

Machine Learning (Andrew Ng, Stanford University)

Source: <https://www.coursera.org/learn/machine-learning/home/welcome>

🔗 Introduction: Types of Machine Learning problems

Supervised Learning

We try to learn from a **training dataset** that we assume is "correct" the parameters for best classification or regression. For that, the algorithm will take into account the combination of variables that are fed. The way in which the features are mathematically processed depends on the kernel function of the ML algorithm.

Unsupervised Learning

Conversely, if the dataset is unlabeled, we try to find structure within the data. For example, clustering algorithms.

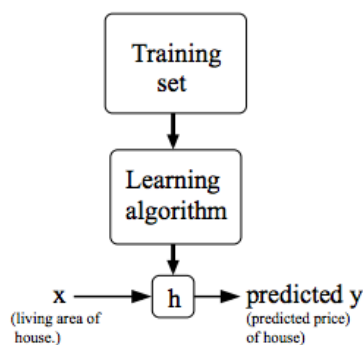
Linear Regression with One Variable

Model and Cost Function

- Model

We represent the model with **m** to denote the number of training examples, **x** to denote the "input" variable/features and **y** to denote the "output"/"target" variable. We feed the *training set* to a learning algorithm that generates a **hypothesis function** that relates the variables in the way we desire.

$$y = h(x_{\text{training set}})$$

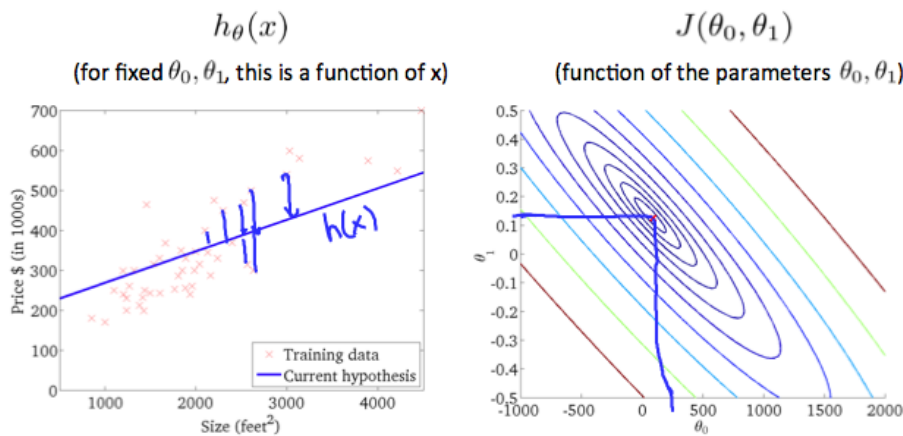


- Cost Function

The **cost function** is used to measure the accuracy of our hypothesis function during optimization of the parameters to training data. For instance, if the hypothesis is linear -we want to fit a linear model to data-, the parameters will be slope and intercept. Usually, we **minimize** the sum squared errors between the training examples and the function of the training examples.

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

We can make a *contour plot* to represent the value of the cost function at each combination of parameters. Lines of the same color correspond to the same cost function value but with different parameter combinations. With two parameters, the optimal combination corresponds to the center of the circle.



Parameter learning

- Gradient descent

Machine learning algorithm to minimize an arbitrary function

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

. We start with some combination of parameters and keep changing them while the function reduces. We take little steps to go down the steepest slope until convergence. With the limitations that finding the global minimum is not guaranteed, and that we will need to select a **learning rate or step size** (α) and the number of **parameter combinations** that can be tested.

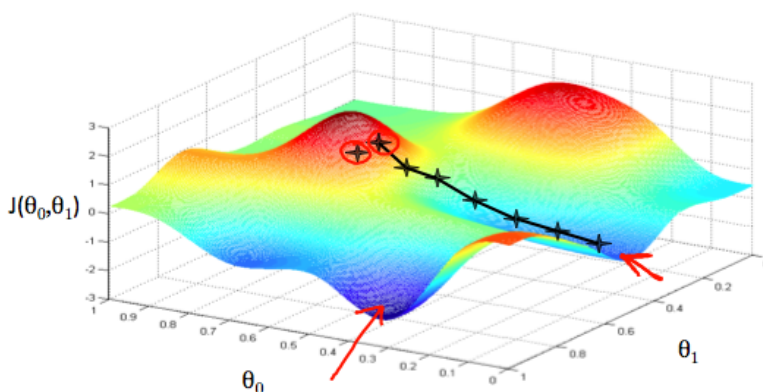
- Algorithm:

Repeat until convergence:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \dots, \theta_n) \text{ where } j = 0, \dots, n_{\text{parameters}}$$

with simultaneous update until convergence.

Importantly, for each iteration, both parameters are updated simultaneously.



The **learning rate** size can cause the gradient descent to be slow to converge if α is too small and diverge if α is too large.

Another issue is that depending on the number of **parameter combinations** that we initialize optimization at a large number of parameter combinations and compare the minimums achieved to be sure that we do not get stuck at an initial local minimum.

If a function is convex, it will always converge to the global minimum (e.g. linear regression).

Now we are considering a "batch" gradient descent where all variables are optimized simultaneously; we make this distinction because other optimization algorithms take sets of variables to optimize sequentially.

Linear Regression with Multiple Variables

Multivariate Linear Regression

We use n to denote the number of features and m the number of examples/observations/rows for each combination of features. Now, we use matrix notation to represent linear combinations of n parameters with n features, for each observation.

Multivariate Gradient Descent

- **Feature Scaling** Scaling the features makes gradient descent faster because changes in one feature may not contribute as much. We can achieve scaling from 0 to 1 by dividing by the sum of all feature values. In general, we scale by **mean normalizing** the data with mean 0 and boundary values -1 and 1:

$$x'_i = \frac{x_i - \mu_i}{std_i}$$

- **Learning Rate:** To ensure that gradient descent works properly, the value of the cost function with the combination of parameters should descend at each iteration. Plotting the values of the cost function at each iteration, the function reaches a plateau; we consider that we achieved convergence.

If the values of the cost functions increase or fluctuate with iterations, we usually should use a smaller α .

Therefore, choosing the right α is completely empirical.

Computing Parameters Analytically

- **Feature transformation** Depending on the insight in the topic we can define transform the features and result into a better model.

For example, house prices with respect to prices usually rise very fast and reach a plateau.

Probably a quadratic function would fit the data well enough, but increasing the size for future predictions would lead to smaller prices, which does not make any sense.

Therefore, we could choose a different polynomial function to change the behavior of the curve generates by our hypothesis function.

Remember to scale features accordingly to the power applied on features.

- **Normal Equation**

To minimize the cost function for each combination of parameters, we perform the partial derivatives of each parameter and solve for 0 through the normal equation. Alternatively to gradient descent, we can implement the normal equation to find the parameters that best fit the data without the need of feature scaling. The main problem of this approach is that computing the inverse is computationally expensive if n is large compared to gradient descent.

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

Examples: $m = 4$.

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

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix}$$

$m \times (n+1)$

$$y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

m -dimensional vector

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

As a rule of thumb, one should use the normal equation up to 10^4 observations.

The non-invertibility of the normal equation may be an issue when implementing this approach. There might be 2 reasons:

a. redundant features (linearly dependent): delete a variable. b. too many features (number of features exceeds the number of observations): delete variables or use regularization.

Logistic Regression

Classification and Representation

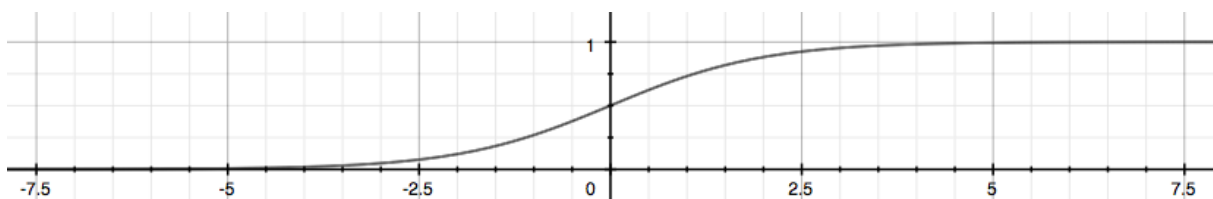
ML in classification problems try discriminate between combinations of features. For example, we could use linear regression to find a threshold between values. But, there are approaches that can better label.

Logistic Regression Model

Even though it is called regression, it is a classification algorithm.

- Hypothesis Representation We want our classifier to give us values between 0 and 1. Then, we can define the hypothesis function as a **sigmoid or logistic function**:

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



Which looks like a switch that is asymptotic at $y=0$ for $-\infty$ and $y=1$ for $+\infty$. Now, we interpret the hypothesis function as a probability of classification:

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

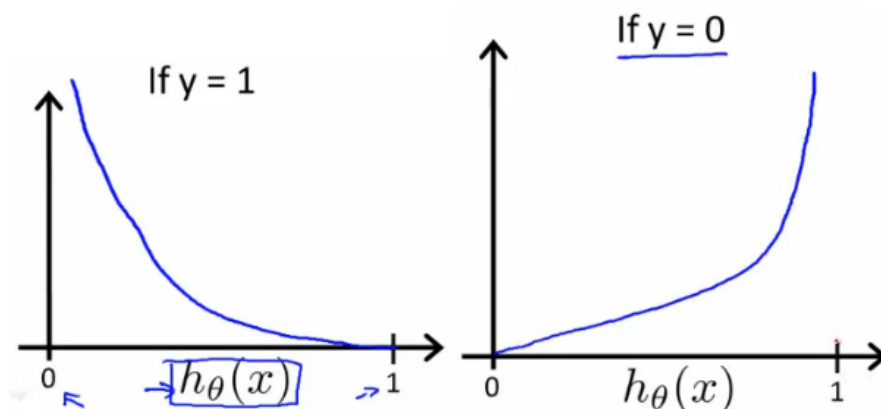
- Decision Boundary

The hypothesis outputs the estimated probability of data being classified as 1 or 0. The sigmoid function per se is greater or equal to 0.5 when $\theta^T x \geq 0$. This hypothesis function will create a **decision boundary** between the two groups; it is a property of the parameters calculated from the dataset. If the dataset has no linearity properties, we can add features by polynomial transformations to get more complex decision boundaries.

- Cost Function

We have a training set with m examples. Now, we use an alternative cost function that compares the predicted with the actual value. If we would use the same cost function than in linear regression, since our hypothesis function is not convex anymore, we would run into a non-convex problem hindering finding a global minimum through gradient descent.

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



- Simplified cost function and gradient descent

We can simplify how the cost function to ease its computation:

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

We will perform gradient descent to find the parameters that minimize the values of the cost function. Even though the hypothesis function has changed, we perform the same parameters update function.

- Advanced Optimization

We need to compute the cost function and its partial derivatives for a number of iterations. A part from Gradient Descent, we could use:

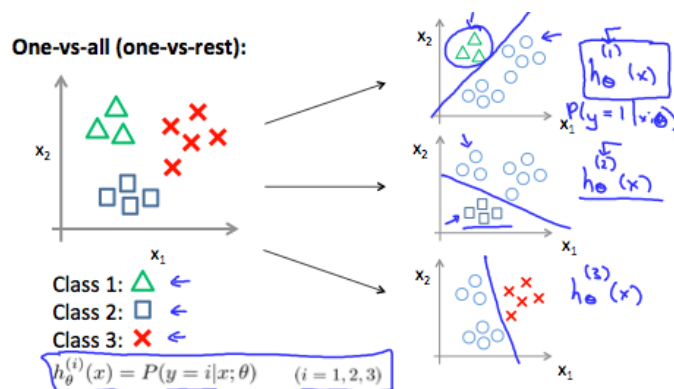
- Conjugate descent: https://en.wikipedia.org/wiki/Conjugate_gradient_method
- BFGS (Broyden-Fletcher-Goldfarb-Shanno): https://en.wikipedia.org/wiki/Broyden-Fletcher-Goldfarb-Shanno_algorithm
- L-BFGS: https://en.wikipedia.org/wiki/Limited-memory_BFGS

We are not going to cover them deeply. Their main advantages are that they do not need to pick α manually and that they usually converge faster than gradient descent. However, they are quite more complex to implement.

Multiclass Classification

- One vs all method

For example, we could automatically tag our emails in multiple classes, or perform multiple diagnosis of patients. We want to implement the same principles of binary classification problems into multiple. If we want to split the data into 3 groups, we would perform 3 binary classifications that will result in 3 fitted classifiers, each of them trained to recognize each class. Finally, we pick the classifier that retrieves the highest probability of being that value.



Regularization

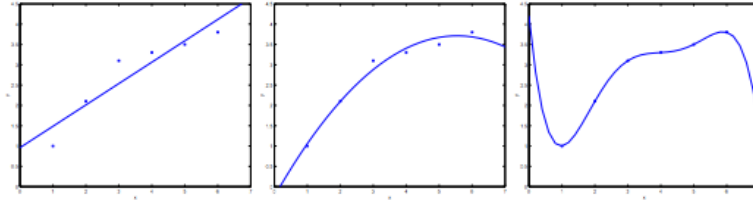
Solving the Problem of Overfitting

When a model does not fit data very well, we consider that it **underfits** data introducing a strong **bias** mathematically to make accurate predictions. **Overfitting** happens when we fit the model to data too well and it performs bad in generalizing the model to test data; unseen examples.

With low-dimensional data, we can easily visualize it. Conversely, to prevent overfitting with high-dimensional data and low number of observations

- Reduce number of features:
 - manually
 - feature selection algorithms.
- Regularization:
 - keep all features, but reduce magnitude/values of parameters θ_j .

- works well when lots of features that contribute a bit in predicting y .



Regularization through cost function

We could introduce a penalty within the cost function making parameters small that results in a "simpler" hypothesis function less prone to overfitting.

Since we do not know which parameters we should shrink, we shrink all of them summing a regularization term that contains a **regularization parameter** (λ).

- Regularized linear regression

$$J(\theta_0) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2$$

If λ is very large we would end up underfitting.

In the parameter update, we only apply regularization from $j=1$ to $j=n$; not $j=0$. This penalty results in a sum term that will decrease the value of the parameter.

$$\theta_0 = [\theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_0^{(i)}] \text{ for } j = 0$$

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{n} \theta_j = [\theta_j (1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}] \text{ for } j=1,2,\dots,n$$

If we perform the **normal equation** method, parameters can be computed through:

$$\theta = (X^T X + \lambda \text{diag}(0, 1, 1, \dots, 1))^{-1} X^T y$$

Recall that if $m < n$, then $X^T X$ is non-invertible. However, when we add the regularization term, then it becomes invertible.

- Regularization for Logistic Regression

We simply add a regularization term based on the parameters to the cost function:

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

And we perform exactly the same update of the parameters.

Neural Networks

Representation

- Non-linear hypotheses

To tackle non-linear problems we could combine features polynomially to get complex decision boundaries. However, for problems with many features initially we would be creating many more features that may lead to overfitting. In addition, we run into a combinatorial explosion problem.

For example, in a computer vision problem, we need to train the model to recognize combinations of pixel intensity values, each of them would be a feature.

- Neurons and the Brain

Neural networks were born trying to make machines mimic the brain. Since computers have improved a lot in the last years, the new computational powers allows. The **one learning algorithm** hypothesis says that if we connect the hearing neurons in the brain to the eyes we would learn how to hear what we see.

- **Model Representation:** The neuron has input and output wires, dendrites and axon, that send the signal to another network. In a computer we use a **logistic unit** constituted of input values (data) and a hypothesis function that will result in different outputs depending on the parameters or weights.

A neural network is a combination of multiple logistic units.

The **input layer** is the layer of data that is interconnected to every **activation unit** i in layer j ($a_i^{(j)}$) if the first hidden layer and second layer of the network.

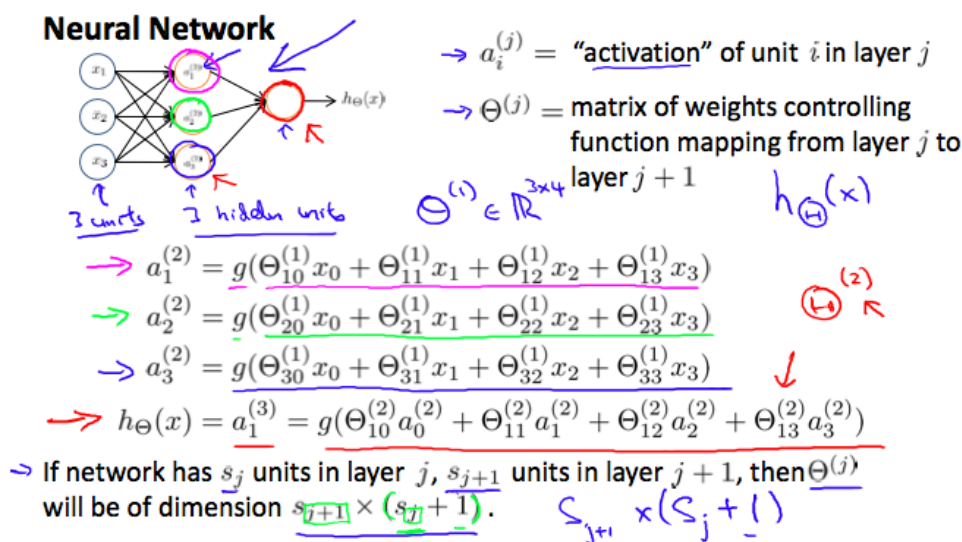
To each activation unit corresponds a $\Theta^{(j)}$ matrix of weights controlling function mapping from layer j to layer $j + 1$.

If the network has s_j units in layer j and s_{j+1} units in layer $j + 1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$ because of the bias element in each activation function.

The +1 comes from the addition in $\Theta^{(j)}$ of the bias nodes, x_0 and $\Theta_0^{(j)}$; the output nodes will not include the bias nodes while the inputs will

In this case, the neural network defines a function h that maps with x 's input values to some space that provisions y .

These hypotheses are parameterized by parameters denoting with a capital Θ so that, as we vary Θ , we obtain different hypothesis and different hypothesis functions.



Andrew N








- Forward propagation: vectorized implementation

The output each hidden layer will be the input of the next layer, in this case, the hypothesis function. We call this process: **forward propagation**.

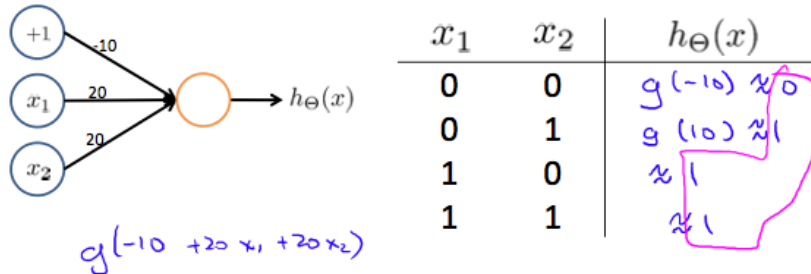
This approach allows to transform the input features in complex ways according to the **network architecture**, without relying only on performing polynomial transformations.

- Computing boolean operations: Depending on the architecture and the parameters, we can represent boolean gates. In the example below, only when $20x_1 + 20x_2 > 10$ the neuron will fire, creating an OR gate.

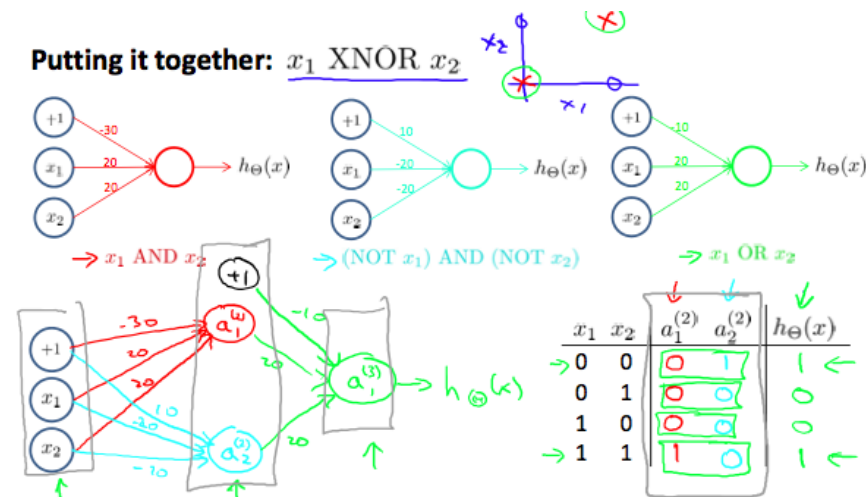
Logic Gates

Name	NOT	AND	NAND	OR	NOR	XOR	XNOR																																																																																																
Alg. Expr.	$\neg A$	AB	\overline{AB}	$A + B$	$\overline{A + B}$	$A \oplus B$	$\overline{A \oplus B}$																																																																																																
Symbol																																																																																																							
Truth Table	<table><tr><th>A</th><th>X</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	X	0	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	0	0	1	0	1	0	0	1	1	1	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	1	0	1	1	1	0	1	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	1	0	1	1	1	0	1	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	0	0	1	0	1	0	0	1	1	1	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	0	0	1	1	1	0	1	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	1	0	1	0	1	0	0	1	1	1
A	X																																																																																																						
0	1																																																																																																						
1	0																																																																																																						
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					

Example: OR function



Combining layers we can increase the complexity of the computation between layers.



- Multiclass classification

If we want to recognize n categories from a dataset, the output layer will have n nodes that will output a vector of 0s and a 1 indicating which is the class predicted.

Learning

- Cost Function and Backpropagation

The cost function that we use in neural networks, we generalize the the for K values that may be the output.

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

The *double sum* simply adds up the logistic regression costs calculated for each cell in the output layer. The *triple sum* simply adds up the squares of all the individual Θ s in the entire network.

Remember that in each layer we add a bias and we do not sum it at the regularization term.

- Backpropagation algorithm in Practice

To compute the gradient term (partial derivative given Θ).

In the forward algorithm we multiply the parameters of each layer, $\Theta^{(l)}$ matrix, with the output of the previous layer with a new bias term, $a^{(l)} = g(z^{(l-1)})$.

Now, the **output layer** results in a vector $\delta^{(l)} = a^{(l)} - y$, which is the difference between the output of the layer and the real values, y .

Then, we move backwards through the layers to keep computing the gradient. In each **hidden layer**, we compute the gradient of the layer, $\delta^{(l-1, \dots, l)}$, we compute the dot product of the parameter matrix of the hidden layer (Θ) with the gradient computed before (layer after in the architecture), and multiply the resulting vectors elementwise with the derivative of the hypothesis function of the actual layer. For example, for the layer before the output layer:

$$\delta^{(l-1)} = (\Theta^{(l-1)})^T \delta^{(l)} \cdot g'(a^{(l-1)})$$

To implement it, we create a matrix, $\Delta_{ij}^{(l)}$, to store the $\delta^{(l)}$. Then, we perform forward propagation to compute all $a^{(l)}$ for $l = 2, 3, \dots, L$. Subsequently, using $y^{(i)}$, we compute $\delta^{(L)} = a^{(L)} - y^{(i)}$, and use it to perform backwards propagation computing $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$. Finally, we update the matrix of $\delta^{(l)}$ s with:

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

We implement regularization through updating matrix D :

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

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

Thus,

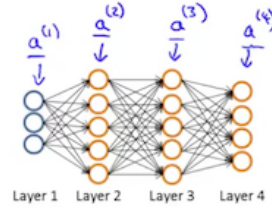
$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

Gradient computation

Given one training example (x, y) :

Forward propagation:

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



- Backpropagation intuition

In forward propagation, each node uses a set of weights to retrieve an the output $z^{(l)}$. In backpropagation, we compute the error of the final prediction through $\delta^{(L)}$ and propagate it to the rest of the network to tune it for best performance.

- Implementation

It is recommended to unroll the $\Theta^{(l)}$ parameters into one big vector to compute to use available optimization functions.

- Gradient checking

To be sure that our implementation is correct, we carry out a gradient checking to be sure that the algorithm is computing the derivative of $J(\Theta)$. We can approximate the slope by:

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

The smaller ϵ the better the approximation of the slope (i.e. $\epsilon = 10^{-4}$).

To check the gradient for each partial derivative we compute the same fraction of the cost function adding ϵ sequentially to each Θ_j :

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

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

- Random Initialization

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly.

Thus, to break symmetry, we initialize each value in Θ with random number between a boundary (e.g. 0 to 1).

- Summary
 - Use a single hidden layer as default. More hidden layers usually have the same number of input units (nodes).
 - In general, the first hidden layer has more input units than the input layer.
 - To train a neural network:
 - a. Randomly initialize weights
 - b. Implement FP to get $h_{(\Theta)}(x^{(i)})$ for any $x^{(i)}$
 - c. Implement code to compute cost function $J(\Theta)$
 - d. Implement BP to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$
 - e. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using BP vs. using numerical estimate of the gradient of $J(\Theta)$. Disable gradient checking afterwards.
 - f. Use gradient descent or another advanced optimization method with backpropagation to try to minimize the non-convex cost function. Since this not a convex problem we may not find the global minimum.

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

```
for i = 1:m,
    Perform forward propagation and backpropagation using example (x(i),y(i))
    (Get activations a(l) and delta terms d(l) for l = 2,...,L
```

- Application: autonomous driving

Advice for Applying Machine Learning

We need to be able to choose which are the arguments for best improvement of the algorithm.

- Get more training examples
- Select sets of features to avoid overfitting
- Try additional features
- Try polynomial features
- Changing λ (regularization constant)

Evaluating a Learning Algorithm

- Evaluating a hypothesis

To start diagnosing our model, we should evaluate our hypothesis. To know if a hypothesis is overfitting we should split the available data in **training** and **test sets**. Usually the division is $\frac{2}{3}$ for training and $\frac{1}{3}$ for testing.

Therefore, if our model is overfitting the data we compute the **test set error**, which corresponds to the cost function $J(\Theta)$ of the model predictions ($h_{\theta}(x_{test})$) and real output (y_{test}).

- Model selection and Train/Validation/Test sets

To perform model selection, we could consider across models that have different polynomial degrees. For each combination of parameters we measure the performance on the test set. However this would not be a good estimate of how the model generalizes because if we test it only once it could be just an optimistic estimate of the generalization error.

Therefore, we usually split the dataset in 3 pieces: training set (60%), cross validation (20%), test set (20%).

Now we will optimize the parameters using the training set. Then, we will test this hypothesis in the cross validation set. Finally, to know how well the model generalizes, we validate the best model according to the cross validation set error using the test set of observations. We expect a lower $J_{test}(\theta)$ than $J_{CV}(\theta)$ because in the model selection we will be fitting another variable (model complexity; i.e. polynomial degree).

Bias vs. Variance

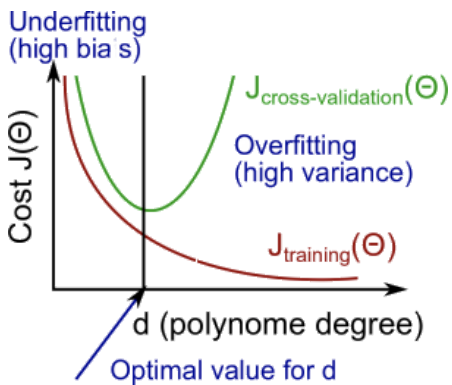
When a model has a high bias it underfits the data, with high variance it overfits the data; we are looking for the "just right" fit.

To visualize bias and variance, we can plot the error vs. the degree of the polynomial d (model complexity).

The training error will decrease with d , while the cross validation error will show a minimum.

If our model suffers from a *bias* problem the training error will be high and similar to the cross validation error.

Differently, if we suffer of overfitting, the training error will be low and very different from the cross validation error.

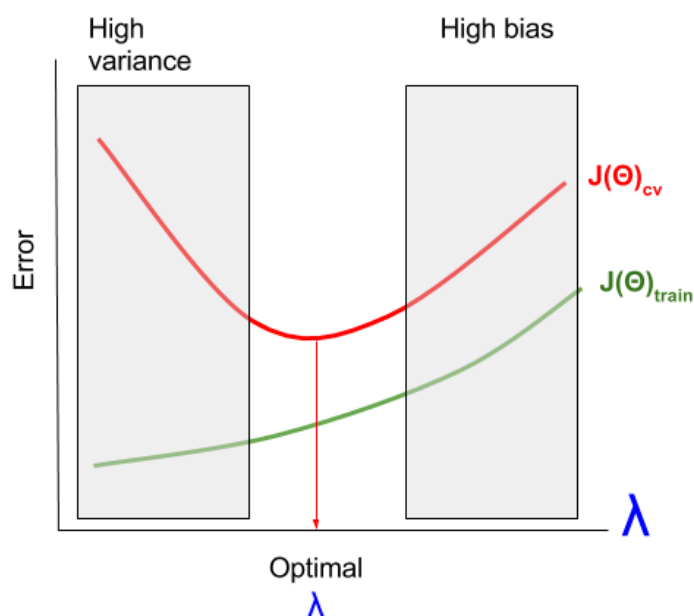


- Regularization and Bias/Variance

To prevent overfitting we implement regularization with a large λ , however we may cause an underfit. Otherwise, if λ is too high we would cause overfitting.

To select the right λ . Usually, we can test a list of 10 different values. For each λ , we can fit the model, use all the parameters to calculate the cross validation error, pick the lowest and compute the test error.

We can directly select the model that minimizes the cross validation error.



- Learning curves

Learning curves help understanding if our model suffers from high bias or variance. We will train the model with increasing number of training set size. We can plot how the training error will increase with higher training set size, while the cross

validation error will decrease.

With high bias (underfitting), the cross validation error will plateau soon and the training error will be initially small and end up being a similar value than the cross validation error. If our model is highly biased, we do not need to train it with more data.

More on Bias vs. Variance

Typical learning curve for high bias (at fixed model complexity):

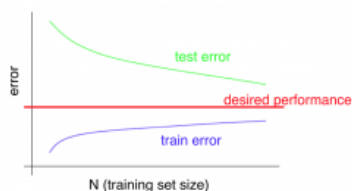


Conversely, if our model has high variance (overfitted), as the training set size increases the training error will be low while the cross validation error will plateau at a high value leaving a gap between the two curves.

In this case, training with more data we would usually improve the performance of the algorithm.

More on Bias vs. Variance

Typical learning curve for high variance (at fixed model complexity):



- Deciding what to try next
 - getting more training examples: fixes variance
 - smaller sets of features: fixes high variance
 - additional features: fixes high bias
 - adding polynomial features: fixes high bias
 - decreasing λ : fixes high bias
 - increasing λ : fixes high variance

Neural networks have many parameters and tend to overfitting. Increasing the number of hidden layers may fix high bias.

Machine Learning System Design

Building a spam filter

Through a supervised learning approach, we need to define x as the features of the email (e.g. 100 words indicative of spam vs. not spam, most frequent words in each labelled case) and y as spam (1) or not (0).

- how to minimize the error:
 - collect lots of data
 - develop sophisticated features based on email routing information
 - develop sophisticated features for message body, e.g. plurals and singulars treated as the same word, capitalized, punctuation, etc.
 - develop sophisticated algorithm to detect misspellings.
- Error analysis

To start, we should implement a simple algorithm to test it on cross-validation data. Then, we should plot **learning curves** to decide whether we should include more data or features that are likely to help. Finally, through **error analysis** we should manually examine the examples from the cross-validation set that our algorithm made errors on and try to spot any systematic trend in what type of examples it is making errors on.

For example, we have 500 examples in our cross validation set. We built an algorithm that misclassifies 100 emails. Now, we should check whether those errors are from a certain type of sender or whether it they may have any feature that could help classification.

Numerical evaluation is extremely important; we need to use error metrics to decide about the performance with the different features or changes that we are introducing (e.g. does the error decrease if we distinguish between lower and upper case?).

Handling Skewed Data

When data is skewed (unbalanced classes), it is difficult to measure the error. If our algorithm makes 99.2% accuracy to 99.5% accuracy; it is not clear if the classification has improved because we have unbalanced classes.

- Precision/Recall We compute a contingency table with True Positive, False Positive, False Negative, True Negative.

$$\text{Precision} = \frac{\text{True Positive}}{\text{\# predicted positive}} = \frac{\text{True Positive}}{\text{True Positive} + \text{False positive}}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{\# actual positives}} = \frac{\text{True Positive}}{\text{True positives} + \text{False negatives}}$$

- Trading off precision and recall

In a logistic regression with a decision threshold, we may increase the threshold to be very sure that a patient has cancer (avoid false positives). In this case the precision will increase but the recall will decrease.

Conversely, to avoid too many cases of cancer (avoid false negatives), we could decrease the threshold increasing the recall but decreasing precision.

We can plot precision vs. recall with different thresholds. Ideally we would like to have a straight line.

How can we merge precision and recall into one score? Instead of using the average, we may use the F_1 Score $= 2 \frac{PR}{P+R}$ that goes from 0 (bad) to 1 (perfect).

Using Large Data Sets

How much data should we train on? On certain conditions training with a lot of data may be positive to get a high accuracy model.

Banko and Brill (2001) compared how different algorithms improved accuracy by increasing the training set size.

The rationale in large data approaches is that we **assume that the feature $x \setminus \text{Epsilon} R^{(n+1)}$ has sufficient information to predict y accurately**; to ensure low bias.

If we use a learning algorithm with lots of parameters that can fit complex functions, with many observations it is unlikely that the models overfits; ensuring low variance.

Support Vector Machines (SVM)

Support vector Machines is an algorithm that is born from logistic regression. In the cost function of logistic regression, when $y=1$, only the first term of the equation retrieves a high value while if z (the prediction) is small.

Large Margin Classification

- Intuition

SVM cost function is constituted by two straight lines: horizontal and diagonal. The cost function to be minimized contains is parametrized differently; it contains two inner cost functions that act as **hypothesis functions** and are not generally averaged; instead they have a constant C .

In this case, the SVM hypothesis function only predicts 1 or 0 given X and θ .

$$\min_{\theta} C \sum_{i=1}^m [y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T x^{(i)})] + \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

Then, if $y=1$, $\text{cost}(z)=0$ we want $z \geq 1$ to reach a minimum of the cost function. Conversely, if $y=0$, to have $\text{cost}(z)=0$ we want $z \leq -1$.

The SVM algorithm tries to separate the data trying to create a decision boundary that results with the margin that best separates the data. Therefore, if there are outliers within the data, we would be more robust to them because they would not lead to the best separation overall.

The highest is C, the less regularized will be our model.

- Mathematics Behind Large Margin classification

Kernels

When we need to create a non-linear decision boundary, a part from polynomial transformation, given x , we can compute the new feature depending on **proximity landmarks** $l^{(1)}, l^{(2)}, l^{(3)}$. Then, we use x and $l^{(i)}$ to compute a **similarity kernel** (e.g. Gaussian kernels = $\exp(-\frac{\|x-l^{(i)}\|^2}{2\sigma^2})$).

If $x \approx l^{(i)}$ then, the kernel will be 1. And 0 otherwise. Each landmark defines a feature.

σ^2 is the variance and creates a more or less steep similarity output. The larger sigma the larger the distribution of the landmark.

Since similarity kernels have different landmarks, for points near to the landmark it will predict 1 and 0 for those that are far, creating a non-linear and complex decision boundaries.

- Choosing the landmarks

We choose the landmarks to be at each point of x . Then, instead of represent the sample data as a vector of f (feature vector). Therefore, we predict $y = 1$ if $\theta^T f \geq 0$, and 1 otherwise.

So the formula changes to:

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

Now, we can compute the regularization term as a vector multiplication since θ is a vector of parameters.

- SVM parameters:
 - $C (= \frac{1}{\lambda})$: large C: lower bias, high variance.
 - σ^2 : if large, the features $f^{(i)}$ vary more smoothly; higher bias, lower variance.

SVMs in Practice

It is recommended to use already optimized libraries. We need to specify the parameter C and the choice of kernel.

- Kernel choices:
 - No kernel (linear kernel)
 - Polynomial kernel: $k(x, l) = (x^T l)^2$ or $k(x, l) = (x^T l + \text{constant})^2$
 - Gaussian kernel: provide the function that transforms the data accordingly. Normalization of the data is very important in this case because we compute the squared.

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

- Multi-class classification We can perform a one-vs-all method: re-label the data into 1s and 0s for each class in y , store the optimized parameters in each case and classify according to the maximum prediction.
- Logistic regression s SVMs

Depending on the the number of features (n) and the number of observations (m):

- n is large, it is better to use logistic regression or SVM without a kernel
- n is small and the amount of data (m) is intermediate: SVMs with Gaussian kernel
- n is small and m is large: create more features, and use logistic regression or SVM without a kernel.

A neural network can work well most likely in all cases, but it may be slower to train.

What matter the most is the capability of troubleshooting.

Unsupervised Learning

We are given data without labels attached to it. We ask the algorithm to find patterns or structures in data.

Clustering has applications for market segmentation, social network analysis, organize computer clusters or astronomical analysis.

Clustering

- K-means

We initialize the algorithm with k **cluster centroids**. Then, we compute the distance of the points to each centroid and divide the data points into k groups, subsequently, we move the centroid into the average of the data in each group until the data points do not change the assigned group.

- Optimization objective

We need to optimize the average distance between the class centroid and its assigned instances:

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

- Random initialization

We should have the number of clusters K lower than the m number of samples. To initialize the centroids, we could use data points randomly. We can get to different solutions in each iteration, getting stuck at different local optima. There is no guarantee of finding the global optima. By initializing K-means many times (50-1000 Monte Carlo simulations depending on the K classes) we hope to get closer to the global optimum more likely.

- Choosing the number of clusters

Choosing the number of clusters is not a simple task because it depends on the type of answers that we want to have.

Generally, we plot the values of the cost function vs. the number of clusters and look for the elbow, where the minimum of the cost functions starts leveling.

If the elbow is not clear, it might indicate that the choice is ambiguous or that data could not be well separated with the given features. Therefore, we should select K according to how well the choice performs in our downstream purposes.

Dimensionality Reduction

- Data compression

When we have a dataset with many features, there might be some features that are highly correlated and thus it is not necessary to have 2 if 1 already explains the data well enough (e.g. measurements in inches or centimeters).

Therefore, we may combine the features into one and achieve data compression by projecting data points on a function, for example.

With a 3D dataset, we can represent data only as a 2D plane with 2 new features instead of 3.

- Data Visualization

We can combine features to visualize data in 2 dimensions. The axis lose indirectly combined the information in the features of each data point.

Principal Component Analysis

- Problem Formulation

PCA tries to find a low dimension projection of the data that best summarizes its features.

For each dimension of the data, PCA finds a vector that minimizes the projected error on data with the condition that each vector is orthogonal (not correlated) to each other.

Differently, than in linear regression PCA minimizes the MSE of the projections of the data points on a vector while linear regression does it in the direction of the y variable.

- Algorithm

Given a training set x .

1. Preprocessing (feature scaling or mean normalization).
2. Find the vector matrix U with a set of vectors u that minimize the Mean Squared Error of the projections of the data on each vector, with the condition that each vector for the next dimension is orthogonal:
 - i. Compute **covariance matrix**:

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

- ii. Compute the **eigenvectors** of matrix Σ :

$$Xv = \lambda v$$

Applying PCA

- Reconstruction from Compressed Representation

Since $z = U_{reduce}^T x$ then, $x_{approx} \approx U_{reduce} z$

- Choosing the number of Principal Components

PCA minimizes the MSE. We can also measure the total variance in the data. Then, as a rule of thumb, we choose k to be the smallest value so that:

$$\frac{\sum_{i=1}^m ||x^{(i)} - x_{approx}^{(i)}||^2}{\sum_{i=1}^m ||x^{(i)}||^2} \geq 0.01$$

So 99% of the variance is retained.

We can access this value using the variance diagonal matrix S resulting from computing the eigenvalues.

- Advice for applying PCA

If we have a supervised problem with images (100x100 pixels), we can reduce the dimensionality of 10'000 to a lower number of dimensions that retains a considerable amount of variance of the data getting rid of uninformative noise that probably does not help data to train the model which becomes faster to train.

We only need to apply it to the training set and reapply the mapping of the training set to the cross-validation and test sets.

Don't address overfitting with PCA, better to use it changing the regularization parameter.

Before implementing PCA, run the algorithm with original data, and implement PCR only if that does not work as we want or as fast as we want.

Anomaly detection

This is not a commonly used ML algorithm. For example, we have a series of aircraft engine features with different examples.

How can we tell if an engine is or not anomalous?

In this kind of problems we perform a **density estimation** of the multidimensional features to predict the probability of an example to be or not anomalous.

Another case is anomaly detection of transactions.

Density Estimation

- Gaussian or Normal Distribution

$$x \sim N(\mu, \sigma^2)$$

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

The Gaussian Probability Density distribution has μ mean and σ^2 variance.

The parameter optimization problem in this cases is, assuming normal distribution of the features, to estimate what are the values of the Gaussian distribution parameters (μ, σ^2).

We can estimate the parameters from the data computing the maximum likelihood. In general:

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

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)^2$$

Building an Anomaly Detection System

We have a training set of m examples with n features. We want to model $P(X)$ that predicts a n dimensional Joint Probability Distribution of Gaussian distributions.

We consider a JPD:

$$P(X) = \prod_{j=1}^n P(x_j|\mu_j, \sigma_j^2)$$

Note that we are assuming independence between the features.

- Algorithm
 1. Choose features x_i that you think might be indicative of anomalous examples.
 2. fit parameters μ_j and σ_j^2 for each feature.
 3. Then, given a new example we will compute $P(x_{new})$ with the previously fitted parameters. We will consider it an anomaly if $P(X_{new}) < \epsilon$, where ϵ is a threshold or decision boundary.
- Developing and Evaluating an Anomaly Detection System

We assume that we have some labeled data, although we assumed this was an unsupervised learning algorithm. Therefore, we have training, cross-validation and testing data sets.

Usually the data is very skewed towards non-anomalous examples. In this case, we would only use non-anomalous data in the training set and the same anomalous in the cross-validation and testing sets.

- Evaluation algorithm
 1. Fit model on training set
 2. Use cross-validation to predict anomalies
 3. Possible evaluation metrics: (NOTE! accuracy will not be a good metric due to skewed classes)
 - Precision/Recall
 - F1 Score
 4. Use cross-validation set to choose ϵ
 5. Use test set for final evaluation
- Anomaly Detection vs. Supervised Learning
 - Pros
 - When classes are very skewed.
 - Cons
 - Many different *types* of anomaly: it is hard for an algorithm to learn from positive examples what the anomalies look like, so future anomalies may look nothing like any of the anomalous examples we have seen so far. Conversely, in supervised learning, we have enough positive examples for algorithm to get a sense of what positive examples are.
- Feature Selection

- Plot data to be sure whether data is gaussian. If the data is not gaussian, we can transform it (e.g. $\log()$, $\sqrt{()}$)
- Create new features → Error Analysis
- Which features may take unusually large or small values in the event of an anomaly → combine them to create a new feature that highlights their effect.

Multivariate Gaussian Distribution

Our JPD assumes that each Gaussian distribution is independent. But, if the variables are indeed correlated we should use a multivariate Gaussian distribution to include the interaction between the variables:

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

- Anomaly detection with Multivariate Gaussian Distribution

1. Fit $P(X)$ parameters estimating:

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

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

2. Apply MGD formula to predict probability of a point and establish a threshold to detect the anomaly.

Indeed, the previous model was considering Σ as a diagonal matrix.

The only problem of MGDs is computing Σ^{-1} if there are many parameters. As a rule of thumb we should have 10 times more observations than features to use MGD.

Predicting Movie Ratings

- Problem Formulation

We have different users that rate movies according to their tastes. They have not seen all possible movies, so based on their previous ratings and others' ratings we would like to predict what would be their rating for the movies that they haven't seen yet, to recommend them or not.

- Content Based Recommendations

A part from the rating of the users, we add features that explain the content (e.g. romance, action) of a movie.

We can treat each user j as a single linear regression problem; each user would have a parameter vector.

$r(i, j) = 1$ if user j has rated movie i (0 otherwise). $y^{(i,j)}$ = rating by user j on movie i (if defined). $\theta^{(j)}$ = parameter vector for user j . $x^{(i)}$ = feature vector for movie i . For user j , movie i , predicted rating: $(\theta^{(j)})^T x^{(i)}$. $m^{(j)}$ = no. movies rated by user j . We need to learn $\theta^{(j)}$.

Optimization objective:

$$\min_{\theta^{(1)}, \dots, \theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^n (\theta_k^{(j)})^2$$

Remember that we do not include the bias in the regularization.

The partial derivatives for the update correspond to:

$$\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} \quad (\text{for } k = 0)$$

$$\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)} \quad (\text{for } k \neq 0)$$

Collaborative Filtering

Suppose we have a dataset without knowing whether movies are romantic or action. We can learn features from the ratings directly, we can infer the values of the features for each movie, if users previously states their movie preferences. Therefore, we can fit the feature vectors for which we can fit the predicted values of how each user rates a movie.

In this case, the problem is the opposite given θ predict x_j , we can do the opposite again, and repeat to optimize both.

- Algorithm to solve θ and x simultaneously:

1. Initialize x and θ with small random values

2. Minimize:

- Cost/Objective Function:

$$\min_{\theta^{(1)}, \dots, \theta^{(n_u)}} \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{(i=1)}^{n_m} \sum_{(k=1)}^n (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{(j=1)}^{n_u} \sum_{(k=1)}^n (\theta_k^{(j)})^2$$

Learning the features this way, there is no need to hard code the feature θ_0

- Gradients:

$$\sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) \theta_k^{(j)} + \lambda x_k^{(i)}$$

$$\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)}$$

Low Rank Matrix Factorization

- Making predictions

How can we give recommendations on what other movies other users might like?

We can make predictions with $\theta^{(j)T} x^{(i)}$ or in the vectorized way: $X\Theta^T$.

- Finding related examples

Predict distances between movies.

- Implementation Details: Mean Normalization

If a user has not rated any movies, we will want to learn 2 features for a user that has not rated any movies, we would get parameters optimize to 0. It does not seem very useful.

With mean normalization, we compute the average rating of each movie. Then we perform mean subtraction to the Y matrix of examples. Now, each movie has an average rating of 0. We will use the mean normalized movie ratings and use the means to make predictions.

Large Scale Machine Learning

We want to use large datasets to train algorithms with low bias. However, with larger datasets, the computational expenses increase dramatically.

Remember that plotting the learning curves we could know whether we do need so much data. Only, when the training and cross-validation learning curves show high variance and no bias.

Gradient Descent with Large Datasets

In gradient descent with linear regression, the most expensive computations is the sum over all examples in the gradient for parameter update.

We can perform **batch gradient descent**.

- Batch/Stochastic Gradient Descent

We randomly shuffle all examples. We measure how well the cost function performs in one example:

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

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

The algorithm modifies the parameters after each training example. Instead than performing updates based on multiple parameters. In this case, the algorithm for optimization moves randomly towards the optimum, usually it just ends up close to the global minimum of the cost function.

- Mini-Batch Gradient descent

We use b examples in each iteration, usually, from 2 to 100 to perform a batch update. It is an intermediate approach. Remember to use vectorization.

- Stochastic Gradient Descent Convergence

We can plot the example costs after a large number of iterations. Usually, it will look very noisy. Use it to increase or decrease the learning rate (α) accordingly.

In the stochastic implementations of gradient descent an make it converge to a global minimum, we should decrease α slowly over iterations:

$$\alpha = \frac{constant_1}{iteration + constant_2}$$

Advanced topics

- Streamlined learning or online learning If we have a continuous streamline of data, we may continuously want to optimize our model.

An online algorithm keeps repeating forever: getting new examples and update θ . Remarkably, this algorithm can adapt to changes in the data with time. For example, we can use it to predict a online store user Click Through Rate; show the articles that the user is likely to click on.

- Map Reduce

Some ML problems are too big to run into a single computer.

What Jeffrey Dean and Sanjay Ghemawat envisioned is to perform batch-gradient descent split a training set in 4 sets. We will perform the summation of the gradient of each set in a different computer and sum the temporary results to get the total summation.

- Data Parallelism

In multi-core machines, we can perform the same approach to each core of the computer.

Application Example:

Photo OCR

- Problem description and pipeline

In a picture we have many different objects. In our problem, we want to do photo recognition of letters in images.

1. Text detection
2. Character segmentation
3. Character classification
4. (Correction system to correct ortographic errors)

We want to design a pipeline that has different machine learning algorithms to have a better overall performance.

- Sliding Windows classifier

We will use a pedestrian detection system. To train our model we have labeled images of the same pixel size with and without a pedestrian. To detect the pedestrians in a larger picture, we slide a window with the same aspect ratio and predict whether there are pedestrians or not. We need to optimize the different sizes of the patch.

For text detection, we use a labeled training set with positive and negative examples and run the sliding window on the larger picture. From our prediction, we **expand** our results whitening the pixels that have whiter pixels at a certain distance. To define the text rectangles we apply a threshold on the minimum ratio of the window that we are interested in detecting.

- Character segmentation

Subsequently, we may train another model on the character segmentation training it with the middle of characters in 1 dimension.

- Character recognition

We may now train another model to recognize the letters.

Getting Lots of Data and Artificial Data

When we have not a lot of data to train our algorithm we can synthesize more from real data.

Adam Coates and Tao Wang proposed a way to take the characters from the computer fonts and put them in different backgrounds. It takes a lot of work to create synthetic data that resembles the real data.

Another approach is to use a real example and distort it to get more. The distortion should be "realistic", not just meaningless random noise.

Note that we should have a low bias classifier first, before expending the effort in creating synthetic examples.

Ways to generate new data:

- Artificial data synthesis
- Correct/label myself
- Crowd source data labeling (e.g. Amazon Mechanical Turk)

Ceiling Analysis: What Part of the Pipeline to Work on Next

We should have an overall metric of our system. Now, we can check the same metric in each step for the test dataset of the pipeline only giving the correctly classified (ground truth) in each step to quantify the loss of prediction power.