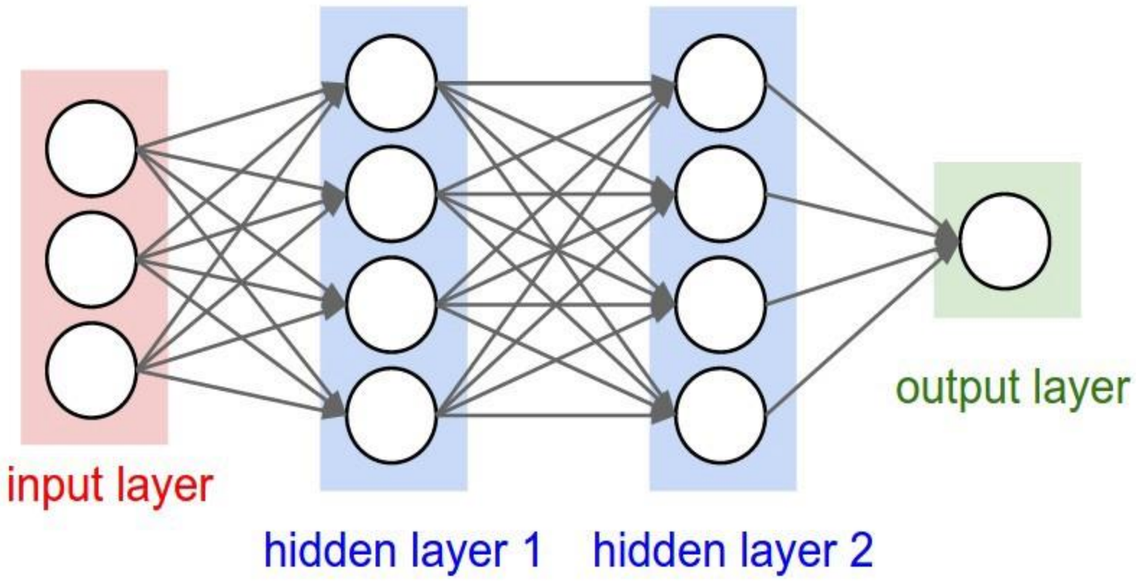
NEURAL NETWORKS

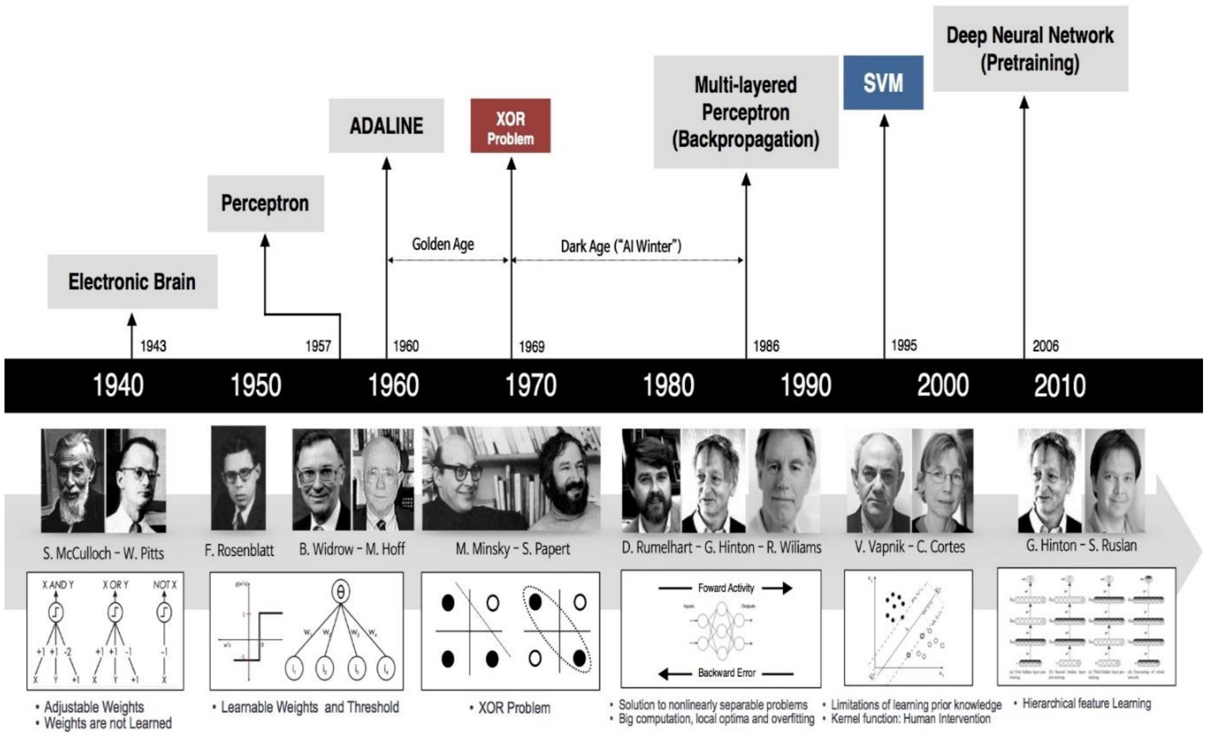
1. What is an artificial neural network?

Artificial neural networks are one of the main tools used in machine learning. As the “neural” part of their name suggests, they are brain-inspired systems which are intended to replicate the way that we humans learn. The neural network itself isn't an algorithm, but rather a framework for many different machine learning algorithms to work together and process complex data inputs. Neural networks consist of input and output layers, as well as (in most cases) a hidden layer consisting of units that transform the input into something that the output layer can use. They are excellent tools for finding patterns which are far too complex or numerous for a human programmer to extract and teach the machine to recognize.



2. Historical background

Neural network simulations appear to be a recent development. However, this field was established before the advent of computers, and has survived at least one major setback and several eras.
Many important advances have been boosted using inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few researchers. These pioneers were able to develop convincing technology which surpassed the limitations identified by Minsky and Papert. Minsky and Papert, published a book (in 1969) in which they summed up a general feeling of frustration (against neural networks) among researchers, and was thus accepted by most without further analysis. Currently, the neural network field enjoys a resurgence of interest and a corresponding increase in funding.
The first artificial neuron was produced in 1943 by the neurophysiologist Warren McCulloch and the logician Walter Pits. But the technology available at that time did not allow them to do too much.


Electronic Brain
Perceptron
ADALINE
XOR Problem
Multi-layered Perceptron (Backpropagation)
SVM
Deep Neural Network (Pretraining)
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch - W. Pitts
F. Rosenblatt
B. Widrow - M. Hoff
M. Minsky - S. Papert
D. Rumelhart - G. Hinton - R. Wiliams
V. Vapnik - C. Cortes
G. Hinton - S. Ruslan
X AND Y
X OR Y
NOT X
Foward Activity
Backward Error
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

3. Neural networks versus conventional computers

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach i.e. the computer follows a set of instructions to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem-solving capability of conventional computers to problems that we already understand and know how to solve. But computers would be so much more useful if they could do things that we don't exactly know how to do.

Neural networks process information in a similar way the human brain does. The network is composed of many highly interconnected processing elements(neurones) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific task. The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is that because the network finds out how to solve the problem by itself, its operation can be unpredictable.

On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is to solved must be known and stated in small unambiguous instructions. These instructions are then converted to a high-level language program and then into machine code that the computer can understand. These machines are totally predictable; if anything goes wrong is due to a software or hardware fault.

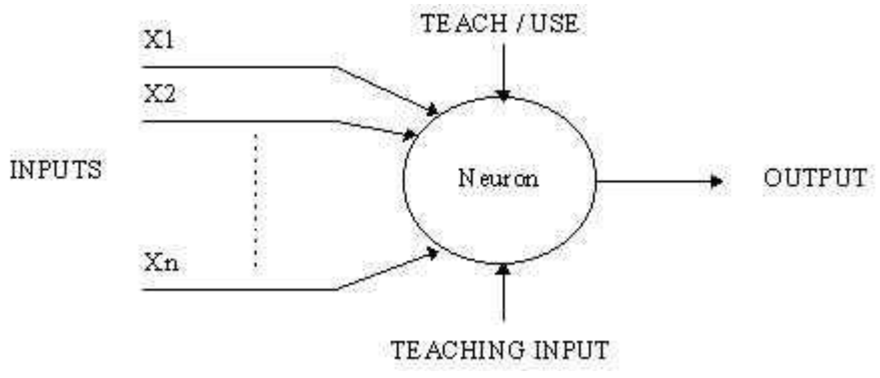
4. How many types of neural network are there?

There are multiple types of neural network, each of which come with their own specific use cases and levels of complexity. The most basic type of neural net is something called a feedforward neural network, in which information travels in only one direction from input to output. A more widely used type of network is the recurrent neural network, in which data can flow in multiple directions. These neural networks possess greater learning abilities and are widely employed for more complex tasks such as learning handwriting or language recognition. There are also convolutional neural networks, Boltzmann machine networks, Hopfield networks, and a variety of others. Picking the right network for your task depends on the data you must train it with, and the specific application you have in mind. In some cases, it may be desirable to use multiple approaches, such as would be the case with a challenging task like voice recognition.

5. An engineering approaches

5.1 A simple neuron

An artificial neuron is a device with many inputs and one output. The neuron has two modes of operation; the training mode and the using mode. In the training mode, the neuron can be trained to fire (or not), for input patterns. In the using mode, when a taught input pattern is detected at the input, its associated output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire or not.



A simple neuron

5.2 Firing rules

The firing rule is an important concept in neural networks and accounts for their high flexibility. A firing rule determines how one calculates whether a neuron should fire for any input pattern. It relates to all the input patterns, not only the ones on which the node was trained.

A simple firing rule can be implemented by using Hamming distance technique. The rule goes as follows:

Take a collection of training patterns for a node, some of which cause it to fire (the 1-taught set of patterns) and others which prevent it from doing so (the 0-taught set). Then the patterns not in the collection cause the node to fire if, on comparison, they have more input elements in common with the 'nearest' pattern in the 1-taught set than with the 'nearest' pattern in the 0-taught set. If there is a tie, then the pattern remains in the undefined state.

For example, a 3-input neuron is taught to output 1 when the input (X1, X2 and X3) is 111 or 101 and to output 0 when the input is 000 or 001. Then, before applying the firing rule, the truth table is;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0/1	0/1	0/1	1	0/1	1

As an example of the way the firing rule is applied, take the pattern 010. It differs from 000 in 1 element, from 001 in 2 elements, from 101 in 3 elements and from 111 in 2 elements. Therefore, the 'nearest' pattern is 000 which belongs in the 0-taught set. Thus, the firing rule requires that the neuron should not fire when the input is 001. On the other hand, 011 is equally distant from two taught patterns that have different outputs and thus the output stays undefined (0/1).

By applying the firing in every column, the following truth table is obtained;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0	0/1	0/1	1	1	1

The difference between the two truth tables is called the *generalisation of the neuron*. Therefore, the firing rule gives the neuron a sense of similarity and enables it to respond 'sensibly' to patterns not seen during training.

5.3 Pattern Recognition - an example

An important application of neural networks is pattern recognition. Pattern recognition can be implemented by using a feed-forward (figure 1) neural network that has been trained accordingly. During training, the network is trained to associate outputs with input patterns. When the network is used, it identifies the input pattern and tries to output the associated output pattern. The power of neural networks comes to life when a pattern that has no output associated with it, is given as an input. In this case, the network gives the output that corresponds to a taught input pattern that is least different from the given pattern.

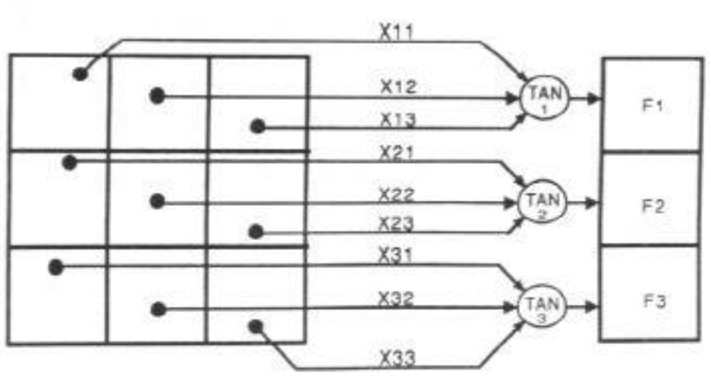
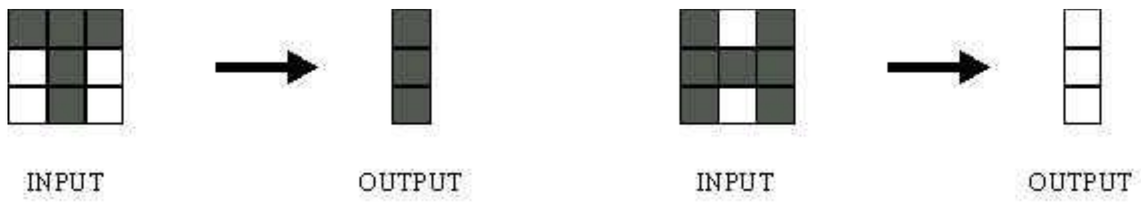


Figure 1.

For example:
The network of figure 1 is trained to recognise the patterns T and H. The associated patterns are all black and all white respectively as shown below.



If we represent black squares with 0 and white squares with 1 then the truth tables for the 3 neurones after generalisation are;

X11:		0	0	0	0	1	1	1	1
X12:		0	0	1	1	0	0	1	1
X13:		0	1	0	1	0	1	0	1
OUT:		0	0	1	1	0	0	1	1

Top neuron

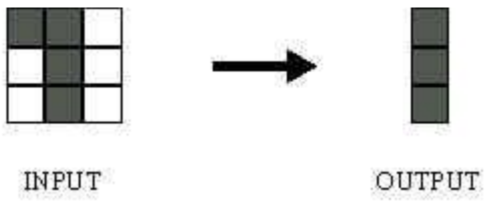
X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0/1	1	0/1	0/1	0	0/1	0

Middle neuron

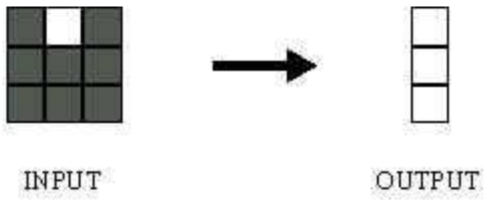
X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0	1	1	0	0	1	0

Bottom neuron

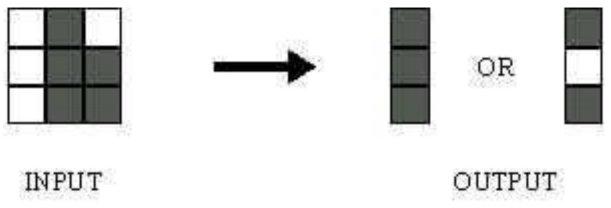
From the tables it can be seen the following associations can be extracted:



In this case, it is obvious that the output should be all blacks since the input pattern is almost the same as the 'T' pattern.



Here also, it is obvious that the output should be all whites since the input pattern is almost the same as the 'H' pattern.



Here, the top row is 2 errors away from a T and 3 from an H. So, the top output is black. The middle row is 1 error away from both T and H, so the output is random. The bottom row is 1 error away from T and 2 away from H. Therefore, the output is black. The total output of the network is still in favour of the T shape.

6. The Learning Process

The memorisation of patterns and the subsequent response of the network can be categorised into two general paradigms:

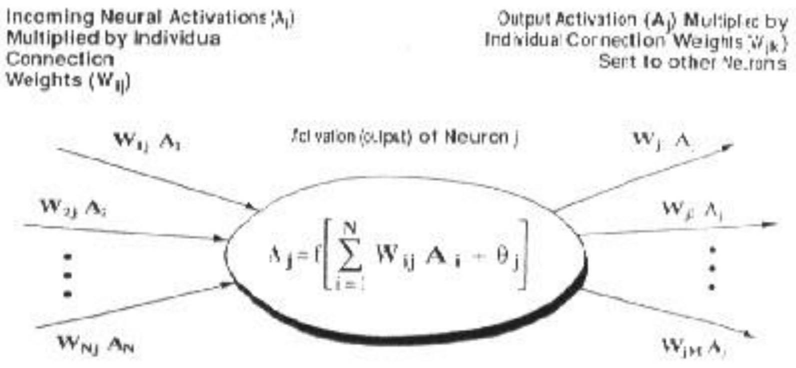
- **associative mapping** in which the network learns to produce a pattern on the set of input units whenever another particular pattern is applied on the set of input units. The associative mapping can generally be broken down into two mechanisms:

  - *auto-association*: an input pattern is associated with itself and the states of input and output units coincide. This is used to provide pattern competition, i.e. to produce a pattern whenever a portion of it or a distorted pattern is presented. In the second case, the network stores pairs of patterns building an association between two sets of patterns.
  - *hetero-association* is related to two recall mechanisms:

    - *nearest-neighbour* recall, where the output pattern produced corresponds to the input pattern stored, which is closest to the pattern presented, and
    - *interpolative* recall, where the output pattern is a similarity dependent interpolation of the patterns stored corresponding to the pattern presented. Yet another paradigm, which is a variant associative mapping is classification, i.e. when there is a fixed set of categories into which the input patterns are to be classified.

- **regularity detection** in which units learn to respond to properties of the input patterns. Whereas in associative mapping the network stores the relationships among patterns, in regularity detection the response of each unit has a particular 'meaning'. This type of learning mechanism is essential for feature discovery and knowledge representation.

Every neural network possesses knowledge which is contained in the values of the connection's weights. Modifying the knowledge stored in the network as a function of experience implies a learning rule for changing the values of the weights.


Incoming Neural Activations (A_i) Multiplied by Individua Connection Weights (W_{ij})
Output Activation (A_j) Multiplied by Individual Connection Weights (W_{jk}) Sent to other Neurons
Activation (output) of Neuron j
$W_{1j}\ A_1$
$W_{2j}\ A_2$
$W_{Nj}\ A_N$
$W_{j1}\ A_j$
$W_{j2}\ A_j$
$W_{jM}\ A_j$
$A_j = f\left[\sum_{i=1}^{N} W_{ij} A_i + \theta_j\right]$

All learning methods used for adaptive neural networks can be classified into two major categories:

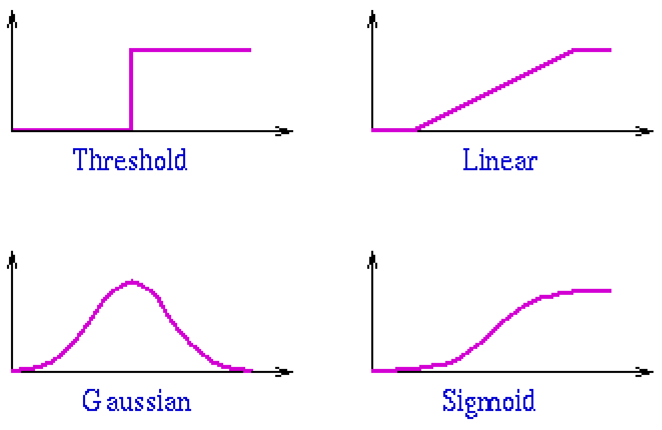
- **Supervised learning** which incorporates an external teacher, so that each output unit is told what its desired response to input signals ought to be. During the learning process global information may be required. Paradigms of supervised learning include error-correction learning, reinforcement learning and stochastic learning.
  An important issue concerning supervised learning is the problem of error convergence, i.e. the minimisation of error between the desired and computed unit values. The aim is to determine a set of weights which minimises the error. One well-known method, which is common to many learning paradigms is the least mean square (LMS) convergence.

- **Unsupervised learning** uses no external teacher and is based upon only local information. It is also referred to as self-organisation, in the sense that it self-organises data presented to the network and detects their emergent collective properties. Paradigms of unsupervised learning are Hebbian learning and competitive learning. Usually, supervised learning is performed off-line, whereas unsupervised learning is performed on-line.

7. Transfer Function

The behaviour of an ANN (Artificial Neural Network) depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

- Linear (or ramp) - For linear units, the output activity is proportional to the total weighted output.
- Threshold - For threshold units, the output is set at one of two levels, depending on whether the total input is greater than or less than some threshold value.
- Sigmoid - For sigmoid units, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurones than do linear or threshold units, but all three must be considered rough approximations.



How to build Neural Network from scratch in Python

Creating a Neural Network class in Python is easy.

```
class
NeuralNetwork:
                  def __init__(self, x, y):
                      self.input      = x
                      self.weights1   = np.random.rand(self.input.shape[1],4)
                      self.weights2   = np.random.rand(4,1)
                      self.y          = y
                      self.output     = np.zeros(y.shape)
```

Training the Neural Network

The output $\hat{y}$ of a simple 2-layer Neural Network is:

$$\hat{y} = \sigma(W_2 \sigma(W_1 x + b_1) + b_2)$$

You might notice that in the equation above, the weights ***W*** and the biases ***b*** are the only variables that affects the output $\hat{y}$.

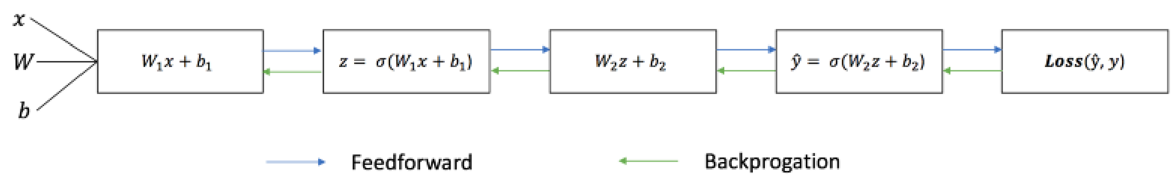
Naturally, the right values for the weights and biases determines the strength of the predictions. The process of fine-tuning the weights and biases from the input data is known as **training the Neural Network.**

Each iteration of the training process consists of the following steps:

- Calculating the predicted output $\hat{y}$, known as **feedforward**

- Updating the weights and biases, known as **backpropagation**

The sequential graph below illustrates the process.



Feedforward

As we've seen in the sequential graph above, feedforward is just simple calculus and for a basic 2-layer neural network, the output of the Neural Network is:

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

Let's add a feedforward function in our python code to do exactly that. Note that for simplicity, we have assumed the biases to be 0.

```
class
NeuralNetwork:
                    def __init__(self, x, y):
                        self.input      = x
                        self.weights1   = np.random.rand(self.input.shape[1],4)
                        self.weights2   = np.random.rand(4,1)
                        self.y          = y
                        self.output     = np.zeros(self.y.shape)

                    def feedforward(self):
                        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
                        self.output = sigmoid(np.dot(self.layer1, self.weights2))
```

However, we still need a way to evaluate the "goodness" of our predictions (i.e. how far off are our predictions)? The **Loss Function** allows us to do exactly that.

Loss Function

There are many available loss functions, and the nature of our problem should dictate our choice of loss function. In this tutorial, we'll use a simple **sum-of-squares error** as our loss function.

$$Sum - of - Squares\ Error = \sum_{i=1}^{n} (y - \hat{y})^2$$

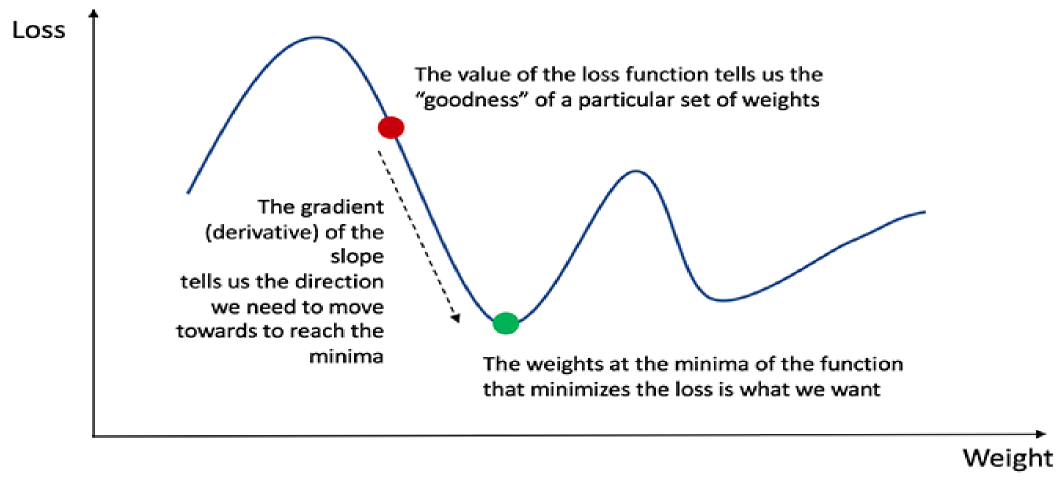
That is, the sum-of-squares error is simply the sum of the difference between each predicted value and the actual value. The difference is squared so that we measure the absolute value of the difference.

Backpropagation

Now that we've measured the error of our prediction (loss), we need to find a way to **propagate** the error back, and to update our weights and biases.

To know the appropriate amount to adjust the weights and biases by, we need to know the **derivative of the loss function with respect to the weights and biases**.

Recall from calculus that the derivative of a function is simply the slope of the function.


Loss
The value of the loss function tells us the "goodness" of a particular set of weights
The gradient (derivative) of the slope tells us the direction we need to move towards to reach the minima
The weights at the minima of the function that minimizes the loss is what we want
Weight

If we have the derivative, we can simply update the weights and biases by increasing/reducing with it. This is known as **gradient descent**.

However, we can't directly calculate the derivative of the loss function with respect to the weights and biases because the equation of the loss function does not contain the weights and biases. Therefore, we need the **chain rule** to help us calculate it.

$$Loss(y,\hat{y}) = \sum_{i=1}^{n}(y - \hat{y})^2$$

$$\frac{\partial\, Loss(y,\hat{y})}{\partial W} = \frac{\partial Loss(y,\hat{y})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W} \qquad \text{where } z = Wx + b$$

$$= 2(y - \hat{y}) * \text{derivative of sigmoid function} * x$$

$$= 2(y - \hat{y}) * \text{z(1-z)} * x$$

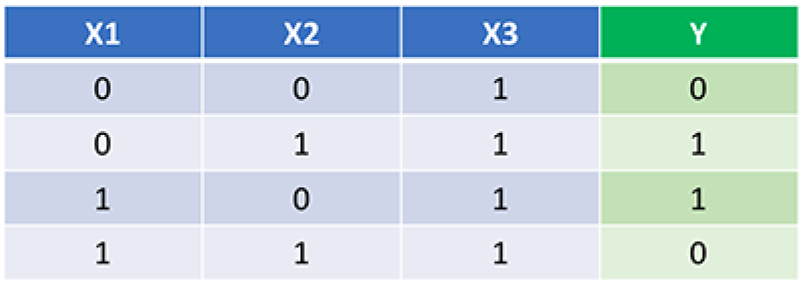
Chain rule for calculating derivative of the loss function with respect to the weights. Note that for simplicity, we have only displayed the partial derivative assuming a 1-layer Neural Network.

Now that we have that, let’s add the backpropagation function into our python code.

```
class NeuralNetwork:
    def __init__(self, x, y):
        self.input      = x
        self.weights1   = np.random.rand(self.input.shape[1],4)
        self.weights2   = np.random.rand(4,1)
        self.y          = y
        self.output     = np.zeros(self.y.shape)

    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.output = sigmoid(np.dot(self.layer1, self.weights2))

    def backprop(self):
        # application of the chain rule to find derivative of the loss function with
respect to weights2 and weights1
        d_weights2 = np.dot(self.layer1.T, (2*(self.y - self.output) *
sigmoid_derivative(self.output)))
        d_weights1 = np.dot(self.input.T,  (np.dot(2*(self.y - self.output) *
sigmoid_derivative(self.output), self.weights2.T) * sigmoid_derivative(self.layer1)))

        # update the weights with the derivative (slope) of the loss function
        self.weights1 += d_weights1
        self.weights2 += d_weights2
```

Putting it all together

Now that we have our complete python code for doing feedforward and backpropagation, let’s apply our Neural Network on an example and see how well it does.

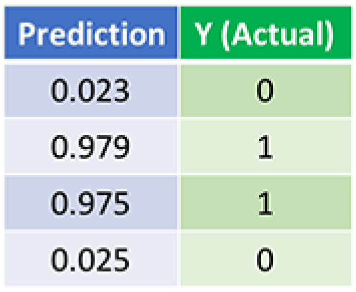


X1	X2	X3	Y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Our Neural Network should learn the ideal set of weights to represent this function. Note that it isn’t exactly trivial for us to work out the weights just by inspection alone.

Let’s train the Neural Network for 1500 iterations and see what happens. Looking at the loss per iteration graph below, we can clearly see the loss **monotonically decreasing towards a minimum.** This is consistent with the gradient descent algorithm that we’ve discussed earlier.

Let’s look at the final prediction (output) from the Neural Network after 1500 iterations.



Prediction	Y (Actual)
0.023	0
0.979	1
0.975	1
0.025	0

References:

https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html

https://github.com/mnielsen/neural-networks-and-deep-learning/tree/master/src

http://neuralnetworksanddeeplearning.com/

https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/

https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6

https://pdfs.semanticscholar.org/9f71/2bd6b2118758152aeace0a0e7a6205325f53.pdf

http://www2.ccc.uni-erlangen.de/publications/ANN-book/publications/ACS-41-94.pdf

https://pdfs.semanticscholar.org/9f71/2bd6b2118758152aeace0a0e7a6205325f53.pdf

https://medium.com/@curiousily/tensorflow-for-hackers-part-iv-neural-network-from-scratch-1a4f504dfa8