```
%pylab inline
from part2 import *
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
NN = NeuralNetwork()
```

```
X
```

```
array([[ 0.3,  1. ],
       [ 0.5,  0.2],
       [ 1. ,  0.4]])
```

```
yHat = NN.forwardPropagation(X)
```

```
yHat
```

```
array([[ 0.64154387],
       [ 0.62463713],
       [ 0.59552991]])
```

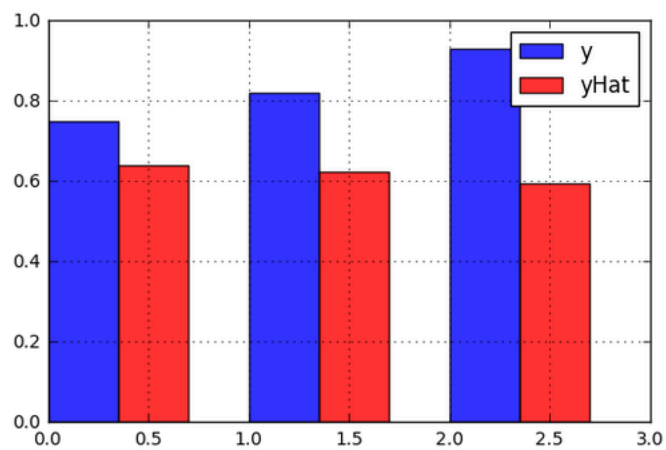
```
y
```

```
array([[ 0.75],
       [ 0.82],
       [ 0.93]])
```

Plot looks like this:

```
#Compare estimate, yHat, to actually score
bar([0,1,2], y, width = 0.35, alpha=0.8)
bar([0.35,1.35,2.35],yHat, width = 0.35, color='r', alpha=0.8)
grid(1)
legend(['y', 'yHat'])
```

```
<matplotlib.legend.Legend at 0x7fca5f3c25d0>
```



We can see our predictions ($\hat{y}$) are pretty inaccurate!

Cost function J

(KPCA) analysis (/python/scikit-learn/scikit_machine_learning_Da nonlinear-mappings-via-kernel-principal-component-analysis.php)

scikit-learn : Logistic Regression, Overfitting & regularization (/python/scikit-learn/scikit-learn_logistic_regression.php)

scikit-learn : Supervised Learning & Unsupervised Learning - e.g. Unsupervised PCA dimensionality reduction with iris dataset (/python/scikit-learn/scikit_machine_learning_Su

scikit-learn : Unsupervised_Learning - KMeans clustering with iris dataset (/python/scikit-learn/scikit_machine_learning_Un

scikit-learn : Linearly Separable Data - Linear Model & (Gaussian) radial basis function kernel (RBF kernel) (/python/scikit-learn/scikit_machine_learning_Li

scikit-learn : Decision Tree Learning I - Entropy, Gini, and Information Gain (/python/scikit-learn/scikt_machine_learning_De

scikit-learn : Decision Tree Learning II - Constructing the Decision Tree (/python/scikit-learn/scikit_machine_learning_Co

scikit-learn : Random Decision Forests Classification (/python/scikit-learn/scikit_machine_learning_Ra

scikit-learn : k-Nearest Neighbors (k-NN) Algorithm (/python/scikit-learn/scikit_machine_learning_k-

To improve our poor model, we first need to find a way of quantifying exactly how wrong our predictions are.

One way of doing it is to use a **cost function**. For a given sample, a cost function tells us how costly our models is.

We'll use sum of square errors to compute an overall cost and we'll try to minimize it. Actually, training a network means minimizing a cost function.

$$J = \sum_{i=1}^{N} (y_i - \hat{y}_i)$$

where the N is the number of training samples.

As we can see from equation, the cost is a function of two things: our **sample data** and the **weights** on our synapses. Since we don't have much control of our data, we'll try to minimize our cost by changing the **weights**.

We have a collection of 9 weights:

$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} W_{11}^{(2)} \\ W_{21}^{(2)} \\ W_{31}^{(2)} \end{bmatrix}$$

and we're going to make our cost (J) as small as possible with a optimal combination of the weights.

Curse of dimensionality

Well, we're not there yet. Considering the 9 weights, finding the right combination that gives us minimum J may be costly.

Let's try the case when we tweek only one weight value ($W_{11}^{(1)}) in the range [-5,5] with 1000 try. Other weights remain untouched with the values of randomly initialized in "__init__()" method:

MACHINE LEARNING ALGORITHMS

```python
class NeuralNetwork(object):
    def __init__(self):
        #Define Hyperparameters
        self.inputLayerSize = 2
        self.outputLayerSize = 1
        self.hiddenLayerSize = 3

        #Weights (parameters)
        self.W1 = np.random.randn(self.inputLayerSize, self.hiddenLayerSize)
        self.W2 = np.random.randn(self.hiddenLayerSize, self.outputLayerSize)
```

Here is the code for the 1-weight:

```python
import time
weightToTry = np.linspace(-5,5,1000)
costs = np.zeros(1000)

startTime = time.clock()
for i in range(1000):
    NN.W1[0,0] = weightToTry[i]
    yHat = NN.forwardPropagation(X)
    costs[i] = 0.5*sum((y-yHat)**2)
endTime = time.clock()
```

```python
elaspsedTime = endTime - startTime
elaspsedTime
```

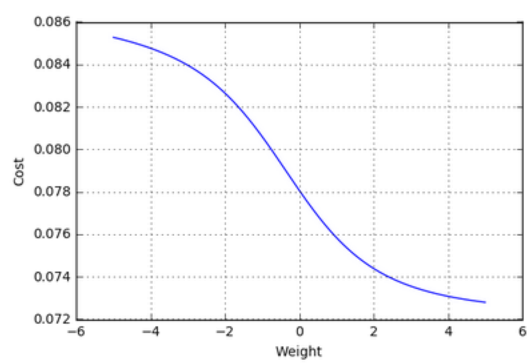
```
0.11225400000000008
```

It takes about 0.11 seconds to check 1000 different weight values for our neural network. Since we've computed the cost for a wide range values of W, we can just pick the one with the smallest cost, let that be our weight, and we've trained our network.

Here is the plot for the 1000 weights:

```python
plot(weightToTry, costs)
grid(1)
ylabel('Cost')
xlabel('Weight')
```

```
<matplotlib.text.Text at 0x7fca5f192310>
```



Note that we have 9! But this time, let's do just 2 weights. To maintain the same precision we now need to check 1000 times 1000, or one million values:

Two-Weight-ElaspsedTimeCode.png

Single Layer Neural Network - Perceptron model on the Iris dataset using Heaviside step activation function (/python/scikit-learn/Perceptron_Model_with_Iri

Batch gradient descent versus stochastic gradient descent (SGD) (/python/scikit-learn/scikit-learn_batch-gradient-descent-versus-stochastic-gradient-descent.php)

Single Layer Neural Network - Adaptive Linear Neuron using linear (identity) activation function with batch gradient descent method (/python/scikit-learn/Single-Layer-Neural-Network-Adaptive-Linear-Neuron.php)

Single Layer Neural Network : Adaptive Linear Neuron using linear (identity) activation function with stochastic gradient descent (SGD) (/python/scikit-learn/Single-Layer-Neural-Network-Adaptive-Linear-Neuron-with-Stochastic-Gradient-Descent.php)

VC (Vapnik-Chervonenkis) Dimension and Shatter (/python/scikit-learn/scikit_machine_learning_V(

Bias-variance tradeoff (/python/scikit-learn/scikit_machine_learning_Bi variance-Tradeoff.php)

Logistic Regression (/python/scikit-learn/logistic_regression.php)

Maximum Likelihood Estimation (MLE) (/python/scikit-learn/Maximum-

For 1 million evaluations, it took an 100 seconds! The real curse of dimensionality kicks in as we continue to add dimensions. Searching through three weights would take a billion evaluations, 100*1000 sec = 27 hrs!

For our all 9 weights, it could take "1,268,391,679,350", 1 trillion millenium!:

```
0.04*(1000**(9-1))/(3600*24*365)/1000
```

```
1268391679350.5835
```

Gradient descent method

So, we may want to use **gradient descent** algorithm to get the weights that take J to minimum. Though it may not seem so impressive in one dimension, it is capable of incredible speedups in higher dimensions.

Actually, I wrote couple of articles on **gradient descent** algorithm:

1. Batch gradient descent algorithm (/python/python_numpy_batch_gradient_descent_algorithm.php)
2. Batch gradient descent versus stochastic gradient descent (SGD) (/python/scikit-learn/scikit-learn_batch-gradient-descent-versus-stochastic-gradient-descent.php)
3. Single Layer Neural Network - Adaptive Linear Neuron using linear (identity) activation function with batch gradient descent method (/python/scikit-learn/Single-Layer-Neural-Network-Adaptive-Linear-Neuron.php)
4. Single Layer Neural Network : Adaptive Linear Neuron using linear (identity) activation function with stochastic gradient descent (SGD) (/python/scikit-learn/Single-Layer-Neural-Network-Adaptive-Linear-Neuron-with-Stochastic-Gradient-Descent.php)

Though we have two choices of the **gradient descent**: batch(standard) or stochastic, we're going to use the **batch** to train our Neural Network.

In **batch** gradient descent method sums up all the derivatives of J for all samples:

$$\sum \frac{\partial J}{\partial W}$$

while the **stochastic** gradient descent (SGD) method uses one derivative at one sample and move to another sample point:

$$\frac{\partial J}{\partial W}$$

Next:

Likelyhood-Estimation-MLE.php)

Neural Networks with backpropagation for XOR using one hidden layer (/python/python_Neural_Networ

minHash (/Algorithms/minHash_Jaccard_S

tf-idf weight (/Algorithms/tf_idf_term_frequen

Natural Language Processing (NLP): Sentiment Analysis I (IMDb & bag-of-words) (/Algorithms/Machine_Learning_I

Natural Language Processing (NLP): Sentiment Analysis II (tokenization, stemming, and stop words) (/Algorithms/Machine_Learning_I

Natural Language Processing (NLP): Sentiment Analysis III (training & cross validation) (/Algorithms/Machine_Learning_I

Natural Language Processing (NLP): Sentiment Analysis IV (out-of-core) (/Algorithms/Machine_Learning_I

Locality-Sensitive Hashing (LSH) using Cosine Distance (Cosine Similarity) (/Algorithms/Locality_Sensitive_H

ARTIFICIAL NEURAL NETWORKS (ANN)

1. Introduction (/python/scikit-learn/Artificial-Neural-Network-ANN-1-Introduction.php)

2. Forward Propagation