Contents

1 Introduction

Machine learning enables computer to complete tasks without being explicitly programmed. With appropriate methods applied, computer can obtain the ability to better complete a task by learning from previous results with respect to this task, which resembles the learning process of human beings. By "better", we mean better performance under the evaluation of some sort of quantative measurement.

Machine learning can be divided into **supervised learning** and **unsupervised learning** according to the data set used.

In supervised learning, the correct output of each case in the data set is already known. The learning algorithm is supposed to reveal the relationship between the input and the output. Supervised learning problems can be categorized into **regression** problems and **classification** problems. In regression problems, the output has continuous value, while in classification the output is discrete.

In unsupervised learning, no correct output is provided in advance. Structure of the data needs to be derived by **clustering** the data based on the relationship among the variables in the data set.

2 Univariate linear regression

2.1 Hypothesis function and cost function

Linear regression with one variable, i.e. univariate linear regression, is the simplest regression problem. A single output needs to be predicted from a single input based on a given dataset containing a series of input/output pairs in which the outputs are correct.

Since it is assumed that the input and the output are linearly relative, we take the **hypothesis function**

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{1}$$

Our target is to find the approriate parameters θ_0, θ_1 that minimize the **cost function**

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

in which $\left(x^{(i)}, y^{(i)}\right), i = 1, 2 \dots m$ are the training examples, and m is the size of the training set. In mathematical languague, the problem we are supposed to solve is $\min\limits_{\theta_0, \theta_1} J(\theta_0, \theta_1)$.

2.2 Gradient descent

A common method to help automatically find the optimal parameters θ_0, θ_1 is **gradient descent**. It starts with some random θ_0, θ_1, and iteratively alter the

values of θ_0, θ_1 according to the rules

$$\theta_j \coloneqq \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \tag{3}$$

until convergence.

Gradient descent algorithm does not necessarily converge to the global minimum for any function. If there exists a few local minimums, the algorithm could wind up at any of them with different initial choices of θ_j. However, cost functions admit only one local minimum, which is hence also the global minimum. Thus gradient descent can be applied here without having to worry about the possibility of converging to a local minimum due to unwise choice of initial value.

For univariate linear regression, (3) becomes

$$\begin{aligned} \theta_0 &\coloneqq \theta_0 - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \\ \theta_1 &\coloneqq \theta_1 - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x^{(i)} \end{aligned} \tag{4}$$

3 Multivariate linear regression

3.1 Hypothesis function and cost function

For linear regression problem with multiple variables rather than one single variable, we have the hypothesis function

$$h_\theta(x) = \theta^{\mathsf{T}} x = \sum_{i=0}^{n} \theta_i x_i \tag{5}$$

in which $x_0 \equiv 1$ and n is the number of features(variables) being considered. And the cost function is

$$J(\theta) = J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \tag{6}$$

And the gradient descent algorithm becomes

$$\theta_j \coloneqq \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \tag{7}$$

in which $j = 0, 1 \ldots n$.

3.2 Feature scaling

Making sure that values of all the features are on similar scale helps gradient descent to converge much more quickly. A practical choice would be to limit each feature into the range of (-1,1). Note that this does not go for $x_0 \equiv 1$.

If a feature x_i is guaranteed to range from a to b, we can use **mean normalization** and substitute x_i with $\frac{x_i-(a+b)/2}{b-a}$. It is also fine to use $\frac{x_i-\mu_i}{\sigma_i}$, in which μ_i and σ_i are respectively the average and standard deviation of x_i.

3.3 Learning rate α

For small enough learning rate α, the cost function should decrease on every iteration. But if α is too small, the convergence may become extremely slow. And if α is too large, the cost function might diverge with the iteration. Name k the number of iterations, drawing the $J(\theta) - k$ plot could help to choose an appropriate learning rate.

3.4 Analytical solution: normal equation

Minimum of the cost function can be found analytically by solving the normal equation

$$\frac{\partial}{\partial \theta_j} J(\theta) = 0, j = 0, 1, \ldots, n \tag{8}$$

For the multivariate linear regression problem, if we define the matrix X

$$X = \begin{bmatrix} \left(x^{(1)}\right)^\mathsf{T} \\ \left(x^{(2)}\right)^\mathsf{T} \\ \cdots \\ \left(x^{(m)}\right)^\mathsf{T} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \ldots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \ldots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \ldots & x_n^{(m)} \end{bmatrix} \tag{9}$$

and the vector

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \cdots \\ y^{(m)} \end{bmatrix} \tag{10}$$

it can be verified that the optimal θ would be

$$\theta = \left(X^\mathsf{T} X\right)^{-1} X^\mathsf{T} y \tag{11}$$

Normal equation method does not require a learning rate to be chosen. Neither is iteration needed. However, since it requires the calculation of $\left(X^\mathsf{T} X\right)^{-1}$, it can become quite slow when n is large, whereas gradient descent still works fine.

What's more, it is possible that $X^\mathsf{T} X$ is non-invertible (singular, degenerate) when there are redundant features or when there are too many features ($n > m$).

In such cases, the *pinv* (persudo inverse) function in octave and matlab will help avoid failure of the calculation. We could also manually delete redundant features or consider using regularisation, which will be covered later.

4 Logistic regression

4.1 Hypothesis function and cost function

Recall that in supervised learning, when the output is discretely valued, the problem is said to be a classification problem. When there are only two classes, the problem is called a logistic regression problem. We will illustrate how to solve such problem.

We use the hyphothesis function

$$h_\theta(x) = g(\theta^\mathsf{T} x) = \frac{1}{1 + e^{-\theta^\mathsf{T} x}} \tag{12}$$

where $g(z) = \frac{1}{1+e^{-z}}$ is called **sigmoid function** or **logistic function**.

$h_\theta(x)$ can be interpreted as the probablity that $y = 1$ with input x, i.e.

$$h_\theta(x) = P(y = 1|x, \theta)$$

If $h_\theta(x) \geq 0.5$, we predict that $y = 1$; otherwise ($h_\theta(x) < 0.5$) we predict $y = 0$. This gives us the decision boundry $\theta^\mathsf{T} x = 0$.

Squared error cost function is no longer appropriate for logistic regression. We use the cost function

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right) \tag{13}$$

The logistic regression problem can now be solved by minimize he cost function: $\min_\theta J(\theta)$. Again we can apply gradient descent

$$\theta_j \coloneqq \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \tag{14}$$

Some examples of more complex optimization algorithms: conjugate gradient, BFGS, L-BFGS. They do not require the manual selection of α and they are usually faster than gradient descent.

4.2 One-vs-all classification

When we have more than two classes, it is possible to solve the classification problem with the algorithm we have developped for logistic regression problems by using the socalled one-vs-all classification algorithm.

For each class $y = i$, we fit a hypothesis function (regression classifier) $h_\theta^{(i)}(x)$ that can be interpreted as the probability of $y = i$, i.e.

$$h_\theta^{(i)}(x) = P(y = i|x, \theta)$$

Equiped with these classifiers, when asked to predict the output for a certain x, we choose the class i that maximizes $h_\theta^{(i)}(x)$.

5 Regularization: solution to overfitting

When too many features are involved in the regression, we could end up with the overfitting problem: the learned hypothesis fits the training set very well so that $J(\theta) \approx 0$, but fails to make good prediction for new examples.

There are a few possible solutions to the overfitting problem.

1. Reduce number of features. Manual selection or using model selection algorithm, which will be covered later.
2. Regularization. All features are kept but magnitudes of θ_js are reduced.

5.1 Cost function of regularization

The principle of regularization is to reduce the magnitudes of parameters θ_j. This can be achieved by adding a squared item to the cost function:

$$J(\theta) = J_0(\theta) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

By convention the added square item does not inclued θ_0.

5.2 Regularized linear regression

For linear regression, we now have the new cost function

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{15}$$

And the iteration equation in gradient descent becomes

$$\begin{aligned}
\theta_0 &\coloneqq \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)} \\
\theta_j &\coloneqq \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}\theta_j\right] \\
&= \theta_j\left(1 - \frac{\alpha\lambda}{m}\right) - \frac{\alpha}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}, j = 1, 2\ldots n
\end{aligned} \tag{16}$$

Intuitively, the $\left(1-\frac{\alpha\lambda}{m}\right)$ factor reduces the magnitude of θ_j at each iteration.

The solution of the normal equation changes into

$$\theta = \left(X^\mathsf{T}X + \lambda \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix} \right)^{-1} X^\mathsf{T}y \tag{17}$$

It can be proved that as long as $\lambda > 0$, the matrix to be inverted is invertible, which solves the problem of non-invertibility of $X^\mathsf{T}X$.

5.3 Regularized logistic regression

For logistic regression, we now have the cost function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}log(h_\theta(x^{(i)})) + (1-y^{(i)})log\left(1-h_\theta(x^{(i)})\right)\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{18}$$

And the iteration equation of gradient descent becomes

$$\begin{aligned} \theta_0 &\coloneqq \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)} \\ \theta_j &\coloneqq \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}\theta_j\right] \\ &= \theta_j\left(1-\frac{\alpha\lambda}{m}\right) - \frac{\alpha}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}, j = 1, 2\dots n \end{aligned} \tag{19}$$

which is actually the same as that of linear regression.

6 Neural network

6.1 Introduction

Linear regression and logistic regression may not be pratical solutions to some machine learning problems because we will end up with too many features to deal with. As an example, suppose we want to conduct a logistic regression on a 100×100 grey scale image (maybe trying to tell if it is an image of a car). If we want to take the quardatic items (x_ix_j) as features, there will be roughly 5×10^7 different features. If higher order items were to be taken into account, which is not uncommon, the number could become even higher. Trying to solve such problem with linear/logistic regression is obviously a dead end. The method to turn to in such case is neural network, which is the state-of-the-art technique for many machine learning applications.

Neural network dates back to a few decades ago when people started to try to mimic the way human brain works with computer. Due to the high requirement of computing capacity, it did not get into the spotlight until recent years. In a human neural cell, an electricity signal is inputted into the cell through an input wire called "dendrite". The cell processes the signal, and then sends a new signal out via an output wire called "axon". In machine learning, a logistic unit inside a neural network works in similar way. Different signals x_i is inputted into the unit, and the unit outputs a new signal

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^\intercal x}} \tag{20}$$

as illustrated in Figure 1.

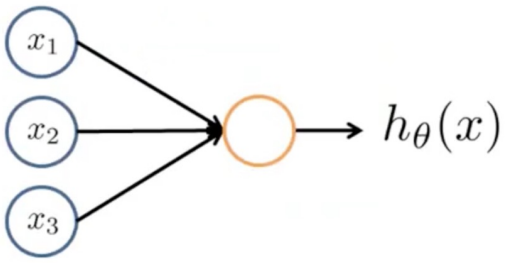

Figure 1: Basic logistic unit in a neural network

When a series of such logistic units, or neurons get connected, they form a neural network, as illustrated by Figure 2.

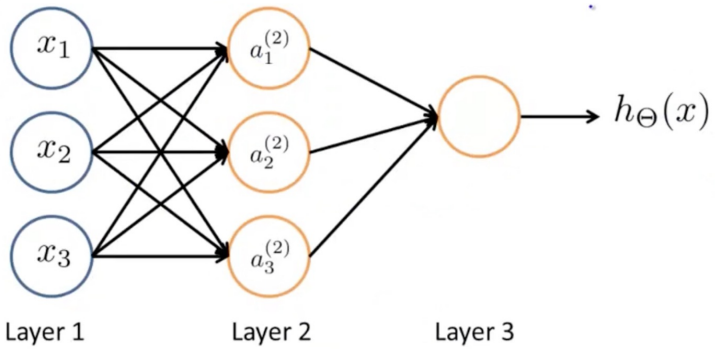

Figure 2: A simple neural network

Quantatively, by denoting the "activation" of unit i in layer j as $a_i^{(j)}$, the matrix of weights controlling function mapping from layer j to layer $j + 1$ as

$\Theta^{(j)}$, we have

$$\begin{aligned} a_1^{(2)} &= g\left(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\right) \\ a_2^{(2)} &= g\left(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\right) \\ a_3^{(2)} &= g\left(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\right) \\ h_\theta(x) = a_1^{(3)} &= g\left(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}\right) \end{aligned} \tag{21}$$

in which $g(z)$ is the sigmoid function. The vectorized version of (21) can be written as

$$a^{(j+1)} = g\left(\Theta^{(j)}\begin{bmatrix}1\\ a^{(j)}\end{bmatrix}\right) \tag{22}$$

If layer j has s_j neurons (not including the bias neuron $a_0^{(j)} = 1$), it is clear that $\Theta^{(j)}$ is a $s_{j+1} * (s_j + 1)$ matrix.

6.2 Examples

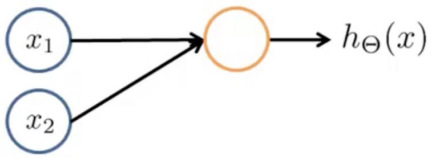

Figure 3: Implementation of logical AND/OR with neural network

If we choose $\Theta^{(1)} = \begin{bmatrix}-30 & 20 & 20\end{bmatrix}$, it can be verified easily that the output will be x_1 AND x_2. Similarly, with $\Theta^{(1)} = \begin{bmatrix}-10 & 20 & 20\end{bmatrix}$, the output will be x_1 OR x_2.

With the neural network shown in Figure 4, we can calculate NOT x_1 with $\Theta^{(1)} = \begin{bmatrix}10 & -20\end{bmatrix}$

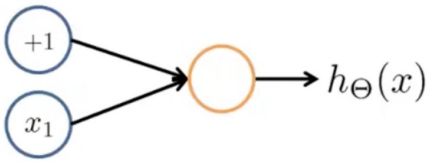

Figure 4: Implementation of logical NOT with neural network

With the examples given above, we can calculate a more complex logical function XNOR. Note that

$$x_1 \text{ XNOR } x_2 = (x_1 \text{ AND } x_2) \text{ OR } (NOT(x_1) \text{ AND } NOT(x_2))$$

Thus we can calculate XNOR with the neural network shown in Figure 5.

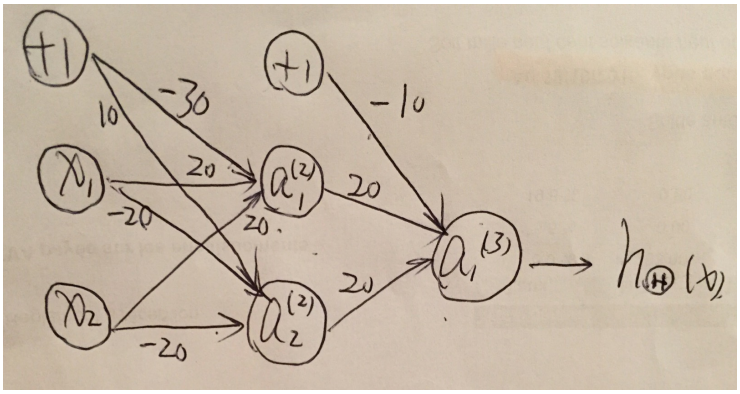

Figure 5: Implementation of logical XNOR with neural network

Here we have $\Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \\ 10 & -20 & -20 \end{bmatrix}$ and $\Theta^{(2)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix}$.

6.3 Cost function

Recall that we have the following cost function for regularized logistic regression

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\Big[y^{(i)}log(h_\theta(x^{(i)})) + (1-y^{(i)})(log(1-h_\theta(x^{(i)})))\Big] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{23}$$

Supose we are trying to solve a K-classification problem with neural network. The cost function of neural network is its generalization

$$\begin{aligned} J(\Theta) = &-\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\Big[y_k^{(i)}log(h_\theta(x^{(i)}))_k + (1-y_k^{(i)})(log(1-(h_\theta(x^{(i)}))_k)\Big] \\ &+ \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2 \end{aligned} \tag{24}$$

in which L is the number of layers. Note that the regularization term does not include Θ_{j0}, which is related to the bias term.

In order to minimize $J(\Theta)$ using gradient descent or other methods, we need to compute $\frac{\partial J(\Theta)}{\partial \Theta_{ij}}$ for all i, j. The method to do this is called backpropagation.

6.4 Backpropagation

Consider one specific training example (x, y). Intuitively speaking, in the backpropagation algorithm, we use $\delta_j^{(l)}$ to denote the "error" of node j in layer l.

For the output layer (layer L), we have

$$\delta_j^{(L)} = a_j^{(L)} - y_j \tag{25}$$

or in the vectorized way

$$\delta^{(L)} = a^{(L)} - y \tag{26}$$

in which $a^{(L)} = h_\Theta(x)$ because this is the output layer. For other layers, we have

$$\delta^{(l)} = (\Theta^{(l)})^\mathsf{T}\delta^{(l+1)} .* g'(z^{(l)}), l = L-1, L-2 \dots 2. \tag{27}$$

Note that the result of $(\Theta^{(l)})^\mathsf{T}\delta^{(l+1)}$ is a vector of dimension s_l+1, while $g'(z^{(l)})$ is of dimension s_l. Since we expect $\delta^{(l)}$ to be of dimension s_l, the first component (the 0th) of $(\Theta^{(l)})^\mathsf{T}\delta^{(l+1)}$ should be discarded.

There is no $\delta^{(1)}$ item because contained in layer 1 are the inputted features of the training example who do not have any "error" associated with them. It's straightforward to demonstrate that $g'(x) = g(x)(1-g(x))$, thus we have

$$\delta^{(l)} = (\Theta^{(l)})^\mathsf{T}\delta^{(l+1)} .* \left[a^{(l)}(1-a^{(l)})\right], l = L-1, L-2 \dots 2. \tag{28}$$

With all $\delta^{(l)}(l = 2, 3 \dots L)$ defined, we now present how the backpropagation algorithm works for a more general training set containing m training examples. We define $\Delta_{ij}^{(l)} = 0$ for all l, i, j, then loop over i from 1 to m:

1. Set $a^{(1)} = x^{(i)}$
2. Forward propagation: calculate $a^{(l)}, l = 2, 3 \dots L$
3. Backpropagation: calculate $\delta^{(l)}, l = L, L-1 \dots 2$
4. $\Delta^{(l)} \coloneqq \Delta^{(l)} + \delta^{(l+1)} \left(a^{(l)}\right)^\mathsf{T}$

After obtaining $\Delta_{ij}^{(l)}$, we define $D_{ij}^{(l)}$ as follow

$$\begin{aligned} D_{ij}^{(l)} &\coloneqq \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)}, \text{ if } j \neq 0 \\ D_{ij}^{(l)} &\coloneqq \frac{1}{m}\Delta_{ij}^{(l)}, \text{ if } j = 0 \end{aligned} \tag{29}$$

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

The implementation of the backpropagation algorithm requires a lot of mathematical details and is thus quite errorprone. In order to check whether our imeplementation is correct, it is helpful to do the **gradient checking**, i.e. comparing the $\frac{\partial}{\partial\theta_i}J(\theta)$ calculated by backpropagation against the numerical value calculated by

$$\frac{\partial}{\partial\theta_i}J(\theta) = \frac{J(\theta_1 \dots \theta_i + \epsilon \dots \theta_n) - J(\theta_1 \dots \theta_i - \epsilon \dots \theta_n)}{2\epsilon}$$

Here θ becomes a 1d vector because in matlab and octave, when minimizing a function with advanced methods(e.g. fminunc), it is required that the gradient be provided in the form of a vector rather than a matrix. In order to comply with this requirement, the Θ matrices need to be unrolled together. The code looks like

```
thetaVec = [Theta1(:); Theta2(:); Theta3(:)];
```

The code for the reversal looks like

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

The initial choice of Θ is subtle. **Symmetry breaking** requires that the matrices be randomly initialized. We will initialize each matrix item with a random value within the range of $(-\epsilon, \epsilon)$. The code looks like

```
Theta1 = rand(10,11) * 2 * INIT_EPSILON - INIT_EPSILON;
```

6.5 Summary

Here is a summary of the implementation of neural network learning algorithm.

1. Randomly initialze weight matrices ($\Theta^{(l)}$).
2. Implement forward propagation to calculate $h_\Theta(x^{(i)})$ with $x^{(i)}$.
3. Implement calculation of the cost function $J(\Theta)$.
4. Implement backpropagation to calculate $\frac{\partial}{\partial \Theta_{ij}} J(\Theta)$.
5. Implement gradient checking to verify the correctness of $\frac{\partial}{\partial \Theta_{ij}} J(\Theta)$ results gained above. Then disable this part of the code to ensure efficiency.
6. Use gradient descent or other optimization methods to minimize $J(\Theta)$.

$J(\Theta)$ is not necessarily convex for neural network, which means it is possible that we wind up at a local minimum rather than the global minimum. In reality this seldom causes problem because even if a local minimum is obtained, it is not too far away from the global minimum and is thus an acceptable solution.

7 Machine learning diagnostics

When the prediction of a hypothesis trained by a learning algorithm turns out to have unacceptable errors, a series of options are available as promising further actions to take:

1. Get more training examples.

2. Try smaller sets of features.
3. Try additional features.
4. Try polynomial features.
5. Adjusting value of regularization parameter λ.

Rather than randomly try different approaches to improve the performance of the algorithm, it is worthwhile to implement machine learning diagnostics, which apply certain tests to figure out what is / isn't working and can thus provide guidance on what should be done next.

7.1 Evaluate a hypothesis: test set

Training the algorithm with all the examples available might end up overfitting the training set. To avoid such possibility, we can divide all examples into the **training set** and the **test set**. A typical choice is to put 70 % of the examples in the training set and 30% of the examples in the test set. Then a series of parameters θ can be learned from the training set by minimizing $J(\theta)$, after which the functionality of the algorithm can be measured by the test set error $J_{test}(\theta)$. For linear regression, we have

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

Similarly, for logistic regression, we have

$$J_{test}(\theta) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} (y_{test}^{(i)} \log(h_\theta(x_{test}^{(i)})) + (1 - y_{test}^{(i)}) \log(1 - h_\theta(x_{test}^{(i)}))) \quad (30)$$

For logistic regression (i.e. 2-classification) problem, another way to define the test set error is

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

in which

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

7.2 Model selection: cross-validation set

In order to choose among different models (e.g. different degrees of polynomial), we need further partition of the examples. All examples should now be divided into 3 groups: **training set**, **cross-validation set** and **test set**. Typically the percentages of the 3 groups are 60%, 20% and 20%.

For each training model candidate $h_\theta^{(i)}(x)$, we train a series of parameters $\theta^{(i)}$ by minimizing $J(\theta)$. Then we calculate the cross-validation set error $J_{cv}(\theta^{(i)})$ for them, in which

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

for linear regression, and similar as (30) for logistic regression. Finally, the model that minimizes $J_{cv}(\theta)$ stands out, and its performance can be measured with the test set error.

7.3 Bias & variance

When the training set is underfitted, the algorithm is said to have high bias, while when the training set is overfitted, it is said to have high variance. We will introduce methods to diagnose such problems.

In a specific linear regression problem, when higher order items are introduced into the hypothesis, the training set error (regularization not considered) will be reduced. However, too many high order items may cause overfit problem, which will increase the cross-validation set error. Figure 6 depicts such trend.

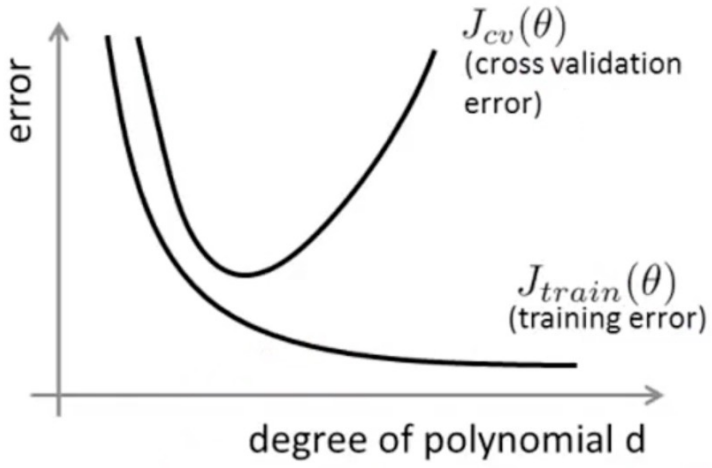

Figure 6: Diagnosing bias and variance with errors: different polynomial degrees

It is obvious that in a high bias (underfit) case, both $J_{cv}(\theta)$ and $J_{train}(\theta)$ are high, while in a high variance (overfit) case, $J_{cv}(\theta)$ is high but $J_{train}(\theta)$ is low.

We introduced regularization as the method to combat overfit, but the choice of the regularization parameter λ can be subtle. It turns out that an appropriate

λ can be chosen following a criteria similar to that of model selection. Also, we can diagose bias and variance following the same rule stated above, as shown in Figure 7. Here the $J_{train}(\theta)$ does not include the regularization item.

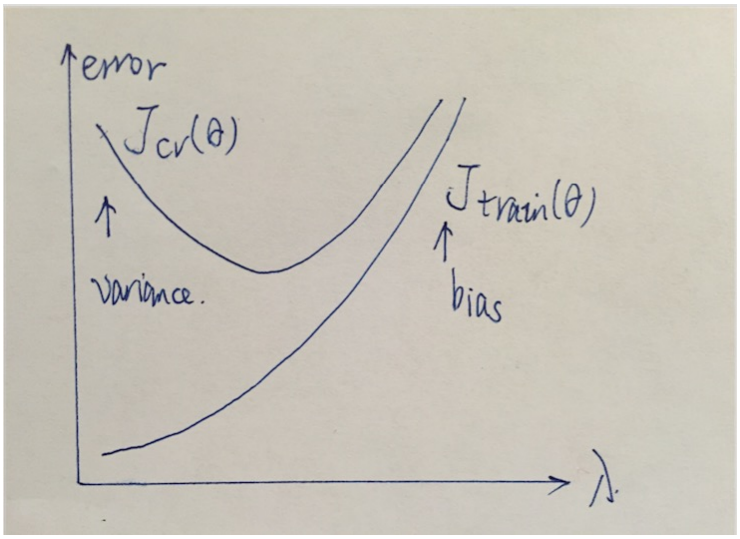

Figure 7: Diagnosing bias and variance with errors: different regularization parameter

7.4 Learning curve

Learning curve is the curve that depicts the variance of the training error and the cross-validation error with the training set size m. It can help estimate whether or not the performance of the algorithm will be improved by adding more examples into the training set.

When more examples are added into the training set, the hypothesis can no longer fit all examples well, thus the training error $J_{train}(\theta)$ will increase, whereas the hypothesis extends better towards new examples, thus the cross-validation error $J_{cv}(\theta)$ decreases. A typical learning curve is shown in Figure 8.

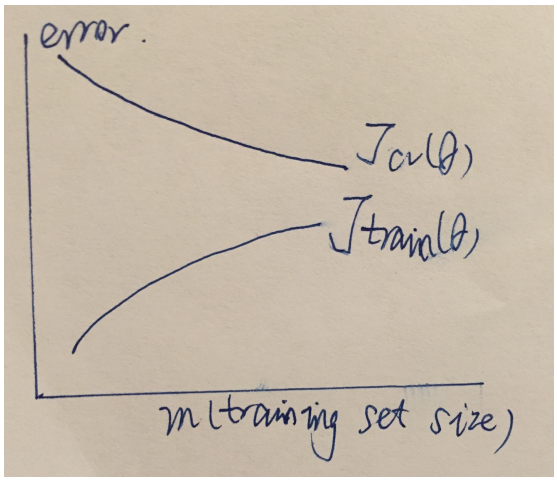

Figure 8: A typical learning curve

In a high-bias case, the examples are underfitted (e.g. trying to fit examples that follow a 5-degree polynomial with a straight line). Even if the size of the training set is increased, the hypothesis will still fail to correctly describe the relationship between the input and the output. Thus, both $J_{cv}(\theta)$ and $J_{train}(\theta)$ will be high, and they will be quite close to each other, as shown in Figure 9. In this case, **collecting more training examples is not likely to help.** Essential adjustment to the hypothesis should be made, i.e. adding more features.

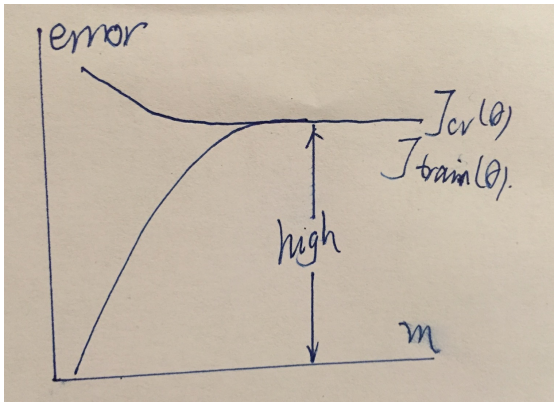

Figure 9: Learning curve with high bias

In a high variance case, the examples are overfitted. $J_{train}(\theta)$ is typically lower than $J_{cv}(\theta)$, as shown in Figure 10. If more examples are added to the training set, the training set becomes "less overfitted", thus $J_{train}(\theta)$ is likely to increase, while $J_{cv}(\theta)$ will decrease. In such case, **collecting more training**

examples will help to improve the performance.

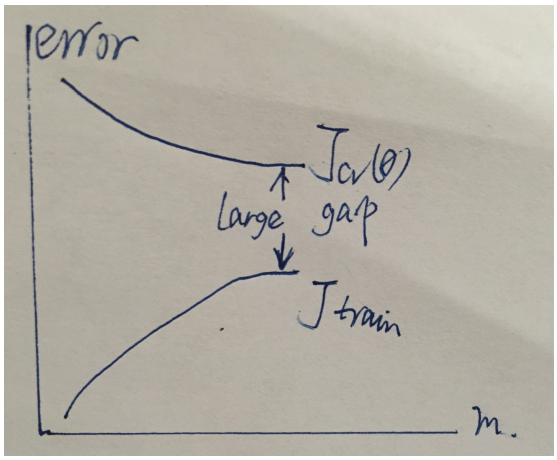

Figure 10: Learning curve with high variance

7.5 Conclusion

We will now conclude this section by providing the effects of the solutions to performance problems listed at the begining.

Get more training examples fixes high variance

Try smaller sets of features fixes high variance

Try additional features fixes high bias

Try polynomial features fixes high bias

Increase regularization parameter λ fixes high variance

Decrease regularization parameter λ fixes high bias

8 Machine learning system design

When designing a machine learnig system, a bunch of different options are available for choice to achieve better performance. Take the example of an email spam classifier, we can:

- Collect large amount of data.
- Develop sophisticated features based on email routing information, e.g. email header.

- Develop sophisticated features for message body, e.g. treating 'discount' and 'discounts', 'dealer' and 'Dealer' as the same word, ignoring punctuations, etc.
- Develop sophisticated algorithm to detect misspellings, e.g. 'magaz1ne', 'm0rtgage', etc.

Rather than randomly making a choice based on gut feeling, systematic analysis methods are available to help choose the best option.

8.1 Error analysis

A recommended approach to a machine learning problem is as follows. Start with a "quick and dirty" version of the algorithm easy to implement. Implement it and test it on the cross-validation set. Plot learning curves to decide if more data or more features are needed. Then conduct what we call **error analysis**: manually examine examples in c-v set on which the algorithm made errors, and try to figure out any systematic trend of such examples. Based on the trends, add new features to the model and see if it helps.

8.2 Handling skewed data

Take the cancer classification as an example. If we have a learning algorithm that diagnoses the test set with 1% error, i.e. 99% of the diagnostics it make are correct, it seems to be a good algorithm. However, provided that only 0.5% of the patients in the training set have cancer, this algorithm is far from satisfactory. Actually, a dumb classifier that always outputs $y = 0$ (no cancer) appears to have better performance. Obviously, in such case, classification error, or classification accuracy, is no longer an appropriate metric of the performance of the algorithm.

Table 1: Definitions for skewed data error metrics

Predicted \ Actual	1	0
1	True Positive	False Positive
0	False Negative	True Negative

Table 1 provides some useful definitions for skewed data error metrics. If the actual class is 1 and the algorithm predicts 1, we say that the algorithm has made a true positive prediction, etc. Now we can define the **precision** and the **recall** of the algorithm.

$$\begin{aligned} \text{Precision} &= \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \\ \text{Recall} &= \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} \end{aligned} \tag{31}$$

Clearly, precision is the percentage of patients who actually have cancer among patients diagnosed to have cancer, while recall is the percentage of patients who are diagnosed to have cancer among patients who actually have cancer.

With precision and recall defined, we can introduce the tradeoff between them. Normally we predict $y = 1$ when $h_\theta(x) \geq 0.5$ and $y = 0$ when $h_\theta(x) < 0.5$. If we want to predict $y = 1$ only when very confident, we can increase the threshold for prediction, e.g. to 0.9. Obviously this will increase the precision but decrease the recall. On the contrary, if we want to alert more patients possible to have cancer, we can decrease the threshold, say to 0.3, and we will end up with lower precision but higher recall.

Taking both precision and recall into account, we define the F_1 score of the algorithm

$$F_1 = \frac{2PR}{P+R} \tag{32}$$

and use it as a measurement of the performance of the algorithm. Generally, an algorithm with a better F_1 score has better overall performance.

8.3 Data

Sufficient and appropriate data is essential to the performance of the algorithm. We should always ensure that the features we use include **enough** information to predict the output correctly. A useful test to determine whether the information is enough is to ask ourselves: is a human expert in this field capable of making a confident predication of the output based on the information we provide?

If we are using low-bias algorithms, e.g. neural network with a lot of hidden layers or a linear regression with a lot of feature, large amount of data will help avoid overfitting and is thus usually preferable.

9 Support vector machine (SVM)

9.1 Cost function

Recall the cost function of logistic regression:

$$\begin{aligned} J(\theta) =& \frac{1}{m}\sum_{i=1}^{m}\left(-y^{(i)}\log\frac{1}{1+e^{-\theta^\mathsf{T}x^{(i)}}} - (1-y^{(i)})\log\left(1-\frac{1}{1+e^{-\theta^\mathsf{T}x^{(i)}}}\right)\right) \\ &+ \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \end{aligned} \tag{33}$$

For a training example (x, y) with $y = 1$, we expect $\theta^\mathsf{T}x \gg 0$ in order to minimize the cost function. Similarly, we expect $\theta^\mathsf{T}x \ll 0$ for an example with $y = 0$. The graph of $f_1(z) = -\log\frac{1}{1+e^{-z}}$ is provided with the blue line in Figure 11, while the graph of $f_0(z) = -\log\left(1-\frac{1}{1+e^{-z}}\right)$ is provided with the blue line in Figure 12.

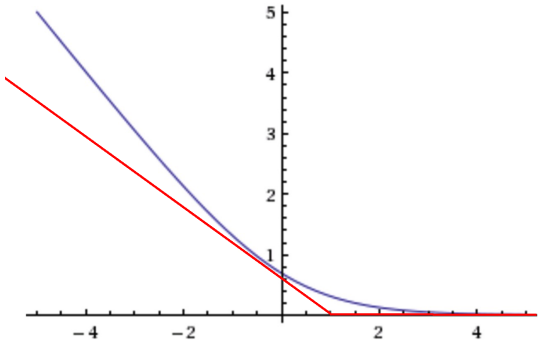

Figure 11: Image of cost1(z)

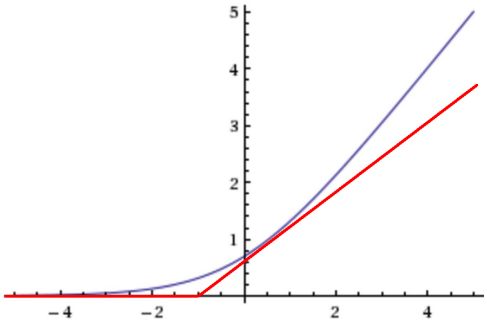

Figure 12: Image of cost0(z)

In SVM, we will use new functions $cost1(z), cost0(z)$ as depicted by the red lines in Figure 11 and Figure 12 to substitute $f_1(z)$ and $f_0(z)$. The new cost function is

$$J(\theta) = C\sum_{i=1}^{m}\left(y^{(i)}cost1(\theta^{\mathsf{T}}x^{(i)}) + (1-y^{(i)})cost0(\theta^{\mathsf{T}}x^{(i)})\right) + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 \tag{34}$$

Note that for the reason of convention, m is dropped and rather than writing the function as $A+\lambda B$, we are now writing it as $CA+B$. C has the same effect as the original $\frac{1}{\lambda}$ when it comes to its effect on regularization.

Unlike logistic regression, in which $h_\theta(x)$ is interpreted as the probablity of $y=1$, SVM has the following hypothesis:

$$h_\theta(x) = \begin{cases} 1, \text{ if } \theta^{\mathsf{T}}x \geq 0 \\ 0, \text{ otherwise} \end{cases} \tag{35}$$

9.2 Large margin classification

It is easy to tell from the graphs of $cost1$ and $cost0$ that for a training example with $y = 1$, in order to minimize the cost function, we expect $\theta^T x \geq 1$(not just ≥ 0). If $y = 0$, we expect $\theta^T x \leq -1$(not just < 0). This builds an extra "safety margin factor" for the SVM. Geometrically, in a linearly separable case, it is ralated to choosing the decision boundary that maximizes its distance from the examples, i.e. conducts the "large margin classification". Figure 13 demonstrates a simple case. Here, large margin classification results in the green line as the decision boundary, rather than the black one or the orange one.

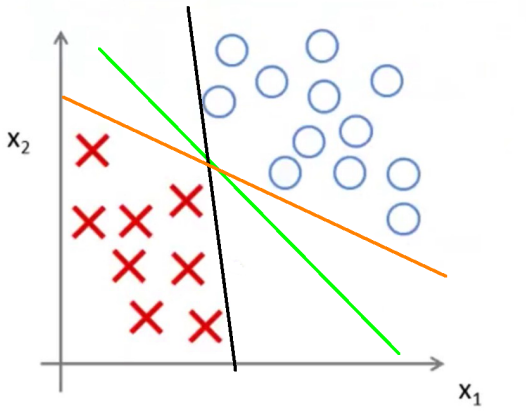

Figure 13: Intuition of large margin classification

When there exist outliners, the regularization factor C ensures that the algorithm does not overfit the examples. Obviously C cannot be too large.

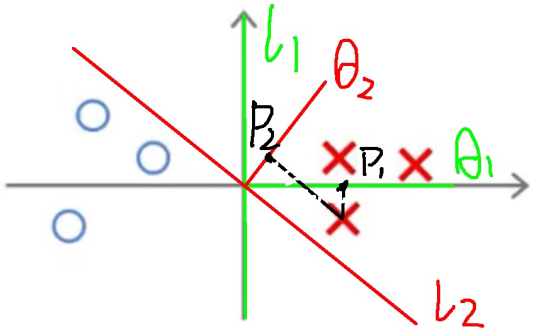

Figure 14: Mathematical background of large margin classification

The mathematical background of large margin classification can be illus-

trated by Figure 14. In this simple 2-d case, our target becomes

$$\begin{aligned} &\min_{\theta} \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 = \min_{\theta}\frac{1}{2}\|\theta\|^2 \\ &\text{s.t.} \begin{cases} \theta^{\mathsf{T}}x^{(i)} \geq 1, \text{ if } y^{(i)} = 1 \\ \theta^{\mathsf{T}}x^{(i)} \leq -1, \text{ if } y^{(i)} = 0 \end{cases} \end{aligned} \tag{36}$$

Note that

$$\theta^{\mathsf{T}}x = \|\theta\|\|x\|\cos\langle\theta, x\rangle = p \cdot \|\theta\|$$

in which p is the projection of x along the direction of θ. In order to minimize $\|\theta\|$, p should be as large as possible for all samples. From Figure 14, obviously l_1 is a better decision boundary because $p_1 > p_2$.

9.3 Kernals

In order to adapt SVMs to develop complex non-linear classifiers, we have to use **kernals**.

One way to develop non-linear classifiers is to use high degree polynomial features. We will end up with a classifier that predicts $y = 1$ if

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2 + \cdots > 0 \tag{37}$$

(37) can also be written as

$$\theta^{\mathsf{T}} f > 0$$

in which

$$\begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ \cdots \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \\ \cdots \end{bmatrix}$$

In complex cases, there could be a lot of polynomial features, causing significant computational difficulty. Our target is to find a better choice of features $f_1, f_2, f_3 \ldots$

Kernal introduces the idea to compute features of an example x according to its proximity to a series of landmarks $l^{(i)}$, e.g.

$$f_i = \text{similarity}\left(x, l^{(i)}\right) = \exp\left(-\frac{\|x - l^{(i)}\|^2}{2\sigma^2}\right) \tag{38}$$

Here we are using **Gaussian Kernal**. When x is close to $l^{(i)}$, this kernal returns approximately 1, while when x is far from $l^{(i)}$, it returns approximately 0.

With features f_i defined as such, we now have SVMs that can conduct non-linear classification. The SVM predicts $y = 1$ when $\theta^{\mathsf{T}} f \geq 0$, and $y = 0$ otherwise.

As for the choice of landmarks $l^{(i)}$, in practice, we use all examples in the training set as landmarks. Thus we will end up with m features. θ can be trained with

$$\min_{\theta} C\sum_{i=1}^{m}\Big(y^{(i)}cost1(\theta^{\mathsf{T}}f^{(i)}) + (1-y^{(i)})cost0(\theta^{\mathsf{T}}f^{(i)})\Big) + \frac{1}{2}\sum_{i=1}^{m}\theta_i^2 \tag{39}$$

Since we have exactly m features, now we have $n = m$.

As mentioned above, C controls the regularization in the same way as $\frac{1}{\lambda}$ before. Thus with large C, the SVM tends to have high variance and low bias, whereas with small C, it tends to have high bias and low variance. The σ in the definition of Gaussion kernal also affects the bias and variance of the SVM. With large σ, features vary more smoothly, and the SVM tends to have high bias and low variance, while with small σ, features vary more rapidly, and it tends to have high variance and low bias.

When using Gaussian kernal, it is important to do feature scaling. Otherwise $\|x-l\|^2$ will be dominated by the component with largest magnitude.

9.4 Logistic regression v.s. SVM

Some guidelines about the choice between logistic regression and SVM:

- When n is large and m is small: use logistic regression or SVM without a kernal (linear kernal).
- When n is small and m is large: add/create more features and then use logistic regression or SVM without a kernal.
- When n is small and m is intermediate: use SVM with Gaussian kernal.

10 Clustering

Unlike supervised learning, in which the training examples are (x, y) pairs, the training set of unsupervised learning contains only x without y label. The first type of unsupervised learning problem that we will introduce is clustering. The target of clustering is to group the training examples $\{x^{(1)}, x^{(2)} \dots x^{(m)}\}$ into a few clusters. Clustering is widely applied in scientific research and industrial practice, such as market segmentation, social network analysis, computing clusters organization as well as astronomial data analysis.

10.1 K-means algorithm

The most popular and the most widely used clustering algorithm is K-means. It takes two inputs: the training set $\{x^{(1)}, x^{(2)} \dots x^{(m)}\}$ and the number of expected clusters K. Here training example $x^{(i)} \in \mathbb{R}^n$. By convention, the $x^0 = 1$ that we used in supervised learning is dropped. K-means is an iterative algorithm that can be described as follows.

1. K cluster centroids $\mu_1, \mu_2 \dots \mu_K (\in \mathbb{R}^n)$ are randomly initialized.
2. Repeat {

   Cluster assignment:

   for $i = 1 : m$, $c^{(i)} :=$ index of cluster centroid closest to $x^{(i)}$, i.e. k that minimizes $\|x^{(i)} - \mu_k\|$.

   Move centroids:

   for $k = 1 : K$, $\mu_k :=$ average(mean) of all samples assigned to cluster k, i.e. $\mu_k := \text{AVG}\{x^{(i)} | c^{(i)} = k\}$.

   }

10.2 Optimization objective

As in supervised learning, the optimisation objective of K-means algorithm is the minimization of a cost function:

$$J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K) = \frac{1}{m} \sum_{i=1}^{m} \|x^{(i)} - \mu_{c^{(i)}}\|^2 \tag{40}$$

This cost function is also called the **distortion function**. Obviously, the cluster assignment process could be interpreted as minimizing $J(c, \mu)$ with respect to c, while the move centroids process could be interpreted as minimizing $J(c, \mu)$ with respect to μ.

10.3 Random initialisation

In the first step of K-means algorithm, the K cluster centroids should be randomly initialized. Usually what we do is to randomly pick K training examples and set μ_k to the values of these training examples. Nonetheless, it is possible that the algorithm may end up with a local optima due to the initial choice of the centroids. In order to combat the problem, we should run K-means several times (50-1000) independently (with different random initialisations), calculate the distortion J for each of the clustering results, and choose the result with the smallest distortion.

10.4 Choice of K

The choice of K could be subtle in a practical clustering problem. "Elbow method" might work in some situations, but more often provides no optimal option. Usually, K-means clustering is run for some later/downstream purpose. The choice of K should aim at better serving the downstream purpose. For instance, if we are running K-means on height/weight data of potential customers of a T-shirt we intend to produce in order to figure out how to segment the customers into groups of different sizes, K should obviously be 3 (S, M, L) or 5(XS, S, M, L, XL), even if the choice of K is completely ambiguous at first sight of the data.

11 Dimensionality reduction

Sometimes we may wish to reduce the dimension of the data for some reason.

Dimensionality reduction could be useful for data compression. If all 3D samples are close to a 2D plane, we can project all samples on this plane, and the 3D data is compressed to 2D. In practice the compression could be huge. For example, in a 8-bit RGB image, each pixel requires 24 bits to store the RGB values, each of them being a 8-bit integer from 0 to 255. If we can cluster the RGB values of all pixels into 16 clusters, which could be carried out with K-means, the RGB values of each pixel could be substituted with the values of the centroid to which it is assigned, and we manage to code the image with 4 bits per pixel plus some overhead (RGB values of the 16 centroids) at the price of the loss of some details. Also, when trying to develop pattern recognization machine learning algorithms such as face detection, the training examples are often of high dimensionality (e.g. 1.6×10^4 dimensions for a 128×128 greyscale image). With proper dimensionality reduction preconditioning of the data, the scale of the data can be significantly reduced (maybe from 10000 dimensions to 100 dimensions), and the algorithm can be significantly accelerated without damaging the ability to solve the problem.

Another situation that calls for dimensionality reduction is when we want to visualize the data. Visualization of the data can sometimes provide some intuitive inspirations on the properties of the data set, but it requires that the data be reduced to 2D or 3D.

Principal component analysis (PCA) is the most popular algorithm for dimensionality reduction.

11.1 Formulation of PCA

Mathematically speaking, we have m n-dimension training examples $x^{(1)}, x^{(2)} \dots x^{(m)} (\in \mathbb{R}^n)$, and we hope to find k directions $u^{(1)}, u^{(2)} \dots u^{(k)} (\in \mathbb{R}^n)$ such that the projections of the training examples $x^{(1)}, x^{(2)} \dots x^{(m)}$ on the subspace spanned by $u^{(1)}, u^{(2)} \dots u^{(k)}$, which are denoted with $z^{(1)}, z^{(2)} \dots z^{(m)} (\in \mathbb{R}^k)$, minimize the projection error

$$Err(U_k) = \frac{1}{m} \sum_{i=1}^{m} \left\| U_k z^{(i)} - x^{(i)} \right\|^2 \tag{41}$$

in which

$$U_k = \left[u^{(1)}, u^{(2)}, \dots, u^{(k)} \right].$$

Note that $z^{(i)}$ could be obtained by

$$z^{(i)} = U_k^{\mathsf{T}} x^{(i)}$$

if we require $u^{(i)}$ to be orthogonal and normalized. Inversely, an approximation of $x^{(i)}$ can be obtained with

$$x_{\text{approx}}^{(i)} = U_k z^{(i)}.$$

11.2 Implementation of PCA

Before applying PCA, the data needs to be preconditioned with data scaling and mean normalization, i.e. x_j should be substituted with $\frac{x_j-\mu_j}{\sigma_j}$, in which μ_j is the mean of x_j and σ_j is its standard deviation.

After the preconditioning of the data, we need to calculate the covariance matrix

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}{x^{(i)}}^{\mathsf{T}}. \tag{42}$$

Then we carry out sigular value decomposition of Σ, which is a builtin function of Matlab

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

We end up with a matrix U that contains all eigen vectors of Σ as its columns. The U_k we want is simply its first k colums.

11.3 Mathematics of SVD

11.4 Choice of k

Typically we choose k to be the smallest value ensuring

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\left\|x_{\text{approx}}^{(i)}-x^{(i)}\right\|^2}{\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)}\right\|^2} \le t \tag{43}$$

in which the threshold t could be 1%, 5%, 10%, etc for different purposes. If $t = 1\%$, we say 99% of variance is retained.

Intuitively, what we should do is to choose the threshold we need, and start from the PCA with $k = 1$. If the final result does not satisfy (43), we go to $k = 2$, and so on. However, the matrix S obtained in svd provides us with a better approach. S is a diagonal matrix that satisfies, for a specific k,

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\left\|x_{\text{approx}}^{(i)}-x^{(i)}\right\|^2}{\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)}\right\|^2} = 1-\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{m} S_{ii}}. \tag{44}$$

This significantly simplifies our calculation.

11.5 Good use v.s. bad use

PCA helps to compress the data, which sometimes helps machine learning algorithms to run faster. It is necessary to reduce the data to 2D or 3D when we want to visulize the data. These are good use of PCA.

PCA helps to reduce the dimensionality of the data, and fewer features bring smaller possibility of overfitting. But PCA is not a good way to address overfitting. Regularization is always a better option.

PCA helps to accelerate machine learning algorithms, but sometimes it is unnecessary because the algorithm could have run satisfactorily fast with the raw data. It is always wise and worthwhile to give it a try with the raw data before implementing PCA. If the algorithm runs too slowly or exhausts the storage (memory/disk) so that the task is unlikely to be completed with the raw data, it would be time to turn to PCA.

12 Anomaly detection

12.1 Introduction

Anomaly detection is an unsupervised learning problem that has some aspects similar to supervised learning problem.

Given m normal(non-anomalous) examples $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$, we are supposed to tell whether a new example x_{test} is anomalous. The approach we will take is to build a density model $p(x)$ and choose a threshold ϵ. If $p(x_{test}) \geq \epsilon$, the new example x_{test} is flagged normal; otherwise it is recognized as an anomaly.

Anomaly detection could be used in fraud detection. Also it can be used to decide whether a product is up to the normal quality standard in manufacturing. What's more, it can be used to monitor the status of computers in a data center.

12.2 Algorithm

Suppose the training examples $x^{(i)} \in \mathbb{R}^n$. We will assume that

$$p(x) = \prod_{i=1}^{n} p(x_i; \mu_i, \sigma_i^2)$$

in which $p(x_i; \mu_i, \sigma_i^2)$ means $x_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$. We are actually taking for granted that different features x_i are independent. Although this is not always true, it does not harm the correctness of the algorithm.

First we will pick out the features x_i that might be indicative of anomalous examples. Then we fit the parameters μ_i, σ_i^2 with

$$\begin{aligned} \mu_j &= \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)} \\ \sigma_j^2 &= \frac{1}{m} \sum_{i=1}^{m} (x_j^{(i)} - \mu_j)^2. \end{aligned} \tag{45}$$

Now that we have the model

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{(2\pi)}\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right), \tag{46}$$

we can calculate $p(x_{test})$ for any new example x_{test} and tell whether it is an anomaly.

12.3 Developing and evaluating an anomaly detection system

Assume we have some labeled data with anomalous and non-anomalous examples (y=0 normal, y=1 anomalous). In order to choose an superior anomaly detection algorithm, we will divide the training examples into, as usual, training set, cross-validation set and test set. Note that in this case, the training set contains only normal examples, while the cv set and the test set contain both normal and anomalous examples.

In a typical case, if we have 10000 normal examples and 20 anomalies, we can put 6000 normal examples into the training set, 2000 normal exmaples and 10 anomalies into the cv set, and 2000 normal examples and 10 anomalies into the test set.

First, we will fit the model $p(x)$ with the training set $\{x^{(1)}, \ldots, x^{(m)}\}$. Then, we will make predictions for the examples in the cv set with

$$y = \begin{cases} 1 \text{ if } p(x) < \epsilon \text{ (anomaly)} \\ 0 \text{ if } p(x) \geq \epsilon \text{ (normal)} \end{cases}. \tag{47}$$

Since the data is quite skewed (most examples are normal), predication accuracy is not a good evaluation metric. We can calculate the numbers of true positive, false positive, true negative, false negative, and then calculate **Precision/Recall**, and finally evaluate the algorithm with the F_1 score.

We will choose the value of ϵ that privides the best F_1 score on the cv set. Then the algorithm can be evaluated by applying the model on the test set.

12.4 Anomaly detection v.s. supervised learning

In the section above, we labeled anomalies with $y = 1$ in order to properly evaluate the algorithm. This could be confusing since anomaly detection is an unsupervised learning algorithm, while labeling examples is what we do in supervised learning. It seems that the problem could be solved directly with supervised learning methods. Here are some guidelines helping us to decide whether to use anomaly detection or supervised learning when facing a specific problem.

1. Anomaly detection should be used when the number of positive examples is very small while the number of negative examples is large. Supervised learning should be used when there are large number of of both positive and negative examples.

2. Anomaly detection should be used when there are many "types" of anomalies, i.e. it seems unlikely that we are able to predict what future anomalies will look like based on the anomalies that have appeared. Supervised

learning should be used when there are enough positive examples to help us get a sense of what an anomaly should "look like", i.e. we are confident that future anomalies will be similar to the ones we have seen.

12.5 Choosing features to use

The choice of features to be used in anomaly detection can be ambiguous and subtle.

When a feature does not "look gaussian" according to its histogram, we can make appropriate transformations to help it become gaussian. For instance, use $log(x), log(x+1), x^{1/2}, x^{1/3}$, etc instead of x.

When the current choice of features fails to differentiate an anomaly from normal examples, we should examine the anomaly and try to come up with a new feature from it and add it to the model.

A useful general principle for the choice of features in anomaly detection is to choose features that are likely to take unusually large or small values in the case of an anomaly.

12.6 Multivariate Gaussian distribution

As mentioned above, we have been taking for granted that $p(x_i)$ are independent from each other. Instead of using $p(x) = \prod_{i=1}^{n} p(x_i)$, we can model $p(x)$ all in one go with multvariate Gaussian distribution:

$$p(x;\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}\left|\Sigma\right|^{1/2}} \exp\left(\frac{1}{2}(x-\mu)^{\mathsf{T}}\Sigma^{-1}(x-\mu)\right) \tag{48}$$

in which Σ is the covariance matrix. This is a better approach when the features chosen are correlated to each other closely.

When using multvariate Gaussian distribution, when given a training set $\{x^{(1)},\ldots,x^{(m)}\}$, we can fit the model with

$$\begin{aligned} \mu =& \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \\ \Sigma =& \frac{1}{m}\sum_{i=1}^{m}(x^{(i)}-\mu)(x^{(i)}-\mu)^{\mathsf{T}}. \end{aligned} \tag{49}$$

When given a new example, we should calculate

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

and flag an anomaly if it turns out that $p(x) < \epsilon$.

It is not difficult to figure out that the original model used above is actually the multvariate gaussian model with

$$\Sigma = \begin{bmatrix} \sigma_1^2 & & & \\ & \sigma_2^2 & & \\ & & \ddots & \\ & & & \sigma_n^2 \end{bmatrix}. \tag{50}$$

When using the original model, we have to manually create extra features to capture anomalies that involve unusual combinations of existing features, while mulvariate gaussian model automatically captures the correlations between features. However, we should be cautious to use mulvariate gaussian when there are a lot of features because it is computationally more expensive (calculation of Σ^{-1}). What's more, mulvariate gaussian requires $m > n$ to ensure the invertibility of Σ. In practice, it should only be used when m is sufficiently larger than n, e.g. $m > 10n$.

13 Recommender systems

13.1 Introduction

A recommender system aims at recommending for its users what can promisingly be of interest for them, e.g. books for customers visiting an online bookstore like Amazon, movies for a user of an online move database like IMDB, people that the user might want to add as friends on a social network like Facebook, etc. It is an important application of machine learning. It is interesting for us because it is one of the algorithms that do not call for manual choice of features.

We will illustrate the components of general recommender system with a movie recommending system as an example. The system recommends movies for users based on ratings given by all users in the database as well as the user's personal rating history. We will denote the total number of movies in the database with n_m and the total number of users with n_u. We will denote $r(i,j) = 1$ if the user j has rated movie i, and $r(i,j) = 0$ otherwise. In the case $r(i,j) = 1$, the rating of movie i given by user j will be denoted as $y^{(i,j)}$. The number of movies that have been rated by user j is denoted with $m^{(j)}$, i.e. $\sum_i r(i,j) = m^{(j)}$.

In order to make correct recommendation, the system will try to predict a user's rating for movies that he has not rated based on his personal taste, which is indicated by his rating history, as well as the category of the movie, which is indicated by other user's rating of the movie. If there are n features to describe movies, we can use an n dimension vector $\theta^{(j)}$ to represent user j's taste, and an n dimension vector $x^{(i)}$ to represent features of movie i. The rating of movie i given by user j could then be predicted with ${\theta^{(j)}}^\mathsf{T} x^{(i)}$.

Suppose we already have the $x^{(i)}$ vectors, then we can learn $\theta^{(j)}$ by minimizing the cost function

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

which could be conducted with gradient descent. Note that by convention, $\theta^{(j)}$ and $x^{(i)}$ does not contain bias component (the 0th component).

13.2 Collaborative filtering algorithm

In the previous section, we assumed that the $x^{(i)}$ vectors are already know, which is unrealistic in practice. Thus, rather than just learning $\theta^{(j)}$, we also have to learn $x^{(i)}$. Now the cost function to be minimized becomes

$$\begin{aligned} J(x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)}) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} \left({\theta^{(j)}}^\mathsf{T} x^{(i)} - y^{(i,j)} \right)^2 \\ + \frac{\lambda}{2} \sum_{j=1}^{n_u} {\theta^{(j)}}^\mathsf{T} \theta^{(j)} + \frac{\lambda}{2} \sum_{i=1}^{n_m} {x^{(i)}}^\mathsf{T} x^{(i)}. \end{aligned} \tag{51}$$

It can be minimized using gradient descent:

$$\begin{aligned} x^{(i)} &:= x^{(i)} - \alpha \left(\sum_{j:r(i,j)=1} \left({\theta^{(j)}}^\mathsf{T} x^{(i)} - y^{(i,j)} \right) \theta^{(j)} + \lambda x^{(i)} \right) \\ \theta^{(j)} &:= \theta^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} \left({\theta^{(j)}}^\mathsf{T} x^{(i)} - y^{(i,j)} \right) x^{(i)} + \lambda \theta^{(j)} \right) \end{aligned} \tag{52}$$

The algorithm is called collaborative filtering algorithm. After we have learned the parameters, we can use them to predict users' ratings for movies. Note that before minimizing the cost function, we have to initialize $x^{(i)}$ and $\theta^{(j)}$ to random small values in order to break the symmetry.

The algorithm could be written in a vectorized way. We will use two matrices X and Θ defined as follows:

$$X = \begin{bmatrix} —{x^{(1)}}^\mathsf{T}— \\ —{x^{(2)}}^\mathsf{T}— \\ \dots \\ —{x^{(n_m)}}^\mathsf{T}— \end{bmatrix}, \Theta = \begin{bmatrix} —{\theta^{(1)}}^\mathsf{T}— \\ —{\theta^{(2)}}^\mathsf{T}— \\ \dots \\ —{\theta^{(n_u)}}^\mathsf{T}— \end{bmatrix}. \tag{53}$$

Then the $n_m \times n_u$ matrix that stores the prediction of the ratings of all movies by all users could simply be expressed by $X\Theta^\mathsf{T}$. This is called **low rank matrix factorization**. In Matlab, the vectorized implementation of the cost function and its derivatives is as follows:

```
J = 0.5 * sum(sum(((X * Theta' - Y) .* R) .^ 2)) + ...
    0.5 * lambda * (sum(sum(Theta .^ 2)) + sum(sum(X .^ 2)));
X_grad = ((X * Theta' - Y) .* R) * Theta + lambda * X;
Theta_grad = ((X * Theta' - Y) .* R)' * X + lambda * Theta;
```

With $x^{(i)}$ and $\theta^{(j)}$ learned, we can recommend unwatched movies for a specific user: simply choose the movies that he has not rated with the highest predicted ratings. We can also find movies that are the most "similar" to a given movie i: simply find movies j with the smallest $\|x^{(i)} - x^{(j)}\|$.

13.3 Mean normalization

If a user has never rated any movie, i.e. $r(i,j) = 0$ for all i, we will end up with $\theta^{(j)}$ being a vector with all elements equal to 0. As a result, the prediction for ratings of all movies by the user will be 0, which is not helpful at all for the recommendation. A reasonable choice is to use the average rating of other users as the prediction for this user. In order to accomplish this, we have to implement **mean normalization**.

For each movie, we will calculate its average rating, and store the result in vector μ. Then we will substract the average rating from all available rating results, and use the substracted results in the algorithm. Finally, the prediction of user j's rating for movie i will be ${\theta^{(j)}}^{\mathsf{T}} x^{(i)} + \mu_i$. This accomplishes mean normalization because it ensures that for a user without any rating history (thus $\theta^{(j)} = 0$), the prediction is the average rating μ_i, while the predictions for other users remain untouched.

As an example, if the rating data we have is $Y = \begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & 0 & ? & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{bmatrix}$, then $\mu = \begin{bmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix}$, and Y should be changed to $Y = \begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 & ? \\ 2.5 & ? & -2.5 & ? & ? \\ ? & 2 & -2 & ? & ? \\ -2.25 & -2.25 & 2.75 & 1.75 & ? \\ -1.25 & -1.25 & 3.75 & -1.25 & ? \end{bmatrix}$.

14 Large scale machine learning

14.1 Problem with large scale data

Take linear regression as an example. When we use gradient descent to learn the parameters, what happens to the parameters in each iteration is

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}, \ j = 0, \ldots, n, \tag{54}$$

in which m is the number of examples in the training set. If the scale of the training set is very large, say $m = 10^9$, we will have to do 10^9 additions in each single step of gradient descent, which is a problem that needs to be addressed, otherwise the large amount of computation will make it impossible to solve the problem in reasonable time. One way to avoid such problem is to check if the problem could be solved with a small training set, say $m = 1000$. If the learing curve shows that the problem does not have high variance (overfit) with the small training set, it might be unnecessary to use the large traing set.

14.2 Stochastic gradient descent

The traditional gradient descent (54) is called **batch gradient descent** because it uses all examples at each iteration. Batch gradient descent ensures that the parameters get closer to the optima at each iteration.

In **stochastic gradient descent**, the examples are first shuffled. Then we will repeat

$$\theta_j \coloneqq \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)},\ j = 0, \ldots, n \tag{55}$$

for $i = 1, \ldots, m$. Rather than altering the value of θ_j after summing up items related to all examples as in batch gradient descent, θ_j is altered a little bit for each example. It is not guaranteed that θ_j gets closer to its optimum after each alteration. Actually θ_j ends up wandering around the optimum continually in a small region. In practice, this is enough to ensure the convergence of the algorithm because the parameters are satisfactory so long as they are close enough to the optima. And in general stochastic gradient descent converges much faster than batch gradient descent. Typically all examples need to be passed 1 to 10 times to obtain a satisfactory result.

14.3 Mini batch gradient descent

Batch gradient descent uses all m examples in each iteration, while in stochastic gradient descent, only 1 example is used in each iteration. What mini batch graident descent does it somewhere in between: b examples are used in each iteration:

$$\theta_j \coloneqq \theta_j - \alpha\frac{1}{b}\sum_{k=i}^{i+b-1}(h_\theta(x^{(k)}) - y^{(k)})x_j^{(k)} \tag{56}$$

b is called the mini batch size. Typically b could be from 2 to 100.

With a judicious choice of b, mini batch gradient descent could run even faster than stochastic gradient descent.

14.4 Convergence

With batch gradient descent, we check the convergence of the algorithm by calculating $J_{train}(\theta)$ after each iteration. This does not make sense for stochastic gradient descent or mini batch gradient descent because the calculation of

$J_{train}(\theta)$ leads to the computation load that we are trying to get rid of due to training set size.

What we can do is to calculate $cost(\theta, (x^{(i)}, y^{(i)}) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$ before updating θ using $(x^{(i)}, y^{(i)})$ at each iteration. Then every 1000 iterations (just an example), we can plot $cost(\theta, (x^{(i)}, y^{(i)})$ averaged over the last 1000 examples processed by the algorithm to see the trend of the cost function, and adjust value of α accordingly.

Learning rate α is usually kept constant. We could choose to use a learning rate that gradually diminishes over time, say $\alpha = \frac{const1}{const2+iterationNumber}$ to make θ_j get closer to the optima in the end, but this will introduce two more paramters for us to tune, and is not always a good idea.

14.5 Online learning

When a continuous stream of data is available, e.g. user searching action on an online shopping website, we can use the mechanism of online learning. Rather than collecting a lot of examples as a traing set and use machine learning algorithms to learning the best parameters, we can adjust the parameters a little bit immediately after obtaining a new example, like we did in stochastic gradient descent, and discard the example without having to store it for further use. Besides facilitating treatment of continuous stream of data, online learning also makes it possible to adapt our model according to changing user preferences.

14.6 Map/reduce and parallelism

The map/reduce approach to large scale machine learning takes advantage of distributed computation. Computation involving addtion of all examples can be divided into a few parts and be sent to a series of machines to conduct the computation, and then the results on all machines are brought together to obtain the final result. Even if we do not have a series of machines available, it is still possible to benefit from the acceleration of parallelism because most modern computers have a few cores that can conduct computation tasks independently.

15 Example: photo OCR