Pattern Recognition and Machine Learning

James L. Crowley

ENSIMAG 3 - MMIS
Lessons 7

Fall Semester 2016
14 Dec 2016

Artificial Neural networks

Outline

Using notation and figures from the Stanford Deep learning tutorial at:
http://ufldl.stanford.edu/tutorial/

Notation

x_d	A feature. An observed or measured value.
$\vec{X}$	A vector of D features.
D	The number of dimensions for the vector $\vec{X}$
$\{\vec{X}_m\}$ $\{y_m\}$	Training samples for learning.
M	The number of training samples.
$a_i^{(l)}$	the activation output of the i^{th} neuron of the l^{th} layer.
$w_{ij}^{(l)}$	the weight for the unit j of layer l and the unit i of layer $l+1$.
b_i^l	the bias term for i^{th} using of the the $l+1^{th}$ layer

Introduction

Artificial Neural Networks

Artificial Neural Networks, also referred to as "Multi-layer Perceptrons", are computational structures composed a weighted sums of "neural" units. Each neural unit is composed of a weighted sum of input units, followed by a non-linear decision function.

Note that the term "neural" is misleading. The computational mechanism of a neural network is only loosely inspired from neural biology. Neural networks do NOT implement the same learning and recognition algorithms as biological systems.

The approach was first proposed by Warren McCullough and Walter Pitts in 1943 as a possible universal computational model. During the 1950's, Frank Rosenblatt developed the idea to provide a trainable machine for pattern recognition, called a Perceptron. The perceptron is an incremental learning algorithm for linear classifiers invented by Frank Rosenblatt in 1956. The first Perceptron was a room-sized analog computer that implemented Rosenblatz learning recognition functions. Both the learning algorithm and the resulting recognition algorithm are easily implemented as computer programs.

In 1969, Marvin Minsky and Seymour Papert of MIT published a book entitled "Perceptrons", that claimed to document the fundamental limitations of the perceptron approach. Notably, they claimed that a perceptron could not be constructed to perform an "exclusive OR".

In the 1970s, frustrations with the limits of Artificial Intelligence research based on Symbolic Logic led a small community of researchers to explore the perceptron based approach. In 1973, Steven Grossberg, showed that a two layered perceptron could overcome the problems raised by Minsky and Papert, and solve many problems that plagued symbolic AI. In 1975, Paul Werbos developed an algorithm referred to as "Back-Propagation" that uses gradient descent to learn the parameters for perceptrons from classification errors with training data.

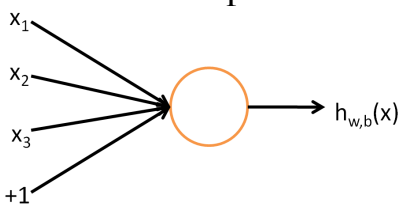
During the 1980's, Neural Networks went through a period of popularity with researchers showing that Networks could be trained to provide simple solutions to problems such as recognizing handwritten characters, recognizing spoken words, and steering a car on a highway. However, results were overtaken by more

mathematically sound approaches for statistical pattern recognition such as support vector machines and boosted learning.

In 1998, Yves LeCun showed that convolutional networks composed from many layers could outperform other approaches recognition problems. Unfortunately such networks required extremely large amounts of data and computation. Around 2010, with the emergence of cloud computing combined with planetary-scale data convolutional networks became practical. Since 2012, Deep Networks have outperformed other approaches for recognition tasks common to computer Vision, Speech and robotics. A rapidly growing research community currently seeks to extend the application beyond recognition to generation of speech and robot actions.

The Artificial Neuron

The simplest possible neural network is composed of a single neuron.



A "neuron" is a computational unit that integrates information from a vector of D features, $\vec{X}$, to the likelihood of a hypothesis, $h_{w,b}()$

$$a = h_{\vec{w},b}(\vec{X})$$

The neuron is composed of a weighted sum of input values

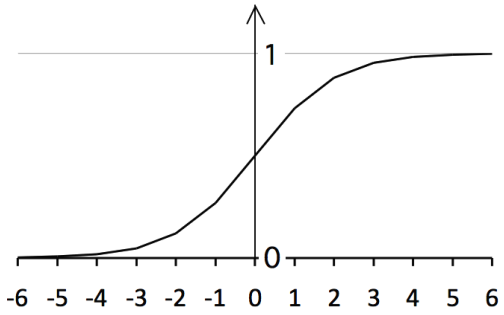
$$z = w_1 x_1 + w_2 x_2 + ... + w_D x_D + b$$

followed by a non-linear "activation" function, $f(z)$ (sometimes written $\phi(z)$)

$$a = h_{\vec{w},b}(\vec{X}) = f(\vec{w}^T \vec{X} + b)$$

Many different activation functions are used.

A popular choice for hidden layers is the sigmoid (logistic) function: $f(z) = \frac{1}{1 - e^{-z}}$



This function is useful because the derivative is: $\frac{df(z)}{dz} = f(z)(1 - f(z))$

This gives a decision function: if $h_{\vec{w}b}(\vec{X}) > 0.5$ POSITIVE else NEGATIVE

Other popular decision functions include the hyperbolic tangent and the softmax.

The hyperbolic Tangent: $f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

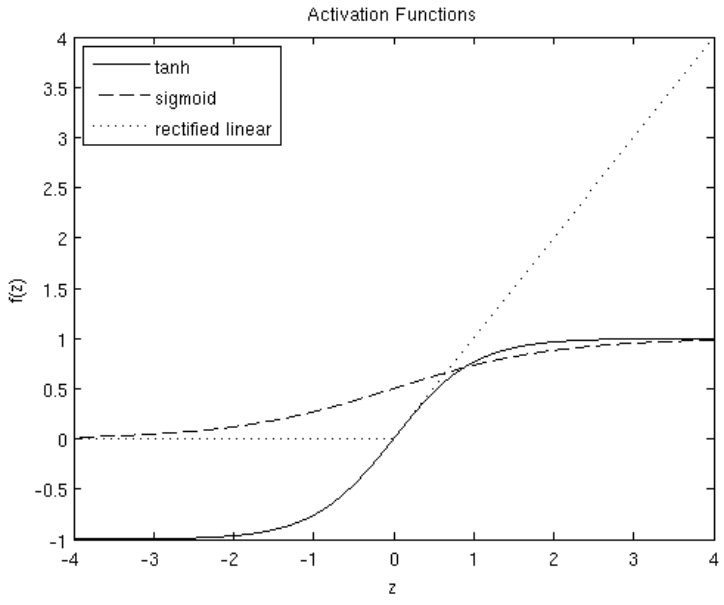
The hyperbolic tangent is a rescaled form of sigmoid ranging over [-1,1]

The rectified linear function is also popular

Relu: $f(z) = \max(0, z)$

While Relu is discontinuous at z=0, for $z > 0$: $\frac{df(z)}{dz} = 1$

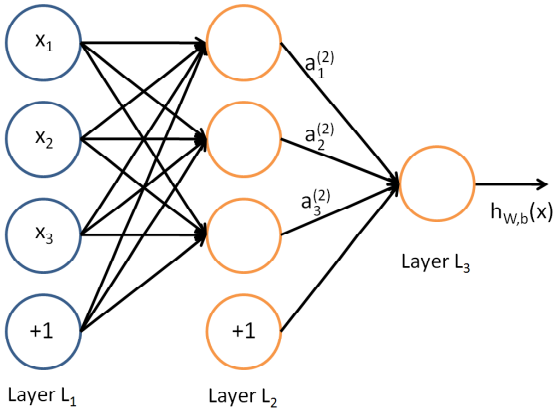
The following plot (from A. Ng) shows the sigmoid, tanh and Relu functions



Note that the choice of decision function will determine the target variable "y" for supervised learning.

The Neural Network model

A neural network is a multi-layer assembly of neurons of the form.
For example, this is a 2-layer network:



The circles labeled +1 are the bias terms.
The circles on the left are the input terms, also called L_1. This is called the input layer. Note that many authors do not consider this to count as a "layer"
The rightmost circle is the output layer (in this case, only one node), also called L_3
The circles in the middle are referred to as a "hidden layer", L_2.

Here we follow the notation used by Andrew Ng.

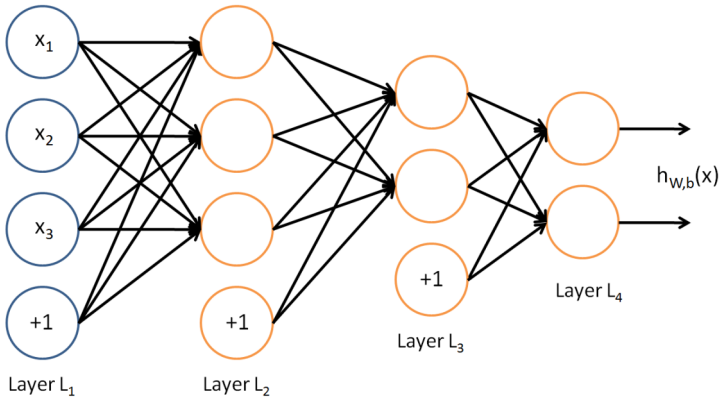
The parameters carry a superscript, referring to their layer.

For example: $a_1^{(2)}$ is the activation output of the first neuron of the second layer.
$W_{13}^{(2)}$ is the weight for input 1 of activation neuron 3 in the second level.

The above network would be described by:

$$a_1^{(2)} = f(w_{11}^{(1)}X_1 + w_{12}^{(1)}X_2 + w_{13}^{(1)}X_3 + b_1^{(1)})$$
$$a_2^{(2)} = f(w_{21}^{(1)}X_1 + w_{22}^{(1)}X_2 + w_{23}^{(1)}X_3 + b_2^{(1)})$$
$$a_3^{(2)} = f(w_{31}^{(1)}X_1 + w_{32}^{(1)}X_2 + w_{33}^{(1)}X_3 + b_3^{(1)})$$
$$h_{\vec{w}b}(\vec{X}) = a_1^{(3)} = f(w_{11}^{(2)}a_1^{(2)} + w_{12}^{(2)}a_2^{(2)} + w_{13}^{(2)}a_3^{(2)} + b_1^{(2)})$$

This can be generalized to multiple layers. For example:



(taken from the Stanford ufldl tutorial, Ng et al)

Note that we can recognize multiple classes by learning multiple $h_{w,b}(x)$ functions.

In general (following the notation of Ng), this can be described by:

L_1 is the input layer.
L_l is the l^{th} layer
L_N is the output layer

$w_{ij}^{(l)}$ denotes the parameter (weight) for the unit j of layer l and the unit i of layer $l+1$.
b_i^l is the bias term for i^{th} using of the the $l+1^{th}$ layer
$a_i^{(l)}$ is the activation i in layer l

Note that many authors prefer to define the input layer as $l=0$. This way the l is the number of hidden layers.

In deriving the regression algorithms for learning, we will use

$$z_i^{(l+1)} = \sum_{j=1}^{N_l} w_{ij}^{(l)} a_i^{(l)} + b^{(l)}$$

It will be more convenient to represent this using vectors: $\vec{z}^{(l)} = \begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_{N_l}^{(l)} \end{bmatrix}$

As defined, the neural network computes a "forward propagation" of the form

$$\vec{z}^{l+1} = \vec{w}^{(l)} \vec{a}^{(l)} + b^{(l)}$$
$$\vec{a}^{(l-1)} = f(\vec{z}^{(l+1)})$$

This is called the “Feed Forward Network”.
Note that feed forward networks do not contain loops.

The weights for a Neural Network are commonly trained using a technique called back-propagation, or “back propagation of errors”.

Back propagation is a form of regression analysis. The most common approach is to employ a form of Gradient Descent.

Gradient descent calculates a loss function for the weights of the network and then iteratively seeks to minimize this loss function. This is commonly performed as a form of supervised learning using labeled training data.

Note that Gradient descent requires that the activation function be differentiable.

Network Structures for Simple Feed-Forward Networks

The architecture of a neural network refers to the number of layers, the number of neurons of each layer and the kind of neurons, and the kinds of neural computation performed.

For a simple feed forward network:

The number of **input neurons** is determined by the number of input values for each observation (size of the feature vector, D)

The number of **output neurons** is the number of classes to be recognized.

The number of hidden layers is determined by the complexity of the recognition and the amount of training data available.

If your training data is linearly separable (if there exists a hyper-plane between the two classes) then you do not even need a hidden layer. Use a simple linear classifiers for example defined by a Support Vector machine or Linear Discriminant Analysis.

A single hidden layer is often sufficient for many problems (although more reliable solutions can be found using Support vector machines or other techniques).

The number of neurons in a hidden layer depends on the complexity of the problem, and the amount of training data available.

In theory, any problem can be solved with a single hidden layer, given enough training data. In practice the quantity of training data is not practical.

Experience shows that some very difficult problems can be solved using less training data by adding additional layers.

However, the spectacular progress of the last few years has been obtained by the introduction of new computational models such as “Convolutional Neural Networks”, “Pooling”, Auto-encoders and Long Term-Short Term Memory. Many of these are actually known pattern recognition techniques recycled into a Neural Network framework.

But before we get to more advanced techniques, we need to look at the basics.

Regression Analysis

The parameters for a feed forward network are commonly learned using regression analysis of labeled training data.

Regression is the estimation of the parameters for a function that maps a set of independent variables into a dependent variable.

$$\hat{y} = f(\vec{X}, \vec{w})$$

Where

$\vec{X}$ is a vector of D independent (unknown) variables.

$\hat{y}$ is an estimate for a variable y that depends on $\vec{X}$.

and

$f()$ is a function that maps $\vec{X}$ onto $\hat{y}$

$\vec{w}$ is a vector of parameters for the model.

Note:

For $\hat{y}$, the “hat” indicates an estimated value for the target value y

$\vec{X}$ is upper case because it is a random (unknown) vector.

Regression analysis refers to a family of techniques for modeling and analyzing the mapping one or more independent variables from a dependent variable.

For example, consider the following table of age, height and weight for 10 females:

M	AGE	H (M)	W (kg)
1	17	163	52
2	32	169	68
3	25	158	49
4	55	158	73
5	12	161	71
6	41	172	99
7	32	156	50
8	56	161	82
9	22	154	56
10	16	145	46

We can use any two variables to estimate the third.
We can use regression to estimate the parameters for a function to predict any feature $\hat{y}$ from the two other features $\vec{X}$.

For example we can predict weight from height and age as a function.

$\hat{y} = f(\vec{X}, \vec{w})$ where $\hat{y} = Weight$, $\vec{X} = \begin{pmatrix} Age \\ Height \end{pmatrix}$ and $\vec{w}$ are the model parameters

Linear Models

A linear model has the form

$$\hat{y} = f(\vec{X}, \vec{w}) = \vec{w}^T \vec{X} + b = w_1 x_1 + w_2 x_2 + ... + w_D x_D + b$$

The vector $\vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{pmatrix}$ are the "parameters" of the model that relates $\vec{X}$ to $\hat{y}$.

The equation $\vec{w}^T \vec{X} + b = 0$ is a hyper-plane in a D-dimensional space,

$\vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{pmatrix}$ is the normal to the hyperplane and b is a constant term.

It is generally convenient to include the constant as part of the parameter vector and to add an extra constant term to the observed feature vector.
This gives a linear model with *D+1* parameters where the vectors are:

$$\vec{X} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_D \end{pmatrix} \text{ and } \vec{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_D \end{pmatrix} \text{ where } w_0 \text{ represents b.}$$

This gives the "homogeneous equation" for the model:

$$\hat{y} = f(\vec{X}, \vec{w}) = \vec{w}^T \vec{X}$$

Homogeneous coordinates provide a unified notation for geometric operations.

With this notation, we can predict weight from height and age using a function learned from regression analysis.

$\hat{y} = f(\vec{X}, \vec{w})$ where $\hat{y} = Weight$, $\vec{X} = \begin{pmatrix} 1 \\ Age \\ Height \end{pmatrix}$ and $\vec{W} = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix}$ are the model parameters, and the surface is a plane in the space (weight, age, height).
In a D dimensional space, linear homogeneous equation is a hyper-plane.
The perpendicular distance of an arbitrary point from the plane is computed as

$$d = w_0 + w_1 x_1 + w_2 x_2$$

This can be used as an error.

Estimation of a hyperplane with supervised learning

In supervised learning, we learn the parameters of a model from a labeled set of training data. The training data is composed of M sets of independent variables, $\{\vec{X}_m\}$ for which we know the value of the dependent variable $\{y_m\}$.
The training data is the set $\{\vec{X}_m\}$, $\{y_m\}$

For a linear model, learning the parameters of the model from a training set is equivalent to estimating the parameters of a hyperplane using least squares.

In the case of a linear model, there are many ways to estimate the parameters:

For example, matrix algebra provides a direct, closed form solution.

Assume a training set of *M* observations $\{\vec{X}_m\}\ \{y_m\}$ where the constant d is included as a "0th" term in $\vec{X}$ and $\vec{w}$.

$$\vec{X} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_D \end{pmatrix} \text{ and } \vec{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_D \end{pmatrix}$$

We seek the parameters for a linear model: $\hat{y} = f(\vec{X}, \vec{w}) = \vec{w}^T \vec{X}$
This can be determined by minimizing a "Loss" function that can be defined as the Square of the error.

$$L(\vec{w}) = \sum_{m=1}^{M} (\vec{w}^T \vec{X}_m - y_m)^2$$

To build or function, we will use the M training samples to compose a matrix **X** and a vector **Y**.

$$X = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ x_{11} & x_{12} & \cdots & x_{1M} \\ x_{21} & x_{22} & \cdots & x_{2M} \\ \cdots & \cdots & \ddots & \vdots \\ x_{D1} & x_{D2} & \cdots & x_{DM} \end{pmatrix} \text{ (D+1 rows by M columns)} \qquad Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{pmatrix} \text{ (M rows).}$$

We can factor the loss function to obtain: $L(\vec{w}) = (\vec{w}^T X - Y)^T (\vec{w}^T X - Y)$

To minimize the loss function, we calculate the derivative and solve for $\vec{w}$ when the derivative is 0.

$$\frac{\partial L(\vec{w})}{\partial \vec{w}} = 2X^T Y - 2X^T X\vec{w} = 0$$

which gives $X^T Y = 2X^T X\vec{w}$ and thus $\vec{w} = (X^T X)^{-1} X^T Y$

While this is an elegant solution for linear regression, it does not generalize to other models. A more general approach is to use Gradient Descent.

Gradient Descent

Gradient descent is a popular algorithm for estimating parameters for a large variety of models. Here we will illustrate the approach with estimation of parameters for a linear model.

As before we seek to estimate that parameters $\vec{w}$ for a model

$$\hat{y} = f(\vec{X}, \vec{w}) = \vec{w}^T \vec{X}$$

from a training set of M samples $\{\vec{X}_m\}$ $\{y_m\}$

We will define our loss function as $\frac{1}{2}$ average error $L(\vec{w}) = \frac{1}{2M} \sum_{m=1}^{M} (f(\vec{X}_m, \vec{w}) - y_m)^2$

where we have included the term $\frac{1}{2}$ to simplify the algebra later.

The gradient is the derivative of the loss function with respect to each term w_d of $\vec{w}$ is

$$\vec{\nabla} f(\vec{X}, \vec{w}) = \frac{\partial f(\vec{X}, \vec{w})}{\partial \vec{w}} = \begin{pmatrix} \frac{\partial f(\vec{X}, w_0)}{\partial w_0} \\ \frac{\partial f(\vec{X}, w_1)}{\partial w_1} \\ \vdots \\ \frac{\partial f(\vec{X}, w_D)}{\partial w_D} \end{pmatrix}$$

where:

$$\frac{\partial f(\vec{X}, w_d)}{\partial w_d} = \frac{1}{M} \sum_{m=1}^{M} (f(\vec{X}_m, \vec{w}) - y_m) x_{dm}$$

x_{dm} is the dth coefficient of the mth training vector. Of course $x_{0m} = 1$ is the constant term.

We use the gradient to "correct" an estimate of the parameter vector for each training sample. The correction is weighted by a learning rate "α"

We can see $\frac{1}{M} \sum_{m=1}^{M} (f(\vec{X}_m, \vec{w}^{(i-1)}) - y_m) x_{dm}$ as the "average error" for parameter $w_d^{(i-1)}$

Gradient descent corrects by subtracting the average error weighted by the learning rate.

Gradient Descent Algorithm

Initialization: *(i=0)* Let $w_d^{(o)} = 0$ for all D coefficients of $\vec{W}$

Repeat until $\left\| L(\vec{w}^{(i)}) - L(\vec{w}^{(i-1)}) \right\| < \varepsilon$: $\vec{w}^{(i)} = \vec{w}^{(i-1)} - \alpha \vec{\nabla} f(\vec{X}, \vec{w}^{(i-1)})$

where $L(\vec{w}) = \frac{1}{2M} \sum_{m=1}^{M} (f(\vec{X}_m, \vec{w}) - y_m)^2$

That is: $w_d^{(i)} = w_d^{(i-1)} - \alpha \frac{1}{M} \sum_{m=1}^{M} (f(\vec{X}_m, \vec{w}^{(i-1)}) - y_m) x_{dm}$

Note that all coefficients are updated in parallel.
The algorithm halts when the change in $\Delta L(\vec{w}^{(i)})$ becomes small:

$$\left\| L(\vec{w}^{(i)}) - L(\vec{w}^{(i-1)}) \right\| < \varepsilon$$

For some small constant ε.

Gradient Descent can be used to learn the parameters for a non-linear model.
For example, when D=2, a second order model would be:

$$\vec{X} = \begin{pmatrix} 1 \\ x_1 \\ x_1^2 \\ x_2 \\ x_1^2 \\ x_1 x_2 \end{pmatrix} \text{ and } f(\vec{X}, \vec{w}) = w_0 + w_1 x_1 + w_2 x_1^2 + w_3 x_2 + w_4 x_2^2 + w_5 x_1 x_2$$

Practical Considerations for Gradient Descent

The following are some practical issues concerning gradient descent.

Feature Scaling

Make sure that features have similar scales (range of values). One way to assure this is to normalize the training date so that each feature has a range of 1.

Simple technique: divide by the range of sample values.

For a training set $\{\vec{X}_m\}$ of M training samples with D values.

Range: $r_D = Max(x_d) - Min(x_d)$

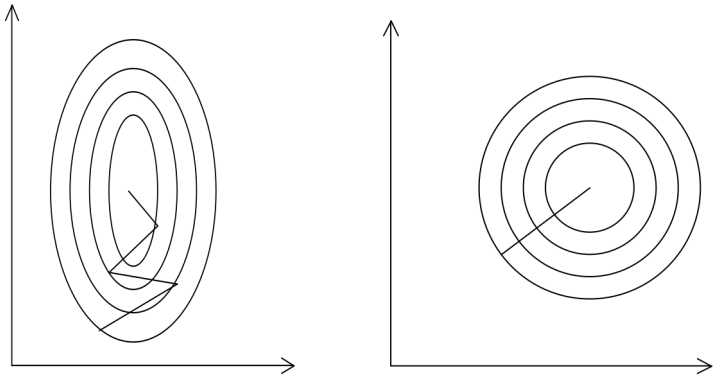
Then

$$\forall_{m=1}^{M} : x_{dm} := \frac{x_{dm}}{r_d}$$

Even better would be to scale with the mean and standard deviation of the each feature in the training data

$$\mu_d = E\{x_{dm}\} \qquad \sigma^2 = E\{(x_{dm} - \mu_d)^2\}$$

$$\forall_{m=1}^{M} : x_{dm} := \frac{(x_{dm} - \mu_d)}{\sigma_d}$$



Note that the value of the loss function should always decrease:

Verify that $L(\vec{w}^{(i)}) - L(\vec{w}^{(i-1)}) < 0$.

if $L(\vec{w}^{(i)}) - L(\vec{w}^{(i-1)}) > 0$ then decrease the learning rate "α"

You can use this to dynamically adjust the learning rate α.
For example, one can start with a high learning rate. Any time that $L(\vec{w}^{(i)}) - L(\vec{w}^{(i-1)}) > 0$
1) reset a ← a/2
2) Recalculate the i^{th} iteration.

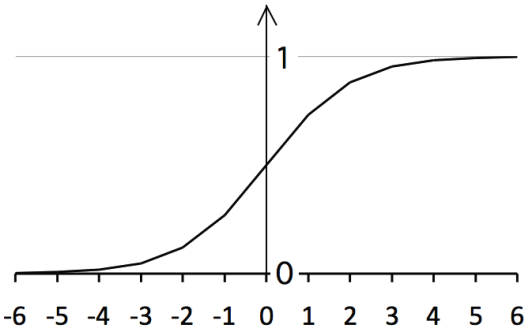
Halt when a < threshold.

Regression for a Sigmoid Activation Function

Gradient descent is easily used to estimate a sigmoid estimation function.

Recall that the sigmoid function has the form $h(\vec{X},\vec{w}) = \frac{1}{1+e^{-g(\vec{X},\vec{w})}}$

This function is differentiable.



When $g_k(\vec{X},\vec{w}) = \vec{w}^T\vec{X}$ $\quad h(\vec{X},\vec{w}) = \frac{1}{1+e^{-\vec{w}^T\vec{X}}}$

and the decision rule is: if $h(\vec{X},\vec{w}) > 0.5$ then Positive else Negative

The cost function for this activation function is:

$$Cost(h(\vec{X},\vec{w}),y) = \begin{cases} -\log(h(\vec{X},\vec{w})) & \text{if } y = 1 \\ -\log(1-h(\vec{X},\vec{w}) & \text{if } y = -1 \end{cases}$$

Thus the loss function would be:

$$L(\vec{w}) = \frac{1}{m}\sum_{m=1}^{M} Cost(h(\vec{X}_m,\vec{w}),y_m)$$

$$L(\vec{w}) = \frac{1}{m}\sum_{m=1}^{M} y_m \log(h(\vec{X}_m,\vec{w}) + (1-y_m)\log(1-h(\vec{X}_m,\vec{w})$$

$$\frac{\partial h(\vec{X},w_d)}{\partial w_d} = \frac{1}{M}\sum_{m=1}^{M}(y_m - h(\vec{X}_m,\vec{w}))x_{dm}$$

Gradient Descent

Repeat until $\left\| L(\vec{w}^{(i)}) - L(\vec{w}^{(i-1)}) \right\| < \varepsilon \quad : \vec{w}^{(i)} = \vec{w}^{(i-1)} - \alpha \vec{\nabla} h(\vec{X}, \vec{w}^{(i-1)})$

where $L(\vec{w}) = \frac{1}{m} \sum_{m=1}^{M} y_m \log(h(\vec{X}_m, \vec{w}) + (1 - y_m) \log(1 - h(\vec{X}_m, \vec{w})$

$$w_d^{(i)} = w_d^{(i-1)} - \alpha \frac{1}{M} \sum_{m=1}^{M} (y_m - h(\vec{X}_m, \vec{w}^{(i-1)})) x_{dm}$$

Regularisation

Overfitting: Tuning the function to the training data.
Overfitting is a common problem with machine learning, particularly when there are many features and not enough training data. One solution is to "regularise" the function by adding an additional term.

replace

$$w_0^{(i)} = w_0^{(i-1)} - \alpha \frac{1}{M} \sum_{m=1}^{M} (y_m - h(\vec{X}_m, \vec{w}^{(i-1)})) x_{0m}$$

with

$$w_d^{(i)} = w_d^{(i-1)} - \alpha \frac{1}{M} \sum_{m=1}^{M} (y_m - h(\vec{X}_m, \vec{w}^{(i-1)})) x_{dm} + \frac{\lambda}{m} w_d^{(i-1)}$$