1 Basics

There are many applications in AI where the solution in not obvious enough that you can build an algorithm to solve it. Machine learning grew as a possible solution to solve such problems - where the computer learns itself (usually from examples or feedback on performance) on how to solve a particular problem. Applications include data mining in large datasets (e.g. the web for finding answers to Google queries); Natural language processing; Computer Vision.

1.1 What is ML?

ML is what gives computers the ability to learn without explicitly being programmed (according to Arthur Samuel).

A well-posed learning problem: a computer program is said to learn from experience E with respect to some task T on some performance measure P, if the performance on T according to P improves with experience E (according to Tom Mitchell).

Supervised Learning where we teach the computer how to do its task by giving input/output examples. **Unsupervised Learning** is where we let the computer learn itself by optimizing for some metric. There are other approaches in ML like *Reinforcement Learning* and *Recommender systems*.

1.2 Supervised Learning

For instance you want to predict housing prices. Given the price of many houses against their size, it is worthwhile finding what sort of function best fits your data (linear, quadratic, etc). If a person wanted to find what price to offer for his house, he/she could use this function. This is supervised learning because with every example (a house size) you told what is the right answer (the price). These examples were used to find the function - which is actually the process of learning. This was a **regression** problem: where the value we are trying to predict is a continuous value (the price). A **classification** problem is to find a discrete value, for example finding if a tumor is malignant or benign (2 distinct values) given the tumor size.

Another approach in classification could be to find cancerous tumors given examples with two features: tumor size and age. You can plot the data over a 2D space - and then find the line which separates malignant vs benign tumors. Now given a new example our prediction would be based on which side of the line does this example lie.

1.3 Unsupervised Learning

In supervised learning, each training sample was accompanied with an expected output value. In unsupervised learning, the data provided has no expected output value. This essentially turns into a **clustering** problem - where we need to find to which category each data instance belongs to. This is good for finding structure in the data. For instance given the people you communicate with the most on Facebook, can you cluster all the people into different "friend" groups.

Clustering is just one form of unsupervised learning. In the **Cocktail party problem** there is a microphone recording two people talking at the same time. One

person might be louder according to the relative positioning of the two people. The problem is to separate out the voices of the two people - this is done by finding structure in the inputs. Note, you might not know what this structure would be even before you start unsupervised learning, i.e. in the example above you might not know that the separation would be based on the loudness of voice.

2 Linear Regression with One Variable

We want to predict housing prices given the size of a house. We will approach it as a *supervised learning* problem, while trying to *regress* the output. Our training set will have m samples, with $\mathbf{x}$ as the input feature with $\mathbf{y}$ as the output variable. $(\mathbf{x}, \mathbf{y})$ is one training example - $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ is the i^{th} example. So a supervised learning algorithm is to come with a **hypothesis function** $h(\mathbf{x}^*) = \mathbf{y}^*$.

When designing a learning algorithm, its important to decide on the representation for h. Linear regression with one variable (univariate linear regression) i.e. fitting a line to our training data can be represented as finding $h_\theta(\mathbf{x}^*) = \theta_0 + \theta_1 \mathbf{x}^*$, where θ_0 is the y-intercept and θ_1 is the gradient.

2.1 Cost Function

Cost function concerns itself with finding the best fit line in our linear regression case. In $h_\theta(\mathbf{x}^*) = \theta_0 + \theta_1 \mathbf{x}^*$, the θ_0, θ_1 are the **parameters**. The main idea is to choose θ's in a way that minimizes the distance to all training examples:

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

The $\frac{1}{2}$ constant is just to make the math that ensues easier. The $J(\theta_0, \theta_1)$ is called the **cost function**. We are essentially measuring the vertical distance of each training point from the hypothesized line - and we use this as an error measure. As it turns out, this squared error cost function works well for most linear regression problems.

2.2 Cost Function intuition

Let us work with a simplified cost function where $\theta_0 = 0$, i.e. a line passing through the origin. Now we have the cost function $J(\theta_1)$. We can plot $J(\theta_1)$ against θ_1 - the optimal parameter would minimize $J(\theta_1)$. Hence finding the derivative of $J(\theta_1)$ will find the θ_1 which best explains our training data. Coming back to our original problem we have:

$$\textbf{Hypothesis} \quad h_\theta(\mathbf{x}^*) = \theta_0 + \theta_1 \mathbf{x}^* \tag{2}$$

$$\textbf{Parameters} \quad \theta_0, \theta_1 \tag{3}$$

$$\textbf{Cost Function} \quad J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2 \tag{4}$$

$$\textbf{Goal} \quad \min_{\theta_0,\theta_1} J(\theta_0, \theta_1) \tag{5}$$

A convenient way to visualize $J(\theta_0, \theta_1)$ is through a 2D contour plot (a plot of $J(\theta_0, \theta_1)$ against θ_0 and θ_1).

2.3 Gradient descent

This is an optimization technique to find the parameters $\boldsymbol{\theta}$ that minimizes our cost function $J(\boldsymbol{\theta})$. The algorithm:

1. Start with some initial parameters $\boldsymbol{\theta}^{(0)}$.
2. Keep adjusting $\boldsymbol{\theta}$ in a way that it reduces $J(\boldsymbol{\theta})$ from its previous value.
3. When any change in $\boldsymbol{\theta}$ only increases $J(\boldsymbol{\theta})$, we have hopefully arrived at the minimum i.e. our optimal value for $\boldsymbol{\theta}$.

Step 2 essentially says we are going to look in all directions and find which "baby step" which takes us down-hill most quickly. In other words we are finding the direction of *steepest descent* in each iteration. For our two variable example we will be updating θ_0, θ_1 by the following two simultaneous assignments:

$$\theta_0^{(i+1)} := \theta_0^{(i)} - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0^{(i)}, \theta_1^{(i)}) \tag{6}$$

$$\theta_1^{(i+1)} := \theta_1^{(i)} - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0^{(i)}, \theta_1^{(i)}) \tag{7}$$

α is the **learning rate**, which controls the speed of gradient descent (how big our step is in each iteration). $\alpha > 0$ always. If α is small, gradient descent can be slow. If α is large, gradient descent can overshoot the minimum, or may even diverge. Gradient descent can converge to a local minimum, even with the learning rate α fixed. As we approach the local minimum, gradient descent will automatically take smaller steps if α is fixed.

For our squared error cost function, our two simultaneous steps would be:

$$\theta_0^{(i+1)} := \theta_0^{(i)} - \alpha \frac{\partial}{\partial \theta_0} \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2$$

$$\theta_0^{(i+1)} := \theta_0^{(i)} - \alpha \frac{1}{2m} \frac{\partial}{\partial \theta_0} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 \mathbf{x}^{(i)} - y^{(i)} \right)^2$$

$$\theta_0^{(i+1)} := \theta_0^{(i)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 \mathbf{x}^{(i)} - y^{(i)} \right) \tag{8}$$

$$\theta_1^{(i+1)} := \theta_1^{(i)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 \mathbf{x}^{(i)} - y^{(i)} \right) \cdot \mathbf{x}^{(i)} \tag{9}$$

We should note that the squared error cost function for linear regression is always a "bow-shaped" function, or in other words, a **convex function**. A *convex function* doesn't have local optimums, instead just has one global optimum. This is good for gradient descent, which is susceptible to getting stuck in local optimas. This is a "Batch" gradient descent algorithm because it looks at the entire "batch" of training samples in each optimization step.

3 Multivariate Linear Regression

The different features for our setting would written as $x_1, x_2, \cdots, x_n$. Now $x_j^{(i)}$ is the j^{th} feature of the i^{th} training example. The feature $\boldsymbol{x}$ can be represented a vector $\boldsymbol{x} \in \Re^n$. What will be the form of our hypothesis with multiple features would be:

$$h_\theta(\boldsymbol{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots \theta_n x_n \tag{10}$$

For convenience of notation, we will define $x_0 = 1$ (i.e. for all i, $x_0^{(i)} = 1$). Now we can use the vector notation as:

$$\boldsymbol{x} = \begin{bmatrix} x_0 & x_1 & \cdots & x_n \end{bmatrix}^{\mathrm{T}} \in \Re^{n+1} \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 & \theta_1 & \cdots & \theta_n \end{bmatrix}^{\mathrm{T}} \in \Re^{n+1} \tag{11}$$

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \theta_0 x_0 + \theta_1 x_1 + \cdots \theta_n x_n \tag{12}$$

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{x} \quad , \tag{13}$$

where $x_0 = 1$ always.

3.1 Multivariate Gradient Descent

Our cost function now looks like this:

$$J(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{x}^{(i)} - y^{(i)} \right)^2 \tag{14}$$

For our n variable example we will be updating $\boldsymbol{\theta}$ by the following n simultaneous assignments:

$$\theta_j^{(i+1)} := \theta_j^{(i)} - \alpha \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}^{(i)}) \tag{15}$$

$$:= \theta_j^{(i)} - \alpha \frac{\partial}{\partial \theta_j} \frac{1}{2m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{x}^{(i)} - y^{(i)} \right)^2 \tag{16}$$

$$:= \theta_j^{(i)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_{\boldsymbol{\theta}} \left(\boldsymbol{x}^{(i)} \right) - y^{(i)} \right) \boldsymbol{x}_j^{(i)} \tag{17}$$

3.2 Feature scaling in Multivariate Gradient Descent

The idea is that if the range of values in each feature is the same, it turns out gradient descent converges more quickly. Normalizing features leads to circular contours. Our aim is to squeeze/stretch all features to approximately $-1 \leq x_i \leq 1$. Hence a feature would become $x_j^{(i)} := x_j^{(i)} / (\max(x_j) - \min(x_j))$. We can also apply **mean normalization** for forcing all features to have 0 mean by $x_j^{(i)} := (x_j^{(i)} - \mu_j) / (\max(x_j) - \min(x_j))$. We can also set $x_j^{(i)} := (x_j^{(i)} - \mu_j) / \sigma_j$. Why does feature scaling work??

3.3 Setting the Learning Rate

To make sure gradient descent is working correctly, we can check if the value of the cost function $J(\boldsymbol{\theta})$ is decreasing with the number of iterations of the algorithm. One convergence test could be $|J^{(i+1)}(\boldsymbol{\theta}) - J^{(i)}(\boldsymbol{\theta})| < \epsilon$. If $J(\boldsymbol{\theta})$ is increasing (or if $J(\boldsymbol{\theta})$ is periodically swaying between low to high values), then it means that gradient descent is over-shooting the minimum, which call for reducing α.

3.4 Polynomial Regression

We can use the same machinery we used for multivariate linear regression to fit polynomial models to the training data. So for instance we were fitting a cubic polynomial model we will have:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3 \tag{18}$$

Note, because we have exponential terms it becomes increasingly important to use feature scaling.

4 Normal Equation

This Normal Equation method solves for $\boldsymbol{\theta}$ analytically. The problem is to minimize for the vector $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} \in \Re^{n+1} \qquad J(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_{\boldsymbol{\theta}}(x^{(i)}) - y^{(i)} \right)^2 \tag{19}$$

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \cdots = 0 \quad \forall j \tag{20}$$

I can construct a **design matrix X** which contains all the training data, where each row is a training sample, and each column is a feature. Note, the first column in $\mathbf{X}$ will be all 1s since $x_0^{(i)} = 1$. We will also have a vector $\boldsymbol{y}$ containing the output values for each training sample. Now the optimal value for $\boldsymbol{\theta}$ is simply:

$$\boldsymbol{\theta}^* = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\boldsymbol{y} \quad , \tag{21}$$

where we used pseudo-inverse to get our solution analytically. There is no need for *feature scaling* in Normal Equation method. If n is not too large, Normal Equation method is relatively faster than gradient descent.

Remember, if you have redundant features $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ would be non-invertible (singular). This singular condition can also come about if you have much more features than training samples, $m \ll n$.

5 Classification by Logistic Regression

Logistic Regression is a machine learning method for classification. There are both **binary classification** problems and **multi-class classification** problems. Linear regression can be adjusted to perform binary classification - where we would fit a line to our labels $y \in \{0, 1\}$ given x. Once the line is found we can find x^* where the line crosses $y = 0.5$. This x^* will give a threshold for giving labels to our testing data. But, this linear regression model is not accurate for classification most of the time. This is apparent from its awkward performance, for instance linear regression can give a $y^* \notin \{0, 1\}$ although we know those are the only valid values. Logistic Regression is an algorithm where the prediction is $0 \leq h_\theta(x) \leq 1$.

5.1 Hypothesis Representation

We want a model $0 \leq h_\theta(x) \leq 1$. In linear regression we had $h_\theta(x) = \theta^{\mathrm{T}} x$. For logistic regression, we will transform this model to:

$$h_\theta(x) = g(\theta^{\mathrm{T}} x) \tag{22}$$

$$\text{where} \quad g(z) = \frac{1}{1 + e^{-z}} \quad , \tag{23}$$

where $g(z)$ is the **sigmoid function**, which is also known as the **logistic function**. It looks like: Now we will need to fit the parameters θ to the data. The output of

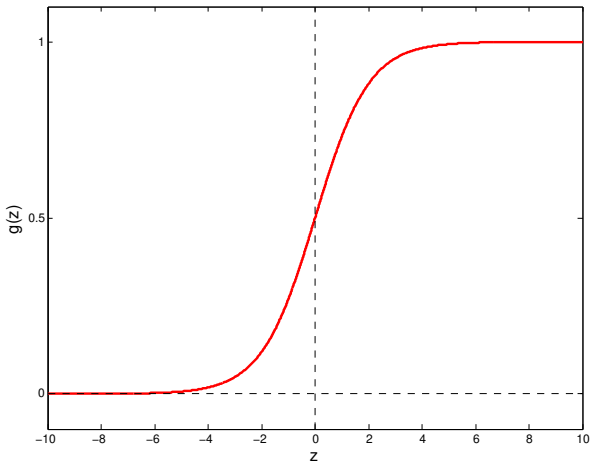


Figure 1: The sigmoid/logistic function.

the classifier can be thought of as the estimated probability that $y = 1$ on any input x. For example we are classifying tumors to be malignant $y = 1$ or benign $y = 0$, according to tumor size x. If $h_\theta(x) = 0.7$, we can say that there is 70% for the tumor to be malignant. Hence, we can also write $h_\theta(x) = \mathrm{P}(y = 1|x; \theta)$, which can be read as $h_\theta(x)$ is the probability that $y = 1$, given x parametrized by θ. Also, remember

$$\mathrm{P}(y = 0|x; \theta) + \mathrm{P}(y = 1|x; \theta) = 1 \quad . \tag{24}$$

5.2 Decision Boundary

Looking at Figure 1, if we can map the input feature x to an appropriate sigmoid function (i.e. find the right $g(z)$), we can predict $y = 1$ if $h_\theta(x) \geq 0.5$ and vice versa. In other words we say $y = 1$ when $z \geq 0$. Following this argument we will set $y = 1$ whenever $\theta^{\mathrm{T}} x \geq 0$.

Suppose $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$. If $\theta^{\mathrm{T}} = \begin{bmatrix} -3 & 1 & 1 \end{bmatrix}^{\mathrm{T}}$, then $x_1 + x_2 \geq 3$ to get classification $y = 1$. This $x_1 + x_2 = 3$ is a line in the input space which decides which *half-space* would be $y = 1$ (for $x_1 + x_2 \geq 3$) and which one would be $y = 0$ (for $x_1 + x_2 < 3$). Hence, this line is aptly named the **decision boundary**. Note that on the line $x_1 + x_2 = 3$, the hypothesis function $h_\theta(x) = 0.5$.

5.2.1 Non-linear Decision Boundaries

If we have data as follows: If we have the non-linear hypothesis function $h_\theta(x) =$

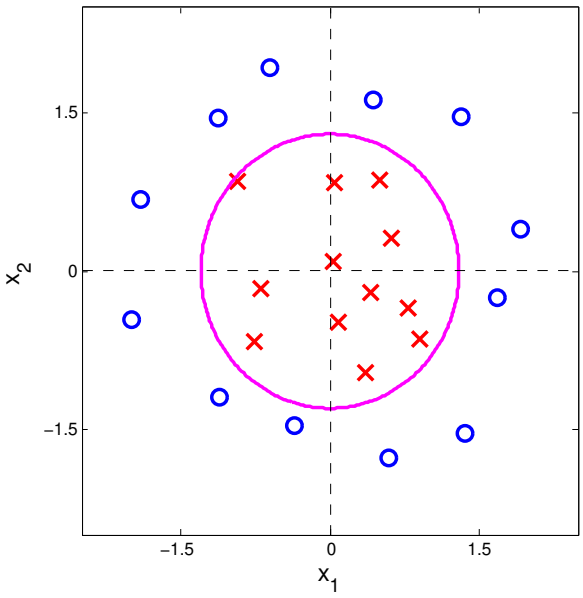


Figure 2: Data requiring non-linear decision boundary. Blue crosses denote $y = 0$ and red circles are $y = 1$. The magenta circle is the non-linear decision boundary as explained in the text.

$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$, where $\theta^{\mathrm{T}} = \begin{bmatrix} -1.3^2 & 0 & 0 & 1 & 1 \end{bmatrix}^{\mathrm{T}}$, we get the equation of a circle $x_1^2 + x_2^2 = 1.3^2$. This circle is shown as a magenta decision boundary in Figure 2. Hence for all points in the region exterior to this circle $x_1^2 + x_2^2 \geq 1.3^2$, we will assign the label $y = 1$. Similarly, for region inside the circle $x_1^2 + x_2^2 < 1.3^2$, we will assign the label $y = 0$. It follows that with higher order polynomials in the hypothesis function, we can have highly complex decision boundaries.

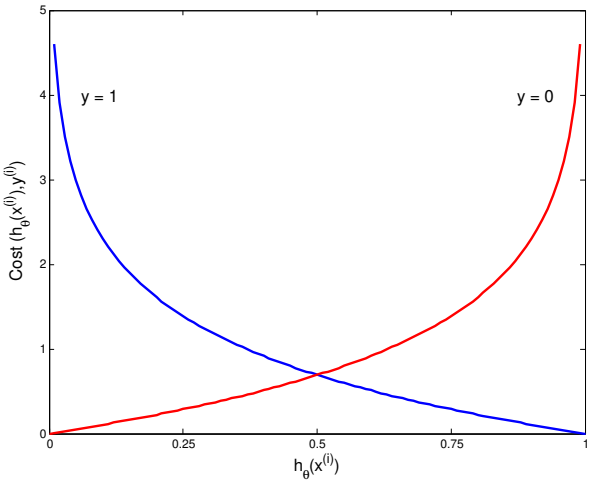


Figure 3: The two cases for the logistic regression cost function.

5.3 Finding parameters for Logistic Regression

The problem of Logistic Regression is:

$$\text{Training Set} \qquad \{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\} \tag{25}$$

$$\ldots \text{with } m \text{ examples} \qquad \boldsymbol{x}^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in \Re^{n+1} \quad \text{where,} \quad x_0^{(i)} = 1,\, y \in \{0, 1\} \quad \forall i$$

$$\text{hypothesis function} \qquad h_\theta(\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{x}}} \tag{26}$$

Given this training set, how do we choose these parameters $\boldsymbol{\theta}$. If we write our cost function as follows

$$J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \mathrm{Cost}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} (h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})^2 \tag{27}$$

$$= \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} \left(\frac{1}{1 + e^{-\boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{x}}} - y^{(i)} \right)^2 \quad , \tag{28}$$

it turns out to be a highly non-convex function (many local optimas). To have a convex formulation, we will turn the cost function into

$$\mathrm{Cost}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)}) = \begin{cases} -\log\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) & \text{if } y^{(i)} = 0 \end{cases} \quad , \tag{29}$$

having separate parts for each label assignment. Note that the cost becomes 0 when $y^{(i)} = h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})$. Also note that the cost $\to \infty$ when $1 - y^{(i)} = h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})$. Remember there are only four cases for these two situations to happen ($y = 0, h_{\boldsymbol{\theta}} = 0$ and $y = 1, h_{\boldsymbol{\theta}} = 1$ and $y = 0, h_{\boldsymbol{\theta}} = 1$ and $y = 1, h_{\boldsymbol{\theta}} = 0$), since y is binary.

5.4 Simplified Cost Function and Gradient Descent

In the previous section we devised a cost function to get the parameters for Logistic Regression. The problem with the cost function in Equation 29 is that it is defined as two separate functions depending on the label itself. We ideally want a cost formulation which is a single equation and involves the label value itself. You can do it in two simple ways:

$$\begin{aligned} \text{Cost}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)}) &= -\log\left(1 - y^{(i)} + (-1)^{1-y^{(i)}} h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) && (30) \\ &= -y^{(i)} \log\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) \; - \; (1 - y^{(i)}) \log\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) \quad . && (31) \end{aligned}$$

Both equations are equivalent but we will concentrate our discussion on Equation 31. Now we have a compact/single function for the logistic regression cost function. Overall, we have

$$J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) \; + \; (1 - y^{(i)}) \log\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) \tag{32}$$

Although we didn't explain (why??) why we choose this particular convex function - it is useful to know that it can be derived from the maximum likelihood estimate. For any training data, now we just need to find $\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. This will be done using *Gradient Descent*. Just a reminder, we just repeat the following step in gradient descent until convergence (simultaneously for all θ_j):

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) \tag{33}$$

Given this minimization, we will get optimal parameter $\boldsymbol{\theta}^*$. After this, if we are given any new $\hat{\boldsymbol{x}}$, our prediction would be $h_{\boldsymbol{\theta}^*}(\hat{\boldsymbol{x}})$, since $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \text{P}(y = 1 | \boldsymbol{x}; \boldsymbol{\theta})$. Computing

the dertivative in Equation 33, we have:

$$\frac{\partial}{\partial\theta_j}J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial\theta_j}y^{(i)}\log\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) + (1-y^{(i)})\log\left(1-h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) \tag{34}$$

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

$$= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\frac{x_j^{(i)}e^{-\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{x}}h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})^2}{h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})} - (1-y^{(i)})\frac{x_j^{(i)}e^{-\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{x}}h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})^2}{1-h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})}$$

$$= -\frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\left(y^{(i)}(1+e^{-\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{x}})-1\right)$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})-y^{(i)}\right)x_j^{(i)} \tag{35}$$

Notice this results in cosmetically the same gradient descent formulation as the one we use for multivariate linear regression (Equation 17). Remember, the ideas of feature scaling used in linear regression also apply to logistic regression.

5.5 Alternatives to Gradient Descent for Regression

Apart from Gradient Descent, there are other faster algorithms such as **Conjugate Gradients**, **BFGS**, and **L-BFGS**. The main intuition behind these methods is a clever inner line-search loop for picking the right α. Some advantages of these algorithms: (1) No need to manually pick α; (2) Often faster than Gradient Descent. The main disadvantage is that implementing them is more complicated than Gradient Descent. For example if we have the problem of finding the optimal value for the cost function $J(\boldsymbol{\theta}) = (\theta_1-5)^2+(\theta_2-5)^2$, we can use the following MATLAB program to solve it:

Listing 1: `fminunc` usage

```
function [jVal, gradient] = costFunction(theta)
        jVal = (theta(1) - 5)^2 + (theta(2) - 5)^2;
        gradient = zeros(2,1);
        gradient(1) = 2*(theta(1)-5);
        gradient(2) = 2*(theta(2)-5);
end

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

By default this program would use a "subspace trust-region method based on the interior-reflective Newton method. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG)." See Mathworks Help for further information on `fminunc`.

5.6 Multi-class Classification - "One vs. All"

Multi-class Classification is simply where the output variables is more than binary. In other words $y \in \{1, 2, \ldots, n\}$ where $n > 2$. **One vs. All** (also called **One vs. Rest**) is one example of multi-class classification algorithm. To demonstrate this algorithm we will consider a problem where $y \in \{1, 2, 3\}$. The key idea would be to use logistic regression, except that now we will break it into three binary problems. In other words when solving for $y = 2$, the positive examples would be $\{\boldsymbol{x}^{(i)}\} : y^{(i)} = 2$ and negative examples would be $\{\boldsymbol{x}^{(i)}\} : y^{(i)} \in \{1, 3\}$. The training stage would give us three decision boundaries, in form of three set of parameters $\boldsymbol{\theta}^{(j)}$ where $j \in \{1, 2, 3\}$. For any new input $\boldsymbol{x}^*$, our prediction for y^* would be class j where $\max_j h_{\boldsymbol{\theta}}^{(j)}(\boldsymbol{x})$. Basically this exactly equivalent to running our three classifiers and seeing which one of them gives the maximum cost.

6 Regularization

If your algorithm is not fitting your data well, it might be an **underfit** OR your model has a **high bias** toward its own hypothesis. On the other hand, if your model learns your training data perfectly, it might be **ovefitting** OR the model has **high variance** - which would lead it to perform quite badly on testing data. In other words the model tries too hard to fit the training data and may fail to generalize to work on unseen examples.

Overfitting: If we have too many features, the learned hypothesis may fit the training set very well (i.e. $J(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})^2 \approx 0$), but fail to generalize to new examples (predict correctly on new examples).

This **overfitting** problem becomes harder to avoid if we have a lot of features and little training data. A couple options to address overfitting:

1. Reduce the number of features:
   - Manually select which features to keep and which to discard.
   - Model selection algorithm (will be discussed later).
2. **Regularization**:
   - Keep all the features, but reduce the magnitude/values of parameters θ_j.
   - Works well when we have lot of features, where each contributes a bit to predicting y.

6.1 Cost Function in Regularization

Let us say we have choice of two cost functions:

$$h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 \tag{36}$$

$$h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4 \tag{37}$$

The cost function in Equation 37 might be able to better fit the training data but might be an overfit as compared to the cost in Equation 36. Nevertheless Equation 37 allows more flexibility in the model and we somehow want to retain that. The main intuition in regularization is to use Equation 37 in our objective function but penalize high order terms. This can be achieved as so:

$$\min_{\boldsymbol{\theta}} \frac{1}{2m}\sum_{i=1}^{m}\left(h_{\boldsymbol{\theta}}(x^{(i)}) - y^{(i)}\right)^2 + L_1\theta_3^2 + L_2\theta_4^2 \quad , \tag{38}$$

where L_1 and L_2 are large numbers so that high order parameters are penalized more. This will force θ_3, $\theta_4 \approx 0$. This function will give us slightly more flexibility over the quadratic cost function given in Equation 36. Overall, having small values for $\theta_0, \theta_1, \ldots \theta_n$ results in simpler hypothesis which are less prone to overfitting (why??). Following this reasoning we want to modify our cost function to shrink all parameters:

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \quad , \tag{39}$$

where λ is the regularization parameter. A smaller λ aim to make our hypothesis fit the training data well, while a large λ keeps our hypothesis simple increasing its generalization capability. If λ is too large, all $\theta_1, \theta_2, \dots \theta_n \approx 0$, hence $h_{\boldsymbol{\theta}}(x) \approx \theta_0$ which is akin of fitting a horizontal line to the data, i.e. it greatly underfits the training data.

6.2 Regularized Linear Regression

We will regularize Linear Regression by $\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ where $J(\boldsymbol{\theta})$ is our modified cost function

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{40}$$

To find our parameters $\boldsymbol{\theta}$, we will use gradient descent to repeat the following $n+1$ simultaneous update steps:

$$\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})x_0^{(i)} \tag{41}$$

$$\theta_j := \theta_j - \alpha\frac{1}{m}\left[\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)} + \lambda\theta_j\right] \qquad \forall\, 0 < j \leq n \tag{42}$$

$$:= \theta_j\left(1 - \alpha\frac{\lambda}{m}\right) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)} \tag{43}$$

Note that, practically $1 - \alpha\frac{\lambda}{m} < 1$, which would have the effect of shrinking θ_j slightly toward 0, over which we are adding the same gradient descent step as before (i.e. $-\alpha\frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)}$). This intuitively shows how gradient descent would result in parameters θ_j which are smaller than what we got in unregularized linear regression.

6.3 Regularized Normal Equation Linear Regression

Remember that our Normal Equation method where we had a design matrix X where each row contained a training example and a vector y having corresponding training labels:

$$\mathbf{X} = \begin{bmatrix}(x^{(1)})^{\mathrm{T}}\\ \vdots \\ (x^{(m)})^{\mathrm{T}}\end{bmatrix} \in \Re^{m\times n+1} \qquad \boldsymbol{y} = \begin{bmatrix}y^{(1)}\\ \vdots \\ y^{(m)}\end{bmatrix} \in \Re^{m} \tag{44}$$

Now we wanted to $\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, so that $\frac{\partial}{\partial\theta_j}J(\boldsymbol{\theta}) = 0 \quad \forall j$. This results in the following normal equation:

$$\boldsymbol{\theta} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\begin{bmatrix}0 & 0 & 0 & \cdots & 0\\ 0 & 1 & 0 & \cdots & 0\\ 0 & 0 & 1 & \cdots & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & 0 & \cdots & 1\end{bmatrix}\right)^{-1} X^{\mathrm{T}}y \tag{45}$$

How???? So consider if $m \leq n$ i.e. if we have fewer examples than features, $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ will be non-invertible/singular. But, given the structure in Equation 45 we always have an invertible matrix when $\lambda > 0$, hence giving us a hypothesis even when we have fewer examples than the number of features. Why???

6.4 Regularized Logistic Regression

Our cost function for logistic regression was

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) \; + \; (1-y^{(i)}) \log\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) \quad , \tag{46}$$

where $h_\theta(\boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{x}}}$. To regularize it, we will take an approach similar to linear regression:

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) \; + \; (1-y^{(i)}) \log\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2 \tag{47}$$

For gradient descent we will end up with steps similar to Equation 41 to 43, where the only difference is in the hypothesis function $h_{\boldsymbol{\theta}}(\boldsymbol{x})$.

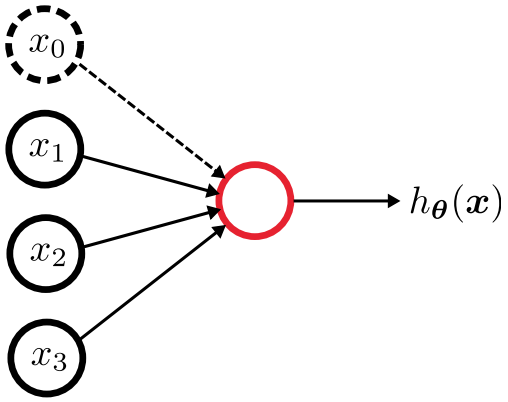


Figure 4: Neuron model: Logistic Unit. You can think of the inputs $\boldsymbol{x}$ as the input *Dendrites*, the red circle is the processing unit, and the output wire leading to $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ is the *Axon*. The dotted line shows the constant bias unit $x_0 = 1$ input.

7 Neural Networks

As the number of features increase, creating a function with polynomial features becomes really expensive. For example, if $\boldsymbol{x} \in \Re^{100}$, the number of parameters required for a function containing all second order features (i.e. $\{x_1^2, x_1x_2, x_1x_3, \ldots, x_1x_{100}, x_2^2, x_2x_3, \ldots\}$) would be ≈ 5000. **Neural Networks** provides a natural way to deal with this problem.

7.1 Model Representation

Each neuron in the brain is a small processing unit. It has a number of input wires called the **Dendrites**, which passes messages to the nucleus, does some processing, and passes the output through its output wires known as the **Axons**. In an *artificial* Neuron model, we will use a simple logistic unit. These networks receive input, let's say $\boldsymbol{x} = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \end{bmatrix}^{\mathrm{T}}$, and work with parameters $\boldsymbol{\theta} = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 & \theta_3 \end{bmatrix}^{\mathrm{T}}$, where, as before, $x_0 = 1$ always. $x_0 = 1$ is known as the **bias unit**. Since its a Sigmoid (logistic) **activation function** $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = 1/\left(1 + e^{-\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{x}}\right)$. Figure 4 shows an example. The parameters $\boldsymbol{\theta}$ are also called **weights** in neural network literature.

We can augment the neuron model to create a neural network. The representation is similar, as given in Figure 5. Let's introduce some new notation: $a_i^{(j)}$ is the activation of unit i in layer j; the neural network is parameterized by $\boldsymbol{\Theta}^{(j)}$, which is a matrix of weights/parameters controlling the function mapping from layer j to layer $j+1$. We will denote row r and column c's value in the parameter matrix as $\boldsymbol{\Theta}^{(j)}_{[r][c-1]}$. The computation involved in a neural network is iterative. For the model given in Figure

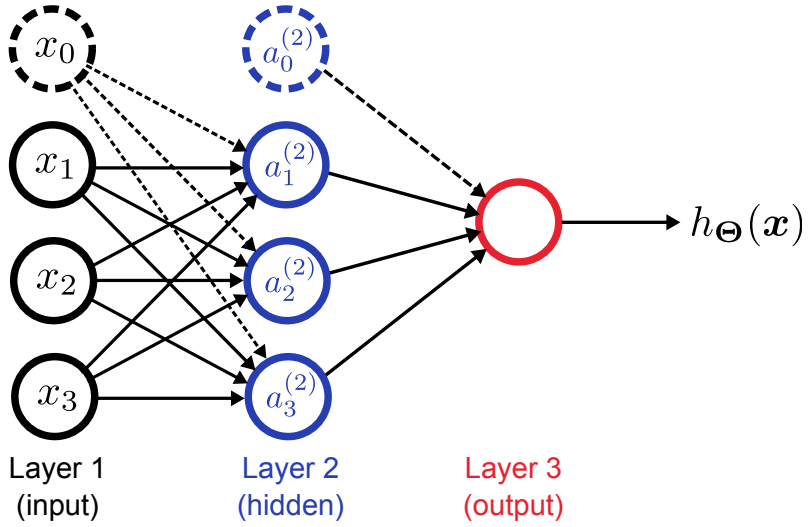


Figure 5: Extension of the neuron model (given in Figure 4) gives this neural network. Each layer's output is passed as an input to the next layer. Layers sandwiched between input and output are called the hidden layers. All layers (except the output layer) also have a bias unit.

5, the computation is as follows:

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\right) = g\left(z_1^{(2)}\right) \quad (48)$$

$$a_2^{(2)} = g\left(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\right) = g\left(z_2^{(2)}\right) \quad (49)$$

$$a_3^{(2)} = g\left(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\right) = g\left(z_3^{(2)}\right) \quad (50)$$

$$h_\Theta(x) = a_1^{(3)} = g\left(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}\right) = g\left(z_1^{(3)}\right) \quad , \quad (51)$$

where $g(\cdot)$ is the sigmoid function. Notice that $\Theta^{(1)} \in \Re^{3\times 4}$ i.e. it is a 3 rows, 4 columns parameter matrix, where row r gives parameters for the activation function $a_r^{(2)}$. If the **network architecture** has s_j units in layer j (excluding the bias unit), and s_{j+1} units in the next layer, then $\Theta^{(j)} \in \Re^{s_{j+1}\times(s_j+1)}$. For our notation it is important to remember that $\Theta^{(j)}$ are the parameters used by layer $j+1$. We will also denote the input at layer j as $a^{(j-1)}$. Hence, $a^{(1)} = x$. Now, we can simply write the computation at layer j as:

$$z^{(j)} = \Theta^{(j-1)}a^{(j-1)} \quad (52)$$

$$a^{(j)} = g\left(z^{(j)}\right) \quad , \quad (53)$$

where $a^{(j)} = \begin{bmatrix} a_0^{(j)} & a_1^{(j)} & \cdots & a_{s_j}^{(j)} \end{bmatrix}^{\mathrm{T}}$. The above algorithm is known as **Forward Propagation**, since we are always pushing the outputs from one layer as inputs to

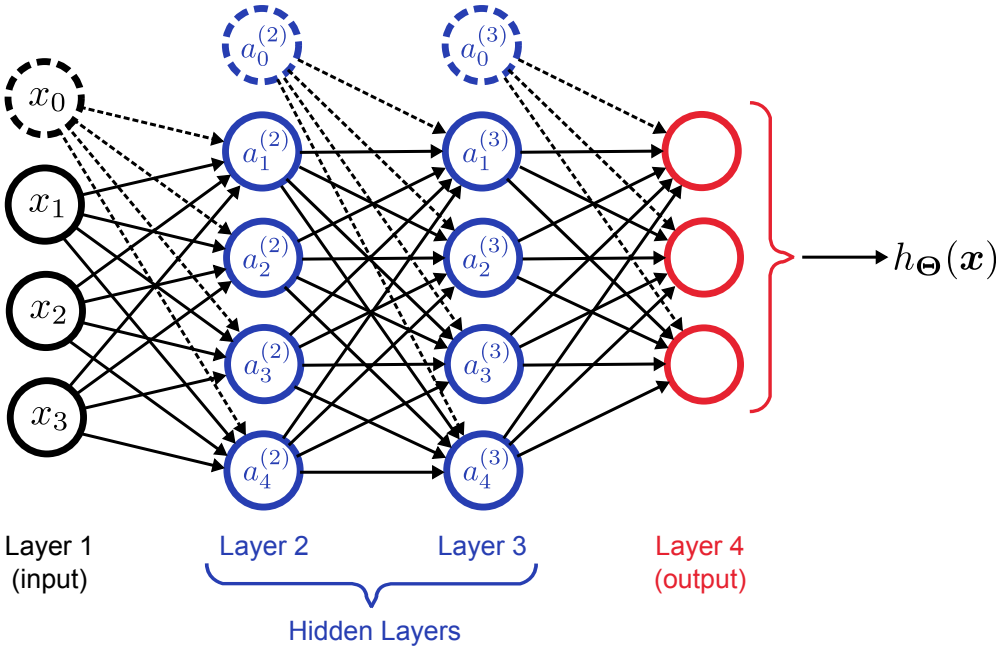


Figure 6: We can extend the neural network model in Figure 5 to a multi-class classification problem. In this example we are trying to classify three possible classes.

the next layer. The idea of learning parameters at layer j from the outputs of layer $j-1$, allows neural networks to learn highly non-linear functions. Adding layers is like adding to the order of features accomodated by our output function.

7.2 Classification of truth-tables

7.3 Multi-class Classification

The original neural network architecture can be extended to act as a multi-class classification algorithm. In this setting if we are classifying K classes, $h_{\boldsymbol{\Theta}}(\boldsymbol{x}) \in \Re^K$. For the example given in Figure 6, $K = 3$. Our aim here would be to have the following binary vector outputs for each of the 3 classes case,

$$h_{\boldsymbol{\Theta}}(\boldsymbol{x}) \approx \begin{cases} \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^{\mathrm{T}} & \text{when class 1} \\ \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^{\mathrm{T}} & \text{when class 2} \\ \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^{\mathrm{T}} & \text{when class 3} \end{cases} . \tag{54}$$

Essentially we train each output unit to classify a specific class. Hence, Figure 6 has 3 output units (at layer 4). Following our figure we can represent out training set of m examples as $\left(x^{(1)}, y^{(1)}\right)$, $\left(x^{(2)}, y^{(2)}\right)$, $\ldots$ $\left(x^{(m)}, y^{(m)}\right)$, where $y^{(i)}$ takes one of three

vector outputs given in Equation 54, i.e:

$$y^{(i)} \in \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\} \tag{55}$$

Now that we know how to represent neural networks in both multi-class and binary classification problems, the question is how to train them. In other words how do we find our parameters set $\boldsymbol{\Theta}^{(j)}$ for all layers. An interesting situation to think about here is the case of two layer neural network. The parameters of a two layer neural network can be easily found by gradient descent methods we have encountered previously.

7.4 Cost function for Neural Network Learning

Some notation: L is the number of layers in the neural network; s_l is the number of units in layer l, when not counting the bias unit. For instance, $L = 4$ and $s_2 = 4$ in Figure 6. Since K is the number of classes being classified, for binary-classification $K = 1$ and there is only 1 output unit giving $y \in \{0, 1\}$. Hence, $h_{\boldsymbol{\Theta}}(\boldsymbol{x}) \in \Re$ and the number of units in the output layer is $s_L = 1$. For multi-class classification when $K \geq 3$ (note $K = 2$ should be treated as a binary classification problem - not as a multi-class problem), there are K output units and $y \in \Re^K$ as demonstrated in Equation 55. Hence, $h_{\boldsymbol{\Theta}}(\boldsymbol{x}) \in \Re^K$. Note that $s_L = K$, i.e. the number of output units is equal to the number of dimensions in the output variable y.

Remember that the cost function for logistic regression was

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) \; + \; (1 - y^{(i)}) \log\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2 \quad . \tag{56}$$

The cost function for a neural network would be a generalization of the logistic regression cost function. Rather than having "one logistic regression output per unit, we will have K of them." The cost function will be

$$J(\boldsymbol{\Theta}) = -\frac{1}{m}\left[\sum_{i=1}^{m} \underbrace{\sum_{k=1}^{K} y_k^{(i)} \log\left(h_{\boldsymbol{\Theta}}(\boldsymbol{x}^{(i)})_k\right) \; + \; (1 - y_k^{(i)}) \log\left(1 - h_{\boldsymbol{\Theta}}(\boldsymbol{x}^{(i)})_k\right)}_{\text{sum over } K \text{ output units}}\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} \left(\boldsymbol{\Theta}_{ji}^{(l)}\right)^2 \quad , \tag{57}$$

where $h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) \in \Re^K$, and $h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})_k$ refers to the k-th element in the vector. Similarly, $y^{(i)} \in \Re^K$, and $y_k^{(i)}$ refers to the k-th element. Remember that $K = 1$ for the binary classification case. As discussed previously, $\boldsymbol{\Theta}^{(l)} \in \Re^{s_{l+1} \times s_l + 1}$, i.e. the number of parameters needed for layer $l + 1$ is equivalent to the number of inputs for layer l, s_l (plus the bias unit) times the number of units (sans the bias unit) in layer $l + 1$, s_{l+1}. Note that for the regularization contraint at layer $l + 1$, $\sum_{i=1}^{s_l}(\cdot)$ sums over the number of inputs, and $\sum_{j=1}^{s_{l+1}}(\cdot)$ sums over the number of units at that layer. The regularization term is easier to understand after looking at Equation 48 to 50.

Although the first term in Equation 57 only sums over the number of output units, $h_{\boldsymbol{\Theta}}(\cdot)$ is a function over all parameters in the neural network. This will allow us to optimize all the parameters in the neural network.

7.5 Backpropagation Algorithm

Given the $J(\boldsymbol{\Theta})$ in Equation 57, we would like to find $\min_{\boldsymbol{\Theta}} J(\boldsymbol{\Theta})$. For this, for a gradient dependent optimization method, we would need to compute $J(\boldsymbol{\Theta})$ and $\frac{\partial}{\partial \boldsymbol{\Theta}_{ji}^{(l)}} J(\boldsymbol{\Theta})$ for any provided $\boldsymbol{\Theta}$. Suppose that we only have one training example, i.e. $m = 1$. This will reduce Equation 57 to

$$J(\boldsymbol{\Theta}) = -\left[\underbrace{\sum_{k=1}^{K} y_k^{(i)} \log\left(h_{\boldsymbol{\Theta}}(\boldsymbol{x}^{(i)})_k\right) \; + \; (1 - y_k^{(i)}) \log\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})_k\right)}_{\text{sum over } K \text{ output units}}\right] + \frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(\boldsymbol{\Theta}_{ji}^{(l)}\right)^2 \quad . \tag{58}$$

First, following the forward propagation model (given in Equation 52 and 53), we can write the following equations for Figure 6:

$$\boldsymbol{a}^{(1)} = \boldsymbol{x} \tag{59}$$

$$\begin{aligned} \boldsymbol{z}^{(2)} &= \boldsymbol{\Theta}^{(1)} \boldsymbol{a}^{(1)} \\ \boldsymbol{a}^{(2)} &= g(\boldsymbol{z}^{(2)}) && = g\left(\boldsymbol{\Theta}^{(1)} \boldsymbol{a}^{(1)}\right) \end{aligned} \tag{60}$$

$$\begin{aligned} \boldsymbol{z}^{(3)} &= \boldsymbol{\Theta}^{(2)} \boldsymbol{a}^{(2)} && = \boldsymbol{\Theta}^{(2)} g\left(\boldsymbol{\Theta}^{(1)} \boldsymbol{a}^{(1)}\right) \\ \boldsymbol{a}^{(3)} &= g(\boldsymbol{z}^{(3)}) && = g\left(\boldsymbol{\Theta}^{(2)} g\left(\boldsymbol{\Theta}^{(1)} \boldsymbol{a}^{(1)}\right)\right) \end{aligned} \tag{61}$$

$$\begin{aligned} \boldsymbol{z}^{(4)} &= \boldsymbol{\Theta}^{(3)} \boldsymbol{a}^{(3)} && = \boldsymbol{\Theta}^{(3)} g\left(\boldsymbol{\Theta}^{(2)} g\left(\boldsymbol{\Theta}^{(1)} \boldsymbol{a}^{(1)}\right)\right) \\ h_{\boldsymbol{\Theta}}(\boldsymbol{x}) = \boldsymbol{a}^{(4)} &= g(\boldsymbol{z}^{(4)}) && = g\left(\boldsymbol{\Theta}^{(3)} g\left(\boldsymbol{\Theta}^{(2)} g\left(\boldsymbol{\Theta}^{(1)} \boldsymbol{a}^{(1)}\right)\right)\right) \end{aligned} \tag{62}$$

Note that if we use Equation 62 to compute the $\frac{\partial}{\partial \boldsymbol{\Theta}_{ji}^{(l)}} J(\boldsymbol{\Theta})$, it would be increasingly impossible, as it calls for derivating repeated invocations of the sigmoid function. We take an alternative approach where we will keep track of $\delta_j^{(l)}$, the error in each unit j in layer l. The error here means how much we would like to change $a_j^{(l)}$ to get to the true value. Looking at figure 6, for each output unit ($L = 4$):

$$\delta_j^{(4)} \; = \; \boldsymbol{a}_j^{(4)} - y_j \; = \; h_{\boldsymbol{\Theta}}(\boldsymbol{x})_j - y_j \tag{63}$$

$$\delta^{(4)} \; = \; \boldsymbol{a}^{(4)} - y \tag{64}$$

For all the previous layers we have the error terms are computed as follows (why????):

$$\delta^{(3)} = \left(\boldsymbol{\Theta}^{(3)}\right)^{\mathrm{T}} \delta^{(4)} \,.*\, g'\left(\boldsymbol{z}^{(3)}\right) \tag{65}$$

$$\delta^{(2)} = \left(\boldsymbol{\Theta}^{(2)}\right)^{\mathrm{T}} \delta^{(3)} \,.*\, g'\left(\boldsymbol{z}^{(2)}\right) \quad , \tag{66}$$

where $g'\left(\boldsymbol{z}^{(3)}\right) = \boldsymbol{a}^{(3)} \,.*\, (1 - \boldsymbol{a}^{(3)})$ and $g'\left(\boldsymbol{z}^{(2)}\right) = \boldsymbol{a}^{(2)} \,.*\, (1 - \boldsymbol{a}^{(2)})$. Note that there is no $\delta^{(1)}$ since there is no "error" in the input term. This algorithm is called **back-propagation** since all the errors are propagated back to previous layers. If we ignore the regularization terms, it can be proved that the partial derivative is equal to

$$\frac{\partial}{\partial \boldsymbol{\Theta}_{ji}^{(l)}} J(\boldsymbol{\Theta}) = \boldsymbol{a}_j^{(l)} \delta_i^{(l+1)} \qquad \text{if } \lambda = 0 \tag{67}$$

Since the total error term for m training samples would just be the sum of the error term computed in Equation 67 for each example, our final algorithm (without regularization) would be as given in Algorithm 1. The last for-loop adds the regularization

Algorithm 1 Neural networks learning by **Back Propagation**

Training set $\left\{\left(\boldsymbol{x}^{(1)}, y^{(1)}\right), \left(\boldsymbol{x}^{(2)}, y^{(2)}\right), \ldots \left(\boldsymbol{x}^{(m)}, y^{(m)}\right)\right\}$
$\Delta_{ji}^{(l)} := 0 \qquad \forall\, l, i, j$
for $i := 1 \to m$ **do**
  $\boldsymbol{a}^{(1)} := \boldsymbol{x}^{(i)}$
  Forward propagation (Eq. 52 and 53) to compute $\boldsymbol{a}^{(l)}$ for $l \in \{2, 3, \ldots, L\}$
  $\delta^{(L)} = \boldsymbol{a}^{(L)} - y^{(i)}$
  for $l := (L-1) \to 2$ **do**
    $g'\left(\boldsymbol{z}^{(l)}\right) := \boldsymbol{a}^{(l)} \,.*\, (1 - \boldsymbol{a}^{(l)})$
    $\delta^{(l)} := \left(\boldsymbol{\Theta}^{(l)}\right)^{\mathrm{T}} \delta^{(l+1)} \,.*\, g'\left(\boldsymbol{z}^{(l)}\right)$
  end for
  $\Delta_{ji}^{(l)} := \Delta_{ji}^{(l)} + \boldsymbol{a}_i^{(l)} \delta_j^{(l+1)}$
end for
for $\forall\, j$ **do**
  if $j = 0$ **then**
    $D_{ji}^{(l)} = \frac{1}{m} \Delta_{ji}^{(l)}$
  else
    $D_{ji}^{(l)} = \frac{1}{m} \Delta_{ji}^{(l)} + \frac{\lambda}{m} \boldsymbol{\Theta}_{ji}^{(l)}$
  end if
end for

term. It can be mathematically proven that $\frac{\partial}{\partial \boldsymbol{\Theta}_{ji}^{(l)}} J(\boldsymbol{\Theta}) = D_{ji}^{(l)}$.

There is one importate point about running the optimization over Algorithm 1. Suppose if we initialize all $\boldsymbol{\Theta}^{(l)}$ to a constant $\mathbf{0}$, this essentially means setting all the weights to 0. This is equivalent to setting $g(\boldsymbol{z}_j^{(l)}) = \frac{1}{2}$. Hence, all units in all layers will exactly the same. A direct corollary of this 0 initialization is that even all error terms

$\delta_k^{(l)}$ will be the same. This would make all partial derivates the same too. Hence, after each update, parameters corresponding to inputs going into each of the s_{j+1} hidden units would be identitical. This is not only true for choosing an initial parameter $\mathbf{0}$ but for any other constant value. Hence it is important to randomly initialize the parameters.

8 Machine Learning Diagnostics

After training, you might figure out that your hypothesis has an unpredictably large error on new, unseen data. Things we can try in an attempt to ammeliorate this:

1. Get more training examples.
2. Try a smaller set of features, by selecting the more informative features manually.
3. Try adding additional features.
4. Adding polynomial features.
5. Changing the regularization parameter, λ.

There is a simple technique to rule out some options in the list above. These machine learning **diagnostic techniques** are helpful in finding avenues which might be more fruitful exploring than others.

8.1 Evaluating a Hypothesis

Once we have a trained hypothesis function $h_{\boldsymbol{\theta}}(\boldsymbol{x})$, we can test it on unseen examples. Before we finalize our classifier we should be worried if the classifier has overfit our training set. If we only had 1 or 2 features (i.e. $\boldsymbol{x} \in \Re$ or $\boldsymbol{x} \in \Re^2$), it would be simple to plot and see how much the hypothesis overfits the dataset. Ofcourse, another method is to see the training error. But, as the number of features increase it is hard to gauge overfitting by just observing the training error, since it might be just high due to a few outliers, while the hypothesis is still overfitting the training.

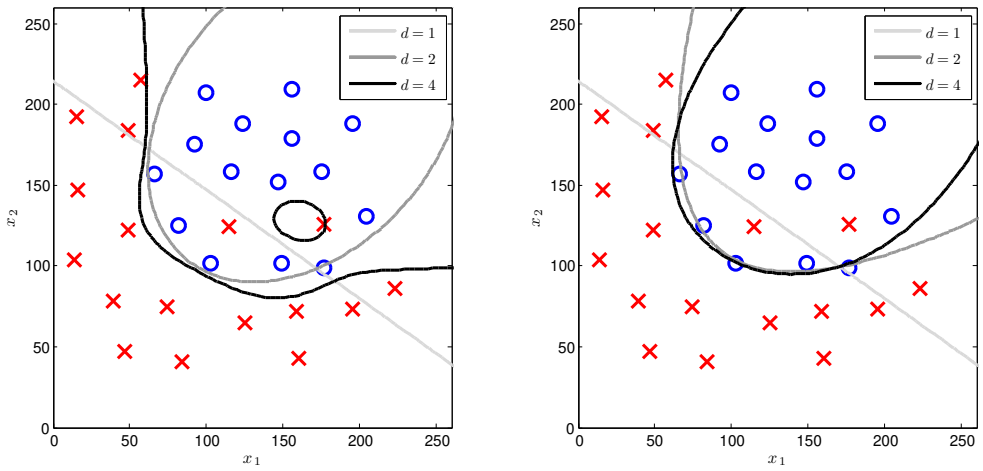
> ✠ *A simple technique to counter overfitting is to divide the training set into two mutually exclusive, collectively exhaustive sets. One set is actually used for training, and the other is used for testing the trained hypothesis.*

Now the training set will be denoted as $\left\{\left(\boldsymbol{x}^{(i)}, y^{(i)}\right) : \forall i \in 1, m\right\}$ and the test set as $\left\{\left(\boldsymbol{x}_t^{(i)}, y_t^{(i)}\right) : \forall i \in 1, m_t\right\}$. The high test error on the m_t test samples is typically indicative of overfitting. It is typical to divide the training set in a 70% to 30% ratio (30% is the test set). We can compute the test set error for linear regression as:

$$J_t(\boldsymbol{\theta}) = \frac{1}{2m_t} \sum_{i=1}^{m_t} \left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_t^{(i)}\right) - y_t^{(i)}\right)^2 \quad . \tag{68}$$

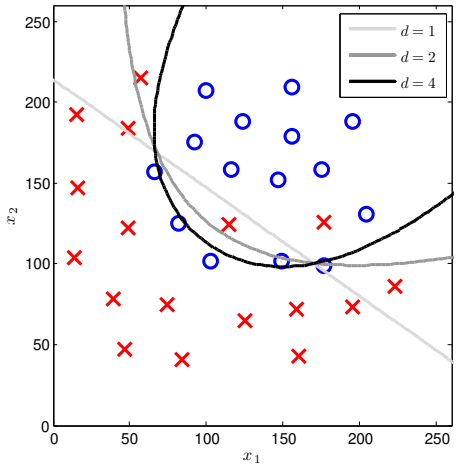
Similarly for logistic regression our test error is:

$$J_t(\boldsymbol{\theta}) = -\frac{1}{2m_t} \sum_{i=1}^{m_t} y_t^{(i)} \log h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_t^{(i)}\right) + \left(1 - y_t^{(i)}\right) \log\left(1 - h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_t^{(i)}\right)\right) \quad . \tag{69}$$

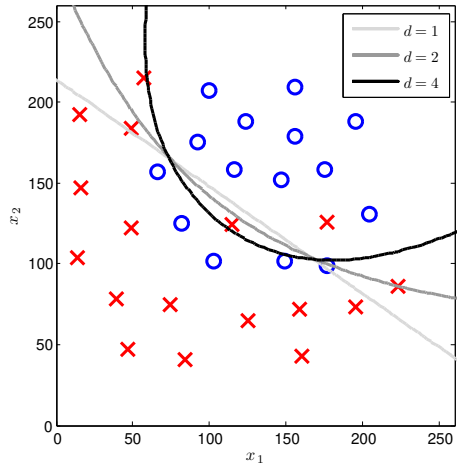


(a) $\lambda = 0$

(b) $\lambda = 0.05$



(c) $\lambda = 0.4$



(d) $\lambda = 2$

Figure 7: Shows the effect of changing the degree of polynomial used as features for building a logistic regression classifier. The features used were $\mathbf{x} = \{x_1^{i-j} x_2^j : \forall j \in \{0, \dots i\}, i \in \{1, \dots d\}\}$, where each graph shows the decision boundary at $d \in \{1, 2, 4\}$ illustrating the transition from underfitting to overfitting as the number of features increase. Each figure was generated with a different regularization parameter, λ. Notice the reduced tedency to overfit as λ increases. Also note how model needs to be more flexible (more features) in order to give a reasonable decision boundary when λ is large.

Another method would be to look at the **misclassification error** (0/1 misclassification error):

$$\text{err}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right), y\right) = \begin{cases} 1 & \text{if } h_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right) \geq 0.5,\ y = 0 \\ & \text{or } h_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right) \leq 0.5,\ y = 1 \\ 0 & \text{otherwise} \end{cases} \tag{70}$$

The test error is this case is simply:

$$J_t(\boldsymbol{\theta}) = \frac{1}{m_t} \sum_{i=1}^{m_t} \text{err}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_t^{(i)}\right), y_t^{(i)}\right) \quad . \tag{71}$$

8.2 Model Selection (and training/validation/test sets)

Suppose we want to find what degree of polynomial to fit to a dataset or what regularization parameter λ to use. These are called **model selection** problems. During the course of this discussion we will explore the ideas of training, validation and test sets.

From our previous discussions we know that the training error is not a good predictor of how well our hypothesis would do on a test set. This is known as the generalization ability of our learned hypothesis.

> ✠ *Once parameter $\theta_0, \theta_1, \ldots \theta_n$ were fit to some set of data (training set), the error of the parameters as measured on that data (the training error $J(\boldsymbol{\theta})$) is likely to be lower than the actual generalization error.*

The model selection problem is the decision to find what polynomial equation to use for the hypothesis. We illustrate the problem on a univariate case where we try a set of hypothesis function with different degree of polynomials d:

$$\begin{aligned}
&d = 1 \ :\ h_{\boldsymbol{\theta}^{(1)}}(x) = \theta_0 + \theta_1 x && \boldsymbol{\theta}^{(1)} \rightarrow J_t\left(\boldsymbol{\theta}^{(1)}\right) \\
&d = 2 \ :\ h_{\boldsymbol{\theta}^{(2)}}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 && \boldsymbol{\theta}^{(2)} \rightarrow J_t\left(\boldsymbol{\theta}^{(2)}\right) \\
&d = 3 \ :\ h_{\boldsymbol{\theta}^{(3)}}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 && \boldsymbol{\theta}^{(3)} \rightarrow J_t\left(\boldsymbol{\theta}^{(3)}\right) \\
&\qquad\qquad \vdots \\
&d = 10 \ :\ h_{\boldsymbol{\theta}^{(10)}}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \ldots \theta_{10} x^{10} && \boldsymbol{\theta}^{(10)} \rightarrow J_t\left(\boldsymbol{\theta}^{(10)}\right)
\end{aligned}$$

In each row, note we also compute the test error with the trained parameters using that polynomial. The naïve approach is to simply choose the polynomial hypothesis exhibiting the lowest test error i.e. $\min_d J_t\left(\boldsymbol{\theta}^{(d)}\right)$. Let us say $d = 5$ exhibits the lowest test error[1]. Now the question is how well does this model generalize? Can we

[1] Note that we are looking at the test error, which is computed on the test data m_t. If the error was computed on the same set used for training, $d = 10$ would have the best chances for producing the lowest test error.

report the test set error $J_t\left(\boldsymbol{\theta}^{(5)}\right)$? The problem in doing so is that $J_t\left(\boldsymbol{\theta}^{(5)}\right)$ is likely an optimistic estimate of the generalization error, since we have in some senses fit an extra parameter (d the degree of the polynomial) to the test set.

To correct these issues we divide our training set into 3 sections:

1. The typical train set: $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)$.
2. The cross-validation set: $\left(\boldsymbol{x}_{cv}^{(i)}, y_{cv}^{(i)}\right)$.
3. The test set: $\left(\boldsymbol{x}_{t}^{(i)}, y_{t}^{(i)}\right)$.

The typical ration of this division is $60:20:20$. The purpose of the cross-validation set is as follows: we use our 10 polynomial hypothesis equations $h_{\boldsymbol{\theta}^{(i)}}(x) \ : \ \forall i \in \{1, 10\}$ to (1) to get the ideal parameters for each polynomial using the train set, $\min_{\boldsymbol{\theta}^{(d)}} J\left(\boldsymbol{\theta}^{(d)}\right)$; (2) we choose the right model d^* which has the minimum error using the cross-validation set, $d^* = \min_d J_{cv}\left(\boldsymbol{\theta}^{(d)}\right)$; and (3) we report an estimate of the generalization error using the test set, $J_t\left(\boldsymbol{\theta}^{(d^*)}\right)$.

8.3 Diagnosing Bias vs. Variance

Lower than expected performance on machine learning algorithm is usually due to bias (underfiiting) or variance (overfitting) problem. Being able to diagnose these two problems are an important part of a practitioners toolbox. [Show how the training error decreases with d and the cross-validation error has a trough on the same curve].

Suppose your learning algorithm is performing less well than you were hoping (i.e. $J_{cv}(\boldsymbol{\theta})$ or $J_t(\boldsymbol{\theta})$ is high), how do you find if it is a bias or a variance problem? For the bias (underfit) case both the cross-validation and training error would be high i.e. $J(\boldsymbol{\theta})$ will be high and $J_{cv}(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta})$. On the other hand, in the variance (overfit) case the training error would be really low while the cross-validation error would be high i.e. $J(\boldsymbol{\theta})$ will be low and $J_{cv}(\boldsymbol{\theta}) \gg J(\boldsymbol{\theta})$.

8.3.1 Regularization Bias and Variance

Consider the $d = 4$ linear regression model with regularization:

$$h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4 \tag{72}$$

$$J(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_{\boldsymbol{\theta}}(x^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2 \tag{73}$$

9 Support Vector Machines

Supervised learning algorithm

The optimization objective. Let's modify first the logistic regression function:

$$h_\theta = \frac{1}{1 + e^{-\theta^T x}} \tag{74}$$

- What we want is that when $y = 1$, we want $h_\theta(x) \approx 1, \theta^T x \gg 0$.
- What we want is that when $y = 0$, we want $h_\theta(x) \approx 0, \theta^T x \ll 0$.

The cost function for SVM is an approximation of the cost function used by logistic regression. The cost function is a flat line at 0 from beyong 1 - and a constant gradient line before that.

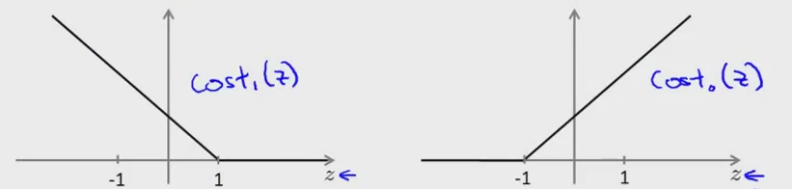


Figure 8: The two cost functions of SVM.

$$\min_\theta C \sum_{i=1}^{m} y^{(i)} \text{cost}_1\left(\theta^T x^{(i)}\right) + (1 - y^{(i)}) \text{cost}_0\left(\theta^T x^{(i)}\right) + \frac{1}{2} \sum_{j=0}^{n} \theta_j^2 \tag{75}$$

SVM doesn't output a probability, instead outputs a prediction:

$$h_\theta(x) = \begin{cases} 1 & \text{if } \theta^T x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{76}$$

What SVM aims to do:

- What we want is that when $y = 1$, we want $\theta^T \geq 1$.
- What we want is that when $y = 0$, we want $\theta^T \leq -1$.

Large Margin Classifier: If C is very large, the optimization problem boils down to:

$$\min_\theta \frac{1}{2} \sum_{i=1}^{n} \theta_j^2 = \min_\theta \frac{1}{2} \|\theta\|^2$$

$$\text{s.t.} \begin{cases} \theta^T x^{(i)} \geq 1 & \text{if } y^{(i)} = 1 \\ \theta^T x^{(i)} \leq -1 & \text{if } y^{(i)} = 0 \end{cases}$$

If the data is linearly separable, the optimization above gives the max-margin classifier.