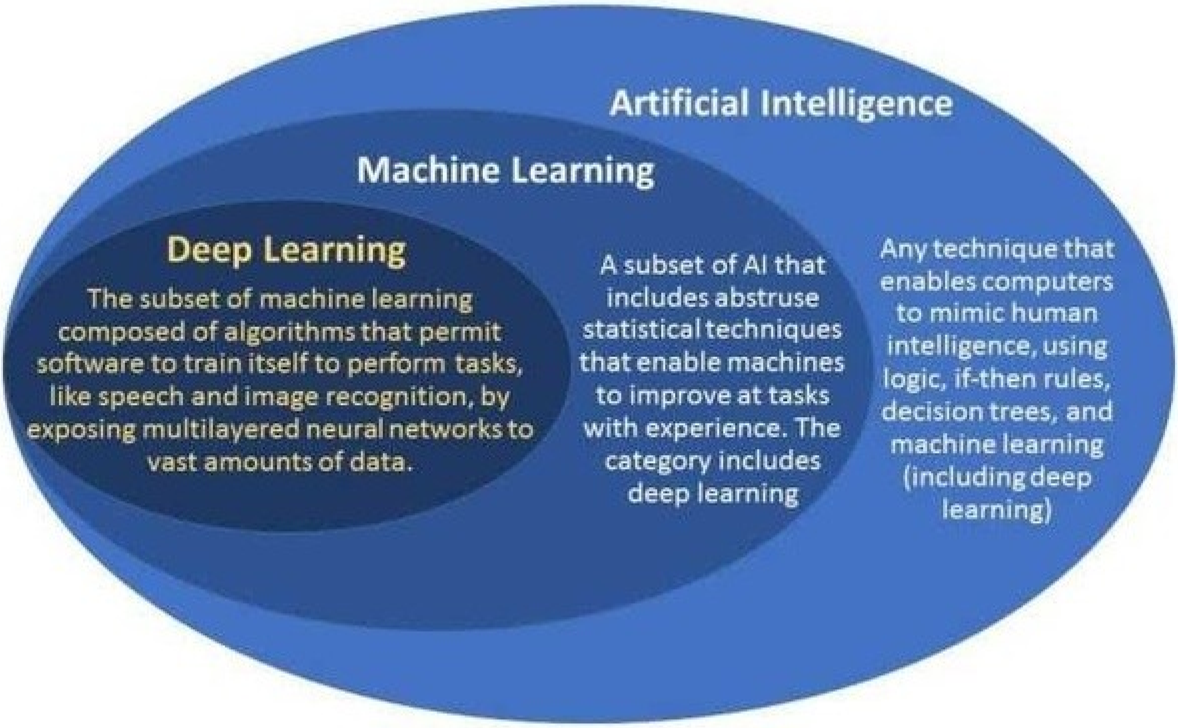
What is Machine Learning/Deep Learning?

Machine Learning is learning from data. The mathematical intuition behind machine learning is to map a function y = f(x) given y and x. We have to approximate f(x) so that the loss is as low as possible. Loss means the error.



What Machine Learning Can Do

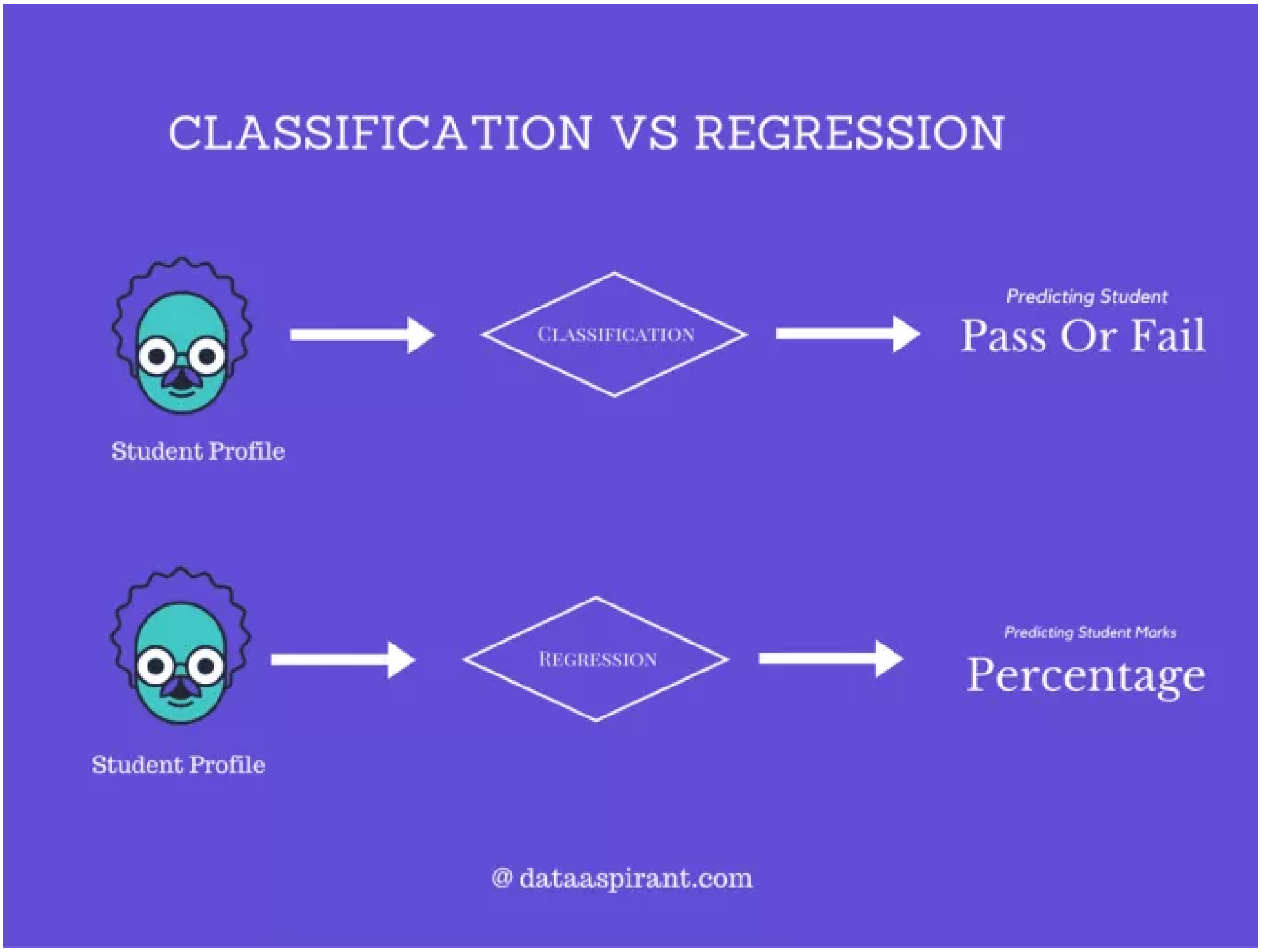
A simple way to think about supervised learning.

INPUT A	RESPONSE B	APPLICATION
Picture	Are there human faces? (0 or 1)	Photo tagging
Loan application	Will they repay the loan? (0 or 1)	Loan approvals
Ad plus user information	Will user click on ad? (0 or 1)	Targeted online ads
Audio clip	Transcript of audio clip	Speech recognition
English sentence	French sentence	Language translation
Sensors from hard disk, plane engine, etc.	Is it about to fail?	Preventive maintenance
Car camera and other sensors	Position of other cars	Self-driving cars

SOURCE ANDREW NG

Machine Learning problem types

-



Python and Numpy (Quick Primer)

Python is a simple and an easy to use programming language.
Numpy is the python library for scientific computing.
You can install numpy using pip.
Pip is similar to Cocoapods and is used to install third party packages in your projects.

Download Python and PIP here: [https://www.python.org/ (https://www.python.org/)](https://www.python.org/) and then in the terminal you can install the packages:
pip3 install numpy
pip3 install keras
pip3 install torch

In [16]:

```
# To print
print ("Hello World")

# Define a function
def sum(a,b):
    return a+b

# Conditional statements
name = "aadit"
if name == "aadit":
    print ("great!")
else:
    print ("okay!")


# Loops
for i in range(5): # This will run for five times
    print (i)
# This is the infinite loop
#while True:
#   pass

# Python also provides dictionaries (key value pair)
cost = {"2000": "bread", "1000": "juice", "5000": "food"}
print (cost["2000"]) # We can access the elements like this.


# Python arrays are called lists
price = [100,200,300,500, 600, 100,500, 600,100, 800, 100]

print (price[0]) # This will access the first element
print (price[1]) # This will access the second element
print (price[2]) # and so on...
print (price[3])
print (price[4])
print (price[5])
print (price[6])


# To define a class
class Person:
    def __init__(self,name): # This is the constructor (self is the instance of
        self.name = name
    def say(self):
        print (self.name)


person = Person("aadit") # You create an instance of the class
person.say() # calling the function

# Inheritance is also supported
# class Name(base_class)
```

```
Hello World
great!
0
1
2
3
```

```
4
bread
100
200
300
500
600
100
500
aadit
```

- In Python the indentation is important.

Now let us see how to use Numpy

Numpy is a package for scientifc computing. Using numpy we can perform linear algebra (matrix multiplication etc)
Numpy arrays are faster that regular Python arrays.

In [1]:

```
import numpy as np # import is used to import a package

a = np.array([0,1,2,3]) # a vector
b = np.array([4,5,6,7]) # another vector
c = np.array([[0,1,2,3], # a matrix
              [4,5,6,7]])

d = np.zeros((5,4)) # (2x4 matrix of zeros)
e = np.random.rand(1,5) # random 2x5
# matrix with all numbers between 0 and 1

print(a)
print(b)
print(c)
print(d)
print(e)

print ("Shapes")
print ("=========")
# (row, column)
print(a.shape)
print (b.shape)
print (c.shape)
print (d.shape)

print (a.T)
print (b.T)
```

```
[0 1 2 3]
[4 5 6 7]
[[0 1 2 3]
 [4 5 6 7]]
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[0.66772435 0.49314048 0.01338067 0.66895993 0.94599639]]
Shapes
=========
(4,)
(4,)
(2, 4)
(5, 4)
[0 1 2 3]
[4 5 6 7]
```

- A shape in a numpy array is nothing but the order of the vector in the form (rows X columns)

In [8]:

```
print(a * 0.1) # multiplies every number in vector "a" by 0.1

print(c * 0.2) # multiplies every number in matrix "c" by 0.2

print(a * b) # multiplies elementwise between a and b (columns paired up)

print(a * b * 0.2) # elementwise multiplication then multiplied by 0.2

print(a * c) # since c has the same number of columns as a, this performs
# elementwise multiplication on every row of the matrix "c"
```

```
[0.  0.1 0.2 0.3]
[[0.  0.2 0.4 0.6]
 [0.8 1.  1.2 1.4]]
[ 0  5 12 21]
[0.  1.  2.4 4.2]
[[ 0  1  4  9]
 [ 0  5 12 21]]
```

Neural Networks: What, Why, How

A Short Introduction

By Aadit Kapoor

Index

- Neural Networks - What
- Neural Networks - Why
- Neural Networks - How
- Neural Networks - Solving a simple classification problem
- Neural Networks - Conversion into Core ML
- A short intro to other machine learning frameworks (Scikit Learn, Tensorflow, Keras, Tensorflow For Swift)
- Thank You! Questions?

Let's us talk about what are neural networks?

- Machine Learning Algorithm (Like SVM, Decision Tree etc)
- Performs well when given a lot of data (Deep Learning)
- Loosely based on how the brain works

History

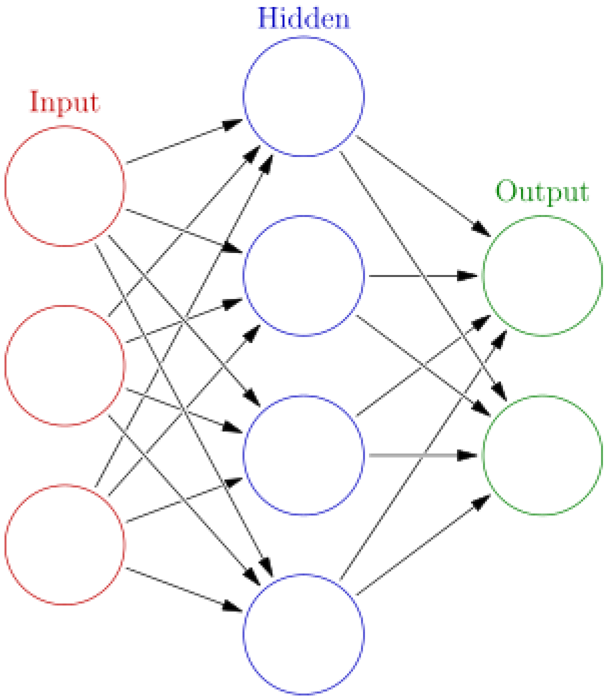
- Warren McCulloch and Walter Pitts[2] (1943) created a computational model for neural networks based on mathematics and algorithms called threshold logic. This model paved the way for neural network research to split into two approaches. One approach focused on biological processes in the brain while the other focused on the application of neural networks to artificial intelligence. This work led to work on nerve networks and their link to finite automata.[3]

Paper: http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf (http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf)

- The perceptron algorithm was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt,[3] funded by the United States Office of Naval Research.[4] The perceptron was intended to be a machine, rather than a program, and while its first implementation was in software for the IBM 704, it was subsequently implemented in custom-built hardware as the "Mark 1 perceptron". This machine was designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors.

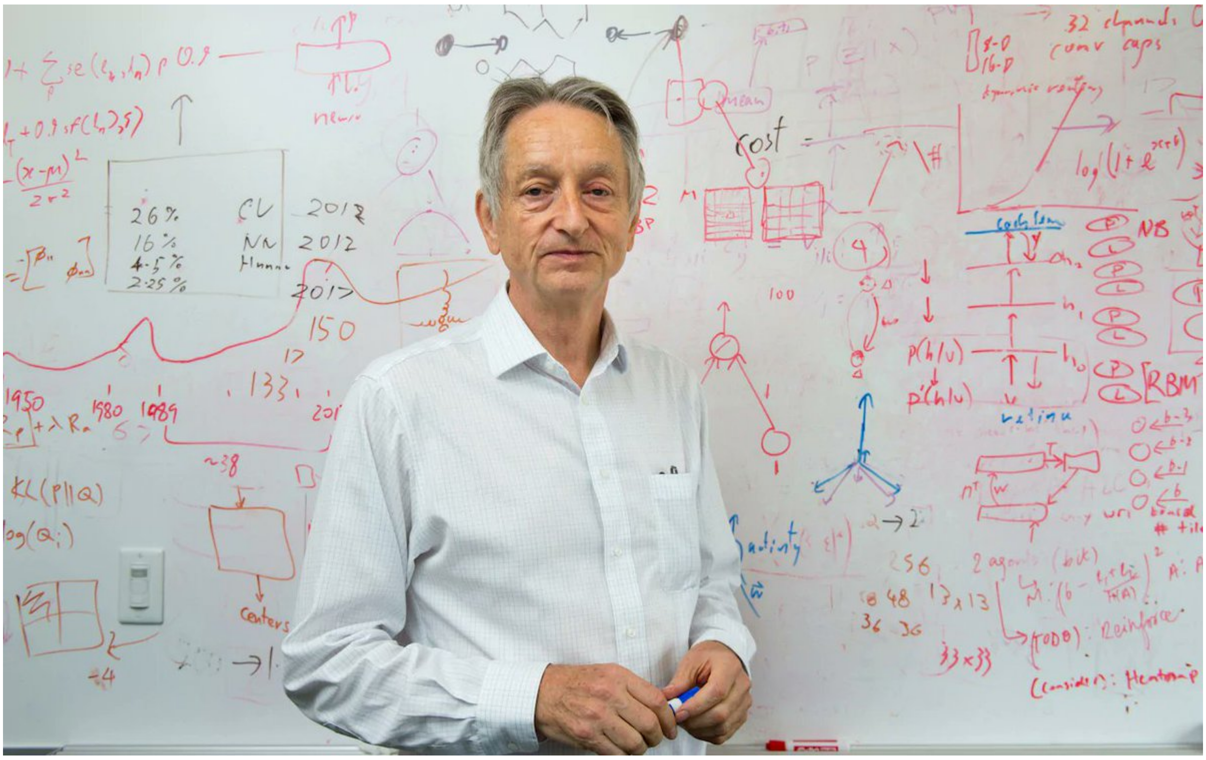
Paper: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=pdf (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=pdf)

Neural Networks



- Given above is a neural network with one hidden layer, 3 inputs and 2 outputs.
- Each layer has a bias node (circle) except the output layer.
- This is a fully connected i.e everything is connected.

- **Mcculloch and Pitts**



- **Geoffrey Hinton**

- **Frank Rosenblatt (Perceptron)**

Properties of a ANN

Components of a neural network

- Neurons (Nodes) (Circles)
- Weights
- Activation Functions (ReLU, tanh, sigmoid, leaky ReLU, selu, elu, etc)
- Loss Function to minimize (also called cost function) (eg: crossentropy, mse, mae etc)
- An optimization technique (Adam (form 1), Gradient Descent, RMSProp etc)
- The NN employs an algorithm called **backpropagation to calculate the gradient (derivative) of the loss (cost) function with respect to the model parameters (weights and bias)** then an optimize algorithm is used to get the direction of the descent and the parameters are updated.
- Backprop is basically using the chain rule cleverly.
- The output neurons are employed with a activation functions typically a softmax (log softmax) or sigmoid.
- Common problems faced by neural nets are vanishing gradient problem and exploding gradient problem.
- **Backprop paper (1986):** [https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf (https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf)](https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf)

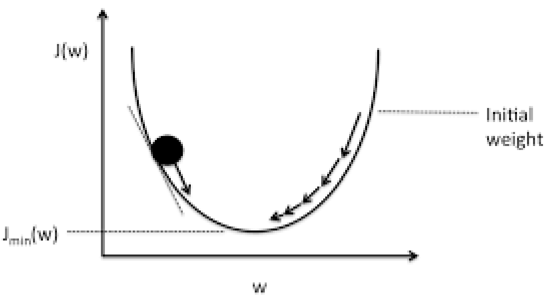
Layers in a neural network.

A basic neural network is defined as having an input layer, some hidden layers and an output layer.
The fully connected (the one you saw above) applies a affine transformation (wx+b). The main idea about neural network is that we update the parameters so that the prediction is as close to the desired output. This is done by changing the weights and biases. A neural network is basically a huge composite function (f(g(h(x))). For any machine learning algorithm our job is to map a function y = f(x) (to find the f(x) given x and y) (input and output).

A neural network perform a weighted sum (sigma(wx+b) and performs an activation to pass the output onto to the next layer.

Gradient Descent

- [http://www.math.usm.edu/lambers/mat419/lecture10.pdf (http://www.math.usm.edu/lambers/mat419/lecture10.pdf)](http://www.math.usm.edu/lambers/mat419/lecture10.pdf)



Schematic of gradient descent.

- The main idea about the algorithm is that we need the reach the local minimum (where the loss is as low as possible).
- To do this we first randomly intialize the weight and descent in direction negative or positive of the gradient.
- new_weight = weight - (alpha * gradient)
- Gradient is the derivative of the loss calculated with respect to the parameters of the model. Alpha (hyperparameter) is the learning rate that basically tells how fast we should go (default value is < 1)
- Gradients are calculated using the backprop alogrithm.

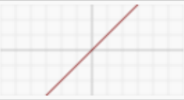
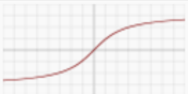
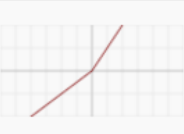
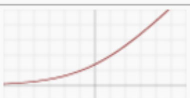
- There are two passes done in a neural network (forward pass and backward pass), the error is calculated at the output neuron and weights are updated accordingly.

Activation Functions

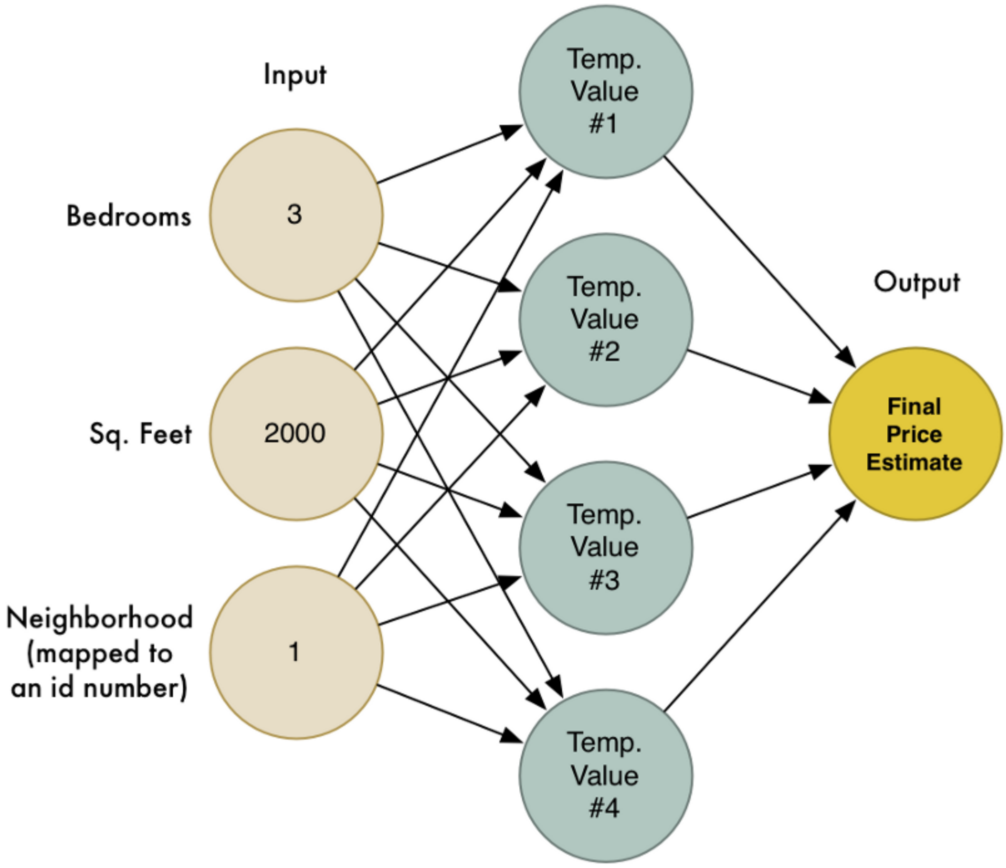
The need for activation functions

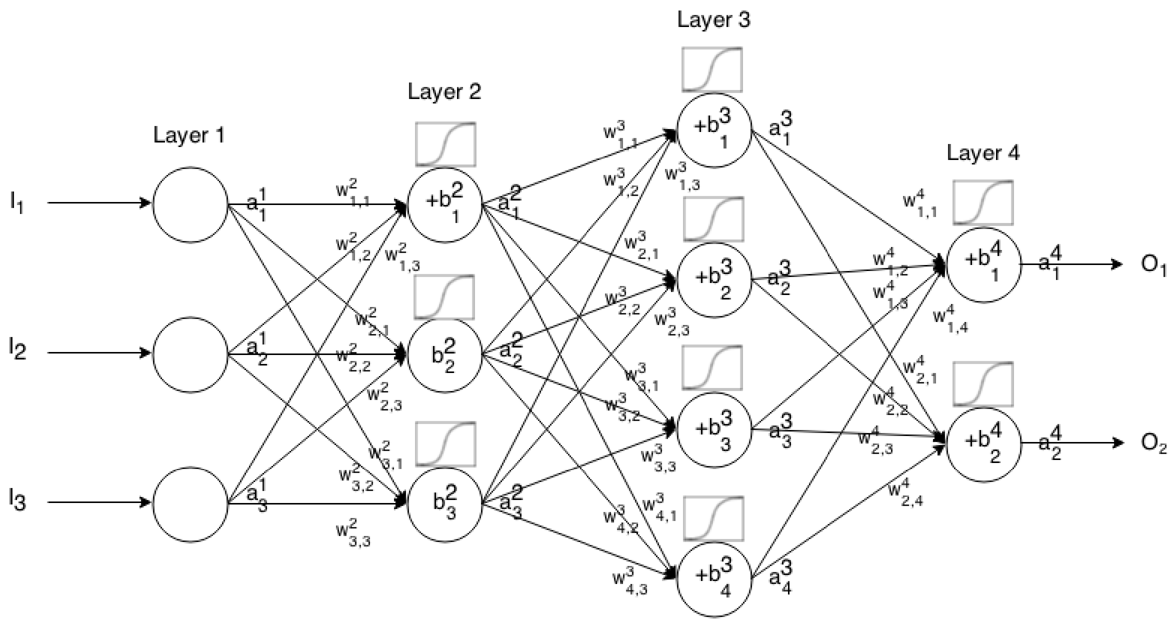
- We use an activation for introducing a non linearity in a network.
- The output of a neuron i.e activation(wx + b) where activation is the function (eg: sigmoid, ReLU, tanh etc)
- When the data is not linearly separable, we have to use a activation function (for example in an XOR problem).
- We also use activation functions to limit the output of a neuron (say between 0 and 1 => sigmoid).
- When the activation function is non-linear, then a two-layer neural network can be proven to be a universal function approximator.
- The identity activation function does not satisfy this property. When multiple layers use the identity activation function, the entire network is equivalent to a single-layer model.
- If there were no activation functions then the output will always be linear.

Some activation functions.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \ge 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \ne 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \ge 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \ge 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \ge 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \ge 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \ge 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \ge 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

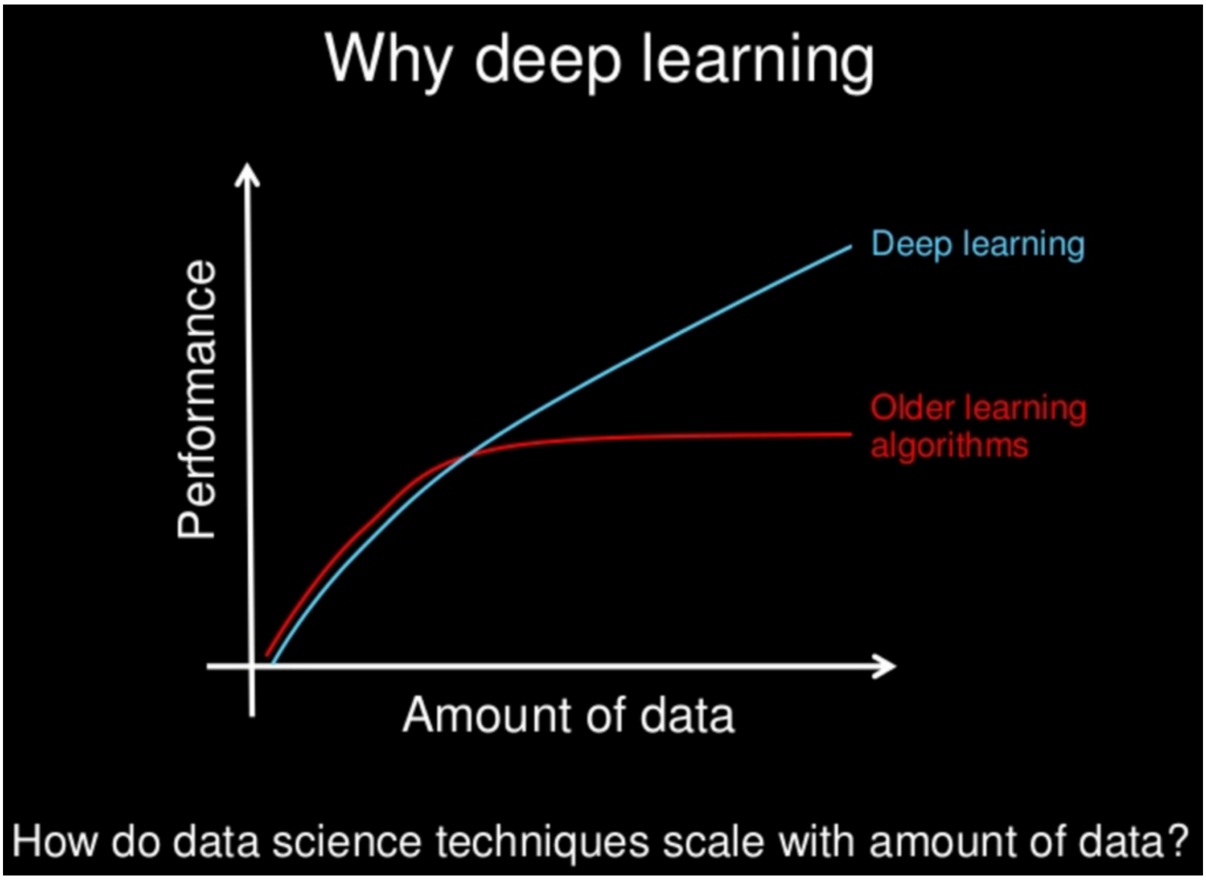
Some illustrations

- 

- Sample problem neural network (the first nodes are the inputs)
- 

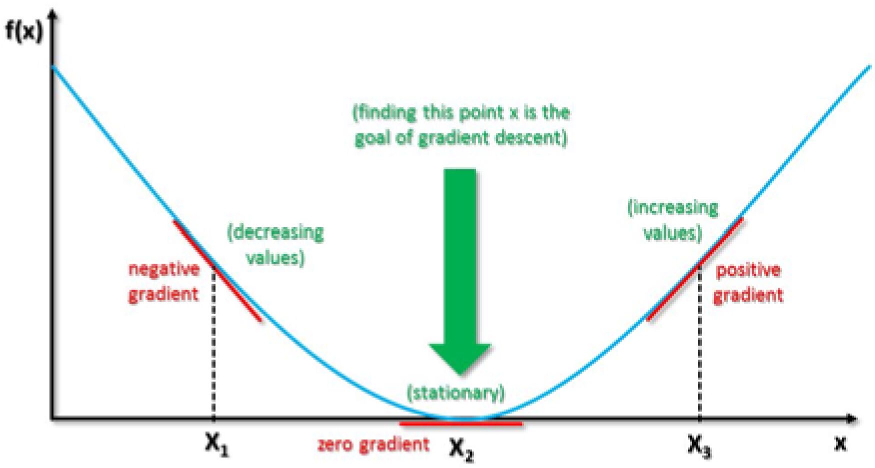
- Inner working of a neural network

Deep Learning is just a neural network with lots of data and lots of hidden layers.



- Older algorithms include (svm, decision tree etc)
- Why is Deep Learning booming

Gradient Descent



- We have the reach the green point.

Linear Regression using Gradient Descent

$$f(x_i) = f_{W,b}(x_i) = b + \sum_{j=1}^{p} W_j x_{ij} \tag{1}$$

$$L(W,b) = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2 \tag{2}$$

$$\frac{\partial L}{\partial W} = \frac{2}{n}\sum_{i=1}^{n}(f(x_i) - y_i)x_i \qquad \frac{\partial L}{\partial b} = \frac{2}{n}\sum_{i=1}^{n}(f(x_i) - y_i) \tag{3}$$

$$\begin{aligned} W &\leftarrow W - \alpha\frac{\partial L}{\partial W} \\ b &\leftarrow b - \alpha\frac{\partial L}{\partial b} \end{aligned} \tag{4}$$

- (1) is the weighted sum of our neural network. (We have the minimize W and x is our inputs)
- (2) is loss function (mse) (prediciton (f(x) - output))^2)
- (3) is the derivative of the loss function wrt to weight (w) and bias (b)
- (4) is updating the weights and biases using delta rule.
- **Our goal is to find those of values of w and b for which the loss (l) is minimum.**

Backprop is also the same except we have a lot parameters for which we need the find the gradients (dL/dW). We do this using the backpropagation algorithm (chain rule) and then we propagate the error signals backward. (updating the weight layer by layer). The process is iterative.

Backprop: A short Introduction

- **At its essence backpropagation is just a clever application of the chain rule.**

$$\frac{df}{dh} = \frac{df}{dg}\frac{dg}{dh}$$

- **States that if you have 3 functions f, g and h with f being a function of g and g being a function of h then the derivative of f with respect to h is equal to the product of the derivative of f with respect to g and the derivative of g with respect to h.**

$$(x^i, y^i)$$

Let us say we have (x,y) training set (m).

$$x^i = \begin{bmatrix} x_1^i \\ x_2^i \\ x_3^i \end{bmatrix}, y^i = \begin{bmatrix} y_1^i \\ y_2^i \\ y_3^i \end{bmatrix}, g^i = \begin{bmatrix} g_1^i \\ g_2^i \\ g_3^i \end{bmatrix}$$

We have a 3 dim vector of inputs (x), outputs (y) and outputs (g)

$$E = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{3}(g_j^i - y_j^i)^2$$

```
This is our loss function where g is the prediction and y is desired o
utput.
```

Objective

- Now the main objective of neural network training is minimize the cost function (loss) by changing the each weight.To do this we use gradient descent, but for applying gradient descent we should have the gradient as the weight updation will happen using (w = w - learning_rate * gradient).
- So to calculate gradient we we backprop, our objective is to calculate the derivative of the error function with respect to the model parameters.

$$\frac{\partial E}{\partial w_{k,j}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{k,j}}$$
$$= \frac{\partial E}{\partial z_j}\frac{\partial \sum_{k=1}^{n} g'_k w_{k,j}}{\partial w_{k,j}}$$
$$= \frac{\partial E}{\partial z_j} g'_k$$
$$= (g_j - y_j) g_j (1 - g_j) g'_k$$

- Using the chain rule we come to this, the derivative of the error with respect to each weight. (z is a function of the output and the weight).
- z = w * d(g) as g is activated using the sigmoid function.
- **g dash is the derivative of the sigmoid function.**

Finally...

- We have the gradient of the loss function with respect to the weights and now we can apply gradient descent to reach the space where the error is minimum.

Note: This is very short introduction to backprop but it encapsulates all the important points.

Images taken: https://towardsdatascience.com/learning-backpropagation-from-geoffrey-hinton-619027613f0 (https://towardsdatascience.com/learning-backpropagation-from-geoffrey-hinton-619027613f0)

More resources to learn about the subject.

- Resources to learn more about the subject.
- http://neuralnetworksanddeeplearning.com/ (http://neuralnetworksanddeeplearning.com/)
- http://www.deeplearningbook.org/ (http://www.deeplearningbook.org/)
- By reading the above papers.
- https://dl.acm.org/citation.cfm?id=668382 (https://dl.acm.org/citation.cfm?id=668382)
- At last by implementing them.
- https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.92157&showTestData (https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.92157&showTestData

Now let us look at how to implement neural networks in code.

- We will be using a combination of Pytorch and Keras.
- Deep Learning frameworks provide us with automatic differentiation.
- Framework provide us a way to encapsulate the backend code.

I will not go in detail with Pytorch and Keras but the main difference between pytorch and keras is that pytorch is defined by dynamic run, you create the computation graph and can run it on the go. With Tensorflow (Keras) the graph is static i.e you first define the graph and then in a session you can run the graph.

In [2]:

```
# Pytorch
import torch
from torch import nn
from torch.autograd import Variable # For automatic gradient calculation
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader

# Keras
import keras
from keras.layers import Dense
from keras import Sequential
from keras import losses
from keras.utils import to_categorical
import keras.backend as K

# Others
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split # For splitting data into
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

Let us first experiment with some activation functions.

In [9]:

```
print ("PyTorch (Dynamic Graph)")
print ("========================================")
x = torch.Tensor([42]) # A tensor
print ("Sigmoid: ", F.sigmoid(x)) # between 0 and 1 (don't use this)
print ("ReLU: ", F.relu(x)) # max(0,a) (linear) (use this for hidden layers)
print ("TanH: ", F.tanh(x)) # between -1 and 1 (can use this)
print ("Elu", F.elu(x)) # Linear (experimentation)
print ("Softmax: ", F.softmax(x, dim=0)) # Genrally used to convert output into
print ()
print ("Keras (Static Graph)")
print ("========================================")
x = np.array([1.0])
print ("Sigmoid: ", K.sigmoid(x))
print ("ReLU: ", K.relu(x)) # max(0,a) (linear) (use this for hidden layers)
print ("TanH: ", K.tanh(x)) # between -1 and 1 (can use this)
print ("Elu", K.elu(x)) # Linear (experimentation)
print ("Softmax: ", K.softmax(x)) # Genrally used to convert output into probab
```

```
PyTorch (Dynamic Graph)
========================================
Sigmoid:  tensor([ 1.])
ReLU:  tensor([ 42.])
TanH:  tensor([ 1.])
Elu tensor([ 42.])
Softmax:  tensor([ 1.])

Keras (Static Graph)
========================================
Sigmoid:  Tensor("Sigmoid_7:0", shape=(1,), dtype=float64)
ReLU:  Tensor("Relu_5:0", shape=(1,), dtype=float64)
TanH:  Tensor("Tanh_5:0", shape=(1,), dtype=float64)
Elu Tensor("Elu_5:0", shape=(1,), dtype=float64)
Softmax:  Tensor("Reshape_11:0", shape=(1,), dtype=float64)
```

- A tensor is basically a vector.
- Scalar (1d tensor)
- List of List (Vector)
- n dimension vector (Tensor)

In [9]:

```
# We can also do this with numpy.
print ("1d: ", torch.Tensor([1]).shape)
print ("2d: ", torch.Tensor([[1,1],[1,1]]).shape)
print ("3d: ", torch.Tensor([[[1,1],[1,1],[1,1]]]).shape)


# A tensor of 4 x 4
print (torch.rand(4,4))
print (torch.rand(10,5))
```

```
1d:  torch.Size([1])
2d:  torch.Size([2, 2])
3d:  torch.Size([1, 3, 2])
tensor([[ 0.3822,  0.9990,  0.7803,  0.2338],
        [ 0.8558,  0.8310,  0.9127,  0.2803],
        [ 0.3497,  0.5993,  0.0770,  0.2481],
        [ 0.5423,  0.4820,  0.7832,  0.5505]])
tensor([[ 0.9939,  0.0113,  0.3742,  0.4728,  0.9508],
        [ 0.1396,  0.1917,  0.9351,  0.6882,  0.9521],
        [ 0.1258,  0.9595,  0.9697,  0.5089,  0.7573],
        [ 0.6951,  0.0931,  0.7793,  0.4472,  0.4248],
        [ 0.2510,  0.9690,  0.1668,  0.3804,  0.8359],
        [ 0.1346,  0.6283,  0.9535,  0.5691,  0.7349],
        [ 0.4912,  0.0603,  0.0962,  0.3202,  0.1519],
        [ 0.7144,  0.0457,  0.7851,  0.5493,  0.3653],
        [ 0.1516,  0.5434,  0.3423,  0.7927,  0.1125],
        [ 0.2801,  0.3594,  0.3292,  0.5315,  0.8926]])
```

Now let us calculate some gradients (differentiation)

In [27]:

```
# Defining a variable x
x = Variable(torch.Tensor([1]), requires_grad=True)
# Variable is a wrapper around Tensor, requires grad provides that the system h
```

In [29]:

```
# Building a function
f_x = 3 * x * x # f(x) = 3x^2
print ("f_x: ", f_x) # Putting x = 1 in f(x)
```

```
f_x:  tensor([ 3.])
```

In [30]:

```
# Calculating gradient
f_x.backward() # wrt to x
# Hence d(f(x)) = 6x
# Putting x = 1, we get 6
```

In [32]:

```
# Checking gradient value
print ("Gradient value: ", x.grad.item())
```

```
Gradient value:  6.0
```

Similarly we can calculate derivatives for other functions.

Creation of a simple neural network.

In [10]:

```
input = 5
hidden = 10
output = 2

# PyTorch
class Net(nn.Module):
    def __init__(self, input, hidden, output):
        super(Net, self).__init__()
        self.l1 = nn.Linear(input, hidden)
        self.l2 = nn.Linear(hidden, output)
        # A single hidden layer network
    def forward(self, x):
        # Getting output from first layer and passing it to the next layer.
        out = F.relu(self.l1(x)) # Pre Activation
        out = self.l2(out)  # We can also use F.softmax(out, dim=1) depending o
        return out

# Keras
model = Sequential()
model.add(Dense(hidden, input_dim=input, activation="relu"))
model.add(Dense(output, activation="softmax"))
```

In [11]:

```
# Printing out summary
net = Net(input, hidden, output)
print (net)
print()
print (model.summary())
```

```
Net(
  (l1): Linear(in_features=5, out_features=10, bias=True)
  (l2): Linear(in_features=10, out_features=2, bias=True)
)

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 10)                60
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 22
=================================================================
Total params: 82
Trainable params: 82
Non-trainable params: 0
_________________________________________________________________
None
```

In [12]:

```
# Printing weights (randomly assigned)
for param in net.named_parameters():
    print (param)
```

```
('l1.weight', Parameter containing:
tensor([[ 0.2472, -0.1372, -0.1391,  0.2868,  0.4423],
        [ 0.1479,  0.2254,  0.1169,  0.0628,  0.2105],
        [ 0.4077,  0.0404, -0.0364,  0.1731, -0.1262],
        [ 0.0456,  0.1840,  0.0185,  0.1073,  0.1492],
        [ 0.3188,  0.1847, -0.4392,  0.1537,  0.2711],
        [-0.2938,  0.0347,  0.3738,  0.2667,  0.0344],
        [ 0.1583, -0.3155, -0.4176, -0.2127, -0.1567],
        [ 0.4094, -0.2945,  0.1113, -0.4440, -0.0841],
        [ 0.4044,  0.2970,  0.0721,  0.1813,  0.4140],
        [-0.4127, -0.2336, -0.0218, -0.2896, -0.1163]]))
('l1.bias', Parameter containing:
tensor([ 0.3519,  0.0010, -0.4226,  0.2627,  0.2823, -0.4133, -0.3418,
         0.0641,  0.4234, -0.3400]))
('l2.weight', Parameter containing:
tensor([[ 0.2574, -0.1282,  0.2090,  0.2112,  0.3162, -0.1087, -0.015
5,
          0.1115, -0.1288, -0.1724],
        [-0.0457, -0.0895, -0.0754, -0.1052,  0.2823,  0.0091, -0.120
4,
         -0.1686, -0.1477,  0.2817]]))
('l2.bias', Parameter containing:
tensor([ 0.2481, -0.2912]))
```

In [13]:

```
model.get_weights() # Getting weights (randomly assigned)
```

Out[13]:

```
[array([[-0.5276527 , -0.26292667, -0.15418386,  0.39451522, -0.061641
57,
          0.3849358 , -0.32673505, -0.3841958 , -0.17214736, -0.104696
69],
        [-0.24589986,  0.08675081, -0.41050506,  0.1896193 ,  0.078901
95,
         -0.3438062 ,  0.60974914, -0.47975755,  0.29143715, -0.227262
47],
        [ 0.06937325,  0.56571954,  0.08736163, -0.629089  , -0.063679
46,
          0.12157995,  0.20947611,  0.2761436 , -0.1319462 , -0.261136
35],
        [ 0.18716145, -0.61267304,  0.09599721, -0.10066444, -0.281196
74,
         -0.19047973, -0.2537911 , -0.1822795 ,  0.06281561, -0.524250
27],
        [ 0.37544662, -0.13515383,  0.08927745,  0.12276739,  0.547505
44,
          0.29646957,  0.48087114,  0.11858374, -0.14628658, -0.479375
6 ]],
       dtype=float32),
 array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32),
 array([[ 0.4600312 ,  0.6465495 ],
        [-0.11415017, -0.57420105],
        [-0.42226726, -0.18428642],
        [ 0.36069387, -0.58035505],
        [ 0.09534305,  0.3909176 ],
        [ 0.27903455,  0.13104397],
        [-0.5756513 , -0.5331563 ],
        [-0.4728182 , -0.48940668],
        [ 0.66696566,  0.40292054],
        [ 0.05157596,  0.389517  ]], dtype=float32),
 array([0., 0.], dtype=float32)]
```

Let us solve a simple classification problem with nn's.

In [14]:

```
# Creating data
data = make_classification(n_samples=50000, n_features=10) # A sample with 5000
```

In [15]:

```
# features are the inputs and labels are the outputs (x,y) in a supervised lear
features, labels = data[0],data[1]
print (data[0])
print (data[1])
# Splitting data into training and testing dataset.
features_train, features_test, labels_train, labels_test = train_test_split(fea
```

```
[[ 0.38906299 -0.8470888   1.20317219 ... -2.77786198 -1.06957387
  -0.82307484]
 [-1.13271081 -0.93484106  0.53226557 ...  1.57993383  0.41133052
  -0.18182803]
 [ 0.89384069  1.54612058 -1.70489036 ...  0.96476255  1.36755476
  -0.91596387]
 ...
 [ 0.35133558  0.3460963  -0.41445323 ...  0.24084999 -0.51595742
  -0.5762611 ]
 [-0.25975114 -1.37364499  1.91941122 ... -0.07888223 -0.62843198
   1.85545095]
 [ 0.57393     0.30496682 -0.0028945  ... -1.52162151  0.48342614
   0.30672199]]
[0 1 1 ... 1 1 0]
```

In [16]:

```
# Checking size (row,columns)
features_train.shape
```

Out[16]:

```
(37500, 10)
```

In [17]:

```
features_test.shape
```

Out[17]:

```
(12500, 10)
```

- We have 37500 records in the train set and 12500 records in the test set.

In [18]:

```
input = features_train.shape[1] # The number of columns (10)
hidden = 100
output = 2 # 2 (the output)


# Let us build the model. (Pytorch)
class Model(nn.Module):

    # hidden is a randomly chosen number which specifies the number of hidden n
    def __init__(self, input, hidden, output):
        super(Model, self).__init__()
        self.l1 = nn.Linear(input, hidden)
        self.l2 = nn.Linear(hidden, hidden)
        self.l3 = nn.Linear(hidden, hidden)
        self.l4 = nn.Linear(hidden, output)

    # Forward pass
    def forward(self, x):
        # As a rule of thumb we use relu on hidden layers. (On a more deeper no
        out = F.relu(self.l1(x))
        out = F.relu(self.l2(out))
        out = F.relu(self.l3(out))
        out = self.l4(out)
        return out

model = Model(input, hidden, output)
```

Our model is built, let us start training...

In [19]:

```
# Defining hyperparameters
# Stochastic gradient descent
# Adam is a variant of gradient descent.
# We can change the lr
optimizer = torch.optim.Adam(model.parameters(), lr=0.001) # lr is the learning
criterion = nn.CrossEntropyLoss() # For classification
epochs = 5 # can change
```

In [20]:

```
# Building dataset
# Converting into tensors
train_dataset = TensorDataset(torch.from_numpy(features_train), torch.from_numpy
test_dataset = TensorDataset(torch.from_numpy(features_test), torch.from_numpy(
```

In [21]:

```
train_loader = DataLoader(train_dataset, shuffle=True, batch_size=64)
test_loader = DataLoader(test_dataset, shuffle=True, batch_size=64)
```

We are splitting our data into batches so that the weight updation occurs after every training example (Stochastic Gradient Descent <=> Mini Batch Gradient Descent)

This is the recommended way (mini batch gradient descent)

Here we are taking batches of data (64) and then updating our weights

In [37]:

```
def train(epochs):
    model.train()
    for epoch in range(epochs):
        print ("epoch #", epoch)
        current_loss = 0
        for (feature, label) in train_loader:
            feature = Variable(feature).float() # requires_grad is False
            label = Variable(label).long() # requires_grad is False
            prediction = model(feature) # forward loss
            loss = criterion(prediction, label) # calculating loss
            current_loss+=loss.item() # Adding loss
            optimizer.zero_grad() # Zeroing gradients (manual way in pytorch)
            loss.backward() # Calculate the gradient
            optimizer.step() # Update the weight
        print ("loss after epoch#:",epoch, ": ", current_loss)
```

In [40]:

```
def test(epochs):
    model.eval()
    # No gradients
    with torch.no_grad():
          for epoch in range(epochs):
            print ("epoch #", epoch)
            current_loss = 0
            for (feature, label) in test_loader:
                feature = Variable(feature).float() # requires_grad is False
                label = Variable(label).long() # requires_grad is False

                prediction = model(feature) # forward pass
                loss = criterion(prediction, label) # Calculating loss
                current_loss+=loss.item() # Loss after iterative over the whole
            print ("loss after epoch#:",str(epoch) + ": ", str(current_loss))

train(epochs)
test(epochs)
# The prediction portion is same.
```

```
epoch # 0
loss after epoch#: 0 :  134.94877634570003
epoch # 1
loss after epoch#: 1 :  134.27247551083565
epoch # 2
loss after epoch#: 2 :  133.40212597697973
epoch # 3
loss after epoch#: 3 :  133.03710904717445
epoch # 4
loss after epoch#: 4 :  132.65547116845846
epoch # 0
loss after epoch#: 0:  47.83670485764742
epoch # 1
loss after epoch#: 1:  47.76737977564335
epoch # 2
loss after epoch#: 2:  47.869622960686684
epoch # 3
loss after epoch#: 3:  47.710053242743015
epoch # 4
loss after epoch#: 4:  47.85326048359275
```

- The loss will decrease on increasing the number of epochs.

For simplicity i will be using the whole data and then updating the weights. (Gradient Descent)

The model structure will be the same. I will only change the optimizer and train, test function.

In [35]:

```
# Converting numpy arrays into Variable (they remember who created them)

# Training dataset
x_train = Variable(torch.from_numpy(features_train)).float()
y_train = Variable(torch.from_numpy(labels_train)).long()


# Testing dataset
x_test = Variable(torch.from_numpy(features_test)).float()
y_test = Variable(torch.from_numpy(labels_test)).long()
```

In [42]:

```
losses_train = []
def train2(epochs):
    global losses_train
    model.train()
    for epoch in range(epochs):
        optimizer.zero_grad()
        pred = model(x_train)
        loss = criterion(pred,y_train)
        print ("epoch #", epoch)
        print ("loss: ", loss.item())
        losses_train.append(loss.item())
        loss.backward()
        optimizer.step()
        # Same as above
```

In [43]:

```
losses_test = []
def test2(epochs):
    global losses_test

    model.eval()
    with torch.no_grad():
        for epoch in range(epochs):
            pred = model(x_train)
            loss = criterion(pred,y_train)
            losses_test.append(loss.item())
            print ("epoch #", epoch)
            print ("loss: ", loss.item())

```

In [44]:

```
train2(epochs) # todo
```

```
epoch # 0
loss:  0.2245241105556488
epoch # 1
loss:  0.22425973415374756
epoch # 2
loss:  0.22396187484264374
epoch # 3
loss:  0.22364023327827454
epoch # 4
loss:  0.22330573201179504
```

In [45]:

```
losses_train
```

Out[45]:

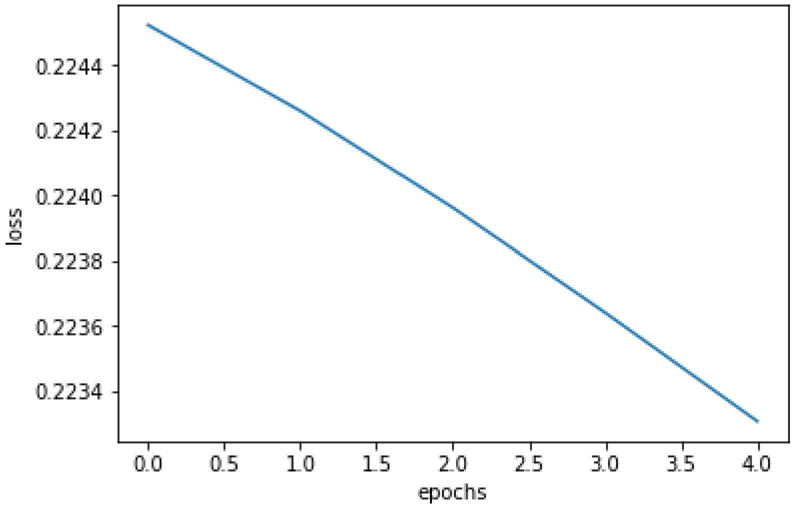
```
[0.2245241105556488,
 0.2242597341537475б,
 0.22396187484264374,
 0.22364023327827454,
 0.22330573201179504]
```

In [46]:

```
# plotting the training loss
plt.plot(losses_train)
plt.xlabel("epochs")
plt.ylabel("loss")
```

Out[46]:

```
Text(0,0.5,'loss')
```



- We can see that the loss is decreasing over epochs.

Now let us predict!

In [116]:

```
# On unseen data
predictions = model(x_test)
```

In [117]:

```
# Finding the highest probability index (log_softmax)
_,predictions = torch.max(predictions, 1)
```

In [119]:

```
print ("the accuracy of the model: ", accuracy_score(labels_test, predictions.da
```

```
the accuracy of the model:  0.88552
```

- Not bad at all! We can tune the hyperparameters (lr, optimzier etc) to get a better accuracy

Now we will build the same model in Keras.

In [22]:

```
net = Sequential()
net.add(Dense(100, input_dim=10, activation="relu"))
net.add(Dense(100, activation="relu"))
net.add(Dense(100, activation="relu"))
net.add(Dense(2, activation="softmax"))

# loss is cross entropy
net.compile(optimizer="adam", loss=keras.losses.categorical_crossentropy, metric
```

In [23]:

```
# Changing training data into one hot encoding form
labels_train_keras = to_categorical(labels_train)
labels_test_keras = to_categorical(labels_test)
```

In [24]:

```
# Fitting the model (training)
history = net.fit(features_train, labels_train_keras, epochs=5) # todo
```

```
Epoch 1/5
37500/37500 [==============================] - 3s 74us/step - loss: 0.
3364 - acc: 0.8574
Epoch 2/5
37500/37500 [==============================] - 3s 70us/step - loss: 0.
3185 - acc: 0.8666
Epoch 3/5
37500/37500 [==============================] - 3s 67us/step - loss: 0.
3154 - acc: 0.8672
Epoch 4/5
37500/37500 [==============================] - 2s 67us/step - loss: 0.
3130 - acc: 0.8684
Epoch 5/5
37500/37500 [==============================] - 3s 67us/step - loss: 0.
3116 - acc: 0.8688
```

Now let us predict

In [25]:

```
predictions = net.predict(features_test)
predictions = np.argmax(predictions, axis=1)
print ("accuracy is: ", accuracy_score(labels_test, predictions))
```

```
accuracy is:  0.86496
```

- We can get a different answer because Keras provides us with everything, it operates like a black box. Everything is configured already.

Now let us convert the model into a core ML model

To install coremltools use pip: pip install coremltools (pip is a python package manager like cocoapods)

For Keras

- To convert a model we use coremltools provided by Apple. (for keras)
- For pytorch Apple has not provided an official tool but we can use onnx built by microsoft and facebook to convert our models into coreml (Pytorch).

In [6]:

```
# For keras
from coremltools.converters.keras import convert
```

```
WARNING:root:Keras version 2.1.6 detected. Last version known to be fu
lly compatible of Keras is 2.1.3 .
WARNING:root:TensorFlow version 1.6.0 detected. Last version known to
be fully compatible is 1.5.0 .
```

In [28]:

```
input_names = []
for i in range(1, 11):
    input_names.append("features" + str(i))
input_names
```

Out[28]:

```
['features1',
 'features2',
 'features3',
 'features4',
 'features5',
 'features6',
 'features7',
 'features8',
 'features9',
 'features10']
```

In [ ]:

```
# *****
net_saved = convert(net,input_names=input_names, output_names='label')
```

In [ ]:

```
net_saved.author = "Aadit Kapoor"
net_saved.short_description = "demo"
net_saved.license = "MIT"
net_saved.save("demo.mlmodel")
# Model is saved
```

For Pytorch

In [29]:

```
from onnx_coreml import convert
import onnx
```

In [40]:

```
dummy = Variable(torch.FloatTensor(37500, 10))
torch.onnx.export(model, dummy, 'demomodel.proto', verbose=True)
model = onnx.load('demomodel.proto')
coreml_model = convert(
    model,
    'classifier',
    image_input_names=['features'],
    image_output_names=['labels'],
    class_labels=[0,1],
)
coreml_model.save("demomodel.mlmodel")
# Model will be saved (*****)
```

Out[40]:

```
Model(
  (l1): Linear(in_features=10, out_features=100, bias=True)
  (l2): Linear(in_features=100, out_features=100, bias=True)
  (l3): Linear(in_features=100, out_features=100, bias=True)
  (l4): Linear(in_features=100, out_features=2, bias=True)
)
```

Some more examples

1. Nutrition Model

- http://localhost:8888/notebooks/Documents/swift-delhi-talk/Calorie%20Predictor/ml-nutrition-database.ipynb (http://localhost:8888/notebooks/Documents/swift-delhi-talk/Calorie%20Predictor/ml-nutrition-database.ipynb)

2. Tic Tac Toe Model

- http://localhost:8888/notebooks/Documents/swift-delhi-talk/tic-tac-toe-ml-project/ml-model/tic-tac-toe.ipynb (http://localhost:8888/notebooks/Documents/swift-delhi-talk/tic-tac-toe-ml-project/ml-model/tic-tac-toe.ipynb)

3. IPL Match predictor

- http://localhost:8888/notebooks/Documents/swift-delhi-talk/ipl-match/winner-predictor.ipynb (http://localhost:8888/notebooks/Documents/swift-delhi-talk/ipl-match/winner-predictor.ipynb)
- http://ipl-predictor.herokuapp.com/home/ (http://ipl-predictor.herokuapp.com/home/)

4. Handwriting Model (mnist)

- http://localhost:8888/edit/Documents/swift-delhi-talk/mnist-model/mnist-without-cnn.py (http://localhost:8888/edit/Documents/swift-delhi-talk/mnist-model/mnist-without-cnn.py)

5. Tensorflow for Swift

- https://www.tensorflow.org/community/swift (https://www.tensorflow.org/community/swift)

6. Machine Learning at Apple

- http://machinelearning.apple.com/ (http://machinelearning.apple.com/)

7. Turicreate (Turi Create simplifies the development of custom machine learning models

- https://github.com/apple/turicreate (https://github.com/apple/turicreate)

Thank you!

If you need any help contact me at: aaditkapoor2000@gmail.com (mailto:aaditkapoor2000@gmail.com).

I am 18 years old and going to start college in August (Just completed my 12th grade this March, so I am free for collaborations :)

In [ ]:

```
1
```