------ Agenda ครับ

1. Meet guest developers from Silicon Valle
2. What you need to know to convince your prospective CTO or developer?
3. How can you communicate your ideas to your CTO or developer?

Warodom Khamphanchai, PhD

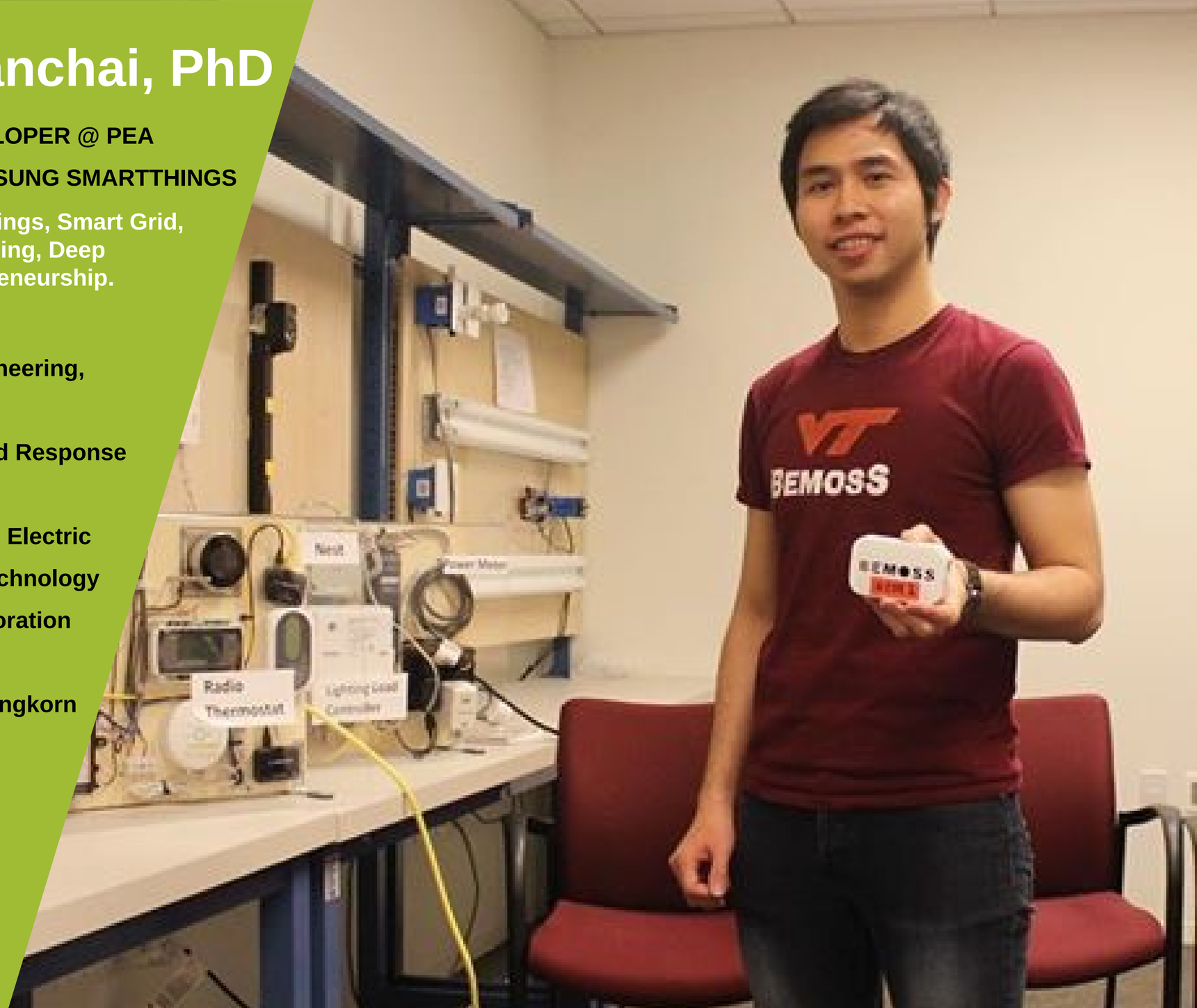
- **LEAD SMART HOME/BUILDING PLATFORM DEVELOPER @ PEA**
- **EX-SOFTWARE DEVELOMENT ENGINEER @ SAMSUNG SMARTTHINGS**

Interests: Home/Building Automation, Internet of Things, Smart Grid, Multi-Agent systems, Data Analytics, Machine Learning, Deep Learning, AI, Energy Audit, and Technology Entrepreneurship.

Education:

- **[2011-2016] Ph.D. in Electrical and Computer Engineering, Virginia Tech**
  Dissertation: An Agent-based Platform for Demand Response Implementation in Smart Buildings
- **[2009-2011] M.E. in Energy (Area of Specialization: Electric Power System Management), Asian Institute of Technology**
  Thesis: A Multi-Agent Based Power System Restoration Approach in Distributed Smart Grid
- **[2005-2009] B.E. in Electrical Engineering, Chulalongkorn University**

Line: kwarodom
LinkedIn: www.linkedin.com/in/kwarodom
Web: kwarodom.wordpress.com
Email: kwardom@vt.edu
Tel: +6695-161-5011 Github: kwarodom

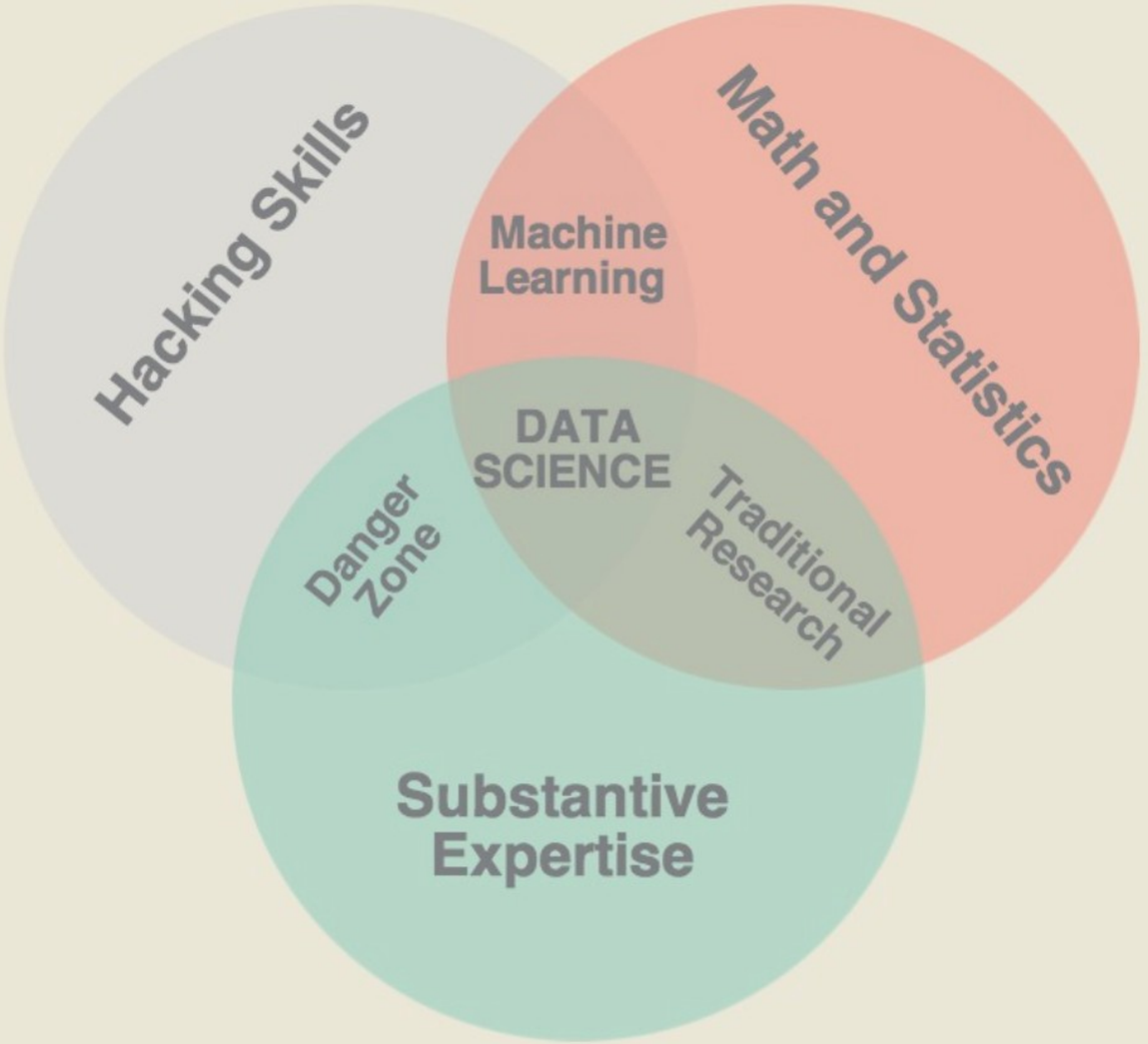


Profile

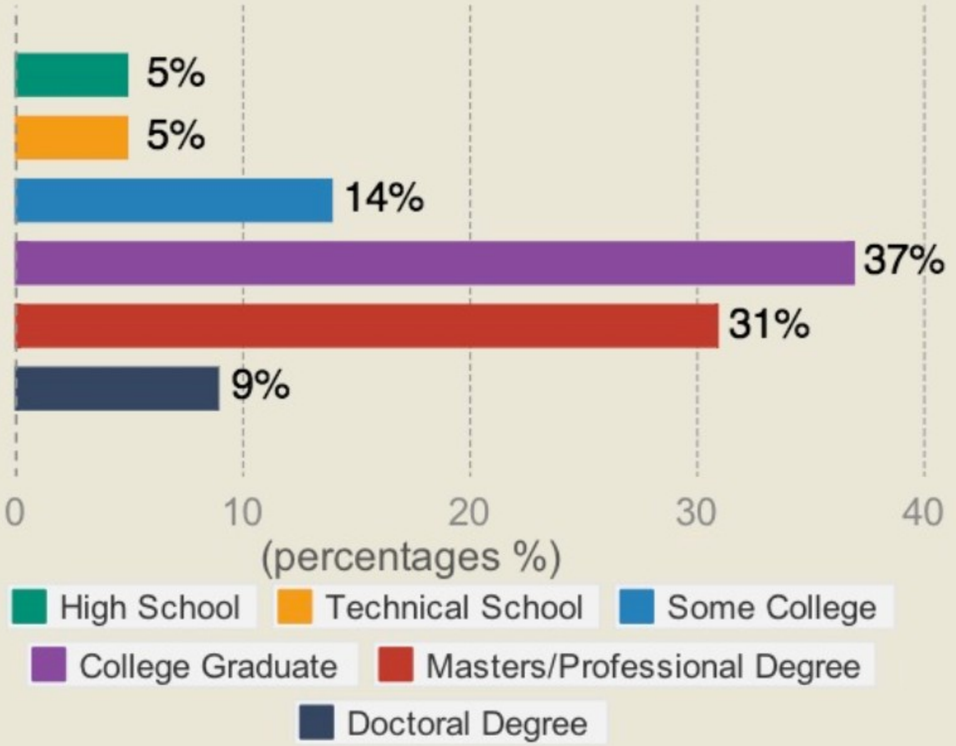
Sorawit Saengkyongam (**James**)

- Data Scientist at Agoda where he works on research and development of personalization algorithms and recommendation systems
- Google Developer Expert in Machine Learning.
- Organize Bangkok Machine Learning Meetup
- Graduated with a major in Mathematical Statistics from Chulalongkorn Unviersity (with first class honours)

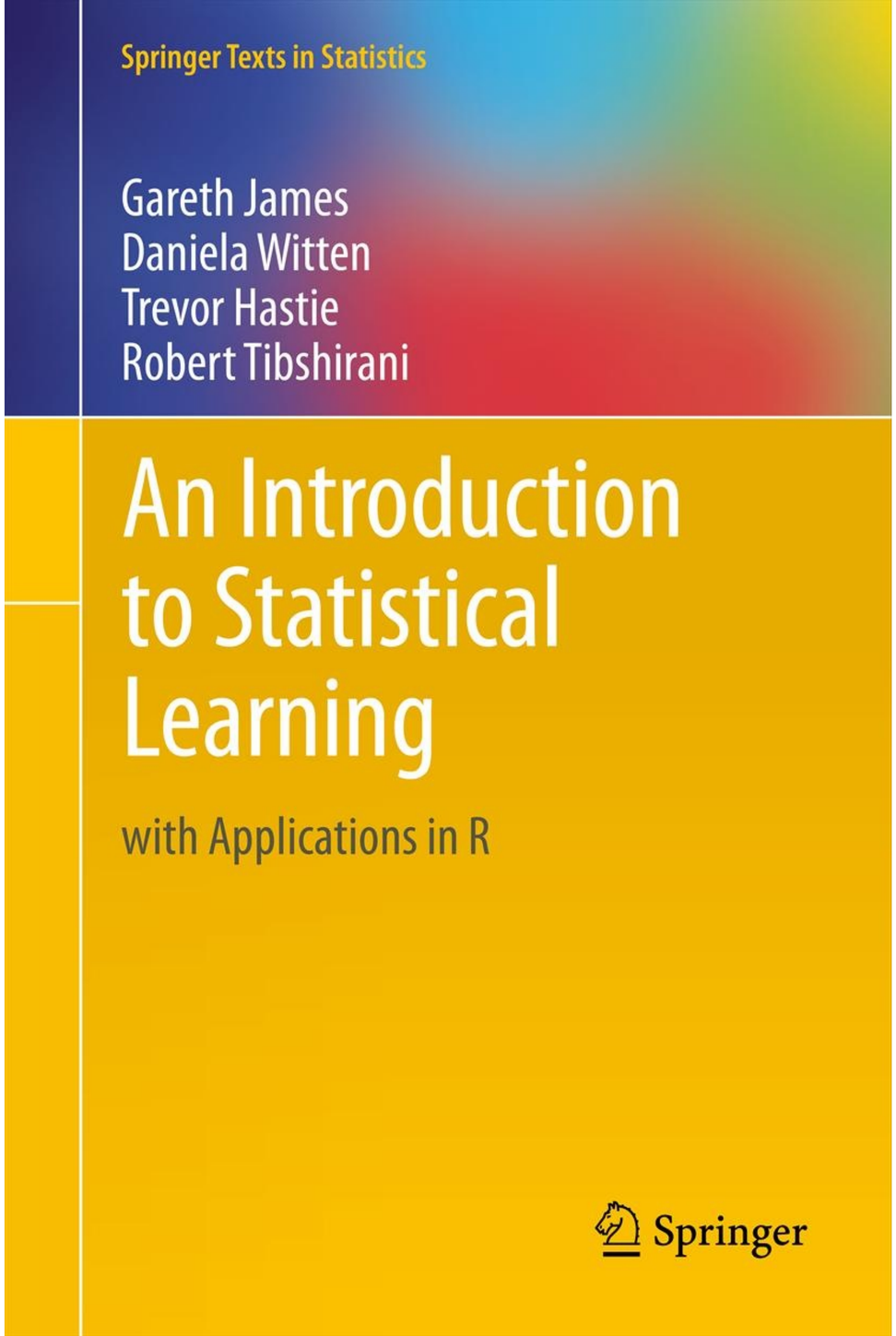
What's a data scientist?



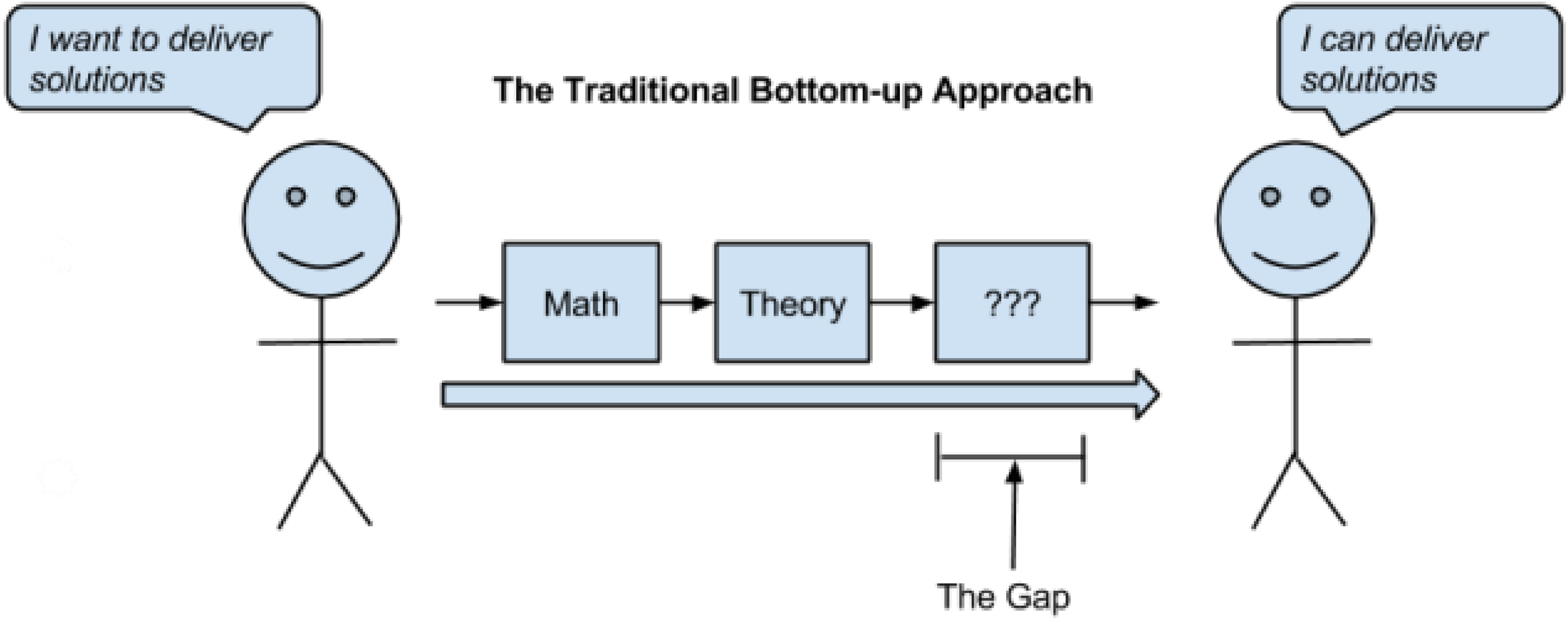
Typical Background



A data scientist is someone who is better at statistics than any software engineer and better at software engineering than any statistician.



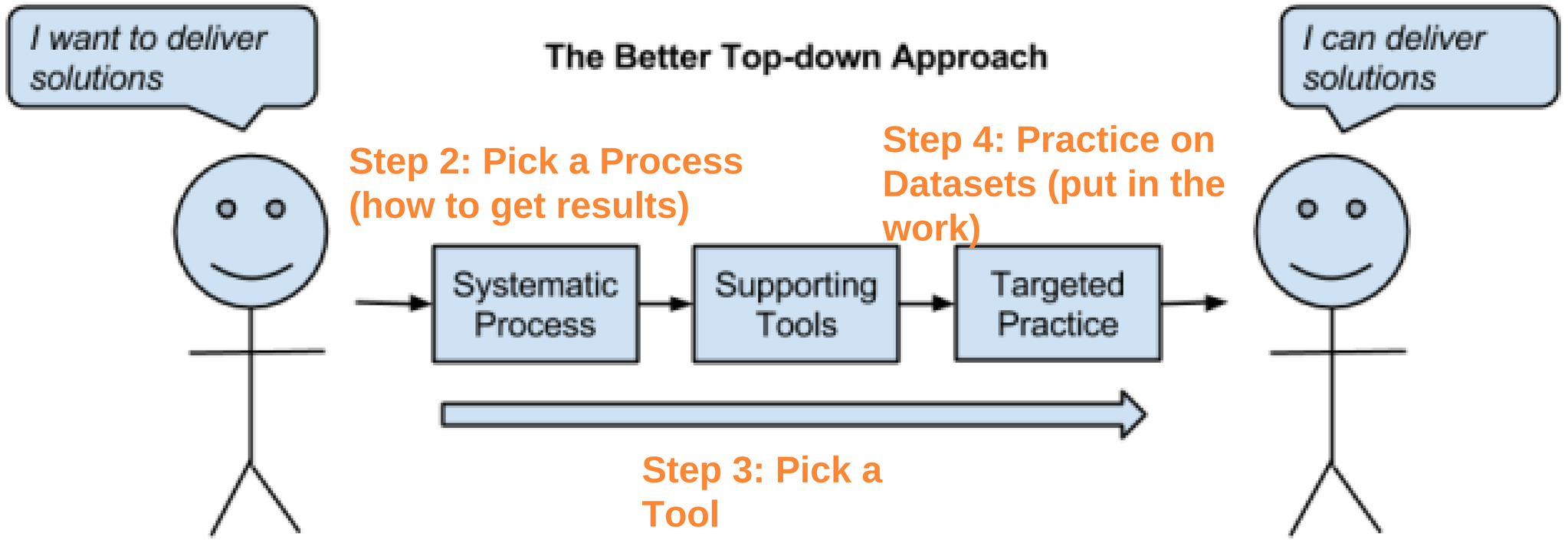
MACHINE LEARNING PRACTIONER



REF: http://machinelearningmastery.com/machine-learning-mastery-method/

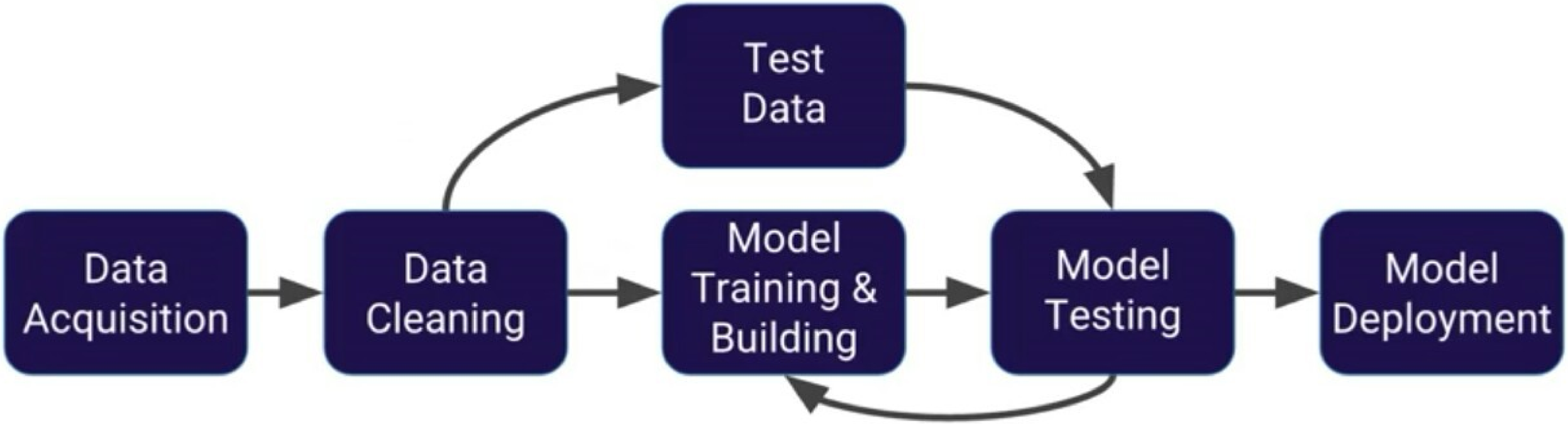
MACHINE LEARNING PRACTIONER

5-Steps To Get Started and Get Good at Machine Learning



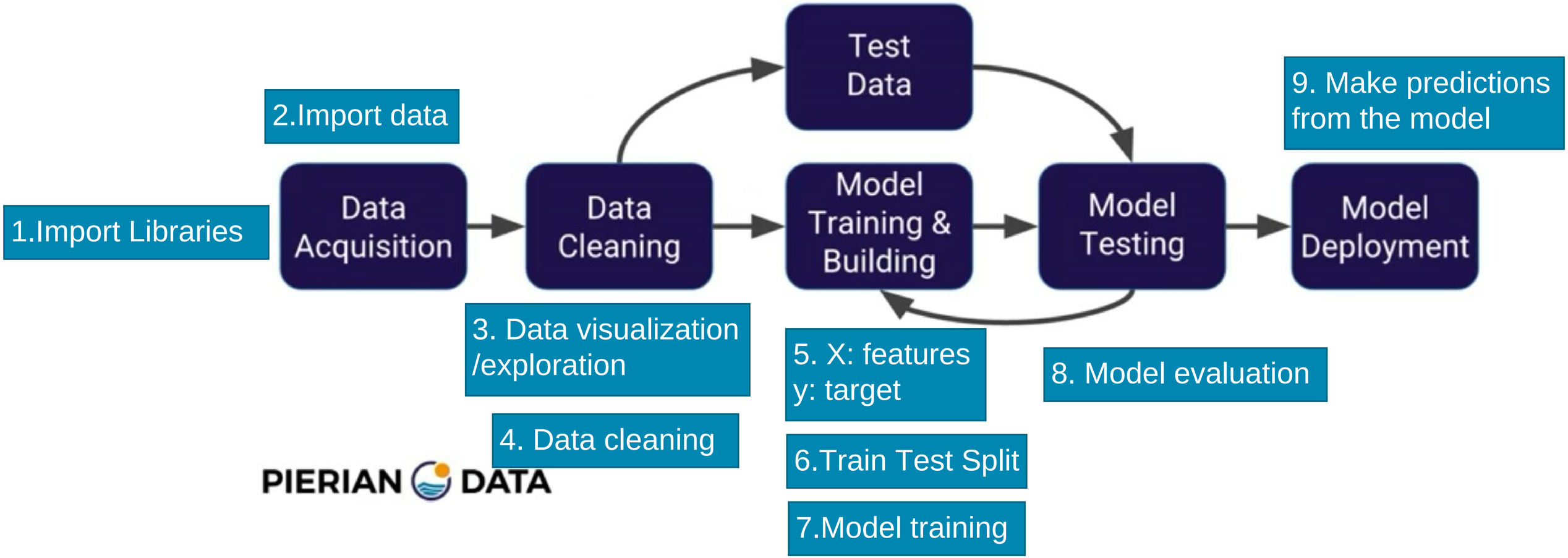
REF: http://machinelearningmastery.com/machine-learning-mastery-method/

MACHINE LEARNING PROCESS



Source: PIERIAN DATA

Machine Learning Process



PIERIAN DATA

Machine Learning

What is Machine Learning?
Two definitions of Machine Learning are offered.

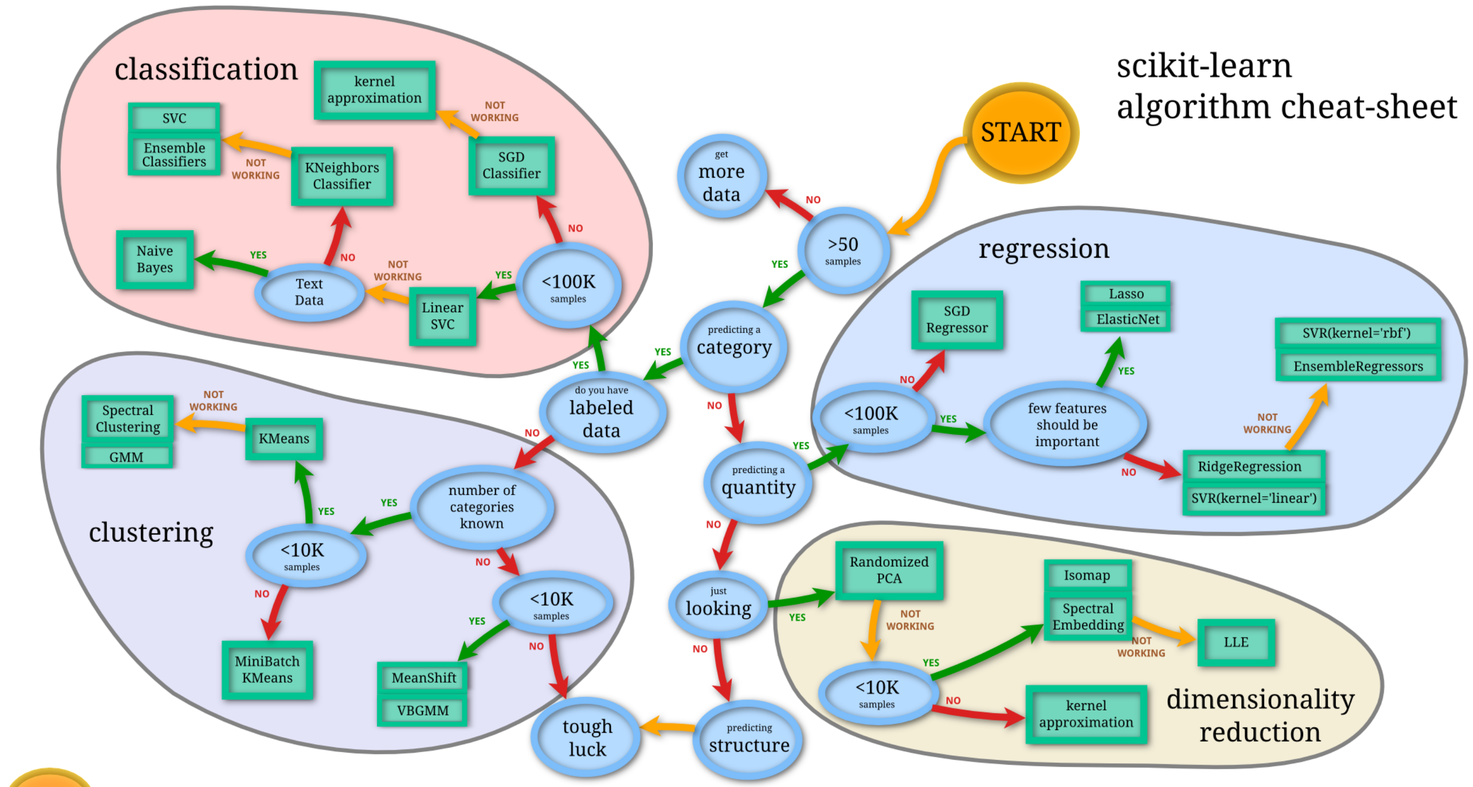
Arthur Samuel described it as: "the field of study that gives computers the ability to learn without being explicitly programmed."

Tom Mitchell provides a more modern definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Example: playing checkers.
E = the experience of playing many games of checkers
T = the task of playing checkers.
P = the probability that the program will win the next game.

Scikit Learn

- Jump-Start Scikit-Learn
  - **What is scikit-learn**
    - Where did it come from
    - What is scikit-learn
    - What are the features
    - Who is using it
    - Documentation
  - **Handling Data**
    - Example Datasets
    - Load from CSV
    - Normalization
    - Standardization
    - Imputing
  - **Supervised Learning**
    - Linear Regression
    - Logistic Regression
    - Linear Discriminant Analysis
    - Quadratic Discriminant Analysis
    - Perceptron
    - Naive Bayes
    - k-Nearest Neighbor
    - Classification and Regression Trees
    - Support Vector Machines
  - **Regularization**
    - Ridge Regression
    - Least Absolute Shrinkage and Selection Operator
    - Least Angle Regression
    - LASSO LARS
    - ElasticNet
  - **Ensemble Methods**
    - Random Forest
    - Extra Trees
    - Adaptive Boosting
    - Gradient Boosting
  - **Advanced**
    - Recursive Feature Elimination
    - Feature Importance
    - Cross Validation
    - Grid Search Parameter Tuning
    - Random Search Parameter Tuning



Supervised Learning

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

Supervised learning problems are categorized into "regression" and "classification" problems.

In a regression problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function.

In a classification problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

Data Preparation

Normalization

```
# Normalize the data attributes for the Iris dataset.
from sklearn.datasets import load_iris
from sklearn import preprocessing
# load the iris dataset
iris = load_iris()
print(iris.data.shape)
# separate the data from the target attributes
X = iris.data
y = iris.target
# normalize the data attributes
normalized_X = preprocessing.normalize(X)
```

Standardisation

```
# Standardize the data attributes for the Iris dataset.
from sklearn.datasets import load_iris
from sklearn import preprocessing
# load the Iris dataset
iris = load_iris()
print(iris.data.shape)
# separate the data and target attributes
X = iris.data
y = iris.target
# standardize the data attributes
standardized_X = preprocessing.scale(X)
```

Data Preparation

Imputing

Data can have missing values. These are values for attributes where a measurement could not be taken or is corrupt for some reason. It is important to identify and mark this missing data. Once marked, replacement values can be prepared. This is useful because some algorithms are unable to work with or exploit missing data.

```
# Mark 0 values as missing and impute with the mean
import numpy as np
import urllib
from sklearn.preprocessing import Imputer
# Load the Pima Indians Diabetes dataset
url = "http://goo.gl/j0Rvxq"
raw_data = urllib.urlopen(url)
dataset = np.loadtxt(raw_data, delimiter=",")
print(dataset.shape)
# separate the data and target attributes
X = dataset[:,0:7]
y = dataset[:,8]
# Mark all zero values as 0
X[X==0]=np.nan
# Impute all missing values with the mean of the attribute
imp = Imputer(missing_values='NaN', strategy='mean')
imputed_X = imp.fit_transform(X)
```

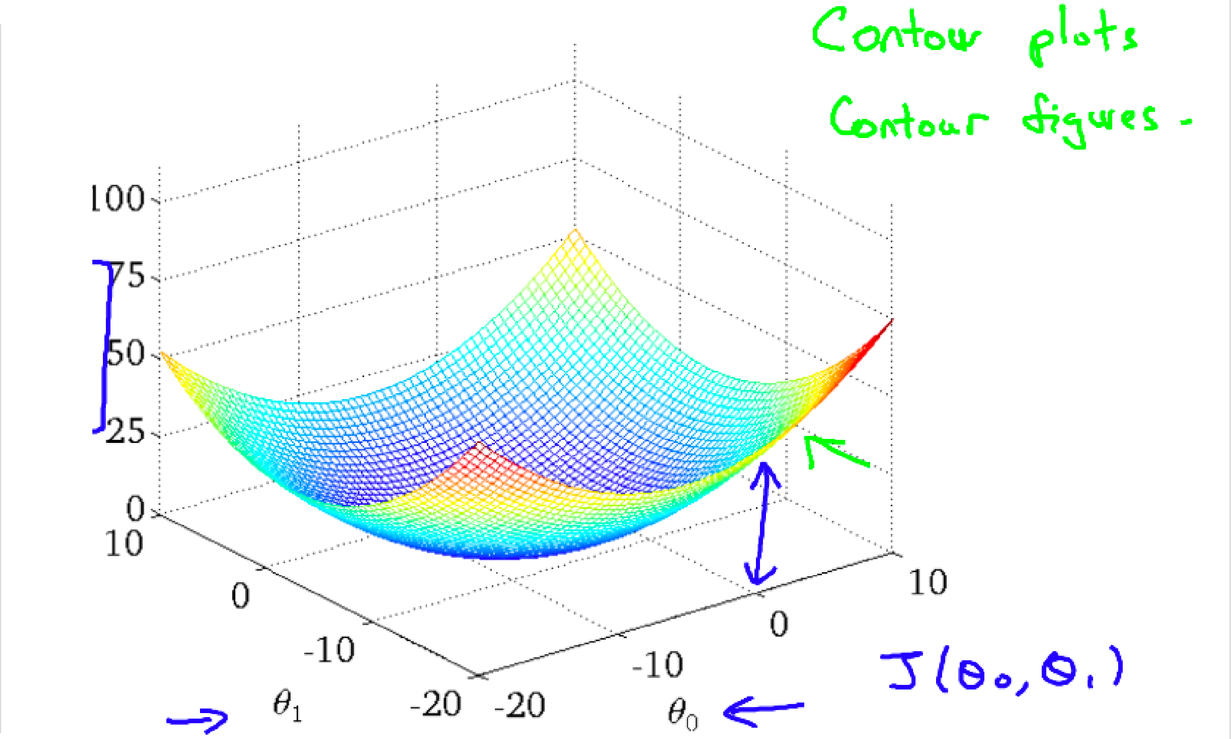
Regression

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

Parameters: θ_0, θ_1

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

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



Andrew Ng

Linear Regression with One Variable

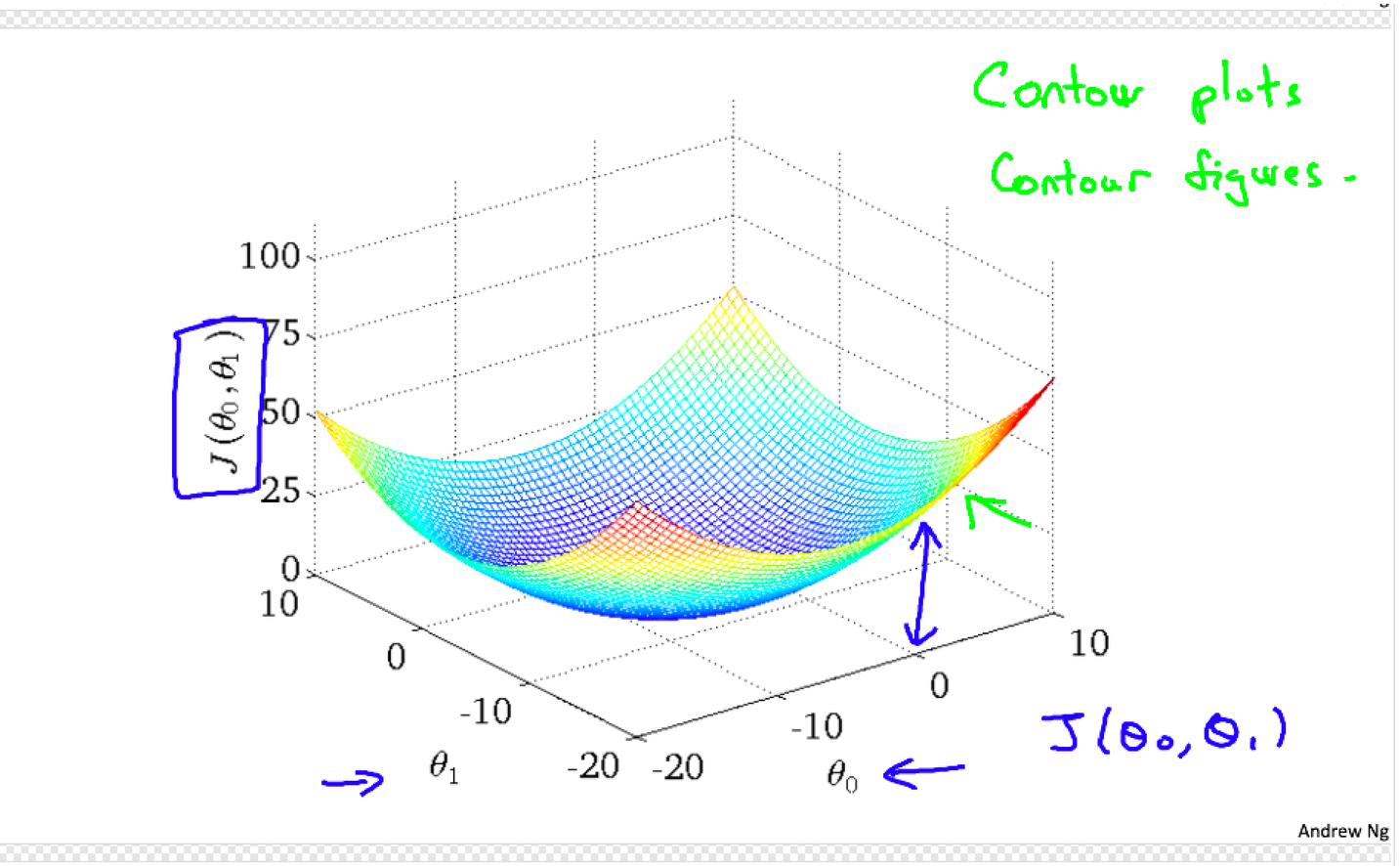
Model and Cost Function

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

Parameters: θ_0, θ_1

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

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



Parameter Learning

Gradient Descent

repeat until convergence: {

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

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

}

Multivariate Linear Regression

Multiple Features

Multiple features (variables).

Size (feet^2) 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
...	...	...	...	...

$m = 47$

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

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.

$x_3^{(2)} = 2$

Multivariate Linear Regression

Multiple Features

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

$x_0 = 1$

Parameters: $\theta_0, \theta_1, \ldots, \theta_n$ $\quad \theta$ $\quad$ n+1 - dimensional vector

Cost function:

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

$J(\theta)$

Gradient descent:

Repeat $\{$

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

$J(\theta)$

$\}$

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

Multivariate Linear Regression

Gradient Descent for Multiple Variables

Gradient Descent

Previously (n=1):

Repeat $\{$

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

$\frac{\partial}{\partial \theta_0} J(\theta)$

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

(simultaneously update θ_0, θ_1)

$\}$

New algorithm $(n \geq 1)$:

Repeat $\{$

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

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

$\}$

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

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

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

$\ldots$

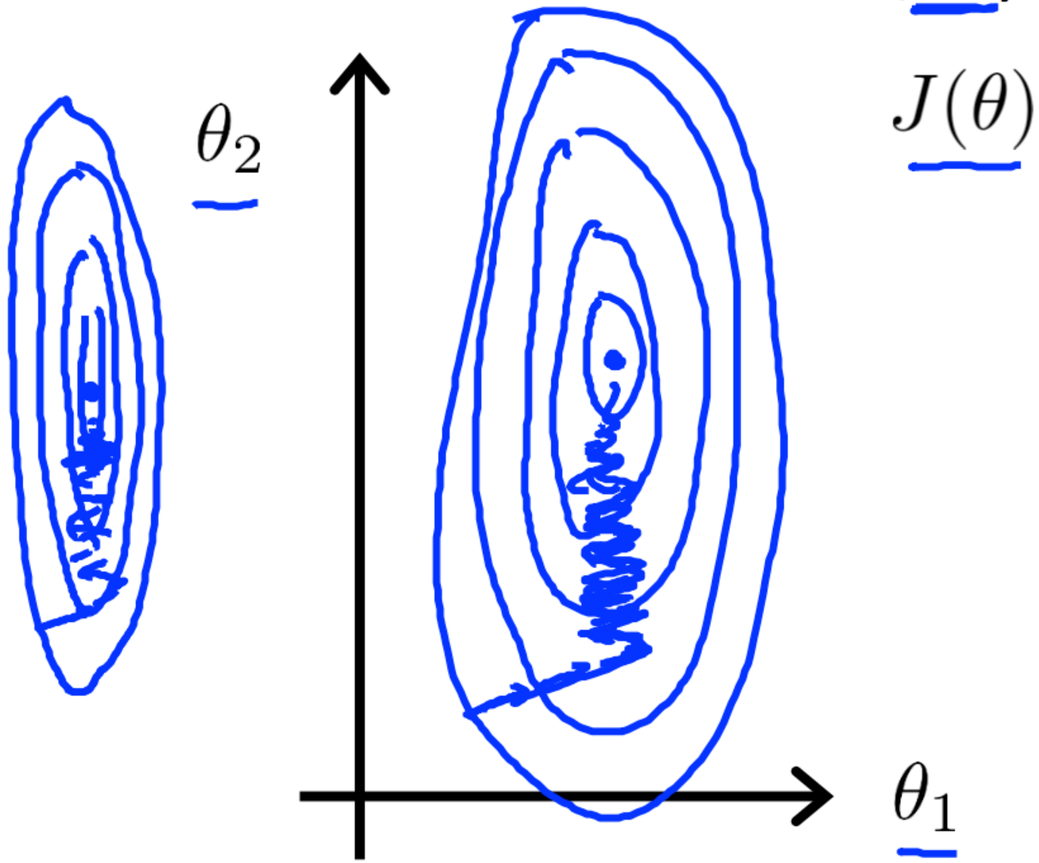
Multivariate Linear Regression

Feature Scaling

Idea: Make sure features are on a similar scale.

E.g. x_1 = size (0-2000 feet²)

x_2 = number of bedrooms (1-5)

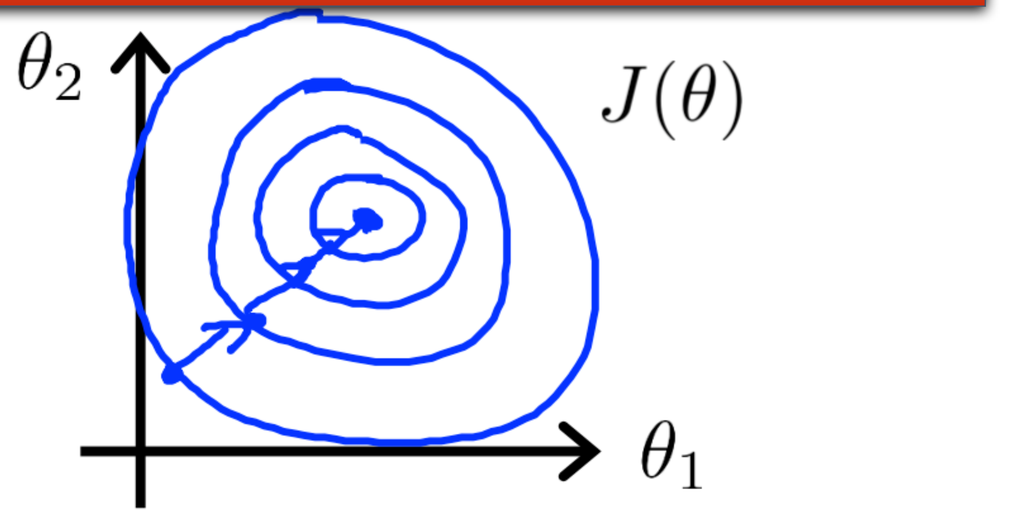


$$x_1 = \frac{\text{size (feet}^2\text{)}}{2000}$$

$$x_2 = \frac{\text{number of bedrooms}}{5}$$

$0 \le x_1 \le 1$ $\quad$ $0 \le x_2 \le 1$

Get every feature into approximately -1 <= xi <= 1



Feature Scaling

Multivariate Linear Regression

Mean normalization

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

E.g. $x_1 = \frac{size - 1000}{2000}$

Average size = 100

$x_2 = \frac{\#bedrooms - 2}{5}$ 4

1 - 5 bedrooms

$-0.5 \leq x_1 \leq 0.5$, $-0.5 \leq x_2 \leq 0.5$

$x_1 \leftarrow \frac{x_1 - \mu_1}{s_1}$ ← avg value of x_1 in training set

range (max - min)

(or standard deviation)

$x_2 \leftarrow \frac{x_2 - \mu_2}{s_2}$

Multivariate Linear Regression

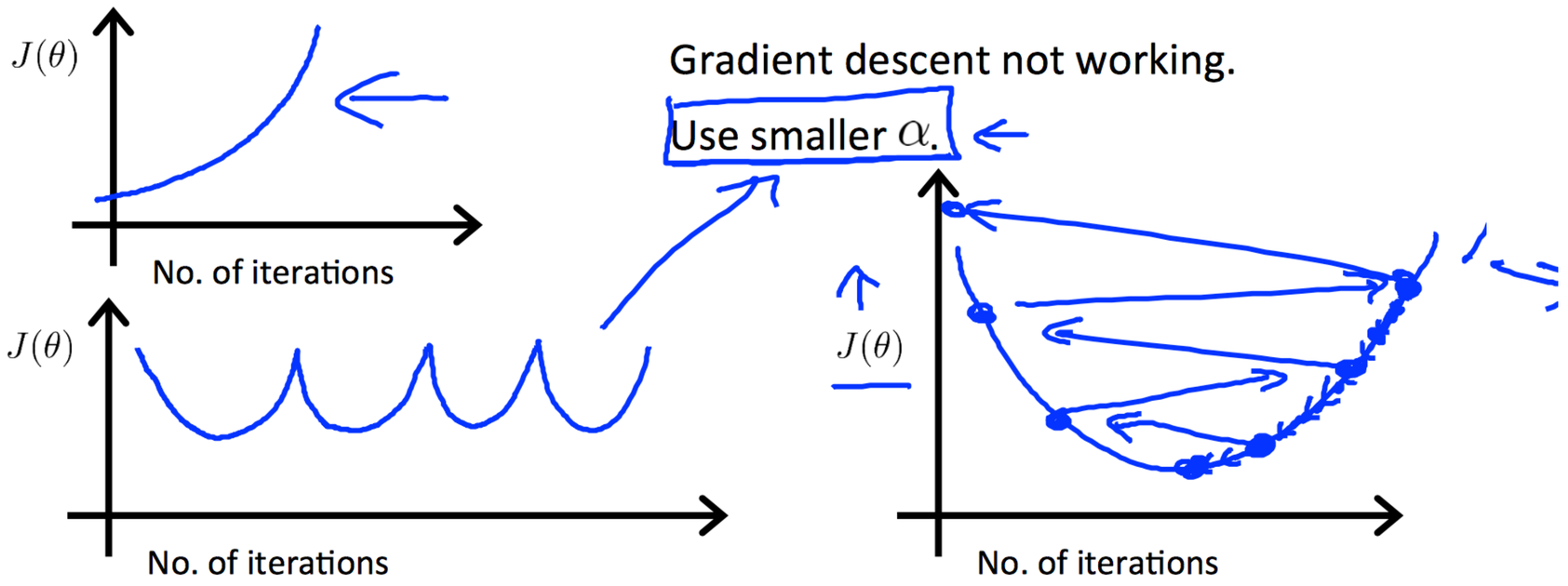
Learning Rate

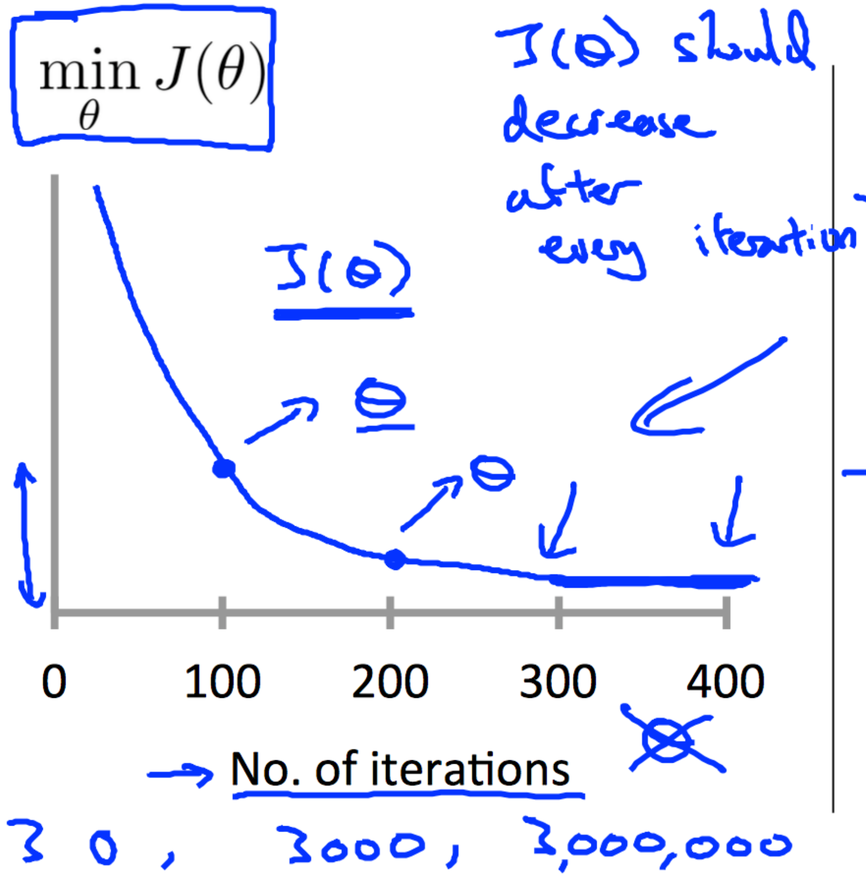
Gradient descent **How to choose learning rate?**

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

If α is too small, slow convergence
If α is too large, J(θ) may not decrease on every iteration, may not convergence

Making sure gradient descent is working correctly.





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

To α choose , try …, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, …

Linear Regression with One Variable

Linear Regression

Linear regression fits a linear model (such as a line in two dimensions) to the data.

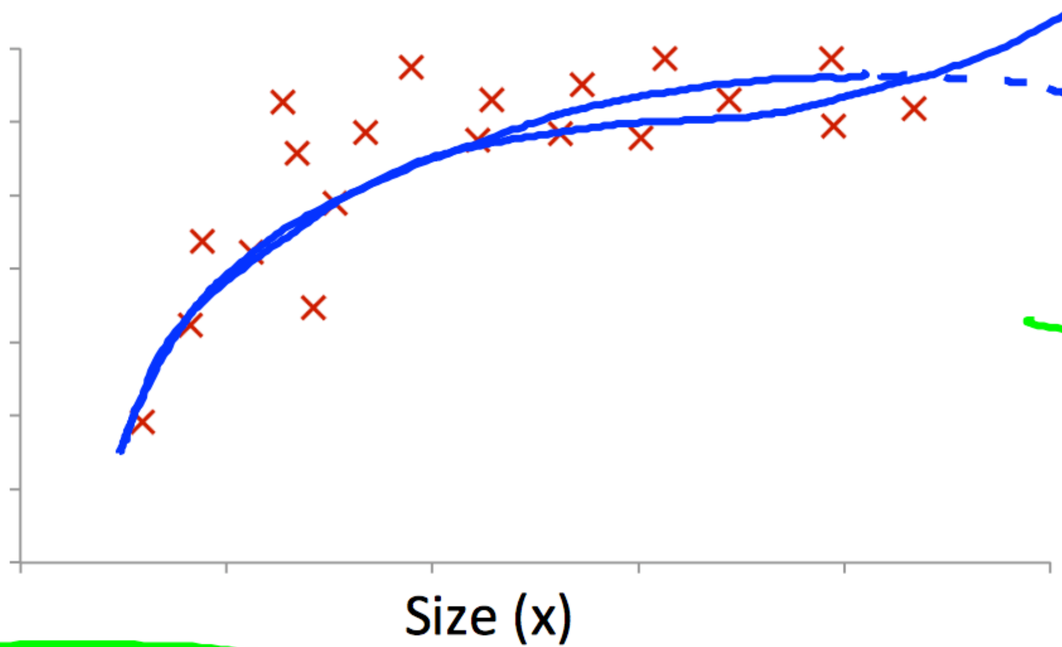
This recipes shows the fitting of a linear regression algorithm on the diabetes dataset.

```
# Linear Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LinearRegression
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a linear regression model to the data
model = LinearRegression()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Polynomial Regression

Feature and Polynomial Regression

Polynomial regression



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

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

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$$
$$= \theta_0 + \theta_1 (size) + \theta_2 (size)^2 + \theta_3 (size)^3$$

$$x_1 = (size)$$
$$x_2 = (size)^2$$
$$x_3 = (size)^3$$

Size: 1 - 1000

Size^2: 1 - 1000,000

Size^3: 1 - 10^9

Computing Parameters Analytically

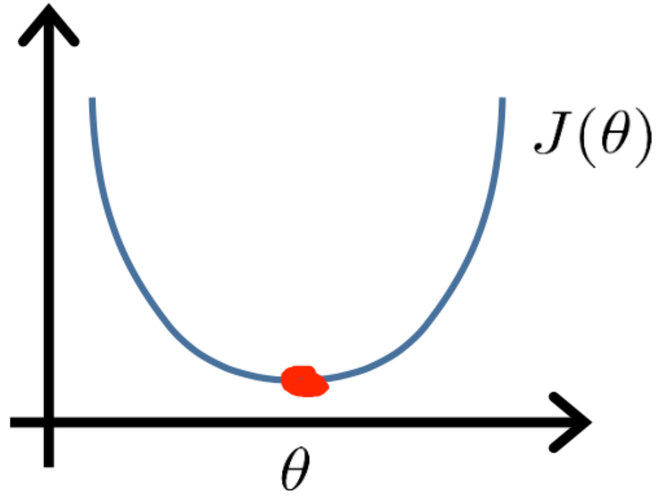
Normal Equation

Intuition: If 1D $(\theta \in \mathbb{R})$

$J(\theta) = a\theta^2 + b\theta + c$

$\frac{d}{d\theta} J(\theta) = \cdots \overset{\text{set}}{=} 0$

Solve for θ



$\theta \in \mathbb{R}^{n+1}$

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

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

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

Normal Equation

Computing Parameters Analytically

Examples: $m = 4$.

x_0	Size (feet²) x_1	Number of bedrooms x_2	Number of floors x_3	Age of home (years) x_4	Price ($1000) 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 x (n+1)

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

m-dimensional vector

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

Computing Parameters Analytically

Normal Equation Noninvertibility

What if $X^T X$ is non-invertible?

- Redundant features (linearly dependent).
  E.g. x_1 = size in feet^2
  ~~x_2 = size in m^2~~
  $1_m = 3.28$ feet
  $x_1 = (3.28)^2 x_2$
- Too many features (e.g. $m \leq n$).
  $m = 10$
  $n = 100$
  $\theta \in \mathbb{R}^{101}$
  - Delete some features, or use regularization.
    ↓ later

Classification

Logistic regression is a method for classifying data into discrete outcomes. For example, we might use logistic regression to classify an email as spam or not spam. In this module, we introduce the notion of classification, the cost function for logistic regression, and the application of logistic regression to multi-class classification.

We are also covering regularization. Machine learning models need to generalize well to new examples that the model has not seen in practice. We'll introduce regularization, which helps prevent models from overfitting the training data.

Classification and Representation

Classification

Classification

- Email: Spam / Not Spam?
- Online Transactions: Fraudulent (Yes / No)?
- Tumor: Malignant / Benign ?

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

0: “Negative Class” (e.g., benign tumor)

1: “Positive Class” (e.g., malignant tumor)

$y \in \{0, 1, 2, 3\}$

Classification and Representation

Hypothesis representation and Cost Function

Logistic regression cost function

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

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

To fit parameters θ :

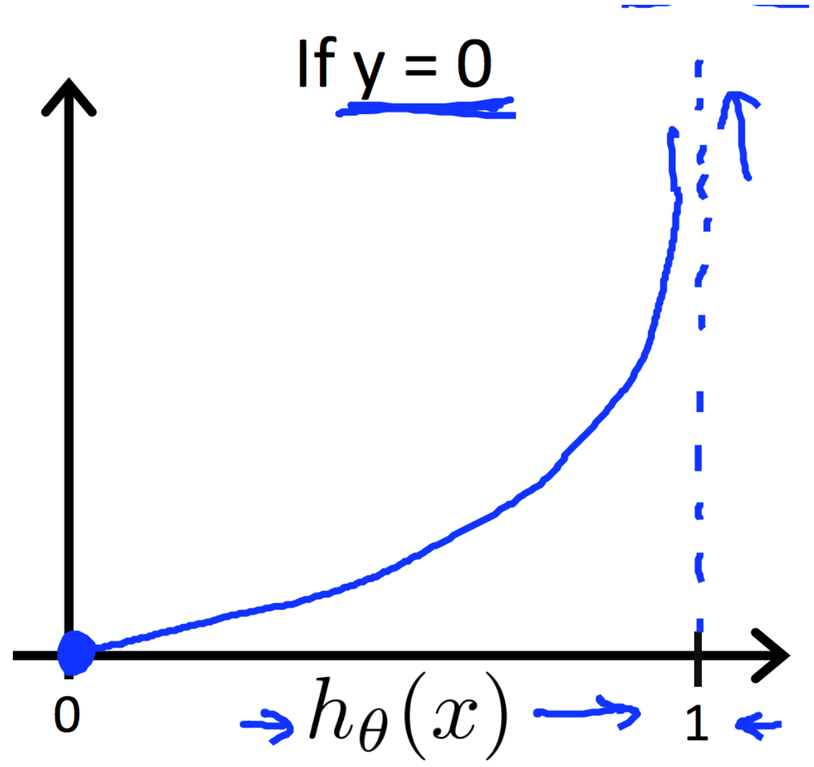
$$\min_\theta J(\theta)$$

Get θ

To make a prediction given new x:

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

$p(y=1 \mid x;\theta)$



Logistic Regression Model

Gradient descent

Gradient Descent

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

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

Repeat $\{$

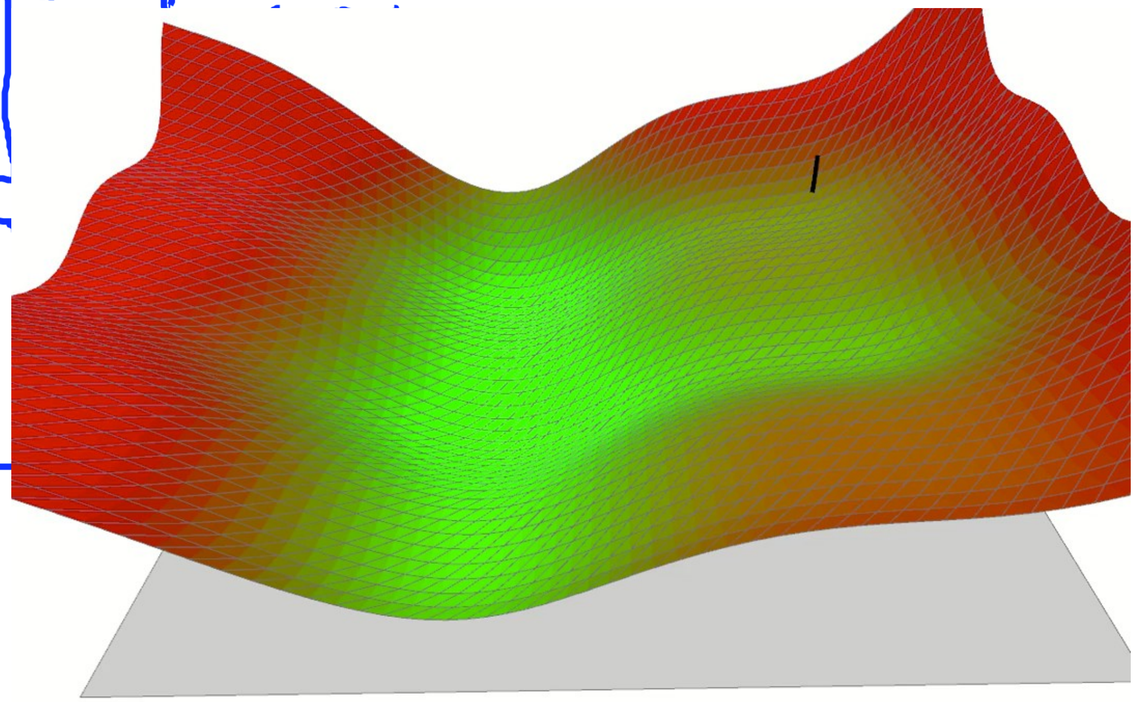
$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(simultaneously update all θ_j)

$\}$

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

Algorithm looks identical to linear regression!



Logistic Regression Model

Advanced optimization

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

Logistic Regression Model

Advanced optimization

We need code to compute

Example: $\min_{\theta} J(\theta)$

$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$ $\theta_1 = 5, \theta_2 = 5.$

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

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

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

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

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

$\theta \in \mathbb{R}^d \quad d \geq 2$

Logistic Regression Model

Advanced optimization

```
def fmin_l_bfgs_b(func, x0, fprime=None, args=(),
                  approx_grad=0,
                  bounds=None, m=10, factr=1e7, pgtol=1e-5,
                  epsilon=1e-8,
                  iprint=-1, maxfun=15000, maxiter=15000, disp=None,
                  callback=None):
    """
    Minimize a function func using the L-BFGS-B algorithm.
```

https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.fmin_l_bfgs_b.html

https://github.com/scipy/scipy/blob/v0.14.0/scipy/optimize/lbfgsb.py#L47

Logistic Regression

Logistic Regression

Logistic regression fits a logistic model to data and makes predictions about the probability of an event (between 0 and 1).

This recipe shows the fitting of a logistic regression algorithm to the iris dataset. Because this is a mutliclass classification problem and logistic regression makes predictions between 0 and 1, a one-vs-all scheme is used (one model is created per class).

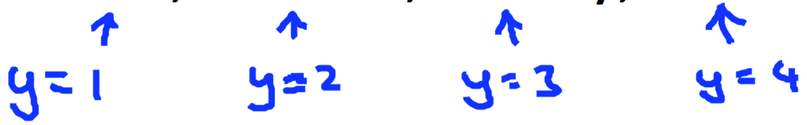
```
# Logistic Regression
from sklearn import datasets
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
# load the iris datasets
dataset = datasets.load_iris()
# fit a logistic regression model to the data
model = LogisticRegression()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

Multiclass Classification

Multiclass classification

Multiclass classification

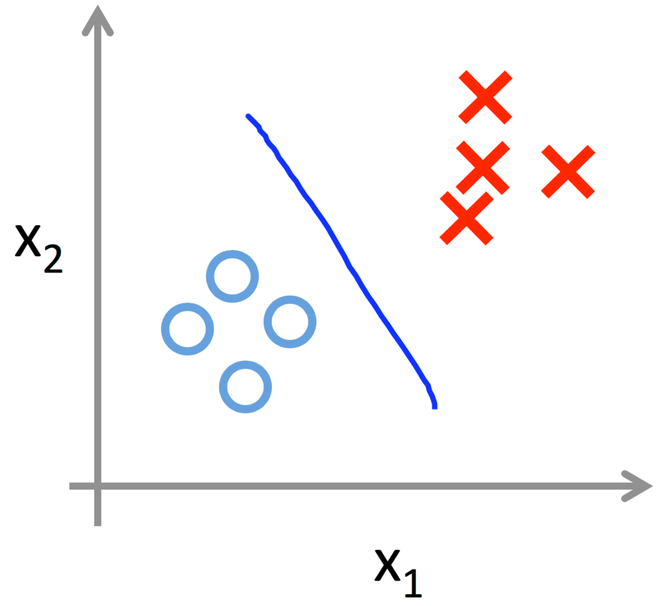
Email foldering/tagging: Work, Friends, Family, Hobby



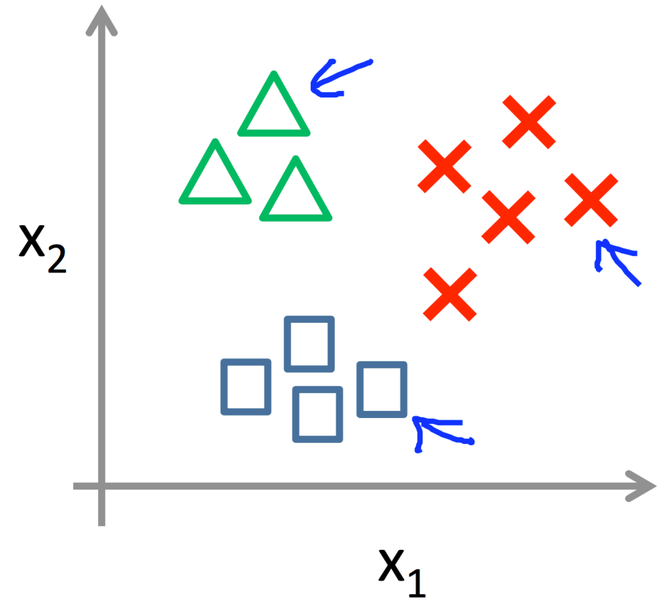
Medical diagrams: Not ill, Cold, Flu

Weather: Sunny, Cloudy, Rain, Snow

Binary classification:



Multi-class classification:



One vs. All

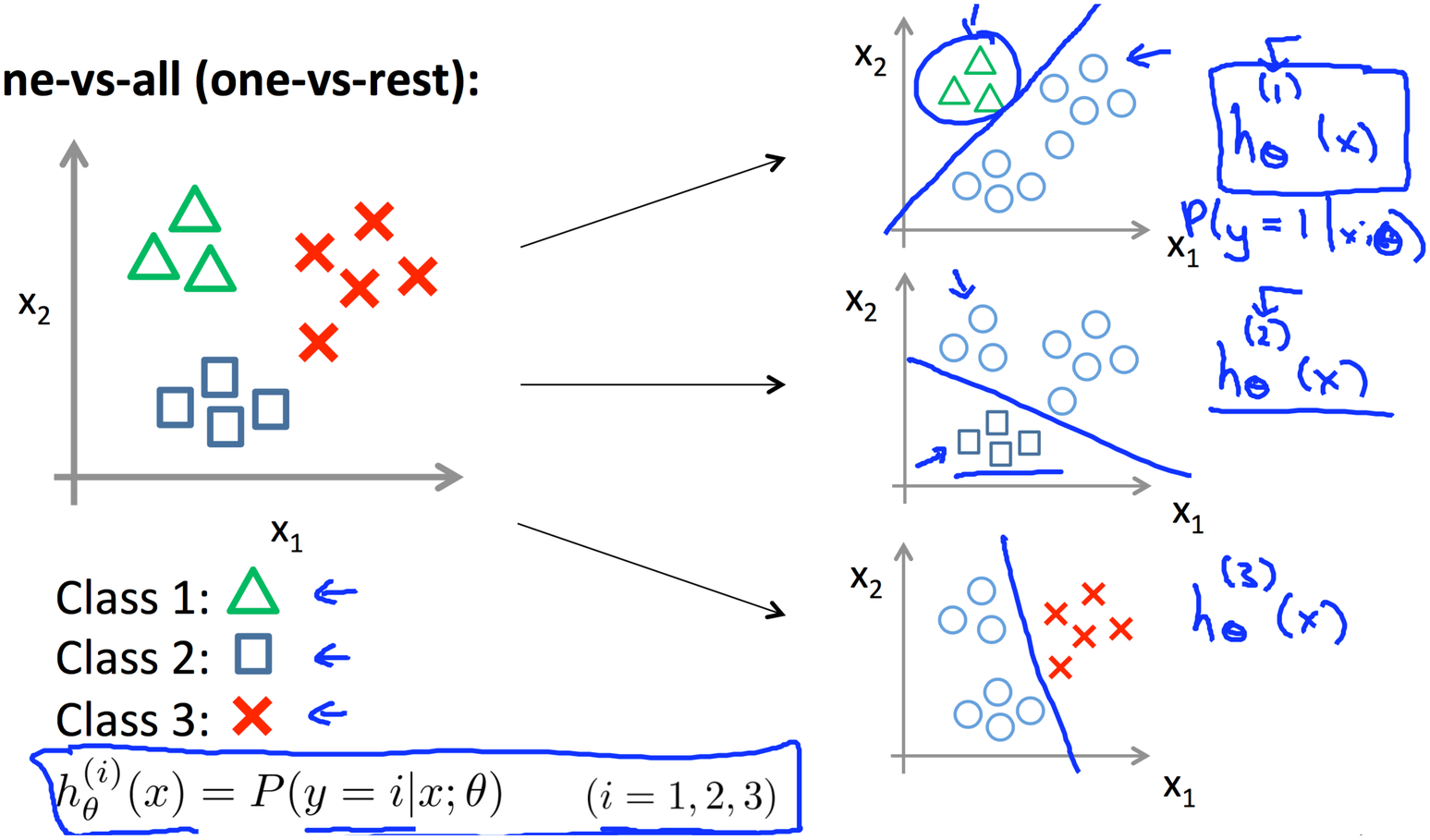
Multiclass Classification

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-vs-all (one-vs-rest):



Linear Discriminant Analysis

Linear Discriminant Analysis

Linear Discriminate Analysis (LDA) fits a conditional probability density function (Gaussian) to each attribute for the class, the discrimination function between the classes is linear.

This recipe shows the fitting of an LDA algorithm to the iris dataset.

```
# Linear Discriminant Analysis
from sklearn import datasets
from sklearn import metrics
from sklearn.lda import LDA
# load the iris datasets
dataset = datasets.load_iris()
# fit a LDA model to the data
model = LDA()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

Quadratic Discriminant Analysis

Quadratic Discriminant Analysis

Quadratic Discriminant Analysis (QDA) fits a conditional probability density function (Gaussian) to each attribute for the class, the discrimination function between the classes is quadratic.

This recipe shows the fitting of an QDA algorithm to the iris dataset.

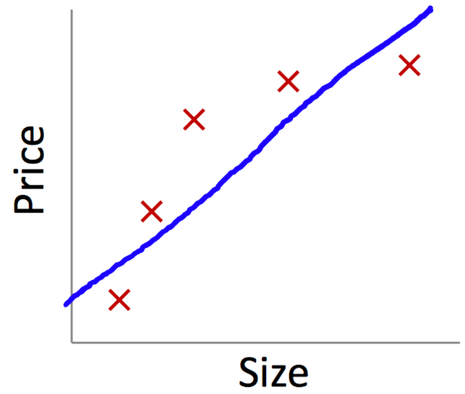
```
# Quadratic Discriminant Analysis
from sklearn import datasets
from sklearn import metrics
from sklearn.qda import QDA
# load the iris datasets
dataset = datasets.load_iris()
# fit a QDA model to the data
model = QDA()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

Regularization

Machine learning models need to generalize well to new examples that the model has not seen in practice. Regularization helps prevent models from overfitting the training data.

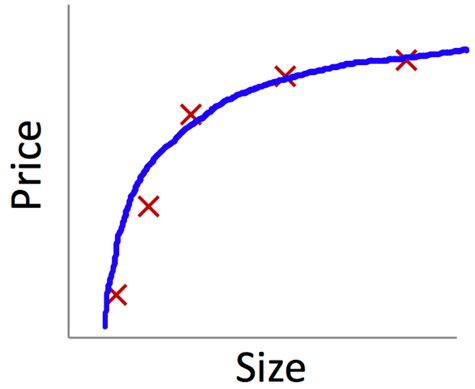
The problem of overfitting

Example: Linear regression (housing prices)



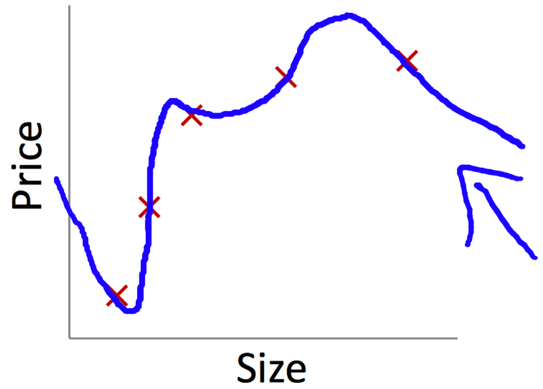
$\rightarrow \theta_0 + \theta_1 x$

"Underfit" "High bias"



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

"Just right"



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

"Overfit" "High variance"

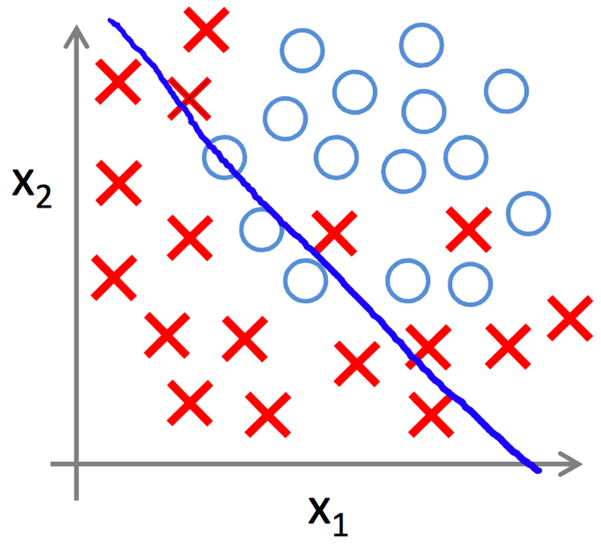
Overfitting: If we have too many features, the learned hypothesis may fit the training set very well ($J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \approx 0$), but fail to generalize to new examples (predict prices on new examples).

Regularization

Machine learning models need to generalize well to new examples that the model has not seen in practice. Regularization helps prevent models from overfitting the training data.

The problem of overfitting

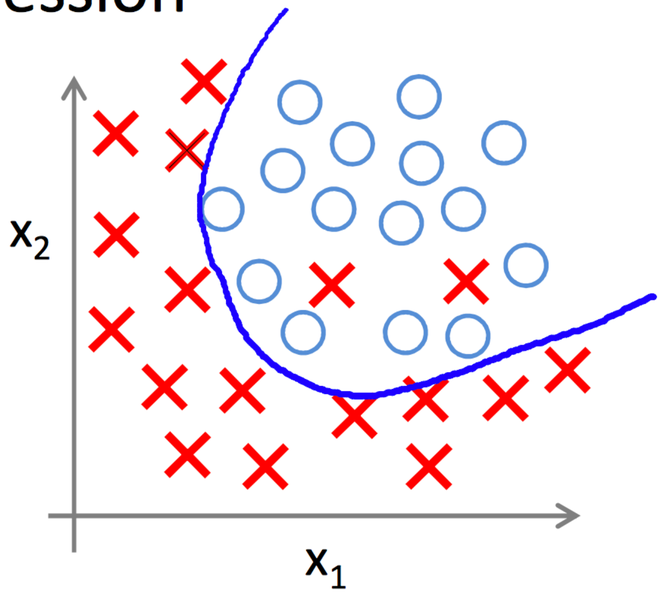
Example: Logistic regression



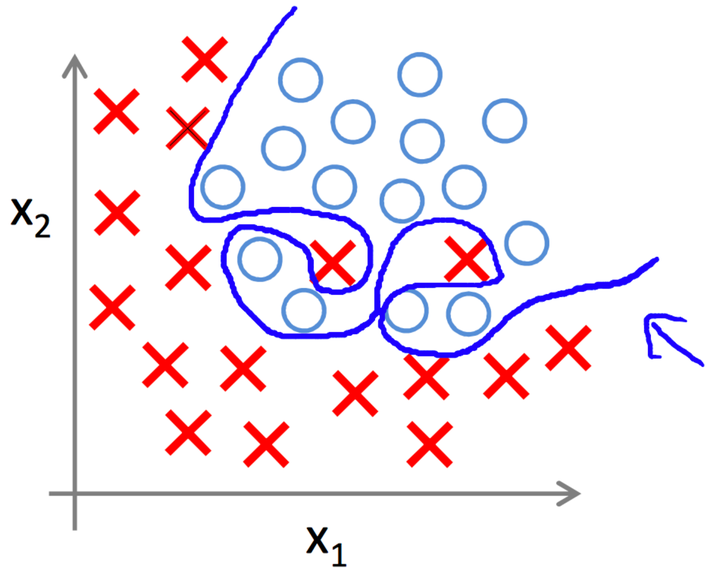
$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$

(g = sigmoid function)

"Underfit"



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



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

"Overfit"

Regularization

Addressing Overfitting

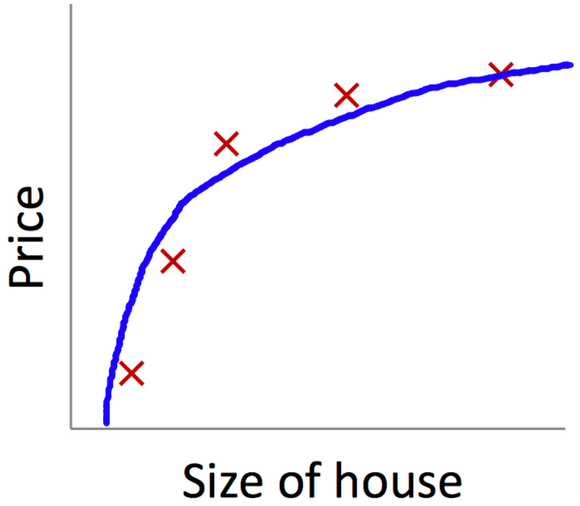
Options:

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

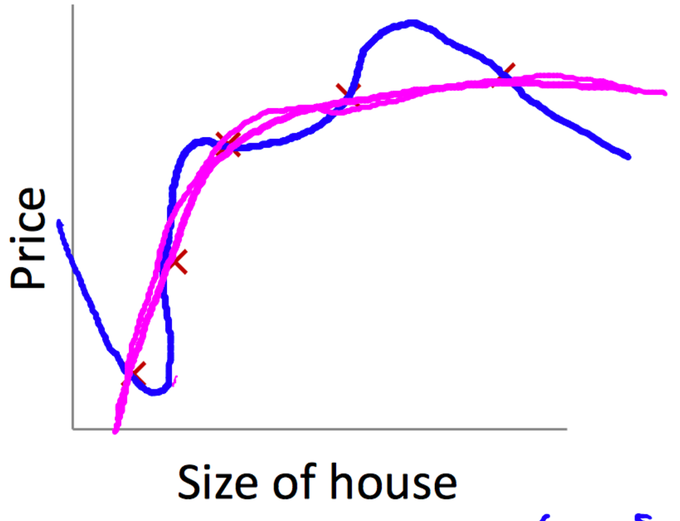
Regularization

Cost function

Intuition



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



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

Suppose we penalize and make θ_3, θ_4 really small.

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

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

Regularization

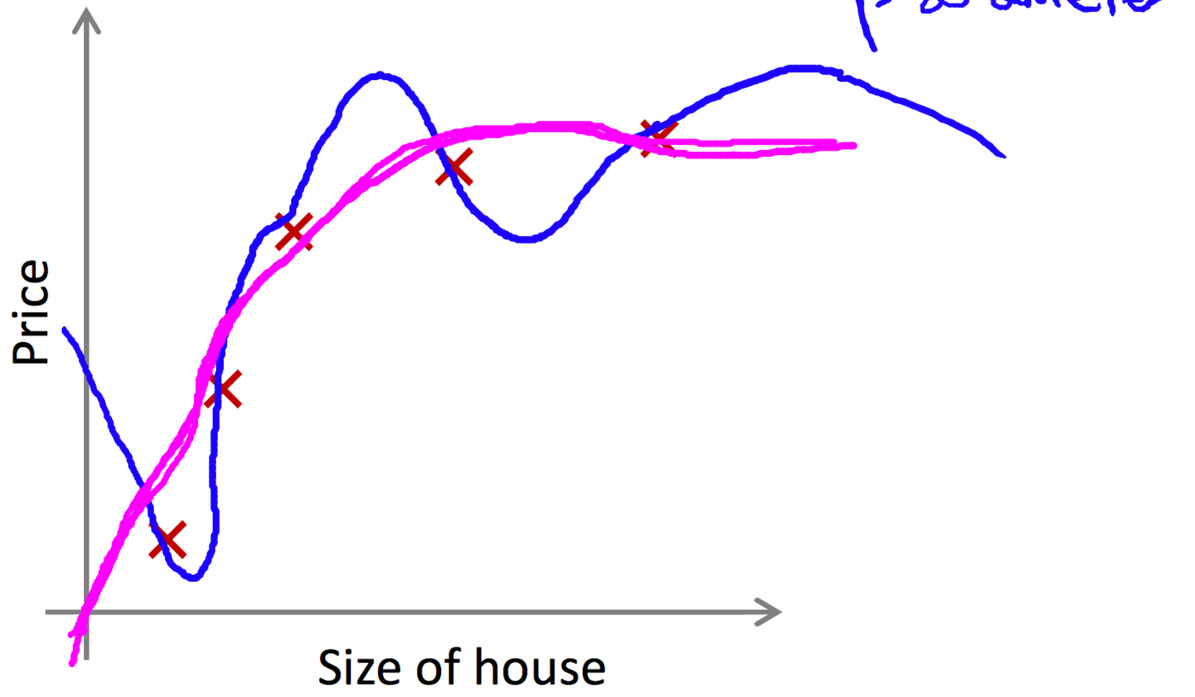
Cost function

Regularization.

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

regularization parameter

$$\min_\theta J(\theta)$$



Regularization

In regularized linear regression, we choose θ to minimize

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

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

- Algorithm works fine; setting λ to be very large can't hurt it
- Algortihm fails to eliminate overfitting.
- Algorithm results in underfitting. (Fails to fit even training data well).
- Gradient descent will fail to converge.

Regularization

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

$$\min_\theta J(\theta)$$

Regularized Linear Regression

Gradient descent

Repeat {

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

$$\theta_j := \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j\right] \quad (j = 1, 2, 3, \ldots, n)$$

}

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

$1 - \alpha\frac{\lambda}{m} < 1$

Normal equation

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

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

$$\min_\theta J(\theta)$$

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

$(n+1) \times (n+1)$

e.g. $n = 2$ $\quad \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

should be invertible

** if non-invertible/singular use >> numpy.linalg.pinv (Moore-Penrose Matrix Inverse)

https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.linalg.pinv.html

Regularization

Regularized Logistic Regression

Gradient descent

Repeat $\{$

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

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

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

$\theta_1 \ldots \theta_n$

$\}$

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

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

Regularization

Advanced Optimization for Regularization

We need code to compute

Advanced optimization

```
function [jVal, gradient] = costFunction(theta)
    jVal = [code to compute J(θ)];
    gradient(1) = [code to compute ∂/∂θ_0 J(θ)];
    gradient(2) = [code to compute ∂/∂θ_1 J(θ)];
    gradient(3) = [code to compute ∂/∂θ_2 J(θ)];
    :
    gradient(n+1) = [code to compute ∂/∂θ_n J(θ)];
```

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

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

$$\left(\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)} \right) - [\blacksquare]$$

$$\left(\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)} \right) - [\blacksquare]$$

fminunc (@costFunction, ...)

$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$ theta(1) theta(2) theta(n+1)

J(θ)

Regularization

Ridge Regression

Ridge Regression (also known as Tikhonov regularization) penalizes a least squares regression model on the square of the absolute magnitude of the coefficients (called the L2 norm).

This recipe shows use of the Ridge Regression algorithm to make predictions for the diabetes dataset.

```
# Ridge Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import Ridge
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a ridge regression model to the data
model = Ridge(alpha=0.1)
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

L1-norm is also known as least absolute deviations (LAD), least absolute errors (LAE). It is basically minimizing the sum of the absolute differences **(S)** between the target value (Y_i) and the estimated values ($f(x_i)$):

$$S = \sum_{i=1}^{n} |y_i - f(x_i)|.$$

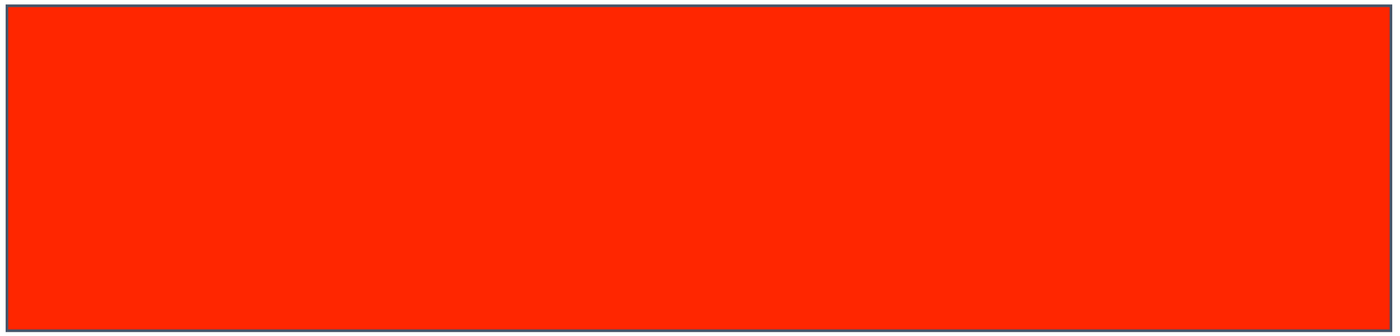
Regularization

Least Absolute Shrinkage and Selection Operator

The Least Absolute Shrinkage and Selection Operator (LASSO) is a regularization method that penalizes a least squares regression model on the absolute magnitude of the coefficients (called the L1 norm).

This recipe shows use of the LASSO algorithm to make predictions for the diabetes dataset.

```
# Lasso Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import Lasso
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a LASSO model to the data
model = Lasso(alpha=0.1)
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```



L2-norm is also known as least squares. It is basically minimizing the sum of the square of the differences **(S)** between the target value (Y_i) and the estimated values ($f(x_i)$):

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Regularization

Least Angle Regression

The Least Angle Regression (LARS) method is a computationally efficient algorithm for fitting a regression model. It is useful for high-dimensional data and is commonly used in conjunction with regularization (such as LASSO).

This recipe shows use of the LARS algorithm to make predictions for the diabetes dataset.

```
# LARS Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import Lars
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a LARS model to the data
model = Lars()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Regularization

LASSO LARS

The Least Angle Regression (LARS) can be used as an alternative method for calculating Least Absolute Shrinkage and Selection Operator (LASSO) fit.

This recipe shows use of the LASSO with LARS algorithm to make predictions for the diabetes dataset.

```
# LassoLars Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LassoLars
# load the iris datasets
dataset = datasets.load_diabetes()
# fit a LASSO using LARS model to the data
model = LassoLars(alpha=0.1)
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Regularization

ElasticNet

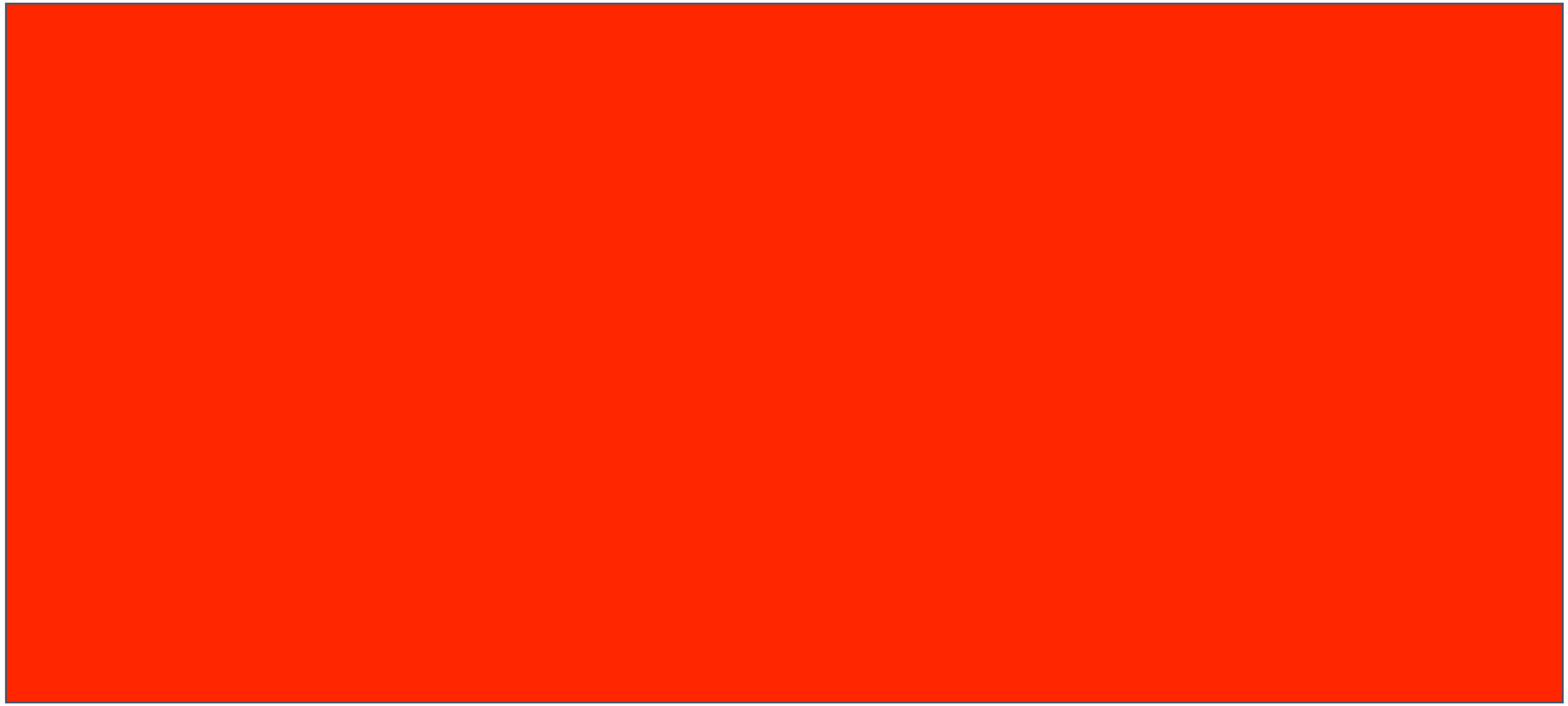
The ElasticNet is a regularized form of regression that penalizes the model with both the L1 norm and the L2 norm.

This recipe shows use of the ElasticNet algorithm to make predictions for the diabetes dataset.

```
# ElasticNet Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import ElasticNet
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a model to the data
model = ElasticNet(alpha=0.1)
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Support Vector Machines

SVMs are considered by many to be the most powerful 'black box' learning algorithm, and by posing a cleverly-chosen optimization objective, one of the most widely used learning algorithms today.



Optimization objective

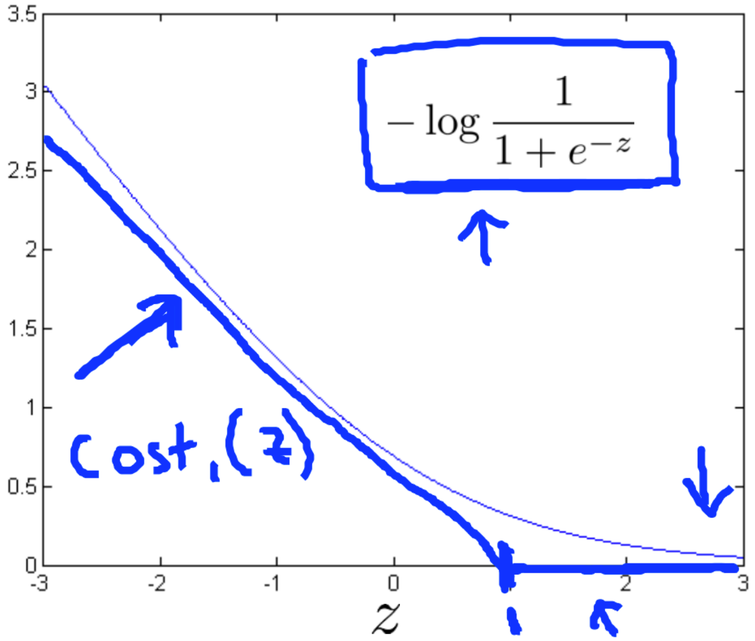
Large Margin Classification

Alternative view of logistic regression

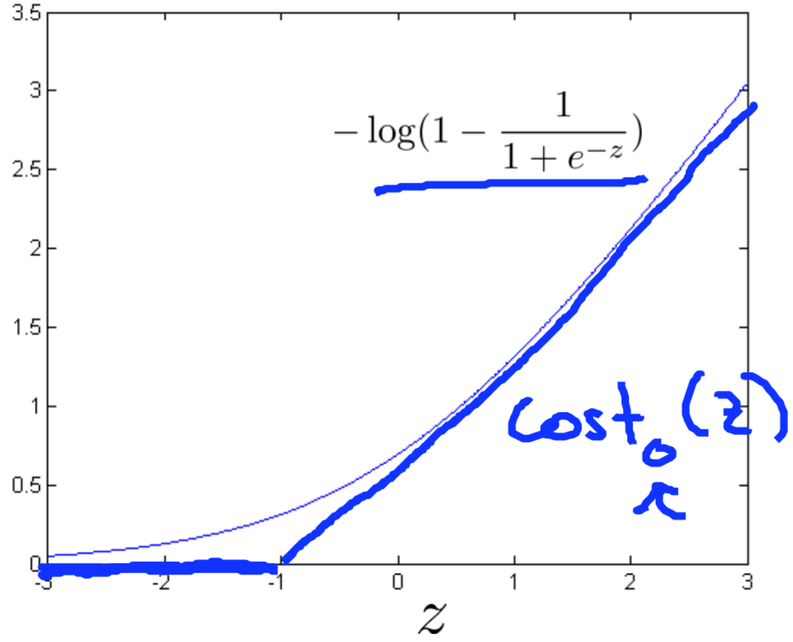
Cost of example: $-(y \log h_\theta(x) + (1-y)\log(1-h_\theta(x)))$

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

If $y = 1$ (want $\theta^T x \gg 0$):



If $y = 0$ (want $\theta^T x \ll 0$):



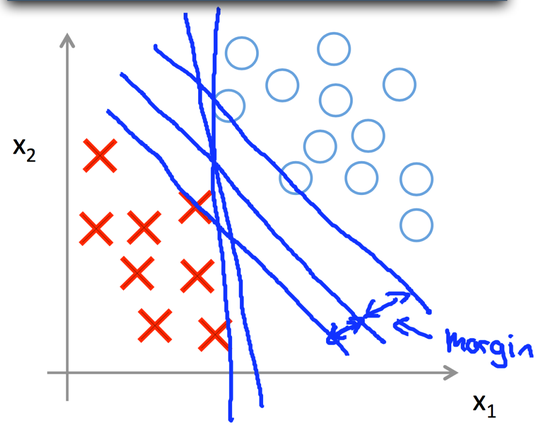
$$\min_\theta \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\left(-\log h_\theta(x^{(i)})\right) + (1-y^{(i)})\left(\left(-\log(1-h_\theta(x^{(i)}))\right)\right]+\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

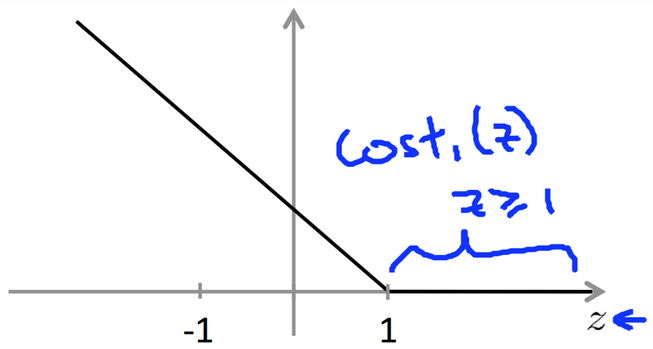
Op

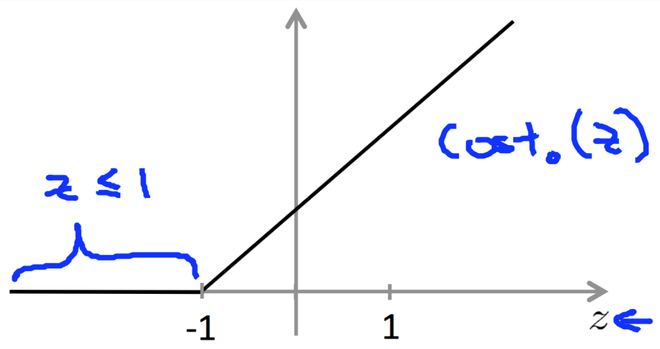
Support vector machine:

Support Vector Machine

Linearly separable cases







θ_j^2

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

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

$\theta^T x \geqslant 1$

$\theta^T x \leqslant -1$

Large Margin Classification

Mathematics behind large margin classification

SVM Decision Boundary

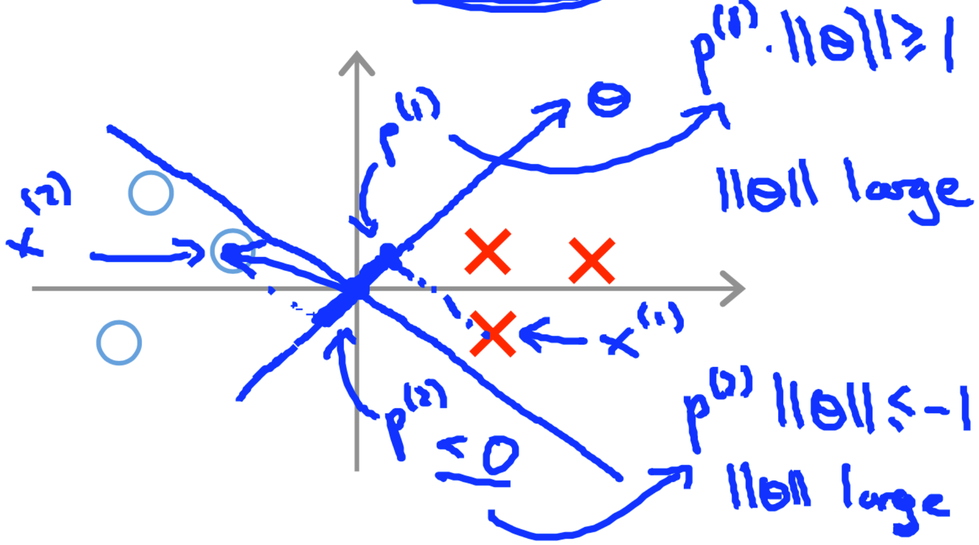
$$\min_{\theta} \frac{1}{2} \sum_{j=1}^{n} \theta_j^2 = \frac{1}{2} \|\theta\|^2$$

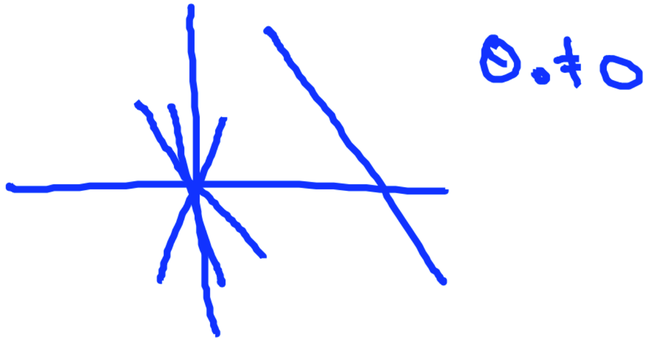
$$\text{s.t.} \quad \begin{cases} p^{(i)} \cdot \|\theta\| \geq 1 & \text{if } y^{(i)} = 1 \\ p^{(i)} \cdot \|\theta\| \leq -1 & \text{if } y^{(i)} = 1 \end{cases}$$

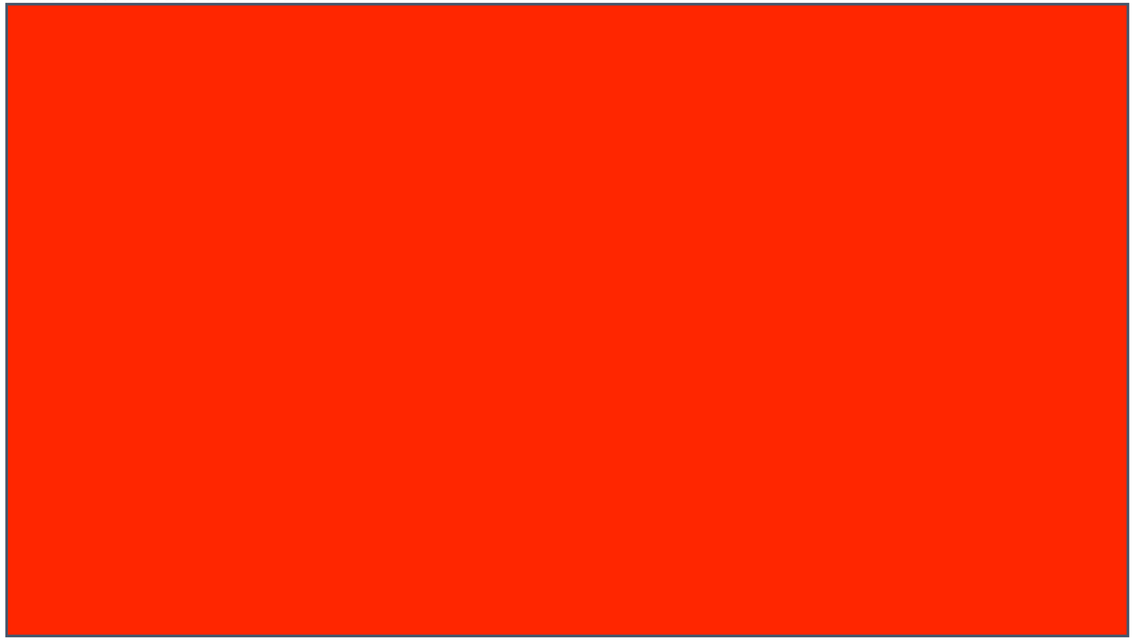
C very large

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

Simplification: $\theta_0 = 0$

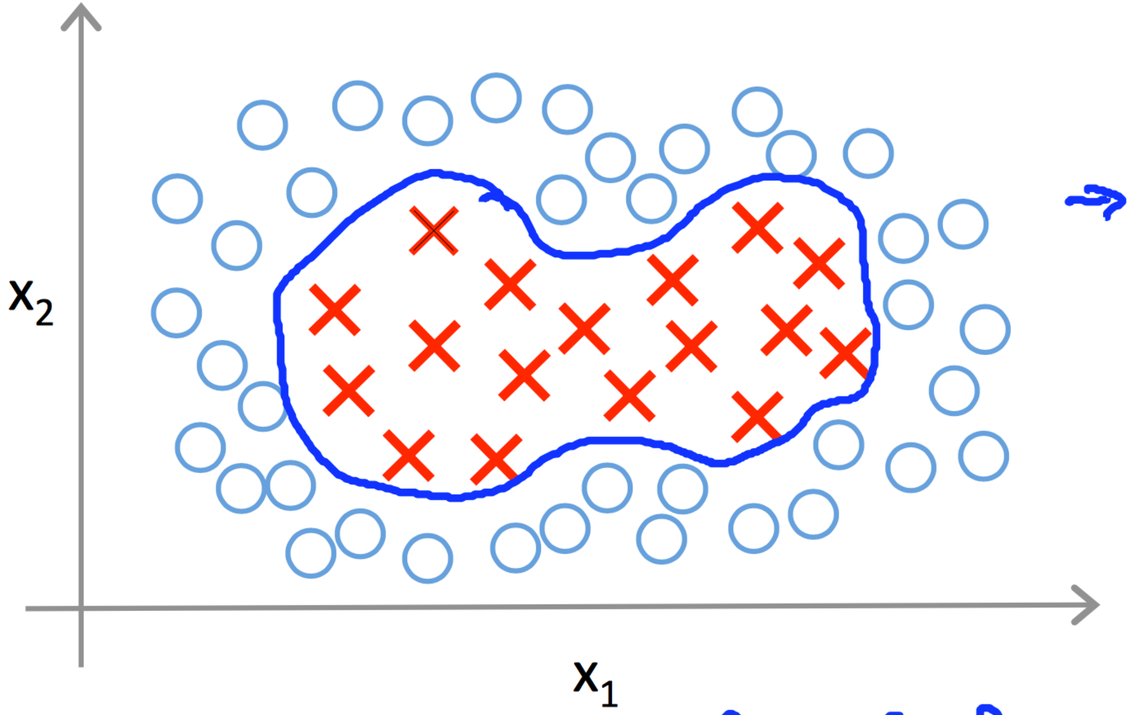






SVM Kernels

Non-linear Decision Boundary



Predict $y = 1$ if

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \theta_5 x_2^2 + \cdots \geq 0$$

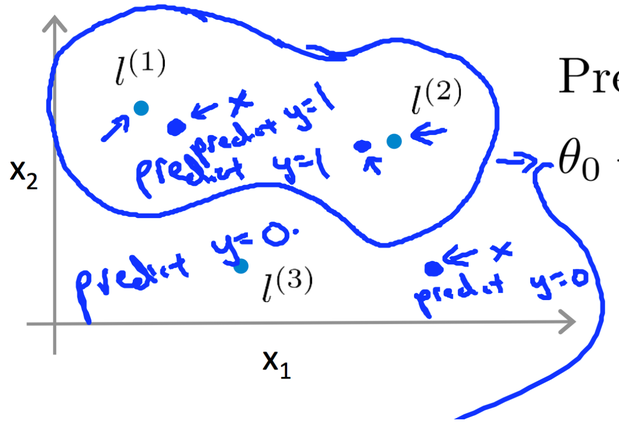
$h_\theta(x) = \begin{cases} 1 & \text{if } \theta_0 + \theta_1 x_1 + \cdots \geq 0 \\ 0 & \text{otherwise} \end{cases}$

$\rightarrow \theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 + \cdots$

$f_1 = x_1, \quad f_2 = x_2, \quad f_3 = x_1 x_2, \quad f_4 = x_1^2, \quad f_5 = x_2^2, \ldots$

Is there a different / better choice of the features $f_1, f_2, f_3, \ldots$?

SVM Kernels



Predict "1" when

$$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0$$

$\theta_0 = -0.5$, $\theta_1 = 1$, $\theta_2 = 1$, $\theta_3 = 0$

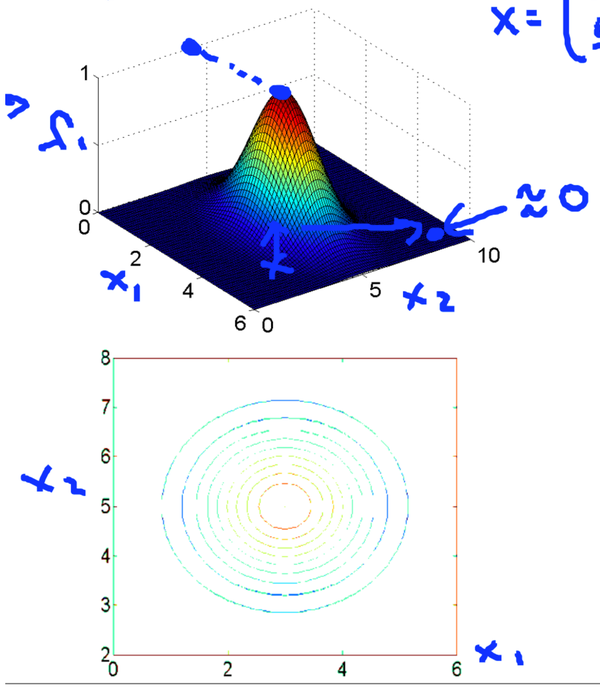
Example:

$$l^{(1)} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

$$f_1 = \exp\left(-\frac{\|x - l^{(1)}\|^2}{2\sigma^2}\right)$$

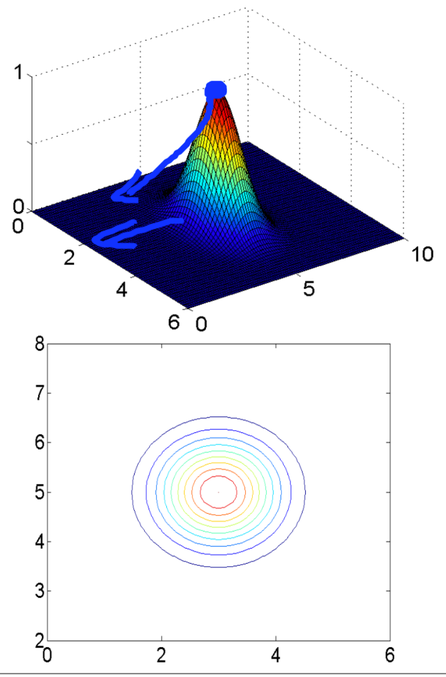
$x = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$

$\sigma^2 = 1$

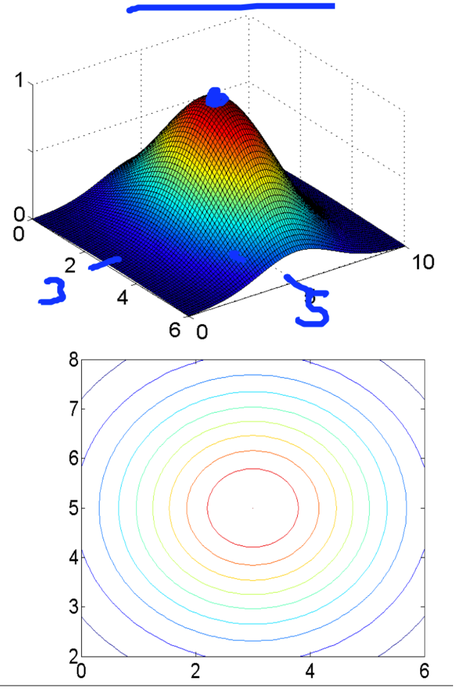


≈ 0

$\sigma^2 = 0.5$



$\sigma^2 = 3$



Using an SVM

SVMs in Practice

[redacted]

[redacted]

Need to specify:

→ Choice of parameter C.

Choice of kernel (similarity function):

E.g. No kernel ("linear kernel")

Predict "y = 1" if $\theta^T x \geq 0$

$\theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n \geq 0$

→ n large, m small

$x \in \mathbb{R}^{n+1}$

[redacted]

$x \in \mathbb{R}^n$, n small

and/or m large

SVMs in Practice

Support Vector Machines

Support Vector Machines (SVM) are a method that uses points in a transformed problem space that best separate classes into two groups. Classification for multiple classes is supported by a one-vs-all method. SVM also supports regression by modeling the function with a minimum amount of allowable error.

This recipe shows use of the SVM algorithm to make predictions for the iris dataset (classification).

```
# SVM Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.svm import SVC
# load the iris datasets
dataset = datasets.load_iris()
# fit a SVM model to the data
model = SVC()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the SVM algorithm to make predictions for the diabetes dataset (regression).

```
# SVM Regression
import numpy as np
from sklearn import datasets
from sklearn.svm import SVR
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a SVM model to the data
model = SVR()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Naive Bayes

Naive Bayes

Naive Bayes uses Bayes Theorem to model the conditional relationship of each attribute to the class variable.

This recipe shows the fitting of an Naive Bayes algorithm to the iris dataset.

```
# Gaussian Naive Bayes
from sklearn import datasets
from sklearn import metrics
from sklearn.naive_bayes import GaussianNB
# load the iris datasets
dataset = datasets.load_iris()
# fit a Naive Bayes model to the data
model = GaussianNB()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

K-Nearest Neighbors

k-Nearest Neighbor

The k-Nearest Neighbor (kNN) method makes predictions by locating similar cases to a given data instance (using a similarity function) and returning the average or majority of the most similar data instances. The kNN algorithm can be used for classification or regression.

This recipe shows use of the kNN algorithm to make predictions for the iris dataset (classification).

```
# k-Nearest Neighbor Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.neighbors import KNeighborsClassifier
# load iris the datasets
dataset = datasets.load_iris()
# fit a k-nearest neighbor model to the data
model = KNeighborsClassifier()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the kNN algorithm to make predictions for the diabetes dataset (regression).

```
# k-Nearest Neighbor Regression
import numpy as np
from sklearn import datasets
from sklearn.neighbors import KNeighborsRegressor
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a model to the data
model = KNeighborsRegressor()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Ensemble Methods

Random Forest

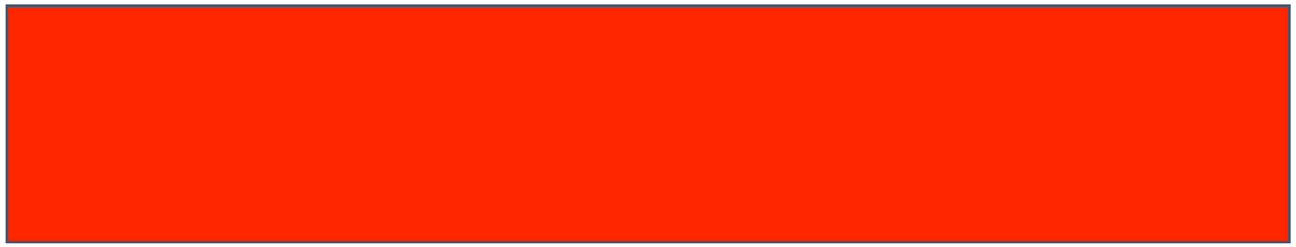
This recipe shows use of the Random Forest algorithm to make predictions for the iris dataset (classification).

```
# Random Forest Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.ensemble import RandomForestClassifier
# load iris the datasets
dataset = datasets.load_iris()
# fit a random forest model to the data
model = RandomForestClassifier()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the Random Forest model to make predictions for the diabetes dataset (regression).

```
# Random Forest Regression
import numpy as np
from sklearn import datasets
from sklearn.ensemble import RandomForestRegressor
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a random forest model to the data
model = RandomForestRegressor()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Extra Trees



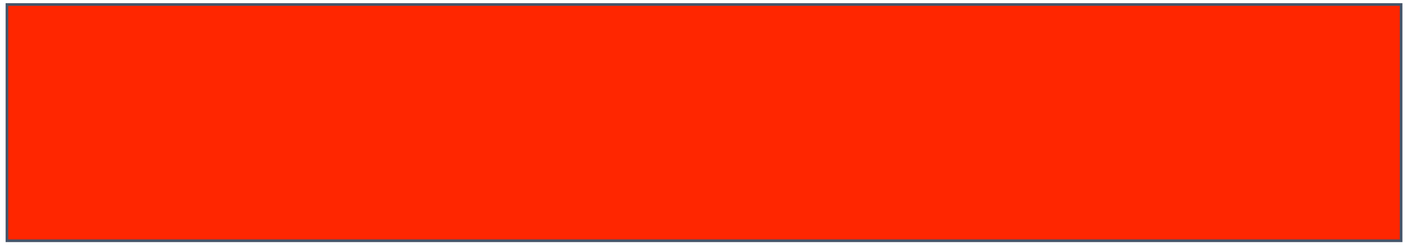
This recipe shows use of the Extra Trees algorithm to make predictions for the iris dataset (classification).

```
# Extra Trees Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.ensemble import ExtraTreesClassifier
# load the iris datasets
dataset = datasets.load_iris()
# fit a Extra Trees model to the data
model = ExtraTreesClassifier()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the Extra Trees algorithm to make predictions for the diabetes dataset (regression).

```
# Extra Trees Regression
import numpy as np
from sklearn import datasets
from sklearn.ensemble import ExtraTreesRegressor
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a extra trees model to the data
model = ExtraTreesRegressor()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Adaptive Boosting



This recipe shows use of the AdaBoost algorithm to make predictions for the iris dataset (classification).

```
# AdaBoost Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.ensemble import AdaBoostClassifier
# load the iris datasets
dataset = datasets.load_iris()
# fit an AdaBoost model to the data
model = AdaBoostClassifier()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the AdaBoost algorithm to make predictions for the diabetes dataset (regression).

```
# Random Forest Regression
import numpy as np
from sklearn import datasets
from sklearn.ensemble import AdaBoostRegressor
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit an AdaBoost model to the data
model = AdaBoostRegressor()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Gradient Boosting

This recipe shows use of the Gradient Boosting algorithm to make predictions for the iris dataset (classification).

```
# Gradient Boosting Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.ensemble import GradientBoostingClassifier
# load the iris dataset
dataset = datasets.load_iris()
# fit a Gradient Boosting model to the data
model = GradientBoostingClassifier()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the Gradient Boosting algorithm to make predictions for the diabetes dataset (regression).

```
# Gradient Boosting Regression
import numpy as np
from sklearn import datasets
from sklearn.ensemble import GradientBoostingRegressor
# load the diabetes dataset
dataset = datasets.load_diabetes()
# fit a Gradient Boosting model to the data
model = GradientBoostingRegressor()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

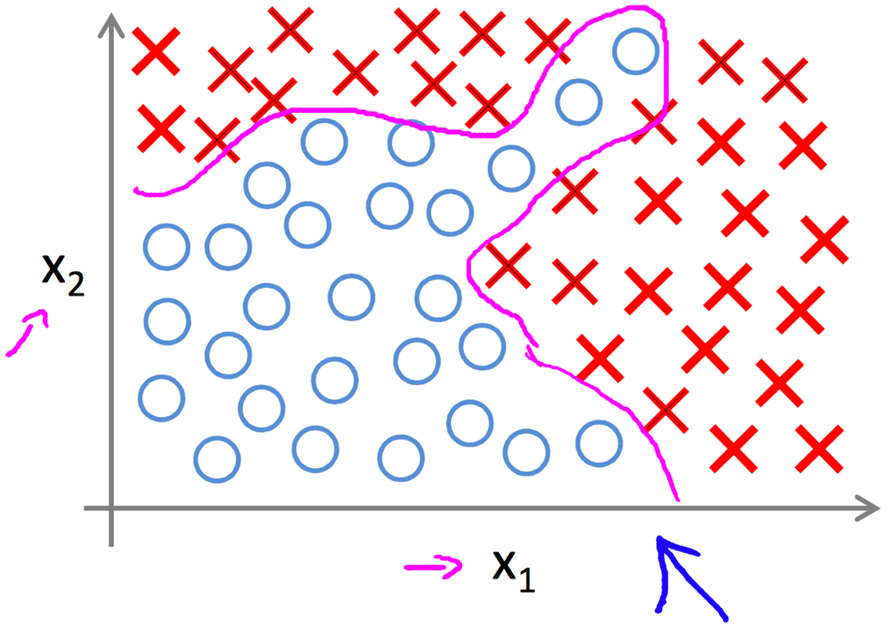
Neural Networks

Neural networks is a model inspired by how the brain works. It is widely used today in many applications: when your phone interprets and understand your voice commands, it is likely that a neural network is helping to understand your speech; when you cash a check, the machines that automatically read the digits also use neural networks.

Motivations

Issues with Non-linear Classification

Non-linear Classification



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

$x_1^2, x_1x_2, x_1x_3, x_1x_4 \dots x_1x_{100}$

$x_2^2, x_2x_3 \dots$

≈ 5000 feature $\quad O(n^2)$

$\approx \frac{n^2}{2}$

$x_1^2, x_2^2, x_3^2, \dots, x_{100}^2$

$x_1x_2x_3, \; x_1^2x_2, \; x_{10}x_{11}x_{17}, \dots$

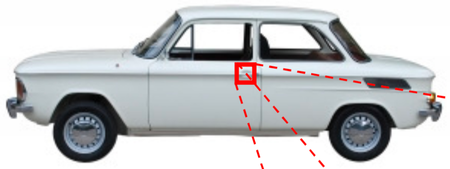
$n = 100$

x_1 = size
x_2 = # bedrooms
x_3 = # floors
x_4 = age
$\dots$
x_{100}

Motivations

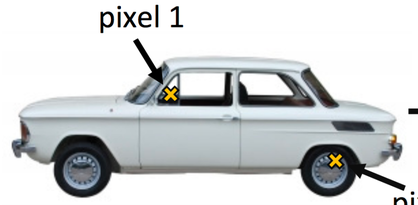
What is this?

You see this:

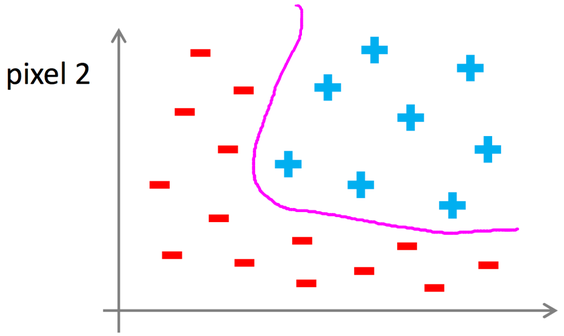


But the camera sees this:

194	210	201	212	199	213	215	195	178	158	182	209
180	189	190	221	209	205	191	167	147	115	129	163
114	126	140	188	176	165	152	140	170	106	78	88
87	103	115	154	143	142	149	153	173	101	57	57
102	112	106	131	122	138	152	147	128	84	58	66
94	95	79	104	105	124	129	113	107	87	69	67
68	71	69	98	89	92	98	95	89	88	76	67
41	56	68	99	63	45	60	82	58	76	75	65
20	43	69	75	56	41	51	73	55	70	63	44
50	50	57	69	75	75	73	74	53	68	59	37
72	59	53	66	84	92	84	74	57	72	63	42
67	61	58	65	75	78	76	73	59	75	69	50



Learning Algorithm



50 x 50 pixel images→ 2500 pixels

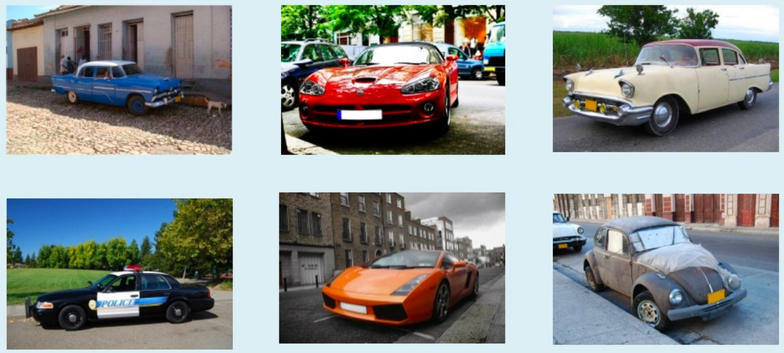
$n = 2500$ (7500 if RGB)

$$x = \begin{bmatrix} \text{pixel 1 intensity} \\ \text{pixel 2 intensity} \\ \vdots \\ \text{pixel 2500 intensity} \end{bmatrix}$$

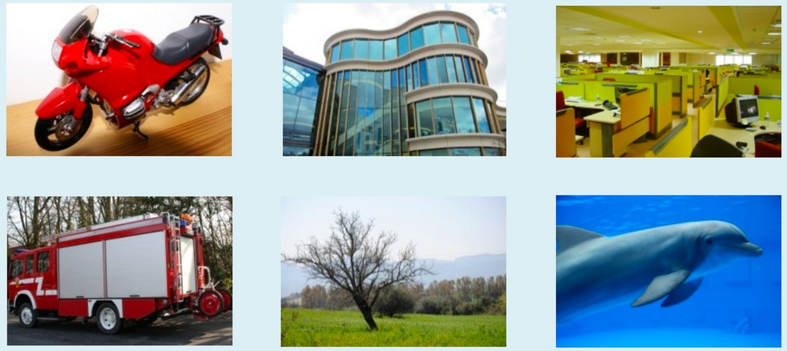
0-255

Quadratic features ($x_i \times x_j$)

$\approx \frac{n^2}{2}$



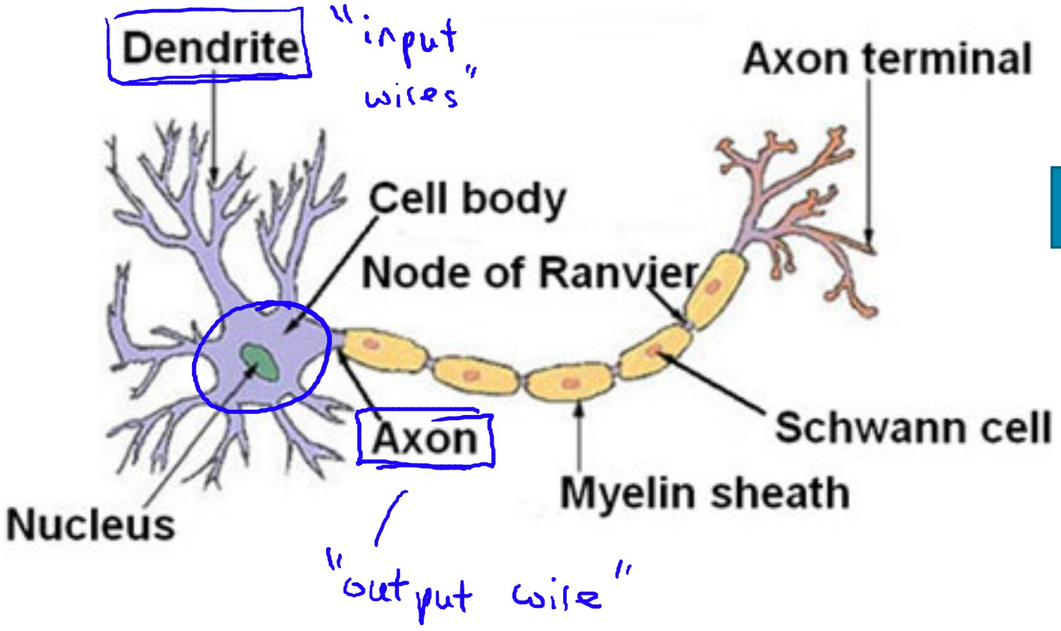
Cars



Not a car

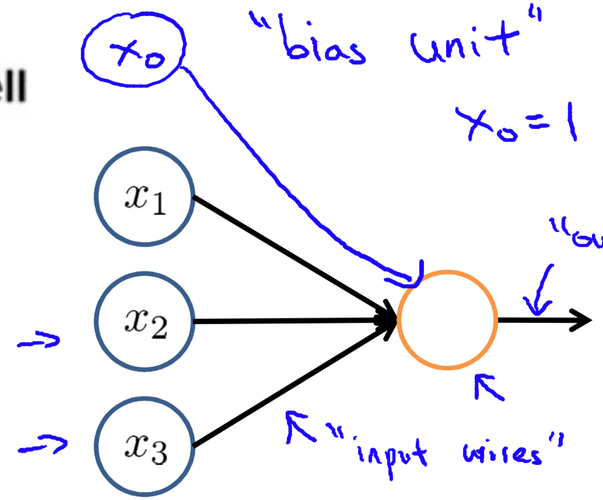
Motivations

Neural Networks perform better for non-linear data with number of features



Non-linear Hypotheses

euron model: Logistic unit



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

"weights" ← (parameters)

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

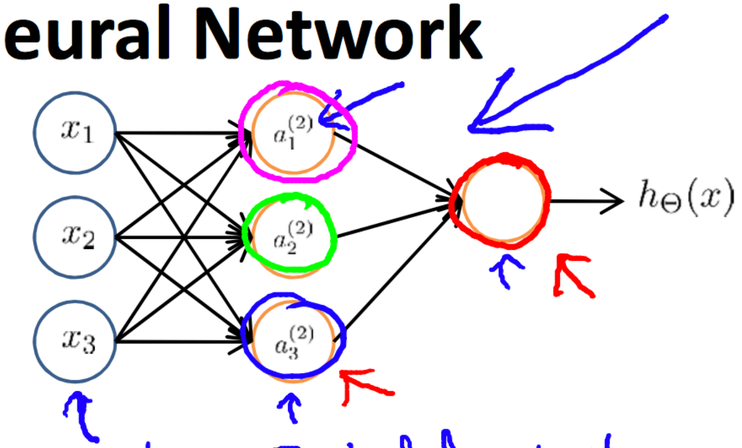
Sigmoid (logistic) activation function.

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

Neural Networks

Model representation I

Neural Network



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

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

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

$h_\Theta(x)$

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

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

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

$a_i^{(2)}$

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

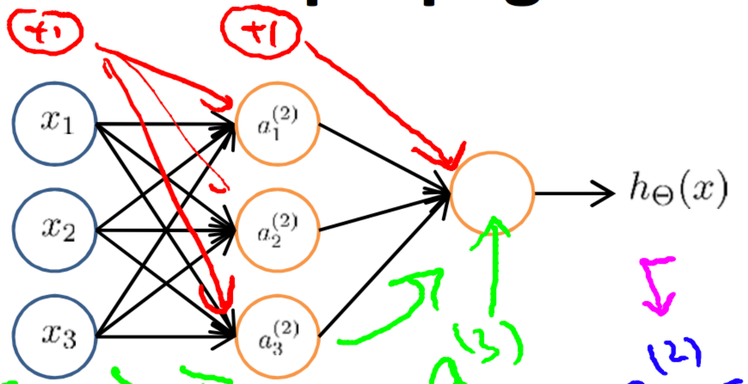
If network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$.

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

Model representation II

Neural Networks

Forward propagation: Vectorized implementation



$a^{(1)} = x$

$a_1^{(2)} = g(z_1^{(2)})$

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

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

$a_3^{(2)} = g(z_3^{(2)})$

$z^{(3)}$

$$
x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}
$$

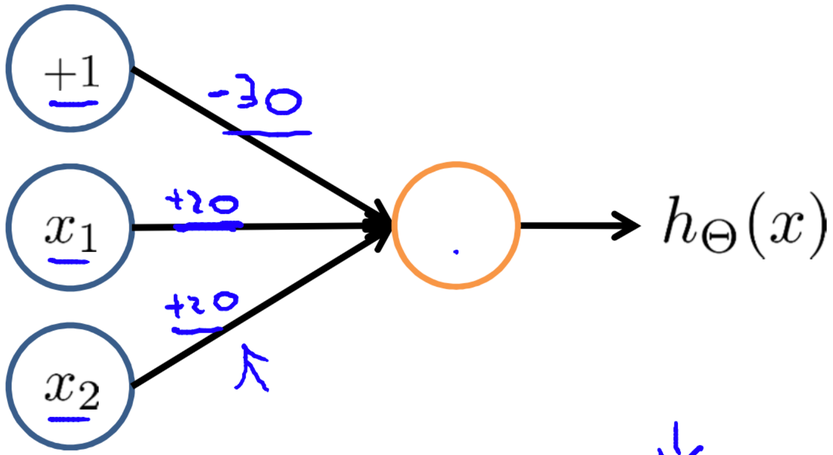
$$
\begin{aligned}
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \qquad \mathbb{R}^3 \qquad \mathbb{R}^3 \\
&\text{Add } a_0^{(2)} = 1. \rightarrow a^{(2)} \in \mathbb{R}^4 \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
h_\Theta(x) &= a^{(3)} = g(z^{(3)})
\end{aligned}
$$

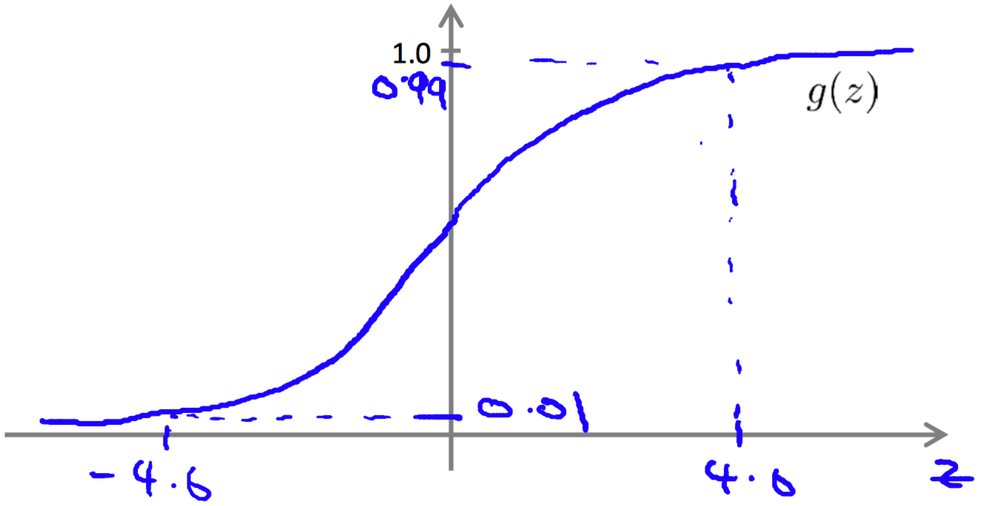
Neural Networks Applications

Example and intuition I

Simple example: AND

- $x_1, x_2 \in \{0, 1\}$
- $y = x_1 \text{ AND } x_2$





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

$\Theta^{(1)}_{10}$ $\Theta^{(1)}_{11}$ $\Theta^{(1)}_{12}$

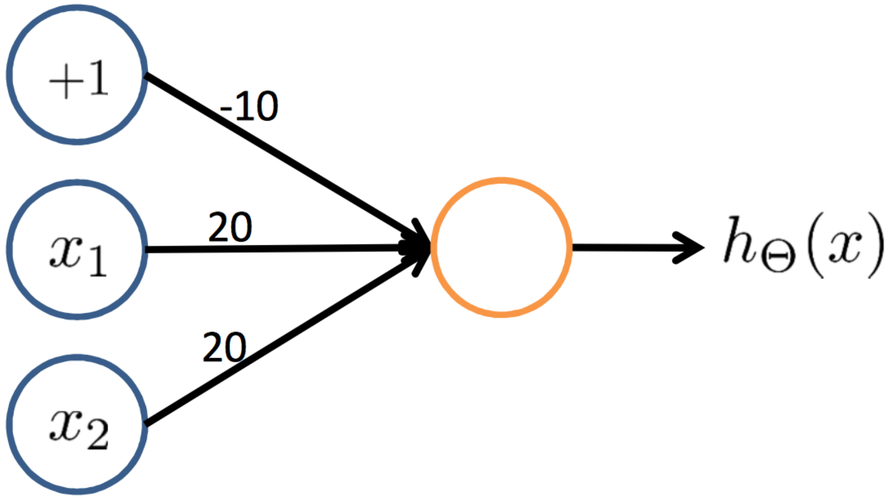
x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

$h_\Theta(x) \approx x_1 \text{ AND } x_2$

Neural Networks Applications

Example and intuition I

Example: OR function



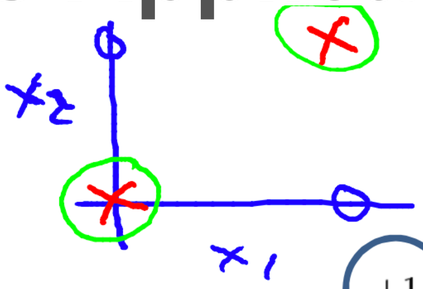
$g(-10 + 20x_1 + 20x_2)$

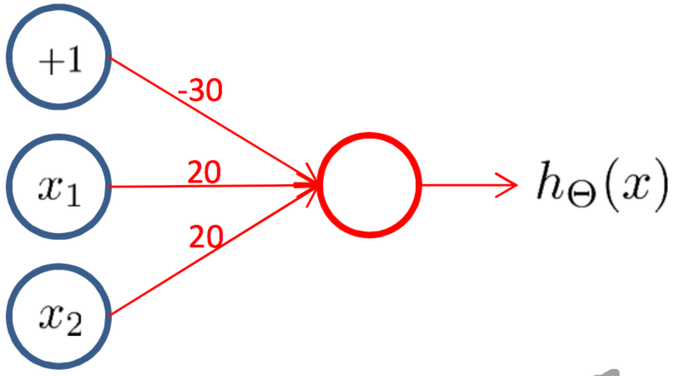
x_1	x_2	$h_\Theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	≈ 1
1	1	≈ 1

Example and intuition II

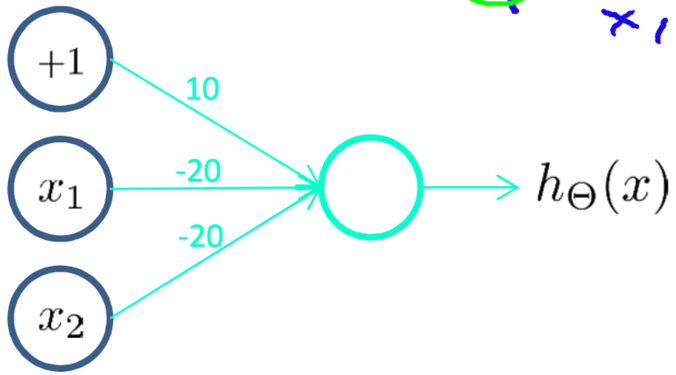
Neural Networks Applications

Putting it together:

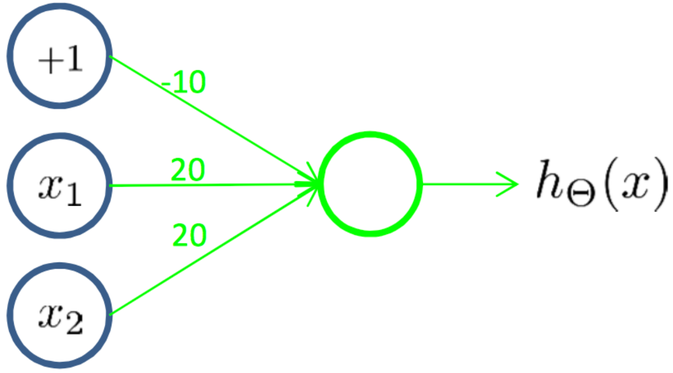




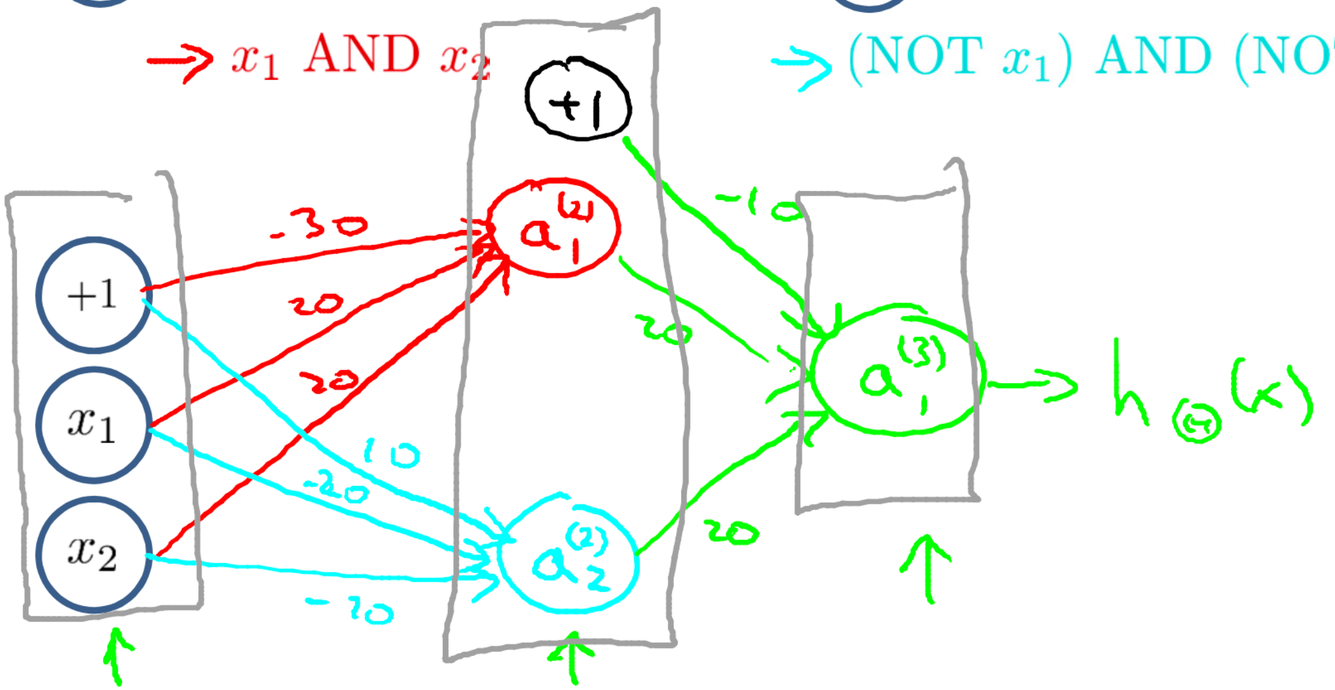
$\rightarrow x_1$ AND x_2



$\rightarrow$ (NOT x_1) AND (NOT x_2)



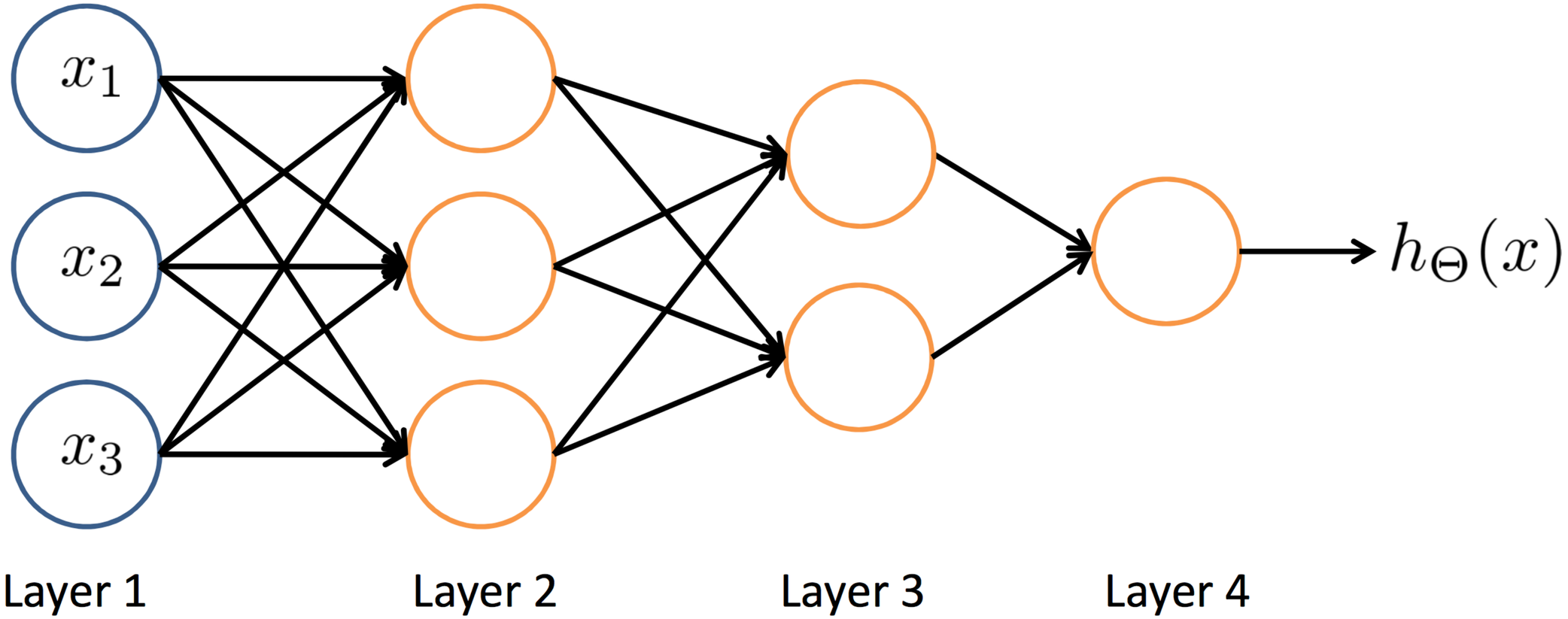
$\rightarrow x_1$ OR x_2



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

Neural Networks Applications

Neural Network intuition



Multiclass classification

Neural Networks Applications

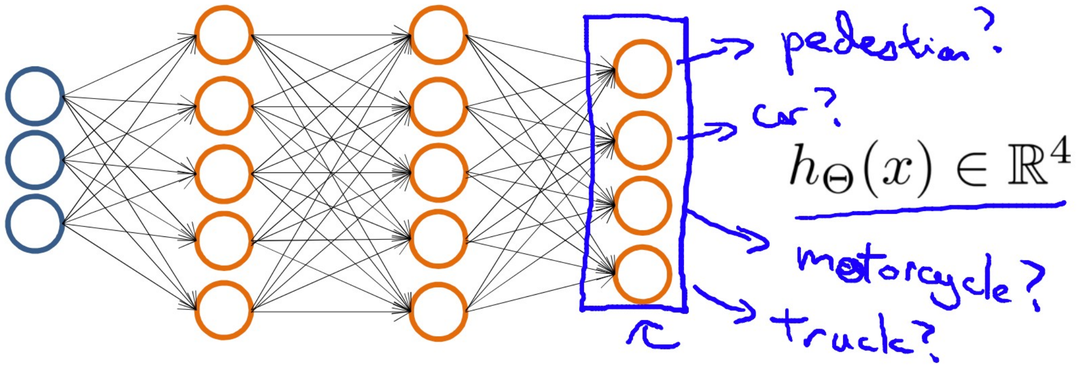
Multiple output units: One-vs-all.

Pedestrian

Car

Motorcycle

Truck



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

when pedestrian when car when motorcycle

$y^{(i)}$ 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

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

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

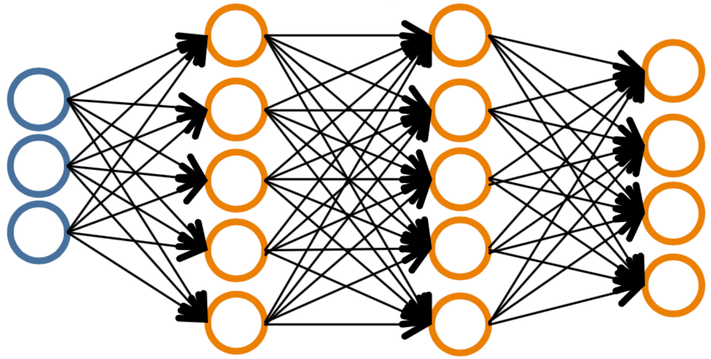
$h_\Theta(x^{(i)}) \approx y^{(i)}$

$\mathbb{R}^4$

Neural Networks: Learning

The Neural Network is one of the most powerful learning algorithm
linear classifier doesn't work)

Neural Network (Classification)



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

$L =$ total no. of layers in network

$s_l =$ no. of units (not counting bias unit) in layer l

Binary classification

$y = 0 \text{ or } 1$

1 output unit

Multi-class classification (K classes)

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

K output units

Cost Function

Cost Function and Backpropagation

Cost function

Logistic regression:

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

Cost Function and Backpropagation

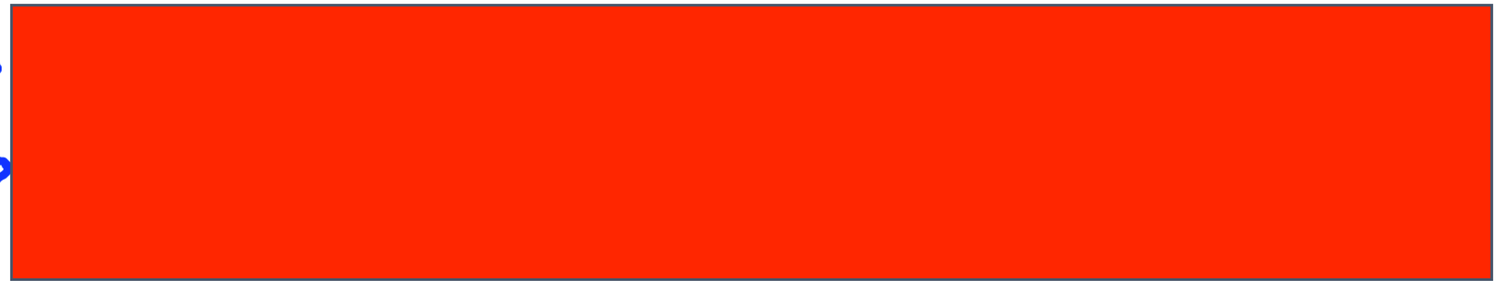
Gradient computation

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

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

$$\min_{\Theta} J(\Theta)$$

Need code to compute:



Cost Function and Backpropagation

Gradient computation

Given one training example (x, y):

Forward propagation:

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

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

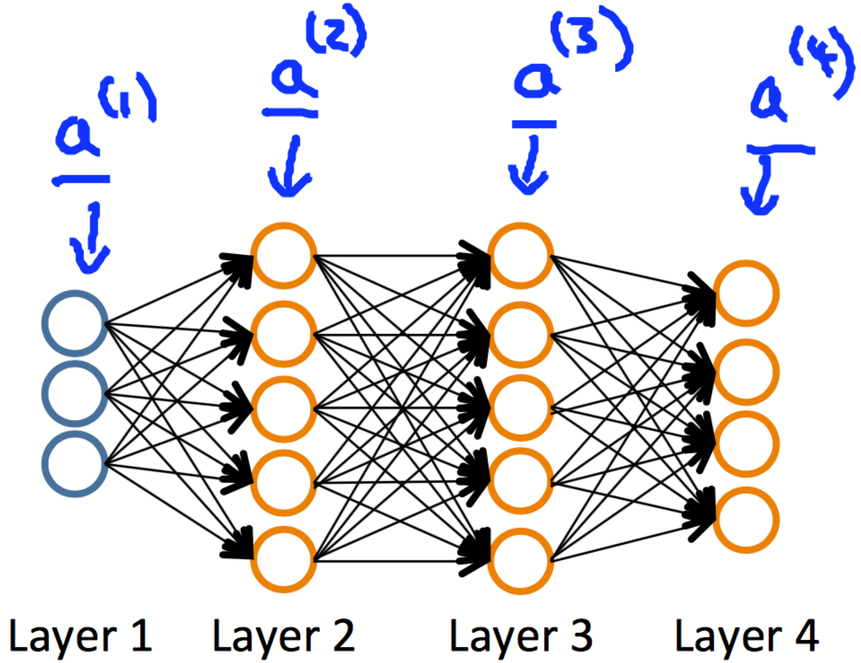
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$

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

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

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

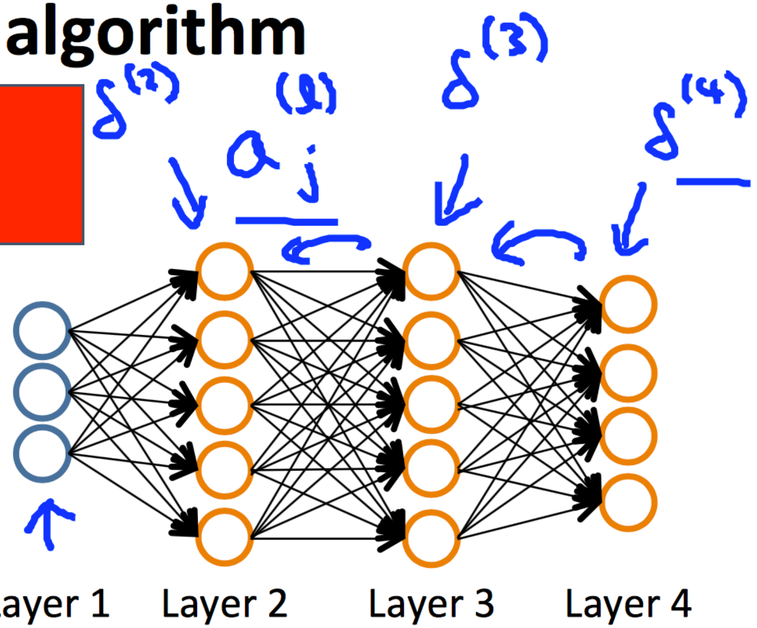
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$



Cost Function and Backpropagation

Backpropagation Algorithm

Gradient computation: Backpropagation algorithm



For each output unit (layer L = 4)

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

$(h_\Theta(x))_j$ $\delta^{(4)} = a^{(4)} - y$

(Do $\delta^{(3)}$)

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$

(ignore λ; if $\lambda = 0$)

Cost Function and Backpropagation

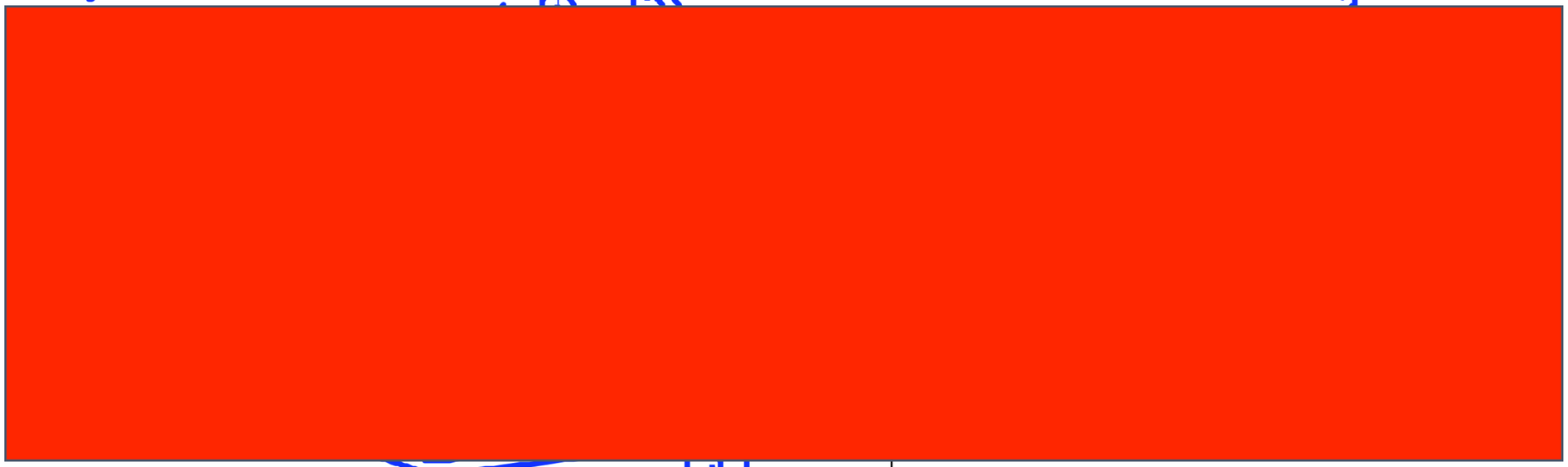
Backpropagation Algorithm

Backpropagation algorithm

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

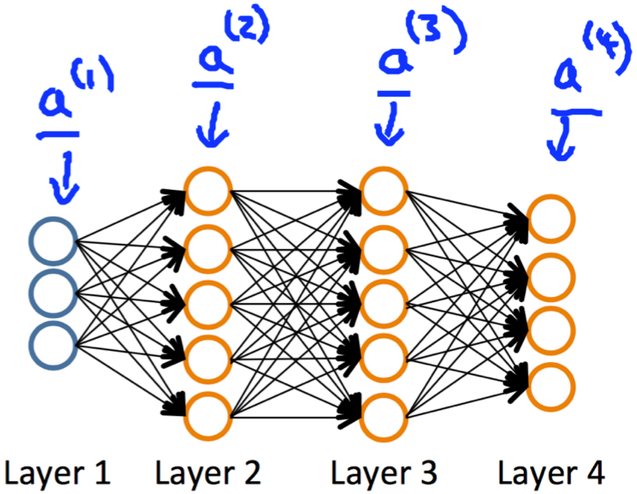
Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j).

(used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$)



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

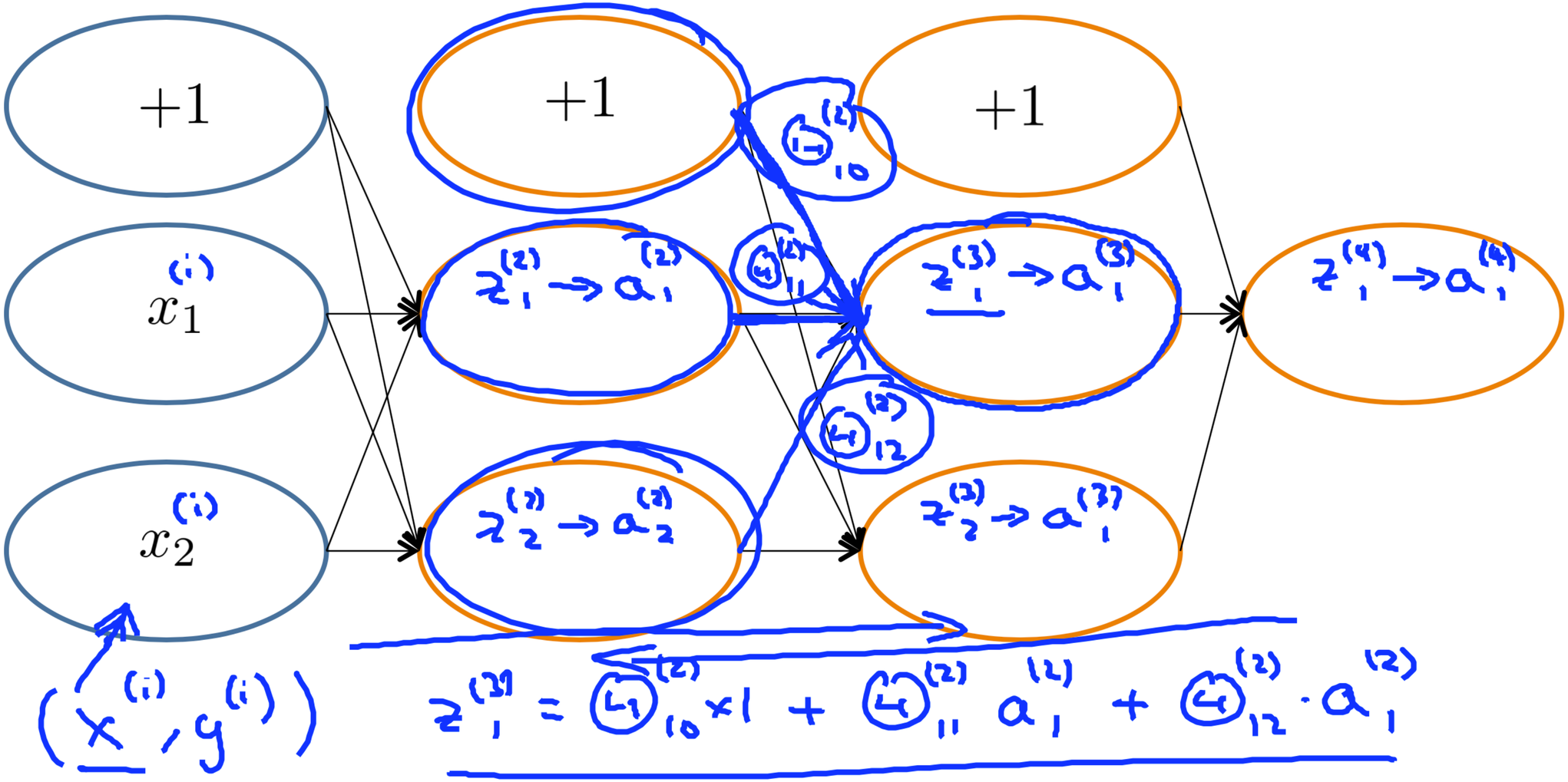
$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} \quad \text{if } j = 0$$



Cost Function and Backpropagation

Backpropagation Intuition

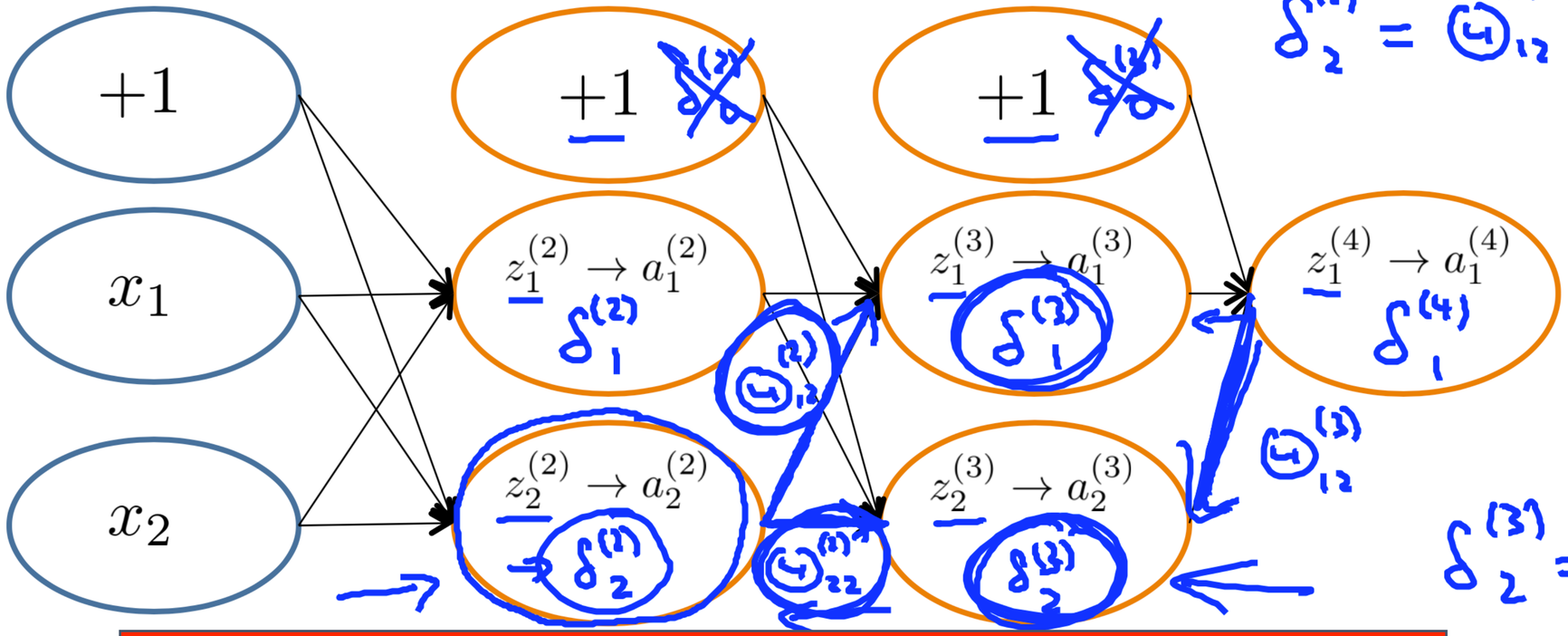
Forward Propagation



Cost Function and Backpropagation

Backpropagation Intuition

Forward Propagation



$$\delta_1^{(4)} = y^{(i)} - a_1^{(4)}$$

$$\delta_2^{(3)} = \Theta_{12}^{(3)} \delta_1^{(4)}$$

$$\delta_2^{(2)} = \Theta_{12}^{(2)} \delta_1^{(3)} + \Theta_{22}^{(2)} \delta_2^{(3)}$$

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

Neural Networks: Learning

Implementation Note: Unrolling Parameters

Advanced optimization

```
function [jVal, gradient] = costFunction(theta)
    ...
```

$\mathbb{R}^{n+1}$ (vectors)

```
optTheta = fminunc(@costFunction, initialTheta, options)
```

Neural Network (L=4):

→ $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (Theta1, Theta2, Theta3)

→ $D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (D1, D2, D3)

To be suitable for optimization algorithm

Neural Networks: Learning

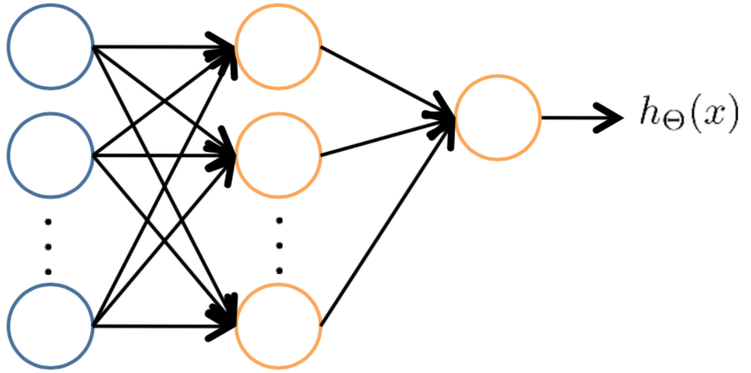
Implementation Note: Unrolling Parameters

Example

$$s_1 = 10, s_2 = 10, s_3 = 1$$

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

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




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

Neural Networks: Learning

Implementation Note: Unrolling Parameters

Learning Algorithm

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

```
fminunc(@costFunction, initialTheta, options)
```

Learning Algorit

- Have initial pa
- Unroll to get `i`

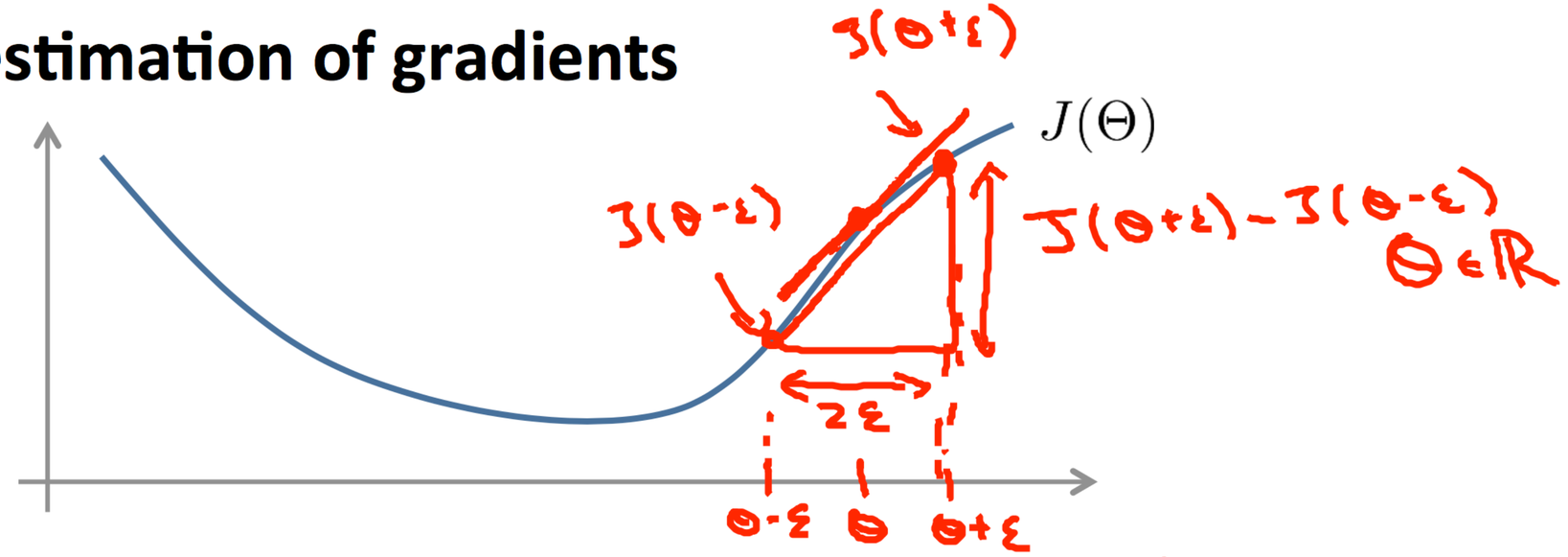
```
fminunc(@costF
```

Unroll ... to get `gradientVec`.

Neural Networks: Learning

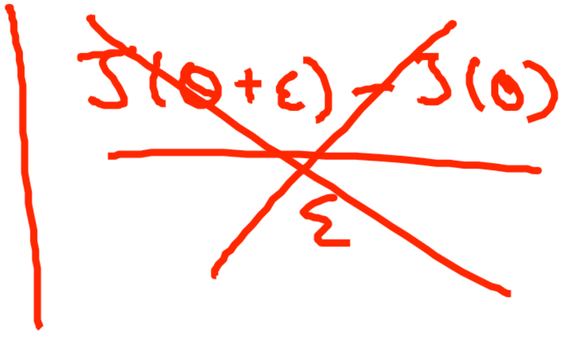
Gradient checking

Numerical estimation of gradients



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

$\varepsilon = 10^{-4}$



To verify forward and backward propagation are moving to the right direction!!!

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

Gradient checking

Backpropagation in Practice

Parameter vector θ

- $\theta \in \mathbb{R}^n$ (E.g. θ is "unrolled" version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$)
- $\theta = [\theta_1, \theta_2, \theta_3, \ldots, \theta_n]$
- $\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1+\epsilon, \theta_2, \theta_3, \ldots, \theta_n) - J(\theta_1-\epsilon, \theta_2, \theta_3, \ldots, \theta_n)}{2\epsilon}$
- $\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2+\epsilon, \theta_3, \ldots, \theta_n) - J(\theta_1, \theta_2-\epsilon, \theta_3, \ldots, \theta_n)}{2\epsilon}$
- $\vdots$
- $\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \ldots, \theta_n+\epsilon) - J(\theta_1, \theta_2, \theta_3, \ldots, \theta_n-\epsilon)}{2\epsilon}$

Backpropagation in Practice

Gradient checking

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

$\theta = [\theta_1, \theta_2, \ldots, \theta_i + \varepsilon, \ldots, \theta_n] \rightarrow \theta_i - \varepsilon$

$\frac{\partial}{\partial \theta_i} J(\theta)$.

Check that gradApprox ≈ DVec

Derivative we got From back prop.

Backpropagation in Practice

Gradient checking

Implementation Note:

- Implement backprop to compute `DVec` (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
- Implement numerical gradient check to compute `gradApprox`.
- Make sure they give similar values.
- Turn off gradient checking. Using backprop code for learning.

Backprop is more computationally efficient

Once we are sure that our backpropagation implementation is correct turn it off!!!

Important:

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

Neural Networks: Learning

Random initialization

Initial value of Θ

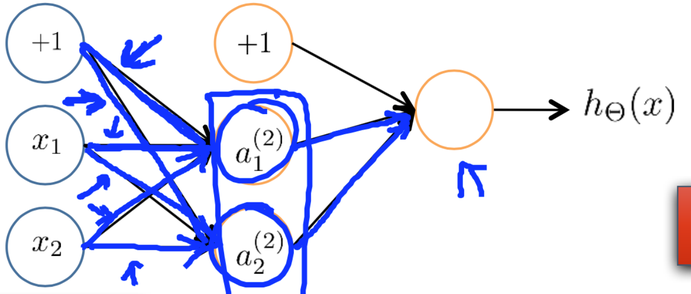
For gradient descent and advanced optimization method, need initial value for Θ.

```
optTheta = fminunc(@costFunction,
           initialTheta, options)
```

Consider gradient descent

Set `initialTheta = zeros(n,1)` ?

Zero initialization



$\rightarrow \Theta_{ij}^{(l)} = 0$ for all i, j, l.

Back prop

Forward prop

$a_1^{(2)} = a_2^{(2)}$. Also $\delta_1^{(2)} = \delta_2^{(2)}$.

$\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$

$\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$

After each update, parameters corresponding to inputs going into each of two hidden units are identical.

$a_1^{(2)} = a_2^{(2)}$

Neural Networks: Learning

Random initialization

Random initialization: Symmetry breaking

Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$

(i.e. $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)

*** not the same epsilon as gradient checking!!!!

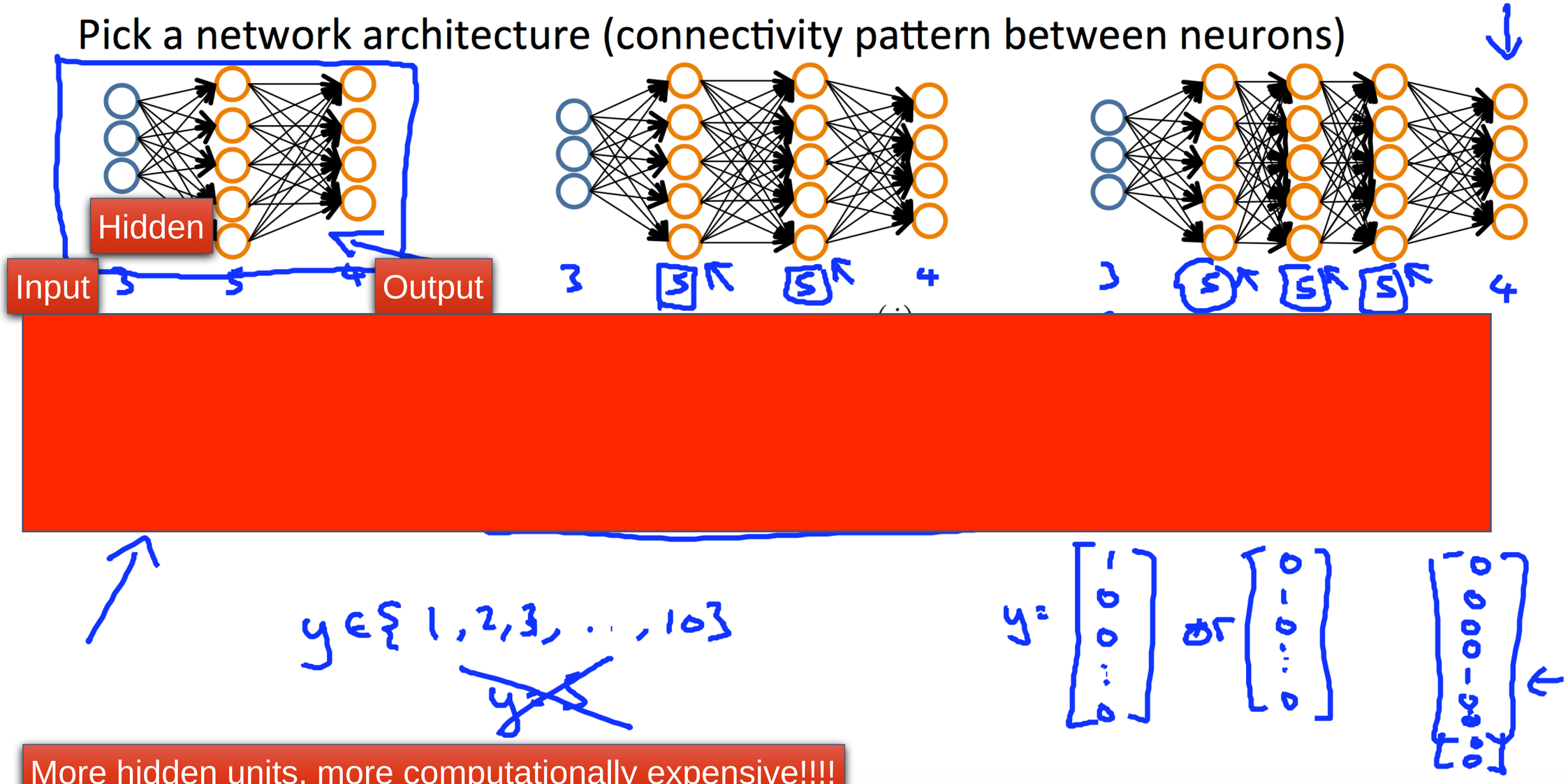
E.g.

Putting it together

Neural Networks: Learning

Training a neural network

Pick a network architecture (connectivity pattern between neurons)



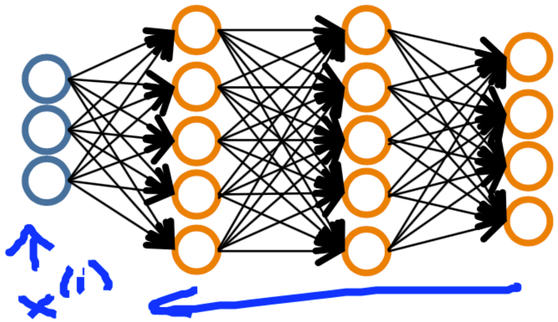
More hidden units, more computationally expensive!!!!

Putting it together

Neural Networks: Learning

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

Try to find efficient ways to do for loop over training examples!!



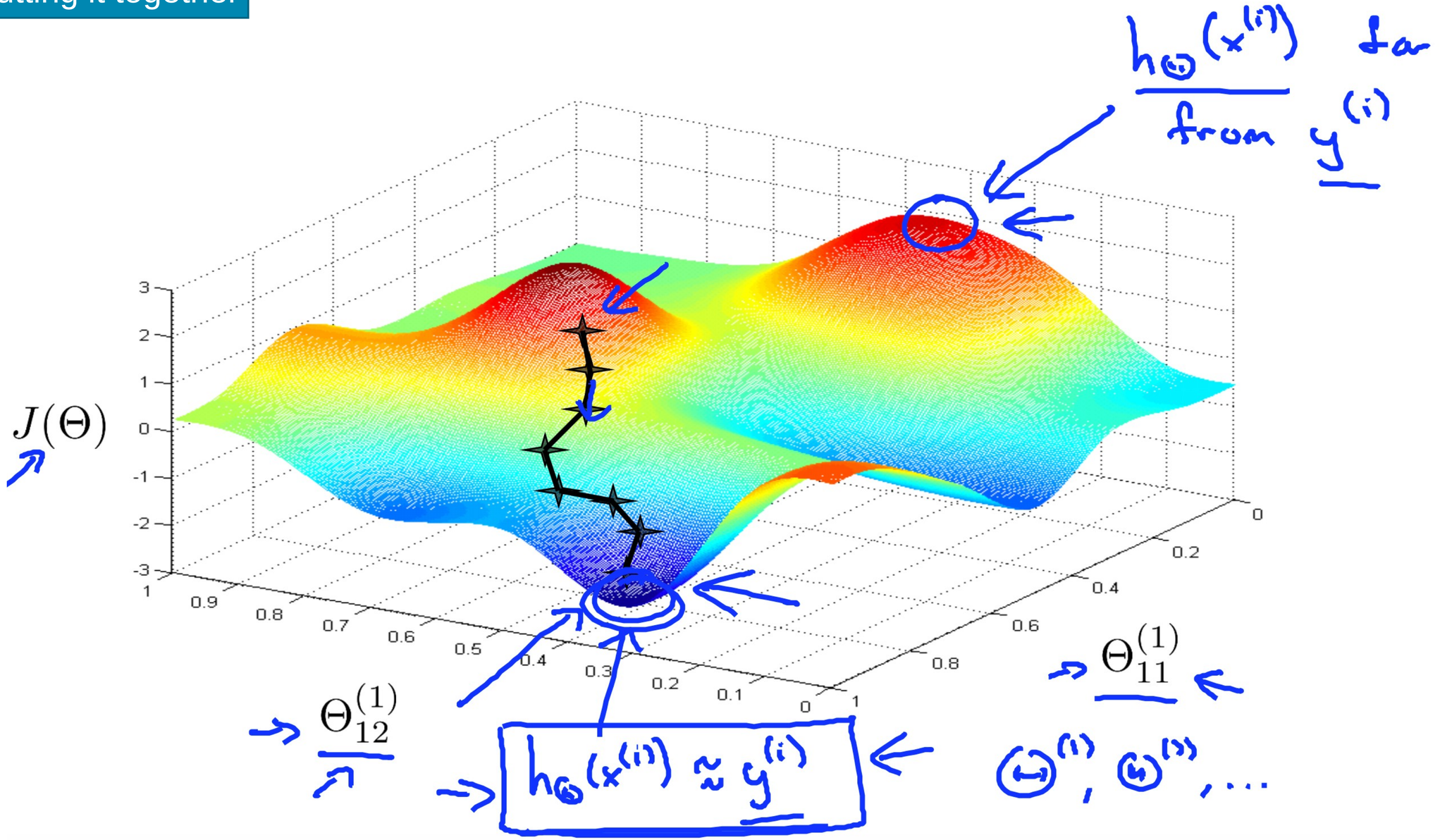
Putting it together

Neural Networks: Learning

$J(\Theta)$ — non-convex.

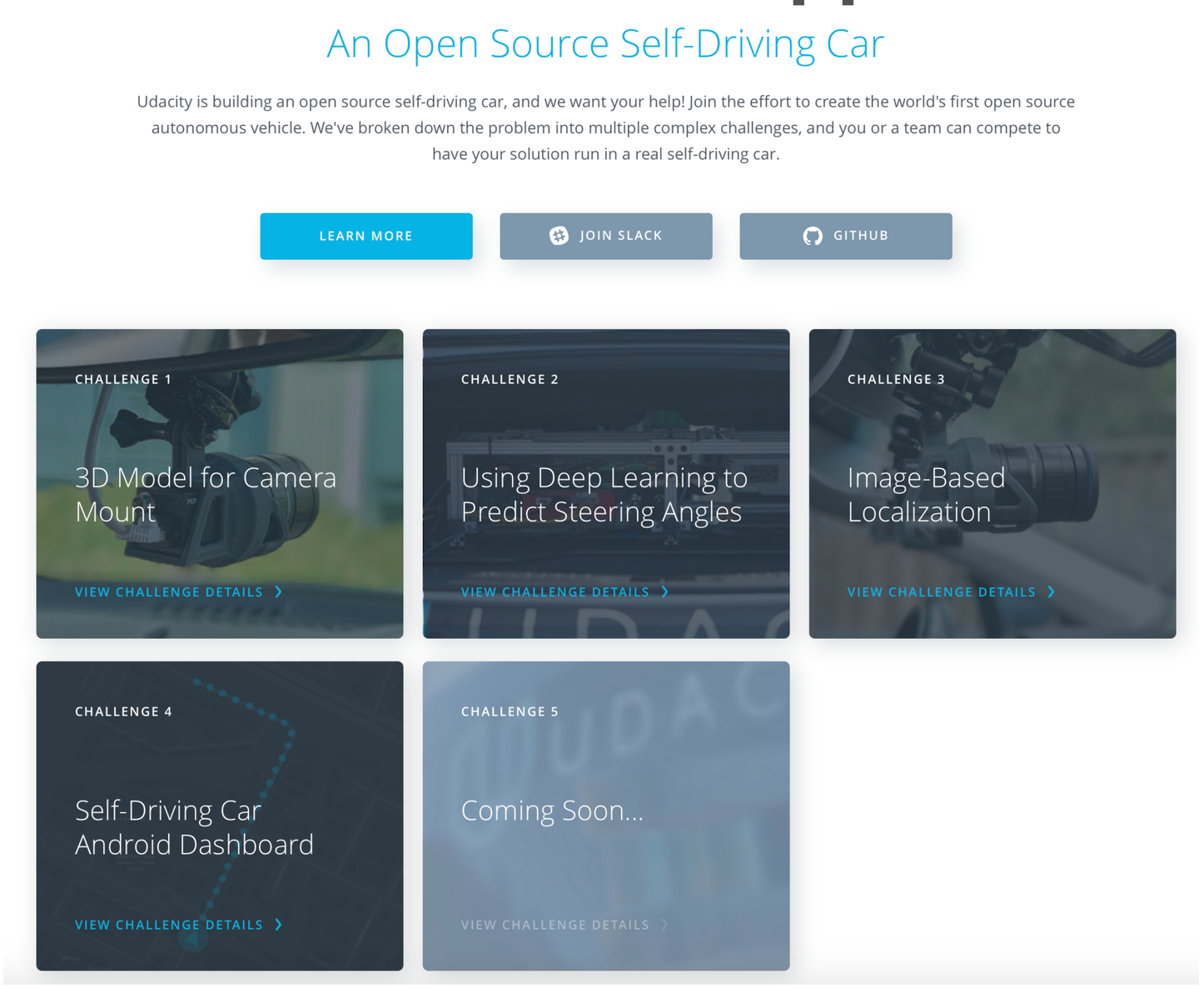
Neural Networks: Learning

Putting it together



Neural Networks Applications

Autonomous driving



https://github.com/udacity/self-driving-car https://www.udacity.com/self-driving-car

Perceptron

Perceptron

The Perceptron is a primitive type of neural network that learns weights for input attributes and transfers the weighted inputs into a network output or prediction.

This recipes shows the fitting of a Perceptron algorithm on the diabetes dataset.

```
# Perceptron
import numpy as np
from sklearn import datasets
from sklearn.linear_model import Perceptron
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a Perceptron model to the data
model = Perceptron()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Advice for Applying Machine Learning

how to tell when a learning algorithm is doing poorly?

describe the 'best practices' for how to 'debug' your learning algorithm and go about improving its performance.

We will also be covering machine learning system design.

you'll need to first understand where the biggest improvements can be made.

In these lessons, we discuss how to understand the performance of a machine learning system with multiple parts, and also how to deal with skewed data.

When you're applying machine learning to real problems, a solid grasp of this week's content will easily save you a large amount of work.

Deciding what to try next

Evaluating a Learning Algorithm

Debugging a learning algorithm:

Suppose you have implemented regularized linear regression to predict housing prices.

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