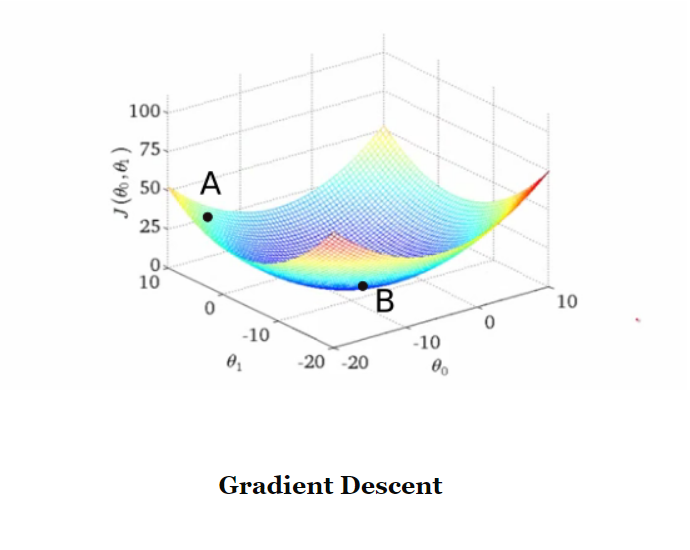
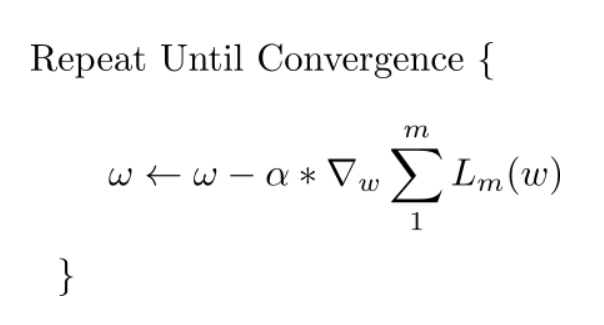
## **Gradient Descent**

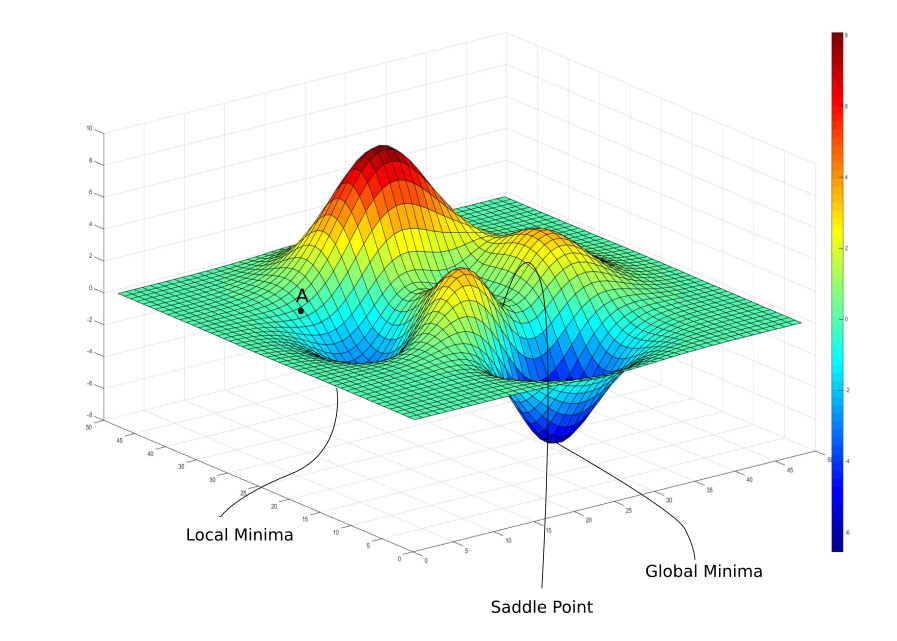
When we initialize our weights, we are at point A in the loss landscape. The first thing we do is to check, out of all possible directions in the x-y plane, moving along which direction brings about the steepest decline in the value of the loss function. This is the direction we have to move in. This direction is given by the direction exactly opposite to the direction of the gradient. The gradient, the higher dimensional cousin of derivative, gives us the direction with the steepest ascent.

To wrap your head around it, consider the following figure. At any point of our curve, we can define a plane that is tangential to the point. In higher dimensions, we can always define a hyperplane, but let's stick to 3-D for now. Then, we can have infinite directions on this plane. Out of them, precisely one direction will give us the direction in which the function has the steepest ascent. This direction is given by the gradient. The direction opposite to it is the direction of steepest descent. This is how the algorithm gets it's name. We perform descent along the direction of the gradient, hence, it's called Gradient Descent.

## **Basic Equations**

The basic equation that describes the update rule of gradient descent is.

## **Challenges with Gradient Descent #1: Local Minima**

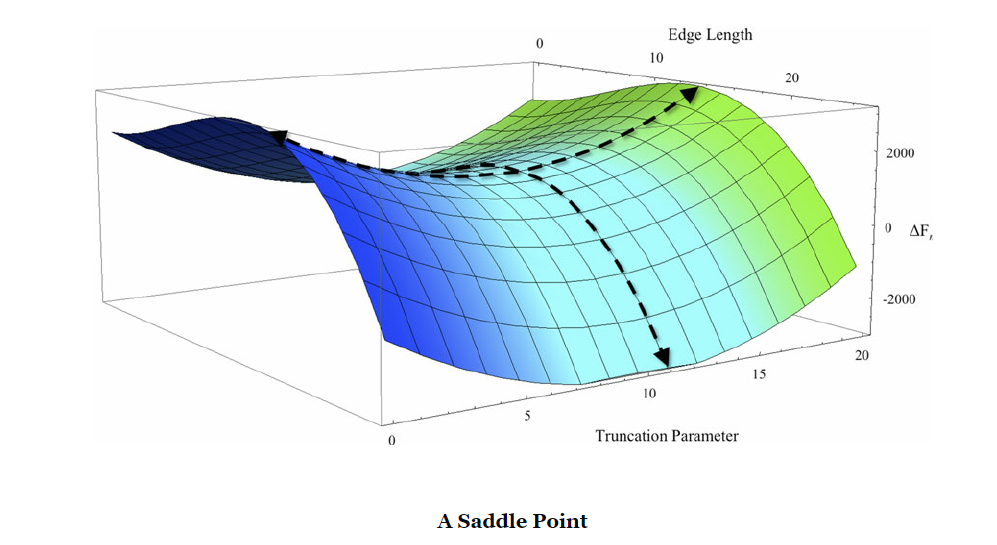
First, neural networks are complicated functions, with lots of non-linear transformations thrown in our hypothesis function. The resultant loss function doesn't look a nice bowl, with only one minima we can converge to. In fact, such nice santa-like loss functions are called **convex** functions (functions for which are always curving upwards) , and the loss functions for deep nets are hardly convex. In fact, they may look like this.

In the above image, there exists a local minima where the gradient is zero. However, we know that they are not the lowest loss we can achieve, which is the point corresponding to the global minima. Now, if you initialze your weights at point A, then you're gonna converge to the local minima, and there's no way gradient descent will get you out of there, once you converge to the local minima.

Gradient descent is driven by the gradient, which will be zero at the base of any minima. Local minimum are called so since the value of the loss function is minimum at that point in a local region. Whereas, a global minima is called so since the value of the loss function is minimum there, *globally* across the entire domain the loss function.

## **Challenges with Gradient Descent #2: Saddle Points**

The basic lesson we took away regarding the limitation of gradient descent was that once it arrived at a region with gradient zero, it was almost impossible for it to escape it regardless of the quality of the minima. Another sort of problem we face is that of saddle points, which look like this.

A saddle point gets it's name from the saddle of a horse with which it resembles. While it's a minima in one direction (*x*), it's a local maxima in another direction, and if the contour is flatter towards the x direction, GD would keep oscillating to and fro in the y - direction, and give us the illusion that we have converged to a minima.

## **Modification of weighting coefficients by means of gradient descent. The Code**

Now we will consider the use of gradient descent in logistic regression. Fortunately, our derivative is expressed by subtraction and multiplication, so it will not be difficult.

We take most of the code from the previous program.

import numpy as np

N = 100

D = 2

X = np.random.randn(N,D)

X[:50, :] = X[:50, :] – 2\*np.ones((50, D))

X[:50, :] = X[:50, :] + 2\*np.ones((50, D))

T = np.array([0]\*50 + [1]\*50)

ones = np.array([[1]\*N]).T

Xb = np.concatenate((ones, X), *axis*=1)

w = np.random.randn(D + 1)

z = Xb.dot(w)

*def* sigmoid(*z*):

return 1/(1 + np.exp(-z))

Y = sigmoid(z)

*def* cross\_entropy(*T, Y*):

E = 0

for i in *xrange*(N)

if T[i] == 1:

E -= np.log(Y[i])

else:

E -= np.log(1 – Y[i])

return E

print cross\_entropy (T, Y)

So, we have two normally distributed classes, one of which centers to a point with coordinates (-2; -2), and the second one centers to a coordinate point (+2; +2). We remove the individual solution for the Bayes classifier. We set the learning factor to 0.1, make 100 iterations and display the result.

learning\_rate = 0.1

for i in *xrange*(100)

if i % 10 == 0:

print cross\_entropy(T, Y)

w += learning\_rate \* np.dot((T-Y).T, Xb)

Y = sigmoid(Xb.dot(w))

print ‘’Final w:’’, w

When we run the program, we see that the error sharply decreases after about 30 iterations. The weighting coefficients are found: one is at zero and two values are at 14. Although this is an extension of the particular case of the solution, the values, which we have obtained, seem to be too large.

.