# Machine Learning

Derek Li

## Contents

# 1 Introduction

# 2 Octave

## 2.1 Plotting data

- Save the plot: **print -dpng 'filename'**
- Divide plot a $1 \times 2$ grid, access first element: **subplot(1, 2, 1)**
- Clears a figure: **clf**
- Visualizes the matrix $A$: **imagesc(A)**

## 2.2 For, while, if statements, and functions

### 2.2.1 For statements

```
for condition,
   execution;
end;
```

### 2.2.2 While statements

```
while condition,
   execution;
end;
```

### 2.2.3 If statements

```
if condition,
   execution;
elseif condition,
   execution;
else
   execution;
end;
```

### 2.2.4 Functions

```
function (output(s)) = FunctionName(variable(s))
   execution;
```

## 2.3 Vectorization

**Example 2.1** $(h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j = \theta^T \mathbf{x})$**.**

```
   prediction = theta' * x;
```

# 3 Univariate Linear Regression

## 3.1 Model representation

The hypothesis function is

$$h_\theta(x) = \theta_0 + \theta_1 x,$$

where $\theta_i$'s are parameters.

## 3.2 Cost function

For univariate linear regression, we define a cost function

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

where $m$ is the size of the training set. Our objective is to solve

$$\min_{\theta_0, \theta_1} J(\theta_0, \theta_1).$$

The cost function $J(\theta_0, \theta_1)$ is also called squared error cost function.

## 3.3 Gradient descent algorithm

**Algorithm 3.1** (Gradient Descent Algorithm)**.** Repeat

$$\left\{ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \right\}$$

until convergence.

**Note 1.** $\alpha$ is called the learning rate, which controls the velocity of gradient descent. If $\alpha$ is too small, gradient descent can be slow; if $\alpha$ is too large, it may fail to converge or even diverge.

**Note 2.** The algorithm requires to update $\theta_j$'s simultaneously.

**Note 3.** Generally, depending on where it is initializing, it can end up with different local optima.

## 3.4 Gradient descent for linear regression

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

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

**Algorithm 3.2** (Gradient Descent Algorithm for Linear Regression)**.** Repeat

$$\left\{ \begin{array}{l} \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \\ \\ \theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} \end{array} \right\}$$

until convergence.

**Note 1.** Cost function for linear regression is convex and thus, it does not have any local optima except for one global optimum. Hence, gradient descent always makes it converge to the global optimum.

**Note 2.** The algorithm is also called Batch Gradient Descent, and "Batch" means each step of gradient descent uses all the training examples.

# 4 Multivariate Linear Regression

## 4.1 Model representation

The hypothesis function is

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

where $x_0 = 1$.

## 4.2 Gradient descent for multivariate linear regression

**Algorithm 4.1** (Gradient Descent Algorithm for Multivariate Linear Regression)**.** Repeat

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

until convergence.

### 4.2.1 Feature scaling

Gradient descent can converge more quickly if the features are on a similar scale. Hence, we want every feature into approximately a $-1 \leqslant x_i \leqslant 1$ range.

**Method 1.** Dividing by the maximum value.
**Method 2.** (Mean normalization)

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

### 4.2.2 Learning rate

For sufficiently small $\alpha$, $J(\theta)$ decreases on every iteration. To choose $\alpha$, try

$$..., 0.001, 0.03, 0.01, 0.03, 0.1, 0.3, 1, ...$$

and plot $j(\theta)$ as a function of the number of iterations. Try to pick the largest possible value or just slightly smaller than the largest reasonable value.

## 4.3 Normal equation

Normal equation is a method to solve for $\theta$ analytically, compared to gradient descent.

Suppose there are $m$ training examples and $n$ features, then

$$\mathbf{x}^{(i)} = \begin{pmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \vdots \\ x_n^{(i)} \end{pmatrix} \in \mathbb{R}^{n+1}.$$

Then the design matrix is

$$X = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(m)})^T \end{pmatrix}.$$

We have

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

and such $\theta$ minimize the $J(\theta)$. The code in Octave is:

```
pinv(X' * X) * X' * y
```

Even $X^T X$ is not invertible (singular/degenerate), **pinv** can still compute $X^T X$. But in this case, we can delete redundant features (linearly dependent). If there are too many features ($m \geqslant n$), then delete some features or use regularization.

Comparison between gradient descent and normal equation:

| Gradient Descent | Normal Equation |
|---|---|
| Needs to choose $\alpha$ | Does not need to choose $\alpha$ |
| Needs many iterations | Does not need to iterate |
| Works well even when $n$ is large | Need to compute $(X^T X)^{-1}, O(n^3)$ |
| | Slow if $n$ is very large |

# 5 Logistic Regression

## 5.1 Hypothesis representation

The hypothesis function is

$$h_\theta(\mathbf{x}) = g(\theta^T \mathbf{x}),$$

where

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

and $g(z)$ is called sigmoid/logistic function. Therefore,

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

We have $h_\theta(\mathbf{x}) = P(y=1|\mathbf{x};\theta)$.

## 5.2 Decision boundary

Suppose $h_\theta(\mathbf{x}) \geqslant 0.5 \Rightarrow y = 1, h_\theta(\mathbf{x}) < 0.5 \Rightarrow y = 0$, then

$$\theta^T \mathbf{x} \geqslant 0 \Rightarrow y = 1, \theta^T \mathbf{x} < 0 \Rightarrow y = 0.$$

The decision boundary is the line that separates the area where $y = 0$ and $y = 1$.

## 5.3 Cost function

For logistic regression, we define

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

then the cost function is

$$\begin{aligned} J(\theta) &= \frac{1}{m}\sum_{i=1}^{m} \text{Cost}(h_\theta(\mathbf{x}^{(i)}), y^{(i)}) \\ &= -\frac{1}{m}\sum_{i=1}^{m} [y^{(i)} \ln(h_\theta(\mathbf{x}^{(i)})) + (1-y^{(i)})\ln(1-h_\theta(\mathbf{x}^{(i)}))]. \end{aligned}$$

## 5.4 Gradient descent for logistic regression

**Algorithm 5.1** (Gradient Descent Algorithm for Logistic Regression)**.** Repeat

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

until convergence.

**Note.** A vectorized implementation is $\theta := \theta - \frac{\alpha}{m} X^T (g(X\theta) - \mathbf{y})$.

## 5.5 Advanced optimization

In addition to gradient descent, there are some optimization algorithms: conjugate gradient, BFGS, L-BFGS. They do not need to manually pick $\alpha$ and are often faster than gradient descent, but they are more complex. In Octave,

```
function [jVal, gradient] = costFunction(theta)
   jVal = ____; % Code to compute J(theta)
   gradient = [____]; % Code to compute paritial derivative of
      J(theta) w.r.t. theta_0, ..., theta_n
end
options = optimset('GradObj', 'on', 'MaxIter', 100);
initialTheta = zeros(2, 1);
[optTheta, functionVal, exitFlag] = fminunc(@costFunction,
    initialTheta, options);
```

## 5.6 Multi-class classification

Train a logistic regression classifier $h_\theta^{(i)}(\mathbf{x})$ for each class $i$ to predict the probability that $y = i$. On a new input $\mathbf{x}$, to make a prediction, pick the class $i$ that maximizes $h_\theta^{(i)}(\mathbf{x})$.

# 6 Regularization

## 6.1 Over-fitting

If we have too many features, the learned hypothesis may fit the training set very well, $J(\theta) \approx 0$, but fail to generalize to new examples.

### 6.1.1 Addressing over-fitting

**Method 1.** Reduce number of features

(1) Manually select which features to keep.

(2) Model selection algorithm.

**Method 2.** Regularization

(1) Keep all the features, but reduce magnitude/values of parameters $\theta_j$.

(2) Works well when we have a lot of features, each of which contributes a bit to predicting $y$.

## 6.2 Cost function

The cost function is

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

where $\lambda\sum_{j=1}^{n}\theta_j^2$ is the regularization term and $\lambda$ is the regularization parameter.

## 6.3 Regularized linear regression

### 6.3.1 Gradient descent

**Algorithm 6.1** (Gradient Descent Algorithm for Regularized Linear Regression)**.** Repeat

$$\left\{\begin{array}{l}\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(\mathbf{x}^{(i)}) - y^{(i)})x_0^{(i)} \\ \theta_j := \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(\mathbf{x}^{(i)}) - y^{(i)})x_j^{(i)}, j = 1, ..., n\end{array}\right\}$$

until convergence.

### 6.3.2 Normal equation

The $\theta$ minimizes $J(\theta)$ is

$$\theta = \left( X^T X + \lambda \begin{pmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix} \right)^{-1} X^T \mathbf{y},$$

where $\begin{pmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix}$ is a $(n+1) \times (n+1)$ matrix.

## 6.4 Regularized logistic regression

### 6.4.1 Cost function

The cost function is

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

### 6.4.2 Gradient descent

**Algorithm 6.2** (Gradient Descent Algorithm for Regularized Logistic Regression)**.** Repeat

$$\left\{ \begin{array}{l} \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(\mathbf{x}^{(i)}) - y^{(i)}) x_0^{(i)} \\ \theta_j := \theta_j (1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}, j = 1, ..., n \end{array} \right\}$$

until convergence.

### 6.4.3 Advanced optimization

In Octave,

```
function [jVal, gradient] = costFunction(theta)
jVal = ____; % Code to compute J(theta)
gradient = [____]; % Code to compute paritial derivative of
    J(theta) w.r.t. theta_0, ..., theta_n
end
options = optimset('GradObj', 'on', 'MaxIter', 100);
initialTheta = zeros(2, 1);
[optTheta, functionVal, exitFlag] = fminunc(@costFunction,
    initialTheta, options);
```

# 7 Neural Network

## 7.1 Model representation

In neural network, the first layer is called the input layer and the final layer is called the output layer.

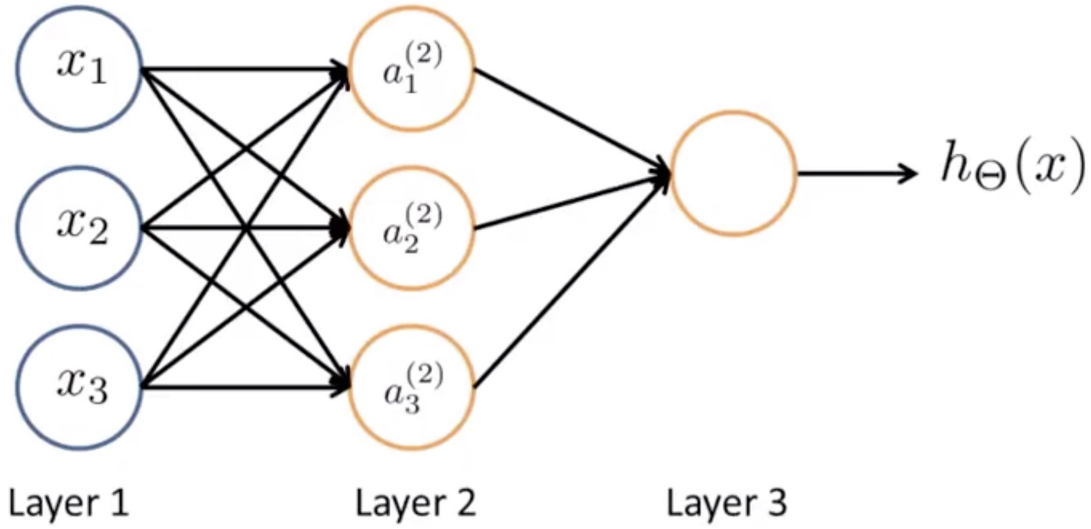


Figure 7.1: Model Representation for Neural Network

$a_i^{(j)}$ is the activation of unit $i$ in layer $j$, and $\Theta^{(j)}$ is the matrix of weights controlling function mapping from layer $j$ to layer $j+1$. For this figure, we have

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

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

For this figure, a vectorized implementation is

$$\mathbf{z}^{(2)} = \Theta^{(1)}\mathbf{x} = \Theta^{(1)}\mathbf{a}^{(1)},$$
$$\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)}),$$

where $\mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}, \mathbf{z}^{(2)} = \begin{pmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{pmatrix}$.

Add $a_0^{(2)} = 1$, then

$$z^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)},$$
$$h_\Theta(\mathbf{x}) = a^{(3)} = g(z^{(3)}).$$

The process is called forward propagation.

In general,

$$\begin{aligned}
\mathbf{a}^{(1)} &= \mathbf{x} \\
\mathbf{z}^{(2)} &= \Theta^{(1)}\mathbf{a}^{(1)} \\
\mathbf{a}^{(2)} &= g(\mathbf{z}^{(2)}) \ \ (\text{add } a_0^{(2)}) \\
\mathbf{z}^{(3)} &= \Theta^{(2)}\mathbf{a}^{(2)} \\
\mathbf{a}^{(3)} &= g(\mathbf{z}^{(3)}) \ \ (\text{add } a_0^{(3)}) \\
&\vdots \\
\mathbf{z}^{(L-1)} &= \Theta^{(L-2)}\mathbf{a}^{(L-2)} \\
\mathbf{a}^{(L-1)} &= g(\mathbf{z}^{(L-1)}) \ \ (\text{add } a_0^{(L-1)}) \\
\mathbf{z}^{(L)} &= \Theta^{(L-1)}\mathbf{a}^{(L-1)} \\
\mathbf{a}^{(L)} &= g(\mathbf{z}^{(L)}) = h_\Theta(\mathbf{x})
\end{aligned}$$

## 7.2 Cost function

The cost function is

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K}\mathbf{y}_k^{(i)}\ln(h_\Theta(\mathbf{x}^{(i)}))_k + (\mathbf{1}-\mathbf{y}_k^{(i)})\ln(1-(h_\Theta(\mathbf{x}^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\Theta_{ji}^{(l)})^2,$$

where $K$ is the number of putput units, $L$ is the total number of layers in network, and $s_l$ is the number of units (not counting bias unit) in layer $l$.

## 7.3 Back-Propagation algorithm

**Algorithm 7.1** (Back-Propagation Algorithm)**.** Suppose training set is $\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), ..., (\mathbf{x}^{(m)}, \mathbf{y}^{(m)})\}$ and set $\Delta_{ij}^{(l)} = 0$.
For $i = 1$ to $m$ :
Set $\mathbf{a}^{(1)} = \mathbf{x}^{(i)}$.
Perform forward propagation to compute $\mathbf{a}^{(l)}$ for $l = 2, ..., L$.
Use $\mathbf{y}^{(i)}$ to compute $\delta^{(L)} = \mathbf{a}^{(L)} - \mathbf{y}^{(i)}$.
Compute $\delta^{(L-1)}, ..., \delta^{(2)}$, where $\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} .* g'(\mathbf{z}^{(l)})$, and $g'(\mathbf{z}^{(l)}) = \mathbf{a}^{(l)} .* (1 - \mathbf{a}^{(l)})$.
$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$. In vector form, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (\mathbf{a}^{(l)})^T$.
$\frac{\partial}{\partial \Theta_{ij}^{(l)}} D_{ij}^{(l)} = D_{ij}^{(l)} := \begin{cases} \frac{1}{m} \Delta_{ij}^{(l)} & \text{if } j = 0 \\ \frac{1}{m} (\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}) & \text{if } j \neq 0 \end{cases}$.

## 7.4 Unrolling parameters

**Example 7.1.** Suppose $s_1 = 10, s_2 = 10, s_3 = 1$, then we have

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

Hence, in Octave,

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

We can get back the original matrices from the unrolled versions:

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

To summarize, we have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ and unroll to get initialTheta to pass fminunc(@costFunction, initialTheta, options).

For the cost function, we get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ from thetaVec and use FP/BP to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$. Then unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get gradientVec.

## 7.5 Gradient checking

For unrolled $\theta = (\theta_1, ..., \theta_n)$, we have

$$\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1 + \varepsilon, \theta_2, ..., \theta_n) - J(\theta - \varepsilon, \theta_2, ..., \theta_n)}{2\varepsilon},$$
$$\vdots$$
$$\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, ..., \theta_n + \varepsilon) - J(\theta, \theta_2, ..., \theta_n - \varepsilon)}{2\varepsilon}.$$

In Octave,

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

Check if gradApprox $\approx$ DVec.

To summarize:

- Implement BP to compute DVec (unrolled).
- Implement numerical gradient check to compute gradApprox.
- Make sure they give similar values.
- Turn off gradient checking. Using BP code for learning.

## 7.6 Random initialization

Initializing all theta weights to zero does not work with neural networks. Instead we can randomly initialize weights for $\Theta$ in Octave:

```
Theta_num = rand(a, b) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

## 7.7 Implementation of neural network

**Step 1.** Pick a network architecture (connectivity pattern between neurons).

**Note 1.** Number of input units is the dimension of features $\mathbf{x}^{(i)}$.

**Note 2.** Number of output units is the number of classes.

**Note 3.** The reasonable default for hidden layer is 1, or if the number is greater than 1, the hidden layers should have same number of hidden units in every layer. Usually, the more units the better.

**Step 2.** Train a neural network.

**Step 2.1** Randomly initialize weights.

**Step 2.2** Implement forward propagation to get $h_\Theta(\mathbf{x}^{(i)})$ for any $\mathbf{x}^{(i)}$.

**Step 2.3** Implement code to compute cost function $J(\Theta)$.

**Step 2.4** Implement backward propagation to compute $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$.

**Step 2.5** Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using BP with using numerical estimate of gradient of $J(\Theta)$, then disable gradient checking code.

**Step 2.6** Use gradient descent of advanced optimization method with BP to try to minimize $J(\Theta)$ as a function of parameters $\Theta$.

# 8 Evaluation of Learning Algorithm

## 8.1 Evaluation of hypothesis

### 8.1.1 Training procedure for linear regression

- Learn parameter $\theta$ from training data (minimizing training error $J(\theta)$ with 70% of all data).
- Compute test set error:

$$J_{\text{test}}(\theta) = \frac{1}{2m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} (h_\theta(\mathbf{x}_{\text{test}}^{(i)}) - \mathbf{y}_{\text{test}}^{(i)})^2.$$

### 8.1.2 Training procedure for logistic regression

- Learn parameter $\theta$ from training data.
- Compute test set error:

$$J_{\text{test}}(\theta) = -\frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} [\mathbf{y}_{\text{test}}^{(i)} \ln h_\theta(\mathbf{x}_{\text{test}}^{(i)}) + (\mathbf{1} - \mathbf{y}_{\text{test}}^{(i)}) \ln(1 - h_\theta(x_{\text{test}}^{(i)}))].$$

- Or compute misclassification error(0/1 misclassification error): Define

$$\text{Err}(h_\theta(\mathbf{x}), \mathbf{y}) = \begin{cases} 1 & \text{if } h_\theta(\mathbf{x}) \geqslant 0.5, y = 0 \text{ or if } h_\theta(\mathbf{x}) < 0.5, y = 1 \\ 0 & \text{otherwise} \end{cases}.$$

The test error is

$$\frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \text{Err}(h_\theta(\mathbf{x}_{\text{test}}^{(i)}), \mathbf{y}_{\text{test}}^{(i)}).$$

## 8.2 Model selection and training/validation/test sets

- Break down the dataset into three sets: Training set(60%), cross validation set(20%), and test set(20%).
- Optimize the parameters in $\Theta$ using the training set for each polynomial degree.
- Find the polynomial degree $d$ with the least error using the cross validation set.
- Estimate the generalization error using the test set with $J_{\text{test}}(\Theta^{(d)})$.

## 8.3 Diagnosing Bias and Variance

- Bias(under-fit): $J_{\text{train}}(\theta)$ will be high and $J_{\text{cv}}(\theta) \approx J_{\text{train}}(\theta)$.
- Variance(over-fit): $J_{\text{train}}(\theta)$ will be high and $J_{\text{cv}}(\theta) \gg J_{\text{train}}(\theta)$.

### 8.3.1 Regularization

- Create a list of $\lambda$, i.e.,

$$\lambda \in \{0, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 1.28, 2.56, 5.12, 10.24\}.$$

- Create a set of models with different degrees or any other variants.
- Iterate through the $\lambda$'s and for each $\lambda$ go through all the models to learn some $\Theta$.
- Compute the cross validation error using the learned $\Theta$ computed with $\lambda$ on the $J_{\text{cv}}(\Theta)$ without regularization.
- Select the best combo that produces the lowest error on the cross validation set.
- Using the beset combo $\Theta$ and $\lambda$, apply it on $J_{\text{test}}(\Theta)$ to see if it has a good generalization of the problem.

## 8.4 Learning curves

### 8.4.1 High bias

- Low training set size causes $J_{\text{train}}(\Theta)$ to be low and $J_{\text{CV}}(\Theta)$ to be high.
- Large training set size causes both $J_{\text{train}}(\Theta)$ and $J_{\text{CV}}(\Theta)$ to be high with $J_{\text{train}}(\Theta) \approx J_{\text{CV}}(\Theta)$.
- If a learning algorithm is suffering from high bias, getting more training data will not help much.

### 8.4.2 High variance

- Low training set size causes $J_{\text{train}}$ to be low and $J_{\text{CV}}(\Theta)$ to be high.
- $J_{\text{train}}(\Theta)$ increases with training set size and $J_{\text{CV}}(\Theta)$ continues to decrease without leveling off. Also $J_{\text{train}}(\Theta) < J_{\text{CV}}(\Theta)$ but the difference between them remains significant.
- If a learning algorithm is suffering from high variance, getting more training data is likely to help.

## 8.5 Debugging a learning algorithm

- Get more training examples - Fixes high variance.
- Try smaller sets of features - Fixes high variance.
- Try getting addition features - Fixes high bias.
- Try adding polynomial features - Fixes high bias.
- Try decreasing $\lambda$ - Fixes high bias.
- Try increasing $\lambda$ - Fixes high variance.

# 9 Machine Learning System Design

## 9.1 Error analysis

The recommended approach to solving machine learning problems is to:

- Start with a simple algorithm, implement it quickly, and test it on cross validation data.
- Plot learning curves to decide if more data, more features, etc. are likely to help.
- Error analysis: Manually examine the errors on examples in the cross validation set and try to spot a trend where most of the errors were made.

## 9.2 Error metrics for skewed classes

We use precision/recall to measure the error for skewed classes.

- Precision $= \frac{\text{True positive}}{\text{Predicted positive}} = \frac{\text{True positive}}{\text{True positive+False positive}}$.
- Recall $= \frac{\text{True positive}}{\text{Actual positive}} = \frac{\text{True positive}}{\text{True positive+False negative}}$.

One way to trade off between precision and recall is using $F$ score:

$$F = 2\frac{PR}{P+R}.$$

## 9.3 Data for machine learning

Large data rationale: Assume feature $\mathbf{x} \in \mathbf{R}^{n+1}$ has sufficient information to predict $y$ accurately. Using a learning algorithm with many parameters causes low bias ($J_{\text{train}}(\theta)$ will be small). Using a very large training set (unlikely to over-fit) causes low variance ($J_{\text{train}}(\theta) \approx J_{\text{test}}(\theta) \Rightarrow J_{\text{test}}(\theta)$ will be small).

# 10 Support Vector Machines

## 10.1 Optimization objective

We want to solve

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

Unlike logistic regression, the support vector machine does not output the probability but makes a prediction of $y$.

## 10.2 Kernels

A Gaussian kernel function is

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

### 10.2.1 Kernels and similarity

For example, take the first landmark

$$f_1 = \text{similarity}(\mathbf{x}, \mathbf{l}^{(1)}) = \exp\left( -\frac{||\mathbf{x} - \mathbf{l}^{(1)}||}{2\sigma^2} \right) = \exp\left( -\frac{\sum_{j=1}^{n} (x_j - l_j^{(1)})^2}{2\sigma^2} \right).$$

If $\mathbf{x} \approx \mathbf{l}^{(1)}$, $f_1 \approx 1$. If $\mathbf{x}$ is far from $\mathbf{l}^{(1)}$, $f_1 \approx 0$.

### 10.2.2 SVM with kernels

Given $(\mathbf{x}^{(1)}, y^{(1)}), ..., (\mathbf{x}^{(m)}, y^{(m)})$, choose $\mathbf{l}^{(1)} = \mathbf{x}^{(1)}, ..., \mathbf{l}^{(m)} = \mathbf{x}^{(m)}$. Compute $\mathbf{f}^{(i)} = \begin{pmatrix} f_0^{(i)} \\ \vdots \\ f_m^{(i)} \end{pmatrix}$, where $f_0^{(i)} = 1$, and solve

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

where $n = m$.

### 10.2.3 SVM parameters

- Large $C$ causes lower bias and higher variance.
- Small $C$ causes higher bias and lower variance.
- Large $\sigma^2$ causes features $f_i$ to vary more smoothly, higher bias and lower variance.
- Small $\sigma^2$ causes features $f_i$ to vary less smoothly, lower bias and higher variance.

## 10.3 SVM in practice

### 10.3.1 Gaussian kernel

If we use Gaussian kernel, we should perform feature scaling before using the Gaussian kernel.

### 10.3.2 Other choices of kernel

Not all similarity functions make valid kernels, and it need to satisfy Mercer's Theorem to make sure SVM packages' optimization run correctly and do not diverge. Here is some kernels available:

- Polynomial kernel
- String kernel
- chi-square kernel
- Histogram intersection kernel, etc.

### 10.3.3 Multi-class classification

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

### 10.3.4 Logistic regression vs. SVM

- If $n$ is large (relative to $m$,) then we use logistic regression, or SVM without a kernel (linear kernel).
- If $n$ is small, $m$ is intermediate, then we use SVM with Gaussian kernel.
- If $n$ is small, $m$ is large, then we add more features, then use logistic regression.
- Neural network is likely to work well for most of these settings but may be slower to train.

# 11 Clustering

## 11.1 $K$-Means algorithm

**Algorithm 11.1** ($K$-Means Algorithm)**.** Take number of clusters $K$ and training set $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ as inputs, where $\mathbf{x}^{(i)} \in \mathbb{R}^n$. Randomly initialize $K$ cluster centroids $\mu_1, ..., \mu_K \in \mathbb{R}^n$. Repeat

$$\left\{\begin{array}{l}\text{for } i = 1 \text{ to } m, c^{(i)} := \text{index (from 1 to } K\text{) of cluster centroid closest to } x^{(i)} \\ \text{for } k = 1 \text{ to } K, \mu_k := \text{average (mean) of points assigned to cluster } k\end{array}\right\}$$

until convergence.

## 11.2 Optimization objective

Let $c^{(i)}$ be the index of cluster $(1, ..., K)$ to which example $\mathbf{x}^{(i)}$ is currently assigned, $\mu_k$ be the cluster centroid $k$, and $\mu_{c^{(i)}}$ be the cluster centroid of cluster to which example $\mathbf{x}^{(i)}$ has been assigned.

The cost function is

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

Then we solve

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

## 11.3 Random initialization

We can randomly initialize $K$-means and run, then get $c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K$ and compute cost function (distortion). We can take random initialization many times and pick clustering that gives lowest cost.

# 12 Principal Component Analysis

## 12.1 Principal component analysis algorithm

Before principal component analysis, we should do data preprocessing. Given training set $x^{(1)}, ..., x^{(m)}$, let $\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$. Replace each $x_j^{(i)}$ with $x_j - \mu_j$. If different features on different scales, scale features to have comparable range of values.

**Algorithm 12.1** (Principal Component Analysis Algorithm)**.** Reduce data from $n$ dimensions to $k$ dimensions: Compute covariance matrix:

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

Compute eigenvectors of matrix $\Sigma$. In Octave:

```
Sigma = (1 / m) * X' * X;
[U, S, V] = svd(Sigma);
```

We get $U \in \mathbb{R}^{n\times n}$, and choose the first $k$ columns to get a $n \times k$ matrix $U_{\text{reduce}}$, then $\mathbf{z} = U_{\text{reduce}}^T \mathbf{x}$. In Octave:

```
Ureduce = U(:, 1:k);
z = Ureduce' * x;
```

## 12.2 Reconstruction from compressed representation

$\mathbf{x}_{\text{approx}} = U_{\text{reduce}}\mathbf{z}$.

## 12.3 Choosing the number of principal components

Define average squared projection error $\frac{1}{m}\sum_{i=1}^{m} ||\mathbf{x}^{(i)} - \mathbf{x}_{\text{approx}}^{(i)}||^2$ and total variation in the data $\frac{1}{m}\sum_{i=1}^{m} ||\mathbf{x}^{(i)}||^2$.

Typically, we choose $k$ to be smallest value so that

$$\frac{\frac{1}{m}\sum_{i=1}^{m}||\mathbf{x}^{(i)} - \mathbf{x}^{(i)}_{\text{approx}}||^2}{\frac{1}{m}\sum_{i=1}^{m}||\mathbf{x}^{(i)}||^2} \leqslant 0.01.$$

In Octave:

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

We get $S \in \mathbb{R}^{n\times n}$ and for given $k$, we just check if

$$\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \geqslant 0.99.$$

## 12.4 Supervised learning speedup

We extract inputs, i.e., unlabeled dataset: $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)} \xrightarrow{PCA} \mathbf{z}^{(1)}, ..., \mathbf{z}^{(m)}$. Then we get the new training set: $((\mathbf{z}^{(1)}, y^{(1)}), ..., (\mathbf{z}^{(m)}, y^{(m)}))$.

# 13 Anomaly Detection

## 13.1 Gaussian distribution

Say $x \in \mathbb{R}$. If $x$ is a distributed Gaussian with mean $\mu$, variance $\sigma^2$, then $x \sim \mathcal{N}(\mu, \sigma^2).p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$.

## 13.2 Anomaly detection algorithm

**Algorithm 13.1** (Anomaly Detection Algorithm)**.** First, choose features $x_i$ that might be indicative of anomalous examples. Then, fit parameters $\mu_1, ..., \mu_n, \sigma_1^2, ..., \sigma_n^2$, and $\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}, \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$. Given new example $\mathbf{x}$, compute $p(\mathbf{x})$ :

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

Anomaly if $p(\mathbf{x}) < \varepsilon$.

## 13.3 Developing and evaluating an anomaly detection system

Fit model $p(\mathbf{x})$ on training set $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$. On a cross validation/test example $\mathbf{x}$, predict

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

Possible evaluation metrics:

- True positive, false positive, false negative, true negative
- Precision/Recall
- $F$ score

Use cross validation set to choose parameter $\varepsilon$.

## 13.4 Anomaly detection vs. supervised learning

- Anomaly detection
  - ∗ Very small number of positive examples ($y = 1$) (0-20 is common).
  - ∗ Large number of negative ($y = 0$) example.
  - ∗ Application: Fraud detection, manufacturing, monitoring machines in a data center, and etc.
- Supervised learning
  - ∗ Large number of positive and negative examples.
  - ∗ Application: Email spam classification, weather prediction, cancer classification, and etc.

## 13.5 Multivariate Gaussian distribution

Say $\mathbf{x} \in \mathbb{R}^n, \mu \in \mathbb{R}^n, \Sigma \in \mathbb{R}^{n \times n}$ (covariance matrix).

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

## 13.6 Anomaly detection using the multivariate Gaussian distribution

Given training set $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$.$\mu = \frac{1}{m} \sum_{i=1}^{m} \mathbf{x}^{(i)}, \Sigma = \frac{1}{m} \sum_{i=1}^{m} (\mathbf{x}^{(i)} - \mu)(\mathbf{x}^{(i)} - \mu)^T$.

**Algorithm 13.2** (Anomaly Detection Algorithm)**.** First, fit model $p(\mathbf{x})$ by setting $\mu, \Sigma$. Then given a new example $\mathbf{x}$, compute $p(\mathbf{x})$ and flag an anomaly if $p(\mathbf{x}) < \varepsilon$.

The original model $p(\mathbf{x}) = p(x_1; \mu_1, \sigma_1^2) \times \cdots \times p(x_n; \mu_n, \sigma_n^2)$ corresponds to multivariate Gaussian where $\Sigma = \begin{pmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \cdots & \sigma_n^2 \end{pmatrix}$.

The differences are:

- Original model:
    - Manually create features to capture anomalies where $x_1, ..., x_n$ take unusual combinations of values.
    - Computationally cheaper (alternatively, scales better to large $n$).
    - Works even if $m$ is small.
- Multivariate Gaussian:
    - Automatically captures correlations between features.
    - Computationally more expensive.
    - Must have $m > n$ ($m \geqslant 10n$) or else $\Sigma$ is non-invertible.

# 14 Recommender Systems

## 14.1 Content-Based recommendations

Let $r(i,j) = 1$ if user $j$ has rated movie $i$ (0 otherwise), $y^{(i,j)}$ be the rating by user $j$ on movie $i$ (if defined), $\theta^{(j)}$ be the parameter vector for user $j$, and $\mathbf{x}^{(i)}$ be the feature vector for movie $i$. Hence, for user $j$ and movie $i$, predicted rating is $(\theta^{(j)})^T(\mathbf{x})^{(i)}$. Let $m^{(j)}$ be the number of movies rated by user $j$. To learn $\theta^{(j)}$, we solve

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

To learn $\theta^{(1)}, ..., \theta^{(n_u)}$, we solve

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

The optimization algorithm is gradient descent:

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

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

## 14.2 Collaborative filtering

Given $\theta^{(1)}, ..., \theta^{n_u}$, to learn $\mathbf{x}^{(i)}$, we need to solve

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

Given $\theta^{(1)}, ..., \theta^{n_u}$, to learn $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(n_m)}$, we need to solve

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

To improve the procedure, we can minimize $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(n_m)}$ and $\theta^{(1)}, ..., \theta^{(n_u)}$ simultaneously, and we have

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

and we solve

$$\min_{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}} J(\mathbf{x}^{(1)}, ..., \mathbf{x}^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}).$$

**Algorithm 14.1** (Collaborative Filtering Algorithm)**.** First, initialize $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}$ to small random values. Then minimize $J(\mathbf{x}^{(1)}, ..., \mathbf{x}^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)})$ using gradient descent (or an advanced optimization algorithm). If we use gradient descent, for every $j = 1, ..., n_u, i = 1, ..., n_m$ :

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

where $\mathbf{x}, \theta \in \mathbb{R}^n$. Finally, for a user with parameters $\theta$ and a movie with learned features $\mathbf{x}$, predict a star rating of $\theta^T \mathbf{x}$.

## 14.3 Vectorization: Low rank matrix factorization

Suppose the predicted ratings matrix is

$$Y = \begin{pmatrix} (\theta^{(1)})^T(\mathbf{x}^{(1)}) & (\theta^{(2)})^T(\mathbf{x}^{(1)}) & \cdots & (\theta^{(n_u)})^T(\mathbf{x}^{(1)}) \\ (\theta^{(1)})^T(\mathbf{x}^{(2)}) & (\theta^{(2)})^T(\mathbf{x}^{(2)}) & \cdots & (\theta^{(n_u)})^T(\mathbf{x}^{(2)}) \\ \vdots & \vdots & \ddots & \\ (\theta^{(1)})^T(\mathbf{x}^{(n_m)}) & (\theta^{(2)})^T(\mathbf{x}^{(n_m)}) & \cdots & (\theta^{(n_u)})^T(\mathbf{x}^{(n_m)}) \end{pmatrix}.$$

We can let

$$X = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(n_m)})^T \end{pmatrix}, \Theta = \begin{pmatrix} (\theta^{(1)})^T \\ (\theta^{(2)})^T \\ \vdots \\ (\theta^{(n_u)})^T \end{pmatrix},$$

then $Y = X\Theta^T$.

# 15 Large Scale Machine Learning

## 15.1 Stochastic gradient descent

Let

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

First, randomly reorder dataset. Then, repeat

$$\left\{\text{for } i := 1, ..., m \left\{\theta_j := \theta_j - \alpha(h_\theta(\mathbf{x}^{(i)}) - y^{(i)})x_j^{(i)} \text{ for every } j = 0, ..., n\right\}\right\}.$$

## 15.2 Mini-batch gradient descent

Say $b = 10, m = 1000$. Repeat

$$\left\{\text{for } i := 1, 11, 21, 31, ..., 991 \left\{\theta_j := \theta_j - \alpha\frac{1}{10}\sum_{k=i}^{i+9}(h_\theta(\mathbf{x}^{(k)}) - y^{(k)})x_j^{(k)} \text{ for every } j = 0, ..., n\right\}\right\}.$$

## 15.3 Stochastic gradient descent convergence

During learning, compute $\text{cost}(\theta, (\mathbf{x}^{(i)}, y^{(i)}))$ before updating $\theta$ using $(\mathbf{x}^{(i)}, y^{(i)})$. Every 1000 (or other numbers) iterations, plot $\text{cost}(\theta, (\mathbf{x}^{(i)}, y^{(i)}))$ averaged over the last 1000 examples processed by algorithm.

## 15.4 Online learning

Repeat

$$\{\text{Get } (x, y) \text{ corresponding to user. Update } \theta \text{ using } (x, y) :$$
$$\theta_j := \theta_j - \alpha(h_\theta(\mathbf{x}) - y)x_j \text{ for } j = 0, ..., n\}.$$

## 15.5 Map reduce and data parallelism

Many learning algorithms can be expressed as computing sums of functions over the training set. Therefore we can speed up via more machines or cores.