

Introduction to neural networks by Rohitash Chandra

Back to perceptron

The perceptron is the building block of neural networks. It typically features a number of connected inputs and computes an output based on the weighted sum of incoming connections (synapses), also known as the activation function.

The below figure is an example of a model where the sum of the incoming links (w_1, w_2, \dots, w_N) over the inputs (x_1, x_2, \dots, x_N) is computed and then the output (Z) goes through an activation function (which is linear step function in this case). The activation function can be changed to hyperbolic tangent (\tanh) and sigmoid depending on the problem (Mitchell, 1997).

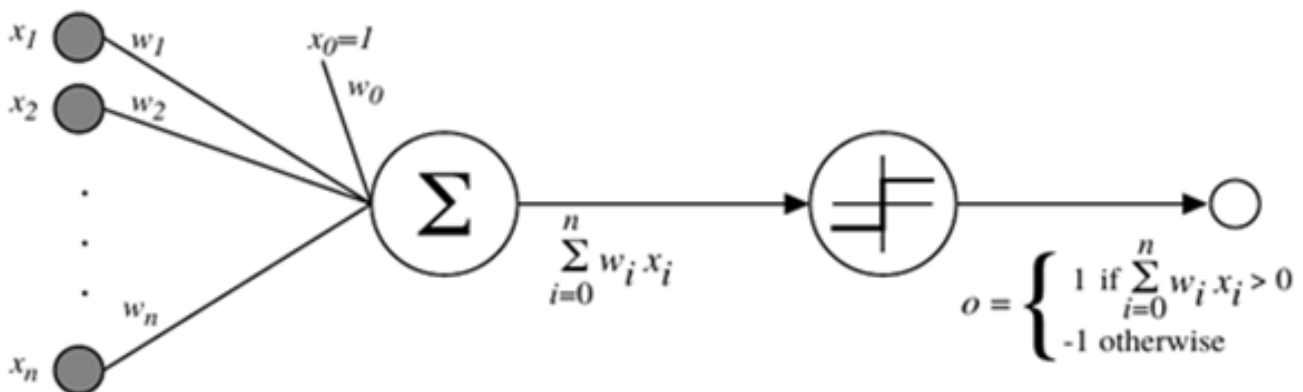


Figure: Activation function. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

The output value o is

$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

Sometimes simpler vector notation can be used as:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise} \end{cases}$$

Summary (Mitchell, 1997):

Perceptron training rule is guaranteed to succeed if

- training examples are separable linearly
- learning rate η is considerably small.

Linear unit training rule uses gradient descent and succeeds

- to converge to hypothesis with minimum squared error (guaranteed)
- if learning rate η is considerably small
- even if the training data has noise and not separable by H

Below is an example of how perceptron is used to learn a simple problem, namely the OR gate. The OR gate is a simple example used for basic machine learning. It is easier to learn the OR gate when compared to the XOR gate. Similar to the XOR gate, the OR gate has 4 instances that have two features x_1 and x_2 with an output (OR) which is either 0 or 1. The goal of the model is to mimic the OR gate, i.e given inputs predict the output that expresses the OR gate.

x_1	x_2	OR	
0	0	0	$w_0 + \sum_{i=1}^2 w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$
1	1	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$

$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$

$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0 \implies w_1 + w_2 > -w_0$$

One possible solution is

$$w_0 = -1, w_1 = 1.1, w_2 = 1.1$$

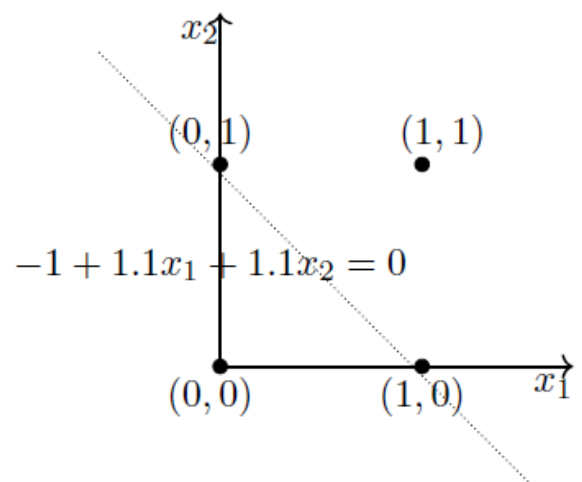


Figure: OR gate function. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

Now learn how a dot product is used to calculate the output before it goes through the activation function of the perceptron.

$$w = [w_0 + w_1 + w_2, \dots, w_n]$$

$$x = [x_0 + x_1 + x_2, \dots, x_n]$$

$$\mathbf{w} \cdot \mathbf{x} = \mathbf{w}^T \mathbf{x} = \sum_{i=0}^n w_i * x_i$$

Let's explore the perceptron learning algorithm. Firstly, we initialise \mathbf{w} with some random vector, we iterate over all the examples in the data, ($P \cup N$) both positive and negative examples. If an input \mathbf{x} belongs to P , ideally the dot product $\mathbf{w} \cdot \mathbf{x}$ would be greater than or equal to 0. If \mathbf{x} belongs to N , the dot product MUST be less than 0. The algorithm converges when all the inputs have been classified correctly.

Algorithm: Perceptron Learning Algorithm

```

P ← inputs with label 1;
N ← inputs with label 0;
Initialize  $\mathbf{w}$  randomly;
while !convergence do
    Pick random  $\mathbf{x} \in P \cup N$  ;
    if  $\mathbf{x} \in P$  and  $\mathbf{w} \cdot \mathbf{x} < 0$  then
        |  $\mathbf{w} = \mathbf{w} + \mathbf{x}$  ;
    end
    if  $\mathbf{x} \in N$  and  $\mathbf{w} \cdot \mathbf{x} \geq 0$  then
        |  $\mathbf{w} = \mathbf{w} - \mathbf{x}$  ;
    end
end
//the algorithm converges when all the
inputs are classified correctly

```

Figure: Perceptron learning. Adapted from "Perceptron learning" by A. Chandra, 2018. Retrieved from <https://towardsdatascience.com/perceptron-learning-algorithm-d5db0deab975>.

The proof of convergence for the perceptron learning rule is given [here](#).

Chandra. A. (2019). Perceptron Learning Algorithm: A Graphical Explanation Of Why It Works. Retrieved from <https://towardsdatascience.com/perceptron-learning-algorithm-d5db0deab975>.

Let's explore the below Python code that demonstrates the above concept using a simple **OR** gate problem with only 2 weights and 1 bias. You can adapt it and test for **AND** gate problem.

Challenge: Test the XOR gate problem to check if the simple perceptron model can represent this problem. Note that the XOR gate is a more difficult problem and the activation function may need to be changed to a sigmoid. Compare this with the [Week 1 lesson](#).



Click on 'Run'.

```
1 #https://gist.githubusercontent.com/Thomascountz/77670d1fd621364bc41a709
2 # Copyright (c) 2018 Thomas Countz
3
4 import numpy as np
5
6
7 class Perceptron(object):
8
9     def __init__(self, no_of_inputs, threshold, learning_rate):
10         self.threshold = threshold
11         self.learning_rate = learning_rate
12         self.weights = np.zeros(no_of_inputs + 1)
13
14     def predict(self, inputs):
```

Neuron: Motivation for neural networks

Artificial neural networks (also known as neural networks) are one of the methods of machine learning that fall under the broad area of artificial intelligence. Artificial intelligence is also known as machine intelligence and has major challenges such as perception, learning, planning, motion and manipulation (robotics), social intelligence, and natural language processing. All of these challenges typically feature some form of machine learning.

Machine learning has become so popular that top technology companies such as Uber, Amazon, Facebook, and Google began advertising related roles a few years back such as machine learning engineers, machine learning scientists, deep learning scientists, and engineers.

Neural networks are prominent machine learning methods. Artificial neural networks are loosely modelled and inspired by biological neural networks such as the human brain that features billions of neurons and trillions of interconnections known as synapses. They are applicable in a range of tasks that include regression, pattern recognition or classification, time series prediction, clustering, and reinforcement learning.

Neural networks are divided into several major categories:

- Feedforward neural networks or multilayer perceptrons (simple neural networks)
- Deep neural networks
- Recurrent neural networks

In this course, we will focus on simple neural networks. We will also discuss deep neural networks.

The below figure shows how two biological neurons are connected by a synapse.

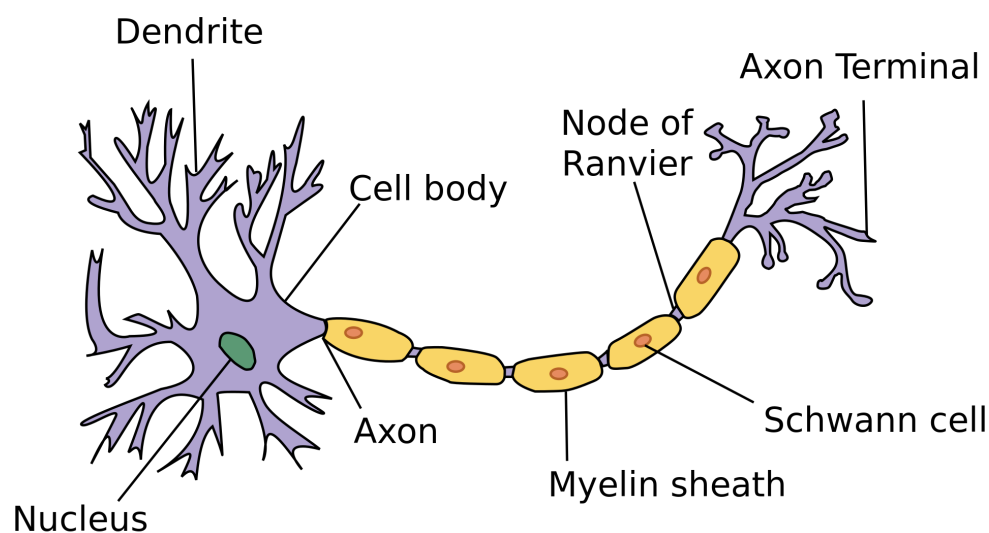


Figure: Neuron connection. Adapted from "Neuron" by Wikipedia, 2020. Retrieved from <https://simple.wikipedia.org/wiki/Neuron>.

Given some experience (data in terms of audio-visual sensory inputs), the brain learns a task by adjusting the synapses that have different forms of electrical charge which are used to represent knowledge. Some simple tasks are learnt while growing, e.g., babies learn to recognise a face and toddlers learn to walk. All these tasks require adjustments in the neural network. Try to recall some simple to complex learning tasks such as learning to drive a car or parallel parking! We do not see how our brain learns but we know it learns.

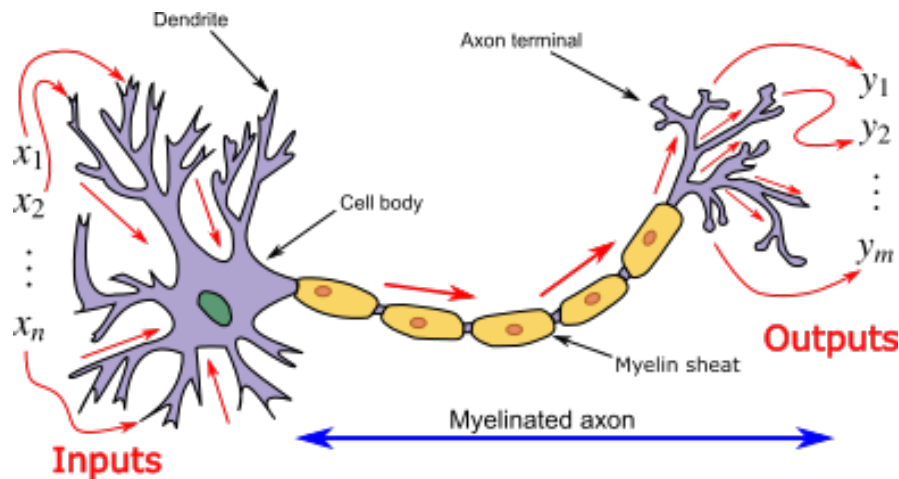


Figure: Neuron and myelinated axon, with the signal flow from inputs at dendrites to outputs at axon terminals. Adapted from "Artificial neural network" by Wikipedia, 2020. Retrieved from https://en.wikipedia.org/wiki/Artificial_neural_network.

Look at the below figures that show how a biological neural connection is used as a motivation to build the building blocks of an artificial neural network. Note the information in the artificial neural connection is stored merely by using vectors and matrices which will be demonstrated in the following lessons.

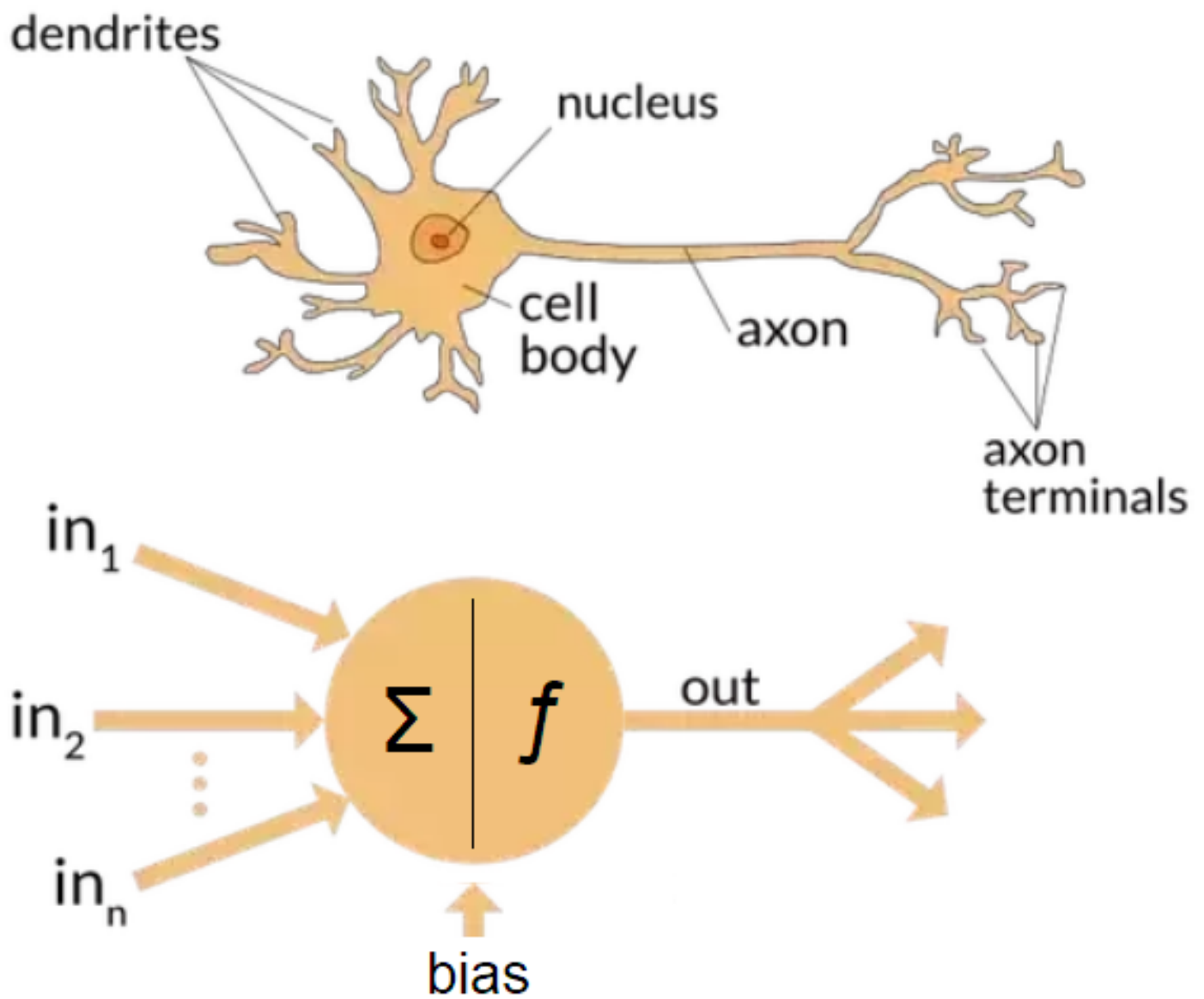


Figure: A biological and an artificial neural with inputs $(in_1, in_2, \dots, in_n)$. Adapted from "The differences between Artificial and Biological Neural Networks" by R. Nagfyi, 2018. Retrieved from <https://towardsdatascience.com/the-differences-between-artificial-and-biological-neural-networks-a8b46db828b7>.

The figure above shows inputs $(in_1, in_2, \dots, in_n)$ connected to a neuron that computes an output based on the incoming connections, weights, biases, and a transfer function. More information will be provided in the following lessons. Note that the bias is a special type of weight that features a constant (typically 1 or -1) as the input.

Applications

There are many learning tasks we can recall. Let's consider how we learned to drive—not forgetting parallel parking! Humans are poor drivers considering that there are probably millions of road accidents every year with more than a few hundred thousand fatalities. Hence, it has been a major challenge for artificial intelligence and machine learning scientists to develop autonomous driving systems, and research has been in progress for over 30 years. However, this technology is slowly becoming a reality.

Let's learn some details of one of the earliest attempts to use neural networks for an autonomous driving system. Essentially an input retina sensor (a special camera) is used for input where the video is unpacked into images. This means it is transformed into greyscale at a lower resolution so that there are not too many inputs for the neural network. Note that a backpropagation network with one hidden layer is used, and the goal of the network is to predict the angle of manoeuvring for steering of the vehicle. The training is done on a certain segment of the road, and the test is performed on the unseen road. During training, the neural network learns from the data that features the input given by the degrees of steering by the human driver.

Please see the link below if you are interested in reading the original paper on the autonomous driving system.



Pomerleau, D.A., (1989). Alvin: An autonomous land vehicle in a neural network. *Advances in neural information processing systems* (pp. 305-313). Retrieved from <https://papers.nips.cc/paper/95-alvin-an-autonomous-land-vehicle-in-a-neural-network.pdf>.

Another example of a neural network for the autonomous driving system is shown below.

Alvin driving 70mph on highways (Mitchell, 1997):

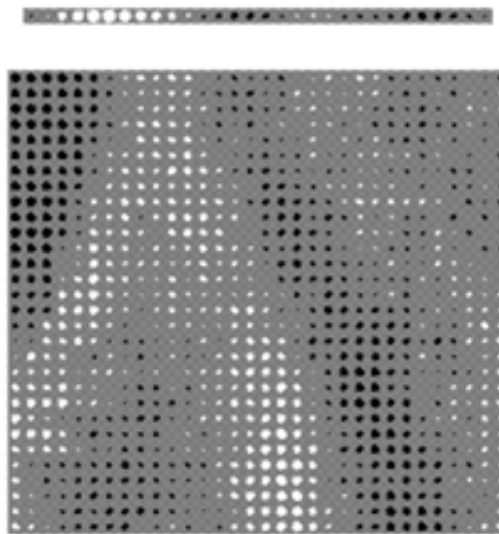
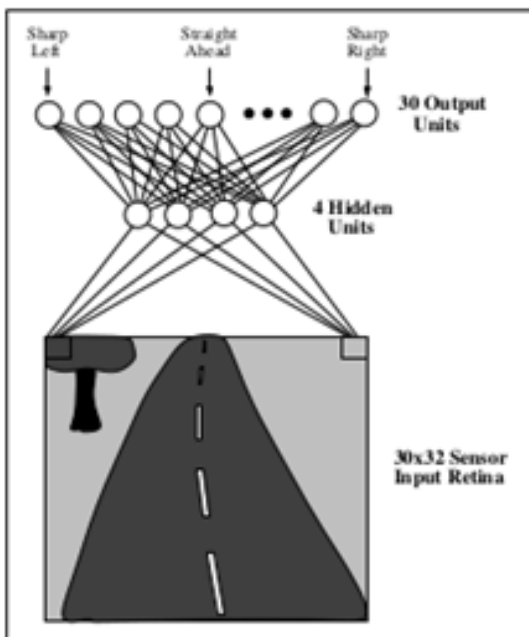


Figure: Neural network to simulate Alvin driving at speed of 70 mph. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

What are some other applications of neural networks and deep learning? There are many examples from healthcare, entertainment, robotics, advertising, and earthquake prediction.

Introduction to a feedforward neural network: Multilayer perceptron

i Learn the fundamentals of a feedforward neural network.

Now that you have recalled perceptron, let's understand the mathematics behind the activation function for neural networks and apply it in Python.

The sigmoid function is essentially the logistic activation function with a different name in the neural networks literature. Below you can see the diagram of a perceptron that uses the sigmoid activation function (Mitchell, 1997).

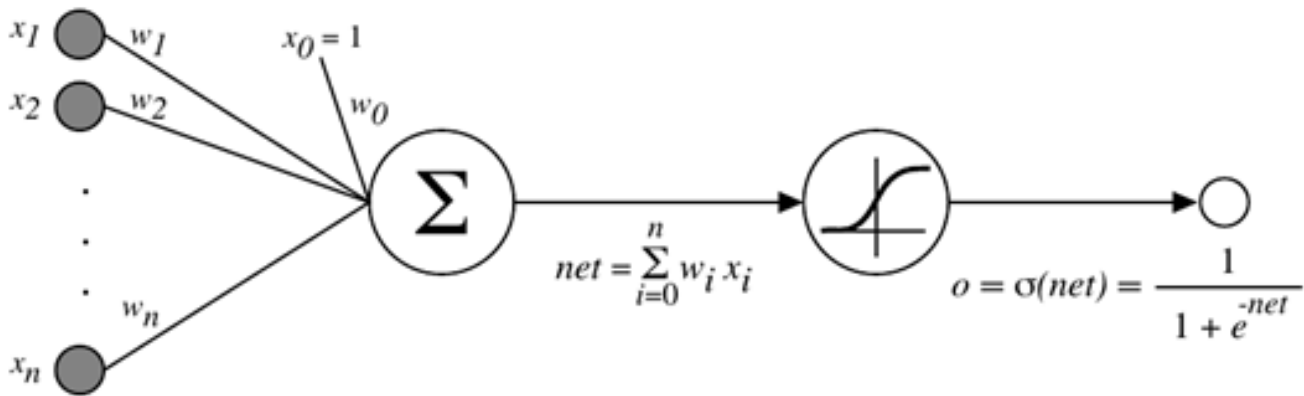


Figure: A perceptron using the sigmoid activation function. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

A multilayer perceptron, also known as feedforward network or the backpropagation network, is essentially a group of perceptrons organised into layers and trained using gradient descent known as the backpropagation algorithm.

Backpropagation algorithm

i Learn more about backpropagation and practise the given codes.

In this section, you will learn about the backpropagation algorithm which is commonly used to train feedforward networks.

The figure below shows a simple feedforward neural network topology with a single hidden layer. It is a fully connected network which means each neuron in a layer is connected to all the neurons in the previous layer. Note that the input layer is not called the neuron but just serves as input. Each neuron in the hidden layer is connected to a bias (special weight) which is not explicitly shown in this case.

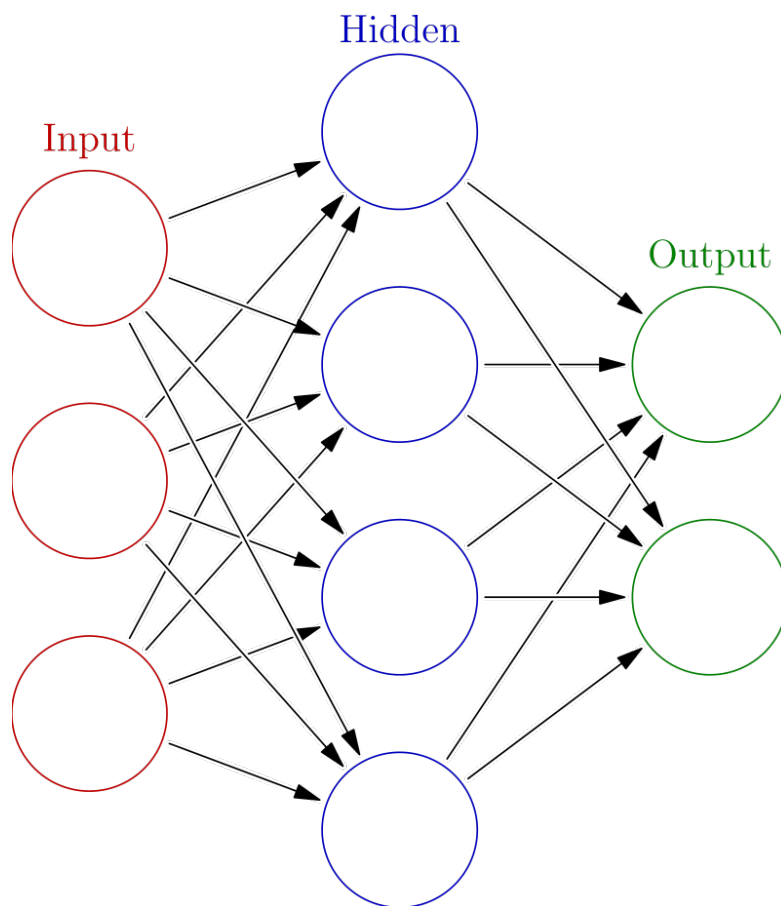


Figure: Neuron connection, Adapted from "Artificial neural network" by Wikipedia, 2020. Retrieved from

https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Colored_neural_network.svg.

The above neural network topology needs to be represented somehow, and hence we use computer

memory in the form of vectors and matrices for representation. The below code is one of the building blocks of the main code that will be used in the rest of the lessons to understand neural networks.



Note you cannot run the below codes given in this activity. You will be able to do so later with the rest of the code in the lessons to follow (See lessons titled FNN: Version one and FNN: Version two).

PYTHON



```
1
2
3 import matplotlib.pyplot as plt
4 import numpy as np
5 import random
6 import time
7
8 class Network:
9
10     def __init__(self, Topo, Train, Test, MaxTime, Samples, MinPer, lear
11         self.Top = Topo # NN topology [input, hidden, output]
12         self.Max = MaxTime # max epocs
13         self.TrainData = Train
14         self.TestData = Test
```

We continue with a detailed explanation of the backpropagation algorithm by covering key phases that are **forward pass** and the **backward pass**.

The forward pass essentially propagates information forward; from the input to one or more hidden layers, and finally to the output layer using a weighted sum of incoming weights attached to a particular neuron. Once the weighted sum is computed, the activation function (sigmoid for example) is used to compute the output of the neuron, which is then used as input in the next layer.

Forward propagation

The figure below gives a vectorised form of computing outputs in the hidden layer. Note W refers to weights from the input to the hidden layer and represents the input layer. b represents the bias in the hidden layer. Each neuron in the hidden layer has a separate bias value which is updated similarly to weight update; details will be given later. z is the output which can be either hidden or output layer.

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \quad \mathbf{a}^{[l]} = g^{[l]}(\mathbf{z}^{[l]})$$

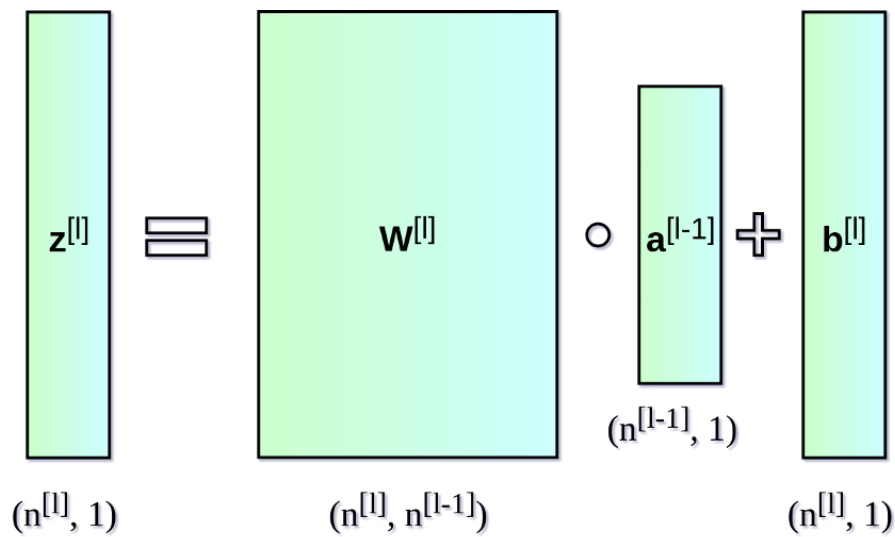


Figure: Forward propagation. Adapted from "Deep Dive into Math Behind Deep Networks" by P. Skalski, 2019. Retrieved from <https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba>.

The code below gives (from FNN: Version two) an overview of the process above, but you cannot run it now. Note that X in the code refers to vector a in the figure above.

PYTHON

```

1
2 #FNN: Version Two
3
4 def ForwardPass(self, X ):
5     z1 = X.dot(self.W1) - self.B1
6     self.hidout = self.sigmoid(z1) # output of first hidden layer
7     z2 = self.hidout.dot(self.W2) - self.B2
8     self.out = self.sigmoid(z2) # output second hidden layer
9

```



```

1 #FNN: Version One
2
3
4     def ForwardPass_Simple(self, input_vec ): # Alternative implementat
5         layer = 0 # input to hidden layer
6         weightsum_first = 0
7
8         for y in range(0, self.Top[layer+1]):
9             for x in range(0, self.Top[layer]):
10                 weightsum_first += input_vec[x] * self.W1[x,y]
11                 self.hidout[y] = self.sigmoid(weightsum_first - self.B1[y])
12                 weightsum_first = 0
13
14         layer = 1 # hidden layer to output

```

The code above (from FNN: Version one) shows a non-vectorised version of the forward pass. It will get the same output as the vectorised version, but a bit slower. The computational time would be more applicable when you have a large neural network.

Loss function

We need a way to monitor the training and have a system that calculates the error during the training process. In Week 1, we identified different error or loss functions such as root-mean-squared error (RMSE) for linear and logistic regression models, which are also applicable to neural networks. In the case of backpropagation training, the sum of squared error (SSE) is used as the loss function. The SSE loss is typically used as it is easier to differentiate it to calculate the gradients during the backward pass.

Examine the code below to understand how the SSE loss function works. It calculates the difference between the actual and predicted value for each case, squares it, and then calculates the sum for all the values.



```

1
2     def sampleEr(self, actualout):
3         error = np.subtract(self.out, actualout)
4         sqerror= np.sum(np.square(error))/self.Top[2]
5
6         return sqerror

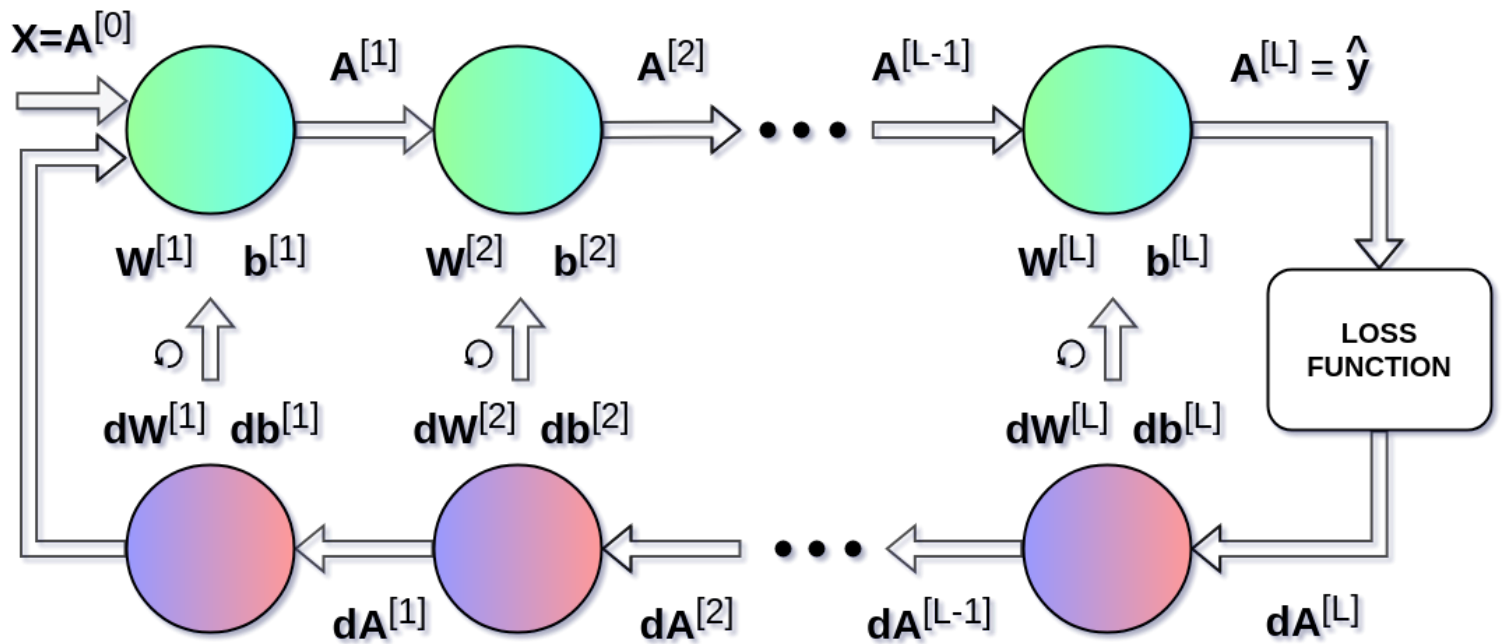
```

Forward pass

In the forward pass of the backpropagation algorithm shown in the below figure, the information from the input layer is propagated to the hidden layer or layers and finally to the output layer using a

weighted sum for each neuron, and the output of the neuron goes through the activation function. The output of the neuron is used as input for the layers ahead. The same procedure is followed for all the instances or samples in the set of training examples.

FORWARD PROPAGATION



BACKWARD PROPAGATION

Figure: Feedforward and backward propagation. Adapted from "Deep Dive into Math Behind Deep Networks" by P. Skalski, 2019. Retrieved from <https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba>.

Backward pass

In the backward pass of the backpropagation algorithm shown in the above figure, the information from the output layer is propagated back by computing gradients first at the output layer and propagating those gradients back to the hidden layer or layers. The gradients are used to calculate weight update for each layer, i.e. output—hidden layer weights, hidden to input layer weights for the case of a single hidden layer neural network. The same procedure is followed for all the instances or samples in the set of training examples.

Below is a set of equations that show how the error gradient is derived in a vectorised format. W refers to the weights and dW refers to the change in weights. We will get into details of this derivation in the upcoming lessons.

$$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$$

$$b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

$$dW^{[l]} = \frac{\partial L}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$


$$db^{[l]} = \frac{\partial L}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^m dZ^{[l](i)}$$

$$dA^{[l-1]} = \frac{\partial L}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]}$$

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]})$$

The code below shows the implementation of a backward pass in the neural network. Note that you cannot run the code at this stage. It will be available for you in the lessons to follow.

The FNN: Version one code (given below) implements the backward pass from scratch using NumPy arrays. It demonstrates how **nested for loops** are used to compute dot products in order to compute the gradients and the weight update.

PYTHON 

```

1 # FNN: Version One
2 # self.Top = [3,4,2]
3     def BackwardPass_Simple(self, input_vec, desired ): # Alternative i
4
5         # compute gradients for each layer (output and hidden layer)
6
7         layer = 2 #output layer
8         for x in range(0, self.Top[layer]):
9             self.out_delta[x] = (desired[x] - self.out[x])*(self.out[x]
10
11         layer = 1 # hidden layer
12         temp = 0
13         for x in range(0, self.Top[layer]):
14             for y in range(0, self.Top[layer+1]):

```

The **FNN: Version two** code (given below) implements the backward pass using NumPy arrays while utilising built-in dot product. The purpose of this code is to show an alternative implementation for faster computation.



```

1 #FNN Version Two
2
3
4
5     def BackwardPass(self, input_vec, desired):
6         out_delta = (desired - self.out)*(self.out*(1-self.out))
7         hid_delta = out_delta.dot(self.W2.T) * (self.hidout * (1-self.hi
8
9         self.W2+= self.hidout.T.dot(out_delta) * self.learn_rate
10        self.B2+= (-1 * self.learn_rate * out_delta)
11
12        self.W1 += (input_vec.T.dot(hid_delta) * self.learn_rate)
13        self.B1+= (-1 * self.learn_rate * hid_delta)

```

Backpropagation algorithm

The backpropagation algorithm essentially employs the above code that presented the forward and the backwards pass for the training samples. **An epoch** is completed when all the instances in the training examples have been utilised. For each instance in the training example, the forward and backward passes are performed to update the weights of the neural network, until the termination condition is satisfied which is given by the maximum number of epochs or minimum value of error (in terms of the SSE).

Below are the steps to implement the backpropagation algorithm (Mitchell, 1997):

Initialise all weights to small random numbers and do the following:

1. For each training example, input the training example to the network and compute the outputs of the network.
2. For each output unit k

$$\delta_k \longleftarrow o_k (1 - o_k) (t_k - o_k)$$

3. For each hidden unit h

$$\delta_h \longleftarrow o_h (1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k$$

4. Update each network weight $w_{i,j}$

$$w_{i,j} \longleftarrow w_{i,j} + \Delta w_{i,j}$$

where

$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

PYTHON



```

1 #FNN: Version Two
2
3     def BP_GD(self):
4
5
6         Input = np.zeros((1, self.Top[0])) # temp hold input
7         Desired = np.zeros((1, self.Top[2]))
8
9
10        Er = []
11        epoch = 0
12        bestmse = 10000 # assign a large number in begining to maintain
13        bestTrain = 0
14        while epoch < self.Max and bestTrain < self.minPerf :
  
```

The code above shows how the backpropagation algorithm uses the forward and backward pass to train a neural network by adjusting the weights and biases.

Test network

Once the neural network has been trained by reaching either of the termination conditions, we load the testing data set to test the performance of the neural network in order to demonstrate the generalisation ability as shown below.

PYTHON



```

1
2
3     def TestNetwork(self, Data, testSize, erTolerance):
4         Input = np.zeros((1, self.Top[0])) # temp hold input
5         Desired = np.zeros((1, self.Top[2]))
6         nOutput = np.zeros((1, self.Top[2]))
7         clasPerf = 0
8         sse = 0
9         self.W1 = self.BestW1
10        self.W2 = self.BestW2 #load best knowledge
11        self.B1 = self.BestB1
12        self.B2 = self.BestB2 #load best knowledge
13
14        for s in range(0, testSize):
  
```

Note that a detailed explanation of the code will be given in lessons to follow as well as in the tutorial video of FNN: Version one.

Training or learning?

But what is a neural network?

Watch the below video that demonstrates the learning process for optical character recognition.

Source: 3BLUE1BROWN. (2017, October 05). *But what is a Neural Network?* [online video]. Retrieved from <https://www.youtube.com/watch?v=aircAruvnKk>.



Understand how an algorithm learns a problem.

Now that we have explained the backpropagation algorithm, it is worthwhile to visualise what happens when it learns a problem.

The below figure shows a multi-layer network where the multilayered perceptron is trying to act as an autoencoder, i.e., mimic the input as the output. Can this target function be learned?

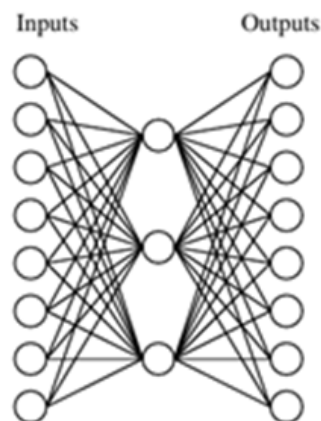


Figure: A multi-layer network. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

Input	Output
10000000	→ 10000000
01000000	→ 01000000
00100000	→ 00100000
00010000	→ 00010000
00001000	→ 00001000
00000100	→ 00000100
00000010	→ 00000010
00000001	→ 00000001

Figure: A target function. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

The below figure shows the examples of outputs of hidden neuron weights (with 3 hidden neurons) after the neural network learns. Given that you have a code of backpropagation neural network, you can easily make these data sets and train to see what happens!

Input	Hidden Values	Output
10000000	→ .89 .04 .08 →	10000000
01000000	→ .01 .11 .88 →	01000000
00100000	→ .01 .97 .27 →	00100000
00010000	→ .99 .97 .71 →	00010000
00001000	→ .03 .05 .02 →	00001000
00000100	→ .22 .99 .99 →	00000100
00000010	→ .80 .01 .98 →	00000010
00000001	→ .60 .94 .01 →	00000001

Figure: Learning hidden layers representations. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

Here is a presentation of the weights over time in terms of epochs during the learning process.

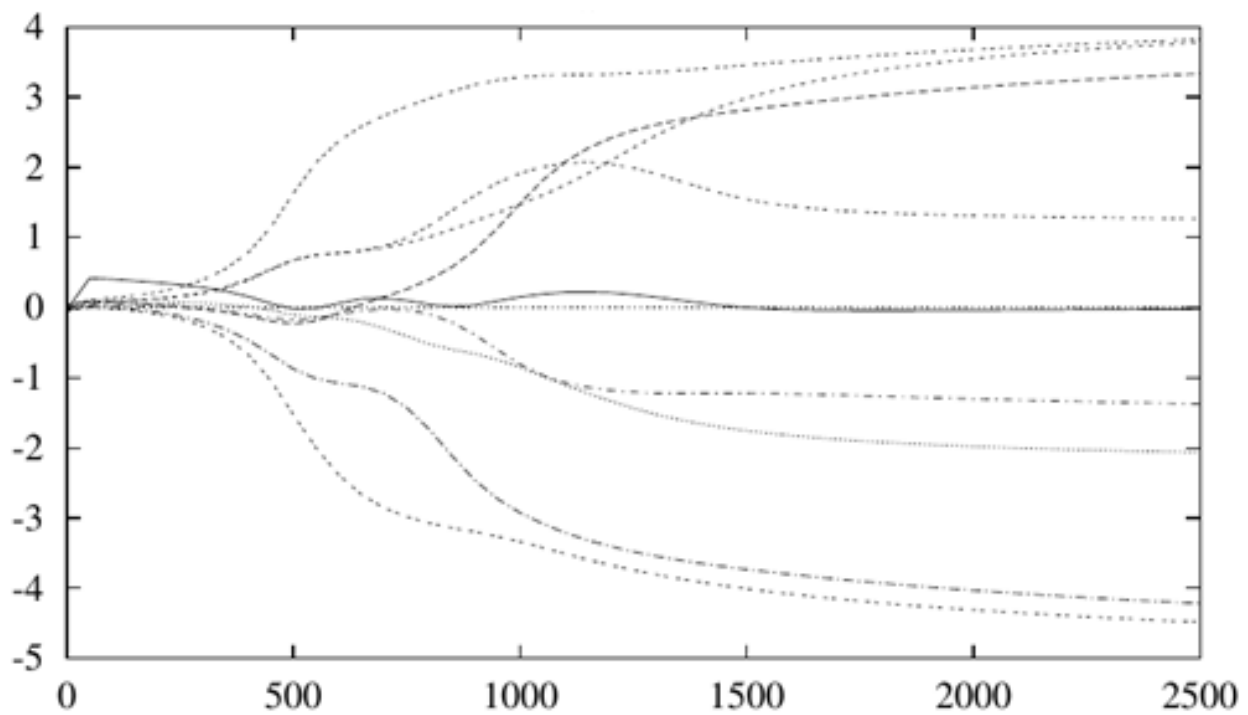


Figure: Weights from inputs to one hidden unit. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

The below figure shows the error at each output unit over time. As you are designing the backpropagation neural network from scratch, you can visualise (plot) error as well by saving the information in a file when it learns and plotting it.

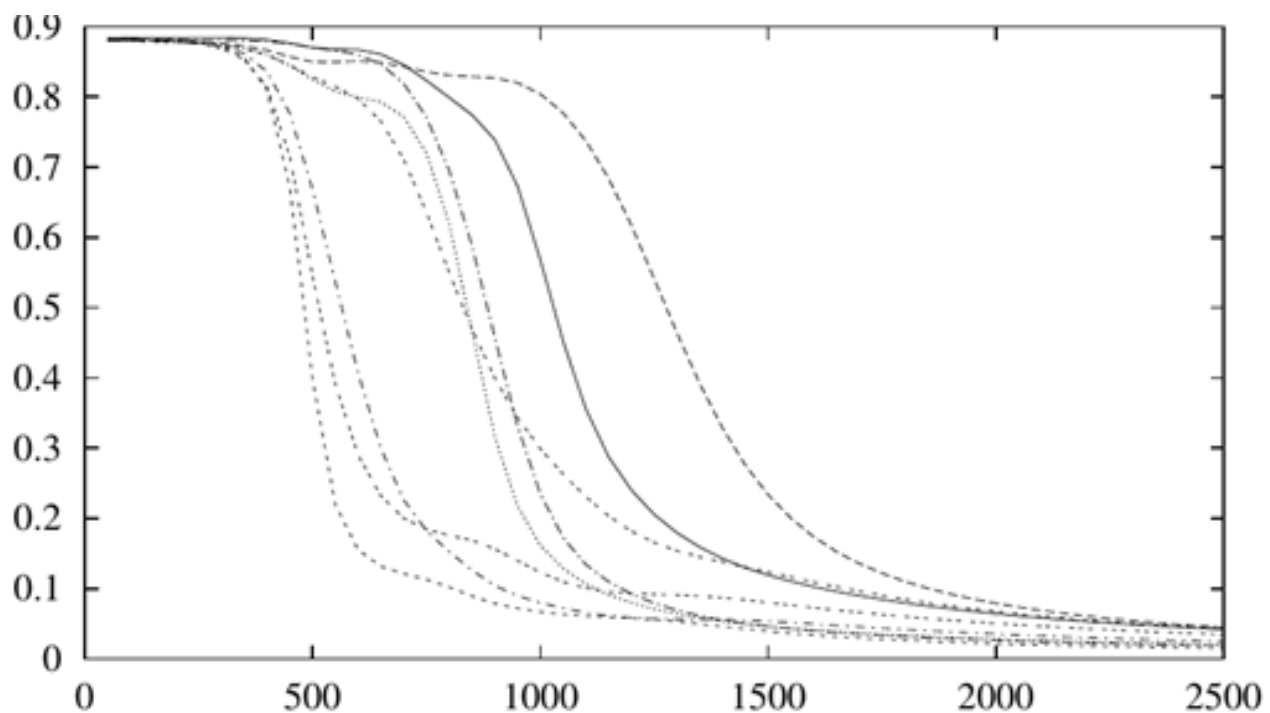


Figure: Sum of squared errors for each output unit. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

Derivation for error gradient



Dive into the derivation for gradient descent.

So far, we showed how the neural network is trained with weight updates, but have not shown the derivation with the gradients taken into account. This section discusses the derivation for error gradient.

The below figure goes back to the sigmoid unit of the perceptron, which we have shown previously, and the derivative of the sigmoid unit is shown (Mitchell, 1997).

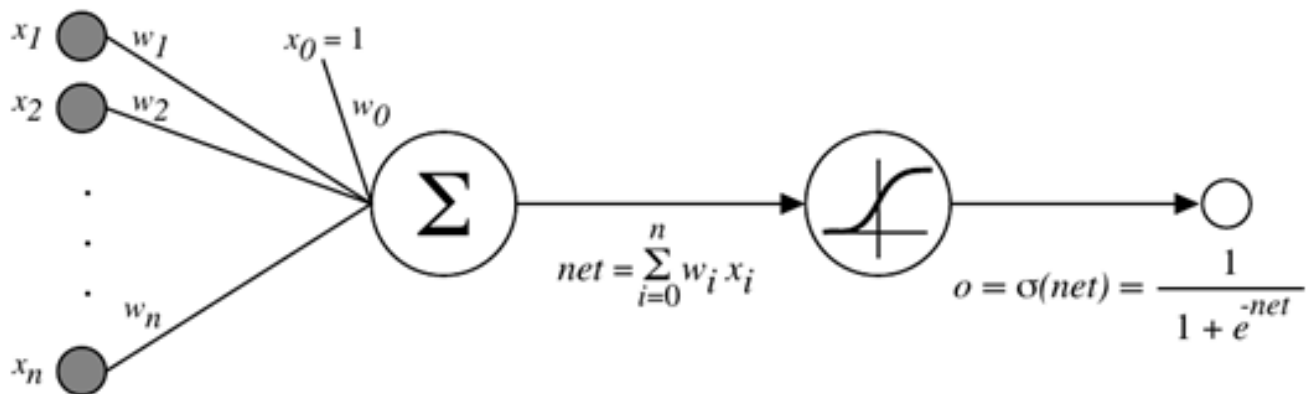


Figure: A perceptron using the sigmoid activation function. Adapted from Machine Learning by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

Earlier, we showed the above with linear activation function and presented the derivation for the gradient computation for weight update. Now we extend that idea to the case where the sigmoid is used instead of linear activation.

$\sigma(x)$ is the sigmoid function

$$\frac{1}{1 + e^{-x}}$$

The property of the function is

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Now derive gradient descent rules to train

- one sigmoid unit
- multilayer networks of sigmoid units which is backpropagation.

Furthermore, the error in terms of the sum-squared error is used to derive the process of computing error gradients from the output back to the hidden and then the input later, for their respective weight updates.

If necessary, revise partial derivatives and chain rule by clicking on the links below:

1. [Tutorial 1](#) Partial derivatives (MathisFun, n.d.)
2. [Tutorial 2](#) Partial derivatives (Khan Academy, n.d.)
3. [Tutorial 3](#) Chain rule (Khan Academy, n.d.)

Note that the chain rule is used for differentiation and below is the derivation for error gradient for a sigmoid unit.

$$\begin{aligned}
 \frac{dz}{dx} &= \frac{dz}{dy} \cdot \frac{dy}{dx} \\
 \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
 &= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
 &= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
 &= \sum_d (t_d - o_d) \left(-\frac{\partial o_d}{\partial w_i} \right) \\
 &= - \sum_d (t_d - o_d) \left(\frac{\partial o_d}{\partial w_i} \right) \left(\frac{\partial net_d}{\partial w_i} \right)
 \end{aligned}$$

But we know:

$$\begin{aligned}
 \left(\frac{\partial o_d}{\partial net_d} \right) &= \left(\frac{\partial \sigma net_d}{\partial net_d} \right) = o_d (1 - o_d) \\
 \left(\frac{\partial net_d}{\partial w_i} \right) &= \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}
 \end{aligned}$$

$$\frac{\partial E}{\partial w_i} = - \sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

Further details regarding derivation:

<https://folk.idi.ntnu.no/keithd/classes/advai/lectures/backprop.pdf>

Further equations: https://www.python-course.eu/neural_networks_backpropagation.php

Advanced topics:

- Convergence proofs:
https://gowerrobert.github.io/pdf/M2_statistique_optimisation/grad_conv.pdf
- <http://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote03.html>

Activation functions and layers



Learn about activation functions and layers.

We covered the basics of activation function in the previous lessons. Here we cover more tails with examples of other prominent activation functions.

Sigmoid/Logistic activation

Note that $S(x)$ is the input computed after performing a weighted sum of incoming or attached units as shown in the perceptron example in the previous lesson.

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

The below figure shows the sigmoid/logistic function output for the given input x .

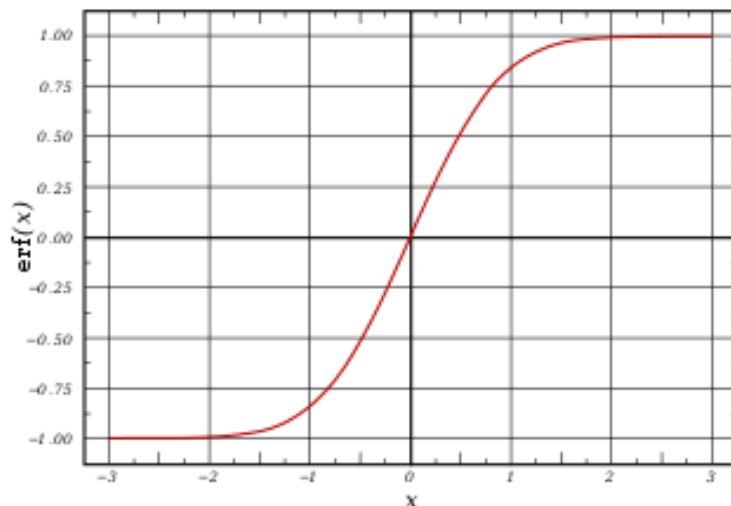


Figure: Sigmoid function w.r.t x



Practise the code on the activation function.

The code below gives the implementation of the equation above. Run the code on the activation function and try to understand how it works. It highlights the output of the sigmoid and the derivative which is essential for weight update by backpropagation.

```

1
2 #source: https://medium.com/@omkar.nallagoni/activation-functions-with-d
3
4 import matplotlib.pyplot as plt
5 import numpy as np
6
7 def sigmoid(x):
8     s=1/(1+np.exp(-x)) # this implements the sigmoid function
9     ds=s*(1-s) # this gives the derivative
10    return s,ds
11 x=np.arange(-6,6,0.01)
12 sigmoid(x)
13 # Setup centered axes
14 fig, ax = plt.subplots(figsize=(9, 5))

```

tanh activation (hyperbolic tangent)

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Below is the visualisation for the \tanh for the given input x .

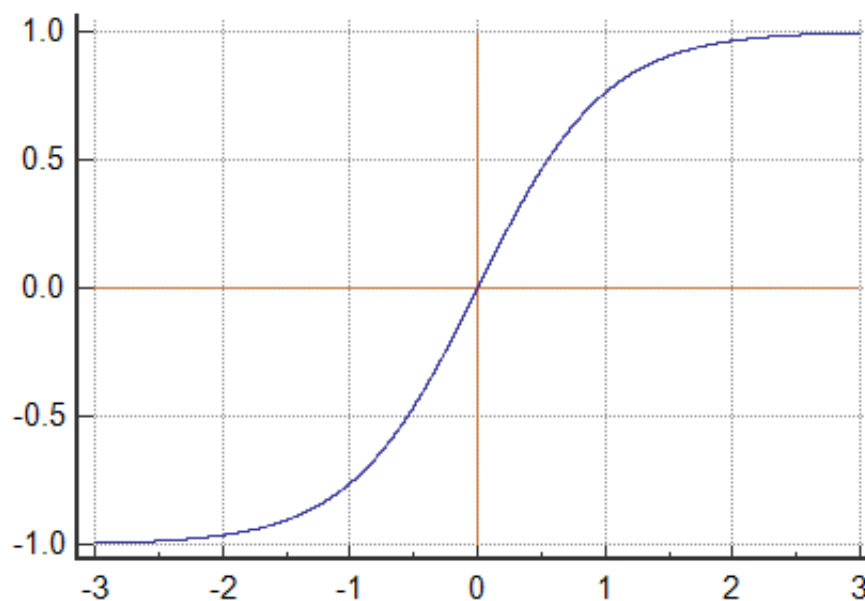


Figure: \tanh function w.r.t x

i Practise the below code to understand how \tanh function works.

Now the run the below code written in Python to understand how \tanh function works.

▶ Run

PYTHON



```
1
2 # source: https://medium.com/@omkar.nallagoni/activation-functions-with-
3 import matplotlib.pyplot as plt
4 import numpy as np
5
6 def tanh(x):
7     t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
8     dt=1-t**2 # this gives the derivative
9     return t,dt
10 z=np.arange(-4,4,0.01)
11 tanh(z)[0].size,tanh(z)[1].size
12 # Setup centered axes
13 fig, ax = plt.subplots(figsize=(9, 5))
14 ax.spines['left'].set_position('center')
```

Due to the different derivative of the sigmoid when compared to *tanh*, you will find the backpropagation algorithm's weight update different for the *tanh* function. Hence you cannot just plug a *tanh* in a backpropagation neural network that is designated for training sigmoid functions and expect it to learn.

ReLU

The rectifier also known as ReLU is an [activation function](#) defined as the positive part of its argument:

$$f(x) = x^+ = \max(0, x)$$

where x is the input.

Softplus

Softplus function provides additional smoothness to ReLU, given x as the input.

$$f(x) = \ln(1 + e^x)$$

Both Relu and Softplus are visualised below for the given input x .

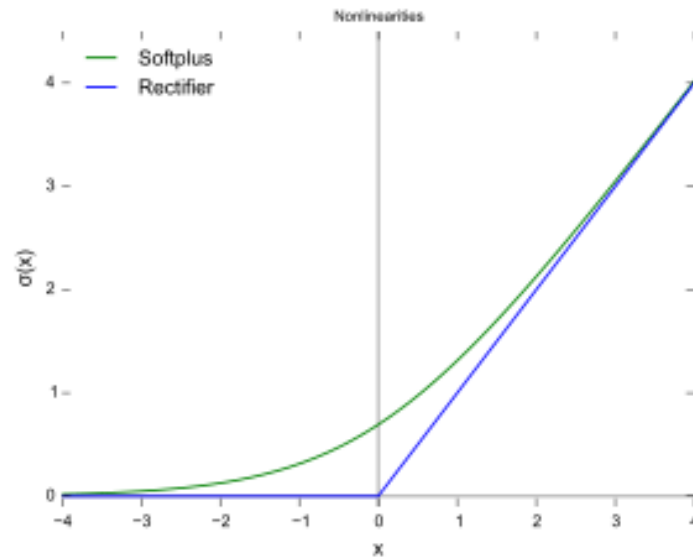


Figure: Relu and Softplus functions w.r.t x

Softmax

Softmax takes a vector z of K real numbers as input and normalises it into a probability distribution which consists of K probabilities that are proportional to the exponentials of the input numbers.

$$\sigma(x) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \text{ for } i = 1, 2, \dots, K \text{ and } z = z_1, z_2, \dots, z_K \in \mathbb{R}^K$$



Run the codes.

You can run the code below and see the outputs from the print statements to get a better understanding.

▶ Run

PYTHON



```
1 import numpy as np
2 a = [1.0, 2.0, 3.0, 4.0, 1.0, 2.0, 3.0]
3 print(np.exp(a))
4 print(np.sum(np.exp(a)))
5
6 soft_max = np.exp(a) / np.sum(np.exp(a))
7 print(soft_max)
8
9
```

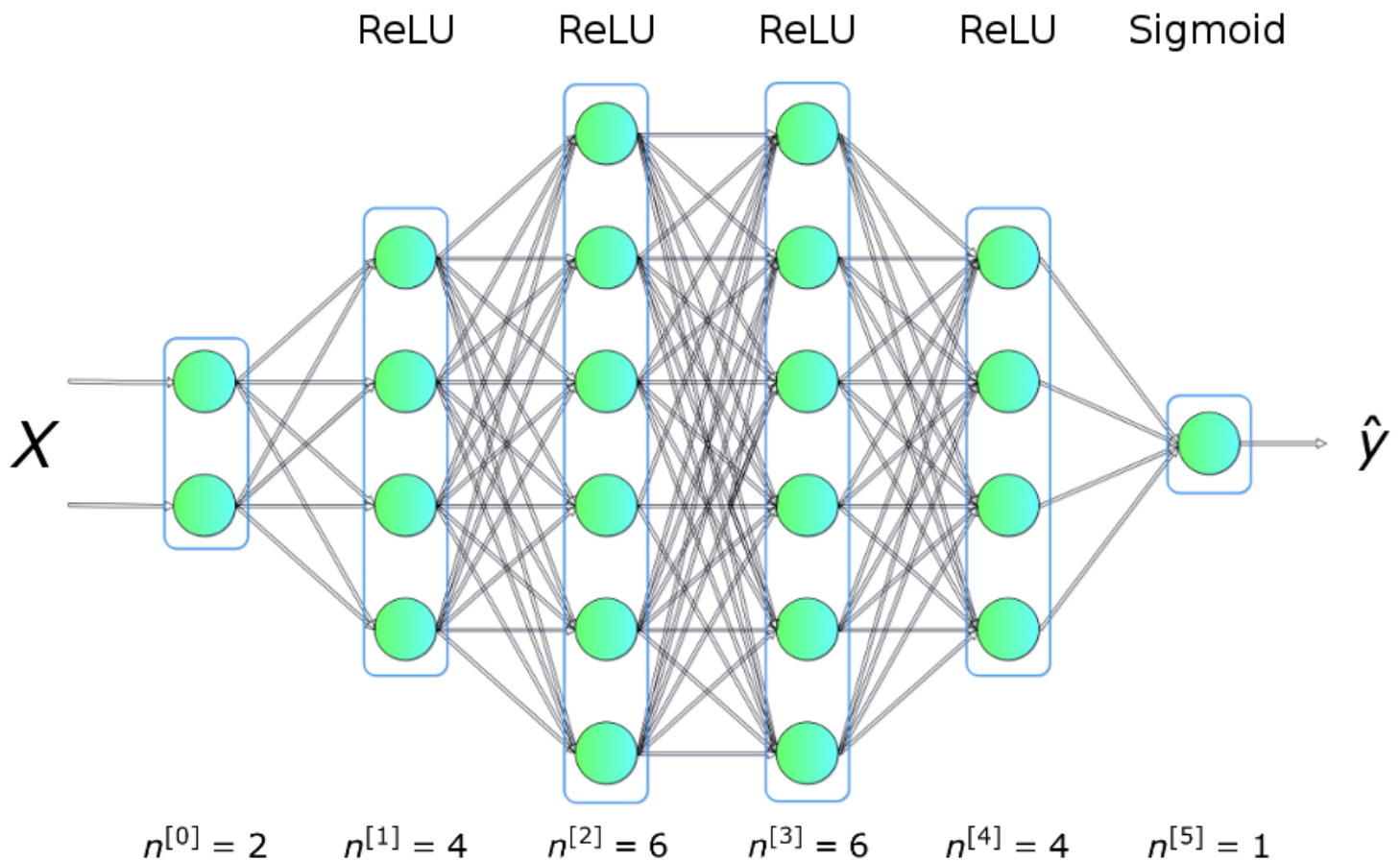
Run the below code to understand softmax in R.

▶ Run

R

```
1 z <- c(1.0, 2.0, 3.0, 4.0, 1.0, 2.0, 3.0)
2 softmax <- exp(z)/sum(exp(z))
3 print(softmax)
```

Look at the below example of sigmoid unit in the output layer used with ReLu. There are 4 hidden layers of the multilayer perceptron.



i

Figure: Sigmoid unit in the output layer used with Relu in 4 hidden layers. Adapted from "Deep Dive into Math Behind Deep Networks" by P. Skalski, 2019 (<https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba>).

Below are additional reading on activation functions to enhance your knowledge.

[Activation functions in Keras](#)

Data processing: One hot encoding

i Learn about the one-hot encoding method to process data.

In the case of classification problems, you need a proper way to represent the class values or outcomes, especially in multi-class problems. Let's learn one of the techniques called one-hot encoding to represent the class values or outcomes in this section.

A one-hot encoding is a representation of categorical variables as binary vectors. The categorical values are typically mapped to integer values and then each integer value is represented as a binary vector, as shown below.

Color		Red	Yellow	Green
Red		1	0	0
Red		1	0	0
Yellow		0	1	0
Green		0	0	1
Yellow		0	0	1

Figure: Example of one hot encoding. Adapted from "Using categorical data with one hot encoding source" by kaggle, n.d. Retrieved from <https://www.kaggle.com/dansbecker/using-categorical-data-with-one-hot-encoding>.

The code below shows how class labels from a data set can be transformed into a one-hot encoding using scikit-learn library.

i Run the below code on one hot encoding.

```
1 #source: https://machinelearningmastery.com/how-to-one-hot-encode-sequen
2
3 from numpy import array
4 from numpy import argmax
5 from sklearn.preprocessing import LabelEncoder
6 from sklearn.preprocessing import OneHotEncoder
7 # define example
8 data = ['cold', 'cold', 'warm', 'cold', 'hot', 'hot', 'warm', 'cold', 'w
9 values = array(data)
10 print(values)
11 # integer encode
12 label_encoder = LabelEncoder()
13 integer_encoded = label_encoder.fit_transform(values)
14 print(integer_encoded)
```


Data processing: UCI machine learning data repository



Let's understand the fundamentals of data processing from a neural networks perspective.

In the case of neural networks, it is best to transform the raw data set or rescale the data set between [0,1]. Different feature groups have different minimum and maximum values in different data sets and it is important to bring them together in the same distribution for better representation and to treat all features equally without any bias.

You can see the below example where scikit-learn library is used to do this. Alternatively, you can write your own code where you can track the maximum and minimum of the different features and then divide them by the maximum. Note that this can be a problem when you have negative values. To overcome this issue, you need to shift the phase and then divide them by the maximum. Do not take absolute values as they will misrepresent the features.

The code below transforms the input features of Iris data set in between 0 and 1.

▶ Run

PYTHON



```
1 # Python code to Rescale data (between 0 and 1)
2 import pandas
3 import scipy
4 import numpy
5 from sklearn.preprocessing import MinMaxScaler
6 url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/ir
7 names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width',
8 dataframe = pandas.read_csv(url, names=names)
9 array = dataframe.values
10 print(array[0:4,:])
11
12 # separate array into input and output components
13 X = array[:,0:4]
14 Y = array[:,4]
```

The figure below shows the different steps taken for data processing.

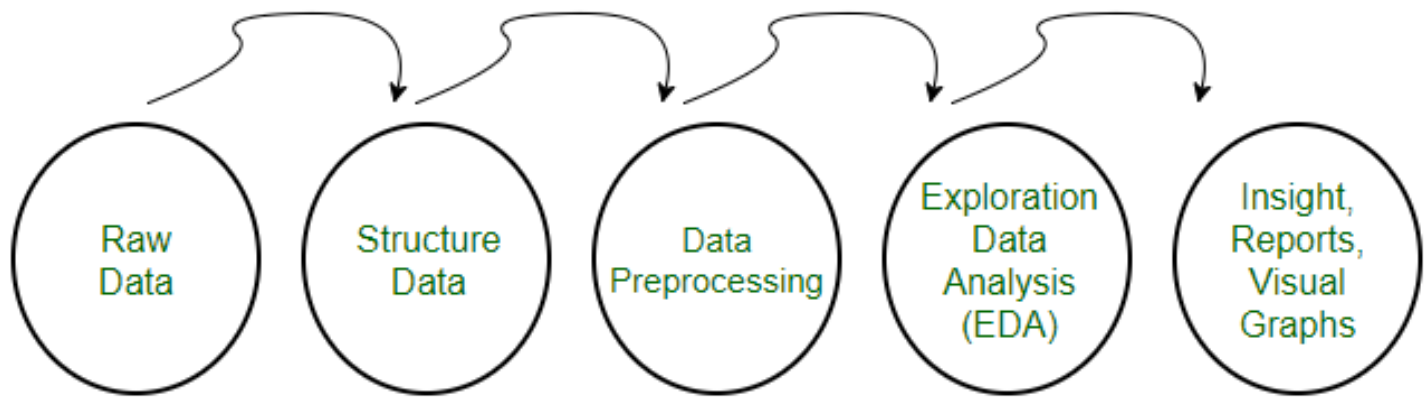


Figure: Steps of data processing. Adapted from "Data Preprocessing for Machine learning in Python" by Geeksforgeeks, 2017. Retrieved from <https://www.geeksforgeeks.org/data-preprocessing-machine-learning-python/>.

i Run the below code to transform an input data set into binary values.

The code below uses the same data set (Iris) as the above code and transforms the input features as binary.

▶ Run

PYTHON



```
1 #https://analyticsindiamag.com/data-pre-processing-in-python/
2
3 from sklearn.preprocessing import Binarizer
4 import pandas
5 import numpy
6 url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/ir
7 names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width',
8 dataframe = pandas.read_csv(url, names=names)
9 array = dataframe.values
10
11 # separate array into input and output components
12 X = array[:,0:4]
13 Y = array[:,4]
14 binarizer = Binarizer(threshold=0.0).fit(X)
```

Number of hidden layers

In this section, we will unfold the relationship between the number of hidden layers and the performance of a neural network. When hidden layers are added to a neural network, it is considered a deep neural network, and its performance is improved.

We note that determining the appropriate number of hidden layers and neurons in the hidden layers are major challenges when designing neural networks for different applications. Although one hidden layer neural network is known as a universal approximator, attempts have been made to check if adding more hidden layers helps training and generalisation. This depends on the problem, and adding more than one hidden layer does not mean you will get better performance. Note that just the addition of more hidden layers can't turn a shallow neural network into a deep neural network and will automatically improve its performance.

Deep neural networks have other architectural properties such as recurrence and convolutional layers that make them more appropriate for larger data sets, particularly multimedia applications. Convolutional neural networks, for instance, have convolutional layers that help in automatic feature extraction and are most appropriate for image data.

Further information:

1. Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5), pp. 359-366. Retrieved from <https://www.sciencedirect.com/science/article/abs/pii/0893608089900208>
2. Trenn, S. (2008). Multilayer perceptrons: Approximation order and necessary number of hidden units. *IEEE transactions on neural networks*, 19(5), pp.836-844. Retrieved from <https://ieeexplore.ieee.org/document/4469950>