

Kevin Chen  
7/24/17

## Machine Learning Notes

Course: <https://www.coursera.org/learn/machine-learning>

### Welcome to Machine Learning

- Machine learning is the science of getting computers to learn, without being explicitly programmed.
- Neural networks mimic how the human brain works. These type of “learning algorithms” can help make truly intelligent machines.

### Introduction

#### Welcome

- Machine learning grew out of work in AI.
- Aimed to provide a new capability for computers
- Examples of uses of Machine Learning:
  - Database mining: processing large datasets
  - Applications that can't be programmed by hand
  - Self-customizing programs. (e.g. product recommendations)
  - Understand human learning

#### What is Machine Learning?

- Arthur Samuel's definition: field of study that gives computers the ability to learn without being explicitly programmed.
- Tom Mitchell's definition: A computer program learns from experience  $E$  with respect to some task  $T$  and some performance measure  $P$ , if its performance on  $T$ , as measured by  $P$ , improves with experience  $E$ .
- Machine learning algorithms:
  - Supervised learning: teach the computer how to do something.
  - Unsupervised learning: let the computer learn by itself.
  - Other algorithms: reinforcement learning and recommender systems

#### Supervised Learning

- Example: apply a regression line (linear, quadratic, etc) to a data set to interpolate a point.
- In supervised learning, you are provided the “right answers” for certain inputs.
  - They are categorized into “regression” or “classification” problems.
- Regression: predict a continuous valued output
- Classification problem: identifying which discrete category an example belongs to.
  - Or rather might find the chance that the example might belong to each category

- An example may consist of more than just one feature (e.g. age and tumor size for predicting whether the cancer is malignant)
- Sometimes you want an infinite number of features (i.e. attributes).
  - The Support Vector Machine can deal with truly infinite number of features

## Unsupervised Learning

- We're given data that doesn't have any labels. We are not told what to do with it, and we're not told what each data point is.
  - We are simply told that this is a data set: can you find some structure in the data?
- Clustering algorithms break data into different groups that you don't know in advance.
- Examples of applications for clustering algorithms: grouping news on the same topic, amount of gene expression, organizing large computer clusters, social network analysis, market segmentation, astronomical data analysis
- We are not giving the algorithm the right answer for the examples in the data set.
  - With unsupervised learning there is no feedback based on the prediction results.
- Cocktail party problem: we are given a set of people speaking and some microphones that (depending on where they are placed) records each person's voice at a certain level.
  - Clustering algorithms will separate each person's voice.
  - It doesn't have to be different people talking – it can be background music, etc.
- Depending on the programming environment, many learning algorithms can be really short programs.
- This class uses the Octave programming environment.
  - Really fast environment – great at prototyping.

## Model and Cost Function

### Model Representation

- In supervised learning, we have a data set that is the training set.
- Notation:
  - $m$  = Number of training examples
  - $x^{(i)}$  = “input” variables/features.  $X$  is the space of input values.
  - $y^{(i)}$  = “output” variables/“target” variable.  $Y$  is the space of output values.
  - $(x, y)$  is a single training example.
  - $(x^{(i)}, y^{(i)})$  is the  $i^{\text{th}}$  training example.
- Supervised learning: start with training set, feed into learning algorithm, which produces an output function  $h$  (stands for hypothesis). This function takes in an input  $x$  and predicts an output  $y$ .
- How do we represent  $h$ ?
  - $h_{\theta}(x) = \theta_0 + \theta_1 x$ . Shorthand:  $h(x)$ .
  - This is a linear regression with one variable, or a univariate linear regression.

### Cost Function

- In the hypothesis,  $h_{\theta}(x) = \theta_0 + \theta_1 x$ ,  $\theta_i$ 's are called parameters.
  - We are given a set of examples  $(x, y)$ 's, and we want to fit a line through them.
  - We will use the cost function to choose those  $\theta_i$ 's.

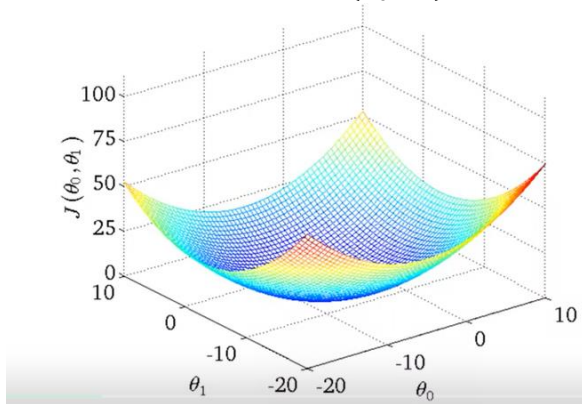
— Squared error cost function:

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

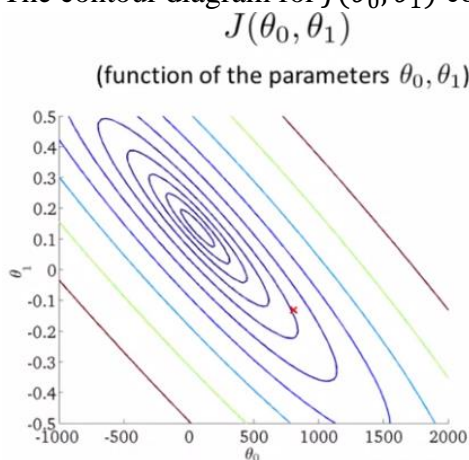
- We divide by  $m$  to find the “average” error
  - We divide by 2 so that the derivative is easier to take.
- Our goal is to minimize the cost function:

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

- For a fixed  $\theta_0, \theta_1$ , the hypothesis  $h_{\theta}(x)$  is a function of  $x$ , whereas the cost function  $J(\theta_0, \theta_1)$  is a function of the parameters  $\theta_0, \theta_1$ .
- The cost function  $J(\theta_0, \theta_1)$  forms an elliptic paraboloid:



- Fixing either  $\theta_0$  or  $\theta_1$  results in a quadratic equation.
- The contour diagram for  $J(\theta_0, \theta_1)$  consists of “ovals”:



- We now have the goal of finding an algorithm that finds the minimum of the  $J(\theta_0, \theta_1)$  function.

## Gradient Descent

- We have some function  $J(\theta_0, \dots, \theta_n)$  and we want  $\min_{\theta_0, \dots, \theta_n} J(\theta_0, \dots, \theta_n)$
- We start with some random  $\theta_0, \dots, \theta_n$ , and keep changing  $\theta_0, \dots, \theta_n$  to reduce  $J(\theta_0, \dots, \theta_n)$ .
- At each iteration, you take a step in the direction of greatest descent (or negative slope) until convergence (i.e. you reach a local minima)
- Outcome depends on where you initialize and your step size.
- Gradient descent algorithm:  
repeat until convergence {
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \dots, \theta_n)$$
}
- $\alpha$  is the learning rate. Controls the step size.
- The parameters  $\theta_0, \dots, \theta_n$  need to be updated simultaneously. Store each  $\theta_j$  is a temporary variable so that you don't use the new value of  $\theta_j$  when calculating  $\theta_{j+1}, \dots, \theta_n$ . Only until all temporary values are computed, do you set  $\theta_j$  to the temporary value.
- If the learning rate  $\alpha$  is too small, gradient descent can be slow. If  $\alpha$  is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.
- Gradient descent can converge to a local minimum, even with the learning rate  $\alpha$  fixed.
  - As we approach a local minimum, gradient descent will automatically take smaller steps, so there is no need to decrease  $\alpha$  over time.

## Gradient Descent for Linear Regression

- $\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{\partial}{\partial \theta_j} \left( \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 \right) = \frac{\partial}{\partial \theta_j} \left( \frac{1}{2m} \sum_{i=1}^m (\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2 \right)$ 
  - $j = 0$ :  $\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})$
  - $j = 1$ :  $\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) * x_1^{(i)}$
- Gradient descent algorithm:  
repeat until convergence {
$$\begin{aligned}\theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \\ \theta_1 &:= \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) * x_1^{(i)}\end{aligned}$$
}
- Recall that we need to update  $\theta_0$  and  $\theta_1$  simultaneously.
- The cost function for linear regression is a convex function (bowl shaped function) so there is only one local optimum, which is the global optimum.
  - Hence, this gradient descent algorithm will converge to the global optimum.
- This type of gradient descent is referred to as “batch” gradient descent, which means that each step of gradient descent uses all the training examples.

- This is true because we use  $\sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})$ , which sums across all  $m$  training examples.

## Linear Algebra Review

— **Note:** this section is condensed as it is mostly review. Only important points here.

—  $A = \begin{bmatrix} 2 & 4 \\ -5 & 3 \\ 1 & 6 \end{bmatrix}$  is a  $3 \times 2$  matrix since it has 3 rows and 2 columns.  $\mathbb{R}^{3 \times 2}$

- $A_{12} = 4$  since the element in the first row and second column is 4.

— Vectors are an  $n \times 1$  matrix.

- $y_i$  is the  $i$ th element.

— We generally use 1-indexed, not 0-indexed vectors, i.e.  $y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}$ .

— By convention, we use upper case to refer to matrices, or lower-case to refer to small numbers or vectors.

— When multiplying two matrices of size  $m \times n$  and  $n \times o$ , the two  $n$ 's must be the same value and the result is an  $m \times o$  matrix.

— You can formulate a set of predictions using matrix-vector multiplication:

- Prediction = Data Matrix \* parameters.
- Example: what predictions would be generated from the hypothesis  $h_{\theta}(x) = -40 + 0.25x$  for house sizes of  $x = 2104, 1416, 1532, 825$ .

$$\begin{bmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1532 \\ 1 & 825 \end{bmatrix} x \begin{bmatrix} -40 \\ 0.25 \end{bmatrix} = \begin{bmatrix} 486 \\ 314 \\ 344 \\ 173 \end{bmatrix} \leftarrow \text{predictions}$$

- Formulating the problem as matrix-vector multiplication can be computationally more efficient for large set of predictions.

— You can calculate a set of predictions for a set of hypotheses using matrix-matrix multiplication.

- Prediction = Data Matrix \* Sets of Parameters
- Example: what predictions would be generated from the hypotheses:

$$1. \quad h_{\theta}(x) = -40 + 0.25x$$

$$2. \quad h_{\theta}(x) = 200 + 0.1x$$

$$3. \quad h_{\theta}(x) = -150 + 0.4x$$

for house sizes of  $x = 2104, 1416, 1532, 825$ .

$$\begin{bmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1532 \\ 1 & 825 \end{bmatrix} x \begin{bmatrix} -40 & 200 & -150 \\ 0.25 & 0.1 & 0.4 \end{bmatrix} = \begin{bmatrix} 486 & 410 & 692 \\ 314 & 342 & 416 \\ 344 & 353 & 464 \\ 173 & 285 & 191 \end{bmatrix}$$

- $i$ th column is the set of predictions for the  $i$ th hypothesis.
- This is computationally faster with parallelism.

- Matrix multiplication is not commutative but is associative.
- Identity matrix: for any matrix  $A$ ,  $A \cdot I = I \cdot A = A$
- Matrix inverse: only square matrices may have an inverse. If  $A$  is an  $m \times m$  matrix and if it has an inverse, then  $A(A^{-1}) = A^{-1}A = I$ .
  - Matrices that don't have an inverse are "singular" or "degenerate". Example:
 
$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$
- Matrix transpose: let  $A$  be an  $m \times n$  matrix and let  $B = A^T$ , then  $B$  is an  $n \times m$  matrix and  $B_{ij} = A_{ji}$ .

## Linear Regression with Multiple Variables

### Intro

- We want to learn a model that predicts something based on multiple variables or features.
- Notation:
  - $n$  = number of features
  - $x^{(i)}$  = input (features) of the  $i$ th training example
  - $x_j^{(i)}$  = value of feature  $j$  in the  $i$ th training example
- Hypothesis:  $h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n$ 
  - For convenience of notation, define  $x_0 = 1$ , i.e.  $\forall i, x_0^{(i)} = 1$ .
  - So  $x = \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1}$  and  $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \dots \\ \theta_n \end{bmatrix}$
  - $h_{\theta}(x) = \sum_{i=0}^n \theta_i x_i = \theta^T x$
- Linear regression with multiple variables is also known as multivariate linear regression.

### Gradient Descent for Multiple Variables

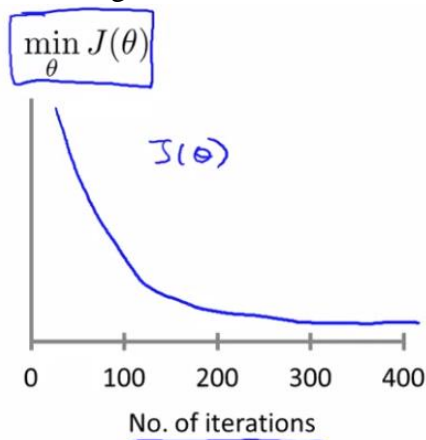
- The algorithm for  $n$  variables is:
 

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

### Gradient Descent for Multiple Variables in Practice

- Feature scaling: make sure features are on a similar scale. This will allow gradient descent to converge faster.
  - If  $x_1$  takes in values between 0 – 2000 (size in feet) and  $x_2$  takes values between 1 – 5 (number of bedrooms), then gradient descent will take a while to meander to a local minimum.

- To fix this, scale the features.  $x_1 = \frac{\text{size}(\text{feet}^2)}{2000}$ ,  $x_2 = \frac{\text{number of bedrooms}}{5}$ .
  - More generally, get each feature into approximately a  $-1 \leq x_i \leq 1$  range.  
General rule of thumb, if range is -3 to 3 or  $-\frac{1}{3}$  to  $\frac{1}{3}$ , then it's ok. Any range larger than -3 to 3 or smaller than  $-\frac{1}{3}$  to  $\frac{1}{3}$  should be scaled.
- Mean normalization: replace  $x_i$  with  $x_i - \mu_i$  to make features have approximately zero mean. (Does not apply to  $x_0 = 1$ ).
- Ex:  $x_1 = \frac{\text{size} - 1000}{2000}$ ,  $x_2 = \frac{\text{\#bedrooms} - 2}{5}$
  - More generally,  $x_i := \frac{x_i - \mu_i}{s_i}$ , where  $\mu_i$  is the average value of  $x_i$  in training set and  $s_i$  is either the range (max - min) or the standard deviation.
- If gradient descent is working correctly, you should ideally get a plot that looks something like this:



- $J(\theta)$  should decrease after every iteration for sufficiently small  $\alpha$ . But if  $\alpha$  is too small, then gradient descent can be slow to converge.
- It's difficult to tell how many iterations it takes for gradient descent to converge.
- Automatic convergence test: declare convergence if  $J(\theta)$  decreases by less than  $\epsilon$
- Gradient descent is not working if  $J(\theta)$  is increasing exponentially, or if it's oscillating up and down. Learning rate is too large, so pick smaller  $\alpha$ .
- Choosing your features: sometimes you may want to create new features that are combinations (or functions) of your existing features.
- For example, if you want to predict housing prices given housing widths and lengths, you might want to create a new feature  $\text{area} = \text{width} * \text{length}$ .
  - Polynomial regression is a special case of creating new features. For example, if  $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$ , then you're essentially defining new features  $x_2 = x_1^2$  and  $x_3 = x_1^3$ . Note that scaling features is especially important here since  $x_1^2$  and  $x_1^3$  can have very different ranges compared to  $x_1$ .

## Normal Equations

- Normal equations: method for analytically solving  $\theta$  to minimize  $J(\theta)$ .

- Recall for 1D cases (i.e.  $\theta \in \mathbb{R}$ ): take the derivative  $\frac{d}{d\theta}J(\theta)$  and set it to zero, and solve for  $\theta$ . But how to deal when  $\theta \in \mathbb{R}^{n+1}$ ?
- If  $\theta \in \mathbb{R}^{n+1}$ , then set  $\frac{\partial}{\partial \theta_j}J(\theta) = 0$  for each  $j = 0, \dots, n$ , giving you  $n + 1$  equations.

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

- Solving this with linear algebra, given  $m$  training examples  $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$  and  $n$  features:

$$\text{Let } X = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & \dots & x_n^{(1)} \\ x_0^{(2)} & x_1^{(2)} & \dots & x_n^{(2)} \\ \dots & \dots & \dots & \dots \\ x_0^{(m)} & x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \text{ and } y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \dots \\ y^{(m)} \end{bmatrix}.$$

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

- Terminology:  $X$  above is known as the design matrix.
- Feature scaling isn't necessary when using the normal equations method.
  - Advantages of normal equations over gradient descent:
    - No need to choose  $\alpha$
    - Doesn't need to iterate. (Gradient descent needs many iterations.)
  - Advantages of gradient descent over normal equations:
    - Works well for large  $n$ . In normal equations,  $(X^T X)^{-1}$  is slow:  $O(n^3)$  for taking the inverse. Gradient descent is  $O(kn^2)$
    - Normal equations don't work for some more sophisticated learning algorithms (we'll have to resort to gradient descent for those algorithms.)
  - When computing  $\theta = (X^T X)^{-1} X^T y$ , what if  $X^T X$  is non-invertible? (aka singular, degenerate). Usually two cases:
    - Redundant features (linearly dependent – 2 features are related via a linear equation). Ex:  $x_1 = \text{size in feet}^2$  and  $x_2 = \text{size in } m^2$
    - Too many features (i.e.  $m < n$ ). Delete some features, or use regularization.
  - The pseudoinverse function should still solve for  $\theta$  even if  $X^T X$  is non-invertible.

## Logistic Regression

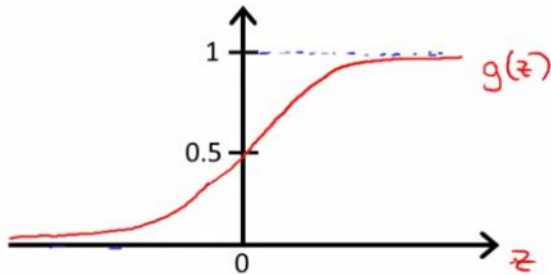
### Classification and Representation

- Examples of classification problems: email (spam / not spam), online transactions: fraudulent (yes / no), and tumors (malignant or benign).
  - $y \in \{0,1\}$ , 0: “Negative class” (e.g. benign tumor), 1: “Positive class” (e.g. malignant tumor). This is binary classification.
  - In multiclass classification,  $y$  can take on 3 or more values.
- For binary classification, one method is to still use linear regression ( $h_\theta(x) = \theta^T x$ ), but threshold the classifier output  $h_\theta(x)$  at a value, say 0.5
  - In other words, if  $h_\theta(x) \geq 0.5$ , predict “ $y = 1$ ”. Else if  $h_\theta(x) < 0.5$ , predict “ $y = 0$ ”.



- However, adding extreme examples (inputs with very large or small values compared to other inputs) can skew the entire linear regression hypothesis, making it much more likely to misclassify certain inputs.
  - Hence, linear regression for classification is often not a good idea.
  - Furthermore, linear regression can output values  $>1$  and  $<0$ , despite  $y \in \{0,1\}$
- Logistic regression is more suitable for classification, as  $0 \leq h_\theta(x) \leq 1$ .
- Logistic regression model:

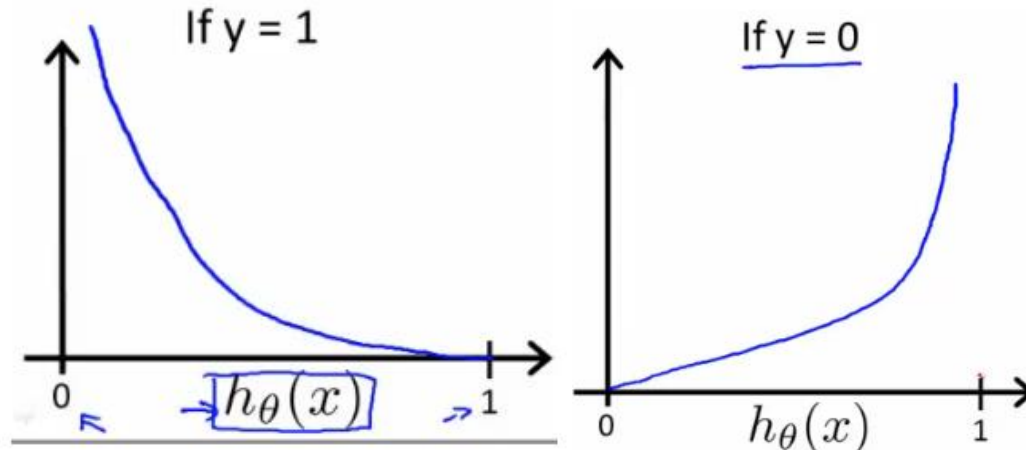
- The sigmoid function (or logistic function) is defined as:  $g(z) = \frac{1}{1+e^{-z}}$



- Asymptotes at  $g(z) = 1$  as  $z \rightarrow \infty$  and  $g(z) = 0$  as  $z \rightarrow -\infty$
  - $g(z) = 0.5$  when  $z = 0$
  - Our hypothesis is now:  $h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$ .
- Interpretation of hypothesis output
- $h_\theta(x)$  = estimated probability that  $y = 1$  on input  $x$ .
  - In other words,  $h_\theta(x) = P(y = 1|x; \theta)$ , or the “probability that  $y = 1$ , given  $x$ , parameterized by  $\theta$ .”
  - Because  $y \in \{0,1\}$ ,  $P(y = 0|x; \theta) + P(y = 1|x; \theta) = 1$  and  $P(y = 0|x; \theta) = 1 - P(y = 1|x; \theta)$
- Decision boundary: one strategy is to predict  $y = 1$  if  $h_\theta(x) \geq 0.5$  (i.e.  $\theta^T x = z \geq 0$ ) and predict  $y = 0$  if  $h_\theta(x) < 0.5$  (i.e.  $\theta^T x = z < 0$ ). In this case, the decision boundary is the function  $\theta^T x = z = 0$ .
- The decision boundary is the line that separates hypotheses that predict  $y = 1$  and  $y = 0$ .
  - The decision boundary is nonlinear if the  $x$  vector contains non-linear terms (e.g.  $x_1^2, x_1^2 x_2$ , etc.)
  - The decision boundary is defined by  $\theta$ , not the training set.
- Cost function: fitting parameters
- Formally defined, given a training set  $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$  with  $m$  examples,  $x \in \mathbb{R}^{n+1}$  with  $n$  features,  $x_0 = 1, y \in \{0,1\}$ ,  $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$ , how to choose parameters  $\theta$ ?
- If we define  $Cost(h_\theta(x), y) = \frac{1}{2}(h_\theta(x) - y)^2$ :
- Then,  $J(\theta) = \frac{1}{m} \sum_{i=1}^m \frac{1}{2} (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{m} \sum_{i=1}^m Cost(h_\theta(x^{(i)}, y^{(i)}))$

- Unfortunately, this approach cause  $J(\theta)$  to be a non-convex function with respect to  $\theta$ , so we may get stuck at a local minimum.

— To fix the non-convex issue, we define  $Cost(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$ .



This is desirable because if:

- Case 1 --  $y = 1$ :  $Cost = 0$  if  $h_\theta(x) = 1, z \rightarrow \infty$ . But as  $h_\theta(x) \rightarrow 0, Cost \rightarrow \infty$ . This captures the intuition that if  $h_\theta(x) = 0$  (i.e. predicting  $P(y = 1|x; \theta) = 0$ ), but if  $y = 1$ , we'll penalize the learning algorithm by a very large cost.
- Case 2 --  $y = 0$ :  $Cost = 0$  if  $h_\theta(x) = 0, z \rightarrow -\infty$ . But as  $h_\theta(x) \rightarrow 1, Cost \rightarrow \infty$ .
- This also gives a convex optimization problem.

— This is a more compact way of expressing the above  $Cost(h_\theta(x), y)$  function:

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

- You can verify that this compact equation produces the above equation under the two possible cases  $y = 0$  or  $y = 1$ .

— Logistic regression cost function can be expressed as:

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

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

- This function is used because it can be derived from statistics using the principle of maximum likelihood estimation, and since the function is convex.
- To fit parameters  $\theta$ , simply find  $\arg \min_{\theta} J(\theta)$ . (see instructions below)
- To make a prediction given new  $x$ : output  $h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$
- A vectorized implementation is:

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

— To minimize  $J(\theta)$ :

Repeat until convergence {

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

Simultaneously update all  $\theta_j$

}

- Note that  $\frac{\partial}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$
- The algorithm above looks identical to linear regression update rule. The only difference is that  $h_{\theta}(x) = \frac{1}{1+e^{-\theta^T x}}$  for logistic regression and  $h_{\theta}(x) = \theta^T x$  for linear regression.
- Vectorized implementation:  
Repeat until convergence {  

$$\theta := \theta - \alpha \frac{1}{m} X^T (g(X\theta) - y)$$

}

- Feature scaling also improves speed of logistic regression.

— Aside from gradient descent, there are other optimization algorithms that are more advanced: fminunc, conjugate gradient, BFGS, and L-BFGS.

- The advantages of the other optimization algorithms are that there is no need to manually pick  $\alpha$ . It is often faster than gradient descent.
- The main disadvantage of the other optimization algorithms is that they're more complex.
- To use existing libraries that implement the above optimization algorithms in Octave, you need to pass in the error function  $J(\theta)$  and all partial derivatives  $\frac{\partial}{\partial \theta_j} J(\theta) \forall j$ .

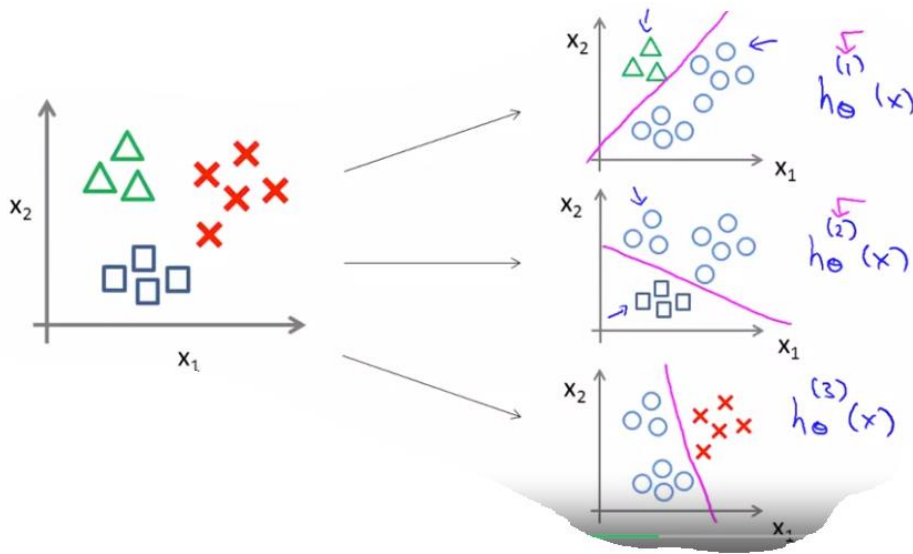
## Multiclass Classification

—  $y$  can now take on a set of discrete natural numbers.

- Example (weather):  $y = 1$  for sunny,  $y = 2$  for cloudy,  $y = 3$  for rain,  $y = 4$  for snow

— One-vs-all (also known as one-vs-rest) multiclass classification:

- Assuming there are  $K$  classes, we train  $K$  binary classifiers.
- For  $i = 1, \dots, K$ , we train a classifier such that  $h_{\theta}^{(i)}(x) = P(y = i|x; \theta)$
- We do this by the following method: for a given  $i$ , set the positive examples to be when  $y = i$  and the negative examples to be  $y \neq i$ .



- To make a prediction, on a new input  $x$ , pick the class  $i$  that maximizes  $\max_i h_{\theta}^{(i)}(x)$

### Solving the Problem of Overfitting

#### — The problem

- Underfitting: if we're applying regression with too simple of a function, this results in "underfitting", or having a "high bias".
- Overfitting: we apply regression with too complex of a function, as it goes up and down too much, resulting in us having a "high variance".
- If we have too many features (i.e. overfitting), 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 no new examples).

#### — Underfitting/overfitting can occur for both linear regression and logistic regression by including too many $x_i$ terms (and combinations of them).

#### — Two main options to address overfitting:

- Reduce number of features: either manually select which features to keep or use a model selection algorithm (see later in the course). Disadvantage of this approach is that we throw away some information we have of the problem.
- Regularization: keep all the features, but reduce magnitude/values of parameters  $\theta_j$ . This works well when we have a lot of features, each of which contributes a bit to predicting  $y$ .

#### — Basics of regularization: imagine we add the magnitude of $\theta_i$ into the cost function (for some specific constant $\lambda$ , called the regularization parameter):

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

- This will encourage the algorithm to make  $\theta_i$  be small.

- Regularization thus leads to a “simpler” hypothesis, which is less prone to overfitting.
- By convention, we don’t penalize  $\theta_0$ .
- $\lambda$  controls the balance between the two objects of fitting the training set well and keeping our hypothesis “simple”.
- If  $\lambda$  is extremely large, then the function doesn’t even fit the training set well. (We get a hypothesis of approximately  $h_\theta(x) = \theta_0$ , resulting in high bias.

— Regularized linear regression for gradient descent algorithm:

Repeat {

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

For  $j = 1, \dots, n$ :

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

}

- The last statement can be alternatively rewritten as:

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

- We note that the expression  $1 - \alpha \frac{\lambda}{m}$  is less than 1 since  $\alpha, \lambda, m > 0$ . Hence, this update shrinks  $\theta_j$  by a factor of  $1 - \alpha \frac{\lambda}{m}$  in addition to applying the normal gradient descent rule in the right expression (i.e.  $-\alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$ ).

— Regularized gradient descent for normal equations:

- Recall  $X = \begin{bmatrix} (x^{(1)})^T \\ \dots \\ (x^{(m)})^T \end{bmatrix}$  and  $y = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(m)} \end{bmatrix}$ , where  $X$  is a  $m \times (n + 1)$  matrix and  $y \in \mathbb{R}^m$ . We minimize  $J(\theta)$  by setting  $\theta = (X^T X)^{-1} X^T y$  without regularization.

- With regularization, we now get  $\theta = \left( X^T X + \lambda \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{bmatrix} \right)^{-1} X^T y$ . (The

matrix  $L = \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{bmatrix}$  is basically the identity matrix except the top-left

element is 0 instead of 1.  $L$  is size  $(n + 1) \times (n + 1)$

- Non-invertibility: Recall that if  $m < n$  (number of examples < number of features),  $X^T X$  is non-invertible/singular/degenerate. However, regularization solves this problem, i.e.  $(X^T X + \lambda \cdot L)$  is invertible.
- Regularized logistic regression for gradient descent:
- To regularize logistic regression, we add  $\frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$  to cost function  $J(\theta)$ :
 
$$J(\theta) = \left[ -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$
  - This is the algorithm incorporating the regularized cost function:  
Repeat {
 
$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_0^{(i)}$$
 For  $j = 1, \dots, n$ :
 
$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j$$
 }
  - The above algorithm looks the same as regularized linear regression, but it is not since  $h_{\theta}(x) = \frac{1}{1+e^{-\theta^T x}}$  here.
- Regularized logistic regression for more advanced methods (e.g. fminunc, conjugate gradient, BFGS, and L-BFGS):
- Specify  $J(\theta) = \left[ -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$
  - Specify each gradient:
 
$$\text{gradient}(1) = \frac{\partial}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_0^{(i)}$$
 For  $j = 1, \dots, n$ ,  $\frac{\partial}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j$  as gradient(2), ..., gradient(n+1).

## Neural Networks

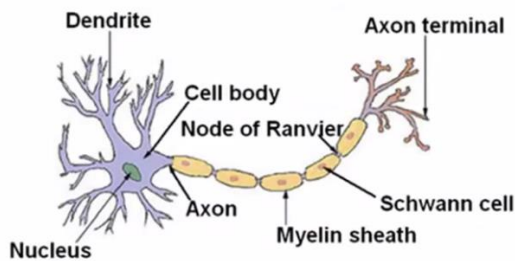
### Motivations

- Neural networks are good for non-linear (complex) hypotheses even when the input feature space is large (i.e. input is a vector of many parameters).
- Linear regression is only good for regression.
  - Logistic regression requires exponentially (relative to number of parameters) many terms to achieve a complex relationship, which may lead to overfitting. This blows up for problems such as computer vision, where inputs are individual pixels. Reducing the number of features reduces expressibility.

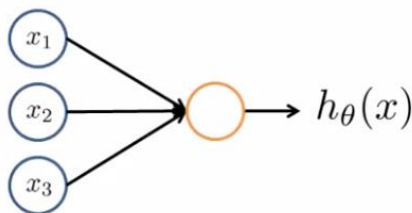
- Neural networks need hidden layer(s) to express non-linear hypotheses.
- Problem formulation: we get a set of positive and negative examples in the training set, and then get a test set.
- Origins: algorithms that try to mimic the brain.
- Widely used in 80s and early 90s, but popularity diminished in the late 90s.
  - Recent resurgence: state-of-the-art technique for many applications. Neural networks are computationally expensive, so new computers made this more feasible.
  - There is a hypothesis that the brain uses just a single learning algorithm. The auditory cortex is used for processing sounds. But when scientists cut off the ear sensor to the auditory cortex in an animal and reroute the eye sensor to the auditory cortex, the auditory cortex learned to see. Sight, sound, and touch can be all learned by the same tissue. This is called neuro-rewiring experiments.
  - Examples of seeing with other sensors: seeing with your tongue, human echolocation (sonar), haptic belt: direction sense, implanting a 3<sup>rd</sup> eye in a frog

### Model Representation

- Neural networks simulate networks of neurons in the brain.
- A single neuron:



- Dendrites are “input wires”. Axon is “output wire”. The cell body is the nucleus.
- Neurons communicate with each other with little pulses of electricity known as spikes.
- Neuron in an artificial neural network:



- Inputs:  $x_1, \dots, x_n$ . Output:  $h_{\theta}(x)$ . Weights (parameters):  $\theta_0, \dots, \theta_n$
  - $h_{\theta}(x) = \frac{1}{1+e^{-\theta^T x}}$ , where  $x = \begin{bmatrix} x_0 \\ \dots \\ x_n \end{bmatrix}$  and  $\theta = \begin{bmatrix} \theta_0 \\ \dots \\ \theta_n \end{bmatrix}$
  - $x_0$  is the “bias unit”.  $x_0 = 1$ , always.
  - Neurons use the sigmoid (logistic) activation function
- First layer is the input layer. Second layer is the first hidden layer. Final layer is called the output layer.

— Notation

- $a_i^{(j)}$  = “activation” (output) of neuron unit  $i$  in layer  $j$ .
- $\Theta^{(j)}$  = matrix of weights controlling function mapping from layer  $j$  to layer  $j + 1$ .
- $\Theta_{mn}^{(j)}$  = weight mapping from the  $n$ th node in layer  $j$  to the  $m$ th node in layer  $j + 1$
- $g$  = activation function

Forward propagation (Vectorized)

— Problem: given a set of inputs  $x$  and weights  $\Theta$ , calculate the output  $h_\theta(x)$

— Example:  $a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \dots + \Theta_{1n}^{(1)}x_n)$  and  $a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \dots + \Theta_{2n}^{(1)}x_n)$

— Generally,  $a_k^{(j)} = g(\Theta_{k0}^{(j-1)}a_0^{(j-1)} + \dots + \Theta_{kn}^{(j-1)}a_n^{(j-1)})$

— We note that  $x_k = a_k^1$

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

— Let  $z_k^{(j)}$  be the sum of weights \* input for the  $k$ th node in layer  $j$

$$z_k^{(j)} = \Theta_{k0}^{(j-1)}a_0^{(j-1)} + \dots + \Theta_{kn}^{(j-1)}a_n^{(j-1)}$$

— We see that  $a_k^{(j)} = g(z_k^{(j)})$

— Let  $x = \begin{bmatrix} x_0 \\ \dots \\ x_n \end{bmatrix}$  and  $z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ \dots \\ z_n^{(j)} \end{bmatrix}$  and  $a^{(j)} = \begin{bmatrix} a_0^{(j)} \\ \dots \\ a_n^{(j)} \end{bmatrix}$  We then get:

$$z^{(j)} = \Theta^{(j-1)}a^{(j-1)} \text{ and } a^{(j)} = g(z^{(j)}), \text{ where } a^1 = x$$

- Recall that we add the bias unit to  $a^{(j)}$  vectors. So for all  $j \geq 1$ ,  $a_0^{(j)} = 1$

— Formalizing the algorithm (assuming there are  $L$  layers):

For each layer  $j$  from 2 to  $L$ :

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

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

Return  $h_\theta(x) = a^{(L)} = g(z^{(L)})$

— This computation is similar to logistic regression, except that the neural network also gets to learn its own features in the hidden layers.

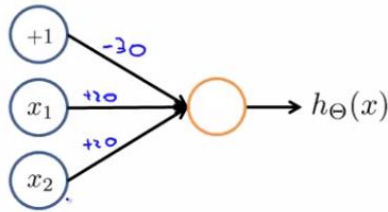
- If we didn't have hidden layers, then this would just be logistic regression.

— The term architecture refers to how the different neurons are connected to each other.

Neural Network Expressibility

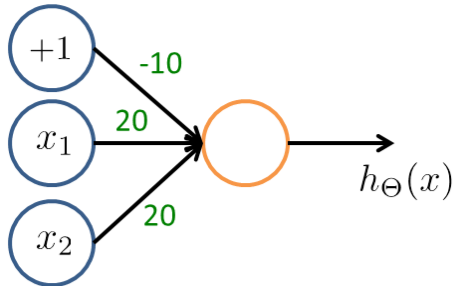
— Create a neural network that models AND given inputs  $x_1$  and  $x_2$



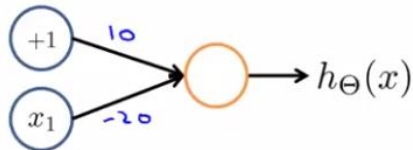


- Give weight  $\theta_{10}^{(1)} = -30$  to the bias,  $\theta_{11}^{(1)} = 20$  to  $x_1$  and  $\theta_{21}^{(1)} = 20$  to  $x_2$
- The output  $h_{\theta}(x) = g(-30 + 20x_1 + 20x_2)$ .  $g(4.6) \approx 1$  and  $g(-4.6) \approx 0$

— Create a neural network that models OR given inputs  $x_1$  and  $x_2$

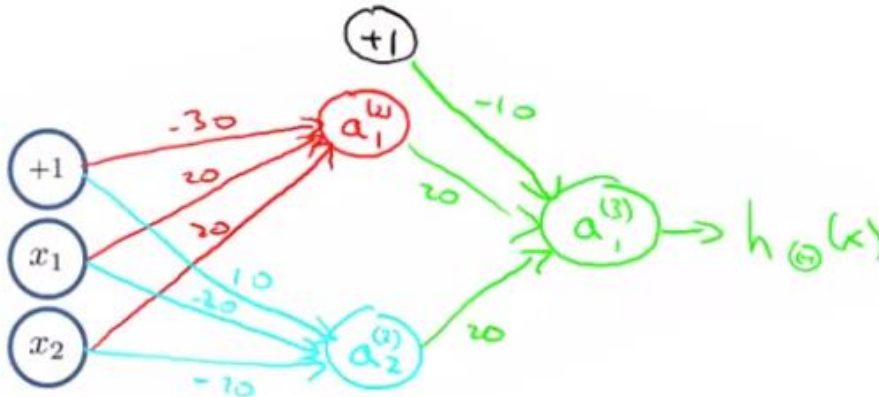


— Create a neural network that models NOT given input  $x_1$ .



— Create a neural network that models XNOR given inputs  $x_1$  and  $x_2$

- Recall  $x_1$  XNOR  $x_2$  is equal to NOT( $x_1$  XOR  $x_2$ ).



- $a_1^{(2)}$  computes  $x_1$  AND  $x_2$
- $a_2^{(2)}$  computes (NOT  $x_1$ ) AND (NOT  $x_2$ )
- $a_1^{(3)}$  computes  $a_1^{(2)}$  OR  $a_2^{(2)}$ .
- $\theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \\ 10 & -20 & -20 \end{bmatrix}$
- $\theta^{(2)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix}$

— Yann LeCun used neural networks to classify handwritten numeric digits.

- Multiclass classification: create an output layer that has multiple nodes, one node for each possible output class.

- For example, you want  $h_{\theta}(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$  if image is a pedestrian,  $h_{\theta}(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$  for

car,  $h_{\theta}(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$  for motorcycle, and  $h_{\theta}(x) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$  for truck

- $y \in \mathbb{R}^K$ , where  $K$  is the number of classes.
- Training set consists of  $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$ , where  $y^{(i)}$  is a vector.

### Cost Function

- Define  $L$  to be the total number of layers and  $s_l$  to be the number of units (not counting bias unit) in layer  $l$ . Let  $K$  be the number of units in the output layer.

- Binary classification:  $y = 0$  or  $1$ . 1 output unit.  $h_{\theta}(x) \in \mathbb{R}$ , The number of units in the last layer,  $s_L = 1$ . Let  $K$  be the number of output units.  $K = s_L = 1$

- Multi-class classification ( $K$  classes).  $y = \mathbb{R}^K$ , e.g.  $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$  for pedestrian,  $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$  for car, and  $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$  for truck.  $h_{\theta}(x) = \mathbb{R}^K$ .  $s_L = K$ .  $K$  output units.  $K \geq 3$

- Cost function

- Recall the cost function for regularized logistic regression was:

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

- Let us denote  $(h_{\theta}(x))_i$  to be the  $i$ th output of  $h_{\theta}(x) \in \mathbb{R}^k$ .

- The regularized cost function for neural networks is:

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

- The difference between the cost function for neural networks and for logistic regression is that we sum across all  $K$  outputs for the neural networks, and we regularize all  $\theta_{ji}^{(l)}$  terms except for  $\theta_{j0}^{(l)}$ , which are the bias terms.

### Backpropagation

- To find  $\min_{\theta} J(\theta)$  via gradient descent, we need  $J(\theta)$  and  $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta)$

- Let  $\delta_j^{(l)}$  be the “error” of node  $j$  in layer  $l$ . In other words, it’s how far off  $a_j^{(l)}$  is.

- We note  $\delta_j^{(L)} = a_j^{(L)} - y_j = (h_{\Theta}(x))_j - y_j$ . We can vectorize it via  $\delta^{(L)} = a^{(L)} - y$ , where each term is a vector whose dimension is equal to the number of output units, or  $s_L$ .

- For the hidden layers:

$$\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} .* g'(z^{(l)}) = (\Theta^{(l)})^T \delta^{(l+1)} .* a^{(l)} .* (1 - a^{(l)})$$

where “.” is a Matlab symbol meaning element-wise multiplication between the two vectors and “1” in the last expression is a vector of 1’s of size  $s_l$ .

- Note that there is no  $\delta^{(1)}$  because the first layer is the input layer, which doesn’t have any error associated with it.

— The term “backpropagation” comes from the fact we compute  $\delta$  starting from the last layer and moving to the first layer.

— Ignoring regularization (or if  $\lambda = 0$ ),  $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$

— Backpropagation algorithm:

Input: training set  $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$

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

For  $i = 1, \dots, m$ :

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

Perform forward propagation to compute  $a^{(l)}$  for  $l = 2, 3, \dots, L$

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

Compute  $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$  (see above algorithm)

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$  (or using matrices:  $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$ )

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

$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$ , which you can use in gradient descent or other optimization algorithm

— Backpropagation intuition

- Focusing on a single example  $(x^{(i)}, y^{(i)})$ , the case of 1 output unit, and ignoring regularization ( $\lambda = 0$ ):

$$\text{cost}(i) = y^{(i)} \log h_{\Theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\Theta}(x^{(i)}))$$

- We note  $\delta_j^{(l)} = \text{“error”}$  for  $a_j^{(l)}$  (unit  $j$  in layer  $l$ ). More formally, the delta values are the derivative of the cost function:  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(i)$  (for  $j \geq 0$ )

- We would like to change the neural networks weights to change the  $z_j^{(l)}$  values, thus changing the neural network output  $h_{\Theta}(x)$  and thus changing the cost.

— Optimization functions such as fminunc require vectors of numbers for the parameters  $\Theta$  and partial derivatives  $D$ , but  $\Theta$  and  $D$  are a vector of matrices, so they need to be unrolled into long vectors before calling fminunc and then reshaped after the call.

- For example, if  $\Theta^{(1)} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ ,  $\Theta^{(2)} = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$ , and  $\Theta^{(3)} = \begin{bmatrix} 3 & 3 & 3 \end{bmatrix}$ , then unrolling  $\Theta$  gives you a  $15 \times 1$  vector whose transpose is equal to:  

$$[1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 3 \quad 3 \quad 3]$$

- Here is the concrete algorithm:  
 Have initial parameters  $\Theta^{(1)}, \dots, \Theta^{(L-1)}$   
 Unroll  $\Theta$  to get initialTheta  
 Call `fminunc(costFunction, initialTheta, options)`

//costFunction returns  $J(\Theta)$  as jVal and  $D$  as gradientVec given  $\Theta$  as thetaVec  
 Function costFuntion(thetaVec):

From thetaVec, reshape to get  $\Theta^{(1)}, \dots, \Theta^{(L-1)}$

Use forward prop and back prop to compute  $D^{(1)}, \dots, D^{(L-1)}$  and  $J(\Theta)$

Unroll  $D^{(1)}, \dots, D^{(L-1)}$  to get  $D$

Return  $D$  as gradientVec and  $J(\Theta)$  as jVal

— Harder to check for bugs: even if  $J(\Theta)$  is decreasing with each iteration of gradient descent, the neural network may actually have a higher level of error. Gradient checking that eliminates most of these problems:

- We note that  $\frac{d}{d\theta}J(\theta) \approx \frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$  using just the *slope*  $= \frac{\Delta y}{\Delta x}$  definition, where  $\theta \in \mathbb{R}$ . This is the two sided difference estimate, which is usually more accurate than the one-sided difference estimate of  $\frac{d}{d\theta}J(\theta) \approx \frac{J(\theta+\epsilon)-J(\theta)}{\epsilon}$
- If  $\theta \in \mathbb{R}^n$ , then the numerical approximation for the gradient is:  

$$\frac{\partial}{\partial \theta_i}J(\theta) \approx \frac{J(\theta_1, \dots, \theta_{i-1}, \theta_i + \epsilon, \theta_{i+1}, \dots, \theta_n) - J(\theta_1, \dots, \theta_{i-1}, \theta_i - \epsilon, \theta_{i+1}, \dots, \theta_n)}{2\epsilon}$$
- Example:

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

- The above algorithm can be used for the unrolled parameters in neural network gradient descent. (The weights in neural network are a 3D matrix, but the above algorithm works on vectors, so that's why we need to unroll the weights.)
- To verify the neural network gradient descent is working, verify the numerical approximation for  $\frac{\partial}{\partial \theta}J(\theta)$  is close to the computed derivative for each unrolled parameter  $D^{(l)}$  (i.e. for  $D^{(1)}, D^{(2)}, \dots$ ).
- 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 for every computation of the costFunction), then your code will be very slow.

— Random initialization

- If we begin with each weight being zero (or for any common value), then the weights coming out of a given node will always all be equal, i.e.  $\Theta_{0n}^{(j)} = \Theta_{1n}^{(j)} = \dots = \Theta_{mn}^{(j)}$ . This is because the associated  $a$ ,  $\delta$ , and  $\frac{\partial}{\partial \theta}$  values are all identical.
  - In the above scenario, all hidden units would be calculating the same feature.
  - Thus, initialize each value of  $\Theta_{ij}^{(l)}$  to a random value in  $[-\epsilon, \epsilon]$
- Pick a network architecture (i.e. the number of layers and number of nodes in each hidden layer).
- Number of input units: Dimension of feature  $x^{(i)}$
  - Number of output units: Number of classes
  - Reasonable default: 1 hidden layer, or if >1 hidden layer, have same number of hidden unit sin every layer (usually the more, the better, but more computationally expensive). You usually want roughly 1-4x the number of units per hidden layer.
- High level algorithm to train a neural network
- Randomly initialize weights
- for each example  $(x^{(i)}, y^{(i)})$ :
- Forward propagate (compute  $a^{(l)}$  for  $l = 2 \dots, L$ ) to get  $h_{\theta}(x^{(i)})$
- Compute cost function  $J(\theta)$
- Backprop to compute partial derivatives  $\delta^{(l)}$  for  $l = L, L - 1, \dots, 2$
- Update  $\Delta^{(l)}$
- Compute  $D$  and  $\frac{\partial}{\partial \theta_{jk}^{(l)}} J(\theta)$
- Use gradient checking to verify  $\frac{\partial}{\partial \theta_{jk}^{(l)}} J(\theta)$  is similar to the numerical estimate.
- (Disable gradient checking after performing it.)
- Use gradient descent or advanced optimization method to minimize  $J(\theta)$
- Note that  $J(\theta)$  is non-convex, so it is susceptible to local-optima, but in practice, this is usually not a huge problem.

## Advice for Applying Machine Learning

### Evaluating a Learning Algorithm

- If your hypothesis is not doing well, here are some things to try next:
- Get more training examples
  - Try smaller sets of features (to prevent overfitting)
  - Try getting additional features (if current set of features is not informative enough)
  - Try adding polynomial features ( $x_1^2, x_2^2, x_1x_2$ , etc)
  - Try increasing or decreasing  $\lambda$
- Machine learning diagnostic: a test that you can run to gain insight into what is or isn't working with a learning algorithm and to gain guidance as to how best to improve its performance.

- Diagnostics can take time to implement, but doing so can be a very good use of your time.
- Evaluate your hypothesis by splitting roughly 70% into the training set and 30% into the test set.

- A random subset should go into the training/test set, especially if data is ordered.

— Notation:

- $m$  = number of training examples
- $m_{test}$  = number of test examples
- $(x_{test}^{(i)}, y_{test}^{(i)})$  denotes the  $i$ th test example
- $J_{test}(\theta)$  is the error on the test set.

— Training/testing procedure for linear regression:

- First, learn parameter  $\theta$  from training data (minimizing training error  $J(\theta)$ )

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

- Second, compute the test error using the learned  $\theta$ :

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

— Training/testing procedure for logistic regression is same as linear regression, except the test set error is:

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

- For classification, you can use misclassification error (0/1 misclassification error):

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

$$\text{And now } J_{test}(\theta) = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_{\theta}(x_{test}^{(i)}), y_{test}^{(i)}).$$

— Once parameters  $\theta$  have been fit to some set of data (training set), the error of the parameters as measured on that data (the training error  $J(\theta)$ ) is likely to be lower than the actual generalization error due to overfitting.

— We cannot just use the test set to select the best model, as this would also likely result in overfitting. This is because the model selection parameters are fitted to the test set.

- Example: I propose the following procedure. Assume we're doing polynomial regression (via linear regression), where  $d$  is the degree of the polynomial. (Ex, if  $d = 3$ ,  $h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$ ). For each polynomial degree  $d = 1, \dots, 10$ , we perform linear regression by calculating the  $\theta$  that minimizes  $J_{train}(\theta)$ . With the optimal  $\theta$ , we calculate  $J_{test}(\theta)$ . We then select the polynomial of degree  $d$  where  $J_{test}(\theta)$  is minimized, and conclude that this model generalizes best. (Example, if polynomial of degree 5 has the smallest  $J_{test}(\theta)$ , we conclude that this model generalizes the best.) We say that the generalization error is  $J_{test}(\theta)$ . But this is problematic: here,  $d$  acts as another parameter for the

model (similar to the  $\theta$  parameters), and is fit to the test set. Hence,  $J_{test}(\theta)$  is likely to be an overly optimistic estimate of generalization error.

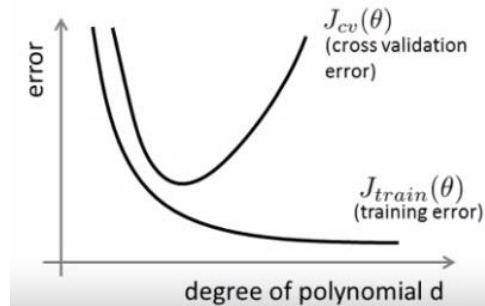
- This issue is similar in nature to how after selecting the best  $\theta$  based on  $J_{train}(\theta)$ , estimating that  $J_{train}(\theta)$  is the generalization error is not correct.
- The solution to this is to split the data set into three sets: training set, cross validation (CV) set (or alternatively “validation set”), and test set.
- Roughly 60% for training, 20% for CV, and 20% for test.
  - Notation:  $(x_{cv}^{(i)}, y_{cv}^{(i)})$  denotes the  $i$ th CV example
- The training and test error are the same as before. The cross validation error is similar:

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

- The cross validation set is now used to set the model.
- We revise the polynomial regression example procedure from above. for each polynomial degree  $d$ , we pick the  $\theta$  that minimizes  $J_{train}(\theta)$ . We then calculate  $J_{cv}(\theta)$  using that  $\theta$  for each  $d$ . We pick the model with the minimum  $J_{cv}(\theta)$ , and estimate that the generalization error of the model is  $J_{test}(\theta)$  for the chosen  $\theta$  and  $d$ .

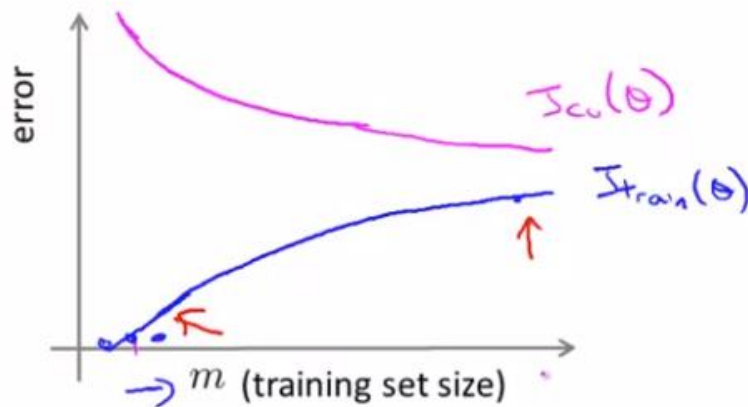
## Bias and Variance

- Diagnosing Bias vs. Variance (i.e. are you underfitting or overfitting?)
- If we plot a graph between the degree of polynomial  $d$  used for the model on the x-axis and  $J(\theta)$  for training and cross-validation, we get:



- The plot of  $J_{cv}(\theta)$  and  $J_{test}(\theta)$  look very similar.
  - From this graph, we see that if you have high  $J_{cv}(\theta)$  and high  $J_{test}(\theta)$  (i.e.  $J_{cv}(\theta) \approx J_{train}(\theta)$ ), you are likely underfitting (i.e. have high bias).
  - If you have high  $J_{cv}(\theta)$  but low  $J_{test}(\theta)$  (i.e.  $J_{test}(\theta) \ll J_{cv}(\theta)$ ), you likely have high variance (overfitting). ( $\ll$  is a math symbol that means “much less than”).
- Regularization: a small  $\lambda$  results in high variance (overfitting), an intermediate  $\lambda$  is “just right”, and a large  $\lambda$  results in high variance (underfitting).
- Notation:  $J(\theta)$  is the training squared-error with the regularization term.
  - $J_{train}(\theta), J_{cv}(\theta), J_{test}(\theta)$ , defined above, are the corresponding squared-error functions that do NOT include the regularization term.

- To choose the regularization parameter  $\lambda$ , for  $\lambda = 0, 0.01, 0.02, 0.04, \dots, 10.24$ , compute the  $\theta$  that minimizes  $J(\theta)$ . Then compute  $J_{cv}(\theta)$  for each  $\lambda$ , and select the  $\theta$  and  $\lambda$  that has the minimum  $J_{cv}(\theta)$ . Finally, report the estimate for the error as  $J_{test}(\theta)$  for the chosen  $\theta$  and  $\lambda$ .
- A small  $\lambda$  results in results in high  $J_{cv}(\theta)$  and  $J_{test}(\theta)$  but low  $J_{train}(\theta)$ . A large  $\lambda$  has high  $J_{cv}(\theta)$ ,  $J_{test}(\theta)$ , and  $J_{train}(\theta)$ . You want to pick the  $\lambda$  that minimizes  $J_{cv}(\theta)$ .
- Learning curves are useful to plot to sanity check the algorithm is working, and for improving the performance of the algorithm.
  - Plot the training set size  $m$  against  $J_{train}(\theta)$  and  $J_{cv}(\theta)$  by artificially limiting the training set by choosing various random subsets.
  - Evaluate  $J_{cv}(\theta)$  against the  $m$  training examples to optimize  $J_{train}(\theta)$ . (Do not evaluate  $J_{cv}(\theta)$  against all training examples.)
  - As the training size increases, the training increases since it is harder to fit the model across all examples. But the cross-validation error decreases because it generalizes better:



- If you have high bias (underfitting), then  $J_{cv}(\theta)$  will flatten out very soon since the overly simple model cannot improve much.  $J_{train}(\theta)$  will be very close to the  $J_{cv}(\theta)$  as  $m$  increases. Both  $J_{train}(\theta)$  and  $J_{cv}(\theta)$  will be high as  $m$  increases. ( $N$  in below figure is  $m$ , the training set size, and  $J_{cv}(\theta)$  looks similar to  $J_{test}(\theta)$ .)

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

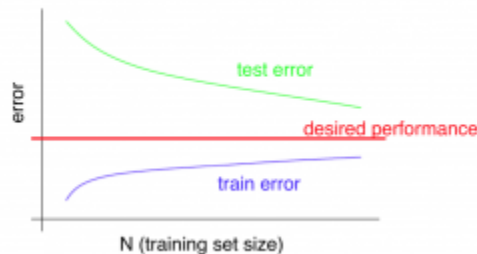


- If a learning algorithm is suffering from high bias, getting more training data will not (by itself) help much.



- If you have high variance (overfitting), then  $J_{train}(\theta)$  will remain small (though still increasing with  $m$ ) since it can fit pretty well. But  $J_{cv}(\theta)$  will remain pretty high.

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



- If the learning algorithm is suffering from high variance, getting more training data is likely to help.
- Conclusion for how to things to try to improve the performance of the learning algorithm.
- Getting more training examples helps fix high variance. (Does not help if you have high bias.)
  - Trying smaller sets of features helps fix high variance. (Does not help if you have high bias.)
  - Getting more additional features helps fix high bias (to form a more complex hypothesis).
  - Adding polynomial features helps fix high bias (to form a more complex hypothesis).
  - Decreasing  $\lambda$  helps fix high bias.
  - Increasing  $\lambda$  helps fix high variance.
- Neural networks and overfitting
- Small neural networks have fewer parameters (fewer hidden units/layers) and are more prone to underfitting. They are computationally cheaper.
  - Large neural networks have more parameters (more hidden units/layers) and are more prone to overfitting. They are computationally expensive. Use regularization  $\lambda$  to address overfitting. Larger neural network with regularization is usually better than smaller neural network if you have enough computational resources.
  - Can use train/cv/test sets to determine architecture of neural network (number of hidden layers and number of hidden units per hidden layer).

## Machine Learning System Design (Building a Spam Email Classifier)

### Prioritizing What to Work On

- One approach for spam email classification: Supervised learning, where  $x$  = features of the email and  $y$  = spam (1) or not spam (0). Feature  $x$  is a list of 100 words that are indicative of spam and not spam. Create a feature vector of size 100, placing a 1 if the corresponding word appears in the email and 0 if the word does not appear.
- How to spend your time to make it have low error? Here are some ideas, but it is difficult to tell which of the options will be most helpful.

- Collect lots of data: may help, but not always.
  - Develop more sophisticated features based on email routing information (from email header).
  - Develop sophisticated features for message body, e.g. should “discount” and “discounts” be treated as the same word? Features about punctuation?
  - Develop sophisticated algorithm to detect misspellings. (Spam detectors may have a harder time detecting these misspelled words as words that normally appear in spam.)
- Recommended approach to prioritize what you should work on:
- Start with a simple algorithm that you can implement quickly (<24 hrs).
  - Plot learning curves to decide if more data, more features, etc. are likely to help.
  - Error analysis: manually examine the examples (in cross validation set) that your algorithm made errors on. See if you spot any systematic trend in what type of examples it is making errors on.
- Error analysis can help provide a systematic way to decide what to prioritize
- For the examples in the cross validation set that were misclassified, classify these examples (e.g. pharmacy (sell drugs), replica (sell fakes), phishing emails) and count the frequency of each category – this shows what categories of examples the algorithm is not performing well in. Next, see what cues (features) you think would have helped the algorithm classify them correctly.
  - We use the cross validation set instead of the test set, since otherwise we may pick features that would bias the test set error.
- Numerical evaluation (aka real-number evaluation) is important: you want to quantify how well your algorithm is performing by using a single, numerical value.
- Error analysis may not be helpful for deciding if trying something new (e.g. treating similar words such as discount/discounts/discounted as the same word) is likely to improve the performance. The only solution is to try it and see if it works, and hence a numerical evaluation (e.g. cross validation error) of algorithms’ performance.
  - (Side note: stemming software, such as “porter stemmer”, can be used to try to classify discount/discounts/discounted as the same word).

## Handling Skewed Data

- Some classes may have very few examples. (Ex: only 0.5% of patients have cancer.) These are skewed classes.
- Naively saying all patients do not have cancer has a 99.5% accuracy. Hence, an improvement from 99.2% accuracy to 99.5% doesn’t necessarily mean you have a better classifier -- you might just be predicting fewer patients have cancer.
- Precision/Recall fixes this.
- True positive: predicted true  $h_\theta(x) = 1$  and example is true  $y = 1$
  - True negative: predicted false and example is false
  - False positive: predicted true and example is false

- False negative: predicted false and example is true.
- Precision: of all examples where we predicted  $y = 1$ , what fraction actually is  $y = 1$ ?
- Precision =  $\frac{\# \text{ True positives}}{\# \text{ predicted positive}} = \frac{\# \text{ True positives}}{\# \text{ True positive} + \# \text{ False positive}}$
- Recall: of all examples where  $y = 1$  actually, what fraction did we predict  $y = 1$ ?
- Recall =  $\frac{\# \text{ True positives}}{\# \text{ actual positives}} = \frac{\# \text{ True positives}}{\# \text{ True positive} + \# \text{ False negative}}$
- Predicting  $y = 1$  all the time will result in 100% recall but low precision.
- By convention, we set  $y = 1$  in presence of rare class that we want to detect (i.e.  $y = 1$  if the example belongs in that rare class).
- Tradeoff between precision and recall: suppose we use logistic regression and we change the decision boundary (i.e. the threshold of predicting 0 or 1) from 0.5 to 0.7. (Now, predict 1 in  $h_\theta(x) \geq 0.7$  and predict 0 if  $h_\theta(x) < 0.7$ ). This will result in higher precision but lower recall.
- You are more confident in the people you say are positive, but you don't capture as many positive cases.
- Suppose we use logistic regression and we lower the threshold for predicting 0 or 1. This results in lower precision but higher recall.
- You are more confident that you capture more of the positive cases, but you are not as confident in the people you say are positive.
- Thus, there is a tradeoff between precision and recall, which can be done by varying the threshold for predicting 0 or 1.
- The  $F_1$  Score (or F score) converts precision and recall into a single numerical value that can be used for numerical analysis.
- The average  $\frac{P+R}{2}$  does not work well. ( $P$  = precision,  $R$  = recall.) You should prefer an algorithm with  $P = 0.45$  and  $R = 0.45$  over one with  $P = 0$  and  $R = 1.0$ , but the average does not.
  - The  $F_1$  Score is better:  $2 \frac{PR}{P+R}$ .
- Remember to select the threshold by optimizing the F score on the cross-validation set, not the test set.

## Using Large Datasets

- Assumption #1: feature  $x \in \mathbb{R}^{n+1}$  has sufficient information to predict  $y$  accurately. In other words, given the input  $x$ , can a human expert confidently predict  $y$ .
- Example: Fill in the blank with (to, two, or too): for breakfast I ate \_\_\_\_ eggs. An “English expert” can fill it in appropriately with “two” given the rest of the sentence as an input feature.
  - Not an example: predict the housing price from only size ( $ft^2$ ) and nothing else. Even a professional realtor would not be able to do this.
- Assumption #2: use a learning algorithm with many parameters (e.g. logistic regression or linear regression with many features; neural network with many hidden units).
- “Low bias” algorithms, so  $J_{train}(\theta)$  will be small.

- If assumption #1 and #2 are true, then using a very large training set would help, as you're unlikely to overfit.
  - $J_{train}(\theta) \approx J_{test}(\theta)$  and thus  $J_{test}(\theta)$  is small (assuming  $J_{train}(\theta)$  is small).

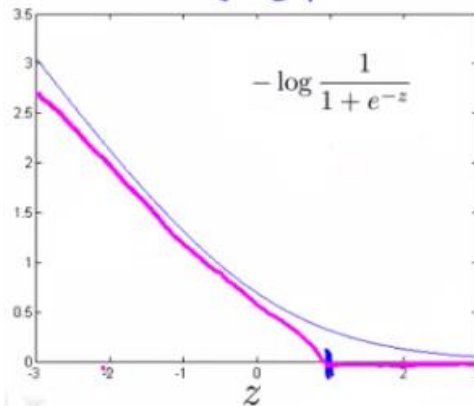
## Support Vector Machines

### Optimization Objective

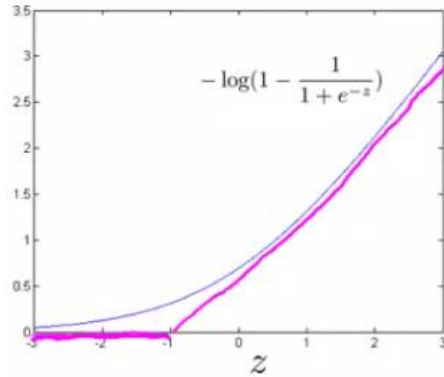
- The support vector machine defines a cost function that can be intuitively transformed from the logistic regression cost function.
- Recall in logistic regression, the cost of a single example is:

$$\begin{aligned}
 J(\theta) &= -(y \log h_{\theta}(x) + (1 - y) \log(1 - h_{\theta}(x))) \\
 &= -y \log \frac{1}{1 + e^{-\theta^T x}} - (1 - y) \log \left(1 - \frac{1}{1 + e^{-\theta^T x}}\right)
 \end{aligned}$$

- If  $y = 1$ , then the cost of a single example in logistic regression is  $-\log \frac{1}{1 + e^{-z}}$ , where  $z = \theta^T x$ . The support vector machine uses similar function for this cost:  $cost_1(z) = \max(0, k(1 - z))$ , for some slope  $k$ . See below graph for comparison. (Magenta line is for support vector machine cost, blue line is for logistic regression cost.)



- If  $y = 0$ , then the cost of a single example in logistic regression is  $-\log \left(1 - \frac{1}{1 + e^{-z}}\right)$ . The support vector machine uses  $cost_0(z) = \max(0, k(1 + z))$ . See below graph for comparison. (Magenta line is for support vector machine cost, blue line is for logistic regression cost.)



— We can transform the regularized logistic regression cost function to the regularized support vector machine cost function by substituting in  $cost_0(z)$  and  $cost_1(z)$ .

- Recall regularized logistic regression function is (we distributed the outermost negative sign into both terms):

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

- Now substitute in  $cost_0(z)$  and  $cost_1(z)$  for SVM cost function:

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

- By convention, we multiply the whole expression by  $m$  and  $C$ , where  $C = \frac{1}{\lambda}$ .  
Multiplying the cost function by a constant doesn't affect what the optimal  $\theta$  is.

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

- To regularize more (i.e. reduce overfitting), we decrease  $C$ , and to regularize less (i.e. reduce underfitting), we increase  $C$ .

— Unlike the hypothesis of a logistic regression, the hypothesis of the support vector machine is not interpreted as the probability of  $y$  being 1 or 0. Instead, it outputs either 1 or 0 (i.e. is a discriminant function):

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

— If  $C$  is very large and the data is linearly separable (i.e. a straight line can separate the positive and negative classes), then the SVM chooses the decision boundary that results in the largest margin, which is the minimum distance from any of the training examples to the decision boundary.

- This increases the robustness of the SVM.
- Thus, the SVM is often referred to as a large margin classifier.

— If  $C$  is very large, then an outlier can dramatically change the decision boundary. If  $C$  is small, then a single outlier would not have much of an effect on the decision boundary.

— Why support vector machines are large margin classifiers for large  $C$  values:

- As a result of the definitions for  $cost_0(z)$  and  $cost_1(z)$ , we add the constraint that  $z \geq 1$  if  $y = 1$  (not just  $z \geq 0$ ) and  $z \leq -1$  if  $y = 0$  (not just  $z < 0$ ). We add these constraints to make the cost of the single example be 0.
- If we set  $C$  to be very large, then we must choose  $\theta$  parameters such that  $\sum_{i=1}^m y^{(i)} cost_1(\theta^T x) + (1 - y^{(i)}) cost_0(\theta^T x) = 0$ , which reduces our cost function to be equal to:

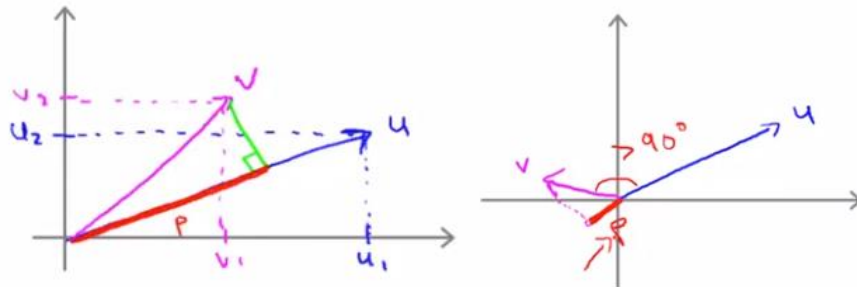
$$J(\theta) = C \cdot 0 + \frac{1}{2} \sum_{j=1}^n \theta_j^2 = \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

- For simplification, assume  $\theta_0 = 0$ . We see that our goal of minimizing the cost function can be reduced to:

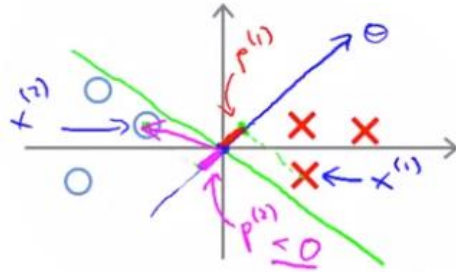
$$J(\theta) = \frac{1}{2} \sum_{j=1}^n \theta_j^2 = \frac{1}{2} (\theta_1^2 + \dots + \theta_n^2) = \frac{1}{2} \left( \sqrt{\theta_1^2 + \dots + \theta_n^2} \right)^2 = \frac{1}{2} \|\theta\|^2$$

Where  $\|\theta\|$  is the norm (or the length) of the  $\|\theta\|$  vector

- Review: the projection of vector  $v$  onto vector  $u$  is found by taking a right angle from  $u$  to the end of  $v$  creating a right triangle. See below diagram. The following property holds:  $u^T v = p \cdot \|u\|$ , where  $p$  is the length of projection of vector  $v$  onto vector  $u$ .  $p$  is negative if  $u$  and  $v$  form an angle greater than  $90^\circ$ .



- So our goal is to  $\min_{\theta} \frac{1}{2} \|\theta\|^2$  s.t.  $\theta^T x^{(i)} \geq 1$  if  $y^{(i)} = 1$  and  $\theta^T x^{(i)} \leq -1$  if  $y^{(i)} = 0$ . Using projections, we can rewrite the optimization problem as:  
 $\min_{\theta} \frac{1}{2} \|\theta\|^2$  s.t.  $p^{(i)} \cdot \|\theta\| \geq 1$  if  $y^{(i)} = 1$  and  $p^{(i)} \cdot \|\theta\| \leq -1$  if  $y^{(i)} = 0$   
 where  $p^{(i)}$  is the projection of  $x^{(i)}$  onto the vector  $\theta$ .
- Assuming  $\theta_0 = 0$ , the decision boundary goes through the origin, and  $\theta$  is the vector perpendicular to the decision boundary that goes through the origin. We see that  $p^{(i)}$  is equal to the margin of training example  $x^{(i)}$ :



- If the margins are small, then the absolute value of  $p^{(i)}$  is small, which means that  $\|\theta\|$  must be large since  $|p^{(i)}| \cdot \|\theta\| \geq 1$ . This is not good since we want to minimize  $\frac{1}{2} \|\theta\|^2$ . If the margins are large, then  $\|\theta\|$  will be small, which is good. Hence, this classifier chooses the largest margin.
- We note that we assumed  $\theta_0 = 0$  for this proof. If  $\theta_0 \neq 0$ , then the decision boundary does not go through the origin, but we take for granted that the statement still holds true (i.e. this classifier picks the largest margin for large  $C$ ).

## Kernels

- The main technique for adapting SVMs to develop complex nonlinear classifiers is to use kernels.
- Notation: let  $f_1$  be feature 1,  $f_2$  be feature 2, etc.
  - Ex: assume we predict  $y = 1$  if  $\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 \geq 0$ . Then  $f_1 = x_1, f_2 = x_2, f_3 = x_1 x_2$ .
- Question: what is the best choice of features  $f_1, f_2, \dots$  of higher order polynomials?
- The Kernel is a similarity function between an example  $x$  and a given landmark  $l^{(i)}$ .
  - A landmark is an artificial example containing values  $x_1, \dots, x_n$ . Given an example  $x$ , we can calculate new features  $f^{(1)}, f^{(2)}, \dots$ , etc, one for each landmark.
  - Given an example and the  $i$ th landmark  $l^{(i)}$ ,  $f_i = s = \text{similarity}(x, l^{(i)}) = \exp\left(-\frac{\|x - l^{(i)}\|^2}{2\sigma^2}\right) = \exp\left(-\frac{\sum_{j=1}^n (x_j - l_j^{(i)})^2}{2\sigma^2}\right)$ . This specific similarity function is known as the Gaussian kernel.
    - If  $x \approx l^1$ , then  $f_1 \approx \exp\left(-\frac{0^2}{2\sigma^2}\right) = 1$
    - If  $x$  is far from  $l^1$ , then  $f_1 = \exp\left(-\frac{\text{large\_number}^2}{2\sigma^2}\right) \approx 0$
- In the kernel (i.e. the similarity function), the smaller  $\sigma$  is, the quicker the value of the feature  $f$  decreases as you move away from the landmark  $l$ .
- Example use of kernels: assume that we predict “1” when  $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0$ , where  $f_1, f_2, f_3$  are computed using landmarks  $l_1, l_2, l_3$  along with kernels, and assume that after training our model, we get  $\theta_0 = -0.5, \theta_1 = 1, \theta_2 = 1, \theta_3 = 0$ . For a given example  $x$ , we will predict “1” if the example is close to  $l_1$  and/or  $l_2$ ; otherwise we predict 0.

— In practice, we place a landmark at the same locations as all the training example, i.e.  $l^{(1)} = x^{(1)}, l^{(2)} = x^{(2)}, \dots, l^{(m)} = x^{(m)}$ . For  $m$  training examples, this gives us  $m$  landmarks.

- Given example  $x$ , we get a feature vector  $f$ , where  $f = \begin{bmatrix} f_0 \\ \dots \\ f_m \end{bmatrix}$ .  $f_0 = 1$ , whereas  $f_1, \dots, f_m$  are calculated using the kernel.
- $f^{(i)}$  represents the feature vector of  $x^{(i)}$ , the  $i$ th training example.

— Training SVMs with kernels: goal is to solve the optimization problem:

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

- Note that  $n = m$  because we have  $m$  features  $f$
  - If  $\theta = \begin{bmatrix} \theta_1 \\ \dots \\ \theta_m \end{bmatrix}$  (i.e. ignoring  $\theta_0$ ), then  $\frac{1}{2} \sum_{j=1}^n \theta_j^2 = \theta^T \theta = \|\theta\|^2$ .
  - Many implementations use an additional matrix  $M$  for the regularization term:  $\theta^T M \theta$ .  $M$  depends on the kernel being used. This modification allows us to scale to large training sets, as we create a new feature for each new training example.
  - Recommended to use off-the-shelf software to solve this cost function instead of implementing it yourself.
- Predicting using SVMs with kernels: given  $x$ , compute features  $f \in \mathbb{R}^{m+1}$ . Predict  $y = 1$  if  $\theta^T f \geq 0$ .
- We could apply the idea of kernels to other algorithms like logistic regression, but the computational tricks applicable to SVMs do not generalize well to the other algorithms. Thus, using kernels with other algorithms is going to be slow.
- Choosing SVM parameters:
- A large  $C$  tends to result in lower bias and high variance. A small  $C$  tends to result in high bias, low variance. Recall  $C = \frac{1}{\lambda}$ .
  - A large  $\sigma^2$  causes features  $f$  to vary more smoothly, resulting in higher bias and lower variance. A small  $\sigma^2$  causes features  $f$  to vary more abruptly, resulting in lower bias and higher variance.

## SVMs in Practice

- Use SVM software package (e.g. liblinear, libsvm, ...) to solve for parameters  $\theta$ .
- Need to specify: choice of parameter  $C$  and choice of kernel (similarity function)
- Ex: no kernel (aka “linear kernel”): predict  $y = 1$  if  $\theta^T x \geq 0$ . Use this kernel if number of features  $n$  is large and number of training examples  $m$  is small. This is because this is a simple model, and helps prevent overfitting.



- Ex: Gaussian kernel:  $f_i = \exp\left(-\frac{\|x-l^{(i)}\|^2}{2\sigma^2}\right)$ , where  $l^{(i)} = x^{(i)}$ . Need to furthermore choose  $\sigma^2$ . Use this kernel if number of features  $n$  is small and/or number of examples  $m$  is large. This creates a more complex model.
  - Some software libraries allow you to specify a kernel function that takes in a training example and a landmark (which itself is an example).
- Perform feature scaling before using the Gaussian kernel, otherwise some features would be largely ignored.
- $\|x - l\|^2 = (x_1 - l_1)^2 + (x_2 - l_2)^2 + \dots + (x_n - l_n)^2$ . If the first feature takes on values between 0 to 4000 and the second feature takes values between 0 and 5, then the second feature would have little impact on the similarity function value.
- Note all similarity functions make valid kernels. They need to satisfy a technical condition called “Mercer’s Theorem” to make sure SVM packages’ optimizations run correctly and do not diverge.
- Linear kernel and Gaussian kernel are by far the most common kernels that satisfy Mercer’s Theorem
  - Polynomial kernel:  $\text{similarity}(x, l) = (x^T l + c)^d$  for some constant  $c$  and degree  $d$ . Polynomial kernel often performs worse than Gaussian kernel and not used often. Used for data where  $x$  and  $l$  are all strictly non-negative.
  - More esoteric kernels: string kernel (can used if input is a string), chi-square kernel, histogram intersection kernel
- Make the choice of kernel and parameters  $C, \sigma^2$ , etc based on the cross-validation data.
- Multiclass classification
- Many SVM already have built-in multiclass classification functionality.
  - Otherwise, use one-vs.-all method. (Train  $K$  SVMs given  $K$  classes, one to distinguish  $y = i$  from the rest, for  $i = 1, 2, \dots, K$ . Get  $\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(K)}$ , and pick class  $i$  with largest  $(\theta^{(i)})^T x$ .)
- Logistic regression vs SVMs
- Let  $n$  be the number of features ( $x \in \mathbb{R}^{n+1}$ ),  $m$  be the number of training examples.
  - If  $n$  is large relative to  $m$ : use logistic regression, or SVM without a kernel (“linear kernel”).
  - If  $n$  is small and  $m$  is intermediate: use SVM with Gaussian kernel
  - If  $n$  is small,  $m$  is large (above ~50,000): create/add more features, then use logistic regression or SVM without a kernel. This is because SVM with Gaussian kernel is too computationally slow.
  - Neural network likely to work well for most of these settings, but may be computationally slower to train.
- SVM optimization problem is a convex optimization problem: always find global minimum (or close to it with optimizations).

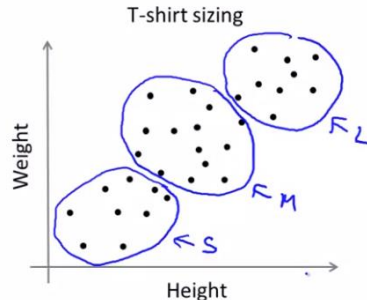
## Unsupervised Learning

### Clustering

- In unsupervised learning, we use unlabeled data, unlike in supervised learning, where we use labeled data.
- In unsupervised learning, we ask the algorithm to find some structure in the data.
- A clustering algorithm is one type of unsupervised learning problem: finding coherent “clusters” or groupings/subsets of data.
- Applications of clustering: market segmentation (grouping customers to target them differently), social network analysis (e.g. look at people’s groups of social networks), organize computing clusters (e.g. organize data centers and servers better by seeing which typically work together), astronomical data analysis (e.g. understand galaxy formation)
- K-means is by far the most popular and widely used clustering algorithm
- K-means high-level algorithm:
  - Randomly initialize  $K$  points, each containing a random value for each feature  $x_i$ . These points are referred to as the “cluster centroids”.  $K$  is the number of clusters you would like.
  - The remaining steps are iterative (looping multiple times), consisting of two steps: cluster assignment step and move centroid step, the former being the first step.
  - Cluster assignment step: Go through each example, and assign each data point to whichever cluster centroid is the closest.
  - Move centroid step: move each cluster centroid to the average value of the examples that are assigned to it.
  - The algorithm converges when the cluster centroids no longer move.
- Formal presentation of the K-means algorithm:
  - Input:  $K$  (number of clusters) and training set  $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ . Each  $x^{(i)} \in \mathbb{R}^n$  (dropping the  $x_0 = 1$  convention).  
Randomly initialize  $K$  cluster centroids:  $\mu_1, \mu_2, \dots, \mu_k \in \mathbb{R}^n$   
Repeat until convergence or max # of iterations {  
    // cluster assignment step  
    for  $i = 1$  to  $m$ :  
        // Below expression is equal to:  $\min_{k \in [1, K]} \|x^{(i)} - \mu_k\|^2$   
         $c^{(i)} := \text{index (from 1 to } K) \text{ of cluster centroid closest to } x^{(i)}$   
    // move centroid step  
    for  $k = 1$  to  $K$ :  
         $\mu_k := \text{average (mean) of points assigned to cluster } k$   
    }  
    }
- In the move centroid step, if there are no points assigned to a particular cluster centroid:
  - More common approach: just eliminate that cluster centroid.
  - If you really need  $K$  classes, then re-randomly initialize that cluster centroid.

— K-means also has applications for non-separated clusters (i.e. data that doesn't have good groupings).

- Ex: you want to know who to try to fit for small, medium, and large t-shirt sizes. K-means could assign the following people to the various t-shirt sizes:



— Notation:  $\mu_{c(i)}$  is the cluster centroid of cluster to which example  $x^{(i)}$  has been assigned.

— Optimization objective:

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

- This cost function is also known as the distortion cost function.
- The cluster assignment step minimizes  $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$  with respect to  $c^{(1)}, \dots, c^{(m)}$  while holding  $\mu_1, \dots, \mu_K$  fixed.
- The move centroid step minimizes  $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$  with respect to  $\mu_1, \dots, \mu_K$  while holding  $c^{(1)}, \dots, c^{(m)}$  fixed.

— With each iteration of K-means, the cost should decrease. If it increases, there's a bug in the implementation.

— Recommended strategy for random initialization: randomly pick  $K$  training examples. Set  $\mu_1, \dots, \mu_K$  to be equal to these  $K$  examples.

- K-means can get stuck at local optima if you're unlucky with random optimization.
- Thus, try multiple random initialization:

For  $i = 1$  to num\_random\_initializations: // ~50-1000 random initializations

    Randomly initialize K-means.

    Run K-means. Get  $c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K$

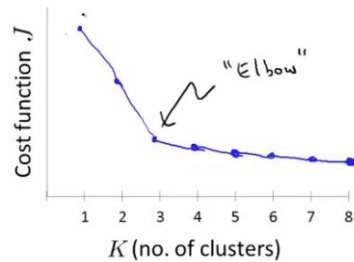
    Compute cost function (distortion)  $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$

— Choosing the right number of clusters is difficult, and is often done by hand.

- What makes it even more difficult is that often there isn't one right answer for a given set of examples, instead being subject to interpretation.

— One method for choosing the right number of clusters is the elbow method:

- Plot the cost  $J$  against the number of clusters  $K$  for  $K = 1, 2, \dots$ , etc. The cost should asymptotically decrease to 0.
- The point where the second derivative increases significantly is the right number of clusters (and looks like the elbow joint in an arm). Example:

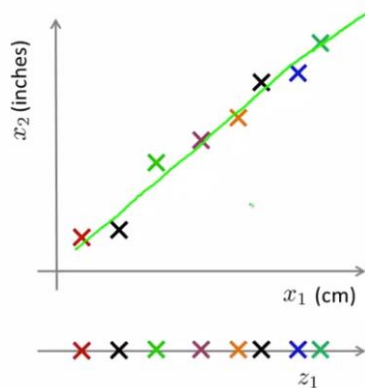


- The elbow method is not used very often, since the curve is often smoother, resulting in no clear “elbow”. The method is worth a shot, but don’t have high expectations for it to solve the problem.
- Sometimes you’re running K-means to get clusters to use for some later/downstream purpose. Another method for choosing  $K$  is to evaluate K-means based on a metric for how well it performs for that later purpose.
- Example: T-shirt sizing. Assume you’re trying to determine the number of different sizes of T-shirts to produce. For a given value of  $K$ , you can evaluate whether all people classified to a particular T-shirt size would fit in that T-shirt well.

### Dimensionality Reduction Motivation

- Dimensionality reduction is another type of unsupervised learning problem
- Applications of dimensionality reduction include data compression and speeding up learning algorithms.
- Data compression by reducing data from 2D into 1D: if two features are highly correlated with each other (or possibly redundant, such as length in meters and length in inches), we can represent each example with just one of the two features by projecting the examples onto the line of best fit.

- Example of data compression (we project each example onto the green line):



- Each example  $x^{(i)}$  was originally 2D (i.e.  $x^{(i)} \in \mathbb{R}^2$ ), but now each  $x^{(i)}$  can be represented by a 1D value  $z^{(i)} \in \mathbb{R}$ .
- This now halves the space/memory requirement to store the data, thus allowing our learning algorithm to run faster.

- The above data compression example of 2D to 1D can be generalized to 3D to 2D (if a plane fits the 3D data very well), or any large dimension to a smaller one (e.g. 1000D to 100D).

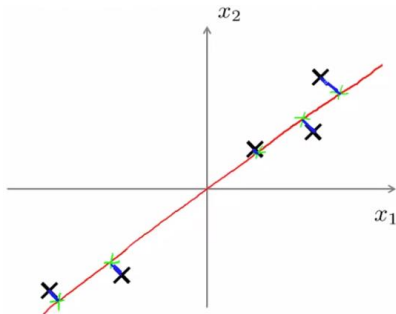
- Notation:  $z_1, \dots, z_k$  represent the compressed features from  $x_1, x_2, \dots, x_n$ , where

$$k \leq n. \text{ A compressed example: } z^{(i)} = \begin{bmatrix} z_1^{(i)} \\ \dots \\ z_k^{(i)} \end{bmatrix}.$$

- Dimensionality reduction is also useful for data visualization: if we have examples with many features, we can reduce the dimensionality of those examples to just 2-3 features, and plot those on a 2D or 3D graph.

### Principal Component Analysis

- By far, the most popular used algorithm for dimensionality reduction is Principal Component Analysis (PCA)
- PCA tries to find the surface with the smallest sum of squares of projection error:



- PCA problem formulation: reduce from  $n$ -dimension to  $k$ -dimension: find  $k$  vectors  $u^{(1)}, \dots, u^{(k)}$  onto which to project the data, so as to minimize the projection error.
  - By convention, we make all  $u^{(i)}$  be unit vectors.
  - For any vector  $u^{(i)}$ , we could have instead used  $-u^{(i)}$  to obtain the same result.
- Note that PCA is not linear regression: linear regression minimizes square “vertical” distances  $(h_{\theta}(x^{(i)}) - y^{(i)})^2$ . In PCA, we minimize the square orthogonal distances.
  - There is also no concept of “y” in PCA. PCA treats all variables symmetrically.
- Data preprocessing: before applying PCA, be sure to perform mean normalization and feature scaling.
- PCA algorithm (reduce data from  $n$ -dimensions to  $k$ -dimensions):
  - Run data preprocessing (mean normalization and feature scaling)
  - Compute “covariance matrix”:  $\Sigma = \frac{1}{m} \sum_{i=1}^m (x^{(i)})(x^{(i)})^T$
  - Compute “eigenvectors” of matrix  $\Sigma$ :  $[U, S, V] = \text{svd}(\Sigma)$
  - Define  $U_{\text{reduce}}$  to be the first  $k$  columns of  $U$ .
  - $z = U_{\text{reduce}}^T x$  // where  $x \in \mathbb{R}^n$  is any example, such as in the training or test set.
  - SVD means “Singular value decomposition”.
  - Instead of using Octave’s “svd( $\Sigma$ )” function, you can also use “eig( $\Sigma$ )”, which computes the same thing, except “svd( $\Sigma$ )” is a bit more numerically stable. The

reason these methods return the same thing is because the covariance matrix always satisfies the mathematical property of being symmetric positive definite.

- Both the  $\Sigma$  matrix and the  $U$  matrix are of dimensions  $n \times n$ .
  - The first  $k$  columns of the  $U$  matrix,  $u^{(1)}, \dots, u^{(k)}$ , are the  $k$  vectors onto which we want to project the data.
  - The  $z$  vector represents the example projected onto  $u^{(1)}, \dots, u^{(k)}$ .
- The computation for the covariance matrix can be vectorized if the examples are in a matrix, where each row is the transpose of an example.

- Let  $X = \begin{bmatrix} x^{(1)T} \\ \dots \\ x^{(m)T} \end{bmatrix}$  be the matrix of examples.
- Then  $\Sigma = \frac{1}{m} X^T * X$

### Applying PCA

- Given a compressed example  $z$ , we can obtain an approximation for the original example  $x$  using the formula:

$$x \approx U_{reduce} \cdot z$$

- Where  $x \in \mathbb{R}^n, U_{reduce} \in \mathbb{R}^{n \times k}, z \in \mathbb{R}^k$
  - The lower the projection error, the better the approximation.
- $k$ , the number of features we compress the  $n$  features to, is referred to as the “number of principle components” (that we retained).
- We typically choose  $k$  to be the smallest value such that:

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

- The top expression,  $\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2$ , is the average squared projection error.
  - The bottom expression,  $\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2$ , is the total variation in the data (distance the data examples are from the origin).
  - Semantically, the inequality means that 99% of the variance is retained.
- One way to choose the  $k$  that satisfies the above inequality is by trying  $k = 1, 2, \dots$  (and computing  $U_{reduce}, z^{(1)}, \dots, z^{(m)}, x_{approx}^{(1)}, \dots, x_{approx}^{(m)}$  for each  $k$ ) but that is inefficient.
- Recall when we call SVD, we get three matrices:  $[U, S, V] = svd(\Sigma)$
  - $S$  is an  $n \times n$  square diagonal matrix (only the diagonal elements are non-zero). Label these diagonal elements as  $s_{11}, s_{22}, \dots, s_{nn}$
  - The following equation holds true:

$$\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} = 1 - \frac{\sum_{i=1}^k s_{ii}}{\sum_{i=1}^n s_{ii}}$$

- Thus, we simply need to call  $\text{svd}(\Sigma)$  once, and pick the smallest value of  $k$  for which  $\frac{\sum_{i=1}^k s_{ii}}{\sum_{i=1}^n s_{ii}} \geq 0.99$ .
- PCA can speed up supervised learning:
  - Imagine your examples have many features. Running supervised learning on these large examples may take a while.
  - With PCA, convert your old training set  $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$  into a new training set with smaller examples  $(z^{(1)}, y^{(1)}), \dots, (z^{(m)}, y^{(m)})$ .
  - Mapping  $x^{(i)} \rightarrow z^{(i)}$  should be defined by running PCA only on the training set. This is because all parameters, including parameters created by PCA such as  $U_{\text{reduce}}$ , should be calculated from the training set. This mapping from  $x^{(i)} \rightarrow z^{(i)}$  can be applied as well to the examples  $x_{cv}^{(i)}$  and  $x_{test}^{(i)}$  in the cross validation and test sets.
- Bad use of PCA: to prevent overfitting.
  - The thinking behind this is to use  $z^{(i)}$  instead of  $x^{(i)}$  to reduce the number of features to  $k < n$ . Since fewer features, less likely to overfit.
  - This might work OK, but isn't a good way to address overfitting. This is because PCA does not use the values  $y$  when throwing away certain data.
  - Use regularization instead.
- Before implementing PCA, first try running whatever you want to do with the original/raw data  $x^{(i)}$ . Only if that doesn't do what you want, then implement PCA and consider using  $z^{(i)}$ .

## Anomaly Detection

### Density Estimation

- Goal of detecting anomalies, which are basically outliers.
- Formally: given a dataset of  $\{x^{(1)}, \dots, x^{(m)}\}$ , is  $x_{test}$  anomalous?
  - We assume the dataset  $\{x^{(1)}, \dots, x^{(m)}\}$  are not anomalous.
- Create a probability model  $p(x)$ . If  $p(x_{test}) < \epsilon$ , we flag it as anomalous. Otherwise, we classify it as OK.
- Application of anomaly detection:
  - Fraud detection:  $x^{(i)}$  are features of user  $i$ 's activities. Model  $p(x)$  from data. Identify unusual users by checking which have  $p(x) < \epsilon$ .
  - Manufacturing: quality assurance. If a specific item behaves differently from the rest, it may be defective.
  - Monitoring computers in a data center:  $x^{(i)}$  = machine  $i$ . Let  $x_1$  = memory use,  $x_2$  = number of disk accesses / second,  $x_3$  = CPU load,  $x_4$  = CPU load / network traffic, etc. If  $p(x) < \epsilon$ , then that machine is behaving anomalous, which may be indicative that it will be going down soon.
- Gaussian/normal distribution review:

- Notation: if variable  $x \in \mathbb{R}$  and  $x$  is normally distributed with mean  $\mu$  and variance  $\sigma^2$ , then we write:  $x \sim \mathcal{N}(\mu, \sigma^2)$
  - Formula:  $p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$
  - Given a dataset  $\{x^{(1)}, \dots, x^{(m)}\}$ , we estimate  $\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}$  and  $\sigma^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)^2$ . These estimates the maximum likelihood estimates of the population parameters  $\mu$  and  $\sigma^2$ .
  - In machine learning, we typically use  $\frac{1}{m}$  for the standard deviation, as opposed to  $\frac{1}{m-1}$  in statistics, as it does not really matter as  $m$  becomes large.
- Given a training set  $\{x^{(1)}, \dots, x^{(m)}\}$ , where each example is  $x \in \mathbb{R}^n$ , then:
- $p(x) = \prod_{j=1}^n p(x_j; \mu_j, \sigma_j^2) = p(x_1; \mu_1, \sigma_1^2) p(x_2; \mu_2, \sigma_2^2) \dots p(x_n; \mu_n, \sigma_n^2)$ , where each feature  $x_j \sim \mathcal{N}(\mu_j, \sigma_j^2)$ .
  - Note the above equation assumes all features are independent. In practice, the algorithm generally works fine whether the variables are independent or not.
  - The problem of estimating  $p(x)$  is referred to as the problem of density estimation.

— Anomaly detection algorithm:

Choose features  $x_i$  that you think might be indicative of anomalous examples.

Fit parameters  $\mu_1, \dots, \mu_m, \sigma_1^2, \dots, \sigma_n^2$ , where:

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

$$\sigma_j^2 = \frac{1}{m} \sum_{i=1}^m (x_j^{(i)} - \mu_j)^2$$

Given new example  $x$ , compute  $p(x)$ :

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

Anomaly if  $p(x) < \epsilon$

- $\mu$  and  $\sigma$  can be vectorized if  $\mu = \begin{bmatrix} \mu_1 \\ \dots \\ \mu_n \end{bmatrix}$  and  $\sigma^2 = \begin{bmatrix} \sigma_1^2 \\ \dots \\ \sigma_n^2 \end{bmatrix}$ :  $\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}$  and  $\sigma^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)^2$ .

## Building an Anomaly Detection System

- To quantitatively evaluate an anomaly detection system, we need some labeled data of anomalous and non-anomalous examples ( $y = 0$  if normal,  $y = 1$  if anomalous).
- Our training set is a set of unlabeled examples (which we assume to be normal examples):  $x^{(1)}, \dots, x^{(m)}$ . Practically speaking, it is okay if a few anomalies slip into the training set.
  - We include the anomalous data in cross-validation and test set:  
 $(x_{cv}^{(1)}, y_{cv}^{(1)}), \dots, (x_{cv}^{m_{cv}}, y_{cv}^{m_{cv}})$  and  $(x_{test}^{(1)}, y_{test}^{(1)}), \dots, (x_{test}^{m_{test}}, y_{test}^{m_{test}})$ .



- Splitting the normal and anomalous examples across training, cv, and test sets:
  - Training set should contain 60% of normal examples ( $y = 0$ ).
  - Cross validation set should contain 20% of normal examples ( $y = 0$ ) and 50% of anomalous examples ( $y = 1$ ).
  - Test set should contain 20% of normal examples ( $y = 0$ ) and 50% of anomalous examples ( $y = 1$ ).
- Not a recommended alternative, but some people use the same examples in the cross validation set and test set (40% of normal examples, 100% of anomalous examples).
- Algorithm to evaluate the anomaly detection system:
 

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

On a cross validation/test example  $x$ , predict

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

Possible evaluation metrics:

  - True positive, false positive, false negative, true negative
  - Precision/recall
  - $F_1$ -score

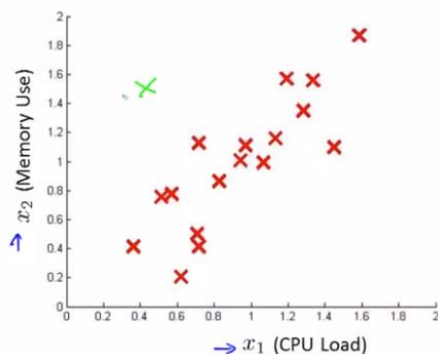
Can use cross validation set to choose parameter  $\epsilon$ .

  - Note that classification accuracy is not a good metric because the data is often skewed: there often are many more normal examples ( $y = 0$ ) compared to anomalous examples ( $y = 1$ ).
  - The cross validation set should be used to determine optimal  $\epsilon$ , what features to use, etc.
- Anomaly detection can also be framed as a supervised learning problem: train a binary classifier via supervised learning (e.g. neural networks) that outputs whether an example is anomalous or not.
- Use Anomaly detection algorithm when you have a very small number of positive (anomalous) examples ( $y = 1$ ) and a large number of negative (normal) examples ( $y = 0$ ).
  - This is because we can fit  $p(x)$  well with a large number of negative examples (i.e. don't need a large number of positive examples).
  - There are many different "types" of anomalies. It is hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far.
- Use supervised learning if there are a large number of positive and negative examples.
  - Enough positive examples for algorithm to get a sense of what positive examples are like; future positive examples likely to be similar to the ones in training set.
- Examples of when to use anomaly detection and supervised learning
  - Anomaly detection: fraud detection (can be a supervised learning problem if many people try to commit fraud), manufacturing, monitoring machines in a data center

- Supervised learning: email spam classifier (we have a large number of spam emails and can know categories they usually fall under), weather prediction (sunny/rainy/etc), cancer classification.
- Choose features that appear to be Gaussian simply by plotting  $p(x_i; \mu_i, \sigma_i^2)$  for feature  $i$ .
  - Applying transformations can make some non-Gaussian variables become Gaussian.
  - Example of transformations:  $\log(x)$ ,  $\log(x + c)$ ,  $x^c$  for some constant  $c$ .
  - Ex of applying transformation: if you apply the log transformation on feature  $x_1$ , then you replace all  $x_1$  with  $\log x_1$
  - The Anomaly detection algorithm might work even if the variables aren't Gaussian, but this usually is a good step to do.
- Use error analysis for coming up with features for the anomaly detection problem
  - One of the most common problems is that the classifier outputs a false negative: an anomalous example is classified as normal.
  - To fix this, examine the anomalous example to see how it differs from the normal examples, and create a new feature to train and classify on.
- Generally choose features that might take on unusually large or small values in the event of an anomaly.
  - Data center monitoring example: you have features  $x_1 = \text{CPU load}$  and  $x_2 = \text{network traffic}$ , and you are worried about a computer getting stuck executing an infinite loop. You might create a new variable  $x_3 = \frac{\text{CPU load}}{\text{network traffic}}$ , as CPU load will increase dramatically with respect to network traffic for an infinite loop scenario, causing  $x_3$  to increase dramatically.

## Multivariate Gaussian Distribution

- Some variables may be highly correlated with each other.



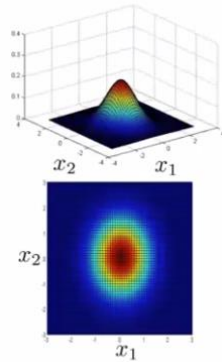
- Using the previous approach of anomaly detection with separate univariate Gaussian distributions, we assume  $x_1$  and  $x_2$  are independent, and hence the green example would be normal since there are more extreme  $x_1$  and  $x_2$  values.
- But this should be classified as anomaly because  $x_1$  and  $x_2$  are correlated.
- The multivariate gaussian distribution fixes this.
  - Don't model  $p(x_1), p(x_2), \dots, p(x_n)$  separately. Model  $p(x)$  all in one go.

- Parameters:  $\mu \in \mathbb{R}^n, \Sigma \in \mathbb{R}^{n \times n}$  (covariance matrix).
- $p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)\right)$ , where  $|\Sigma|$  is the determinant of the  $\Sigma$  matrix,  $T$  is transpose, and  $\Sigma^{-1}$  is the inverse of the  $\Sigma$  matrix.

— If  $\Sigma$  is a diagonal matrix, then the variables are independent.

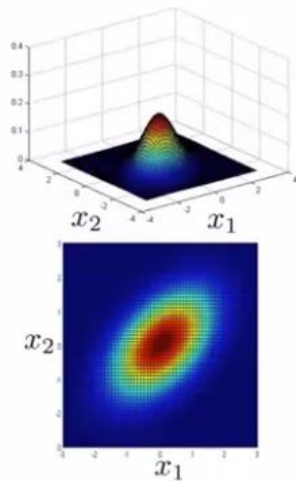
- If the variables are independent, the variance of  $x_i$  is equal to  $\Sigma_{ii}$
- The contour diagram of the Gaussian distribution would be a set of concentric ellipses, where each axis of the variable runs parallel to the graph axes. Example:

$$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 0.6 & 0 \\ 0 & 1 \end{bmatrix}$$

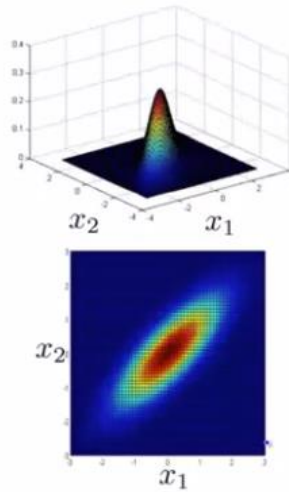


— There is correlation between two variables if  $\Sigma$  is not a diagonal matrix. Example:

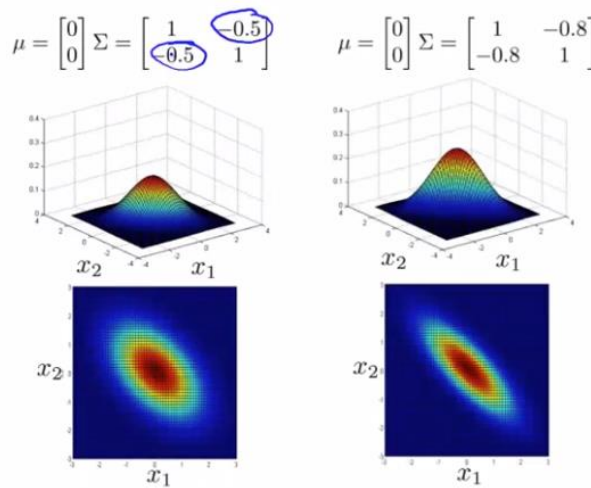
$$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$$



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



- The ellipsoid would be going from top-left to bottom-right if the off-diagonal entry in  $\Sigma$  were negative.



— Calculating  $\mu$  and  $\Sigma$  for a multivariate Gaussian distribution:

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

— Anomaly detection with multivariate Gaussian algorithm:

Fit model  $p(x)$  by setting

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

Given a new example  $x$ , compute:

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

Flag an anomaly if  $p(x) < \epsilon$ .

— The multiple univariate Gaussian distribution model (where  $p(x) = \prod_{j=1}^n p(x_j; \mu_j, \sigma_j^2)$ ) corresponds to the multivariate Gaussian distribution model under the constraint that the axis of the contours of  $p(x; \mu, \Sigma)$  are axis aligned.

- Mathematically speaking, the constraint is where  $\Sigma$  is a diagonal matrix with  $\Sigma =$

$$\begin{bmatrix} \sigma_1^2 & & 0 \\ & \sigma_2^2 & \\ 0 & & \dots \\ & & & \sigma_3^2 \end{bmatrix}$$

— When to use univariate Gaussian model vs. multivariate Gaussian model. Univariate Gaussian model generally used more often.

- Use univariate Gaussian model to manually create features to capture anomalies where  $x_1, x_2$  take unusual combinations of values. Computationally cheaper, scaling to large  $n$  (many features). Okay even if  $m$  (training set size) is small.
- Use multivariate Gaussian model to automatically capture correlations between features. Computationally more expensive, since need to compute  $\Sigma^{-1}$ . Must have  $m > n$  (and non-linearly dependent features), or else  $\Sigma$  is non-invertible. Possibly use multivariate Gaussian model if  $m \geq 10n$ .

## Recommender Systems

— Motivation:

- Many companies like Google, Facebook, Netflix generate lots of profit from recommender systems.
- Recommender systems are a possible setting where we can have the machine learning algorithm automatically learn to pick features instead of having us to manually pick them.

### Predicting Movie Ratings

— Assume each user can rate each movie using zero to five stars.

— Notation:

- $n_u$  = number of users
- $n_m$  = number of movies
- $m^{(j)}$  = number of movies rated by user  $j$
- $r(i, j) = 1$  if user  $j$  has rated movie  $i$
- $y^{(i, j)}$  = rating given by user  $j$  to movie  $i$  (defined only if  $r(i, j) = 1$ ).

— Problem formulation: given a table between users and movies (where each cell represents a rating, which can be empty if a user did not rate a particular movie), create a learning algorithm that predicts what rating a user would give for a movie they have not yet watched.

— “Content based recommendations”: one approach for building a recommender system. This assumes we have features for the different movies.

- Assume we have a set of features for each movie, such as  $x_1 \in [0,1]$  being the degree to which a movie is a romance movie and  $x_2 \in [0,1]$  being the degree to which a movie is an action film.
- This means each movie can be represented as a feature vector:  $x^{(i)} \in \mathbb{R}^{n+1} = \begin{bmatrix} x_0^{(i)} \\ \dots \\ x_n^{(i)} \end{bmatrix}$ , where  $x_0^{(i)} = 1$  for all  $i$ .

— For each user  $j$ , learn a parameter  $\theta^{(j)} \in \mathbb{R}^{n+1}$ . Predict user  $j$  as a rating movie  $i$  with  $(\theta^{(j)})^T x^{(i)}$  stars.

- For example, if the movie “Romance Forever” is movie 2, has feature  $x_1 = 0.9$  (degree of romantic film) and  $x_2 = 0.01$  (degree of action film), then  $x^{(2)} = \begin{bmatrix} 1 \\ 0.9 \\ 0.01 \end{bmatrix}$ . We want to predict Carol’s (user 3) rating of “Romance Forever”. Assume

we learn user 3’s parameter  $\theta^{(3)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}$ . Then we predict.  $(\theta^{(3)})^T x^{(2)} = 5$ , so that means we expect user 3 to give a 5 star rating to movie 2.

- To learn  $\theta^{(j)}$ , we want  $\min_{\theta^{(j)}} \frac{1}{2} \sum_{i:r(i,j)=1} \left( (\theta^{(j)})^T (x^{(i)}) - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{k=1}^n \left( \theta_k^{(j)} \right)^2$ .
  - Note the summation is across all values of  $i$  where  $r(i, j) = 1$  (i.e. we sum across all movies where user  $j$  has rated movie that movie).
  - This is basically linear regression: we try to minimize the sum of square difference between predicted rating and actual rating.
  - In normal linear regression, we would divide this entire expression by  $\frac{1}{2m^{(j)}}$  instead of  $\frac{1}{2}$ , but multiplying by a constant doesn't affect the outcome and is simpler here.

- Of course, we want to learn all of our users, so we want to learn  $\theta^{(1)}, \dots, \theta^{(n_u)}$ :

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

- Gradient descent update:

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

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

- We have separate gradient descent update rules since we don't apply the regularization term to the bias term (i.e. when  $k = 0$ ).
- $\frac{\partial}{\partial \theta_k^{(j)}} J(\theta^{(1)}, \dots, \theta^{(n_u)}) = \left( \sum_{i:r(i,j)=1} \left( (\theta^{(j)})^T (x^{(i)}) - y^{(i,j)} \right) x_k^{(i)} + \lambda \theta_k^{(j)} \right)$

## Collaborative Filtering

- Collaborative Filtering is another approach for recommender systems, except we do not need access to features for the movies.

- The algorithm does feature learning, which means it learns what features to use.

- Problem motivation

- Assume that we have user preferences, that means we have  $\theta^{(1)}, \dots, \theta^{(n_u)}$ . For

example, if  $\theta^{(2)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}$ , that means they might love romance films if  $\theta_1$

corresponds with romance films, but hate action films if  $\theta_2$  corresponds to action films. ( $\theta_0$  corresponds to the intercept term.)

- Our first goal is to learn feature vectors  $x^{(1)}, \dots, x^{(n_m)}$  such that  $(\theta^{(i)})^T x^{(j)} = y^{(i,j)}$  if  $r(i, j) = 1$

- Optimization algorithm to learn one movie's feature vector: given  $\theta^{(1)}, \dots, \theta^{(n_u)}$ , to learn  $x^{(i)}$ :

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

- We want to learn all the movies' feature vector, so given  $\theta^{(1)}, \dots, \theta^{(n_u)}$ , we want to learn  $x^{(1)}, \dots, x^{(n_m)}$ :

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

— Gradient descent update:

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

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

— To summarize, we can:

- Estimate  $\theta^{(1)}, \dots, \theta^{(n_u)}$  given  $x^{(1)}, \dots, x^{(n_m)}$  (and movie ratings) using the approach of content-based recommendations.
- Estimate  $x^{(1)}, \dots, x^{(n_m)}$  given  $\theta^{(1)}, \dots, \theta^{(n_u)}$  (and moving ratings) using the approach presented above.
- This lends itself to a chicken-and-egg problem: do  $x^{(i)}$ 's or  $\theta^{(j)}$ 's come first?

— It turns out we can solve the chicken-and-egg problem by guess initial small random  $\theta$ 's. Use the initial  $\theta$  to get  $x$ , and use  $x$  to get better  $\theta$ , use the better  $\theta$  to get better  $x$ , use the better  $x$  to get even better  $\theta$ , and so on.

- We use small random values instead of initializing to all 0's to break symmetries, ensuring the algorithm learns features  $x^{(1)}, \dots, x^{(n_m)}$  that are different from each other.
- We can do this more efficiently by putting the two optimization functions together:

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

- This algorithm has the property that if you hold  $\theta^{(1)}, \dots, \theta^{(n_u)}$  constant, then you would be solving “content-based recommendations” problem. If you hold  $x^{(1)}, \dots, x^{(n_m)}$  constant, then you would be solving the above problem of learning feature vectors for each movie given user preferences.
- We note that when we are learning the features, we are dropping the intercept term ( $\theta_0$ ) and bias term ( $x_0$ ), so  $x \in \mathbb{R}^n$  and  $\theta \in \mathbb{R}^n$ . This is because we are learning all the features, so no need to hard-code a feature that is always 1 – the algorithm can choose such a feature for itself if it wants one.

— Collaborative filtering algorithm:

Initialize  $x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)}$  to small random values.

Minimize  $J(x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)})$  using gradient descent (or an advanced optimization function. If we're using gradient descent:

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

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

For user  $j$  (having parameters  $\theta^{(j)}$ ) and movie  $i$  (with learned features  $x^{(i)}$ ), predict a star rating of  $(\theta^{(j)})^T x^{(i)}$ .

- The term “collaborative filtering” refers to the fact there are multiple users effectively collaborating to get better moving ratings for everyone since for each user that rates a subset of the movies, the algorithm can better learn the movies’ feature vector.

### Low Rank Matrix Factorization

- Vectorizing our approach:

- Represent the table of movie ratings via a matrix  $Y$ , where  $Y_{ij} = y^{(i,j)}$

- Represent our movie features  $X = \begin{bmatrix} (x^{(1)})^T \\ \dots \\ (x^{(n_m)})^T \end{bmatrix}$  and user preferences  $\Theta =$

$$\begin{bmatrix} (\theta^{(1)})^T \\ \dots \\ (\theta^{(n_w)})^T \end{bmatrix}. \text{ The table of predicted ratings can be calculated as } X\Theta^T, \text{ where the } i\text{th}$$

row and  $j$ th column of this table is our prediction of  $y^{(i,j)}$ .

- The collaborative filtering algorithm is also referred to as low rank matrix factorization since  $X\Theta^T$  produces a low rank matrix.
- We can use the collaborative filtering algorithm to find related movies (or related products if applying this algorithm to a non-movie scenario):
  - For each product  $i$ , we learn a feature vector  $x^{(i)} \in \mathbb{R}^n$ . How to find products  $j$  related to product  $i$ ?
  - If  $\|x^{(i)} - x^{(j)}\|$  is small, then products  $i$  and  $j$  are similar. To find the  $k$  most similar products to  $i$ , find the  $k$  products  $j$  that have the smallest  $\|x^{(i)} - x^{(j)}\|$ .
- One problem with the current approach is that, for a user that has given no ratings, we will predict that they will give 0 stars to all movies. Assume user  $j$  is this user who hasn’t given any ratings.
  - This is because of the cost equation. In the first summation, there are no  $(i,j)$  for user  $j$  such that  $r(i,j) = 1$ , so that returns 0. The second summation doesn’t affect  $\theta^{(j)}$ . Thus, the third summation causes  $\theta^{(j)}$  to be a vector of 0’s. Performing  $(\theta^{(j)})^T x^{(i)}$  for user  $j$  and any movie  $i$  would result in 0, so we predict 0 stars for any movie rated by user  $j$ .



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

- Mean normalization can solve this problem.

— Mean normalization procedure:

- Compute a vector  $\mu = \begin{bmatrix} \mu_1 \\ \dots \\ \mu_{n_m} \end{bmatrix}$ , where  $\mu_i$  is the average rating of movie  $i$ . Set  $Y = Y - \mu$ . Each row of  $Y$  should now add up to 0, so each movie now has an average rating of 0.
- Now use  $Y$  as the dataset for collaborative filtering to learn  $\theta^{(j)}$  and  $x^{(i)}$ .
- We need to add back the mean to predict the rating. More specifically, for user  $j$ , on movie  $i$ , predict:  $(\theta^{(j)})^T (x^{(i)}) + \mu_i$ .
- With mean normalization, we predict  $\mu_i$  for movie  $i$  if the user has not rated any movies.

— We note feature scaling is not relevant here (only mean normalization is) because the ratings are all on the same scale (e.g. 0 to 5 stars).

## Large Scale Machine Learning

### Gradient Descent with Large Datasets

- Batch gradient descent can be very computationally expensive if the dataset is very large (i.e.  $m$  is very large), since it has to loop through all  $m$  examples to update each parameter. Example for linear regression:  $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$ 
  - First check if randomly picking a smaller subset would work just fine by plotting the learning curves.
  - Recall that if  $J_{cv}(\theta)$  is much larger than  $J_{train}(\theta)$ , then having more examples would help. If not, then making the model more complex could help make the large number of examples useful to prevent overfitting.
- Two main ideas for dealing with very big data sets: stochastic gradient descent and map reduce
- Assuming we're using squared error: define the cost of a single training example  $(x^{(i)}, y^{(i)})$  to be equal to:

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

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

— Stochastic gradient descent algorithm:

Randomly shuffle dataset

Repeat:

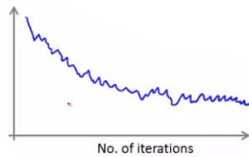
For  $i = 1, \dots, m$  (for each training example):

For  $j = 0, \dots, n$  (for each feature):

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} \text{cost}(\theta, (x^{(i)}, y^{(i)}))$$

- Ex, for linear regression,  $\frac{\partial}{\partial \theta_j} \text{cost}(\theta, (x^{(i)}, y^{(i)})) = (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$
  - It's generally good to randomly shuffle the dataset to ensure we visit the examples in an arbitrary order, helping to speed up convergence.
  - In batch gradient descent, we have to pass through all training examples before we can update  $\theta$  (i.e. uses all  $m$  examples in each iteration). In stochastic gradient descent, we update  $\theta$  after each training example (i.e. use 1 example in each iteration), and thus making progress faster.
- After each iteration (i.e. after each update of  $\theta$ ), stochastic gradient descent generally moves the parameters to the global optimum, but not always, thus taking a longer/windier path to the global optimum. (Batch gradient descent always moves the parameters closer to the global optimum after each iteration.)
- Stochastic gradient descent wanders around a region around the global optimum once it has converged.
- Stochastic gradient descent is faster than batch gradient descent because it updates the parameters  $m$  times after passing through  $m$  examples, whereas batch gradient descent only updates the parameters once after passing through  $m$  examples.
- Stochastic gradient descent may need only to run the outermost loop once if the examples are large enough. Otherwise, it needs to run it  $\sim 10x$  at most.
- Mini-batch gradient descent:
- While batch gradient descent uses all  $m$  examples in each iteration (i.e. for each update of  $\theta$ ) and stochastic gradient descent uses 1 example in each iteration, mini-batch gradient descent uses  $b$  examples in each iteration.
  - $b$  is referred to as the mini-batch size. Typical choices are  $b = 2$  to 100.
- Mini-batch gradient descent algorithm:
- Repeat:
- For  $i = 1, 1 + b, 1 + 2b, \dots, m - b + 1$  (for each batch of examples):
- For  $j = 0, \dots, n$  (for each feature):
- $$\theta_j := \theta_j - \alpha \frac{1}{b} \sum_{k=i}^{i+b-1} \text{cost}(\theta, (x^{(k)}, y^{(k)}))$$
- Ex, for linear regression,  $b = 10$ , and  $m = 1000$ :
- Repeat:
- For  $i = 1, 11, 21, 31, \dots, 991$ :
- For  $j = 0, \dots, n$ :
- $$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_{\theta}(x^{(k)}) - y^{(k)}) x_j^{(k)}$$
- Mini-batch gradient descent is faster than batch gradient descent if  $b$  is a lot smaller than  $m$ , since we update  $\theta$  after just looking at just  $b$  examples instead of  $m$ .
- It can be faster than stochastic gradient descent if you can vectorize the gradient update rule.
- Checking if stochastic gradient descent is converging:

- During learning, compute  $cost(\theta, (x^{(i)}, y^{(i)}))$  before updating  $\theta$  using  $(x^{(i)}, y^{(i)})$ . Every  $k$  iterations, plot  $cost(\theta, x^{(i)}, y^{(i)})$  averaged over the last  $k$  iterations processed by the algorithm. (Typically use  $k \approx 1000$ .)
  - The  $cost(\theta, x^{(i)}, y^{(i)})$  averaged over the last  $k$  iterations should typically go down as the number of iterations goes up.
- Using a smaller learning rate will cause the error to drop more slowly as number of iterations increases, but could help find a slightly better solution. This is because noisy oscillations are present in stochastic gradient descent, and a smaller learning rate will reduce the magnitude of those oscillations.



- Increasing  $k$  (the number of examples we are taking the average error of) will smoothen the curve and reduce oscillations, but will reduce the amount of data points in the plot.
  - Increasing  $k$  may make it easier to notice trends if there are too many oscillations for a smaller  $k$ .
  - If the average error increases as the number of iterations increases, then use a smaller learning rate.
- Typically in stochastic gradient descent,  $\alpha$  is held constant, which results in oscillations around global minimum once it converges. If you actually want stochastic gradient descent to converge to global minimum, then you can slowly decrease  $\alpha$ .
- Example:  $\alpha = \frac{c_1}{iterationNumber + c_2}$  for two constants  $c_1$  and  $c_2$ .
  - People often don't do this because of the need to select  $c_1$  and  $c_2$ .

## Online learning

- Online learning: an algorithm that learns from a continuous stream of data coming in
- Approach is similar to stochastic gradient descent: we update  $\theta$  immediately as an example (or small set of examples) come in. We discard the examples afterwards.
- Example scenario: a shipping service website where you offer to ship user's package for a price given origin and destination. Data comes in in real time of whether a user chose to use the shipping service ( $y = 1$ ) or not ( $y = 0$ ). Features  $x$  capture properties of user, of origin/destination, and asking price. We want to learn  $p(y = 1|x; \theta)$ .
- Possible techniques: logistic regression, neural networks, etc.
  - We use each example only once and in real time. Example for logistic regression: Whenever a new example  $(x, y)$  comes in:

For  $j = 0, \dots, n$  (for each feature):

$$// \frac{\partial}{\partial \theta_j} cost(\theta, (x, y)) = (h_\theta(x) - y) \cdot x_j$$

$$\theta_j := \theta_j - \alpha(h_\theta(x) - y) \cdot x_j$$

- We no longer use the notation  $x^{(i)}, y^{(i)}$  because we don't have a set of examples like before, so indexing isn't as necessary.
- If number of examples is small, consider to not use an online learning algorithm to reuse the examples multiple times.
- Online learning algorithm can adapt to changing user preferences.
- Example scenario: product search. User searches for product to buy. We return 10 results each time.  $x$  = features of product, how many words in user query matches the name of product, how many words in query match description of product, etc.  $y = 1$  if user clicks on link.  $y = 0$  otherwise. Goal of learning  $p(y = 1|x; \theta)$ .
  - Approach to solving problem: whenever a user searches on our site, we obtain 10 examples  $(x, y)$  to update  $\theta$  with: one for each of the 10 search results.
  - We then can use this data to return the 10 most likely products the user will buy each time they search.
- People who try to predict whether a user clicks on a link (such as the above 2 scenarios) are trying to solve the problem of finding the “predicted CTR”, where CTR = click-through rate.
- Other examples: choosing special offers to show users, customized selection of news articles, production recommendation, etc.

### Map Reduce and Data Parallelism

- We can multithread batch or mini-batch gradient descent by splitting up the training set or batch into equal sized groups: one group for each CPU.
- Each CPU computes  $\sum_i \frac{\partial}{\partial \theta_j} \text{cost}(\theta, (x^{(i)}, y^{(i)}))$  for group of examples it is responsible for. Afterwards, the results of all the CPUs are combined together by  $\theta_j := \theta_j - \alpha \frac{1}{m} \sum \text{cpu\_results}$
- Concrete example: assume 300 examples, 3 machines, linear regression, and batch gradient descent. We effectively want to perform  $\theta_j := \theta_j - \alpha \frac{1}{300} \sum_{i=1}^{300} (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$  for all features  $j = 0, \dots, n$ .
  - Let machine 1 be responsible for  $(x^{(1)}, y^{(1)}), \dots, (x^{(100)}, y^{(100)})$ . It computes  $\text{temp}_j^{(1)} = \sum_{i=1}^{100} (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$ .
  - Let machine 2 be responsible for  $(x^{(101)}, y^{(101)}), \dots, (x^{(200)}, y^{(200)})$ . It computes  $\text{temp}_j^{(2)} = \sum_{i=101}^{200} (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$ .
  - Let machine 3 be responsible for  $(x^{(201)}, y^{(201)}), \dots, (x^{(300)}, y^{(300)})$ . It computes  $\text{temp}_j^{(3)} = \sum_{i=201}^{300} (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$ .
  - Once all machines have calculated  $\text{temp}$  values, combine the results to perform  $\theta_j := \theta_j - \alpha \frac{1}{300} (\text{temp}_j^{(1)} + \text{temp}_j^{(2)} + \text{temp}_j^{(3)})$ .
  - Repeat above steps for all features  $j = 0, \dots, n$  and for each iteration of gradient descent.

- Whenever you are summing across the training examples to calculate  $J_{train}(\theta)$  and/or  $\frac{\partial}{\partial \theta_j} J_{train}(\theta)$ , map-reduce can be used to parallelize the bulk of the calculations. (Can be used for linear regression, logistic regression, neural networks, etc.)
- Map-reduce can be applicable on multi-core CPUs and GPUs, so you don't have to worry about network latency if you were parallelizing over multiple computers.
  - Some numerical linear algebra libraries automatically optimize over multiple threads, so vectorizing the implementation may be an alternative to map-reduce.
- Hadoop is a popular open source implementation of Map-reduce.

## Photo OCR

### Problem Description and Pipeline

- Photo OCR = photo optical character recognition.
  - An algorithm that detects where there is text in a picture and reads the text in those regions (outputting it as a string).
- OCR for documents is not too difficult, but for pictures is a harder problem.
- Photo OCR machine learning pipeline:
  - Detect regions of text in the photos
  - Segment the characters (identify the boundaries of each character) in each region.

Example:



- Recognize each character by classifying them. Example:
  - Optionally: perform error checking for misclassified letters. Example: “cleaning” is probably “cleaning”, where the letter l was misclassified as number 1.
- A machine learning pipeline is a system with multiple stages/components, several of which may use machine learning.
- Identifying regions of text is more difficult than identifying regions of individual pedestrians because the aspect ratio is variable for text (but pedestrians usually have similar aspect ratios).


### Sliding Windows


- Sliding window technique:
  - First, train a machine learning model using supervised learning on a set of labeled training examples, where each training example is a tiny picture snippet (e.g. 82 x 36 pixels). (If the image contains the object you're trying to identify, it's a positive example.)
  - Then, in an image, overlay a rectangle of the same snippet size (e.g. 82 x 36 pixels in previous example) on the top left of the image. This rectangle represents

the sliding window. Run the neural network on that snippet to see if the object is present there. Next, scan the running window like reading a book: moving to the right and then down, running the neural network after each movement. Example of sliding window detecting pedestrians:



- Repeat the above step of sliding the window for different window sizes to capture objects that appear larger in the image. Not to rescale the window to the size the neural network was trained on (e.g. rescale to 82 x 36).
  - The number of pixels you move the sliding window each time is the step size or stride parameter. Performs the best if the step size is 1, but is more computationally expensive.
- After performing the sliding window technique, you now have spots in the image where the classifier believes the object is in the image. This, in effect, can be represented as a heatmap of rectangles of positively predicted snippets.
- If the spots are small, you may want to expand the spots in the heatmap to join close-by regions of pixels belonging to a positive region. Mathematically, this can be achieved via the following algorithm: for each pixel in the image, if it is within  $k$  pixels ( $k \approx 5$  to 10) to a positive region, then label that pixel as part of a positive region.
  - Finally, draw bounding boxes around each region, using simple heuristics to filter out noise. (For example, text should be much longer than wider, so get rid of regions that are taller than wider. Additionally, get rid of tiny spots.)
- After identifying regions of text, we can use a 1D sliding window to perform character segmentation (identify the boundaries of each character)
- Train the ML model on positive examples of snippets showcasing the midpoint between two characters and negative examples on snippets that don't showcase the midpoint between two characters.

Positive example: 


Negative example: 

- Note that this is one-dimensional: the sliding window height should be the height of the image region:



- Character classification can be done via a standard ML supervised learning technique.

## Getting Lots of Data through Artificial Data

- There are two main ways to create artificial data to generate larger training sets: creating data from scratch and amplifying a small training set to a larger one.
  - The specific method used to create artificial data varies among applications, and thus requires some thought.
- Artificial data synthesis for photo OCR: goal of creating picture snippets of a character in them. Example:  is an “M”.
  - Create data from scratch: use various word fonts. For a given character in a given font, overlay it with a random background image. Then add various distortions and effects. Compare synthetic data with real data to make sure they appear similar. If successful, you essentially have an unlimited supply of examples.
  - Use existing examples: create synthetic data by introducing distortions in the existing examples.
- Synthesizing data by introducing distortions for speech recognition (i.e. train a ML model to perform speech-to-text):
  - Given an original audio clip of high quality, you can create many more examples by adding various background noises to simulate audio on bad cellphone connection, audio with noisy crowd in background, audio with noisy machinery in background, etc.
- Distortion introduced should be representative of the type of noise/distortions in the test set. It usually does not help to add purely random/meaningless noise in your data.
- Other considerations of getting more data:
  - Make sure you have a low bias/high variance classifier before expending the effort. (Plot learning curves). If not, then make the model more complicated (e.g. add more hidden units/layers in neural network).
  - Consider how much work would it be to get 10x as much data as we currently have. If it's not too hard to collect/label data yourself, it may be better to do that instead of trying to create realistic artificial data. Another way to get more data is to use “crowd source” (e.g. Amazon Mechanical Turk): hire people on the web to label large training sets.

## Ceiling Analysis

- Ceiling analysis is a technique that can provide guidance on what parts of the ML pipeline might be the best use of your time to work on.
- Assume that your pipeline consists of several components:  $c_1, \dots, c_k$ . Further assume that your pipeline currently has an output score (e.g. accuracy) of  $p\%$ . For each component  $c_i$ , manually provide correct annotations on the test set for components  $c_1, \dots, c_i$  to simulate how well the pipeline performs if  $c_1, \dots, c_i$  were perfect, and observe how much  $p$  increases. The component that increases  $p$  the most is the one you may want to dedicate most resources to.

- Concrete example (photo OCR): recall the pipeline consists of text detection, character segmentation, and character recognition. Assume our current system has a score of 72%. Now, we replace the text detection component with labels that are 100% accurate in the test set and observe that the pipeline now has an 89% accuracy. Next, we repeat the procedure for character segmentation, replacing the output of text detection and character segmentation with 100% correct labels. Assume this increases to 90% overall accuracy. Next, we replace output of text detection, character segmentation, and character recognition with correct labels, observing 100% accuracy. Text detection saw 17% increase, character segmentation saw 1% increase, and character recognition saw 10% increase, indicating that working on text detection has the highest possible improvement in pipeline performance (followed by character recognition and then character segmentation).
- If the system has components in parallel, then order those components in arbitrary order. In other words, envisioning our system as a graph where components are nodes and edges connect components that feed their output as input, order all components in a valid topologically sorted order.