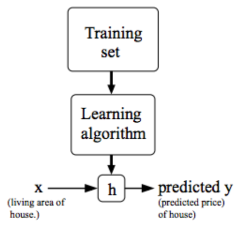
Machine Learning by Stanford University

Vocabulary

Machine Learning: Field of study that gives computers the ability to learn without being explicitly programmed [1].
Supervised Learning: In supervised learning, we are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output. Supervised learning problems are categorized into "regression" and "classification" problems. In a regression problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results in a discrete output.
Unsupervised Learning: Unsupervised learning allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables. We can derive this structure by clustering the data based on relationships among the variables in the data.

Linear regression



- n: number of features
- m: number of training set
- α: Learning Rate

Hypothesis:$h_\theta(x) = \theta^T x$ with $x_0 = 1$
Parameters:$\theta_0, \theta_1, ..., \theta_n$
Cost function:$J(\theta) = \frac{1}{2m}\sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$
Gradient descent:Repeat
$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$
simultaneously update for every j={0,...,n}
Feature Scaling: for fast convergence, Get every feature into approximately a $-1 \le x_i \le 1$ range

Normal Equation

$\theta = (X^T X)^{-1} X^T y$
There is no need to do feature scaling with the normal equation.

Comparaison

Gradient Descent	Normal Equation
Need to choose α	No need to choose α
Needs many iterations	No need to iterate
O(kn^2)	O(n^3) for $X^T X$
Works well when n is large	Slow if n is very large

Classification

Sigmoid function
$h_\theta(x) = p(y = 1/x;\theta) = \frac{1}{1+\exp -\theta^T x}$
Decision Boundary
$h_\theta(x) \ge 0.5 \Rightarrow y = 1 \quad h_\theta(x) < 0.5 \Rightarrow y = 0$
Logistic regression cost fonction
$J(\theta) = \frac{1}{m}\sum_{i=1}^m \mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)})$

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

$$\begin{cases} \mathrm{Cost}(h_\theta(x), y) = 0 & \text{if } h_\theta(x) = y \\ \mathrm{Cost}(h_\theta(x), y) \to \infty & \text{if } y = 0 \text{ and } h_\theta(x) \to 1 \\ \mathrm{Cost}(h_\theta(x), y) \to \infty & \text{if } y = 1 \text{ and } h_\theta(x) \to 0 \end{cases}$$

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

Gradient Descent Repeat
$\theta_j := \theta_j - \alpha \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
simultaneously update for every j={0,...,n}
Advanced Optimization Matlab

Listing 1: Function

```
function [jVal, gradient] = costFunction(
    theta)
  jVal = [...code to compute J(theta)
      ...];
  gradient = [...code to compute
      derivative of J(theta)...];
end
```

Listing 2: Fminunc

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

Multiclass Classification

One-vs-all
Train a logistic regression classifier $h_\theta(x)$ for each class i to predict the probability that $y = i$.
$h^i(\theta) = P(y = i/x;\theta)$
To make a prediction on a new x, pick the class i that maximizes $h_\theta(x)$.

Overfitting

There are two main options to address the issue of overfitting:

- Reduce the number of features.
- Regularization: Keep all the features, but reduce the magnitude of parameters θ_j

Regularization Cost Function

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

Gradient Descent

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

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

$$j \in \{1, 2...n\}\}$$

Normal Equation

$$\theta = \left(X^T X + \lambda \cdot L\right)^{-1} X^T y$$

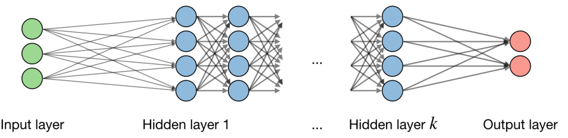
$$\text{where } L = \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}$$

If $m < n$, then $X^T X$ is non-invertible. However, when we add the term λL, then $X^T X + \lambda L$ becomes invertible.

Neural Networks I

Neural networks are a class of models that are built with layers.Commonly used types of neural networks include convolutional and recurrent neural networks.

Architecture



By noting i the i^{th} layer of the network and j the j^{th} hidden unit of the layer, we have:

$$z_j^{[i]} = w_j^{[i]^T} x + b_j^{[i]}$$

where we note w, b, z the weight, bias and output respectively.

$$h_\theta(x) = a^{(j+1)} = g(z^{(j+1)})$$

where $z^{(j+1)} = \Theta^{(j)} a^{(j)}$
If network has s_j units in layer j and s_{j+1} units in layer j+1, then $\Theta^{(j)}$ will be of dimension $s_{j+1} * s_j + 1$.

Cost Function

Let's first define a few variables that we will need to use:

- L = total number of layers in the network
- S_l = number of units (not counting bias unit) in layer l
- K = number of output units/classes

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

Back propagation Algorithm

Given training set $\{(x^{(1)}, y^{(1)}) \cdots (x^{(m)}, y^{(m)})\}$
Set $\Delta_{i,j}^{(l)}$:= 0 for all (l,i,j), (hence you end up having a matrix full of zeros)
For training example t =1 to m:

Neural Networks II

1. Set $a^{(1)} := x^{(t)}$
2. Perform forward propagation to compute $a^{(l)}$ for l=2,3,...,L
3. Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$
   Where L is our total number of layers and $a^{(L)}$ is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y. To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:
4. Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ using
   $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)})\ .*\ a^{(l)}\ .*\ (1 - a^{(l)})$
   The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by $z^{(l)}$.
5. $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)}\delta_i^{(l+1)}$ or with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$
   Hence we update our new Δ matrix.

   $$D_{i,j}^{(l)} := \frac{1}{m}\left(\Delta_{i,j}^{(l)} + \lambda\Theta_{i,j}^{(l)}\right), \text{ If }] \neq 0.$$
   $$D_{i,j}^{(l)} := \frac{1}{m}\Delta_{i,j}^{(l)} \text{ If } j = 0$$

   The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. Thus we get $\frac{\partial}{\partial\Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

Reference

References

[1] Machine Learning andrew ng. `https://www.coursera.org/learn/machine-learning`.