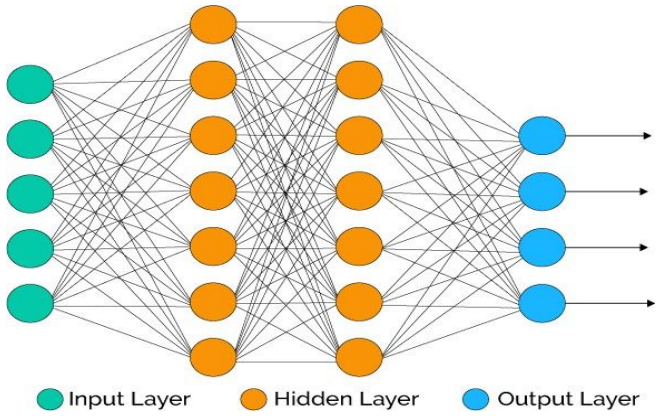
Neural Networks: Learning

Set Up

1. Let L be the number of layers in the network
2. Let s_l be the number of units in layer l
3. Let K be the number of output units

Recall that θ is a weight matrix, and in particular it is the weight matrix for the output layer.

Recall that each unit in a layer is actually a logistic regression unit.



Logistic Regression Cost Function

Logistic regression is a binary classification problem, in which we try to predict whether the response y is $\{0, 1\}$ based on predictors $x_1, \ldots, x_p$ by finding parameters $\beta_0, \ldots, \beta_p$ such that there is an approximated functional relationship between y and $\beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$

Goal: Given a training set, learn a regression function $h : X \to Y$ so that $h(x)$ is a "good" predictor for the corresponding value of y

Suppose you are given training samples $(\bar{x}_1, y_1), \ldots, (\bar{x}_n, y_n)$, where $\bar{x}_i$ is a vector of the form $\begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_p \end{bmatrix} \in \mathbb{R}^{p+1}$

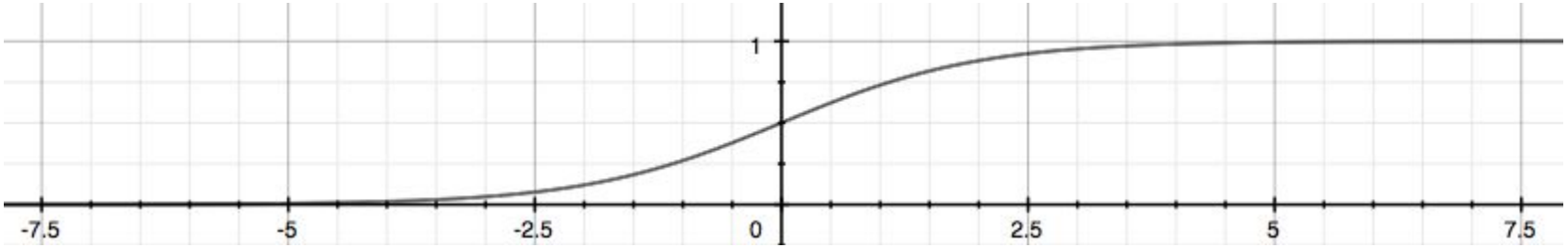
Let $\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix} \in \mathbb{R}^{p+1}$

Note that $\beta^T \cdot \bar{x}$ is precisely $\beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$

Since y is defined strictly from $\{0, 1\}$, define h_β as:

$h_\beta(x) = g(z)$, where $z = \beta^T x$ and $g(z) = \dfrac{1}{1 + e^{-z}}$

$g(z)$ is known as a sigmoid function



Is $h_\beta(x)$ a good predictor of y?

Define the cost function for h_β and a $\bar{x}$ by:

$$\text{Cost}(h_\beta(x), y) = -(y \ \log(h_\beta(x)) + (1-y)\log(1-h_\beta(x)))$$

Note that the output of $\log(h_\beta(x))$ is negative, which results in scaling by -1

It follows that the Cost Function for n training samples is:

$$J(\beta) = \frac{1}{n}\sum_{i=1}^{n} \text{Cost}(h_\beta(\bar{x}_i), y_i)$$

A vectorized implementation is the following:

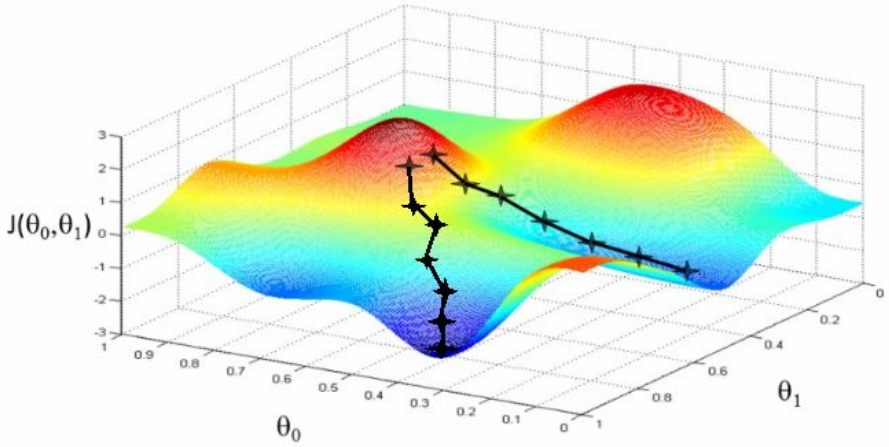
$$h = g(X\beta)$$
$$J(\beta) = \tfrac{1}{n} \cdot \left(-y^T \log(h) - (1-y)^T \log(1-h)\right)$$

Gradient Descent for Logistic Regression

Note that we are optimizing the logistic regression cost function via gradient descent, which is the typical way to optimize the neural network cost function.

$$\beta := \beta - \frac{\alpha}{n} X^T (g(X\beta) - \vec{y})$$

where α is some learning rate determined by the user.



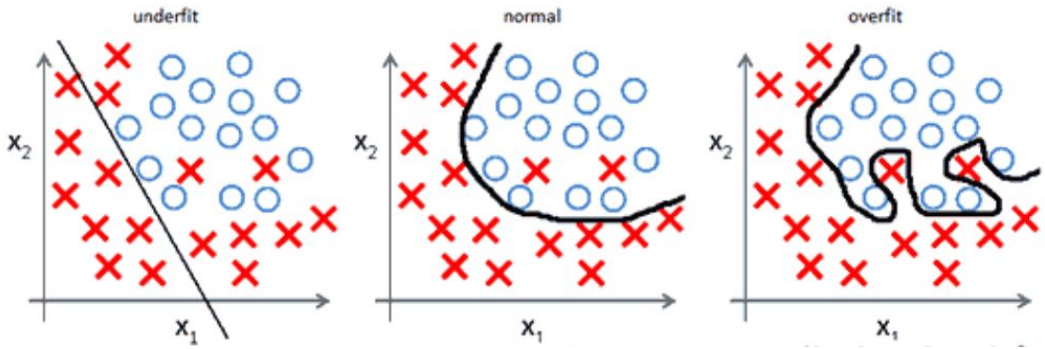
Addressing the Problem Overfitting

Overfitting, or high variance, is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

Overfitting can be addressed by a regularization parameter λ; If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost.

The regularized cost function of logistic regression is the following:

$$J(\beta) = \frac{1}{n}\sum_{i=1}^{n} \text{Cost}(h_\beta(\bar{x}_i), y_i) + \frac{\lambda}{2n}\sum_{j=1}^{n} \beta_j^2$$



Cost Function for Neural Networks

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\text{Cost}(h_\theta(\bar{x}_i), y_i)_k + \frac{\lambda}{2n}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\theta_{j,i}^{l},$$

where $\text{Cost}(h_\theta(\bar{x}_i), y_i)_k$ is the cost function for the k^{th} output node.

Recall that θ is the weight matrix of the final output layer, while θ^l is the matrix of weights for the function mapping the hidden layers l to $l+1$.

The double summation simply adds up the logistic regression costs calculated for each unit in the output layer.

The triple summation simply adds up the squares of all the individual θ_{ij}'s in the entire network.

Putting It All Together

First, choose a network architecture by determining the layout of the neural network, which includes the number of hidden units in each layer and the total number of layers.

We have the following:

- Number of input units is equal to the dimension of the features of $\bar{x}_i$, i.e. p
- Number of output units is the number of classes you are trying to classify
- Number of hidden units per layer, usually more is better (must balance with cost of computation as it increases with more hidden units)
- Defaults: 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

Training a Neural Network

1. Randomly initialize the weights
2. Calculate $h_\theta(\bar{x}_i)$ for all $\bar{x}_i$
3. Implement the cost function
4. Implement back propagation
5. Use gradient descent to minimize the cost function with the weights in theta.

Cost Function to minimize:

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

To find the Θ's that minimizes the cost function, we need to:

- compute J(Θ)
- compute d/dΘ

The Motivation:

By performing forward and backpropagation, we can compute d/dΘ by finding the contribution of each node to the error of the neural network in predicting *y*.

Notation

x: training set features
y: response variable
Θ: weights
L: total # of layers
$\mathbf{s_l}$: # of units in layer l
m: size of training set
K: # of classes (k=1 if binary)

Forward Propagation

Algorithm:

Given one training example (***x***,*y*)

- Start at layer 1 and compute the activation of each unit

For example, to compute $a_2^{(3)}$:

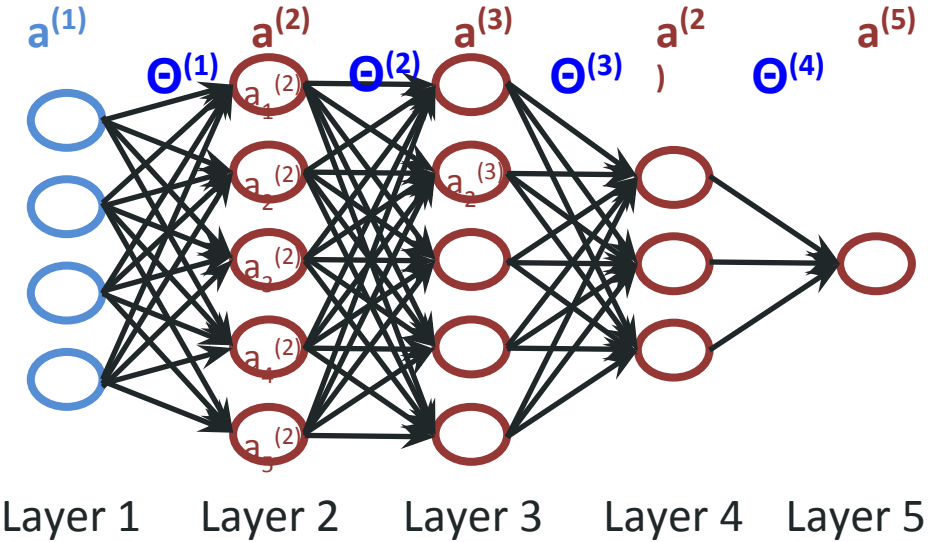
$$a_2^{(3)} = g\left(\Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \cdots + \Theta_{15}^{(2)} a_5^{(2)}\right)$$

$$= g\left(\Theta_1^{(2)} a^{(2)}\right)$$

$$a^{(3)} = g\left(\Theta^{(2)} a^{(2)}\right)$$

$$\mathbf{a}^{(i)} = \begin{cases} \mathbf{x} & \text{for } i = 1 \\ g(\Theta^{(i-1)}\mathbf{a}^{(i-1)}) & \text{for } i > 1 \end{cases}$$

$$g_\Theta(x) = \frac{1}{1 + e^{-\Theta^T x}}$$



Backpropagation

Algorithm:

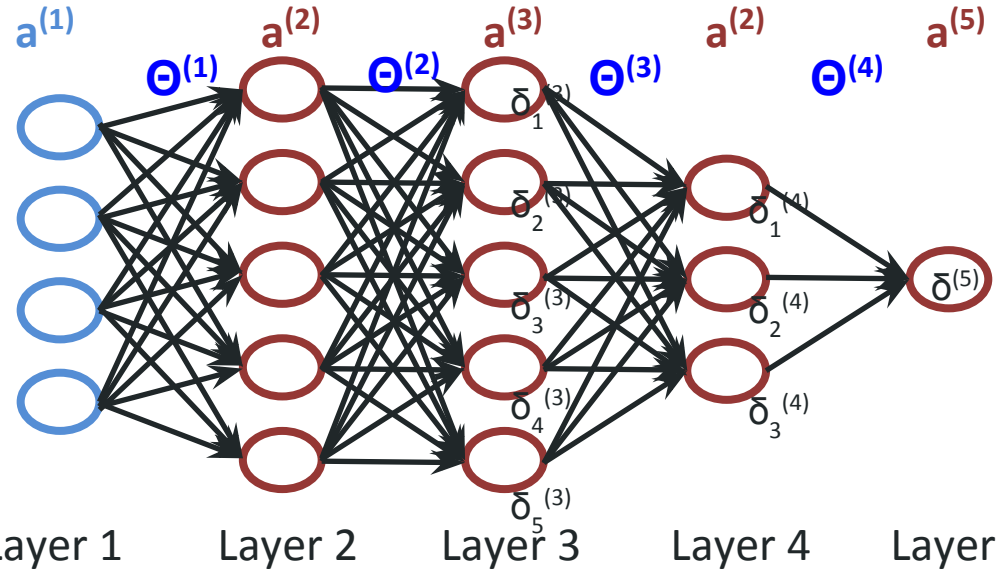
Having performed forward propagation:

- Start at layer L and compute the "error" of each node according to:

$$\delta^{(i)} = \begin{cases} a^{(L)} - y & \text{for } i = L \\ (\Theta^{(i)} \bullet \delta^{(i+1)})\, g'(\Theta^{(i-1)} a^{(i-1)}) & \text{for } 1 < i < L \end{cases}$$

Ignoring λ, the derivatives we want to compute are:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$



Putting it all together

To compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$:

Set $\Delta_{ij}^{(l)} = 0$

For $i = 1$ to m:

- Set $a^{(1)} = x^{(i)}$
- Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$
- Using y^i, compute $\delta^{(L)} = a^{(L)} - y^i$
- Perform backpropagation to compute $\delta^{(l)}$ for $l = (L-1), (L-2), \ldots, 2$
- Set $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
  i.e. $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} \cdot a^{(l)}$

Compute

$$D_{ij}^{(l)} = \begin{cases} \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & j \neq 0 \\ \frac{1}{m} \Delta_{ij}^{(l)} & j = 0 \end{cases}$$

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