Machine Learning

Lectured by Andrew Ng on Coursera

2017

1 Introduction

Definition (Machine Learning - Tom Mitchell, 1998)**.** A computer program is said to *learn* from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

2 Linear Regression

Notation (Training data)**.** We have a set of training examples, denoted by $x \in \mathbb{R}^n \times \mathbb{R}^m$ (m examples of n features). Concretely, $x^{(i)}$ denotes the features of the i^{th} training example, with $x_j^{(i)}$ being the value of feature j in the i^{th} training example.

Conventionally, we take an extra row of ones as the 0^{th} feature ($x_0 := 1$) and denote the corresponding $(m+1) \times n$ matrix X the *design matrix*.

The output variable is $y \in \mathbb{R}^m$.

Definition (Linear hypothesis)**.** Our hypothesis is

$$h_\theta(x) = \theta^T X$$

where $\theta \in \mathbb{R}^{n+1}$ is a vector of parameters to be estimated.

Definition (Squared Loss Cost Function)**.** The *cost function*

$$\begin{aligned} J(\theta) &= \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^i) - y^i\right)^2 \\ &= \frac{1}{2m} (\theta^T X - y)^T (\theta^T X - y) \end{aligned}$$

represents a measure of the fit of a particular choice of parameters θ.

We estimate θ as the value minimising $J(\theta)$.

2.1 Gradient Descent

Gradient descent involves following the gradient of this cost function to find its minimum. In practice, taking an initial estimate θ_0 and *learning rate* α we iteratively compute

$$\theta := \theta - \alpha \boldsymbol{\nabla} J$$

that is

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

for $j \in \{0, \ldots, n+1\}$.

For linear regression,

$$\begin{aligned} \frac{\partial}{\partial \theta_j} J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^i) - y^i \right) x_j^{(i)} \\ &= \frac{1}{m} X(\theta^T X - y) \end{aligned}$$

We declare convergence when $J(\theta)$ decreases by less than ϵ in a single iteration.

The learning rate α must be chosen carefully. Too large and convergence may not occur, too small and convergence will be slow.

2.1.1 Feature scaling

We can speed up gradient descent via *feature scaling* and *mean normalization*. Intuitively, this involves scaling each feature so that the steps taken along the gradient are roughly uniform across the dimensions of the feature space. Concretely, we set

$$x = \frac{x - \mu}{\sigma}$$

where μ is the (row-wise) mean of x and σ is the (row-wise) standard deviation of x.

2.2 Analytic solution

The minimum of the cost function can also be found analytically (solve the system of equations $\boldsymbol{\nabla} J = 0$ to obtain the exact solution

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

in the cast where $(X^T X)$ is invertible. Even in the singular case, we can take a numerical solution with the pseudo-inverse, e.g. via the Octave function `pinv`. The singular case can occur when there are redundant (linearly dependent) features, or too many features ($m \leq n$).

3 Logistic Regression - Classification

We now restrict $y \in \{0, 1\}^m$ so that we now have the problem of *classifying* an observation x.

Definition (Classification hypothesis)**.** Our hypothesis is

$$\begin{aligned} h_\theta(x) &= g(\theta^T x) \\ &= \frac{1}{1 + e^{-\theta^T x}} \end{aligned}$$

where $g(z)$ is the *sigmoid* function, so that $0 \leq h_\theta(x) \leq 1$. We interpret

$$\begin{aligned} h_\theta(x) &= \mathbb{P}_\theta(y = 1 \mid x) \\ &= \mathbb{P}(y = 1 \mid x, \theta) \end{aligned}$$

and 'predict' that $y = 1$ if $h_\theta(x) >= 0.5$, i.e. if $\theta^T x >= 0$. The surface $h_\theta(x) = 0.5$ is known as the *decision boundary*.

The squared loss cost function is not convex in the case of logistic regression, we instead use the logistic cost function.

Definition (Logistic Cost function)**.** The *logistic* cost function is

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

where

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

This form of cost function can be derived from *maximum likelihood estimation* for the binomial distribution.

Gradient descent applies in the same way, and moreover we again find that

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

though of course with the new hypothesis function.

3.0.1 Advanced optimization

There exist other, more complex algorithms to minimum the cost function, such as

(i) Conjugate gradient

(ii) BFGS

(iii) L-BFGS

which don't involve picking a learning rate and are often faster.

3.1 Multiclass classification

For the case of classifying $y \in \{1, \ldots, k\}$ we can apply the principle of *one-vs-all* to train k classifiers $h_\theta^{(i)}$ each of which is predicting the probability $\mathbb{P}_\theta(y = i \mid x)$ (against all other classes). The final prediction is then taken as the class i maximising $h_\theta^{(i)}$.

3.2 Addressing overfitting

With too many features, the learned hypothesis may fit the training set very well ($J(\theta) \approx 0$ but fail to generalize to new examples. This is known as *overfitting*.

This can be addressed by reducing the number of features (manually or alorithmically), or via regularization.

3.2.1 Regularization

The idea is too keep all features, but reduce the magnitude of the parameters θ_j. This works well when there are lots of features, each contributing a bit to predicting y.

To achieve this, we *penalize* the parameters inside the cost function.

For *linear regression*, we have

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

N.B. we conventionally do not penalize θ_0, so it is excluded from the sum.

The analytic solution is then

$$\theta = \left(X^TX + \lambda\begin{bmatrix}0 & & & \\ & 1 & & \\ & & 1 & \\ & & & \ddots \\ & & & & 1\end{bmatrix}\right)^{-1} X^Ty$$

which does not suffer from the problem of non-invertibility.

For *logistic regression*, we have

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

As with the learning rate, λ must be appropriately chosen: too small and the regulation will have little effect, but too large will lead to *underfitting*!

4 Neural Networks

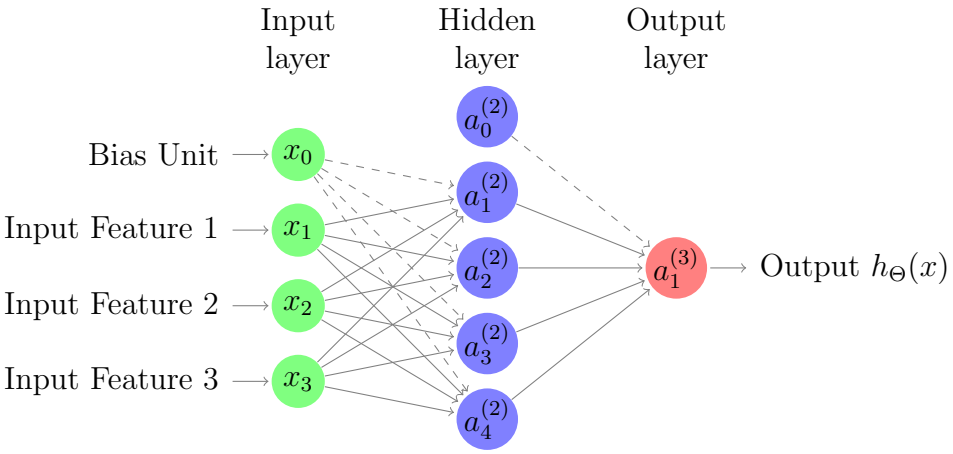
The neuron model mimics the biological neuron: depending on some *activation function* of the inputs (via *dendrites*), the unit outputs (via *axon*) some value (*signal*). The *neural network* simply connects such units in a sequence of layers, where the output of the unit in one layer is input for each unit in the next layer.

The activation function is the sigmoid logistic function $g(z) = \frac{1}{1+e^{-z}}$. Thus, each layer is a series of logistic regression models trained on the previous layer (to which we also add a *bias unit* of constant value 1).

The first layer is the actual *input* to the model, the last layer the *output*, while the intermediate layers are known as *hidden layers*. See Figure 1[1].

[1] adapted from http://www.texample.net/tikz/examples/neural-network/

Figure 1: Representation of Neural Network



4.1 Forward propagation

Forward propagation describes the method by which, given a trained neural network, we can compute predictions.

Consider a neural network with L layers, with s_j units in layer j.

Let $x_1, \ldots, x_{s_0}$ be the training data, to which we add $x_0 := 1$. Set $a^{(1)} := x$.

Let $\Theta^{(j)}$ be the parameters (weights) of the mapping between layers j and $(j+1)$, so that the i^{th} row of $\Theta^{(j)}$ is the logistic regression parameters for the i^{th} unit in layer $(j+1)$. Thus $\Theta^{(j)}$ is of dimension $s_{j+1} \times (s_j + 1)$.

The *activations* of the units in layer j is given by the s_j dimensional vector $a^{(j)}$ where

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

and at each layer we also add the bias unit $a_0^{(j)} := 0$. The output is $h_\Theta(x) = a^{(L)}$ (which may be multi-dimensional).

4.2 Multiclass classification

Given labels $\{1, \ldots, K\}$, we model label k as the unit coordinate vector for dimension k, i.e. a vector with 1 in dimension k, and 0 in all other dimensions.

For example, to represent $K = 3$ classes, we would use

$$1 \to \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, 2 \to \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \text{ and } 3 \to \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

4.3 Examples of neural networks

As illustration of how neural networks can represent more advanced models than simple linear or logistic regression, we consider (approximate) representations of well-known binary operations.

Example (AND)**.** With $x_1, x_2 \in \{0, 1\}$ and $y = x_1 \wedge x_2$.

We use a neural network with no hidden layer, and one output unit (i.e. simple logistic regression!). Take $\Theta^{(1)} = \begin{pmatrix} -30 & 20 & 20 \end{pmatrix}$.

Figure 2: XNOR Neural Network

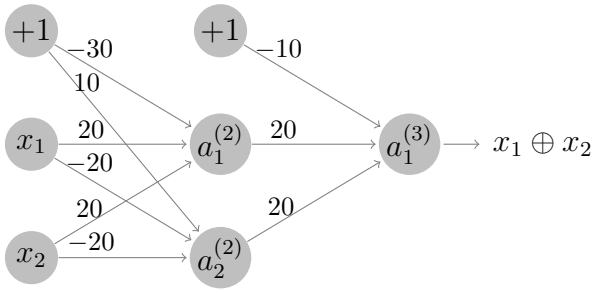


Table 1: XNOR neural network

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

Example (OR)**.** With notation above, but to represent $y = x_1 \vee x_2$, we can take $\Theta^{(1)} = \begin{pmatrix} -10 & 20 & 20 \end{pmatrix}$.

Example (NOT)**.** With $x_1 \in \{0, 1\}$ and $y = \neg x_1$, we can take $\Theta^{(1)} = \begin{pmatrix} 10 & -20 \end{pmatrix}$.

Example (XNOR)**.** We can represent $y = \neg(x_1 \oplus x_2)$, by combining the above examples into a neural networks with one hidden layer, using

$$\Theta^{(1)} = \begin{pmatrix} -30 & 20 & 20 \\ 10 & -20 & -20 \end{pmatrix}$$

and

$$\Theta^{(2)} = \begin{pmatrix} -10 & 20 & 20 \end{pmatrix}$$

which is taking the hidden layer activations as $a_1^{(2)} = x_1 \wedge x_2$ and $a_2^{(2)} = (\neg x_1) \wedge (\neg x_2)$, and the output is $y = h_\Theta(x) = a_1^{(3)} = a_1^{(2)} \vee a_2^{(2)}$. See Figure 2 and Table 1.

4.4 BackpropAgation

Definition (Cost function for Neural Network)**.** Suppose we have m training examples $x^{(i)}, \ldots, x^{(m)}$, and K classes (so we represent the observed outputs as $y_i \in \{0, 1\}^K$, as described in §4.2). Then the cost function is given by

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

where $h_\Theta(x^{(i)})$ is the prediction as obtained by forward-propagation (described in §4.1) for training example i, and λ is the *regularization parameter* (as in §3.2.1).

The method of *backpropagation* describes a way to compute the gradients $\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l)}}$ so that we can apply an optimization algorithm (such as gradient-descent) to minimize $J(\Theta)$.

Intuitively, we run a forward pass (§4.1) to compute the activations throughout the network, then for each node j in layer l we compute an *error term* $\delta_j^{(l)}$ that measures how much that node was 'responsible' for errors in the output. For the output layer, this can simply be the distance between $h(x)$ and y, while for the hidden layers we can take a weighted (by Θ) averaged of the errors in the following layer.

Definition (Backpropagation Algorithm)**.** For a given training example $x^{(t)}$, we compute the gradient as follows:

(i) Set the input layer $a^{(1)} := x^{(t)}$.

(ii) Perform forward propagation to compute the activations $(z^{(j)}, a^{(j)})$ for layers $2, \dots, L$.

(iii) For the output layer L, compute $\delta^{(L)} = a^{(L)} - y$, where y is as in §4.2.

(iv) For layers $l = L-1, \dots, 2$, compute $\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} \odot g'(z^{(l)})$, where $g'(z^{(l)}) = a^{(l)} \odot (1 - a^{(l)})$. [2]

(v) Accumulate $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$.

Finally, we have

$$\frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{(l)}} = \begin{cases} \frac{1}{m}\left(\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)}\right) & \text{for } j \neq 0 \\ \frac{1}{m}\Delta_{i,j}^{(l)} & \text{for } j = 0 \end{cases}$$

4.4.1 Gradient checking

It can be helpful in debugging to check the gradient computed in backpropagation against a numerical approximation, for example using the symmetric difference quotient $\frac{f(x+h)-f(x-h)}{2h}$ in each dimension of Θ, and comparing to the gradient from backpropagation which should be similar.

4.4.2 Random initialization

The initial parameters must be randomly initialized because if otherwise we started at $\Theta = 0$ uniformly then by symmetry the parameters in each layer would remain equal on every iteration.

[2] Here I am using the notation $\odot$ to mean element-wise multiplication (i.e. $.*$ in Matlab).