$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Vanishing gradient problem in deep neural networks

Vanishing Gradient Problem is a difficulty found in training certain Artificial Neural Networks with gradient based methods (e.g Back Propagation). In particular, this problem makes it really hard to learn and tune the parameters of the earlier layers in the network. This problem becomes worse as the number of layers in the architecture increases.
This is not a fundamental problem with neural networks - it's a problem with gradient based learning methods caused by certain activation functions. Let's try to intuitively understand the problem and the cause behind it.

Problem

Gradient based methods learn a parameter's value by understanding how a small change in the parameter's value will affect the network's output. If a change in the parameter's value causes very small change in the network's output - the network just can't learn the parameter effectively, which is a problem.
This is exactly what's happening in the vanishing gradient problem -- the gradients of the network's output with respect to the parameters in the early layers become extremely small. That's a fancy way of saying that even a large change in the value of parameters for the early layers doesn't have a big effect on the output.
Let's try to understand when and why does this problem happen.

Cause

Vanishing gradient problem depends on the choice of the activation function. Many common activation functions (e.g sigmoid or tanh) 'squash' their input into a very small output range in a very non-linear fashion. For example, sigmoid maps the real number line onto a "small" range of [0, 1]. As a result, there are large regions of the input space which are mapped to an extremely small range. In these regions of the input space, even a large change in the input will produce a small change in the output - hence the gradient is small.
This becomes much worse when we stack multiple layers of such non-linearities on top of each other. For instance, first layer will map a large input region to a smaller output region, which will be mapped to an even smaller region by the second layer, which will be mapped to an even smaller region by the third layer and so on. As a result, even a large change in the parameters of the first layer doesn't change the output much.
We can avoid this problem by using activation functions which don't have this property of 'squashing' the input space into a small region. A popular choice is Rectified Linear Unit which maps x to $max(0,x)$.
Hopefully, this helps you understand the problem of vanishing gradients. I'd also recommend reading along this iPython notebook which does a small experiment to understand and visualize this problem, as well as highlights the difference between the behavior of sigmoid and rectified linear units.

Machine learning is solving challenging problems that impact everyone around the world. Problems that we thought were impossible or too complex to solve are now possible with this technology. Using TensorFlow, we've already seen great advancements in many different fields. For example:

- Astrophysicists are using TensorFlow to analyze large amounts of data from the Kepler mission to discover new planets.
- Medical researchers are using ML techniques with TensorFlow to assess a person's cardiovascular risk of a heart attack and stroke.
- Air Traffic Controllers are using TensorFlow to predict flight routes through crowded airspace for safe and efficient landings.
- Engineers are using TensorFlow to analyze auditory data in the rainforest to detect logging trucks and other illegal activities.

•Scientists in Africa are using TensorFlow to detect diseases in Cassava plants to improving yield for farmers.

%**matplotlib** is a magic function in IPython. ... %**matplotlib inline** sets the backend of **matplotlib** to the '**inline**' backend: With this backend, the output of plotting commands is displayed **inline** within frontends like the Jupyter notebook, directly below the code cell that produced it.

Character	Byte order	Size	Alignment
@	native	native	native
=	native	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	network (= big-endian)	standard	none

Format	C Type	Python type	Standard size	Notes
x	pad byte	no value		
c	char	string of length 1	1	
b	signed char	integer	1	(3)
B	unsigned char	integer	1	(3)
?	_Bool	bool	1	(1)
h	short	integer	2	(3)
H	unsigned short	integer	2	(3)
i	int	integer	4	(3)
I	unsigned int	integer	4	(3)
l	long	integer	4	(3)
L	unsigned long	integer	4	(3)
q	long long	integer	8	(2), (3)
Q	unsigned long long	integer	8	(2), (3)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	string		
p	char[]	string		
P	void *	integer		(5), (3)

The Problem of overfitting

A common issue in machine learning or mathematical modeling is overfitting, which occurs when you build a model that not only captures the signal but also the noise in a dataset.

Because we want to create models that generalize and perform well on different data-points, we need to avoid overfitting.

In comes regularization, which is a powerful mathematical tool for reducing overfitting within our model. It does this by adding a penalty for model complexity or extreme parameter values, and it can be applied to different learning models: linear regression, logistic regression, and support vector machines to name a few.

Below is the linear regression cost function with an added regularization component.

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

The regularization component is really just the sum of squared coefficients of your model (your beta values), multiplied by a parameter, lambda.

Lambda

Lambda can be adjusted to help you find a good fit for your model. However, a value that is too low might not do anything, and one that is too high might actually cause you to underfit the model and lose valuable information. It's up to the user to find the sweet spot.

Cross validation using different values of lambda can help you to identify the optimal lambda that produces the lowest out of sample error.

Regularization methods (L1 & L2)

The equation shown above is called Ridge Regression (L2) - the beta coefficients are squared and summed. However, another regularization method is Lasso Regreesion (L1), which sums the absolute value of the beta coefficients. Even more, you can combine Ridge and Lasso linearly to get Elastic Net Regression (both squared and absolute value components are included in the cost function).

L2 regularization tends to yield a "dense" solution, where the magnitude of the coefficients are evenly reduced. For example, for a model with 3 parameters, B1, B2, and B3 will reduce by a similar factor.

However, with L1 regularization, the shrinkage of the parameters may be uneven, driving the value of some coefficients to 0. In other words, it will produce a sparse solution. Because of this property, it is often used for feature selection- it can help identify the most predictive features, while zeroing the others.

It also a good idea to appropriately scale your features, so that your coefficients are penalized based on their predictive power and not their scale.

As you can see, regularization can be a powerful tool for reducing overfitting.

Logistic regression

Command: Statistics
└ Regression
└ Logistic regression

Description

Logistic regression is a statistical method for analyzing a dataset in which there are one or more independent variables that determine an outcome. The outcome is measured with a dichotomous variable (in which there are only two possible outcomes).

In logistic regression, the dependent variable is binary or dichotomous, i.e. it only contains data coded as 1 (TRUE, success, pregnant, etc.) or 0 (FALSE, failure, non-pregnant, etc.).

The goal of logistic regression is to find the best fitting (yet biologically reasonable) model to describe the relationship between the dichotomous characteristic of interest (dependent variable = response or outcome variable) and a set of independent (predictor or explanatory) variables. Logistic regression generates the coefficients (and its standard errors and significance levels) of a formula to predict a *logit transformation* of the probability of presence of the characteristic of interest:

$$logit(p) = b_0 + b_1X_1 + b_2X_2 + b_3X_3 + \ldots + b_kX_k$$

where p is the probability of presence of the characteristic of interest. The logit transformation is defined as the logged odds:

$$odds = \frac{p}{1-p} = \frac{probability\ of\ presence\ of\ characteristic}{probability\ of\ absence\ of\ characteristic}$$

and

$$logit(p) = \ln\left(\frac{p}{1-p}\right)$$

The derivative of logistic cost function

In what follows, the superscript (i) denotes individual measurements or training "examples."

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

$$\underset{\text{linearity}}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \frac{\partial}{\partial \theta_j} \log\left(h_\theta \left(x^{(i)} \right) \right) + (1 - y^{(i)}) \frac{\partial}{\partial \theta_j} \left(\log(1 - h_\theta \left(x^{(i)} \right) \right) \right]$$

$$\underset{\text{chain rule}}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \frac{\frac{\partial}{\partial \theta_j} \left(h_\theta \left(x^{(i)} \right) \right.}{h_\theta \left(x^{(i)} \right)} + (1 - y^{(i)}) \frac{\frac{\partial}{\partial \theta_j} \left(1 - h_\theta \left(x^{(i)} \right) \right)}{1 - h_\theta \left(x^{(i)} \right)} \right]$$

$$\underset{h_\theta(x) = \sigma(\theta^\top x)}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \frac{\frac{\partial}{\partial \theta_j} \sigma \left(\theta^\top x^{(i)} \right)}{h_\theta \left(x^{(i)} \right)} + (1 - y^{(i)}) \frac{\frac{\partial}{\partial \theta_j} \left(1 - \sigma \left(\theta^\top x^{(i)} \right) \right)}{1 - h_\theta \left(x^{(i)} \right)} \right]$$

$$\underset{\sigma'}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \frac{\sigma \left(\theta^\top x^{(i)} \right) \left(1 - \sigma \left(\theta^\top x^{(i)} \right) \right) \frac{\partial}{\partial \theta_j} \left(\theta^\top x^{(i)} \right)}{h_\theta \left(x^{(i)} \right)} \right.$$

$$\left. - (1 - y^{(i)}) \frac{\sigma \left(\theta^\top x^{(i)} \right) \left(1 - \sigma \left(\theta^\top x^{(i)} \right) \right) \frac{\partial}{\partial \theta_j} \left(\theta^\top x^{(i)} \right)}{1 - h_\theta \left(x^{(i)} \right)} \right]$$

$$\underset{\sigma(\theta^\top x) = h_\theta(x)}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \frac{h_\theta \left(x^{(i)} \right) \left(1 - h_\theta \left(x^{(i)} \right) \right) \frac{\partial}{\partial \theta_j} \left(\theta^\top x^{(i)} \right)}{h_\theta \left(x^{(i)} \right)} \right.$$

$$\left. - (1 - y^{(i)}) \frac{h_\theta \left(x^{(i)} \right) \left(1 - h_\theta \left(x^{(i)} \right) \right) \frac{\partial}{\partial \theta_j} \left(\theta^\top x^{(i)} \right)}{1 - h_\theta \left(x^{(i)} \right)} \right]$$

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

$$\underset{\text{distribute}}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[y^i - y^i h_\theta \left(x^{(i)} \right) - h_\theta \left(x^{(i)} \right) + y^{(i)} h_\theta \left(x^{(i)} \right) \right] x_j^{(i)}$$

$$\underset{\text{cancel}}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} - h_\theta \left(x^{(i)} \right) \right] x_j^{(i)}$$

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

The derivative of the sigmoid function is

$$\frac{d}{dx}\sigma(x) = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right)$$

$$= \frac{-(1+e^{-x})'}{(1+e^{-x})^2}$$

$$= \frac{e^{-x}}{(1+e^{-x})^2}$$

$$= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{e^{-x}}{1+e^{-x}}\right)$$

$$= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right)$$

$$= \sigma(x)\left(\frac{1+e^{-x}}{1+e^{-x}} - \sigma(x)\right)$$

$$= \sigma(x)\,(1-\sigma(x))$$

A regression model that uses L1 regularization technique is called ***Lasso Regression*** and model which uses L2 is called ***Ridge Regression***.

The key difference between these two is the penalty term.

Ridge regression adds "*squared magnitude*" of coefficient as penalty term to the loss function. Here the *highlighted* part represents L2 regularization element.

Here, if *lambda* is zero then you can imagine we get back OLS. However, if *lambda* is very large then it will add too much weight and it will lead to under-fitting. Having said that it's important how *lambda* is chosen. This technique works very well to avoid over-fitting issue.

Lasso Regression (Least Absolute Shrinkage and Selection Operator) adds "*absolute value of magnitude*" of coefficient as penalty term to the loss function.

Again, if *lambda* is zero then we will get back OLS whereas very large value will make coefficients zero hence it will under-fit.

The **key difference** between these techniques is that Lasso shrinks the less important feature's coefficient to zero thus, removing some feature altogether. So, this works well for **feature selection** in case we have a huge number of features.

Traditional methods like cross-validation, stepwise regression to handle overfitting and perform feature selection work well with a small set of features but these techniques are a great alternative

when we are dealing with a large set of features.

Cost function

Unfortunately we can't (or at least shouldn't) use the same cost function MSE (L2) as we did for linear regression. Why? There is a great math explanation in chapter 3 of Michael Neilson's deep learning book [5], but for now I'll simply say it's because our prediction function is non-linear (due to sigmoid transform). Squaring this prediction as we do in MSE results in a non-convex function with many local minimums. If our cost function has many local minimums, gradient descent may not find the optimal global minimum.

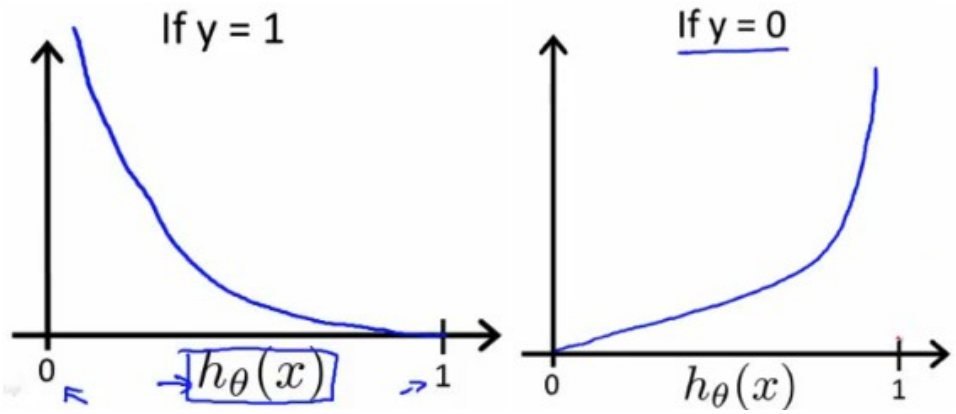
Math

Instead of Mean Squared Error, we use a cost function called Cross-Entropy, also known as Log Loss. Cross-entropy loss can be divided into two separate cost functions: one for y=1 and one for y=0.

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

$$\text{Cost}(h_\theta(x), y) = -\log(h_\theta(x)) \qquad \text{if y} = 1$$
$$\text{Cost}(h_\theta(x), y) = -\log(1 - h_\theta(x)) \qquad \text{if y} = 0$$

The benefits of taking the logarithm reveal themselves when you look at the cost function graphs for y=1 and y=0. These smooth monotonic functions [7] (always increasing or always decreasing) make it easy to calculate the gradient and minimize cost. Image from Andrew Ng's slides on logistic regression [1].



The key thing to note is the cost function penalizes confident and wrong predictions more than it rewards confident and right predictions! The corollary is increasing prediction accuracy (closer to 0 or 1) has diminishing returns on reducing cost due to the logistic nature of our cost function.

Above functions compressed into one

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))]$$

Multiplying by y and $(1-y)$ in the above equation is a sneaky trick that let's us use the same equation to solve for both y=1 and y=0 cases. If y=0, the first side cancels out. If y=1, the second side cancels out. In both cases we only perform the operation we need to perform.