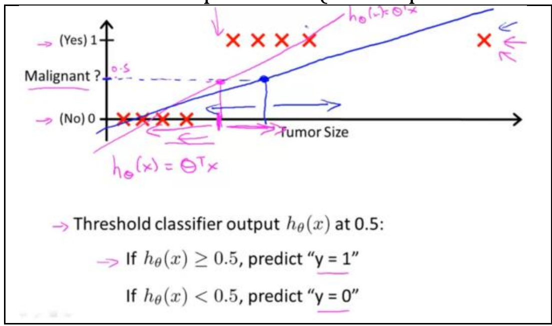
A tip on normalization

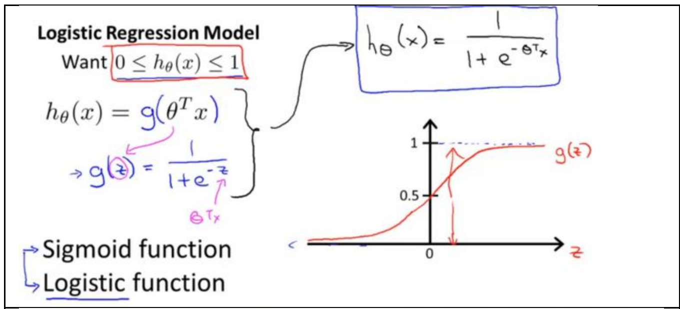
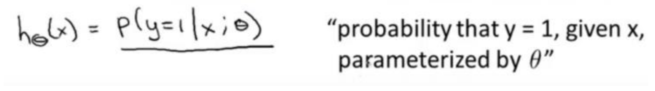
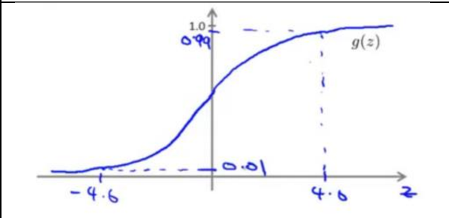
Notice that when you normalize the features you might not need to calculate the mean and deviation based on all dataset, because these quantities converge quite fast.

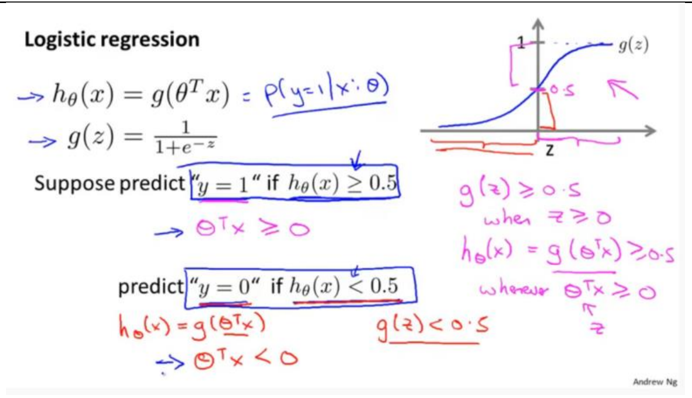
Normalization on the weighted sum values
Apart from normalizing the features, it's been shown that it is also important to normalize the internal state of the ANNs too, the weighted sum values in other words. A technique to do this is called batch normalization. I guess that t1his is the reason why you initialize weights with He and similar techniques. They try to achieve a weighted sum which is in the same range as its inputs (zero mean and unit variance)

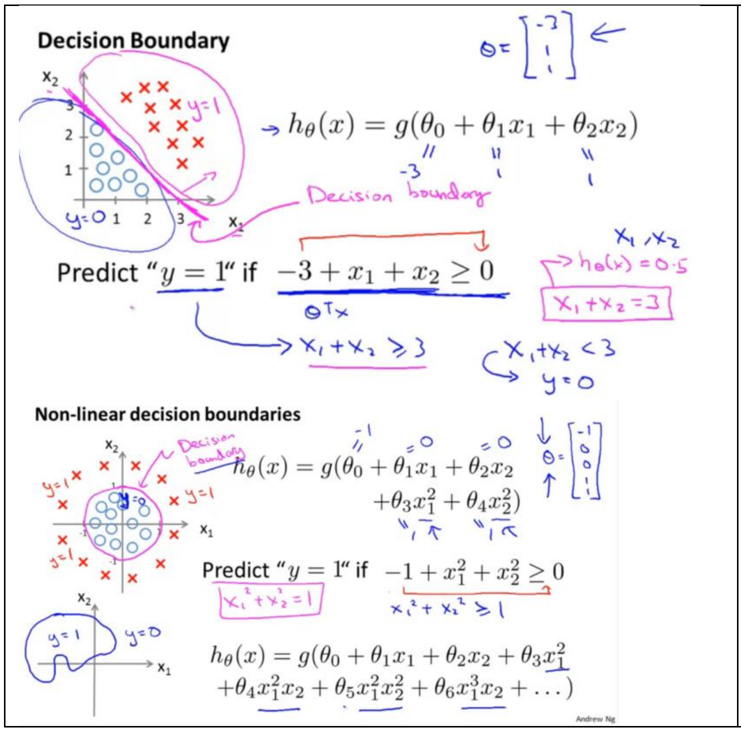
Logistic (sigmoid) Regression

For classification problems (the outputs are discrete values)

	Linear regression is not good for classification problems. You might get lucky and it might produce good predictions (using a threshold classifier output) but just one outlier can make it worthless.

 Logistic Regression Model Want $0 \leq h_\theta(x) \leq 1$ $h_\theta(x) = g(\theta^T x)$ $g(z) = \frac{1}{1+e^{-z}}$ $\theta^T x$ $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$ Sigmoid function Logistic function 1 0.5 0 g(z) z	In logistic regression we want the prediction h(x) to be between 0 and 1. To achieve that we transform h(x) (passing through a function g) in such a way that it would always produce values in the desired range. Specifically we transform it to a logistic (sigmoid) function. In linear regression h(x)= $\theta^T x$ while in logistic regression we pass that through the sigmoid function. Now a training point x^i corresponds to a specific z value ($z=\theta^T x$) and this z value has a g(z) output which is the prediction value between 0 and 1.
 $h_\theta(x) = P(y=1 \mid x; \theta)$ "probability that y = 1, given x, parameterized by θ"	We interpret the output of the prediction function as a probability of the specific input to give 1 as output. For example for a specific x1 if h(x1)=0.7 it means that the feature (the characteristic) with value x1 has 70% probability of being 1 (whatever 1 might mean, for example cat). The formal way of saying this is by saying that the prediction h(x) gives the probability that y=1 given x parameterized by θ.
	Have in mind these values. A value of **4.6** corresponds to a logistic value of 0.99 and **-4.6** to 0.01

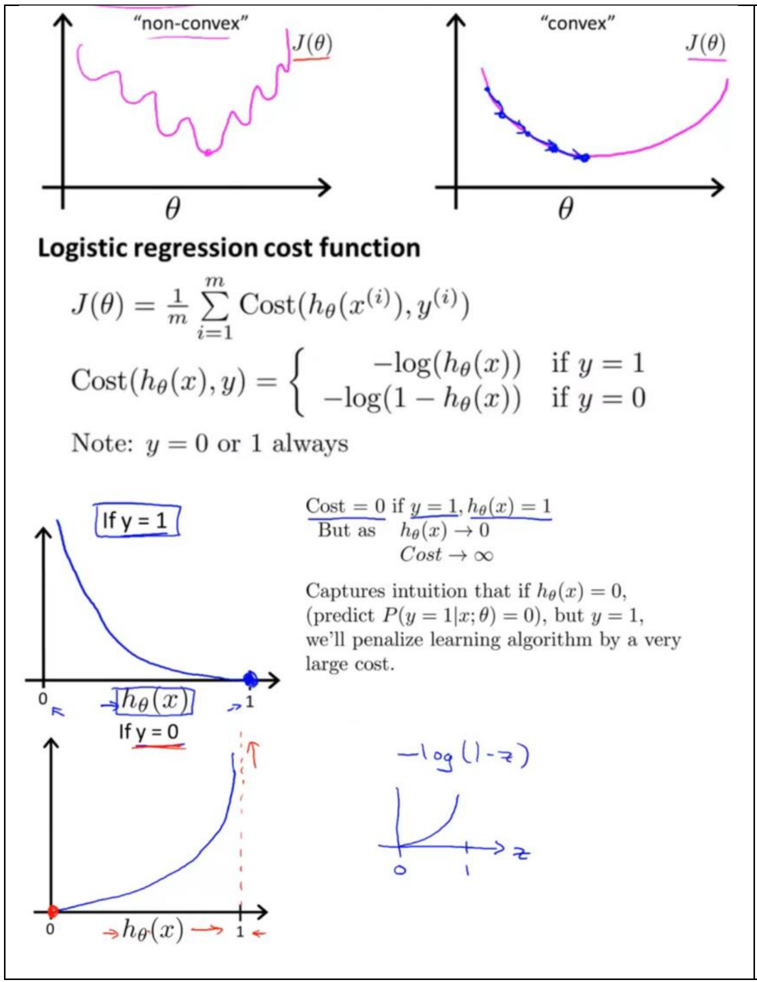
 Logistic regression $h_\theta(x) = g(\theta^T x) = P(y=1 \mid x; \theta)$ $g(z) = \frac{1}{1+e^{-z}}$ Suppose predict "$y = 1$" if $h_\theta(x) \geq 0.5$ $\theta^T x \geq 0$ predict "$y = 0$" if $h_\theta(x) < 0.5$ $h_\theta(x) = g(\theta^T x)$ $\theta^T x < 0$ $g(z) \geq 0.5$ when $z \geq 0$ $h_\theta(x) = g(\theta^T x) \geq 0.5$ whenever $\theta^T x \geq 0$ z $g(z) < 0.5$ 1 0.5 g(z) Andrew Ng	As with linear regression we try to minimize the cost by finding the proper parameter values θ. For a given data point x, each different θ gives a different z which gives a different g(z). By modifying θ you don't affect the logistic function which is always the same, being 0.5 for z=0. What we do is this. For each training point (x,y) if y=1 then we say that the correct prediction for that x, is h(x) or g(z) to be >= 0.5. But g(z) is >= 0.5 when z>=0 or $\theta^T x$>=0. This means that for points of the data set that have y=1, $\theta^T x$ must be >=0.


Decision Boundary
$\theta = \begin{bmatrix} -3 \\ 1 \\ 1 \end{bmatrix}$
$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$
y=1
y=0
Decision boundary
Predict "$y = 1$" if $-3 + x_1 + x_2 \geq 0$
$\theta^T x$
x_1, x_2
$h_\theta(x) = 0.5$
$x_1 + x_2 = 3$
$x_1 + x_2 \geq 3$
$x_1 + x_2 < 3$
$y = 0$
Non-linear decision boundaries
Decision boundary
$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$
$\theta = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$
Predict "$y = 1$" if $-1 + x_1^2 + x_2^2 \geq 0$
$x_1^2 + x_2^2 = 1$
$x_1^2 + x_2^2 \geq 1$
$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^2 x_2^2 + \theta_6 x_1^3 x_2 + \dots)$
Andrew Ng

Suppose that you have somehow calculated the correct values for the parameters θ, in this example [-3 1 1]. The function $\theta^T x$ expands to -3+x1+x2. If we want to predict 1 for points x for which y=1, then as we said **we want $\theta^T x>0$ -> -3+x1+x2>0 -> x1+x2>3.** This equation defines a line. If a point x is on the right of that line then z or $\theta^T x$ would be greater than 0 and g(z) would be greater than 0.5 so the prediction would be 1. This line is composed of all points x for which the prediction is exactly 0.5 and is called the **decision boundary**. The decision boundary is a property of the prediction function (of its parameters) and not of the training data (the parameters though are chosen based on the training data). **In linear regression we plot the prediction function itself, while in logistic regression we equate it to 0 which makes it a whole different function (the decision boundary) and plot that**.

As with polynomial regression we can add some polynomial terms to the prediction function in order to achieve non-linear decision boundaries.

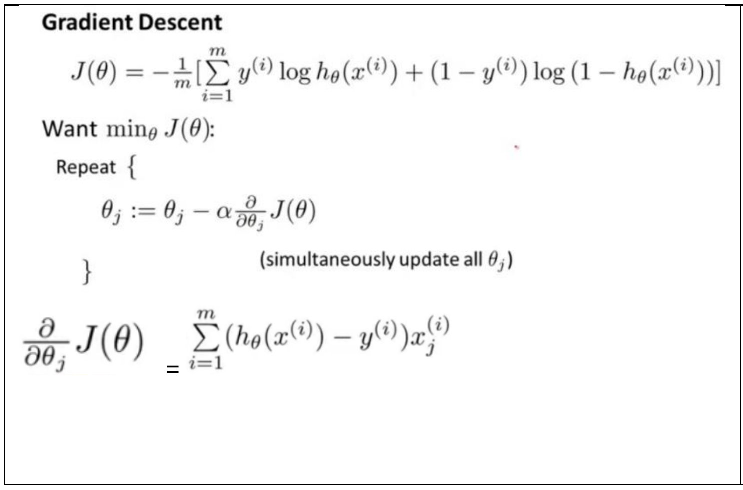
If z has a large variance it takes values that span the non linear part of the g(z).


"non-convex"
"convex"
$J(\theta)$
θ
Logistic regression cost function
$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)})$
$\mathrm{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$
Note: $y = 0$ or 1 always
If y = 1
Cost = 0 if $y = 1, h_\theta(x) = 1$
But as $h_\theta(x) \to 0$, $Cost \to \infty$
Captures intuition that if $h_\theta(x) = 0$, (predict $P(y = 1|x;\theta) = 0$), but $y = 1$, we'll penalize learning algorithm by a very large cost.
$h_\theta(x)$
If y = 0
$-\log(1-z)$
z

If we use the linear regression cost function due to the fact that the h(x) is now a non linear function (the logistic function), the squared cost function J(x) would be non-convex. This means that gradient descent would not be guaranteed to converge to the global minimum. So we have to define a different cost function.

The new cost function is actually two functions.
This way if y=1 and the prediction h(x)=1 the function -logh(x) gives 0 which means that the cost is 0 since the prediction is correct. But if the prediction is 0 then the cost is infinite.
The opposite applies to cases for which y=0.

We chose that specific cost function based on maximum likelihood estimation. It also has the nice property that it is convex.



Gradient Descent

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

Want $\min_\theta J(\theta)$:

Repeat {

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

} (simultaneously update all θ_j)

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

The cost function can be written in the following form too:

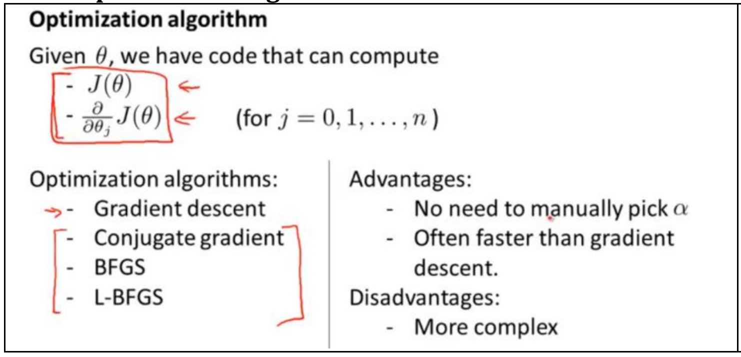
$$Cost(h_\theta(x), y) = -y\log(h_\theta(x)) - (1-y)\log(1-h_\theta(x))$$

which is a more compact form. Notice that the first term is a product of a quantity with y and the second term a product of a quantity with 1-y. so if y=1 the second term becomes 0 and when y=0 the first term becomes 0.

It turns out that the partial derivative of this cost function is of the same generic form with the one for linear regression with the difference that now $h_\theta(x) = 1/1 + e^{-\Theta^T \cdot x}$. We can also vectorize the calculations.

Have in mind that feature scaling is also needed for gradient descent to converge efficiently in logistic regression too.

Other optimization algorithms



Optimization algorithm

Given θ, we have code that can compute

- $J(\theta)$
- $\frac{\partial}{\partial\theta_j} J(\theta)$ (for $j = 0, 1, \ldots, n$)

Optimization algorithms:

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

Advantages:

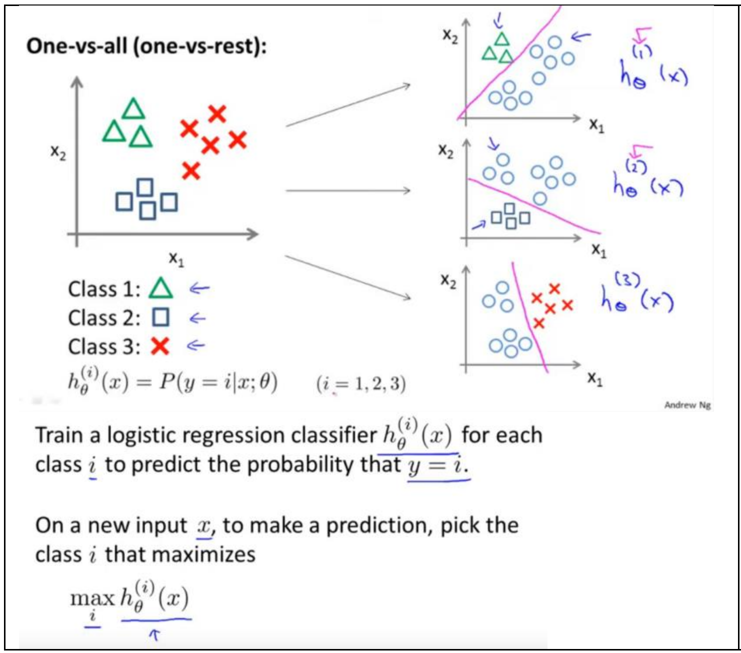
- No need to manually pick α
- Often faster than gradient descent.

Disadvantages:

- More complex

Apart from gradient descent there are other optimization methods too, that usually converge much faster. In addition you don't have to explicitly define the learning rate since they use a line search algorithm to automatically define a learning rate which can be different for each step.
Similar to gradient descent these algorithms use the cost function and its partial derivatives so you need to provide these to them.
Their disadvantage is that they are more complex to implement so you need to pick a library that uses a good implementation.

Multiclass classification
One versus All method



Train a logistic regression classifier $h_\theta^{(i)}(x)$ for each class i to predict the probability that $y = i$.

On a new input x, to make a prediction, pick the class i that maximizes

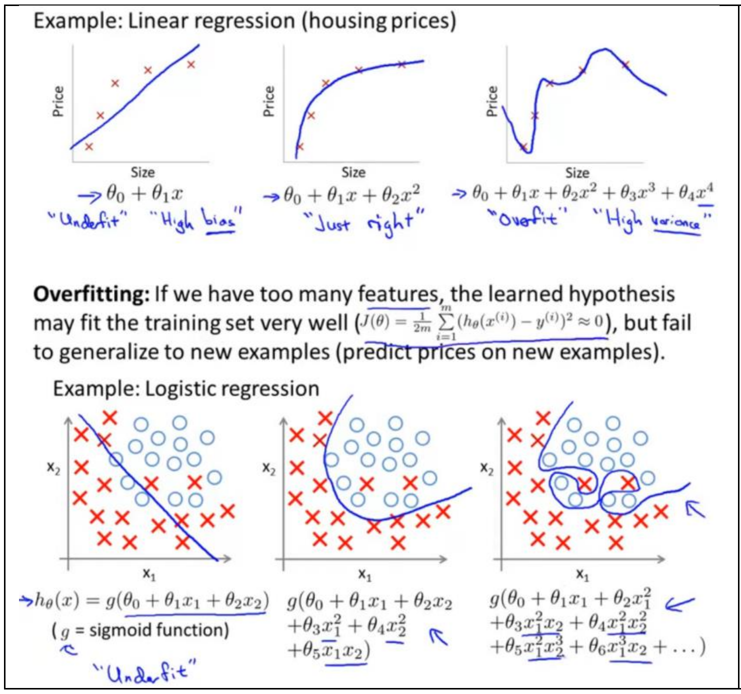
$$\max_i h_\theta^{(i)}(x)$$

One method for doing it is the One vs All method where we train a logistic regression classifier for each class. Then when we have a new data point and we want to classify it we calculate the predictions from all classifiers and the one that is more certain about the new input wins.

Overfitting

Intro

Overfitting means your model does much better on the training set than on the test set. It fits the training data too well and generalizes bad. The main reason for overfitting is sparse data.


Price
Size
"Undefit" "High bias"
"Just right"
"Overfit" "High variance"
x_2
x_1
"Undefit"

Example: Linear regression (housing prices)

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

Overfitting: 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 (predict prices on new examples).

Example: Logistic regression

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

(g = sigmoid function)

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

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

We say that the hypothesis has high variance or high bias.

High bias (underfit)
For example, if we use a line, the model has a preconception that the price of houses depend linearly on the size despite the data to the contrary

High variance (overfit)
If we use a high order polynomial then it has the possibility to fit to a great variety of data sets, it is very flexible, <u>the space of possible hypothesis is too large</u>, too variable and we don't have enough data to constrain it to give us a good hypothesis

Why too many features cause overfitting
Adding more features expands the hypothesis space making the data more sparse and this might lead to overfitting problems.

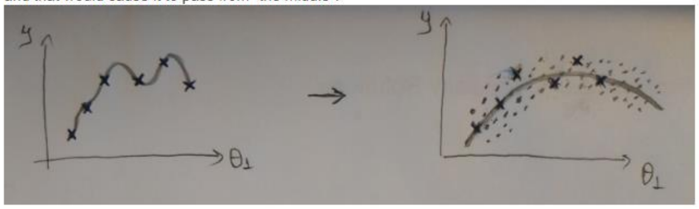
I had exactly the same question and since I wasn't able to really understand the reason based on the existing answers, I made some additional search and thought about it for a while. Here is what I found. Feel free to correct me where I'm wrong.

The main reason for overfitting is **sparse data** (for a given model).

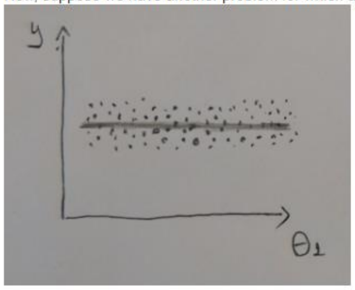
The three reasons that are mentioned in this answer could be narrowed down to "sparse data" for your given problem. This is an important concept to understand since the sparsity of the data depends on the number of features.

It is always easier to understand any concept if you think of it in its simplest form and find a way to visualize it.

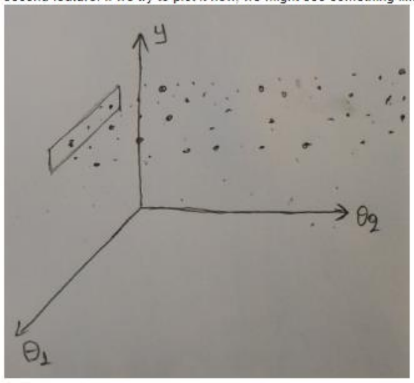
So let's see first how sparse data can cause overfitting using a two dimensional plot for a problem with one parameter. If your hypothesis is a high order polynomial and the number of data points small, then it would just overfit to the data. If you had more data points though it wouldn't overfit since it would have to minimize the average error from many data points (more than what it could overfit to) and that would cause it to pass from "the middle".



Now, suppose we have another problem for which apparently, data is not sparse.



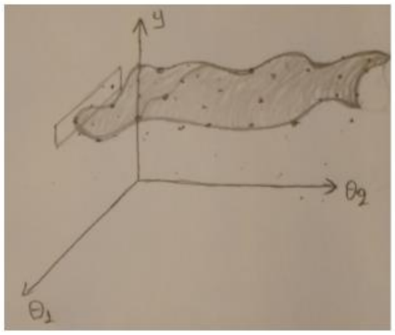
Later though we learn that there is an additional feature in this same problem. This would mean that each data point of the existing training set has also a second value that describes it, the value of the second feature. If we try to plot it now, we might see something like this:



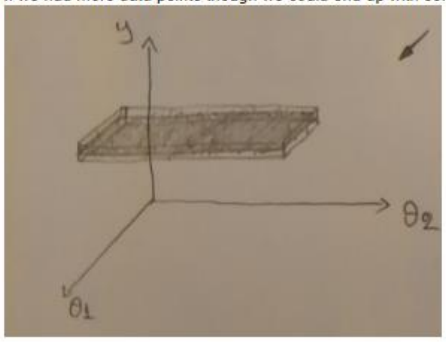
This means that what we were seeing before, was just the projection of the data points in the y,θ1 plane and that was the reason for which we mistakenly assumed that we had enough data points. Now that we see the problem in its entirety, we can say that the data points are not enough, that data is sparse.

Thus, by adding one additional feature we expanded the space of our problem adding one more dimension to it and the data points which are part of this space, were expanded with it.

So if we try to fit a hypothesis to this data we might get something like this, which is probably an overfit.



If we had more data points though we could end up with something like this:



In conclusion, adding more features expands the hypothesis space making the data more sparse and this might lead to overfitting problems.

Addressing overfitting

- <u>Plotting the hypothesis</u>

When we have only one or two features it is very easy to plot the hypothesis function over the features (a 2d or 3d plot) and not only guess what polynomial order we want to use for the hypothesis but also check at the end if the hypothesis is overfitting to the data. If it does then we can choose a polynomial with lower order and try again. When we have many features though it becomes difficult to visually understand what's happening. In these cases we must use other techniques to address overfitting.

- <u>Reduce the number of features</u>

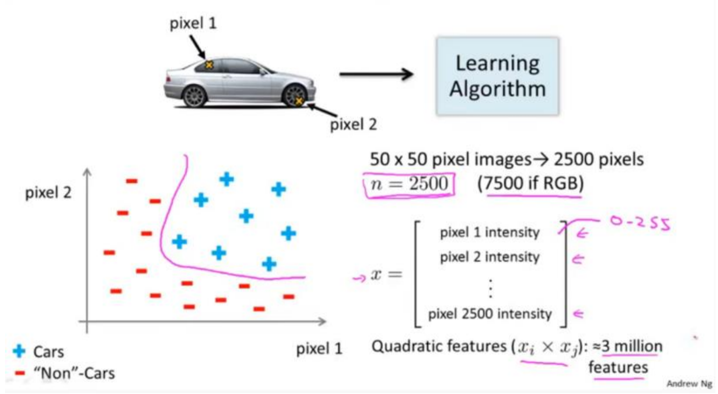
One way to address overfitting is to reduce the number of features. We can either select manually which features to keep or use a model selection algorithm that automatically selects which features to keep. The disadvantage is that all features might be important so by omitting some, we lose valuable information.

- <u>Regularization</u>

Reduce the magnitude of the parameters θ. works well if all features contribute a bit to predicting y so we can't reduce the number of features.

Neural networks

A Neural network is a classifying algorithm that is useful for the creation of non linear hypothesis (non linear function of the input). For problems with many features it is a much more efficient classifier in relation to logistic regression.

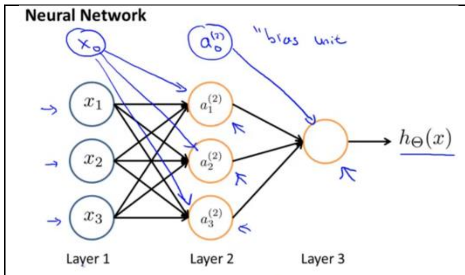

pixel 1
pixel 2
Learning Algorithm
pixel 2
pixel 1
Cars
"Non"-Cars
50 x 50 pixel images→ 2500 pixels
$n = 2500$ (7500 if RGB)
0-255
$x =$ [pixel 1 intensity, pixel 2 intensity, ⋮, pixel 2500 intensity]
Quadratic features ($x_i \times x_j$): ≈3 million features
Andrew Ng

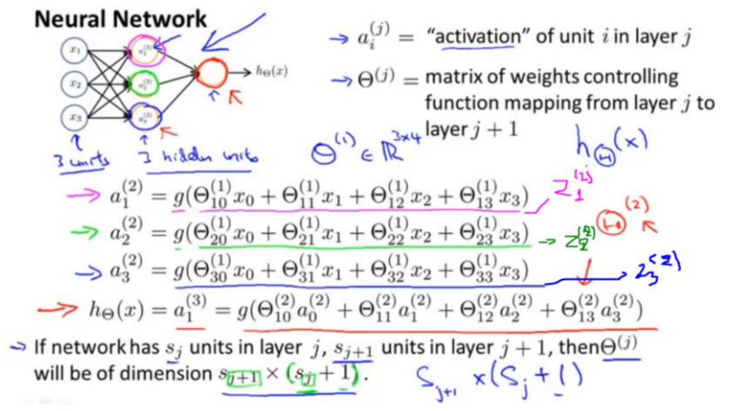
The input to the classifier is the intensity values of the pixels of the input image. It has to classify the input as car or non-car based on them. So the number of input features is the number of pixels.

Theoretically you could use polynomial logistic regression but the problem is that the number of quadratic terms ($x1^2$, x1x2, x1x3, … x1xn) in the hypothesis polynomial is $O(n^2/2)$ and the number of cubic terms is $O(n^3)$. So for a problem with n=100 features you end up with hundreds of thousands of "designed" features (the high order terms) in the hypothesis. This would result in overfitting and performance problems and make it a non viable solution. So we need an alternative.

A picture 50 by 50 pixels has 2500 pixels. If we use the greyscale values for describing pixel intensity and use them as features for a classification problem, then we have 2500 features which would lead in millions of designed features in a polynomial hypothesis.
(the chart's axis are pixel intensity values)

An alternative classification algorithm is an artificial neural network.





Neural Network

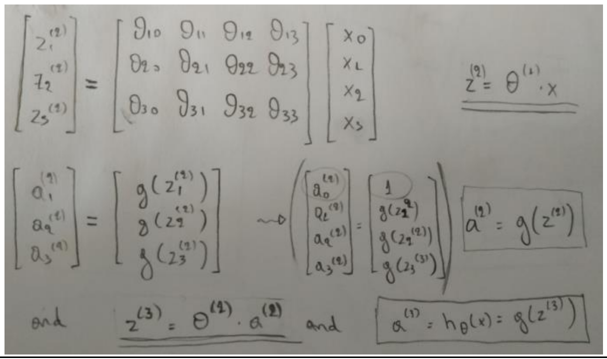
$\rightarrow a_i^{(j)}$ = "activation" of unit i in layer j

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

3 units   3 hidden units $\Theta^{(1)} \in \mathbb{R}^{3\times4}$ $h_\Theta(x)$

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3) \quad z_1^{(2)}$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3) \rightarrow z_2^{(2)} \quad \Theta^{(2)}$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3) \rightarrow z_3^{(2)}$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

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



$$\begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix} = \begin{bmatrix} \theta_{10} & \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{20} & \theta_{21} & \theta_{22} & \theta_{23} \\ \theta_{30} & \theta_{31} & \theta_{32} & \theta_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad z^{(2)} = \Theta^{(1)} \cdot x$$

$$\begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} = \begin{bmatrix} g(z_1^{(2)}) \\ g(z_2^{(2)}) \\ g(z_3^{(2)}) \end{bmatrix} \rightsquigarrow \left(\begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} = \begin{bmatrix} 1 \\ g(z_1^{(2)}) \\ g(z_2^{(2)}) \\ g(z_3^{(2)}) \end{bmatrix} \right) \boxed{a^{(2)} = g(z^{(2)})}$$

and $z^{(3)} = \Theta^{(2)} \cdot a^{(2)}$ and $\boxed{a^{(3)} = h_\theta(x) = g(z^{(3)})}$

A neural network defines a function h that maps from an input space x to a prediction space y. By varying the parameters Θ (which are now called weights) we get a different mapping, or in other words a different function or hypothesis h.

Each neuron's sum of weighted inputs, pass through a logistic function g and the output of that is the neuron's output. This applies also to the output layer. So in this setup neurons output (activation) is always between 0 and 1. The hypothesis is the activation of the only unit of the output layer which is also between 0 and 1.

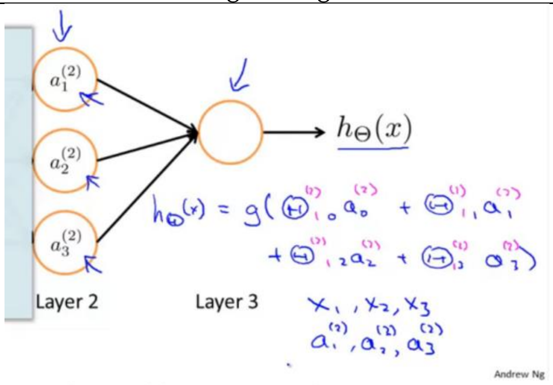
Notice that there is also an additional input x0 which is not always shown in the graphs, that is called **a bias unit** and has a value of 1 (x0=1, though its weights might change). Why do we need it? see explanation below.

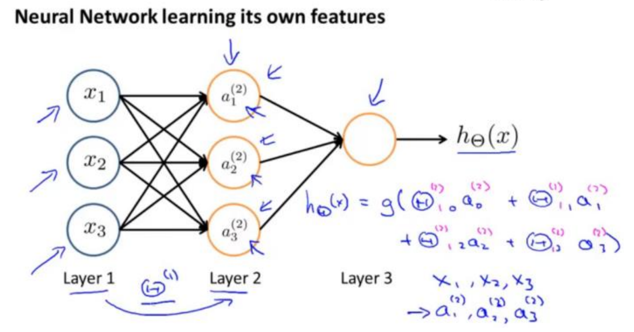
($a_1^{(2)}$ stands for "activation", meaning the output value of the first unit of the second layer)

This process of computing h(x) is called **forward propagation**. We propagate the activation of the input units (the input values) forward to the next layer's hidden units and so on.

The calculations for forward propagation can be vectorized. Notice that we add also a bias unit to the hidden layer whose value (activation) is 1. These activation values will be weighted and similarly will produce the final output (the activation of the output unit)

Neural network as logistic regression


$a_1^{(2)}$
$a_2^{(2)}$
$a_3^{(2)}$
$h_\Theta(x)$
$h_\Theta(x) = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$
Layer 2
Layer 3
x_1, x_2, x_3
$a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$
Andrew Ng

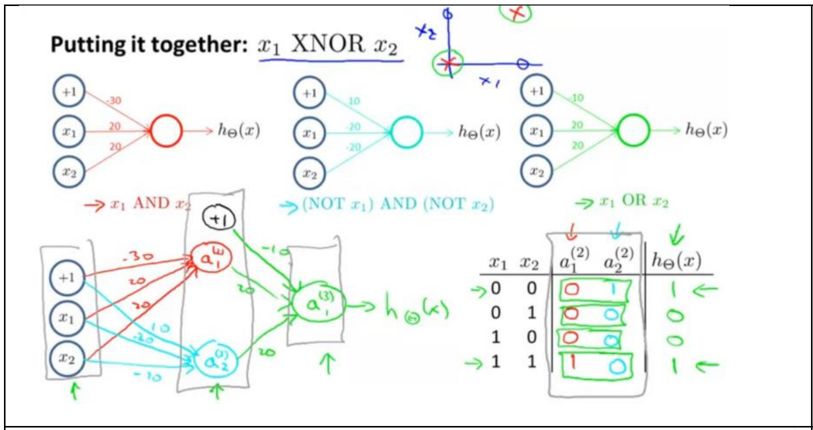

Neural Network learning its own features
x_1
x_2
x_3
$a_1^{(2)}$
$a_2^{(2)}$
$a_3^{(2)}$
$h_\Theta(x)$
$h_\Theta(x) = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$
Layer 1
Layer 2
Layer 3
$\Theta^{(1)}$
x_1, x_2, x_3
$\rightarrow a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$

If you have a neural network with no hidden units then it works exactly as a logistic regression algorithm. The hypothesis produced is g(z) where z= $\theta^T x$.

If instead there is a hidden layer then the inputs to the output unit are not the original features but some other values, some other new complex features that have been learned from the original ones with logistic regression. <u>Each hidden unit represents a new complex feature</u>. This mechanism allows the ANN to be able to form very complex non linear hypothesis to fit to the input data.

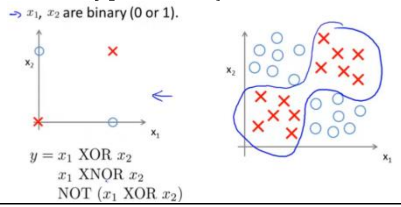
The units of the last layer of a ANN (as any other unit of an ANN) are doing logistic regression on their inputs.

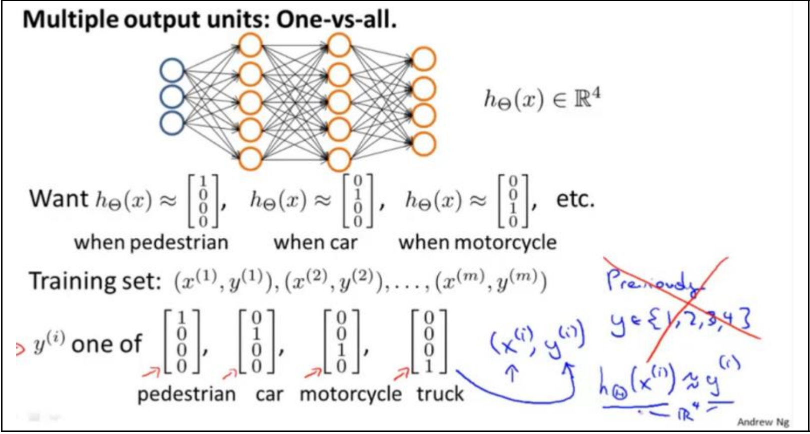
XNOR: Sometimes referred to as an "Equivalence Gate," the gate's output requires both inputs to be the same to produce a high output.	Implementing some logical functions with ANNs. The XNOR implementation requires a hidden layer which



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	0	1

is composed of two units each of which calculates a slightly more complex function of the inputs, namely AND and (NOT x1)AND(NOT x2). Then these more complex inputs are used to implement an even more complex output. This gradual complexity is the reason why ANNs could form really complex non linear hypothesis (and thus decision boundaries).





Multiple output units: One-vs-all.

$h_\Theta(x) \in \mathbb{R}^4$

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

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

$y^{(i)}$ one of $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$
pedestrian car motorcycle truck

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

~~Previously~~ ~~$y \in \{1,2,3,4\}$~~

$h_\Theta(x^{(i)}) \approx y^{(i)}$ $\mathbb{R}^4$

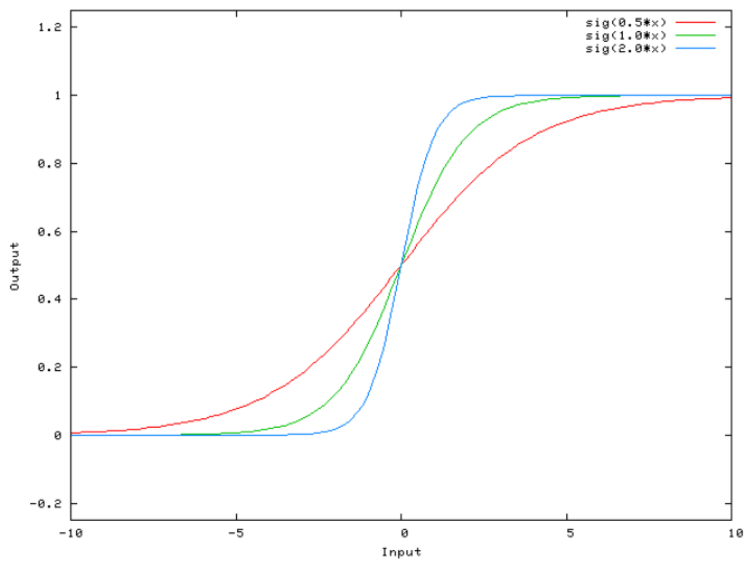
Andrew Ng

For multi class classification we use a method similar to the One vs All in logistic regression. We have one output unit for each class. This time the output is a vector.

Why we need a bias

A simple way to understand what the bias is: it is somehow similar to the constant b of a linear function y = ax + b. It allows you to move the line up and down to fit the prediction with the data better. Without b the line always goes through the origin (0, 0) and you may get a poorer fit.

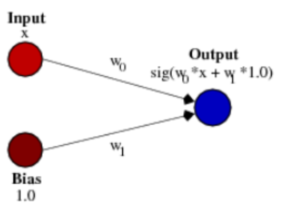
Consider this 1-input, 1-output network that has no bias



In effect, a bias value allows you to shift the activation function to the left or right, which may be critical for successful learning.

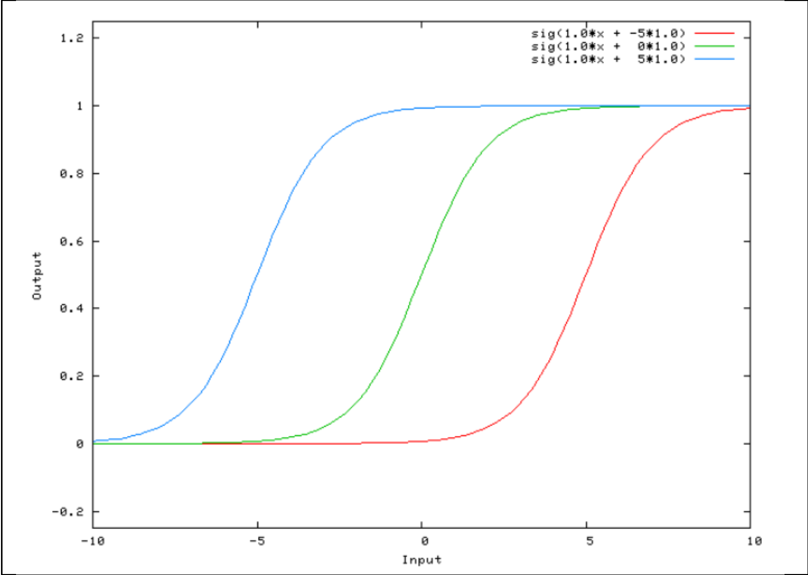
Changing the weight w0 essentially changes the "steepness" of the sigmoid. That's useful, but what if you wanted the network to output 0 when x is 2? Just changing the steepness of the sigmoid won't really work -- you want to be able to shift the entire curve to the right.

That's exactly what the bias allows you to do. If we add a bias to that network, like so:



Having a weight of -5 for w1 shifts the curve to the right, which allows us to have a network that outputs 0 when x is 2.

How it works: The bias (multiplied by its weight) is added to the weighted sum before it passes through the sigmoid function. If it is -10, then for the specific feature to be recognized (the sigmoid to be > 0.5) the



sum without the bias should be > 10 instead of simply greater than 0. So the sigmoid has been shifted to the right. **This kind of makes this specific feature to need a large degree of certainty in the input, for it to be activated. Another way to think of the bias is a number that makes the specific neuron to tend to be active or inactive.**

Sigmoid

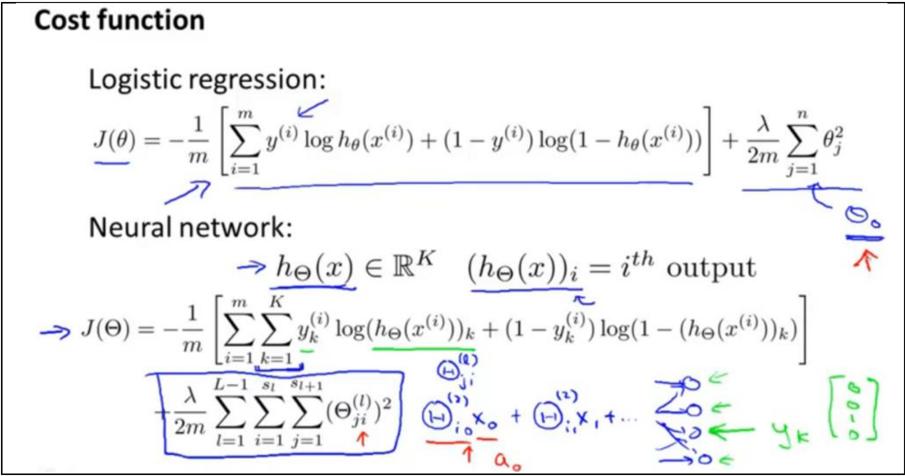
How positive is this?

$$\sigma(w_1a_1 + w_2a_2 + w_3a_3 + \cdots + w_na_n \boxed{-10})$$

"bias"

Cost function

The cost function is a generic form of the logistic regression cost function. The first term sums the cost of each output unit for each training data point. The regularization term just scales all the weights of the network except from the bias units since we don't want them to become 0. The reason that we added them in the first place was to contribute some non zero input to each layer.



Cost function

Logistic regression:

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

θ_0

Neural network:

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

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

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

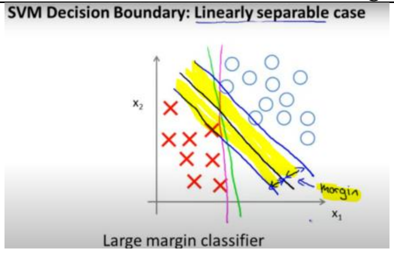
$\Theta_{ji}^{(l)}$ $\quad \Theta_{i0}^{(2)}x_0 + \Theta_{i1}^{(2)}x_1 + \ldots$ $\quad a_0$ $\quad y_k$

L: is the number of layers
sl: is the number of units (not counting the bias unit) in layer l.
k: is the number of output units

Support Vector Machines

SVM for linear boundaries

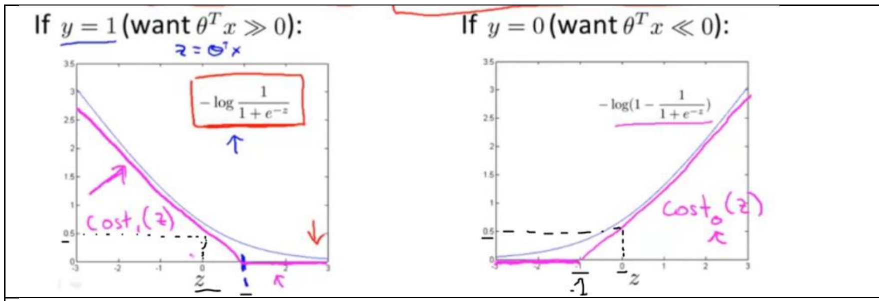
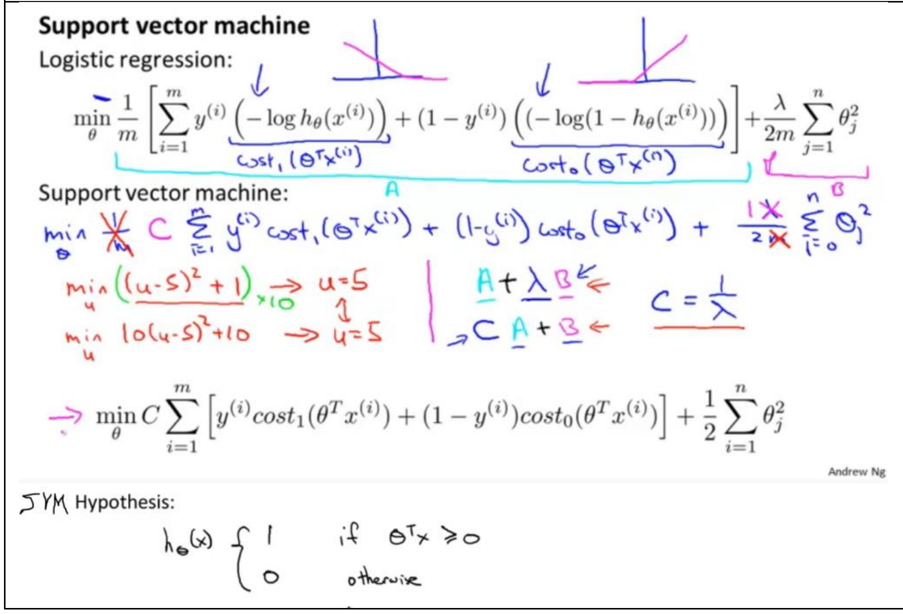
SVM for linear boundaries is a large margin classifier



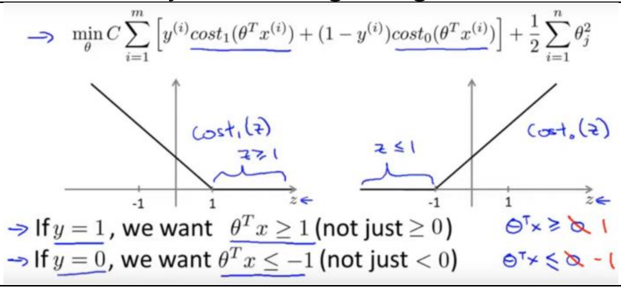
It is an other classification algorithm that offers some computational advantages and an easier optimization problem in relation to logistic regression. <u>It can model both linear and non linear decision boundaries</u>.

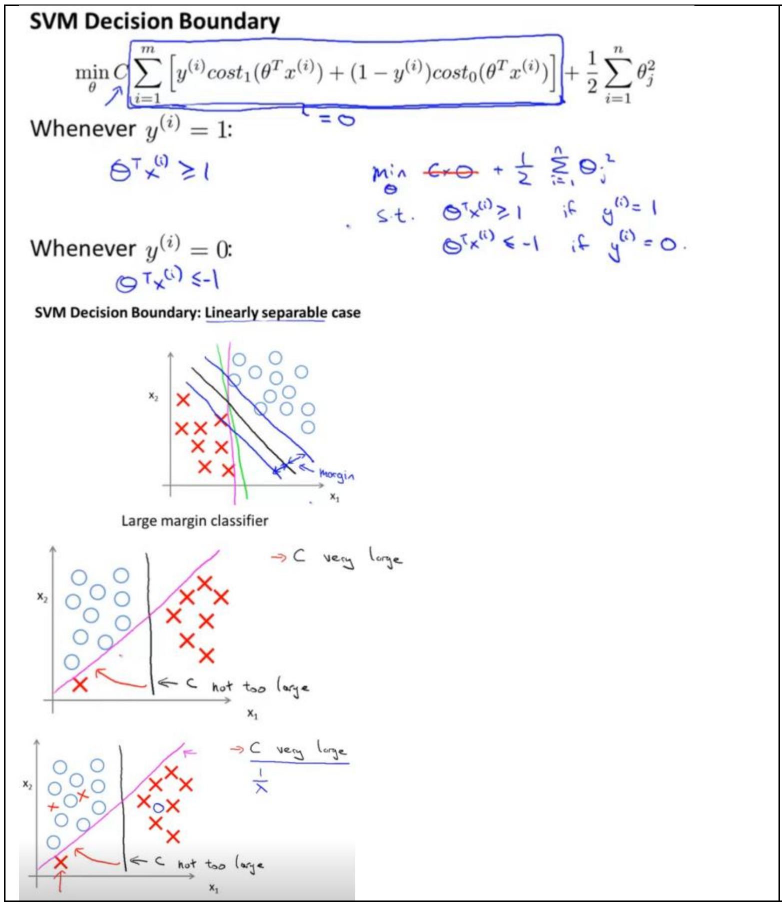
In case of linear boundaries the characteristic of SVM is that it produces a large margin boundary. In both cases there are mathematical tricks that formulate the problem in such a way that it is computationally efficient. The first one is the linear cost function. Another one for the linear case is the large value of C which gives the large margin.

For the non linear case we use the kernel method and a trick is the transformation of the regularization term of the cost function. In all cases the SVM optimization is a convex optimization problem (the cost function is convex) so a global minimum will always be found (as opposed to using a neural network)

	The pink lines are the SVM cost functions relative to z. They are a linear approximation of the logistic regression cost functions. This is what makes SVM more computationally efficient.
 **Support vector machine** Logistic regression: $\min_\theta \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \underbrace{\left(-\log h_\theta(x^{(i)})\right)}_{cost_1(\theta^T x^{(i)})} + (1-y^{(i)}) \underbrace{\left((-\log(1-h_\theta(x^{(i)}))\right)}_{cost_0(\theta^T x^{(n)})}\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$ Support vector machine: $\min_\theta \cancel{\frac{1}{m}}\, C \sum_{i=1}^{m} y^{(i)} cost_1(\theta^T x^{(i)}) + (1-y^{(i)}) cost_0(\theta^T x^{(i)}) + \frac{1\cancel{\lambda}}{2\cancel{m}} \sum_{i=0}^{n} \theta_j^2$ $\min_u \left((u-5)^2+1\right) \rightarrow u=5$ $\times 10$ $\quad$ $A + \lambda B \leftarrow$ $\min_u 10(u-5)^2+10 \rightarrow u=5$ $\quad$ $\rightarrow C A + B \leftarrow$ $\quad$ $C = \frac{1}{\lambda}$ $\rightarrow \min_\theta C \sum_{i=1}^{m}\left[y^{(i)} cost_1(\theta^T x^{(i)}) + (1-y^{(i)}) cost_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$ Andrew Ng SVM Hypothesis: $h_\theta(x) \begin{cases} 1 & \text{if } \theta^T x \geq 0 \\ 0 & \text{otherwise} \end{cases}$	The cost function is transformed to CA+B instead of the A+λB of the logistic regression where C=1/λ (it doesn't mean that if C=1/λ then the two expressions are equal. It means that the optimization of the two expressions will give the same optimum values) An other difference is that it produces 1 or 0. not like logistic regression which can give probability (for example 0.8)

The reason why SVM is a large margin classifier

 $\rightarrow \min_\theta C \sum_{i=1}^{m}\left[y^{(i)} \underline{cost_1(\theta^T x^{(i)})} + (1-y^{(i)}) \underline{cost_0(\theta^T x^{(i)})}\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$ $cost_1(z)$ $z \geq 1$ $z \leq 1$ $cost_0(z)$ -1 1 z → If $y = 1$, we want $\theta^T x \geq 1$ (not just ≥ 0) $\quad$ $\theta^T x \geq \cancel{0}\ 1$ → If $y = 0$, we want $\theta^T x \leq -1$ (not just < 0) $\quad$ $\theta^T x \leq \cancel{0}\ -1$	When C is very large (let's say 100000) which means that λ is very small so that regularization is small (and variance is large), the SVM model has an interesting property. It tries to separate the positive and negative examples with as big of a margin as possible. The big margin is a consequence of the minimization problem the objective function of which contains only the B term for large values of C.



SVM Decision Boundary

$$\min_\theta C \underbrace{\sum_{i=1}^{m}\left[y^{(i)} cost_1(\theta^T x^{(i)}) + (1-y^{(i)}) cost_0(\theta^T x^{(i)})\right]}_{=0} + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$

Whenever $y^{(i)} = 1$:

$\theta^T x^{(i)} \ge 1$

$$\min_\theta \cancel{C \times 0} + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$

s.t. $\theta^T x^{(i)} \ge 1$ if $y^{(i)} = 1$

$\theta^T x^{(i)} \le -1$ if $y^{(i)} = 0$.

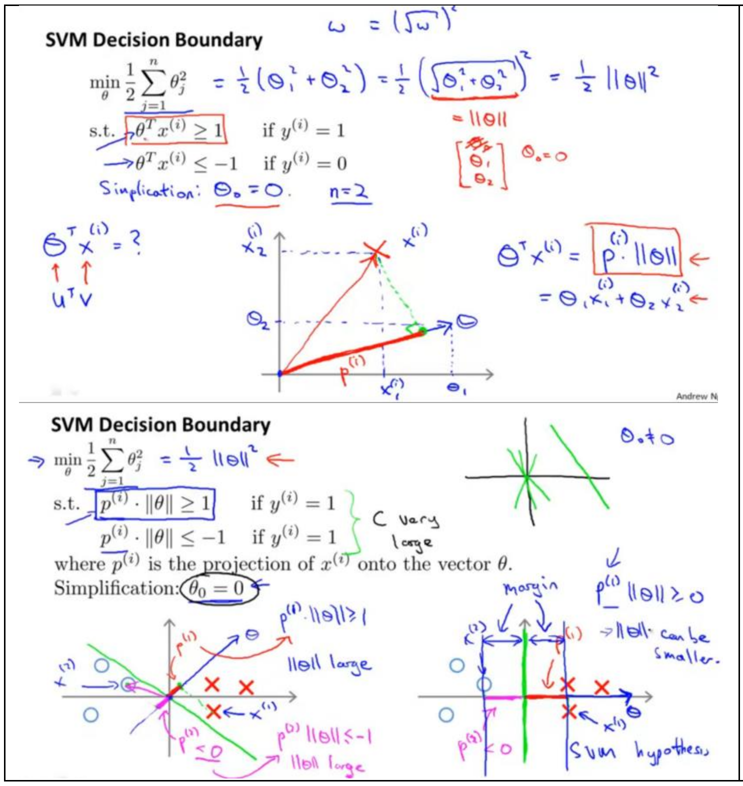
Whenever $y^{(i)} = 0$:

$\theta^T x^{(i)} \le -1$

SVM Decision Boundary: Linearly separable case

Large margin classifier

A large margin classifier is sensitive to outliers (because it has large variance so it has space for "overfitting"). The larger you choose the C to be, the more sensitive SVM will be to outliers and will produce the purple line instead of the black one, in a case like this.
If the C is a bit smaller (but still large so that you get this large margin effect) SVM can handle outliers like in the example, or can even be used for cases that are not 100% linearly separable but have some outliers within the opposite region.



SVM Decision Boundary

$\omega = (\sqrt{\omega})^2$

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

$\sqrt{\theta_1^2+\theta_2^2} = \|\theta\|$

s.t. $\boxed{\theta^T x^{(i)} \ge 1}$ if $y^{(i)} = 1$

$\theta^T x^{(i)} \le -1$ if $y^{(i)} = 0$

Simplication: $\theta_0 = 0$. $n=2$

$\begin{bmatrix}\theta_0\\ \theta_1\\ \theta_2\end{bmatrix}$ $\theta_0 = 0$

$\theta^T x^{(i)} = ?$

$u^T v$

$\theta^T x^{(i)} = \boxed{p^{(i)} \cdot \|\theta\|}$

$= \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)}$

SVM Decision Boundary

$$\min_\theta \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 = \frac{1}{2}\|\theta\|^2$$

s.t. $\boxed{p^{(i)} \cdot \|\theta\| \ge 1}$ if $y^{(i)} = 1$

$p^{(i)} \cdot \|\theta\| \le -1$ if $y^{(i)} = 1$

C very large

where $p^{(i)}$ is the projection of $x^{(i)}$ onto the vector θ.

Simplification: $\theta_0 = 0$

$\theta_0 \neq 0$

$p^{(1)} \cdot \|\theta\| \ge 1$
$\|\theta\|$ large
$p^{(1)}$
$x^{(1)}$
$x^{(2)}$
$p^{(2)} \le 0$
$p^{(2)}\|\theta\| \le -1$
$\|\theta\|$ large
margin
$p^{(1)}\|\theta\| \ge 0$
$\rightarrow \|\theta\|$ can be smaller.
$p^{(2)} < 0$
SVM hypothesis

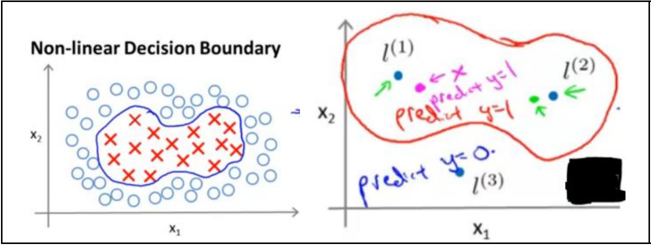
Here is the mathematical explanation of the reason for which the minimization problem leads to a large margin decision boundary when C is very large.

We transform the objective and constraints expressing them as vector norms and dot products. $\theta^T x$ is a dot product between the two vectors, which can be represented as the multiplication between the projection of x to θ and the norm of θ.

($\theta_0=0$ means that the decision boundary passes through the origin)

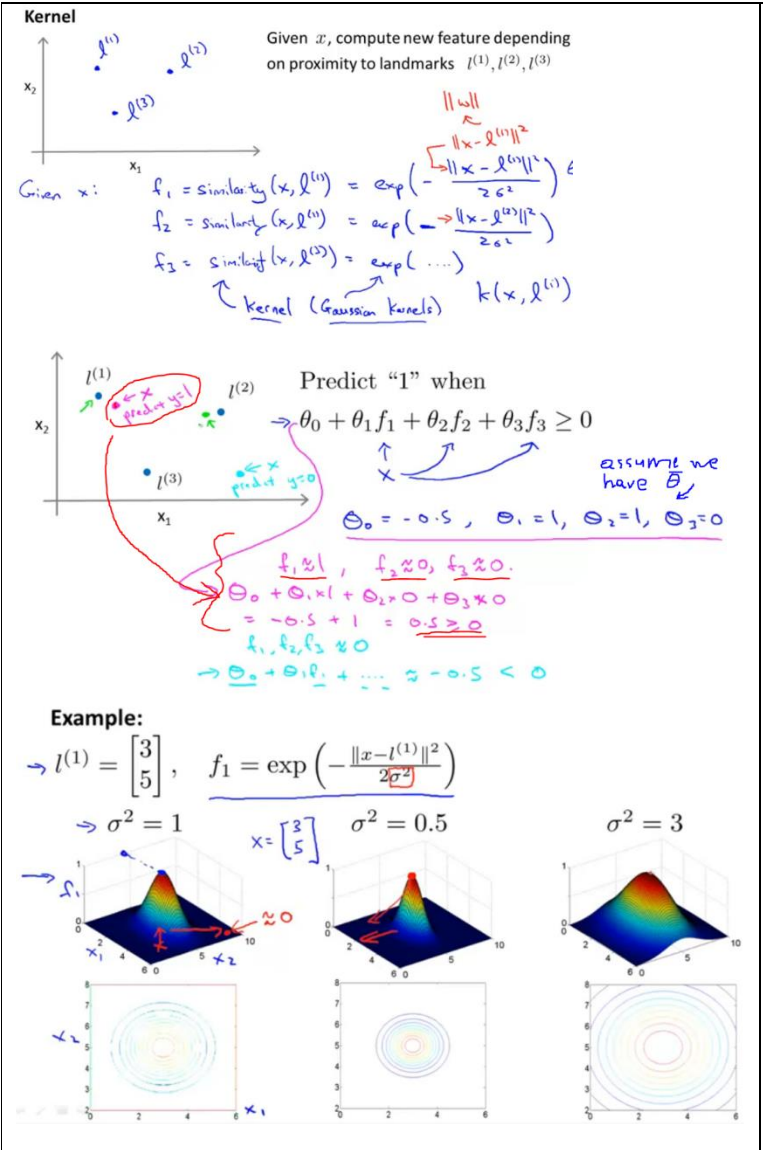
It can be proven that **the vector θ is orthogonal to the decision boundary**. Knowing that, we can measure the projections of vectors x to θ and conclude that a decision boundary with big margin gives larger projections in relation to one with small margin.
If the projections are small, then the norm (magnitude) of θ should be large so that p*θ>1. But the objective is to minimize norm of θ. So the SVM optimization will not produce such a solution. Instead it will produce a solution where the projection to the θ are large so that θ can take small values.

SVM for non linear boundaries



[redacted]

We use the **kernels method**. We select some specific points in the feature space which are called landmarks denoted by l, and the hypothesis learns to predict 1 for input points close to some of them and 0 for input points close to the rest of them. The result is the formation of highly non linear decision boundaries.


Kernel
Given x, compute new feature depending on proximity to landmarks $l^{(1)}, l^{(2)}, l^{(3)}$
$\|w\|$
Given x:
$f_1 = \text{similarity}(x, l^{(1)}) = \exp\left(-\frac{\|x-l^{(1)}\|^2}{2\sigma^2}\right)$
$f_2 = \text{similarity}(x, l^{(2)}) = \exp\left(-\frac{\|x-l^{(2)}\|^2}{2\sigma^2}\right)$
$f_3 = \text{similarity}(x, l^{(3)}) = \exp(\ldots)$
kernel (Gaussian kernels)
$k(x, l^{(i)})$
Predict "1" when
$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0$
assume we have θ
$\theta_0 = -0.5,\ \theta_1 = 1,\ \theta_2 = 1,\ \theta_3 = 0$
$f_1 \approx 1,\ f_2 \approx 0,\ f_3 \approx 0$
$\theta_0 + \theta_1 \times 1 + \theta_2 \times 0 + \theta_3 \times 0 = -0.5 + 1 = 0.5 \geq 0$
$f_1, f_2, f_3 \approx 0$
$\theta_0 + \theta_1 f_1 + \ldots \approx -0.5 < 0$
predict y=1
predict y=0
Example:
$l^{(1)} = \begin{bmatrix}3\\5\end{bmatrix}, \quad f_1 = \exp\left(-\frac{\|x-l^{(1)}\|^2}{2\sigma^2}\right)$
$\sigma^2 = 1$
$x = \begin{bmatrix}3\\5\end{bmatrix}$
$\sigma^2 = 0.5$
$\sigma^2 = 3$
f_1
≈ 0
x1
x2

First we select some **landmarks**

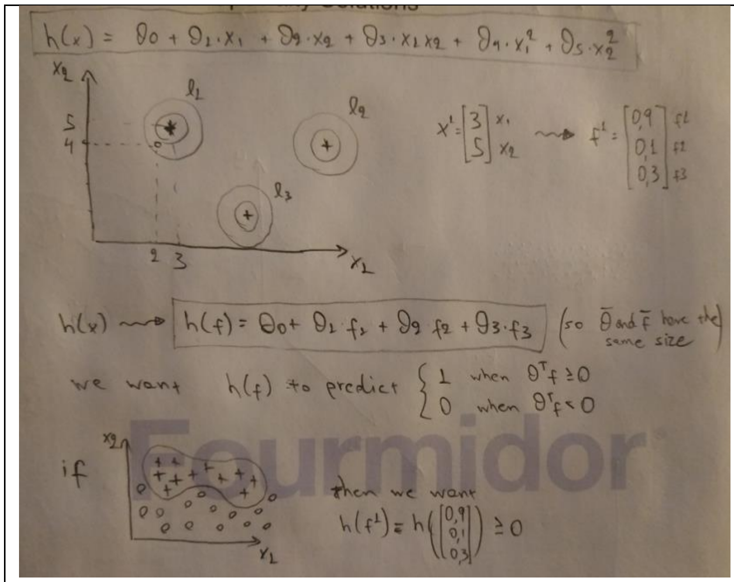
Then we compute new features denoted f, based on the proximity of the original inputs to the landmarks. This means that we transform each input vector x to a vector f the dimensionality of which is given by the number of landmarks, since each input vector x which can be represented by a point in the graph, has one proximity/similarity value for each landmark (so if the number of landmarks is smaller than the number of features then we are performing a dimensionality reduction). The hypothesis is formulated as a first order polynomial of the new features f (instead of a high order polynomial of original features x). The similarity of an input vector with a landmark is given by a specific function which is called a **kernel**. One type of kernel is the Gaussian kernel.

$e^{-\frac{|x-l^i|^2}{2\sigma^2}}$ where |x-l| is the magnitude of the distance between vector x and l that can be calculated by subtracting their components.

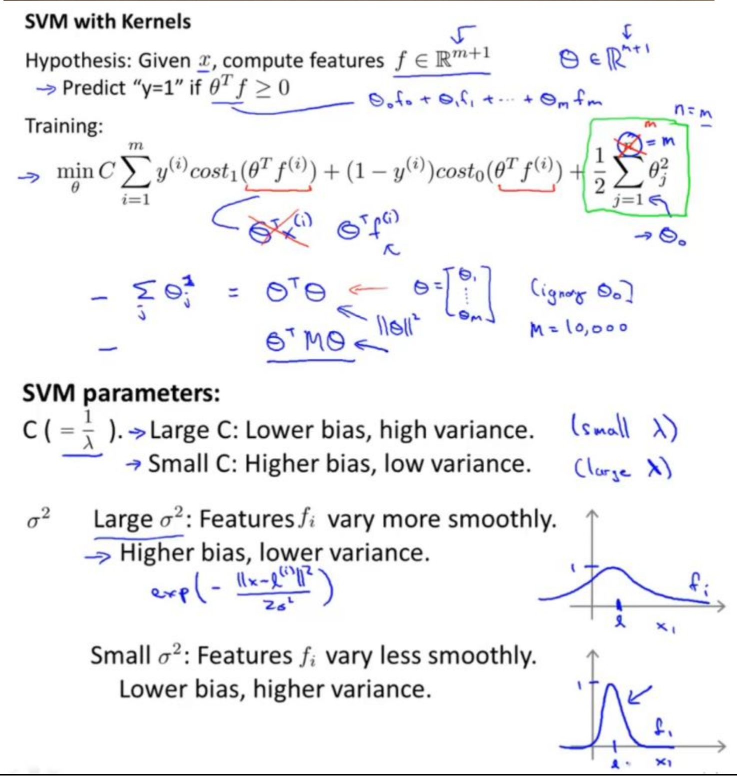
What the kernel does, is to produce a value close to 1 if a point is close to a landmark and close to 0 if it is far from it. Specifically, when an original feature x is close to a landmark the Gaussian kernel gives a value close to 1, so the feature that describes the similarity with that landmark would be close to 1. If an original feature x is far from a landmark then the feature value that describes its similarity with that landmark would be close to 0.

The σ coefficient defines the smoothness of the kernel or in other words the area around the landmark, within which a point gets a value close to 1. In the example of the graph, the landmark location is at x1=3, x2=5.

Selecting landmarks



$h(x) = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \theta_3 \cdot x_1 x_2 + \theta_4 \cdot x_1^2 + \theta_5 \cdot x_2^2$
x_2
5
4
l_1
l_2
l_3
2 3
x_1
$x^1 = \begin{bmatrix} 3 \\ 5 \end{bmatrix} \begin{matrix} x_1 \\ x_2 \end{matrix} \rightsquigarrow f^1 = \begin{bmatrix} 0,9 \\ 0,1 \\ 0,3 \end{bmatrix} \begin{matrix} f_1 \\ f_2 \\ f_3 \end{matrix}$
$h(x) \rightsquigarrow \boxed{h(f) = \theta_0 + \theta_1 \cdot f_1 + \theta_2 \cdot f_2 + \theta_3 \cdot f_3}$ (so $\bar{\theta}$ and $\bar{f}$ have the same size)
we want $h(f)$ to predict $\begin{cases} 1 & \text{when } \theta^T f \geq 0 \\ 0 & \text{when } \theta^T f < 0 \end{cases}$
if
x_2
x_1
then we want
$h(f^1) = h\left(\begin{bmatrix} 0,9 \\ 0,1 \\ 0,3 \end{bmatrix}\right) \geq 0$
Fourmidor



SVM with Kernels

Hypothesis: Given $\underline{x}$, compute features $\underline{f \in \mathbb{R}^{m+1}}$ $\quad \theta \in \mathbb{R}^{n+1}$

$\rightarrow$ Predict "y=1" if $\underline{\theta^T f} \geq 0$ $\quad \theta_0 f_0 + \theta_1 f_1 + \dots + \theta_m f_m$

Training: $\quad n = \underline{m}$

$$\rightarrow \min_\theta C \sum_{i=1}^{m} y^{(i)} cost_1(\underbrace{\theta^T f^{(i)}}) + (1 - y^{(i)}) cost_0(\underbrace{\theta^T f^{(i)}}) + \boxed{\frac{1}{2} \sum_{j=1}^{\cancel{n} = m} \theta_j^2}$$

$\cancel{\theta^T x^{(i)}} \quad \theta^T f^{(i)} \qquad \rightarrow \theta_0$

$- \sum_j \theta_j^2 = \theta^T \theta \leftarrow \theta = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_m \end{bmatrix}$ (ignore θ_0)

$\|\theta\|^2 \qquad M = 10,000$

$- \underline{\theta^T M \theta}$

SVM parameters:

C ($= \underline{\frac{1}{\lambda}}$). $\rightarrow$ Large C: Lower bias, high variance. (small λ)

$\rightarrow$ Small C: Higher bias, low variance. (large λ)

σ^2 $\quad \underline{\text{Large } \sigma^2}$: Features f_i vary more smoothly.

$\rightarrow$ Higher bias, lower variance.

$\exp\left(- \frac{\|x - l^{(i)}\|^2}{2\sigma^2}\right)$

Small σ^2: Features f_i vary less smoothly.

Lower bias, higher variance.

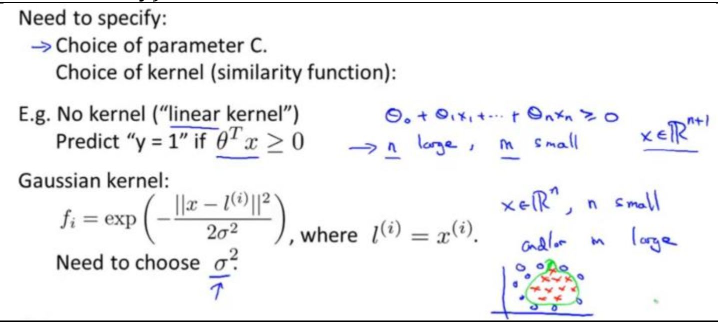
What we do in practice is to select as landmarks all the points of the training set. So the parameters θ will be of the same size with the training set and not larger, since in SVM with kernels we only use linear features (θ0+θ1f1+θ2f2+...θmfm). We don't use higher order polynomials for the hypothesis.

In this case we don't use a large value for C as we do for linear boundaries. A trick that makes the minimization of the cost function more efficient is to transform the last term as $\Theta^T M \Theta$ where M is a matrix that depends on the kernel we use.

You could apply kernels to logistic regression too, but the reason that we don't is that the computational tricks that make kernels in SVM run efficiently, don't apply to logistic regression.

Tips for running SVM

(with a library)



Need to specify:

$\rightarrow$ Choice of parameter C.

Choice of kernel (similarity function):

E.g. No kernel ("linear kernel") $\quad \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n \geq 0$

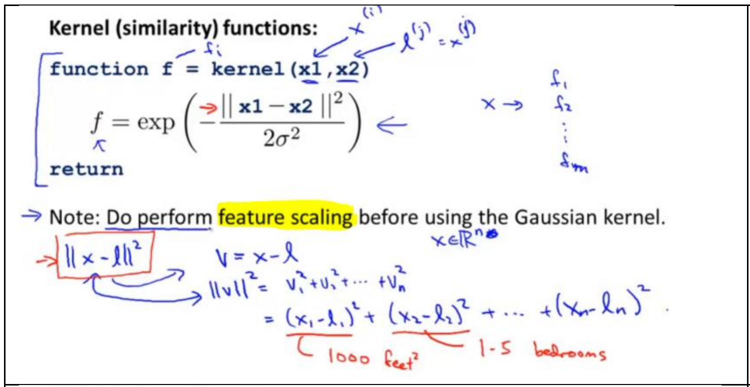
Predict "y = 1" if $\underline{\theta^T x} \geq 0$ $\quad \rightarrow \underline{n}$ large, $\underline{m}$ small $\quad \underline{x \in \mathbb{R}^{n+1}}$

Gaussian kernel:

$f_i = \exp\left(-\frac{\|x - l^{(i)}\|^2}{2\sigma^2}\right)$, where $l^{(i)} = x^{(i)}$. $\quad x \in \mathbb{R}^n$, n small and/or m large

Need to choose $\underline{\sigma^2}$.

One other commonly used kernel is the "**linear kernel**" which means that we don't use a kernel but instead we use a linear hypothesis function (first order polynomial) of the original inputs x. it is just a linear classifier (forming a linear decision boundary). y=1 if θTx>=0. this can be useful in cases which the number of features is large and the number of training examples is small, so if you use a high variance hypothesis you might risk overfitting.



Kernel (similarity) functions:

```
function f = kernel(x1,x2)

    f = exp(-||x1 - x2||^2 / (2σ^2))

return
```

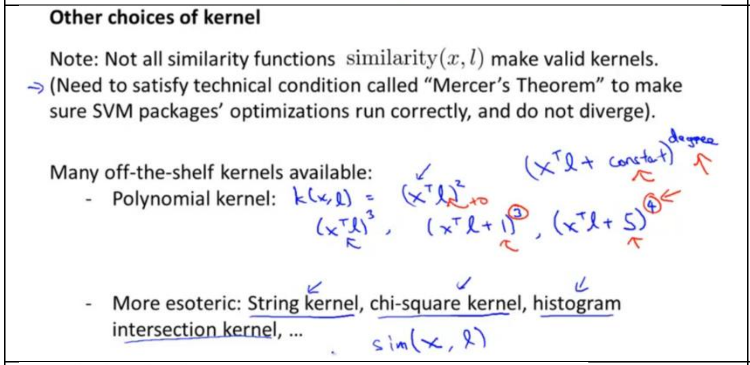
Note: Do perform feature scaling before using the Gaussian kernel.

$x^{(i)}$, $l^{(j)} = x^{(j)}$, f_i, $x \to f_1, f_2, \vdots, f_m$

$\boxed{\|x-l\|^2}$ $v = x - l$ $x \in \mathbb{R}^n$

$\|v\|^2 = v_1^2 + v_2^2 + \dots + v_n^2 = \underbrace{(x_1-l_1)^2}_{1000\ feet^2} + \underbrace{(x_2-l_2)^2}_{1\text{-}5\ bedrooms} + \dots + (x_n-l_n)^2$

Notice that we must do feature scaling before using the Gaussian kernel, so that the SVM gives the same weight to all features and not just to the one with larger values.



Other choices of kernel

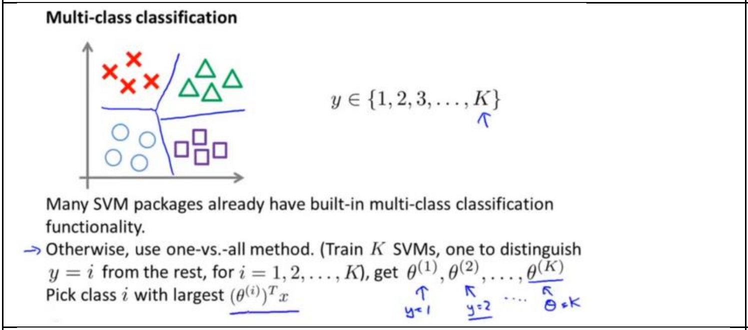
Note: Not all similarity functions $\text{similarity}(x,l)$ make valid kernels. (Need to satisfy technical condition called "Mercer's Theorem" to make sure SVM packages' optimizations run correctly, and do not diverge).

Many off-the-shelf kernels available:
- Polynomial kernel: $k(x,l) = (x^T l)^2$, $(x^T l)^3$, $(x^T l + 1)^3$, $(x^T l + 5)^4$, $(x^T l + \text{constant})^{\text{degree}}$
- More esoteric: String kernel, chi-square kernel, histogram intersection kernel, ... $\text{sim}(x,l)$

Mercer's Theorem
In almost all cases SVM use either Gaussian or linear kernels.

A kernel must satisfy this condition so that it allows us to use a large class of optimization to solve efficiently.

String kernel: Similarity between two strings



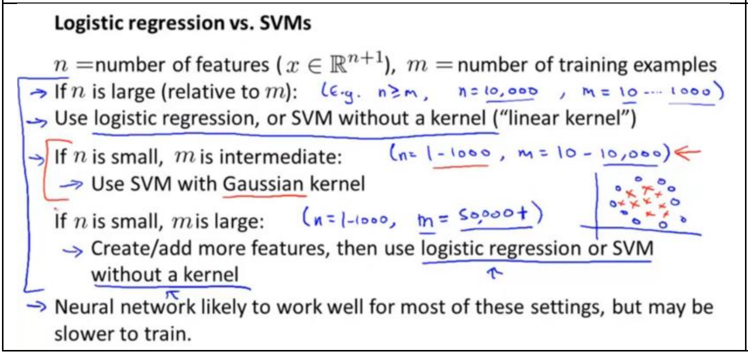
Multi-class classification

$y \in \{1,2,3,\dots,K\}$

Many SVM packages already have built-in multi-class classification functionality.
Otherwise, use one-vs.-all method. (Train K SVMs, one to distinguish $y=i$ from the rest, for $i=1,2,\dots,K$), get $\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(K)}$
Pick class i with largest $(\theta^{(i)})^T x$

$y=1$ $y=2$ … θ_K

Multi class classification



Logistic regression vs. SVMs

$n =$ number of features ($x \in \mathbb{R}^{n+1}$), $m =$ number of training examples
- If n is large (relative to m): (E.g. $n \geq m$, $n = 10{,}000$, $m = 10 \dots 1000$)
  - Use logistic regression, or SVM without a kernel ("linear kernel")
- If n is small, m is intermediate: ($n = 1\text{-}1000$, $m = 10 - 10{,}000$)
  - Use SVM with Gaussian kernel
- If n is small, m is large: ($n = 1\text{-}1000$, $m = 50{,}000+$)
  - Create/add more features, then use logistic regression or SVM without a kernel
- Neural network likely to work well for most of these settings, but may be slower to train.

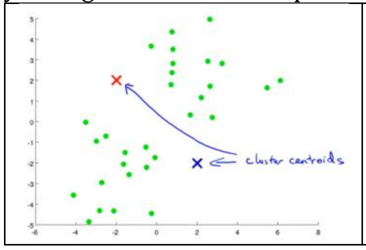
When m is huge SVM with Gaussian kernel might be slow.

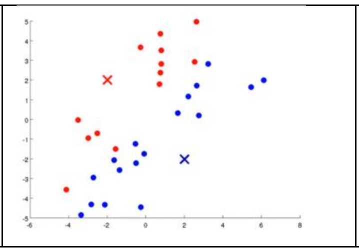
In general algorithms are important but what is more important you have and how skilled you are in error analysis and debugging of your algorithm, to designing new features etc.

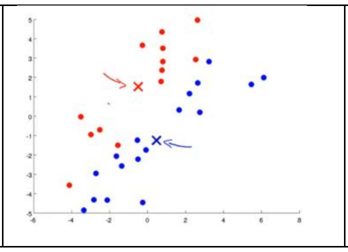
---> Unsupervised Learning <---

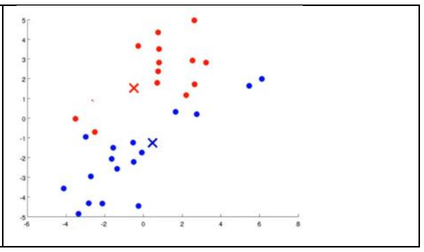
K-means

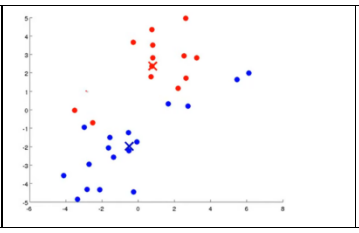
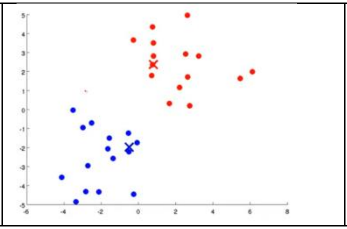
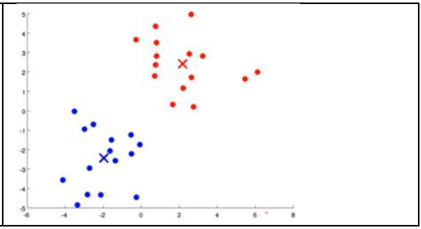
It is an unsupervised learning algorithm.
It is an iterative algorithm with two internal steps. The first one is the **cluster assignment** and the second one the **move centroid** step. It iteratively repeats these steps until convergence when no or only a few inputs change cluster after a move centroid step. It turns out that you can define a minimization problem for K means but the objective function is not convex so you might stuck to local optima.

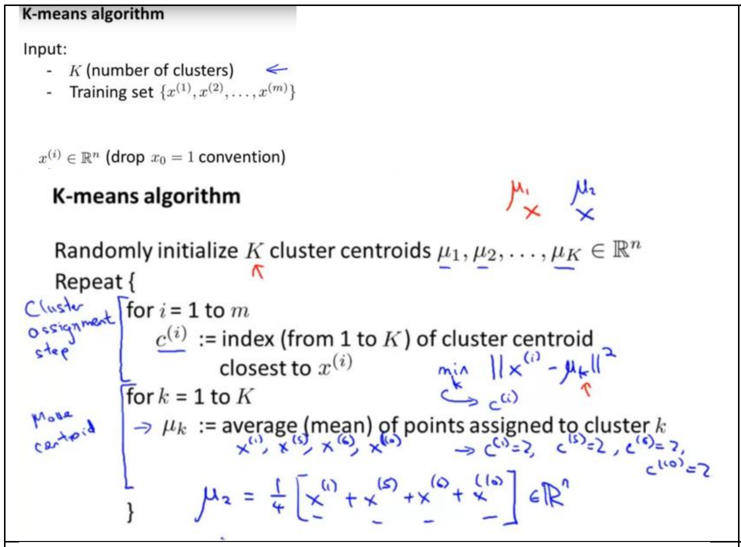










K-means algorithm

Input:
- K (number of clusters)
- Training set $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$

$x^{(i)} \in \mathbb{R}^n$ (drop $x_0 = 1$ convention)

K-means algorithm

$\mu_1 \quad \mu_2$

```
Randomly initialize $K$ cluster centroids $\mu_1, \mu_2, \ldots, \mu_K \in \mathbb{R}^n$
Repeat {
    for $i$ = 1 to $m$
        $c^{(i)}$ := index (from 1 to $K$) of cluster centroid closest to $x^{(i)}$
    for $k$ = 1 to $K$
        $\mu_k$ := average (mean) of points assigned to cluster $k$
}
```

Cluster assignment step

$\min_k ||x^{(i)} - \mu_k||^2 \rightarrow c^{(i)}$

Move centroid

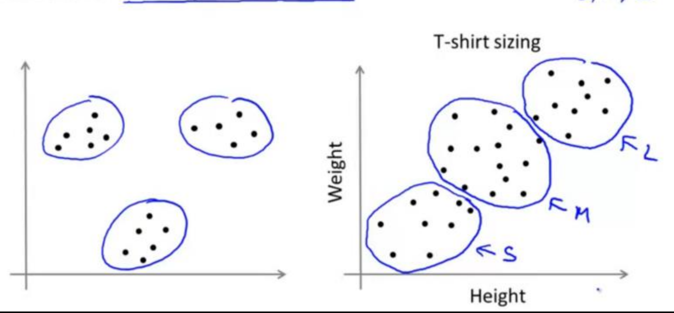
$x^{(1)}, x^{(5)}, x^{(6)}, x^{(10)} \rightarrow c^{(1)}=2, c^{(5)}=2, c^{(6)}=2, c^{(10)}=2$

$$\mu_2 = \frac{1}{4}\left[x^{(1)} + x^{(5)} + x^{(6)} + x^{(10)}\right] \in \mathbb{R}^n$$

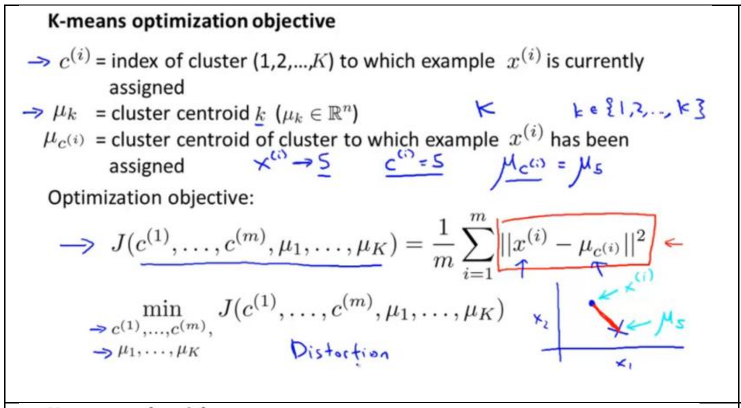
Random initialization of centroids the number of which is equal to the number of clusters that we want to identify.
Each data point is assigned to a centroid based on which one is closest to.
The centroids are moved to the average position of their assigned examples and cluster assignment is repeated.

If a centroid has zero inputs associated with it then we either completely remove it from the centroids or randomly reinitialize it.

K-means for non-separated clusters



K means can also work for non separated clusters.



K-means optimization objective

$\rightarrow c^{(i)}$ = index of cluster (1,2,...,K) to which example $x^{(i)}$ is currently assigned

$\rightarrow \mu_k$ = cluster centroid k ($\mu_k \in \mathbb{R}^n$) $\quad K \quad k \in \{1,2,\ldots,K\}$

$\mu_{c^{(i)}}$ = cluster centroid of cluster to which example $x^{(i)}$ has been assigned $\quad x^{(i)} \rightarrow 5 \quad c^{(i)} = 5 \quad \mu_{c^{(i)}} = \mu_5$

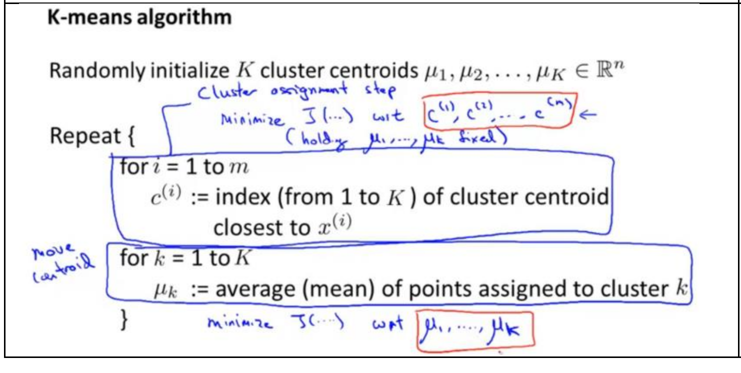
Optimization objective:

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

$$\min_{\substack{c^{(1)},\ldots,c^{(m)},\\ \mu_1,\ldots,\mu_K}} J(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K)$$

Distortion

It turns out that you can define a minimization problem for K-means so that the process will stop when the optimum values c and μ are found. The cost function measures the average distance of each data point from its cluster centroid and is called the distortion of the k-means algorithm.



K-means algorithm

```
Randomly initialize $K$ cluster centroids $\mu_1, \mu_2, \ldots, \mu_K \in \mathbb{R}^n$
Repeat {
    for $i$ = 1 to $m$
        $c^{(i)}$ := index (from 1 to $K$) of cluster centroid closest to $x^{(i)}$
    for $k$ = 1 to $K$
        $\mu_k$ := average (mean) of points assigned to cluster $k$
}
```

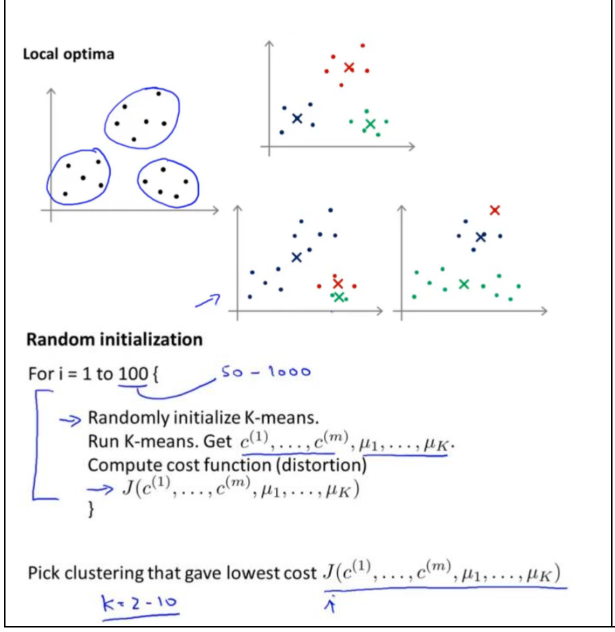
Cluster assignment step: minimize $J(\ldots)$ wrt $\boxed{c^{(1)}, c^{(2)}, \ldots, c^{(m)}} \leftarrow$ (holding $\mu_1, \ldots, \mu_K$ fixed)

move centroid: minimize $J(\ldots)$ wrt $\boxed{\mu_1, \ldots, \mu_K}$

The cluster assignment step which assigns each point to the centroid that is closest to it, actually minimizes the distortion (the cost function) with respect to c holding μ fixed, since it minimizes the distance of the points from their assigned centroid.
While the move centroid step minimizes the cost with respect to the centroids, since by moving to the average position minimizes the average distance from it.

So the minimization is done in two steps.

Local and global optima

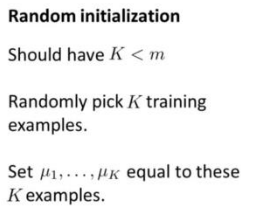


Random initialization

```
For i = 1 to 100 {
    Randomly initialize K-means.
    Run K-means. Get $c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K$.
    Compute cost function (distortion)
    $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$
}
```

Pick clustering that gave lowest cost $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$

The objective function is **not convex** so you might stuck to local optima. The solution which will be found, depends on the selection of the initial centroids.
A typical way to initialize them is to randomly pick k data points and define the centroids there.



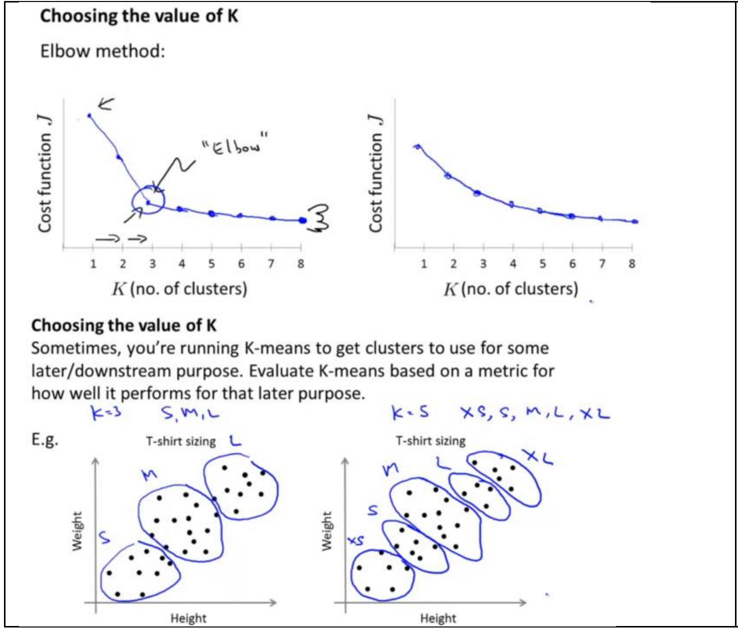
Random initialization

Should have $K < m$

Randomly pick K training examples.

Set $\mu_1, \dots, \mu_K$ equal to these K examples.

For k between 2-10 there are high chances of being locked in local optima so the solution is to run k means with a lot of different initialization values (50-1000 different runs) record the final cost function for each one and pick the one with the lowest.
Notice that if k is very large, larger than 10 for example, then there are very low chances of being locked in a local optimum so you can avoid the multiple initialization approach.



Choosing the value of K

Sometimes, you're running K-means to get clusters to use for some later/downstream purpose. Evaluate K-means based on a metric for how well it performs for that later purpose.

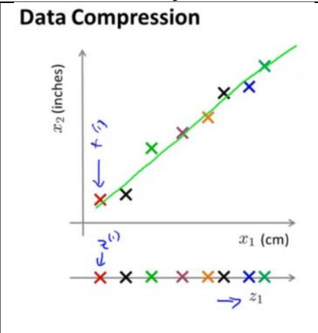
Choosing the number of clusters is mainly a manual process. There is a method called the elbow method but it is not always applicable.

One other common way is to choose it based on a business need and then evaluate the selection based on the performance on the actual business.

Principal Components Analysis

Principal Components Analysis (PCA) is the most common algorithm used for dimensionality reduction (reduction to a linear supspace)

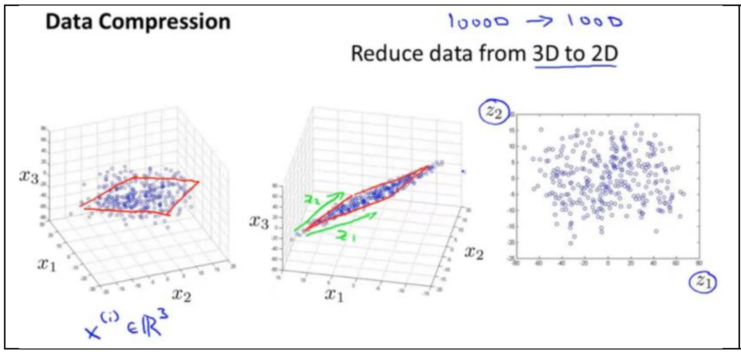
Dimensionality reduction



When you have highly correlated features (for example the length in cm and in inches) then you can combine these features into one. The line in the example is not a perfect line due to round off errors.

You do this by projecting the points in a line or a surface or an equivalent lower dimension object.

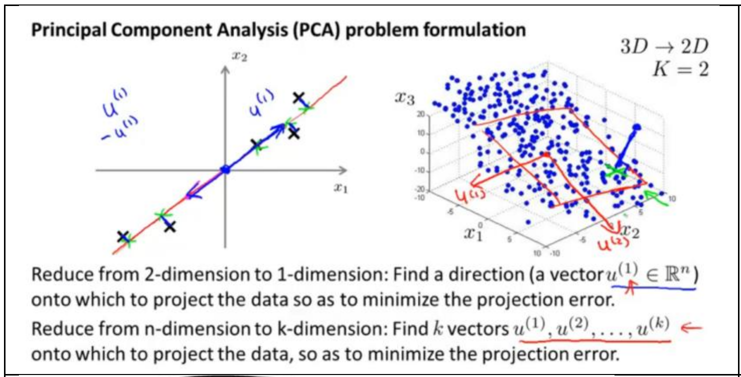
For example if you have 50 features describing a country's state (GDP, mortality, literacy etc.) you could maybe decrease the number of features to two, one that describes



the state of the country as a whole and one that describes the state of the country per capita.

The most used algorithm for dimensionality reduction is Principal Components Analysis (PCA).

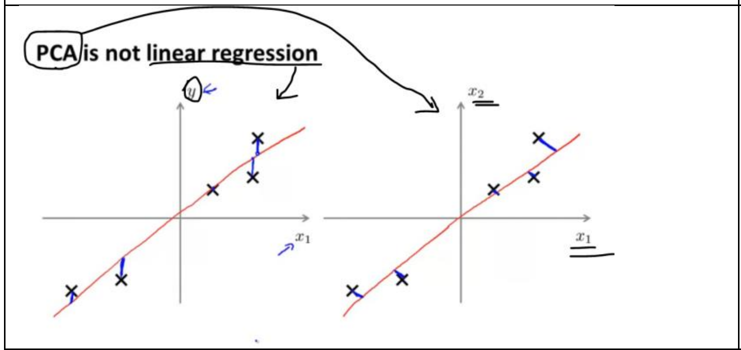
Principal Components Analysis
It tries to find a lower dimension surface (of lower dimensions in relation to the original feature space) onto which to project the data. This surface is found by minimizing the square of the projection errors (the distance that the data points must be moved). In other words, we want to get the original data set which is an n dimensional object (x->R^n) and find a lower dimension representation of it (z >R^k). The number k is the number of principal components that we retain. Actually what PCA does is trying to compresses the data by keeping as much information from them as possible. You can also go from the compressed data back to an approximation of the original data (reconstruction from compressed representation).


Principal Component Analysis (PCA) problem formulation

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

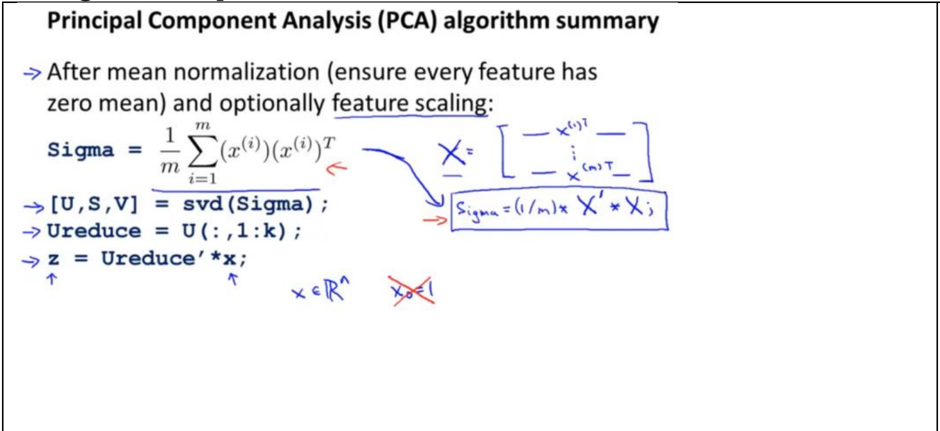
In case of a reduction from n to k dimensions, we want to find k n-dimensional vectors (which define a k dimensional space) and project the data onto the linear subspace spanned by these vectors.

Have in mind that you need to perform Mean normalization and scaling before applying PCA.



Notice the difference between PCA and linear regression. PCA tries to minimize the projection distances (called projections errors in the context of PCA) while linear regression tries to minimize the prediction error which is a different distance. This means that the optimum solution would be different for each algorithm.

PCA algorithm implementation


Principal Component Analysis (PCA) algorithm summary

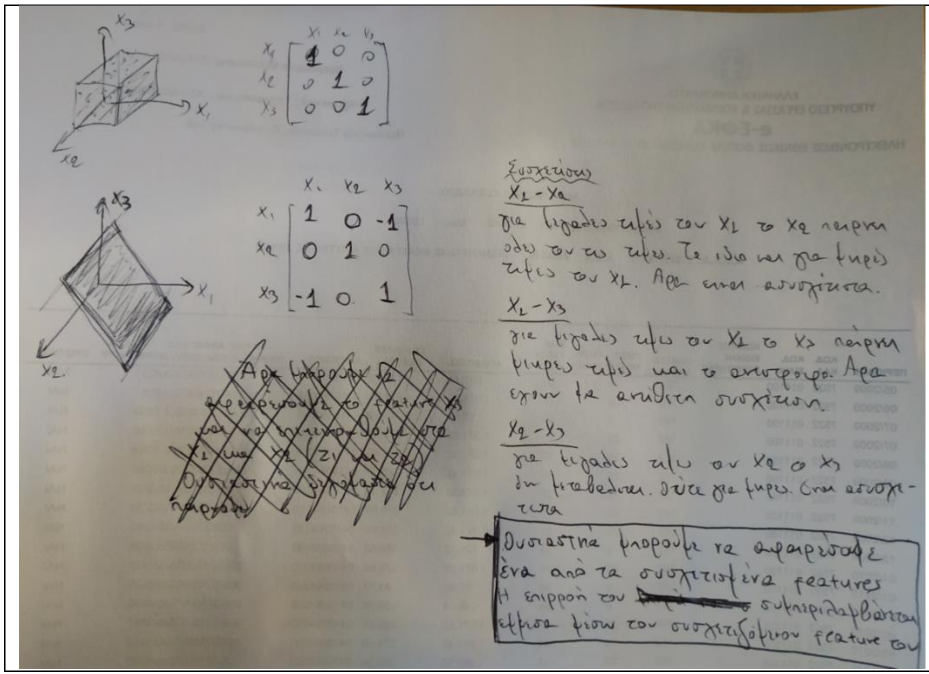
→ After mean normalization (ensure every feature has zero mean) and optionally feature scaling:

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

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

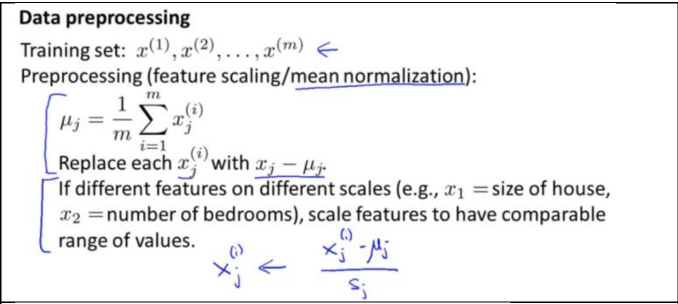
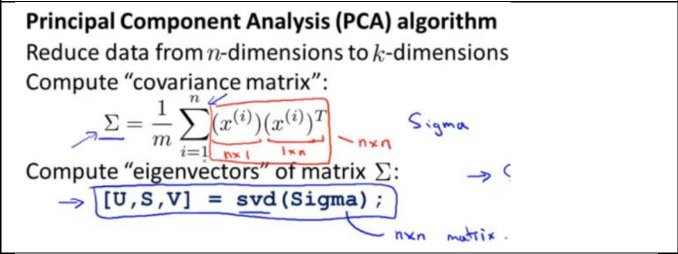
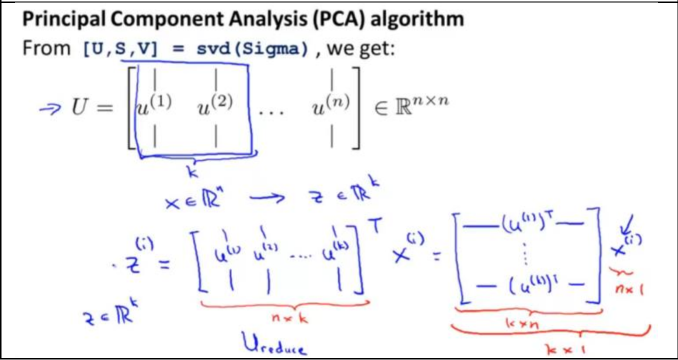
$X = \begin{bmatrix} — x^{(1)T} — \\ \vdots \\ — x^{(m)T} — \end{bmatrix}$
Sigma = (1/m) * X' * X;
$x \in \mathbb{R}^n$
$\cancel{x_0 = 1}$

Sigma is the covariance matrix (multiplying features with each other, see below for intuition).
Then we calculate the eigenvectors of this matrix which will be as many as the dimension of this orthogonal matrix, which is the dimensions of the original feature space. If we want to represent this object with a k dimensional space we select the first k eigenvectors (the first k columns)
The new input vectors z are calculated by multiplying the selected eigenvectors matrix with the original feature vectors.

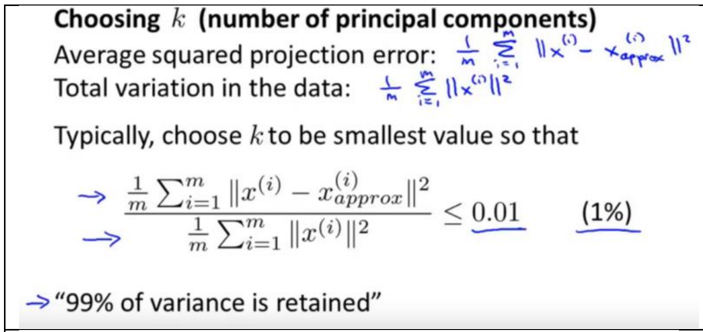
 $$\begin{array}{c|ccc} & x_1 & x_2 & x_3 \\ \hline x_1 & 1 & 0 & 0 \\ x_2 & 0 & 1 & 0 \\ x_3 & 0 & 0 & 1 \end{array}$$ $$\begin{array}{c|ccc} & x_1 & x_2 & x_3 \\ \hline x_1 & 1 & 0 & -1 \\ x_2 & 0 & 1 & 0 \\ x_3 & -1 & 0 & 1 \end{array}$$ ~~[illegible]~~ Συσχετίσεις $x_1 - x_2$ για μεγάλες τιμές του x_1 το x_2 παίρνει όλες τις τιμές. Το ίδιο και για μικρές τιμές του x_1. Άρα είναι ασυσχέτιστα. $x_1 - x_3$ για μεγάλες τιμές του x_1 το x_3 παίρνει μικρές τιμές και το αντίστροφο. Άρα έχουν μια αντίθετη συσχέτιση. $x_2 - x_3$ για μεγάλες τιμές του x_2 το x_3 δεν επηρεάζεται. Ίδια για μικρές. Είναι ασυσχέτιστα Ουσιαστικά μπορούμε να αφαιρέσουμε ένα από τα συσχετισμένα features. Η επιρροή του ~~[illegible]~~ συμπεριλαμβάνεται έμμεσα μέσω του συσχετισμένου feature του	

The mathematical proof of why this process results in the k surface that minimizes the projection errors is not presented.

In more detail

 **Data preprocessing** Training set: $x^{(1)}, x^{(2)}, \ldots, x^{(m)} \leftarrow$ Preprocessing (feature scaling/mean normalization): $\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$ Replace each $x_j^{(i)}$ with $x_j - \mu_j$. If different features on different scales (e.g., $x_1 =$ size of house, $x_2 =$ number of bedrooms), scale features to have comparable range of values. $x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{s_j}$	
 **Principal Component Analysis (PCA) algorithm** Reduce data from n-dimensions to k-dimensions Compute "covariance matrix": $\rightarrow \Sigma = \frac{1}{m}\sum_{i=1}^{n} \underbrace{\boxed{\underbrace{(x^{(i)})}_{n\times 1}\underbrace{(x^{(i)})^T}_{1\times n}}}_{n\times n}$ Sigma Compute "eigenvectors" of matrix Σ: $\rightarrow$ `[U,S,V] = svd(Sigma);` $n\times n$ matrix.	The covariance matrix always satisfies a mathematical property called symmetric positive semi definite, so the svd (singular value decomposition) and eig octave methods give the same result, the same eigenvectors. The svd is numerically more stable than the eig method.
 **Principal Component Analysis (PCA) algorithm** From `[U,S,V] = svd(Sigma)`, we get: $\rightarrow U = \left[\begin{array}{cccc} \vert & \vert & & \vert \\ u^{(1)} & u^{(2)} & \ldots & u^{(n)} \\ \vert & \vert & & \vert \end{array}\right] \in \mathbb{R}^{n\times n}$ k $x \in \mathbb{R}^n \longrightarrow z \in \mathbb{R}^k$ $z^{(i)} = \underbrace{\underbrace{\left[\begin{array}{cccc} \vert & \vert & & \vert \\ u^{(1)} & u^{(2)} & \ldots & u^{(k)} \\ \vert & \vert & & \vert \end{array}\right]}_{\substack{n\times k \\ U_{reduce}}}^T x^{(i)} = \underbrace{\underbrace{\left[\begin{array}{c} — (u^{(1)})^T — \\ \vdots \\ — (u^{(k)})^T — \end{array}\right]}_{k\times n} \underbrace{x^{(i)}}_{n\times 1}}_{k\times 1}$ $z \in \mathbb{R}^k$	

Choosing k



Choosing k (number of principal components)

Average squared projection error: $\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}-x^{(i)}_{approx}\|^2$

Total variation in the data: $\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2$

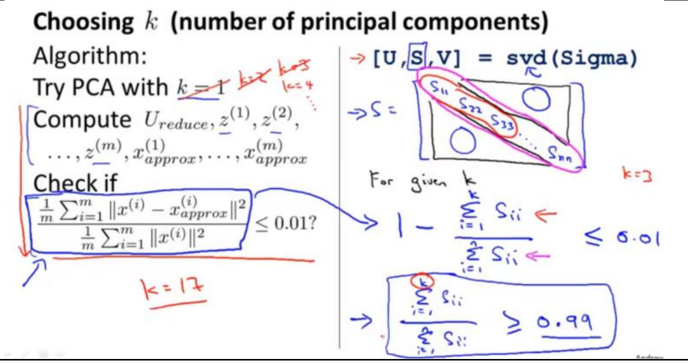
Typically, choose k to be smallest value so that

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}-x^{(i)}_{approx}\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2} \le 0.01 \qquad (1\%)$$

"99% of variance is retained"

A way to get an insight for it is to think the 3d to 2d reduction. If the points are more or less upon a plane then the projections would be small and the ratio would be small. But if there are scattered all over the 3d space then the projections would be large and so will be the ratio.

So you choose the least amount of principal components so that 99% of variance is retained. So you can compress the data by a large factor by keeping a very high percentage of its variance.



Choosing k (number of principal components)

Algorithm:

Try PCA with ~~$k=1$~~ $k=2$ $k=3$ $k=4$

Compute $U_{reduce}, z^{(1)}, z^{(2)}, \ldots, z^{(m)}, x^{(1)}_{approx}, \ldots, x^{(m)}_{approx}$

Check if

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}-x^{(i)}_{approx}\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2} \le 0.01?$$

$k=17$

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

$$S = \begin{bmatrix} S_{11} & & & \\ & S_{22} & & \\ & & S_{33} & \\ & & & \ddots \\ & & & & S_{nn} \end{bmatrix}$$

$k=3$

For given k

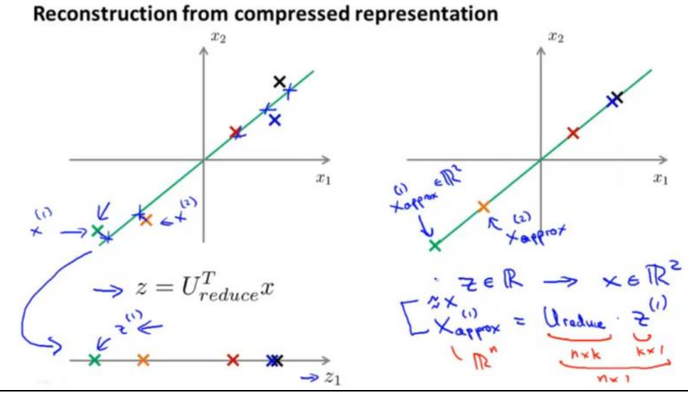
$$1 - \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \le 0.01$$

$$\boxed{\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \ge 0.99}$$

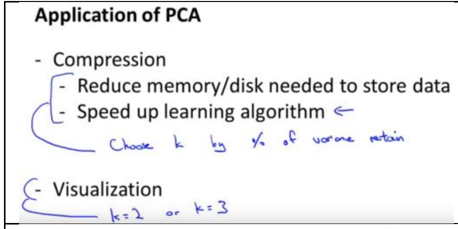
<u>You don't have to do the first option of iterating through k and running PCA for each one</u>. It turns out that the ratio can be calculated from the S matrix which is diagonal. So you run PCA once and then just iterate k to compute different ratios and find the one that you want.

95 to 99% retained variance is a commonly used range and it is usually retained by a 5-10 times fewer dimensions.

Reconstruction from compressed representation



Good and bad use of PCA



Application of PCA

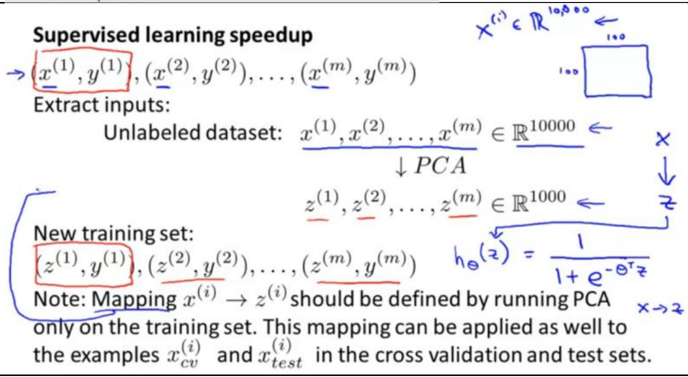
- Compression
  - Reduce memory/disk needed to store data
  - Speed up learning algorithm

  Choose k by % of variance retain

- Visualization

  k=2 or k=3

PCA is used for compression or visualization purposes (to make some reduction to 2d or 3d so that you can visualize something useful)



Supervised learning speedup

$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$ $\qquad x^{(i)} \in \mathbb{R}^{10000}$ $\quad 100 \times 100$

Extract inputs:

Unlabeled dataset: $x^{(1)}, x^{(2)}, \ldots, x^{(m)} \in \mathbb{R}^{10000}$

$\downarrow PCA$

$z^{(1)}, z^{(2)}, \ldots, z^{(m)} \in \mathbb{R}^{1000}$

$x \rightarrow z$

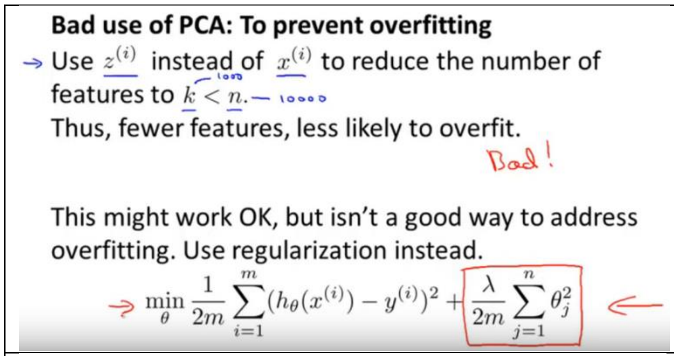
New training set:

$(z^{(1)}, y^{(1)}), (z^{(2)}, y^{(2)}), \ldots, (z^{(m)}, y^{(m)})$ $\qquad h_\theta(z) = \frac{1}{1+e^{-\theta^T z}}$

Note: Mapping $x^{(i)} \rightarrow z^{(i)}$ should be defined by running PCA only on the training set. This mapping can be applied as well to the examples $x^{(i)}_{cv}$ and $x^{(i)}_{test}$ in the cross validation and test sets. $\quad x \rightarrow z$

Typical application in supervised learning.

Notice that we only use the training set for finding Ureduce.



Bad use of PCA: To prevent overfitting

→ Use $z^{(i)}$ instead of $x^{(i)}$ to reduce the number of features to $\underset{1000}{k} < \underset{10000}{n}$.

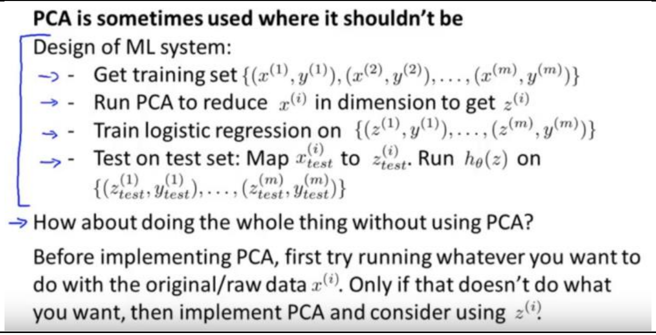
Thus, fewer features, less likely to overfit. Bad!

This might work OK, but isn't a good way to address overfitting. Use regularization instead.

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

But PCA must not be used without reason since **despite the fact that it can retain a high variance of the data, it doesn't take into consideration the labels** (the y values) and this means that you might lose some valuable information.

Don't use it for overfitting, use regularization instead.



PCA is sometimes used where it shouldn't be

Design of ML system:

→ - Get training set $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

→ - Run PCA to reduce $x^{(i)}$ in dimension to get $z^{(i)}$

→ - Train logistic regression on $\{(z^{(1)}, y^{(1)}), \ldots, (z^{(m)}, y^{(m)})\}$

→ - Test on test set: Map $x_{test}^{(i)}$ to $z_{test}^{(i)}$. Run $h_\theta(z)$ on $\{(z_{test}^{(1)}, y_{test}^{(1)}), \ldots, (z_{test}^{(m)}, y_{test}^{(m)})\}$

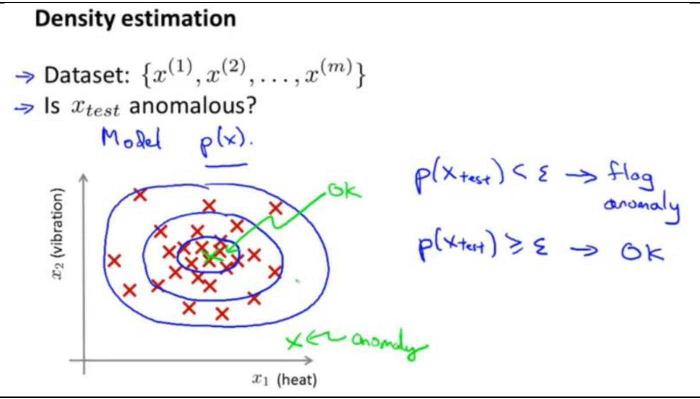
→ How about doing the whole thing without using PCA?

Before implementing PCA, first try running whatever you want to do with the original/raw data $x^{(i)}$. Only if that doesn't do what you want, then implement PCA and consider using $z^{(i)}$.

Don't use it if you can do it with the original features.

Anomaly detection

Gaussian distribution



Density estimation

→ Dataset: $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$

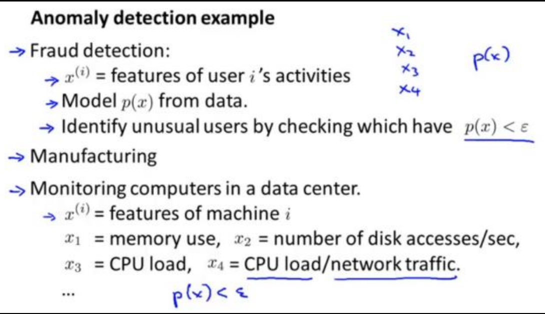
→ Is x_{test} anomalous?

Model $p(x)$.

$p(x_{test}) < \varepsilon \rightarrow$ flag anomaly

$p(x_{test}) \geq \varepsilon \rightarrow$ OK

We generally consider that the examples of the dataset are non-anomalous. We want an algorithm to tell us if a new example xtest is anomalous. We do it like this: Given this unlabeled training set we will build a model p(x) that gives the probability of x. then if a specific xtest has very low probability lower than a threshold, then we say that it is an anomaly.
The problem of estimating p(x) is called the density estimation problem.



Anomaly detection example

→ Fraud detection:

→ $x^{(i)}$ = features of user i's activities

→ Model $p(x)$ from data.

→ Identify unusual users by checking which have $p(x) < \varepsilon$

x_1 x_2 x_3 x_4 $p(x)$

→ Manufacturing

→ Monitoring computers in a data center.

→ $x^{(i)}$ = features of machine i

x_1 = memory use, x_2 = number of disk accesses/sec,

x_3 = CPU load, x_4 = CPU load/network traffic.

...

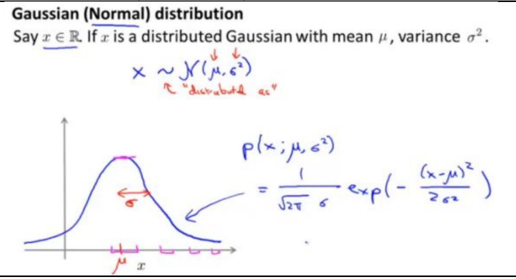
$p(x) < \varepsilon$

Applications of anomaly detection

Some possible features used for fraud detection:

1. How often a user logs in
2. The number of pages he visits per session
3. The number of transactions per day
4. The number of posts of the user in the forum
5. The typing speed of the user

The Normal (Gaussian) distribution

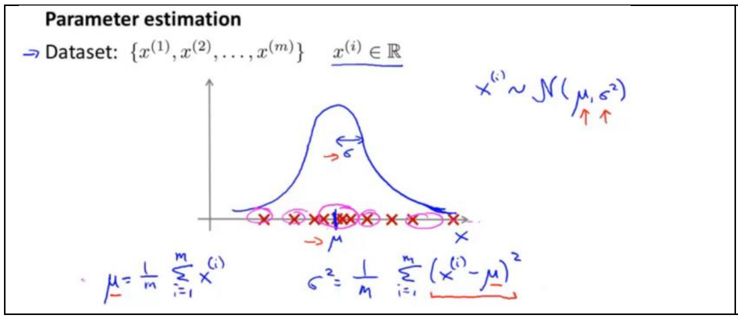


Gaussian (Normal) distribution

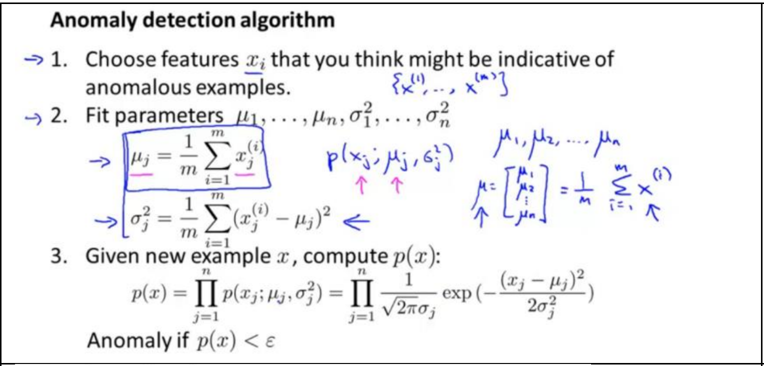
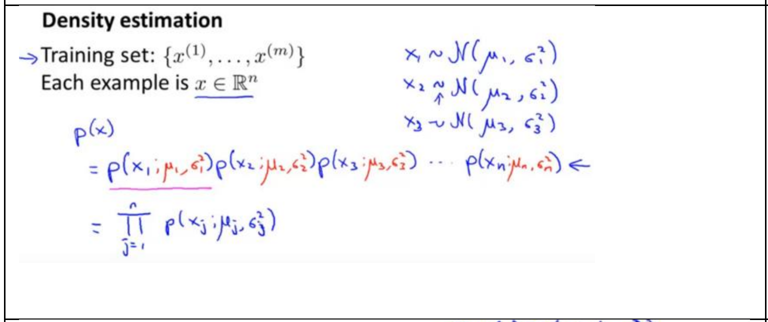
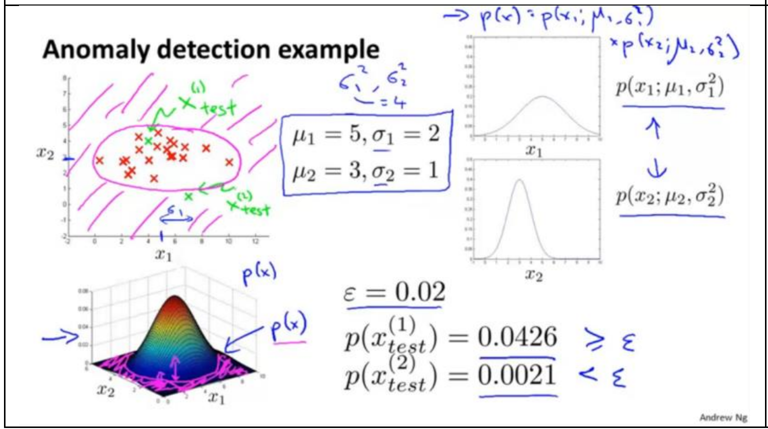
Say $x \in \mathbb{R}$. If x is a distributed Gaussian with mean μ, variance σ^2.

$x \sim \mathcal{N}(\mu, \sigma^2)$ "distributed as"

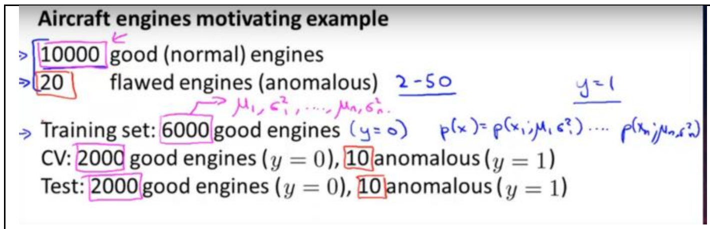
$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\,\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

 **Parameter estimation** → Dataset: $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ $\underline{x^{(i)} \in \mathbb{R}}$ $x^{(i)} \sim \mathcal{N}(\mu, \sigma^2)$ $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$ $\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu)^2$ 	If you suspect that the samples are distributed according to a normal distribution then you can try to estimate the values of the parameters μ and σ from the dataset (for each feature). The estimates calculated by these formulas are the maximum likelihood estimates of the parameters μ and σ.

.

 **Anomaly detection algorithm** → 1. Choose features x_i that you think might be indicative of anomalous examples. $\{x^{(1)}, \dots, x^{(m)}\}$ → 2. Fit parameters $\mu_1, \dots, \mu_n, \sigma_1^2, \dots, \sigma_n^2$ → $\boxed{\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)}}$ $p(x_j; \mu_j, \sigma_j^2)$ $\mu_1, \mu_2, \dots, \mu_n$ → $\sigma_j^2 = \frac{1}{m} \sum_{i=1}^{m} (x_j^{(i)} - \mu_j)^2$ ← $\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_n \end{bmatrix} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$ 3. Given 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 $p(x) < \varepsilon$	Try to think features that describe the general properties of the system that you examine hoping that some of them would take small or large values in unusual cases There are vectorized versions of parameter estimation
 **Density estimation** → Training set: $\{x^{(1)}, \dots, x^{(m)}\}$ Each example is $\underline{x \in \mathbb{R}^n}$ $x_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ $x_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$ $x_3 \sim \mathcal{N}(\mu_3, \sigma_3^2)$ $p(x)$ $= p(x_1; \mu_1, \sigma_1^2) p(x_2; \mu_2, \sigma_2^2) p(x_3; \mu_3, \sigma_3^2) \cdots p(x_n; \mu_n, \sigma_n^2)$ ← $= \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$	We assume that each feature is distributed according to a normal distribution which we want to estimate. <u>We also assume that the features are independent with each other so the probability of a specific feature vector (a combination of specific feature values) is given by the product of its individual feature s probabilities.</u> Notice that we assume independence on the values of the features, but it turns out that this algorithm works well even if the features are not independent.
 **Anomaly detection example** → $p(x) = p(x_1; \mu_1, \sigma_1^2) \times p(x_2; \mu_2, \sigma_2^2)$ $\sigma_1^2, \sigma_2^2 = 4$ $\boxed{\begin{matrix} \mu_1 = 5, \sigma_1 = 2 \\ \mu_2 = 3, \sigma_2 = 1 \end{matrix}}$ $p(x_1; \mu_1, \sigma_1^2)$ $p(x_2; \mu_2, \sigma_2^2)$ $\varepsilon = 0.02$ $p(x_{test}^{(1)}) = 0.0426 \geq \varepsilon$ $p(x_{test}^{(2)}) = 0.0021 < \varepsilon$ Andrew Ng $x_{test}^{(1)}$ $x_{test}^{(2)}$ x_1 x_2 $p(x)$	

Evaluating anomaly detection
in order to evaluate it we need a small number of known anomalous data points. We fit a model p(x) using the training data set of known non anomalous data points and we split the anomalous points to the CV and test sets. We use the <u>F1 score</u> (precision/recall trade off) for evaluation since the classes that we have are very skewed because we only have a small amount of known anomalous data. A small number of anomalous data points can be contained in the training set without affecting the whole process.



Aircraft engines motivating example

10000 good (normal) engines

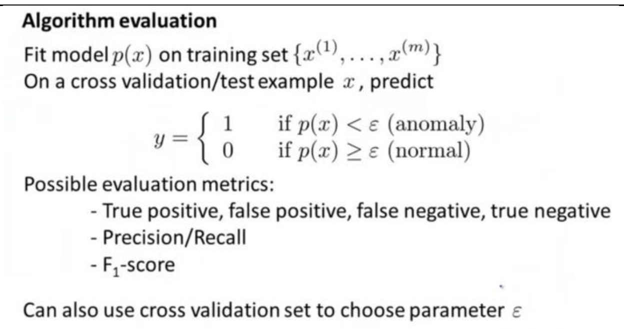
20 flawed engines (anomalous) 2-50

Training set: 6000 good engines $(y=0)$ $p(x) = p(x_1; \mu_1, \sigma_1^2) \cdots p(x_n; \mu_n, \sigma_n^2)$

CV: 2000 good engines $(y = 0)$, 10 anomalous $(y = 1)$

Test: 2000 good engines $(y = 0)$, 10 anomalous $(y = 1)$

$\mu_1, \sigma_1^2, \ldots, \mu_n, \sigma_n^2$
$y=1$



Algorithm evaluation

Fit model $p(x)$ on training set $\{x^{(1)}, \ldots, x^{(m)}\}$

On a cross validation/test example x, predict

$$y = \begin{cases} 1 & \text{if } p(x) < \varepsilon \text{ (anomaly)} \\ 0 & \text{if } p(x) \geq \varepsilon \text{ (normal)} \end{cases}$$

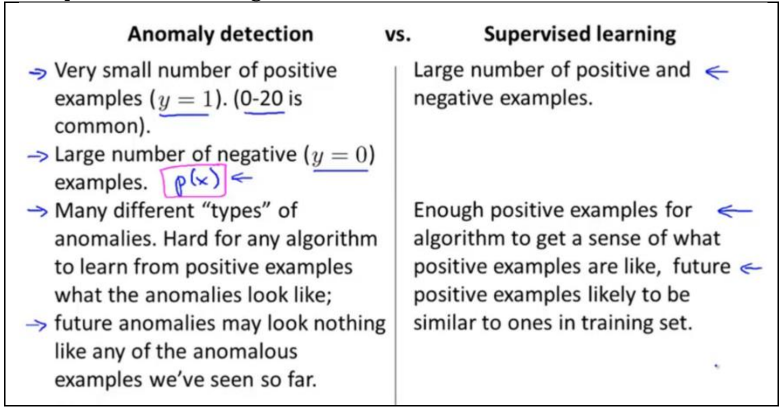
Possible evaluation metrics:

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

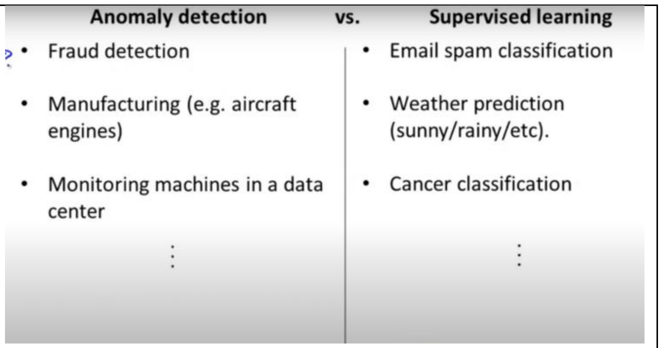
Can also use cross validation set to choose parameter ε

Try many ε values and pick the one with the larger F1 score.

Vs supervised learning



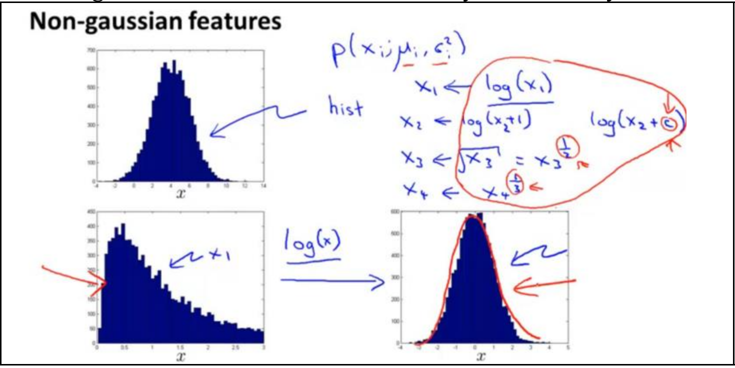
Anomaly detection	**vs. Supervised learning**
Very small number of positive examples $(y = 1)$. (0-20 is common).	Large number of positive and negative examples.
Large number of negative $(y = 0)$ examples. $p(x)$	
Many different "types" of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far.	Enough positive examples for algorithm to get a sense of what positive examples are like, future positive examples likely to be similar to ones in training set.



Anomaly detection	**vs. Supervised learning**
• Fraud detection	• Email spam classification
• Manufacturing (e.g. aircraft engines)	• Weather prediction (sunny/rainy/etc).
• Monitoring machines in a data center	• Cancer classification
⋮	⋮

Choosing features

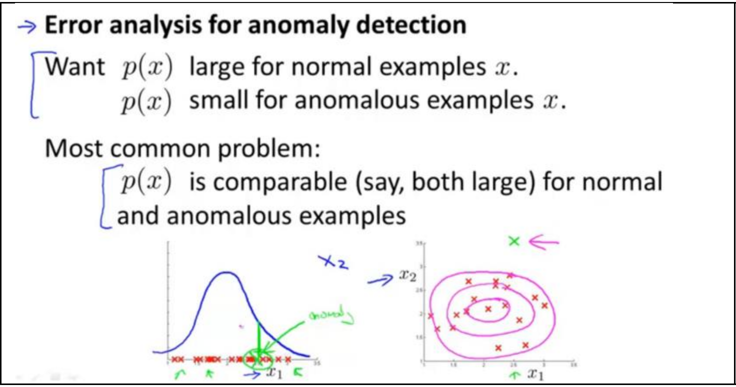
Choosing features is critical on how well your anomaly detection algorithm works



Non-gaussian features

-is the feature a normal distribution?

Transform data to make it more like a normal distribution (for example take log(x+c), or $x^{1/c}$ etc). so $x_{new}=\log(x)$



Error analysis for anomaly detection

Want $p(x)$ large for normal examples x.

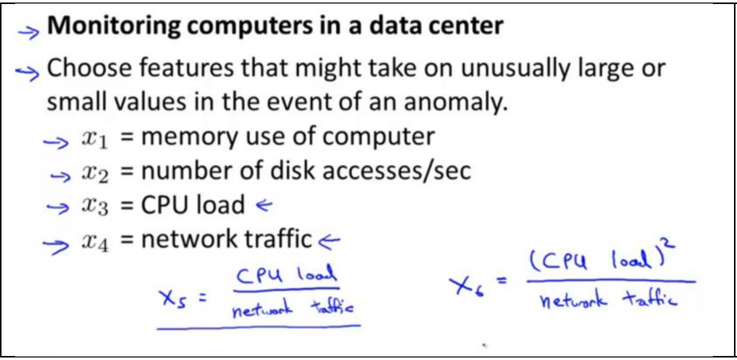
$p(x)$ small for anomalous examples x.

Most common problem:

$p(x)$ is comparable (say, both large) for normal and anomalous examples

By examining the green example which is an anomaly that wasn't captured by the x1 feature we can understand that there is another additional feature x2 that makes this example being anomalous.

Choose features after an error analysis (manually checking the error predictions of the algo and modify your features accordingly).



Monitoring computers in a data center

Choose features that might take on unusually large or small values in the event of an anomaly.

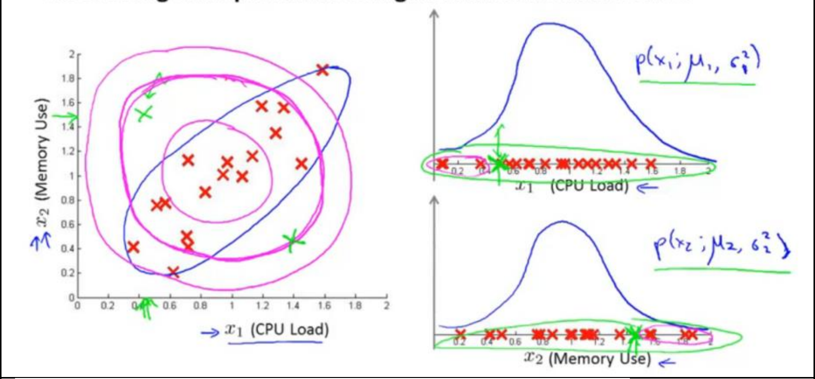
- x_1 = memory use of computer
- x_2 = number of disk accesses/sec
- x_3 = CPU load
- x_4 = network traffic

$x_5 = \frac{\text{CPU load}}{\text{network traffic}}$ $\quad x_6 = \frac{(\text{CPU load})^2}{\text{network traffic}}$

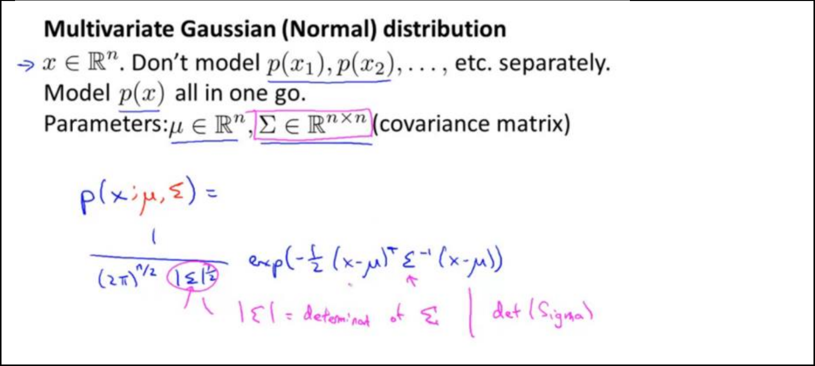
Chose/create features that take enormously large or small values in case of anomaly

Multivariate Gaussian distribution

Motivating example: Monitoring machines in a data center



Assume you have the upper left corner green example. If you have just two features that follow the normal distribution, then the green example is within the normal examples for each individual normal distribution. But obviously it is an anomaly. The reason that it is not caught, is that having two features with normal distribution assumes a circular probability surface (the pink circles). But, we want the blue ellipsis.



Multivariate Gaussian (Normal) distribution

$x \in \mathbb{R}^n$. Don't model $p(x_1), p(x_2), \ldots$, etc. separately.
Model $p(x)$ all in one go.
Parameters: $\mu \in \mathbb{R}^n$, $\Sigma \in \mathbb{R}^{n\times n}$ (covariance matrix)

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

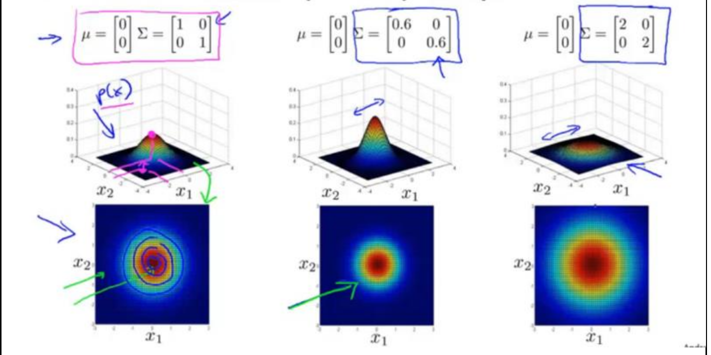
$|\Sigma|$ = determinant of Σ | det(Sigma)

The solution is a multivariate gaussian distribution which gives a probability distribution as a function of both variables (actually as a function of their mean and their covariance matrix).

Σ is the covariance matrix.

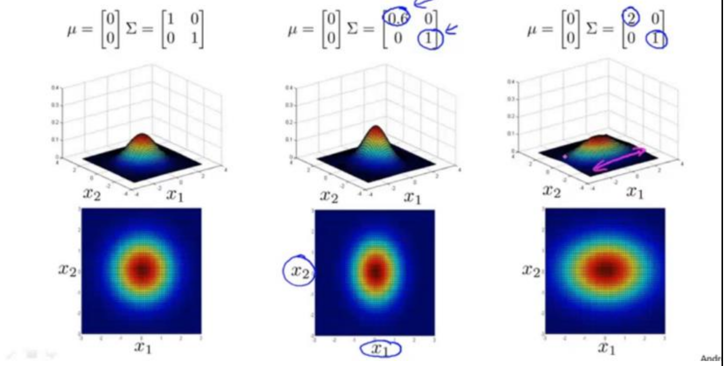
The covariance matrix

Multivariate Gaussian (Normal) examples

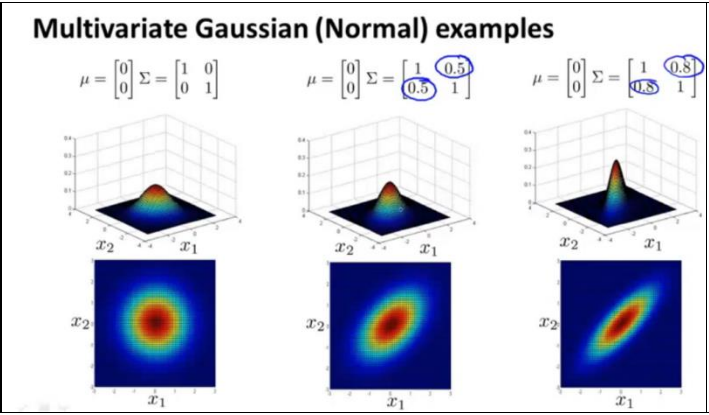


The diagonal of Σ simply shows the variance of each single feature.

Multivariate Gaussian (Normal) examples



If there is a covariance between the features you get this kind of probability distributions.



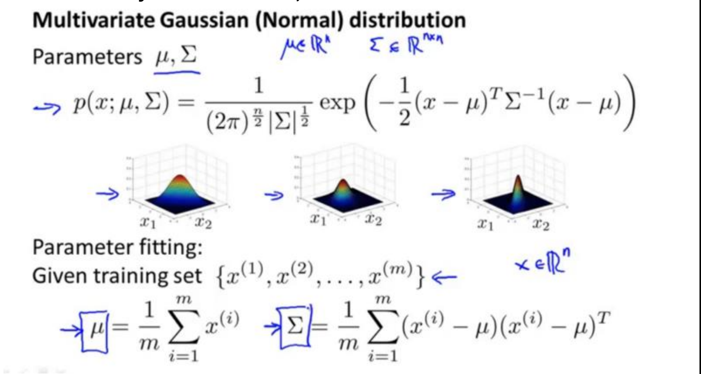
Multivariate Gaussian (Normal) examples

$\mu = \begin{bmatrix}0\\0\end{bmatrix} \Sigma = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$ $\quad \mu = \begin{bmatrix}0\\0\end{bmatrix} \Sigma = \begin{bmatrix}1 & 0.5\\0.5 & 1\end{bmatrix}$ $\quad \mu = \begin{bmatrix}0\\0\end{bmatrix} \Sigma = \begin{bmatrix}1 & 0.8\\0.8 & 1\end{bmatrix}$

Have in mind that the surface or volume (or higher) of a probability distribution is 1.

Putting it together

This is how you calculate μ and Σ



Multivariate Gaussian (Normal) distribution

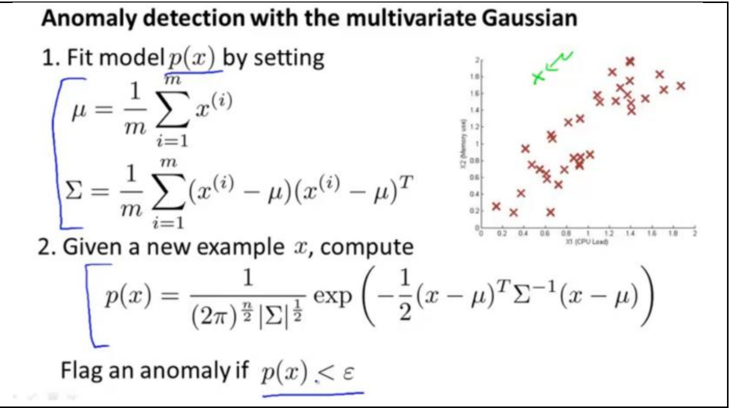
Parameters μ, Σ $\quad \mu \in \mathbb{R}^n \quad \Sigma \in \mathbb{R}^{n \times n}$

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

Parameter fitting:

Given training set $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\} \leftarrow$ $\quad x \in \mathbb{R}^n$

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



Anomaly detection with the multivariate Gaussian

1. Fit model $p(x)$ by setting

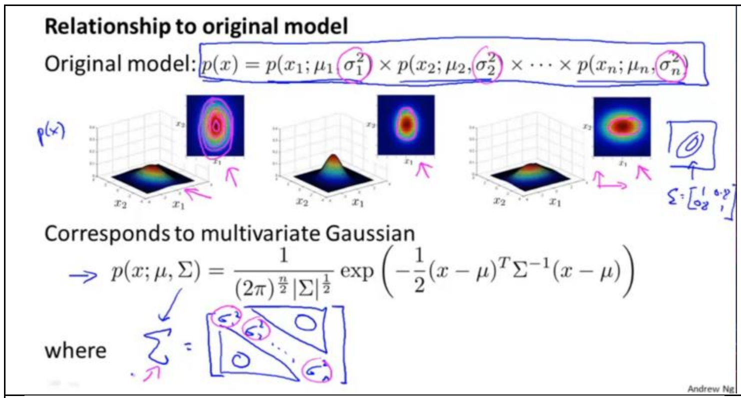
$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

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

2. Given a new example x, compute

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

Flag an anomaly if $p(x) < \varepsilon$



Relationship to original model

Original model: $\boxed{p(x) = p(x_1; \mu_1, \sigma_1^2) \times p(x_2; \mu_2, \sigma_2^2) \times \cdots \times p(x_n; \mu_n, \sigma_n^2)}$

$p(x)$ $\quad \Sigma = \begin{bmatrix}1 & 0.8\\0.8 & 1\end{bmatrix}$

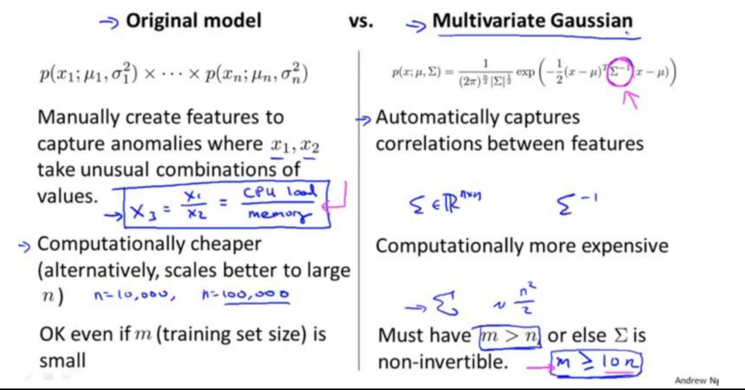
Corresponds to multivariate Gaussian

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

where $\Sigma = \begin{bmatrix}\sigma_1^2 & & 0\\ & \sigma_2^2 & \\ & \ddots & \\ 0 & & \sigma_n^2\end{bmatrix}$

Andrew Ng

Notice that the original model (Gaussian for each feature) is a special case of the multivariate case, where the axis of the gaussian distribution are parallel to the features axis (there is no covariance between features).



$\rightarrow$ Original model	vs. $\rightarrow$ Multivariate Gaussian
$p(x_1; \mu_1, \sigma_1^2) \times \cdots \times p(x_n; \mu_n, \sigma_n^2)$	$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} \lvert\Sigma\rvert^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right)$
Manually create features to capture anomalies where x_1, x_2 take unusual combinations of values. $\rightarrow \boxed{x_3 = \frac{x_1}{x_2} = \frac{\text{CPU load}}{\text{memory}}}$	$\rightarrow$ Automatically captures correlations between features
	$\Sigma \in \mathbb{R}^{n \times n} \quad \Sigma^{-1}$
$\rightarrow$ Computationally cheaper (alternatively, scales better to large n) $n = 10{,}000$, $n = 100{,}000$	Computationally more expensive
	$\rightarrow \Sigma \sim \frac{n^2}{2}$
OK even if m (training set size) is small	Must have $\boxed{m > n}$ or else Σ is non-invertible. $\rightarrow \boxed{m \geq 10n}$

Andrew Ng

The original model is much more efficient computationally, because the multivariate case requires matrix inverting, so it is always preferred if possible.

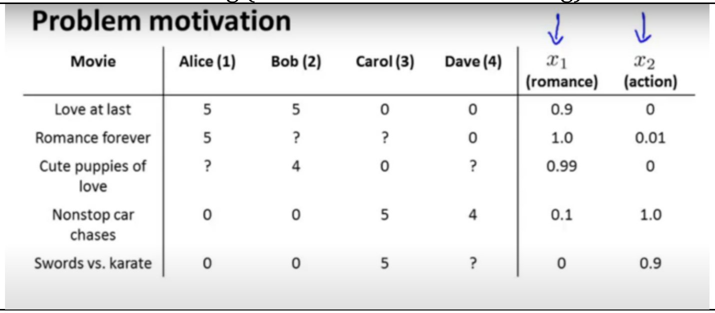
If there is a rare combination of feature values (like in the example that was considered normal but was in reality anomalous) then in order to use the original model, you can redesign your features. For example take the ratio of the two features. This way even if the values are normal for each feature individually, the ratio of the values would be not normal.

Non invertible covariance matrix

- If you have redundant (linearly dependent) features (x1=x2, x3=x4+x5 etc.) then the covariance matrix might not be invertible and multivariate gaussian couldn't be used.
- If m<n

Recommender systems

Collaborative filtering (automated feature choosing)

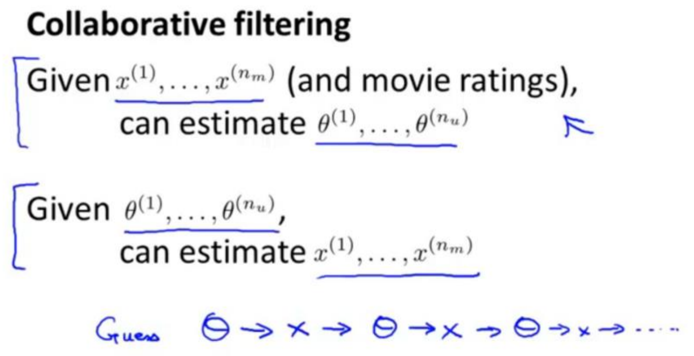


Problem motivation

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	0.9	0
Romance forever	5	?	?	0	1.0	0.01
Cute puppies of love	?	4	0	?	0.99	0
Nonstop car chases	0	0	5	4	0.1	1.0
Swords vs. karate	0	0	5	?	0	0.9

If we have the features and their values we can make linear regression to each user to calculate the θ parameters that form a hypothesis that match the user's ratings. Then use that hypothesis to predict what rating the user will give in new movies and recommend to them movies that they will like.

But in the general case we don't have the features. There is a way to find them automatically though. If we somehow have the θ parameters for each user (we ask them to tell us what movies they like), we can use it to calculate the feature values. It is the same formulas as linear regression but now you have Θ and solve for x.



Collaborative filtering

Given $x^{(1)}, \dots, x^{(n_m)}$ (and movie ratings),
can estimate $\theta^{(1)}, \dots, \theta^{(n_u)}$

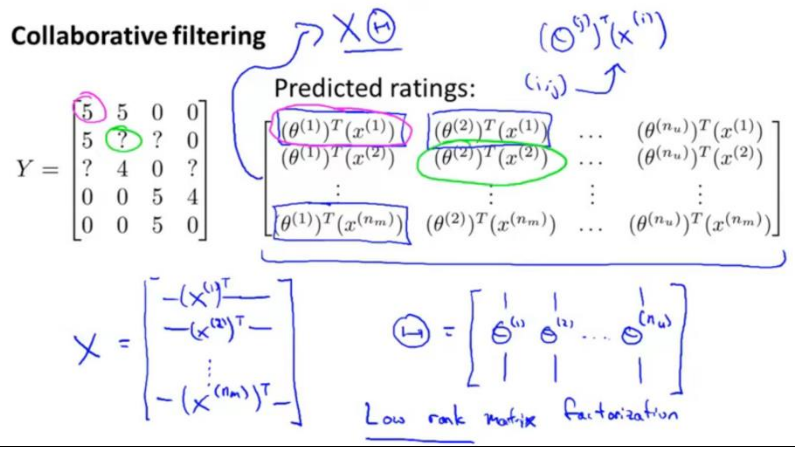
Given $\theta^{(1)}, \dots, \theta^{(n_u)}$,
can estimate $x^{(1)}, \dots, x^{(n_m)}$

Guess $\Theta \rightarrow x \rightarrow \Theta \rightarrow x \rightarrow \Theta \rightarrow x \rightarrow \dots$

But what happens if you don't have nor θ neither x? then you can guess a random initial Θ and estimate a x from it. <u>notice that you must choose the number of features, so the number of parameters θ, but you don't know what these features are</u>. Then use that x to estimate a new better θ and so on until convergence. This is the **collaborative filtering** algorithm. initially you don't know what these learned features represent, but you can introspect them and see. For example if you have used two features, <u>the algorithm might learn the "romance" and "action" features on its own</u>.

For this collaborative filtering technique to work, each user must have rated many movies and each movie must be rated by many users.

Vectorized implementation
Also called low rank matrix factorization



Collaborative filtering $\rightarrow X\Theta$ $\quad (\Theta^{(j)})^T(x^{(i)})$

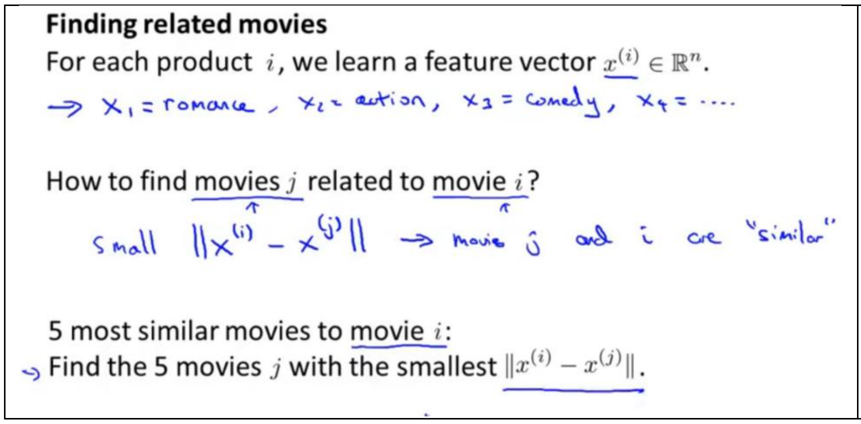
Predicted ratings: (i,j)

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 5 & ? & ? & 0 \\ ? & 4 & 0 & ? \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{bmatrix} \quad \underbrace{\begin{bmatrix} (\theta^{(1)})^T(x^{(1)}) & (\theta^{(2)})^T(x^{(1)}) & \dots & (\theta^{(n_u)})^T(x^{(1)}) \\ (\theta^{(1)})^T(x^{(2)}) & (\theta^{(2)})^T(x^{(2)}) & \dots & (\theta^{(n_u)})^T(x^{(2)}) \\ \vdots & \vdots & \vdots & \vdots \\ (\theta^{(1)})^T(x^{(n_m)}) & (\theta^{(2)})^T(x^{(n_m)}) & \dots & (\theta^{(n_u)})^T(x^{(n_m)}) \end{bmatrix}}$$

$$X = \begin{bmatrix} -(x^{(1)})^T- \\ -(x^{(2)})^T- \\ \vdots \\ -(x^{(n_m)})^T- \end{bmatrix} \qquad \Theta = \begin{bmatrix} | & | & & | \\ \Theta^{(1)} & \Theta^{(2)} & \dots & \Theta^{(n_u)} \\ | & | & & | \end{bmatrix}$$

<u>Low rank</u> matrix factorization

Y is the matrix of ratings. Num of columns is the num of users. Num of rows is the num of movies.



Finding related movies

For each product i, we learn a feature vector $x^{(i)} \in \mathbb{R}^n$.

$\rightarrow$ x_1 = romance, x_2 = action, x_3 = comedy, x_4 = ...

How to find movies j related to movie i?

small $\|x^{(i)} - x^{(j)}\|$ $\rightarrow$ movie j and i are "similar"

5 most similar movies to movie i:

Find the 5 movies j with the smallest $\|x^{(i)} - x^{(j)}\|$.

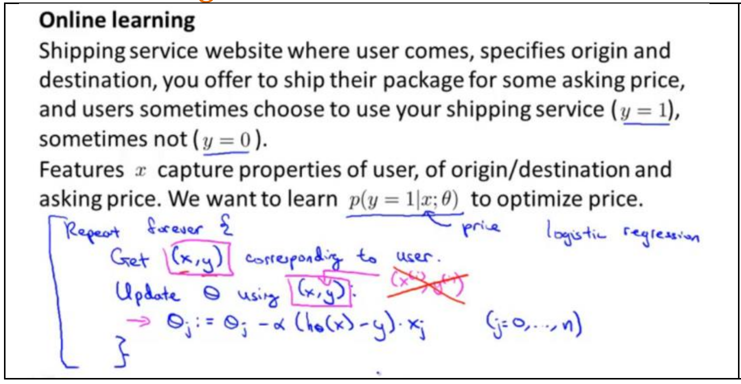
Mean normalization

Sometimes it might be useful to so mean normalization to the data as a pre processing step. This would be useful to predict ratings for users that haven't rated any movie. The prediction would be that they will rate a movie with its mean value.

Large scale machine learning tricks

Tricks to help the efficiency of ML algorithm when dealing with big data (100m examples etc.)

Online learning



Online learning

Shipping service website where user comes, specifies origin and destination, you offer to ship their package for some asking price, and users sometimes choose to use your shipping service ($y = 1$), sometimes not ($y = 0$).

Features x capture properties of user, of origin/destination and asking price. We want to learn $p(y = 1|x; \theta)$ to optimize price.

price

logistic regression

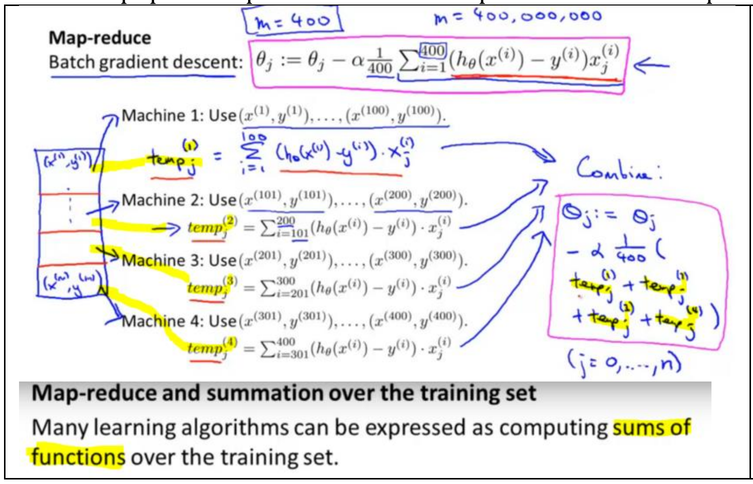
```
Repeat forever {
    Get (x,y) corresponding to user.
    Update θ using (x,y): ~~(x^(i),y^(i))~~
    → $\theta_j := \theta_j - \alpha(h_\theta(x) - y)\cdot x_j$      $(j=0,\ldots,n)$
}
```

When there are a lot of data coming in real time, for example website visitors on a large website, you can implement online learning on them. You use each new example and train your algorithm with just this example. Then you discard this example. This is the point. Not having to store and handle all that data. The first hypothesis is not good. But as new examples come in the hypothesis keeps improving. It also responds to changes in the trend.

If your real time data is not large, then you have to group let's say 1000 examples and train a model on them and continue like this in batches. This is not online learning.

Map Reduce

This is a very important technique that allows many real world cases to run fast. It is as important as stochastic gradient descent. There are popular implementations of map reduce like Hadoop.



Map-reduce

$m = 400$ $\quad$ $m = 400{,}000{,}000$

Batch gradient descent: $\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$

Machine 1: Use $(x^{(1)}, y^{(1)}), \ldots, (x^{(100)}, y^{(100)})$.

$temp_j^{(1)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

Machine 2: Use $(x^{(101)}, y^{(101)}), \ldots, (x^{(200)}, y^{(200)})$.

$temp_j^{(2)} = \sum_{i=101}^{200} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

Machine 3: Use $(x^{(201)}, y^{(201)}), \ldots, (x^{(300)}, y^{(300)})$.

$temp_j^{(3)} = \sum_{i=201}^{300} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

Machine 4: Use $(x^{(301)}, y^{(301)}), \ldots, (x^{(400)}, y^{(400)})$.

$temp_j^{(4)} = \sum_{i=301}^{400} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

Combine:

$\theta_j := \theta_j - \alpha \frac{1}{400} (temp_j^{(1)} + temp_j^{(2)} + temp_j^{(3)} + temp_j^{(4)})$

$(j = 0, \ldots, n)$

Map-reduce and summation over the training set

Many learning algorithms can be expressed as computing sums of functions over the training set.

You have a cluster of machines. You split the training set into equal parts, and send each part to a different machine in the cluster. This is the map step.

Each machine calculates the summation over the examples in its part. Then the result is send to a master node that combines them together to calculate the total sum. This is the reduce step.

If an algorithm's main computation load is to compute sums of many terms, then you can easily parallelize that work with map reduce.

ML pipelines

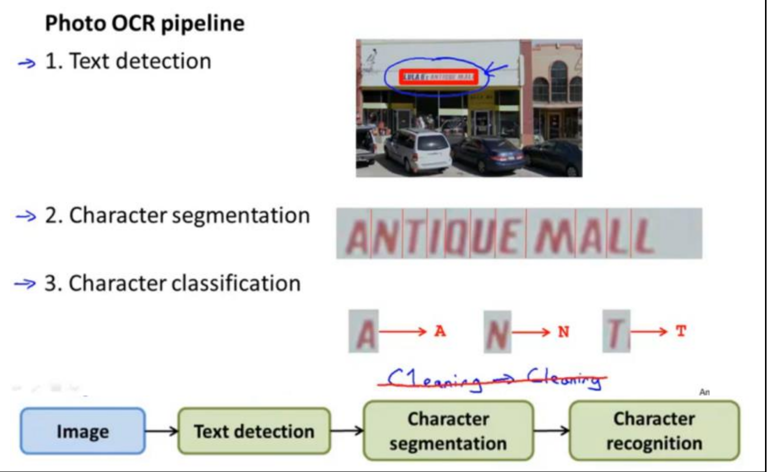
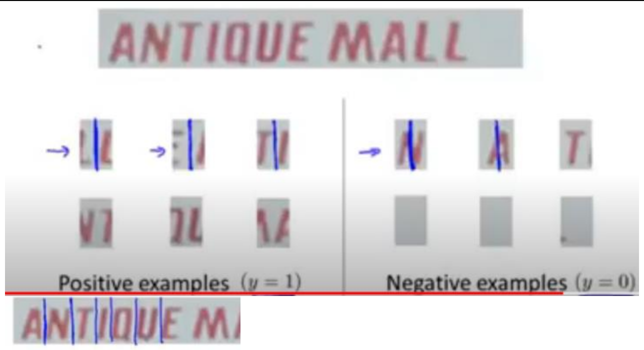


Photo OCR pipeline

1. Text detection
2. Character segmentation
3. Character classification

Text detection
It works with a sliding window. Notice that for pedestrians it's easier to identify them because the aspect ratio of a human is the same no matter his size. But for the text this is not the case.
For pedestrians you train a classifier to detect pedestrians for images of a certain aspect ratio. You get a sliding window of a small specific aspect ratio and parse the image. Then with a larger one and so on. Notice that the classifier that you trained, gets a 82*36 pixels image as input to tell if there is a pedestrian or not. So for the larger windows you have to resize the sliced image to that size.
For text you train a classifier to detect letters in a rectangular image. Notice that you have to do data augmentation to get more data for your classifier. Then you parse the image with windows of various sizes. Then you enlarge the areas that have letters and form a parallelogram around the areas that have all text.

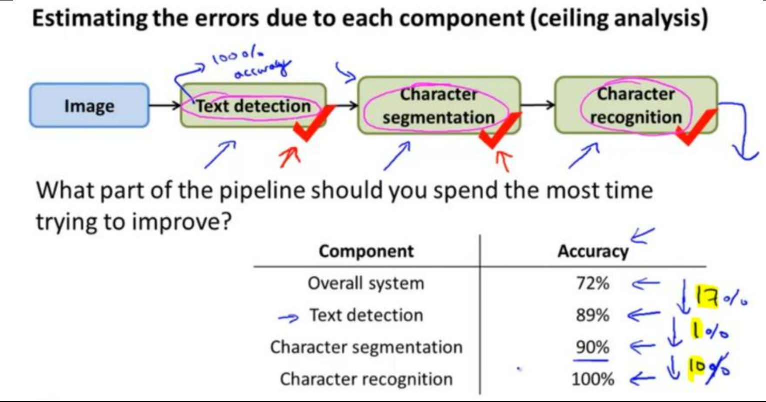
Character segmentation



You train a classifier to detect if there is a character split in an image of specific size. Then you use a sliding window on the detected text of the previous step and use the classifier to detect character splits. This way you split the word in characters and you can get images that contain complete characters.
Then you move on to the next step, which is to detect what character is in the image.

Diagnostics of ML pipelines

Ceiling analysis


Estimating the errors due to each component (ceiling analysis)

What part of the pipeline should you spend the most time trying to improve?

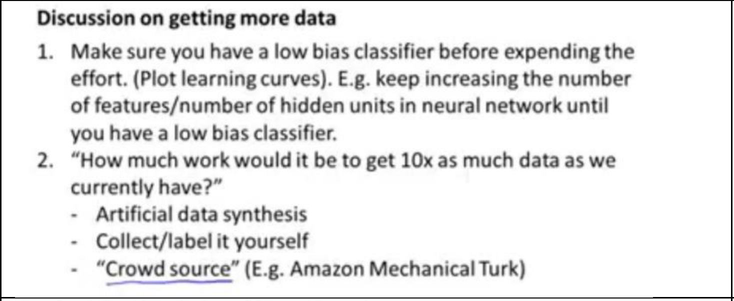
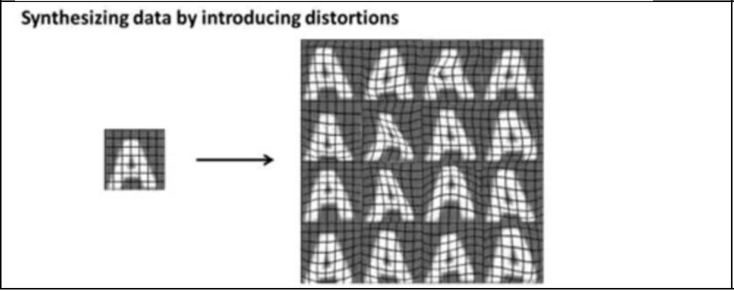
Component	Accuracy
Overall system	72%
Text detection	89%
Character segmentation	90%
Character recognition	100%

Initially you measure the overall system accuracy. But you want to identify which of the subsystems is more important for the overall accuracy.
All subsystems have an error rate. So, the text detection might detect areas without text, or skip areas with text. Character segmentation will not work properly in these cases. So the text detection affects the error rate of the character segmentation. To disentangle the error rates we apply the ceiling analysis.

We manually fix all the errors produced by the text detection system, so all of its prediction are correct. It now has zero error rate. Then we measure the overall accuracy. It has been increased by a certain magnitude. This is the contribution of the text detection to the overall system accuracy.
Then we do the same with the character segmentation and remeasure the overall rate.

This way we see the contributions of each subsystem and we can detect the most important one, so that we can focus our efforts on it

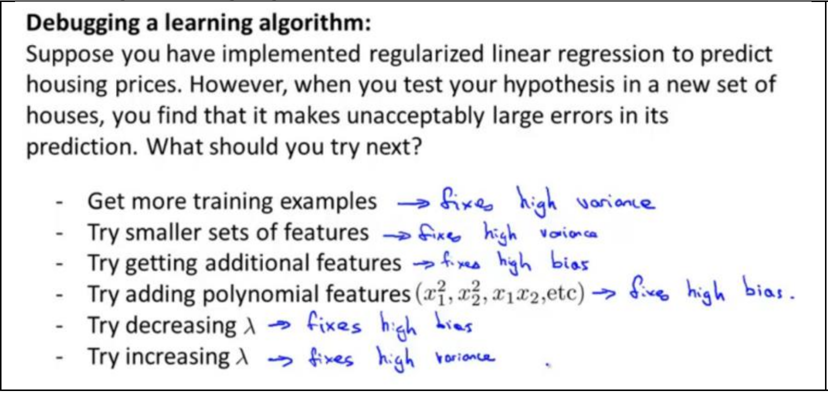
Data augmentation
Distortions (generated data in general) should be representation of the type of noise/distortions in the test set

 **Discussion on getting more data** 1. Make sure you have a low bias classifier before expending the effort. (Plot learning curves). E.g. keep increasing the number of features/number of hidden units in neural network until you have a low bias classifier. 2. "How much work would it be to get 10x as much data as we currently have?" - Artificial data synthesis - Collect/label it yourself - "Crowd source" (E.g. Amazon Mechanical Turk)	
 **Synthesizing data by introducing distortions**	Get digits from different fonts and paste them in front of different backgrounds, apply some blurring, rotation etc.

Practical

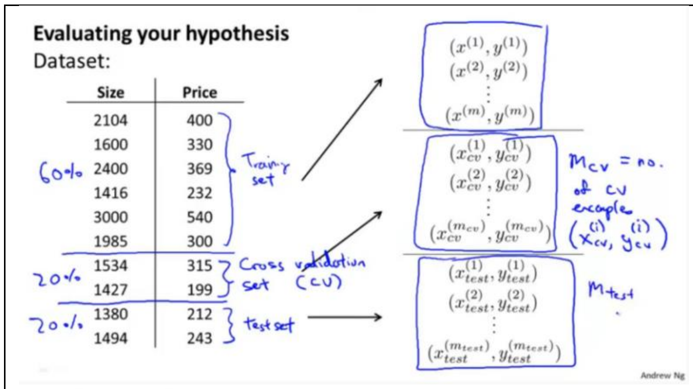
Diagnostics

Evaluating learning algorithms

 **Debugging a learning algorithm:** Suppose you have implemented regularized linear regression to predict housing prices. However, when you test your hypothesis in a new set of houses, you find that it makes unacceptably large errors in its prediction. What should you try next? - Get more training examples → fixes high variance - Try smaller sets of features → fixes high variance - Try getting additional features → fixes high bias - Try adding polynomial features $(x_1^2, x_2^2, x_1x_2, \text{etc})$ → fixes high bias. - Try decreasing λ → fixes high bias - Try increasing λ → fixes high variance	There are methods that you can use to determine which of these steps isn't the source of the problem so that you don't lose valuable resources trying to improve it, for example there might be no need to collect more data. The problem might be somewhere else.

Generalization error and model selection
(Training set, test set and cross validation set)
We always must choose some model parameters, for example for linear regression the degree of polynomial d or the regularization constant λ. The performance of a hypothesis is measured with the generalization error which shoes how well the hypothesis function generalizes in new examples.

A way to select these parameters is with a DOE study. Train 10 hypothesis of different degree and check the error on the cross validation set. Pick the one with the smallest error. Notice though that the cv error is not the real generalization error of this hypothesis. The real generalization error is the error on a different set, the test set. For this reason, you need 3 sets.



Evaluating your hypothesis

Dataset:

Size	Price
2104	400
1600	330
2400	369
1416	232
3000	540
1985	300
1534	315
1427	199
1380	212
1494	243

60% Training set

20% Cross validation set (CV)

20% test set

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

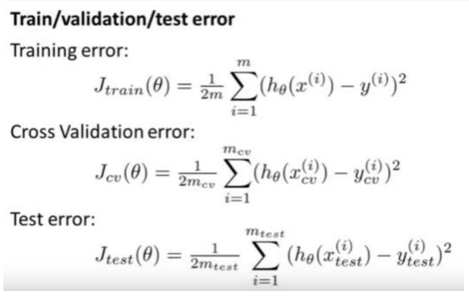
$(x_{cv}^{(1)}, y_{cv}^{(1)})$, $(x_{cv}^{(2)}, y_{cv}^{(2)})$, $\vdots$, $(x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})$

m_{cv} = no. of cv example $(x_{cv}^{(i)}, y_{cv}^{(i)})$

$(x_{test}^{(1)}, y_{test}^{(1)})$, $(x_{test}^{(2)}, y_{test}^{(2)})$, $\vdots$, $(x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$

m_{test}

Andrew Ng



Train/validation/test error

Training error:

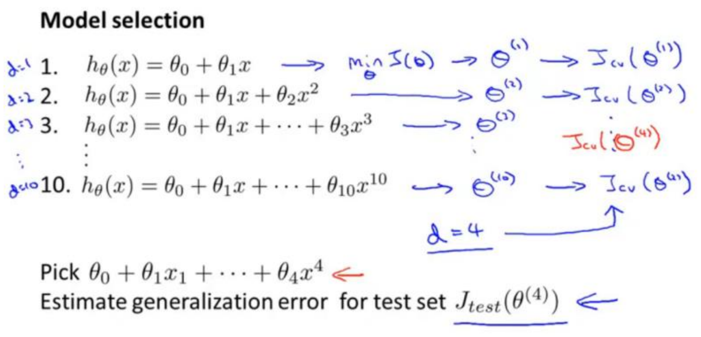
$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Cross Validation error:

$$J_{cv}(\theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

Test error:

$$J_{test}(\theta) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}(h_\theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$$



Model selection

$d=1$ 1. $h_\theta(x) = \theta_0 + \theta_1 x \longrightarrow \min_\theta J(\theta) \rightarrow \theta^{(1)} \longrightarrow J_{cv}(\theta^{(1)})$

$d=2$ 2. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 \longrightarrow \theta^{(2)} \longrightarrow J_{cv}(\theta^{(2)})$

$d=3$ 3. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_3 x^3 \longrightarrow \theta^{(3)}$

$\vdots$ $\qquad J_{cv}(\theta^{(4)})$

$d=10$ 10. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10} \longrightarrow \theta^{(10)} \longrightarrow J_{cv}(\theta^{(10)})$

$d=4$

Pick $\theta_0 + \theta_1 x_1 + \cdots + \theta_4 x^4 \leftarrow$

Estimate generalization error for test set $\underline{J_{test}(\theta^{(4)})} \leftarrow$

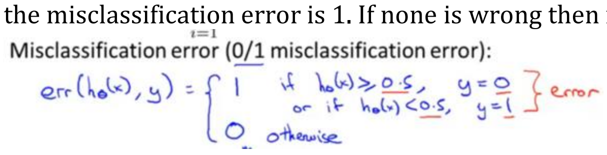
In order to evaluate a hypothesis, to get the generalization error it is best to split our data to three sets not just two. A training set, a cross validation set (CV) and a test set with proportions of 60/20/20. (or 70/30 if no cv set)

If we only used a test set
we try 10 different degrees of polynomial. We measure the test set cost for each one and pick the one with the smallest cost. But notice that this test set cost isn't the generalization cost. Since we picked the hypothesis based on the costs on the test set, we can say that <u>the hyperparameter d has been fit to the test set.</u>

Actually, we call **cross validation set** the one that we use to evaluate the cost for each hypothesis, and **test set** the one we use to evaluate the generalization error

Notice that the formula for cv and test errors do not contain the regularization term. They just measure how well the hypothesis fits to the data.

For classification there is an alternative test set metric that might be easier to interpret, the misclassification error. It is 1 if there is an error and 0 if there isn't. if all test examples are wrong (1) then the misclassification error is 1. If none is wrong then it is 0.

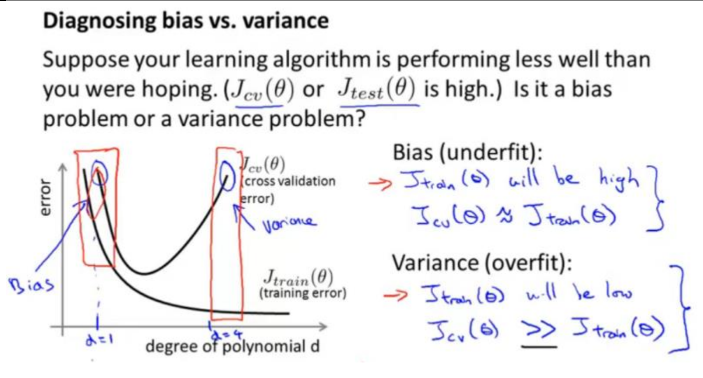


Misclassification error (0/1 misclassification error):

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

High bias or high variance problem

Or in other words underfitting or overfitting.



Diagnosing bias vs. variance

Suppose your learning algorithm is performing less well than you were hoping. ($\underline{J_{cv}(\theta)}$ or $\underline{J_{test}(\theta)}$ is high.) Is it a bias problem or a variance problem?

Bias (underfit):

$\rightarrow J_{train}(\theta)$ will be high

$J_{cv}(\theta) \approx J_{train}(\theta)$

Variance (overfit):

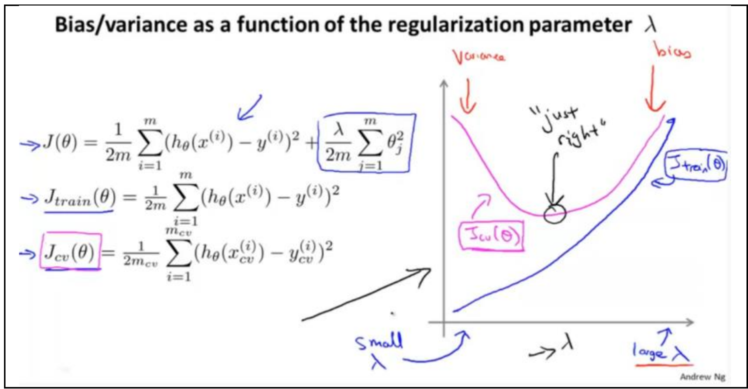
$\rightarrow J_{train}(\theta)$ will be low

$J_{cv}(\theta) \underline{\gg} J_{train}(\theta)$

One method that you can use in order to identify if your model suffers from overfitting or underfitting is to plot the training set error and the cross validation set error versus the hyperparameter of interest (d for linear regression) in the same plot.

As d increases the hypothesis can better fit to the data so the training error would decrease. The CV error would be large initially for small values of d since the hypothesis isn't flexible enough to fit well to the data it would decrease as d increases but from a point on would start increase again since if we have a very flexible hypothesis it would overfit which means that the cost in the CV set would be large.

Large values for both means bias problem.
Large for CV but small for training set means great variance problem.

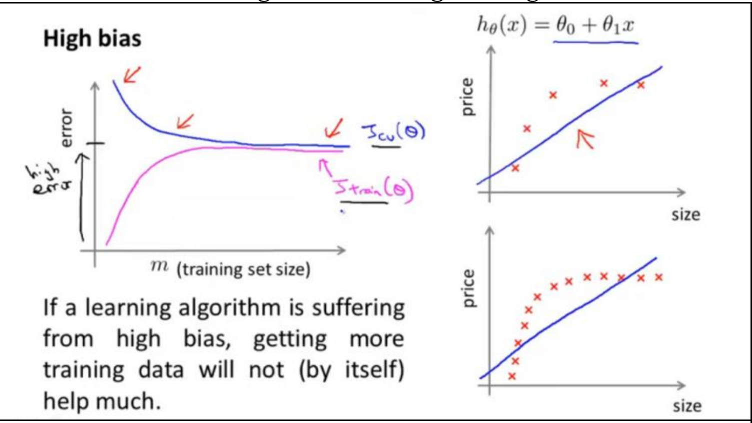


In order to select a good value for the regularization constant λ, you can use a similar plot of the training and CV costs over the regularization constant λ.

Notice that we use the regularized cost function to train the model, but we use the unregularized cost function for the plot so that we can compare it with the CV cost which is unregularized too.

Learning curves

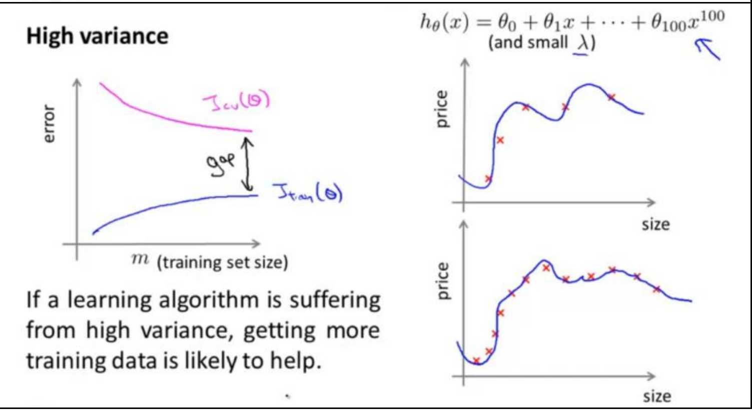
Now we plot the training and CV error with respect to the training examples. The training examples is a fixed number for a given problem but for the sake of plotting these curves we evaluate the costs for smaller numbers of training examples in order to see the effect of the training data size and gain insights about the bias or variance of our model.



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

Characteristic of high bias situation is that both the training and the CV errors are high and also similar to each other.

In high bias cases more data don't help. In these cases you might try to add more features or additional hidden units and recreating a learning curve. Now more data might help.



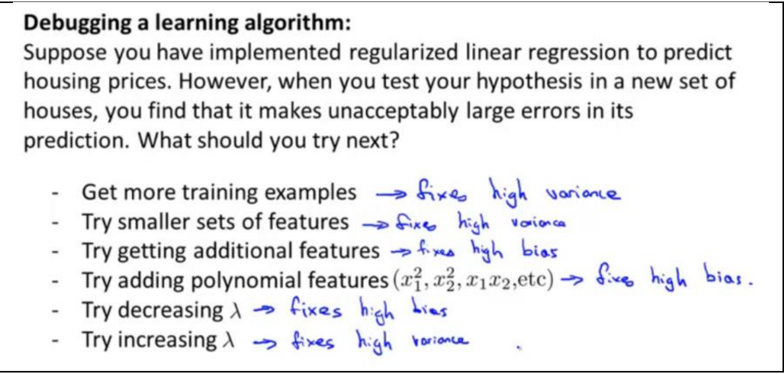
If a learning algorithm is suffering from high variance, getting more training data is likely to help.

The characteristic of high variance is that training error is small and CV error is large. There is a big gap between them.

In high variance situations more data might help (the curves would eventually converge to a lower value)

As I understand it if the training error and the cross validation error are at the same level and this level is low then our hypothesis has learned well from the data and generalizes well too.

In conclusion



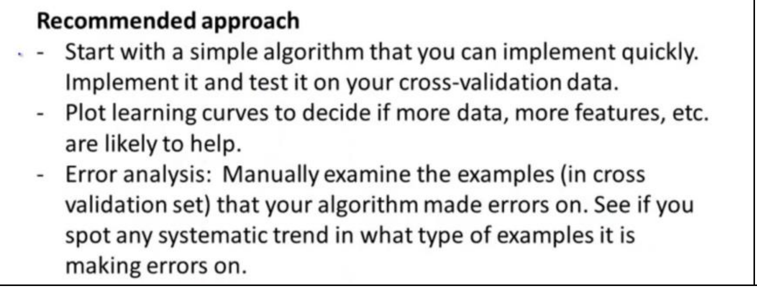
Debugging a learning algorithm:

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

- Get more training examples $\rightarrow$ fixes high variance
- Try smaller sets of features $\rightarrow$ fixes high variance
- Try getting additional features $\rightarrow$ fixes high bias
- Try adding polynomial features $(x_1^2, x_2^2, x_1x_2, \text{etc})$ $\rightarrow$ fixes high bias.
- Try decreasing λ $\rightarrow$ fixes high bias
- Try increasing λ $\rightarrow$ fixes high variance

Machine learning system design

How to start working on a problem



Recommended approach

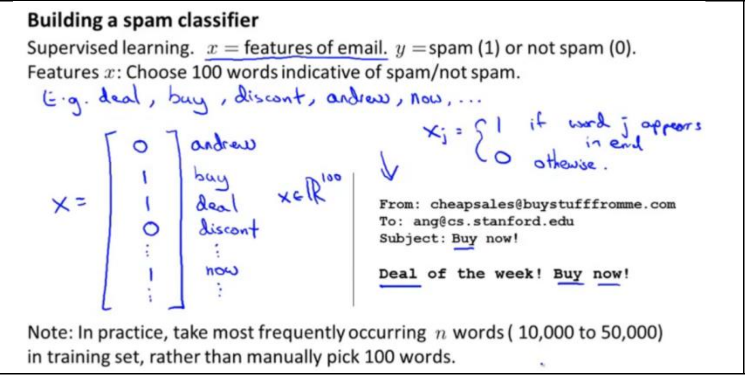
- Start with a simple algorithm that you can implement quickly. Implement it and test it on your cross-validation data.
- Plot learning curves to decide if more data, more features, etc. are likely to help.
- Error analysis: Manually examine the examples (in cross validation set) that your algorithm made errors on. See if you spot any systematic trend in what type of examples it is making errors on.

Error analysis

The process of manually examining the errors that the model does on the cross validation set, to get some insights on what you need to do to improve it.

You can use a numerical evaluation for example the CV cost for testing various alterations and tuning.

For example, deciding what to do for building a spam classifier.



Building a spam classifier

Supervised learning. x = features of email. y = spam (1) or not spam (0).

Features x: Choose 100 words indicative of spam/not spam.

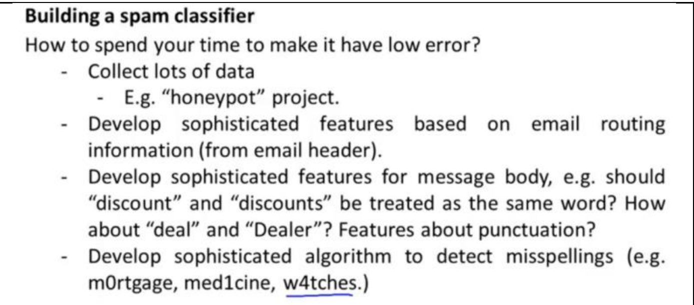
E.g. deal, buy, discount, andrew, now, ...

$$x = \begin{array}{c|l} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 1 \\ \vdots \end{bmatrix} & \begin{array}{l} \text{andrew} \\ \text{buy} \\ \text{deal} \\ \text{discont} \\ \vdots \\ \text{now} \\ \vdots \end{array} \end{array} \qquad x \in \mathbb{R}^{100}$$

$$x_j = \begin{cases} 1 & \text{if word } j \text{ appears in email} \\ 0 & \text{otherwise.} \end{cases}$$

```
From: cheapsales@buystufffromme.com
To: ang@cs.stanford.edu
Subject: Buy now!

Deal of the week! Buy now!
```

Note: In practice, take most frequently occurring n words (10,000 to 50,000) in training set, rather than manually pick 100 words.

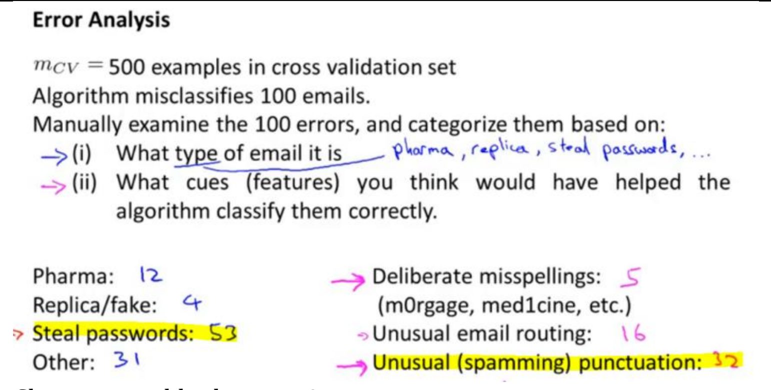


Building a spam classifier

How to spend your time to make it have low error?

- Collect lots of data
  - E.g. "honeypot" project.
- Develop sophisticated features based on email routing information (from email header).
- Develop sophisticated features for message body, e.g. should "discount" and "discounts" be treated as the same word? How about "deal" and "Dealer"? Features about punctuation?
- Develop sophisticated algorithm to detect misspellings (e.g. m0rtgage, med1cine, w4tches.)

Error analysis



Error Analysis

m_{CV} = 500 examples in cross validation set

Algorithm misclassifies 100 emails.

Manually examine the 100 errors, and categorize them based on:

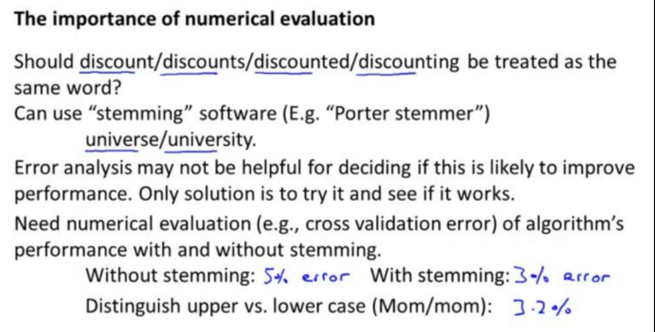
→ (i) What type of email it is — Pharma, replica, steal passwords, ...

→ (ii) What cues (features) you think would have helped the algorithm classify them correctly.

Pharma: 12	→ Deliberate misspellings: 5 (m0rgage, med1cine, etc.)
Replica/fake: 4	→ Unusual email routing: 16
→ Steal passwords: 53	→ Unusual (spamming) punctuation: 32
Other: 31	

Choose to tackle the most important error cases

Stemming is a technique used to consider similar words as the same one.



The importance of numerical evaluation

Should discount/discounts/discounted/discounting be treated as the same word?

Can use "stemming" software (E.g. "Porter stemmer")

universe/university.

Error analysis may not be helpful for deciding if this is likely to improve performance. Only solution is to try it and see if it works.

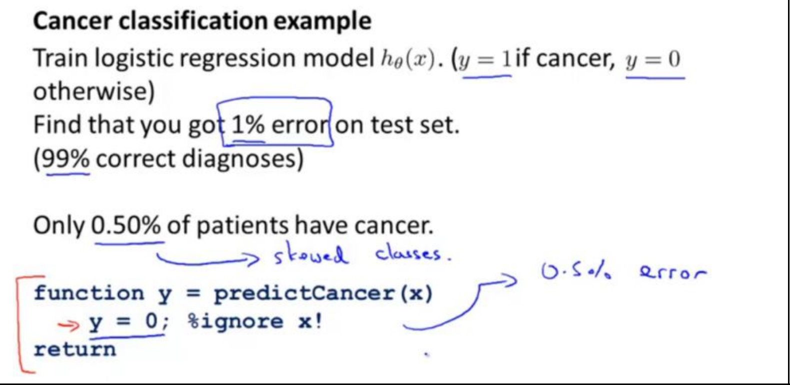
Need numerical evaluation (e.g., cross validation error) of algorithm's performance with and without stemming.

Without stemming: 5% error With stemming: 3% error

Distinguish upper vs. lower case (Mom/mom): 3.2%

F1 Score

Skewed classes and Precision/Recall



Cancer classification example

Train logistic regression model $h_\theta(x)$. ($y = 1$ if cancer, $y = 0$ otherwise)

Find that you got $\boxed{\text{1\% error}}$ on test set.

(99% correct diagnoses)

Only 0.50% of patients have cancer.

→ skewed classes.

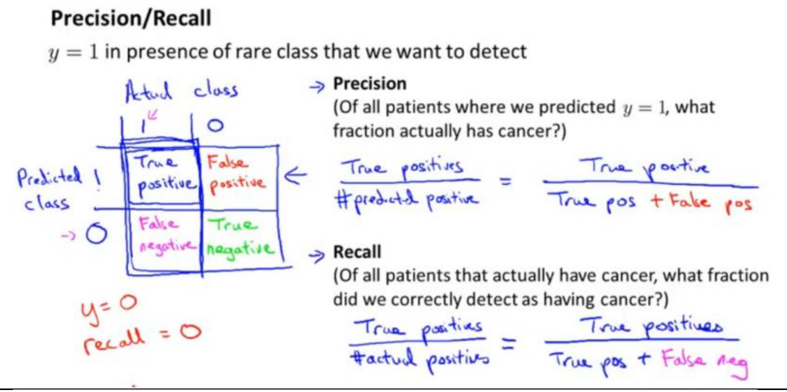
```
function y = predictCancer(x)
  y = 0; %ignore x!
return
```

→ 0.5% error

Skewed classes

The case in which we have much more examples from one class than from the other class. In these cases the cross validation error would not be a good metric for the performance of the model. If the CV error in a binary classification problem is 1% you might think it is good, but if only 0.5% of the training and cv sets is of one class, you could have an algorithm that just predicts 0 all the time and does better than your 1% error model.

In cases like these we need other metrics



Precision/Recall

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

		Actual class 1	0
Predicted class	1	True positive	False positive
	0	False negative	True negative

$y = 0$
recall $= 0$

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

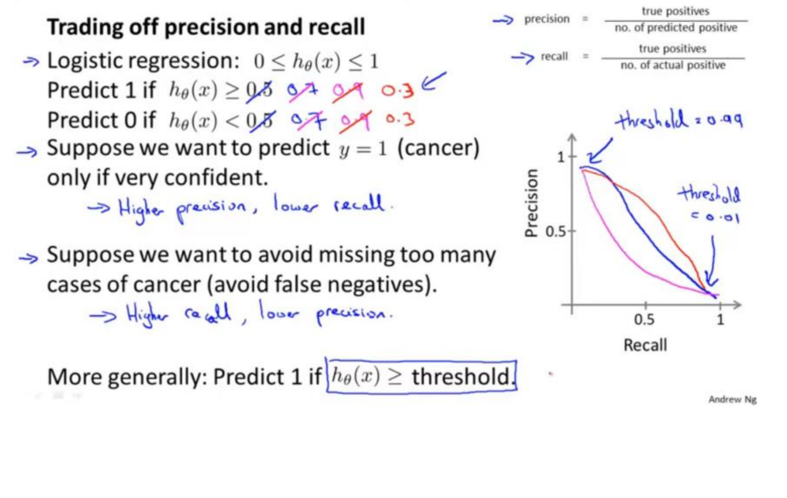
$$\frac{\text{True positives}}{\text{\#predicted positive}} = \frac{\text{True positive}}{\text{True pos} + \text{False pos}}$$

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

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

Precision = Positive Predictive Value = TP/ (TP + FP)
Recall = Sensitivity = TP / (TP + FN)

In the above example the algorithm that predicts always 0, would have a recall of 0, so we would know that it hasn't learned from the data even though it has a 0.5% error.



Trading off precision and recall

→ precision $= \frac{\text{true positives}}{\text{no. of predicted positive}}$

→ recall $= \frac{\text{true positives}}{\text{no. of actual positive}}$

→ Logistic regression: $0 \le h_\theta(x) \le 1$
Predict 1 if $h_\theta(x) \ge$ ~~0.5~~ ~~0.7~~ ~~0.9~~ 0.3
Predict 0 if $h_\theta(x) <$ ~~0.5~~ ~~0.7~~ ~~0.9~~ 0.3

→ Suppose we want to predict $y = 1$ (cancer) only if very confident.
→ Higher precision, lower recall.

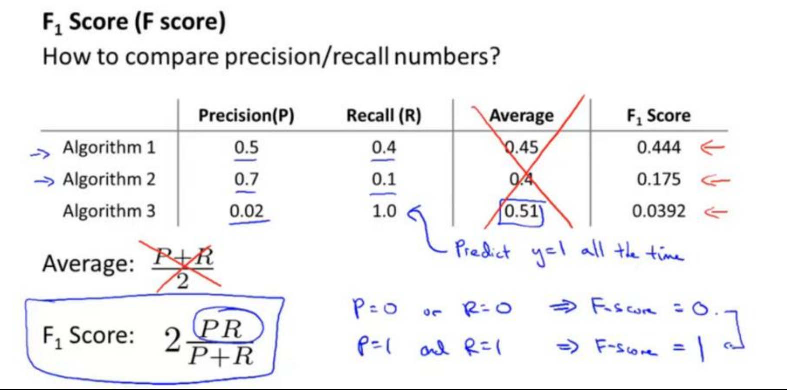
→ Suppose we want to avoid missing too many cases of cancer (avoid false negatives).
→ Higher recall, lower precision.

More generally: Predict 1 if $\boxed{h_\theta(x) \ge \text{threshold}}$.

Every classification algorithm exhibits a tradeoff between precision and recall. If you want to increase precision you have to reduce the false positives, which means that you will make your model harder to output a positive. This would mean though, that it would be easier to predict negative which will increase the false negatives.

If you want to predict 1 only if you are confident enough, then you might choose to predict 1 only if h(x)>0.7 for example instead of >0.5. Such a threshold would give a high precision but a low recall.

There is a single metric that is used in order to evaluate algorithms based on their precision and recall values. It is the F1 score.



F_1 Score (F score)

How to compare precision/recall numbers?

	Precision(P)	Recall (R)	~~Average~~	F_1 Score
→ Algorithm 1	0.5	0.4	~~0.45~~	0.444 ←
→ Algorithm 2	0.7	0.1	~~0.4~~	0.175 ←
Algorithm 3	0.02	1.0	~~0.51~~	0.0392 ←

Predict $y=1$ all the time

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

F_1 Score: $\boxed{2\frac{PR}{P+R}}$

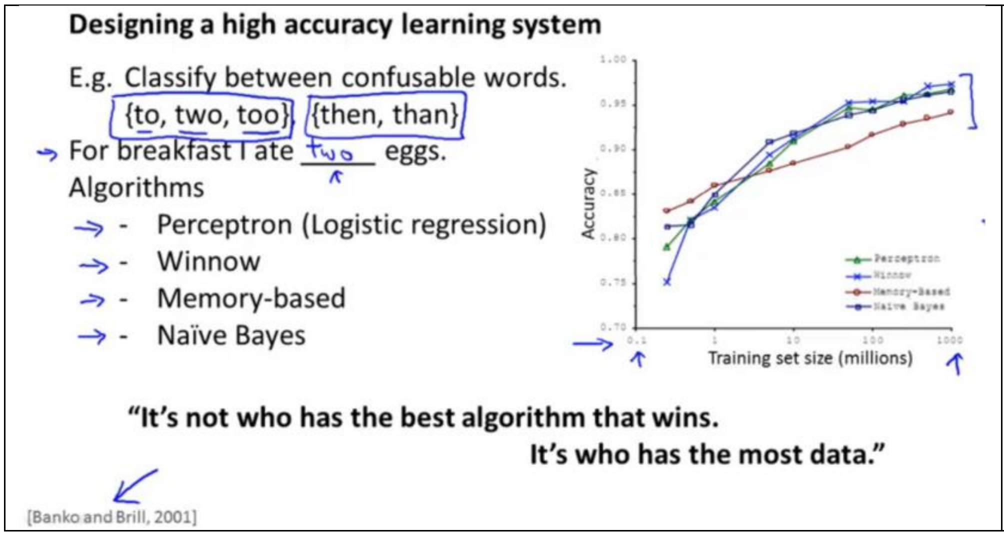
$P=0$ or $R=0$ ⇒ F-score $= 0$.
$P=1$ and $R=1$ ⇒ F-score $= 1$

F1 score
It short of gives a greater weight to the low value either this is recall or precision. If one of them is close to 0 then it would be close to 0 too.

<u>In practice you can try a range of different threshold values and evaluate their F1 score on the CV set to pick the one you want</u>.

The reason for which the F1 score is important in problems with skewed classes, is that in these cases it is difficult to grade the model based on the error rate. If the classes are not skewed and you have let's say a 50/50 case, then if the error rate on the test set is 1% you know the algorithm is good. Because if it always predicted 1 or 0 the error rate would be 50%. But if the classes are skewed like in the cancer case, then you must use the F1 score.

Common approach for structuring a good prediction model
In general, a way to think of structuring a good prediction model is first to ensure that the input features have sufficient information to predict the output accurately. A way to evaluate that is to show the features to a human expert on the field and ask them if they can confidently predict the output based on these features. Then you must think about the bias and variance of the model. First you must ensure that you have a low bias model by choosing one with many parameters and then ensure that this model will have a low variance too, by collecting a huge amount of data, so that the model would not be able to overfit.
(We already saw in which cases we might need additional data, so this is just a clarification)



Designing a high accuracy learning system

E.g. Classify between confusable words.
{to, two, too} {then, than}
For breakfast I ate $\underline{\text{two}}$ eggs.

Algorithms
- Perceptron (Logistic regression)
- Winnow
- Memory-based
- Naïve Bayes

"It's not who has the best algorithm that wins.
It's who has the most data."

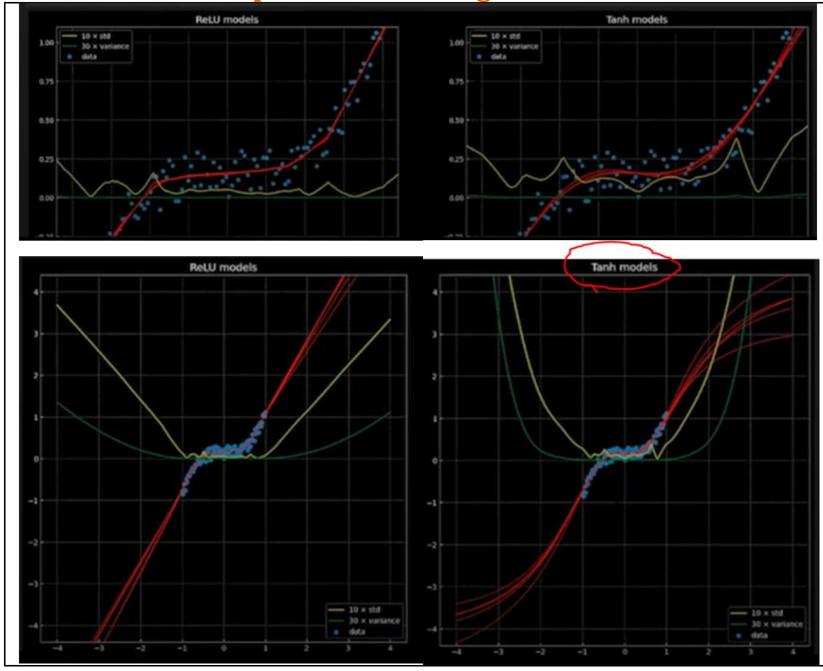
[Banko and Brill, 2001]

This research paper from Banko and Brill (2001) was very influential (although these algorithms are not currently used that much). Usually more data means better performance. But this is true only in some cases as we already saw. It is true when
1. The features have sufficient information for the output
2. The model has low bias
3. Then there is a meaning in trying in having a lot of data

Ceiling analysis

Ceiling analysis

Variance between predictions in regression

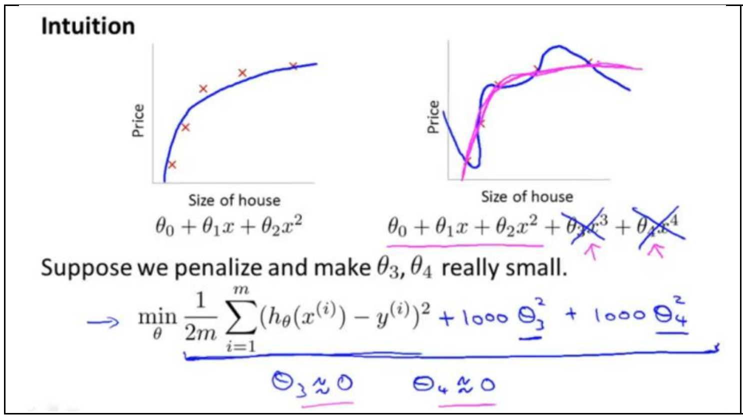


Each red line is a NN with different parameters. The green line is the variance of these different NNs with each other, the variance between multiple predictions essentially. **The smaller the variance the closer you are to the training cloud (the training manifold).** This is an important take away that can tell you how well you do. Because it is not that easy to know how well your regressor does. So you train multiple of them, and you check the variance between them. If it is small, or smaller than some other models then you might be good.

Notice that when using ReLU the out of data domain prediction is linear while if you use tanh or sigmoid it is not linear. I think you just extend the last part of the linear piecewise function in one case and the sigmoid like one in the other.

Regularization

Intro



Intuition

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

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

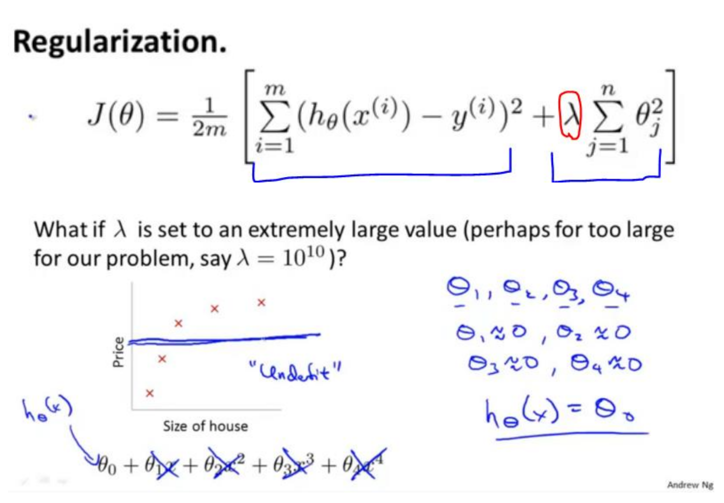
Suppose we penalize and make θ_3, θ_4 really small.

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

$\theta_3 \approx 0 \qquad \theta_4 \approx 0$

For an intuition of how regularization helps to address overfitting, we can think of a 2d hypothesis where we penalize θ3 and θ4, forcing them to have low values. This means that their contribution to the hypothesis function becomes small, leaving only θ0 and θ1 as important and this results to a curve that looks like a quadratic one with a few disturbances that fits well to the data.

We enforce parameters to become small by adding a big scalar for these in the cost function. So in order to minimize the cost function these parameters must be really small. (I guess we use the squared parameters so that the sign of the parameter makes no difference)



Regularization.

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

What if λ is set to an extremely large value (perhaps for too large for our problem, say $\lambda = 10^{10}$)?

$\theta_1, \theta_2, \theta_3, \theta_4$

$\theta_1 \approx 0,\ \theta_2 \approx 0$

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

$\underline{h_\theta(x) = \theta_0}$

Price
Size of house
"Underfit"
$h_\theta(x)$
$\theta_0 + \cancel{\theta_1 x} + \cancel{\theta_2 x^2} + \cancel{\theta_3 x^3} + \cancel{\theta_4 x^4}$
Andrew Ng

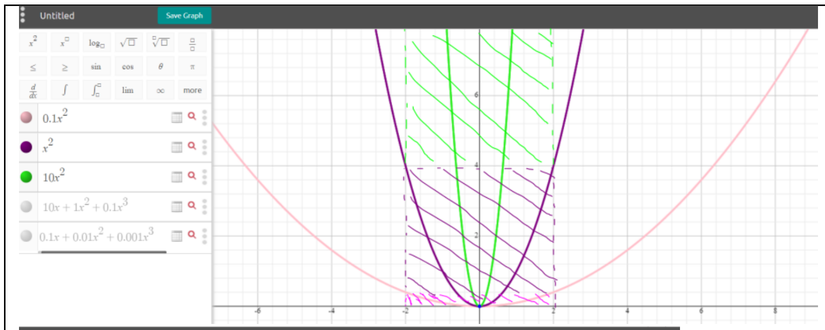
It can be shown that if you use small values for the parameters Θ of the hypothesis function, has the effect of producing smoother (simpler) hypothesis functions. A way to think of it is that by making all parameters small, if we have scaled feature values between 0 and 1, then the lower order parameters would be more important. If we have $h(x)=\theta0+\theta1x+\theta2x^2+...$ if x is 0.1 then x^2 is 0.01 which results in small contribution for θ2 to h(x).

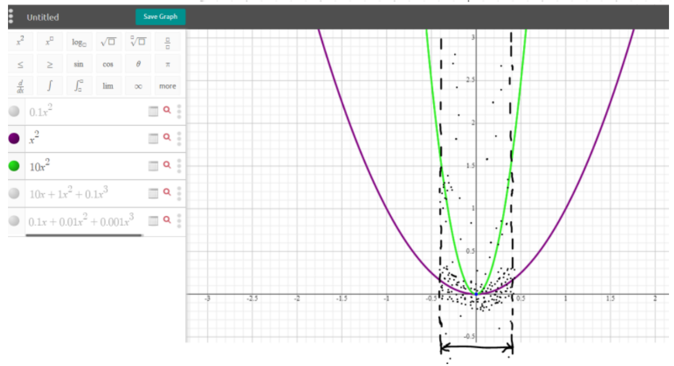
We regularize by adding an additional term in the cost function. The constant **λ** is called the **regularization parameter** and controls the magnitude of the parameters. Usually we don't regularize θ0.

Ultimately if all parameters become extremely small then the hypothesis function would be equal to θ0 which is a constant value that we usually don't regularize. So if we end up with a constant h(x) it is an indication that the parameters are too small and we have to choose them to be a little larger.

$A+\lambda B$ Essentially you minimize a function of that form. By controlling the magnitude of constant λ you control which of these two terms you think is more important. Λ actually controls the variance of your hypothesis. You select a tradeoff between how well you want your hypothesis to fit to the training data (A term more important than B) or how small you want the hypothesis parameters to be so that the hypothesis is not that flexible (B larger than A).

Some intuitions

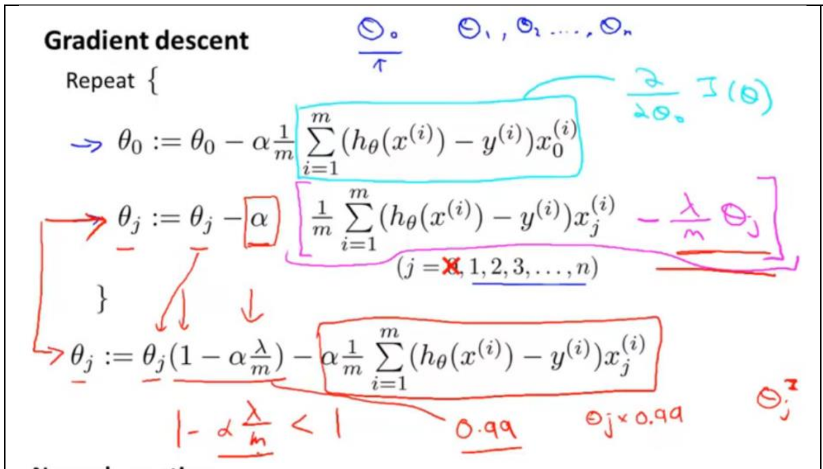




This might be an intuition on why smaller weights mean smoother functions and consequently less overfitting chances.
By decreasing the magnitude of the coefficients (regularizing) you decrease the size of the space that the hypothesis function defines. Thus the hypothesis function can't overfit to data outside of that space.
If your hypothesis can do everything from being a straight line to wiggling in every direction like a sine wave that can also go up and down, it's much more likely to pick up and model random perturbations in your data that isn't a result of the underlying signal but the result of just lucky chance in that data set

For given data within let's say -2 to 2 range, the small coefficient function has less variance while the large coefficient ones, can take larger values so they can overfit if there was data points with large values.

Regularized linear regression



Gradient descent

$\underset{\uparrow}{\theta_0}$ $\theta_1, \theta_2, \dots, \theta_n$

Repeat {

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

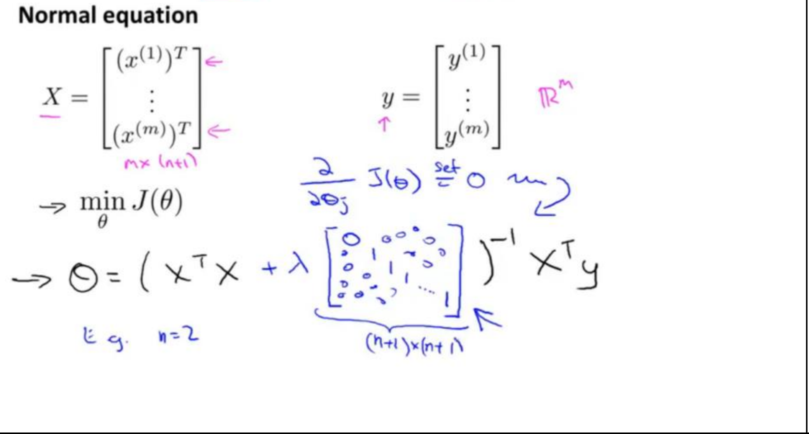
$$\rightarrow \theta_j := \theta_j - \boxed{\alpha} \left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} - \frac{\lambda}{m}\theta_j\right]$$

$$(j = \cancel{0}, \underline{1, 2, 3, \dots, n})$$

}

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

$1 - \alpha\frac{\lambda}{m} < 1$ 0.99 $\theta_j \times 0.99$ θ_j^2



Normal equation

$$X = \begin{bmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix} \leftarrow \quad m \times (n+1) \qquad y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \quad \mathbb{R}^m$$

$$\rightarrow \min_\theta J(\theta) \qquad \frac{\partial}{\partial\theta_j}J(\theta) \overset{set}{=} 0$$

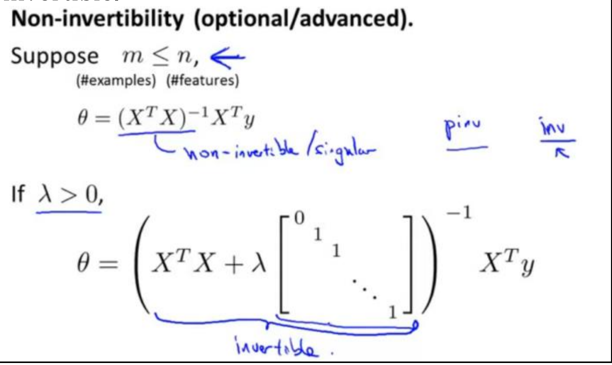
$$\rightarrow \Theta = \Big(X^TX + \lambda \underbrace{\begin{bmatrix} 0 & & & \\ & 1 & & \\ & & 1 & \\ & & & \ddots \\ & & & & 1 \end{bmatrix}}_{(n+1)\times(n+1)}\Big)^{-1} X^Ty$$

E.g. $n=2$

The partial derivative for the regularized cost function of a linear regression problem is given by the formula inside the pink brackets.

Doing algebra we end up with a more compact form which is actually the same as the standard linear regression with the difference that Θj is multiplied by a constant (1-αλ/m) which is usually smaller than 1 since α is small and m is large. So during regularized gradient descent parameters θ become smaller faster.

For the normal equation method the new form is as shown. Notice that with regularization you avoid the non invertibility problem too. It can be shown that in the regularized form since λ>0 the matrix is always invertible.



Non-invertibility (optional/advanced).

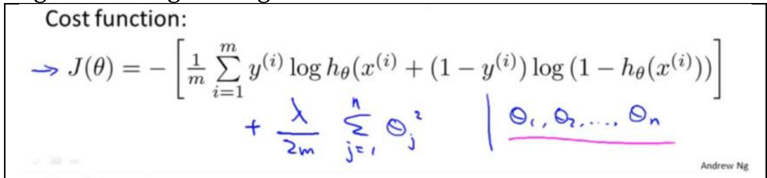
Suppose $m \leq n$, ←
(#examples) (#features)

$$\theta = \underbrace{(X^TX)^{-1}}_{\text{non-invertible / singular}} X^Ty \qquad \text{pinv} \qquad \text{inv}$$

If $\lambda > 0$,

$$\theta = \underbrace{\left(X^TX + \lambda\begin{bmatrix} 0 & & & \\ & 1 & & \\ & & 1 & \\ & & & \ddots \\ & & & & 1 \end{bmatrix}\right)}_{\text{invertible.}}^{-1} X^Ty$$

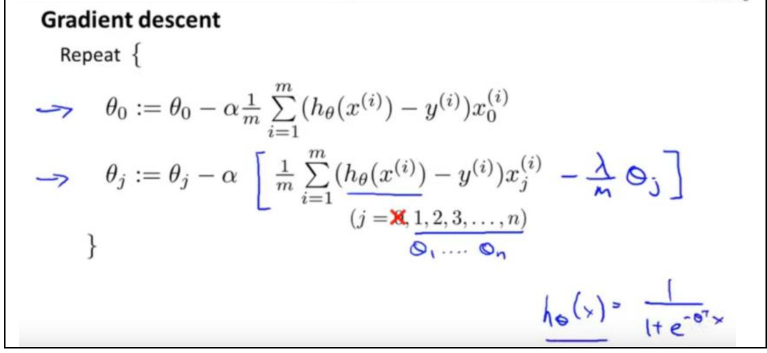
Regularized logistic regression



Cost function:

$$\rightarrow J(\theta) = -\left[\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log h_\theta(x^{(i)}) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \quad \Big| \ \underline{\theta_1, \theta_2, \dots, \theta_n}$$

Andrew Ng



Gradient descent

Repeat {

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

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

$$(j = \cancel{0}, \underline{1, 2, 3, \dots, n})$$
$$\theta_1, \dots, \theta_n$$

}

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

The formula for partial derivative of the cost function

Norms

L0 norm
The number of components of a vector that are non zero

L1 norm
The sum of the absolute values of the components of the vector. In context of DL minimizing L1 norm of a vector is called sparse coding, it makes the vector more sparse. It is a convex function so you can run gradient based optimization. It is differentiable. Minimizing L1 norm implicitly minimizes L0 norm (which is a non convex problem).

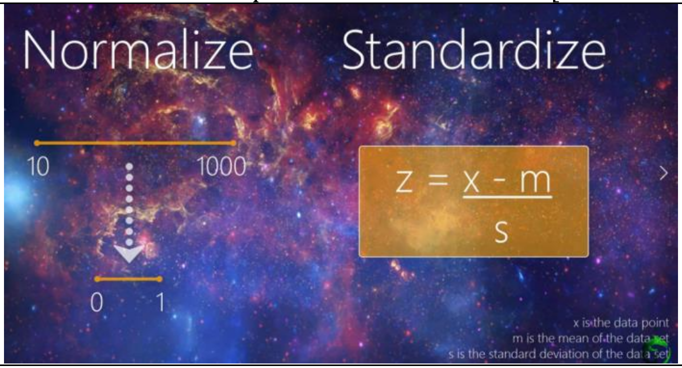
L2 norm
Sum of the squares of vector components, and the get the square root of this.

In Practice

All these are regularization techniques, which means they make the weights small.

Data normalization or Data Standardization

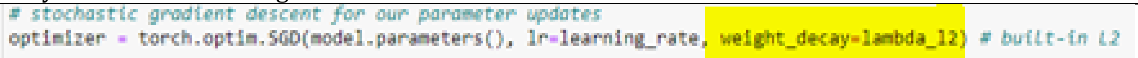
In the data preprocessing step, we normalize or standardize the data. Both operations have the target of putting all features on the same scale. If the input data has four features [x1 x2 x3 x4] then these all x1s and all x2s etc. will lie within the same range.


Normalize
Standardize
10
1000
0
1
z = (x - m) / s
x is the data point
m is the mean of the data set
s is the standard deviation of the data set

Normalized data lie between 0 and 1
Standardized data have mean=0 and std=1 (do if x_i follows a normal distribution all x_{is} will lie roughly between -1.5 and +1.5 (within 3 stds)

Weight decay

Usually L2 norm of the weight matrix

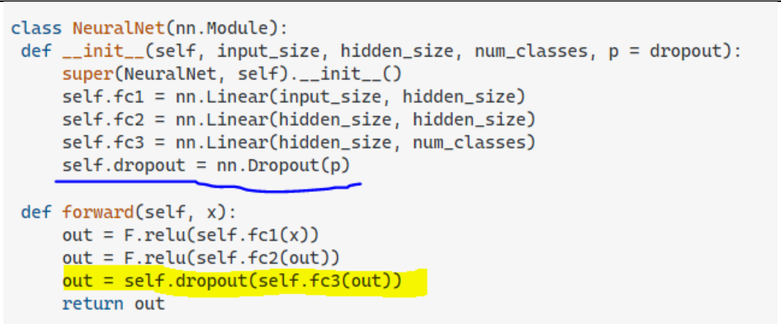


```
# stochastic gradient descent for our parameter updates
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=lambda_l2) # built-in L2
```

Defined in the optimizer setup

Dropout

Only active during training mode (net.train() in pytorch). At each pass there is a probability for neurons to be dropped out (not existing). So the network has to learn more robustly (I guess it forces the model to have some redundancy on neurons, so a feature might be represented by more than one neurons so when one is shut off the other representing that feature might be on??? and they learn to represent only the most important features)
At training a neuron has probability p of dropout.
At testing we multiply weights outgoing from the hidden neurons by 1-p.
So if it had a probability of 70% being off during training (so it was shut off 70% of the time), then it contributes for 30% of its value on test.



```
class NeuralNet(nn.Module):
 def __init__(self, input_size, hidden_size, num_classes, p = dropout):
     super(NeuralNet, self).__init__()
     self.fc1 = nn.Linear(input_size, hidden_size)
     self.fc2 = nn.Linear(hidden_size, hidden_size)
     self.fc3 = nn.Linear(hidden_size, num_classes)
     self.dropout = nn.Dropout(p)

 def forward(self, x):
     out = F.relu(self.fc1(x))
     out = F.relu(self.fc2(out))
     out = self.dropout(self.fc3(out))
     return out
```

You select to which layer the Dropout is applied.

Have in mind that if you use dropout of 20% then in order to keep the norm of the vector constant despite the 20% reduction of components, you need to increase the magnitude of the other components by 1/(1-0.2)=1.25

Batch normalization

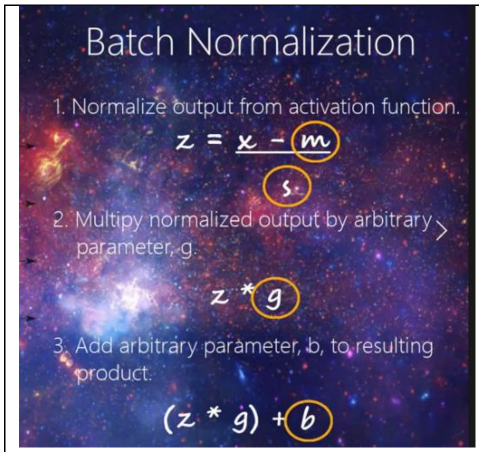
https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739 good explanation and practical tips
https://blog.janestreet.com/l2-regularization-and-batch-norm/ batch norm vs weight decay

normalize

scale and shift
Unlike the input layer, which requires all normalized values to have zero mean and unit variance, Batch Norm allows its values to be shifted (to a different mean) and scaled (to a different variance). It does this by multiplying the normalized values by a factor, gamma, and adding to it a factor, beta. Note that this is an element-wise multiply, not a matrix multiply.

What makes this innovation ingenious is that these factors are not hyperparameters (ie. constants provided by the model designer) but are trainable parameters that are learned by the network. In other words, each Batch Norm layer is able to optimally find the best factors for itself, and can thus shift and scale the normalized values to get the best predictions.

Moving average of means and std
In addition, Batch Norm also keeps a running count of the Exponential Moving Average (EMA) of the mean and variance. During training, it simply calculates this EMA but does not do anything with it. At the end of training, it simply saves this value as part of the layer's state, for use during the Inference phase.
In inference we have only one sample. So to normalize it we use the EMA mean and std stored in the model.



Batch Normalization

1. Normalize output from activation function.

$$z = \frac{x - m}{s}$$

2. Multipy normalized output by arbitrary parameter, g.

$$z * g$$

3. Add arbitrary parameter, b, to resulting product.

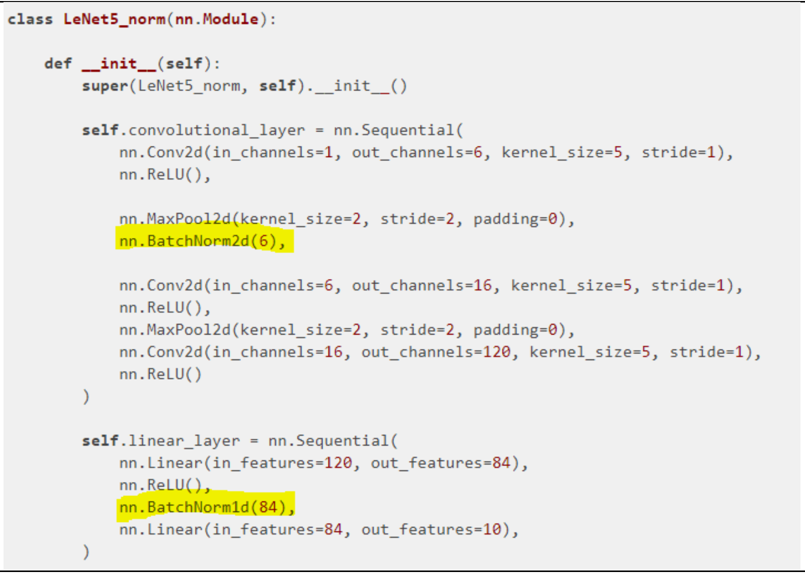
$$(z * g) + b$$

This is what batch norm does. These 4 parameters are all learnable during training. Operations 2 and 3 set a new std and mean for the data. This happens per batch. So in a layer with 4 neurons (4 hidden features x1, x2, x3, x4) and batch size of 256, we normalize x1 using its 256 values (calculating its mean and std). same for the other features.

This process makes the weights not become imbalanced, some being vary large and some very small.
Increases training speed.

Batch norm Vs weight decay
When used together with batch normalization in a convolutional neural net with typical architectures, an L2 objective penalty no longer has its original regularizing effect. Instead it becomes essentially equivalent to an adaptive adjustment of the learning rate!

So you either use one or the other I guess



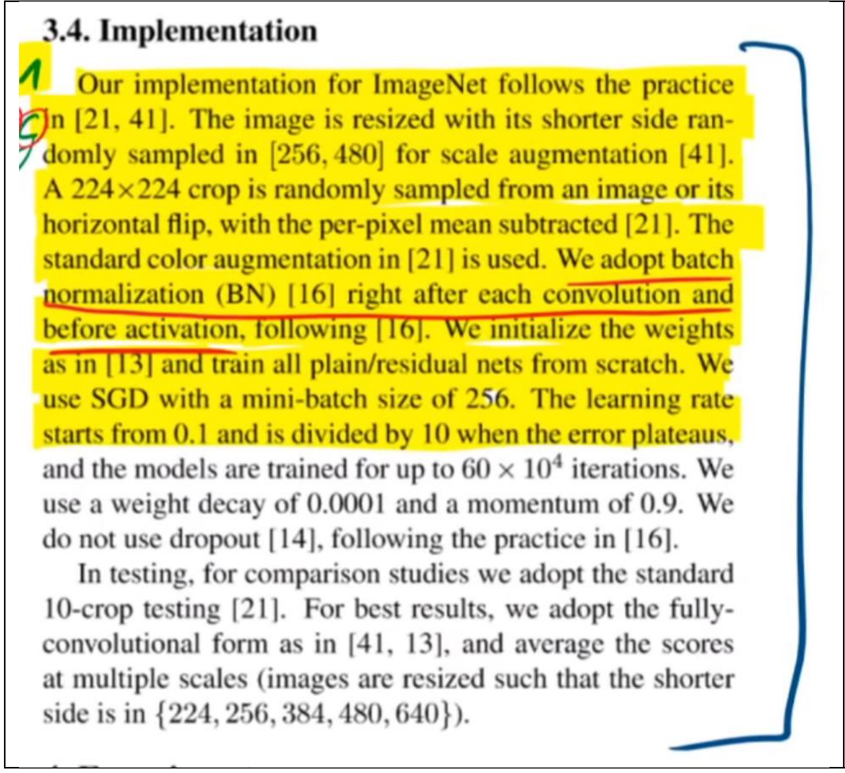
```
class LeNet5_norm(nn.Module):

    def __init__(self):
        super(LeNet5_norm, self).__init__()

        self.convolutional_layer = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1),
            nn.ReLU(),

            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
            nn.BatchNorm2d(6),

            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, stride=1),
            nn.ReLU()
        )

        self.linear_layer = nn.Sequential(
            nn.Linear(in_features=120, out_features=84),
            nn.ReLU(),
            nn.BatchNorm1d(84),
            nn.Linear(in_features=84, out_features=10),
        )
```

In pytorch you define where you want the batch norm to take place. After which layers.

Tips

We use Keras callbacks to implement:

- Learning rate decay if the validation loss does not improve for 5 continues epochs.
- Early stopping if the validation loss does not improve for 10 continues epochs.
- Save the weights only if there is improvement in validation loss.



3.4. Implementation

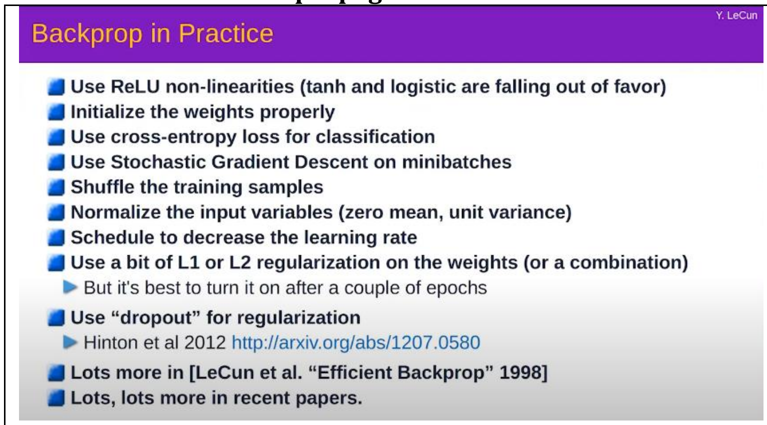
Our implementation for ImageNet follows the practice in [21, 41]. The image is resized with its shorter side randomly sampled in [256, 480] for scale augmentation [41]. A 224×224 crop is randomly sampled from an image or its horizontal flip, with the per-pixel mean subtracted [21]. The standard color augmentation in [21] is used. We adopt batch normalization (BN) [16] right after each convolution and before activation, following [16]. We initialize the weights as in [13] and train all plain/residual nets from scratch. We use SGD with a mini-batch size of 256. The learning rate starts from 0.1 and is divided by 10 when the error plateaus, and the models are trained for up to 60×10^4 iterations. We use a weight decay of 0.0001 and a momentum of 0.9. We do not use dropout [14], following the practice in [16].

In testing, for comparison studies we adopt the standard 10-crop testing [21]. For best results, we adopt the fully-convolutional form as in [41, 13], and average the scores at multiple scales (images are resized such that the shorter side is in {224, 256, 384, 480, 640}).

Many of the tricks used in DL in this paragraph.

- If you change the batch size you need to find a different set of hyperparameters for your model too.
- A higher batch size often has a similar effect to lowering learning rate.
- After some epochs the cost starts increasing and you have to decrease the learning rate to overcome this region.
- Dropout?
- Batch normalization?
- Example of different data distribution between the evaluation and the training set: images in evaluation set were all unique, while training set could contain same images (but with different captions). I added more training data along the way to overcome this and make the error rate decrease further.
- Then @_arohan_who had been suggesting great ideas along the way mentioned that I should try to train in full precision, and update beta1 momentum (to overcome the cost plateau)

Tricks for efficient backpropagation



Y. LeCun

Backprop in Practice

- Use ReLU non-linearities (tanh and logistic are falling out of favor)
- Initialize the weights properly
- Use cross-entropy loss for classification
- Use Stochastic Gradient Descent on minibatches
- Shuffle the training samples
- Normalize the input variables (zero mean, unit variance)
- Schedule to decrease the learning rate
- Use a bit of L1 or L2 regularization on the weights (or a combination)
  - But it's best to turn it on after a couple of epochs
- Use "dropout" for regularization
  - Hinton et al 2012 http://arxiv.org/abs/1207.0580
- Lots more in [LeCun et al. "Efficient Backprop" 1998]
- Lots, lots more in recent papers.

L1 and L2 regularization. We add a term to the cost with the effect of making the weights smaller at every iteration.

Dropout
In pytorch dropout is implemented as a layer. It is applied to the output of a layer, and it randomly makes some of its activations 0. This forces the network to not distribute the information of some

Weight initialization
The concept is that the weights are initialized randomly in a way that if a unit has many inputs the weights are smaller than if it had few inputs. Usually the inputs are normalized (0 mean unit variance). So the weighted sum increases by the sqrt(number of input units). We want the weighted sum to be fairly the same size both when there are many and few inputs and we want it to be fairly in the same size with each inputs (0 mean unit variance). So the weights become smaller by a factor of 1/sqrt(nOfUnits).

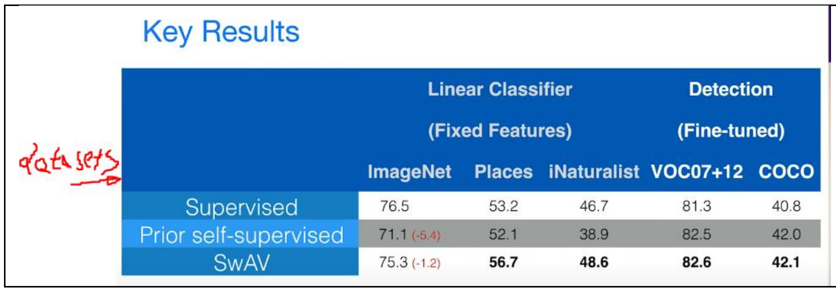
This is built into pytorch. There are a few options (He, Xavier, LeCun) but they are all based to the same concept.

- He initialization works better for layers with ReLu activation.
- Xavier initialization works better for layers with sigmoid activation.

patterns in many nodes and become more robust. It has been shown that dropout can decrease the generalization error in many cases. But it might not be always necessary.

There are many variations of SGD. the learning rate can be a scalar, or a diagonal matrix, or a full matrix. It can be constant, or usually it is decreased according to a schedule (pytorch has many built in schedules). ADAMS is a kind of SVG where the learning rate is a diagonal matrix the elements of which change in every step. In the optim package of pytorch there are a lot of them to choose from.

Datasets

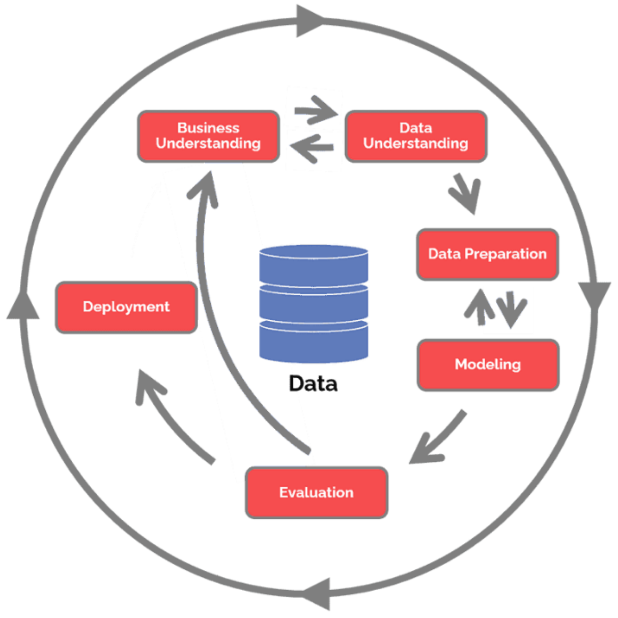


Key Results

	Linear Classifier (Fixed Features)			Detection (Fine-tuned)	
	ImageNet	Places	iNaturalist	VOC07+12	COCO
Supervised	76.5	53.2	46.7	81.3	40.8
Prior self-supervised	71.1 (-5.4)	52.1	38.9	82.5	42.0
SwAV	75.3 (-1.2)	**56.7**	**48.6**	**82.6**	**42.1**

CRISP DM

CRoss Industry Standard Process for Data Mining (CRISP-DM)



The CRoss Industry Standard Process for Data Mining (CRISP-DM) is a process model that serves as the base for a data science process. It has six sequential phases:

1. Business understanding – What does the business need?
2. Data understanding – What data do we have / need? Is it clean?
3. Data preparation – How do we organize the data for modeling?
4. Modeling – What modeling techniques should we apply?
5. Evaluation – Which model best meets the business objectives?
6. Deployment – How do stakeholders access the results?
7. Published in 1999 to standardize data mining processes across industries, it has since become the most common methodology for data mining, analytics, and data science projects.

Data science teams that combine a loose implementation of CRISP-DM with overarching team-based agile project management approaches will likely see the best results.

Active learning

Learning rate schedule

You choose one schedule like cosine annealing.

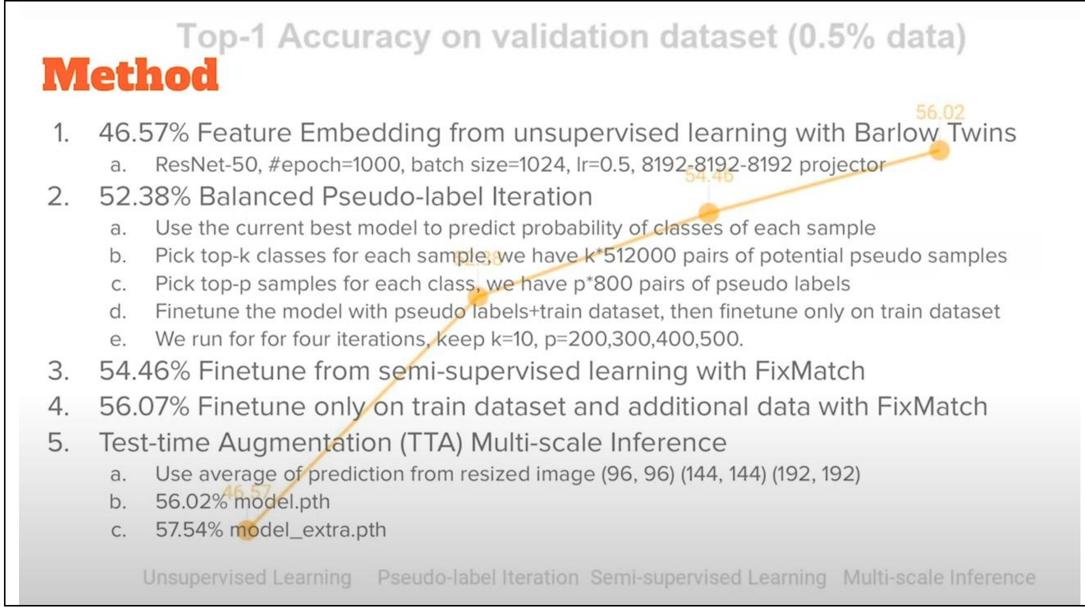
Pseudo labeling

Test-time augmentation

Model size, batch size and GPU

Could not go with deeper model than ResNet-34 because it wouldn't fir on a single GPU with the large batch size input

Some Methods overview

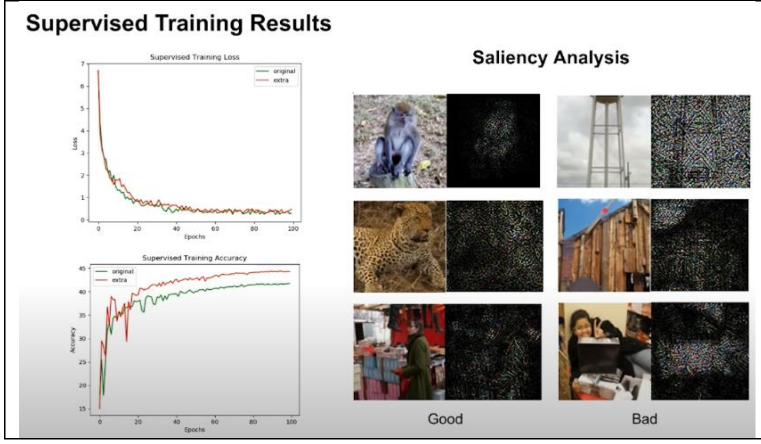
	Evaluating 5 different methods or models.

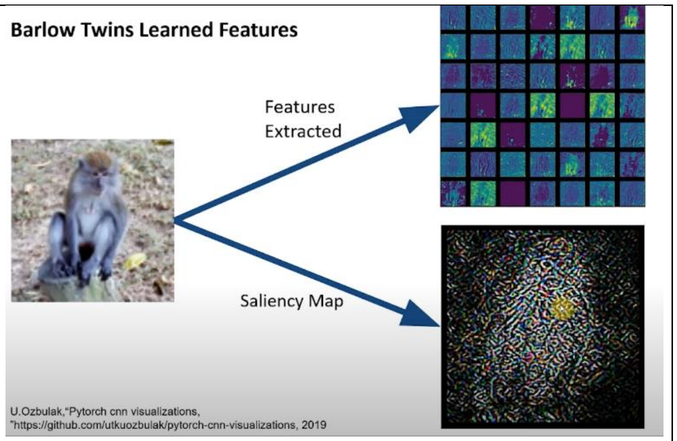
Top-1 Accuracy on validation dataset (0.5% data)

Method

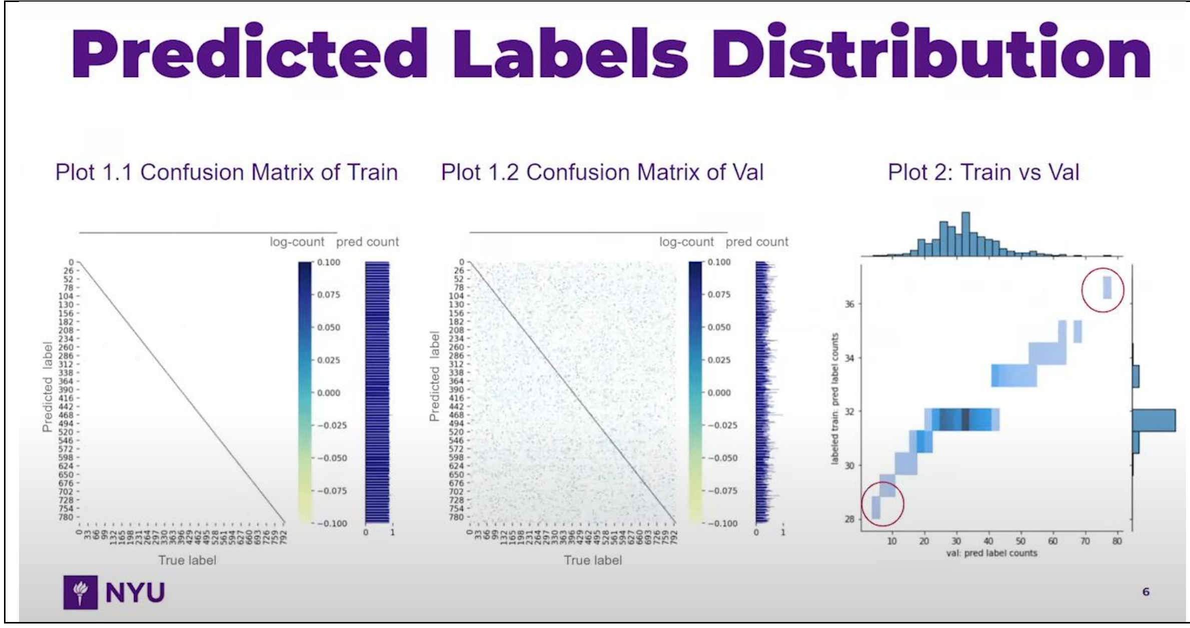
1. 46.57% Feature Embedding from unsupervised learning with Barlow Twins
   a. ResNet-50, #epoch=1000, batch size=1024, lr=0.5, 8192-8192-8192 projector
2. 52.38% Balanced Pseudo-label Iteration
   a. Use the current best model to predict probability of classes of each sample
   b. Pick top-k classes for each sample, we have k*512000 pairs of potential pseudo samples
   c. Pick top-p samples for each class, we have p*800 pairs of pseudo labels
   d. Finetune the model with pseudo labels+train dataset, then finetune only on train dataset
   e. We run for for four iterations, keep k=10, p=200,300,400,500.
3. 54.46% Finetune from semi-supervised learning with FixMatch
4. 56.07% Finetune only on train dataset and additional data with FixMatch
5. Test-time Augmentation (TTA) Multi-scale Inference
   a. Use average of prediction from resized image (96, 96) (144, 144) (192, 192)
   b. 56.02% model.pth
   c. 57.54% model_extra.pth

Saliency analysis

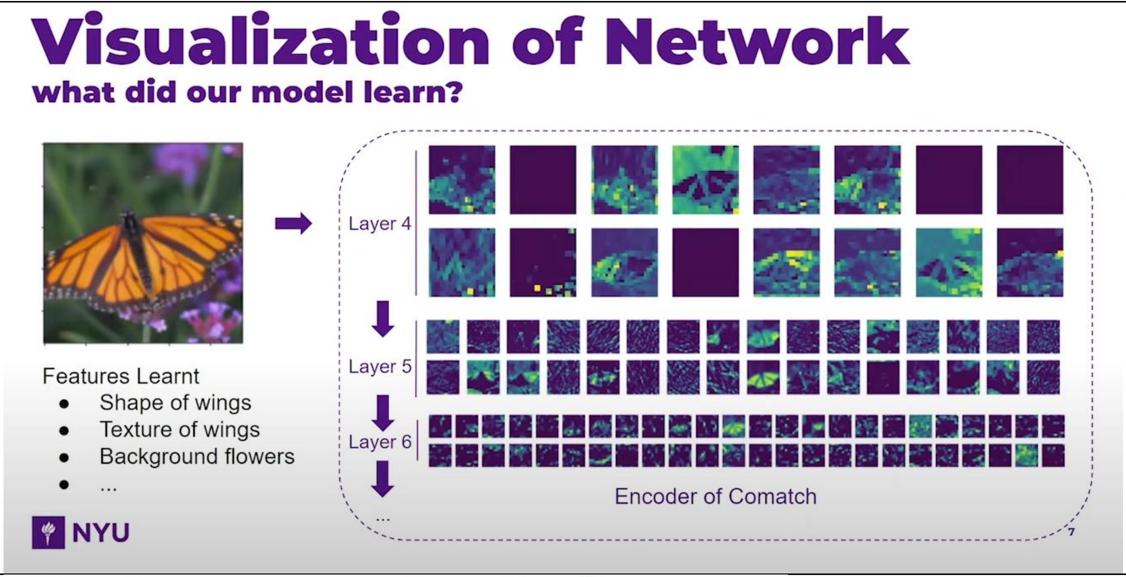
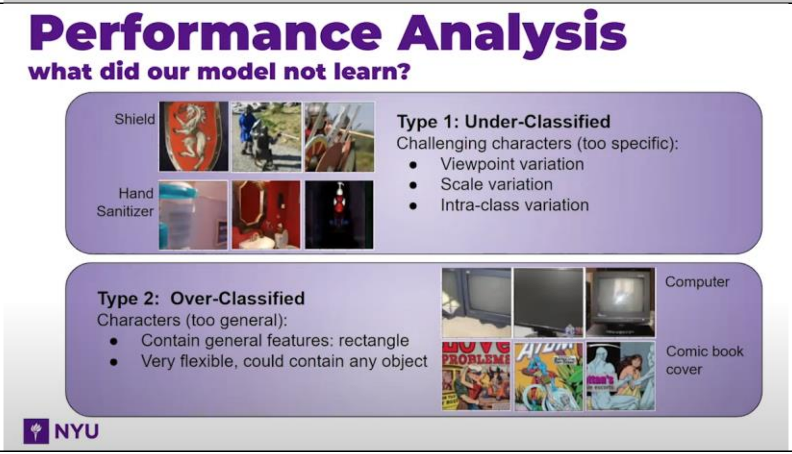



Barlow Twins Learned Features
Features Extracted
Saliency Map
U.Ozbulak,"Pytorch cnn visualizations, "https://github.com/utkuozbulak/pytorch-cnn-visualizations, 2019

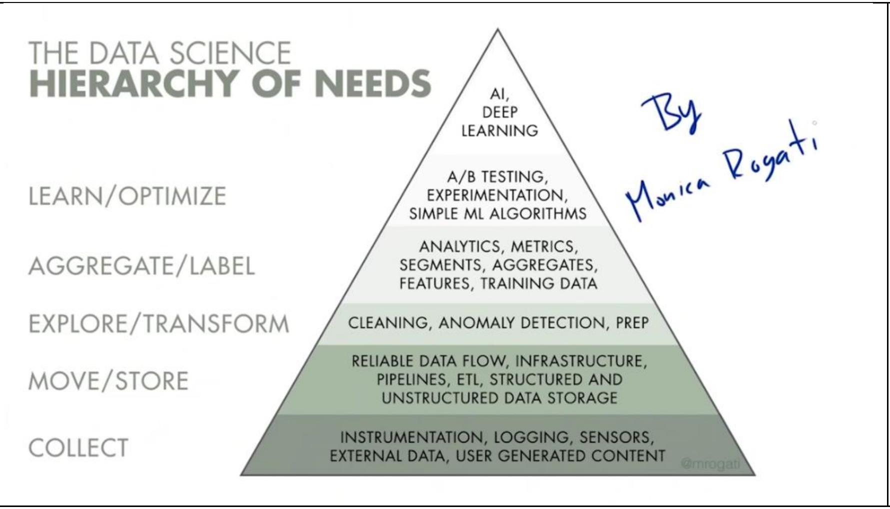
Confusion matrix



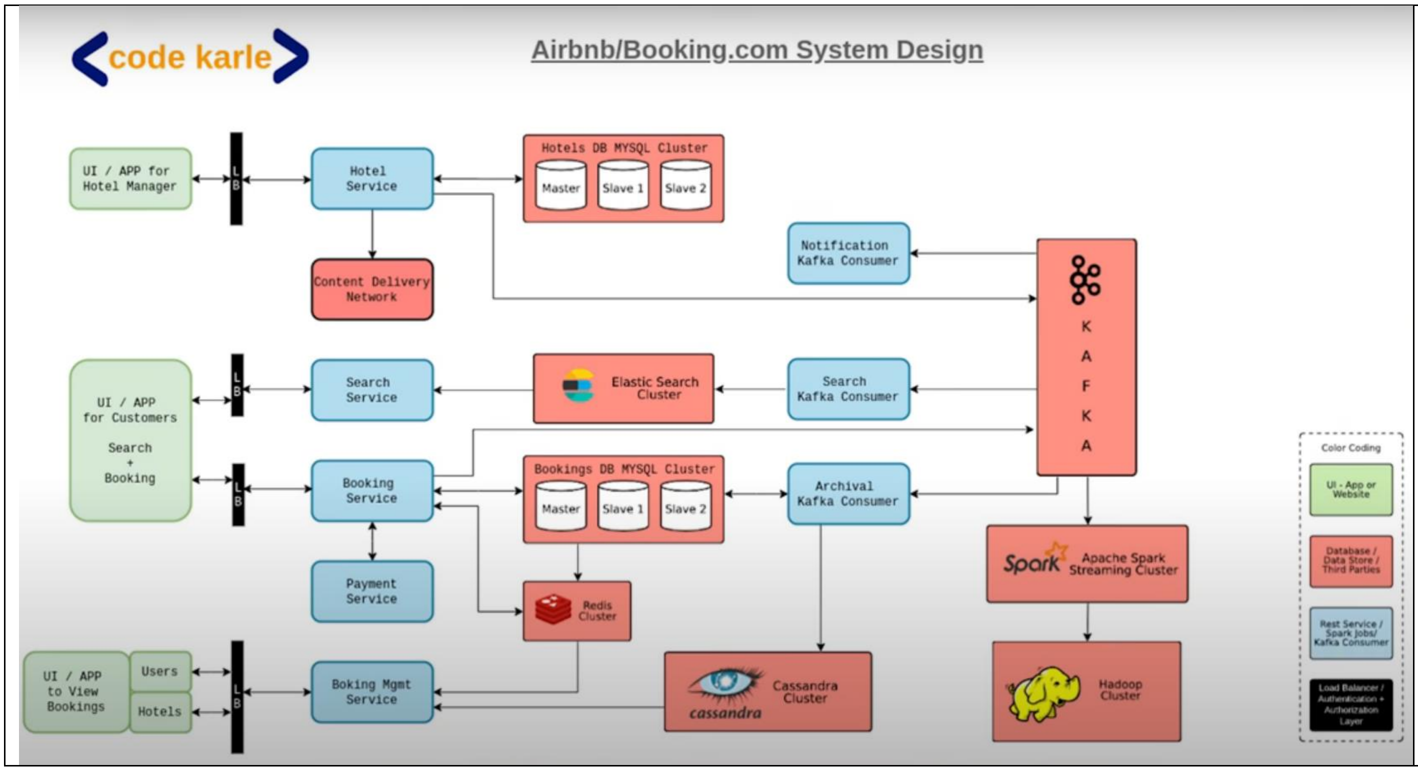
What we learned what we didn't

 **Visualization of Network** **what did our model learn?** Features Learnt • Shape of wings • Texture of wings • Background flowers • ... Layer 4 Layer 5 Layer 6 ... Encoder of Comatch NYU	Visualization of feature maps
 **Performance Analysis** **what did our model not learn?** **Type 1: Under-Classified** Challenging characters (too specific): • Viewpoint variation • Scale variation • Intra-class variation **Type 2: Over-Classified** Characters (too general): • Contain general features: rectangle • Very flexible, could contain any object Shield Hand Sanitizer Computer Comic book cover NYU	

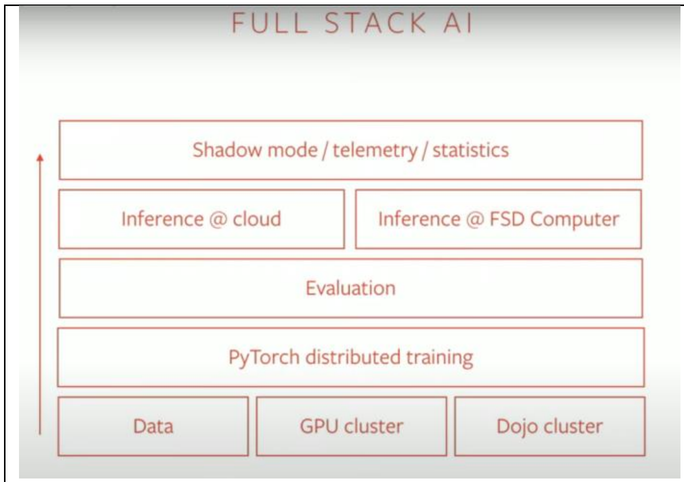
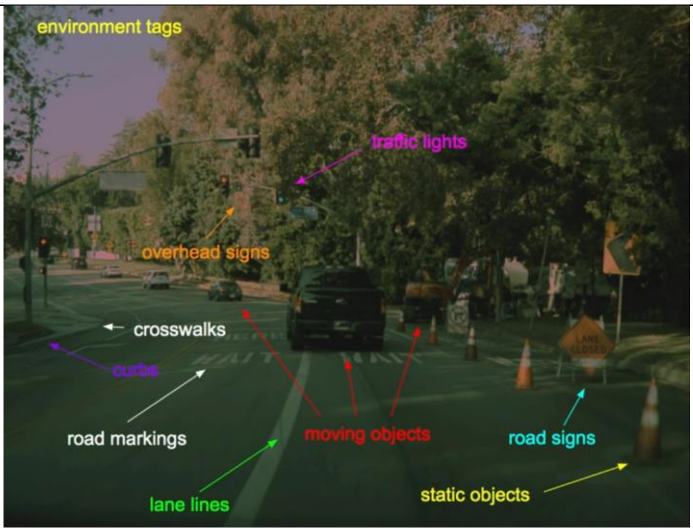
Industry tips

 THE DATA SCIENCE HIERARCHY OF NEEDS LEARN/OPTIMIZE AGGREGATE/LABEL EXPLORE/TRANSFORM MOVE/STORE COLLECT AI, DEEP LEARNING A/B TESTING, EXPERIMENTATION, SIMPLE ML ALGORITHMS ANALYTICS, METRICS, SEGMENTS, AGGREGATES, FEATURES, TRAINING DATA CLEANING, ANOMALY DETECTION, PREP RELIABLE DATA FLOW, INFRASTRUCTURE, PIPELINES, ETL, STRUCTURED AND UNSTRUCTURED DATA STORAGE INSTRUMENTATION, LOGGING, SENSORS, EXTERNAL DATA, USER GENERATED CONTENT @mrogati By Monica Rogati	

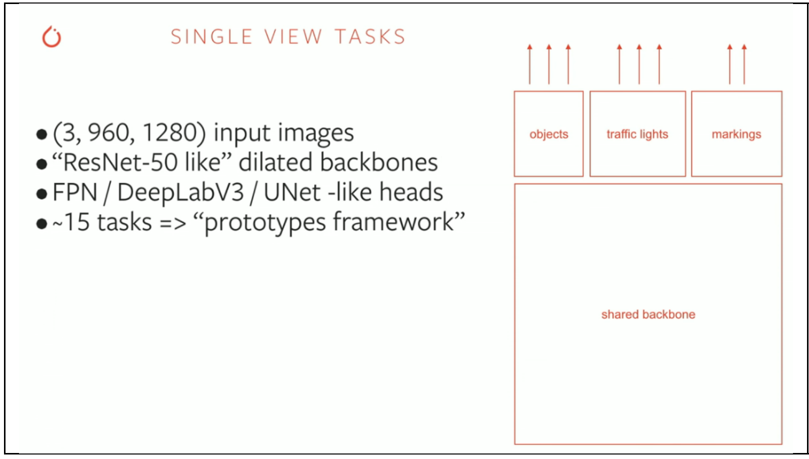
Airbnb



Tesla

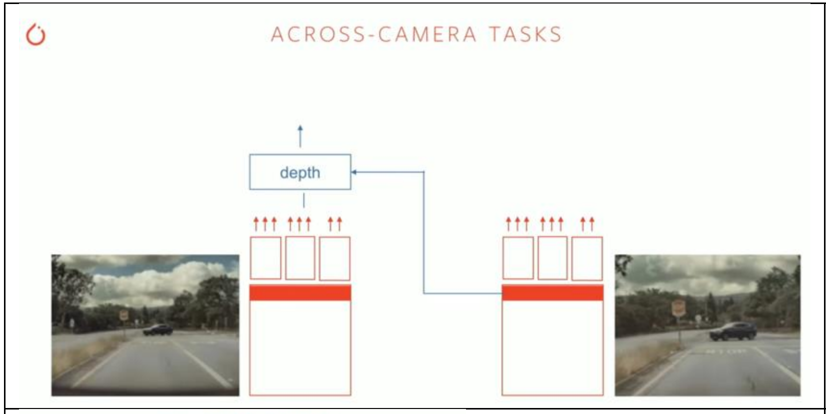
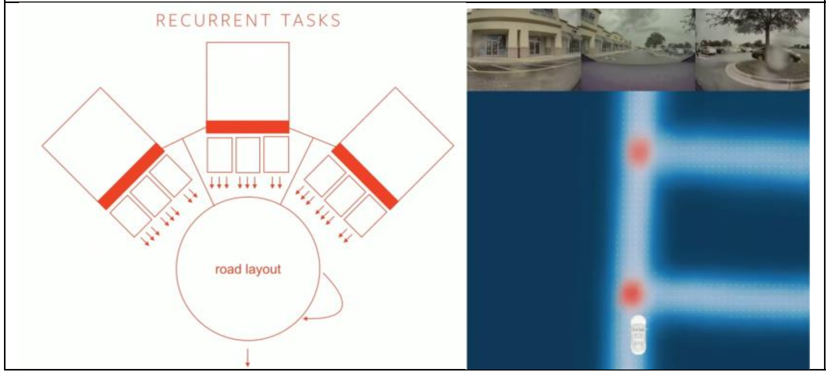
 FULL STACK AI Shadow mode / telemetry / statistics Inference @ cloud Inference @ FSD Computer Evaluation PyTorch distributed training Data GPU cluster Dojo cluster	 environment tags traffic lights overhead signs crosswalks curbs road markings moving objects road signs lane lines static objects ~1000px*1000px images running on ResNet-50 like models (2019)

HydraNets (shared backbone multiple heads)
They can't afford to have a NN for each individual task because the number of tasks are too many (~100) so they have to amortize some of the computation and put some of the tasks on shared backbones. They call these networks hydra nets (shared backbone with multiple heads)

 SINGLE VIEW TASKS • (3, 960, 1280) input images • “ResNet-50 like” dilated backbones • FPN / DeepLabV3 / UNet -like heads • ~15 tasks => “prototypes framework” 	DeepLabVV3 is a model architecture for semantic segmentation UNet is a model architecture for semantic segmentation FPN is a model architecture for object detection

There are single view tasks and across camera tasks. In the former the NN has to predict from a single image. In the later it has to process the scene from multiple cameras (for example estimating depth). Depth of each individual pixel. You have to borrow features from other hydra nets. So you have one hydra net for each camera. If you need more than one camera inputs for a specific task, you have to combine the output features of multiple hydra nets and add an additional layer of processing on top of them (optionally recurrent)

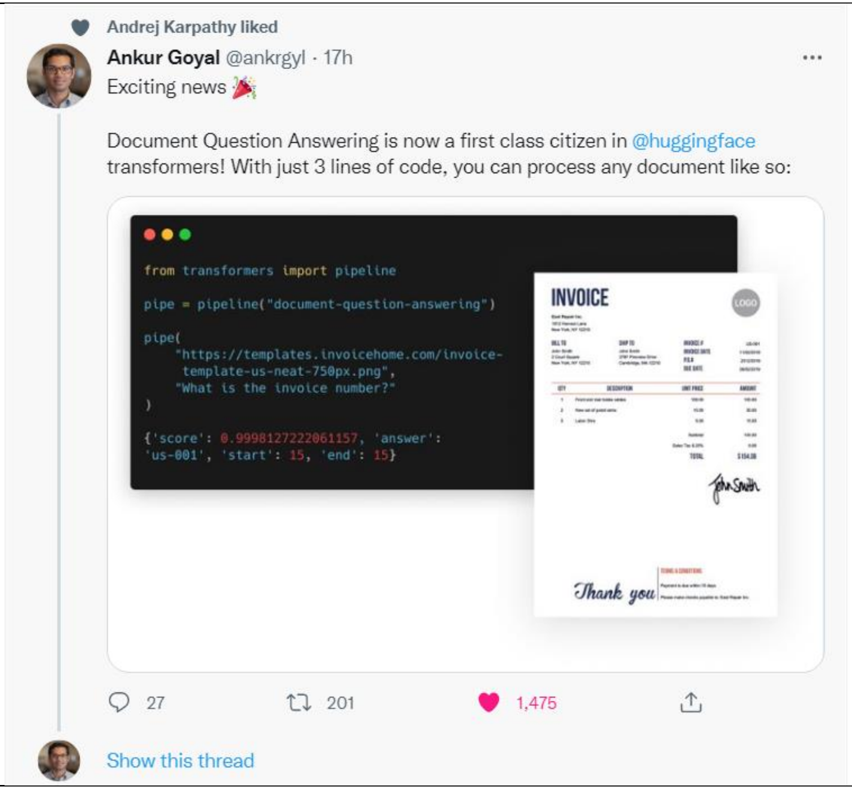
Across camera tasks

 ACROSS-CAMERA TASKS 	Predict the depth (depth network)
 RECURRENT TASKS 	Predict the road layout (layout network) The input is from 3 cameras (out of the 8 a tesla car has). From these 3 they predict the road layout. Now the networks predictions are not in image space but in top down space. So the stitching up of the 3 cameras happens inside of this RNN (while the pave ways edges where stitched up from individual images predictions “manually” in c++ code)

RNNs on videos

Deep Understanding Tesla FSD (5 parts)
https://saneryee-studio.medium.com/deep-understanding-tesla-fsd-part-1-hydranet-1b46106d57

Code

	Hugginface

Andrej Karpathy liked

Ankur Goyal @ankrgyl · 17h

Exciting news 🎉

Document Question Answering is now a first class citizen in @huggingface transformers! With just 3 lines of code, you can process any document like so:

```
from transformers import pipeline

pipe = pipeline("document-question-answering")

pipe(
    "https://templates.invoicehome.com/invoice-
     template-us-neat-750px.png",
    "What is the invoice number?"
)

{'score': 0.9998127222061157, 'answer':
'us-001', 'start': 15, 'end': 15}
```

27 201 1,475

Show this thread

Benchmarks

BIG-bench and BIG-bench Lite (2021)

The Beyond the Imitation Game Benchmark (BIG-bench) is a collaborative benchmark intended to probe large language models and extrapolate their future capabilities. The more than 200 tasks included in BIG-bench are summarized by keyword here, and by task name here

BIG-bench Lite (BBL) is a small subset of 24 diverse JSON tasks from BIG-bench. It is designed to provide a canonical measure of model performance, while being far cheaper to evaluate than the full set of more than 200 programmatic and JSON tasks in BIG-bench.

https://github.com/google/BIG-bench

Vision

ImageNet

Speech

LibriSpeech

Text (NLP)

GLUE benchmark suite