Week 3: Introduction to neural networks

Introduction

Week 3 introduces you to the fundamentals of neural networks that are prominent methods for machine learning. You will extend your knowledge on gradient descent, linear regression, and logistic regression this week.

This week you will find out how gradient descent is used to train simple neural networks using the backpropagation algorithm. You will cover the details of the backpropagation algorithm and show how the error gradients are calculated. You will learn about the basic neural network architectures and activation functions. You will then implement neural networks by using benchmark synthetic and real-world datasets to demonstrate the applications for classification problems. Finally, you will cover data processing methods that are prominently used in industrial applications of neural networks.

Recommended readings

We recommend you to **first go through the lessons and then see the related topics in the textbook**. Note that **the textbook readings are not mandatory.** They are meant to provide additional information for this week's lessons.

The following chapters of the course textbook will help you with this week's lesson:

Géron, A. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Ed.). O'Reilly Media, Inc.

- Chapter 4: Training Models
- Chapter 10: Introduction to Artificial Neural Networks with Keras

Mitchell, T. (1997). Machine Learning. McGraw-Hill.

- Chapter 4: Neural Networks

All readings are available from the course Leganto reading list.

- Additional text: Friedman, J. H. (2017). *The elements of statistical learning: Data mining, inference, and prediction*. springer open. https://web.stanford.edu/~hastie/Papers/ESLII.pdf (open book)

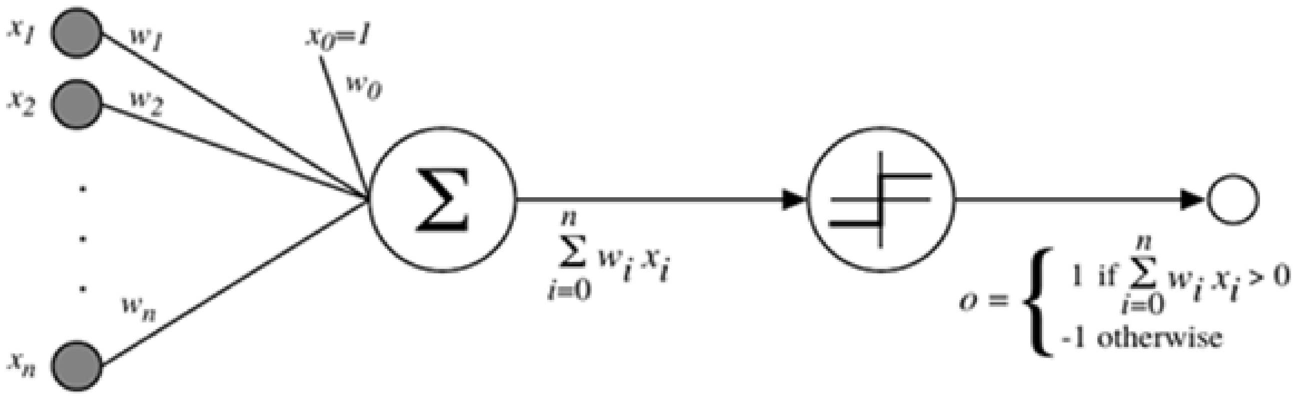
This week's lessons were prepared by Dr. R. Chandra. If you have any questions or comments, please email directly: rohitash.chandra@unsw.edu.au

Back to perceptron

Let's recall perceptron learning rules and apply them in a neural network.

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 $(w_1, w_2, ...w_N)$ over the inputs $(x_1, x_2, ...x_N)$ 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).



x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

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

The output value o is

$$o(x_1, ...x_n) = \begin{cases} 1 & if w_0 + w_1x_1 + ...w_nx_n > 0 \\ -1 & otherwise \end{cases}$$

Sometimes simpler vector notation can be used as:

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

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

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

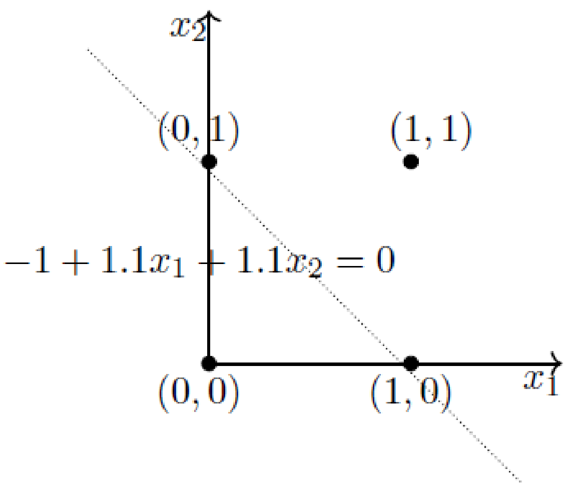


Figure: OR gate function. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

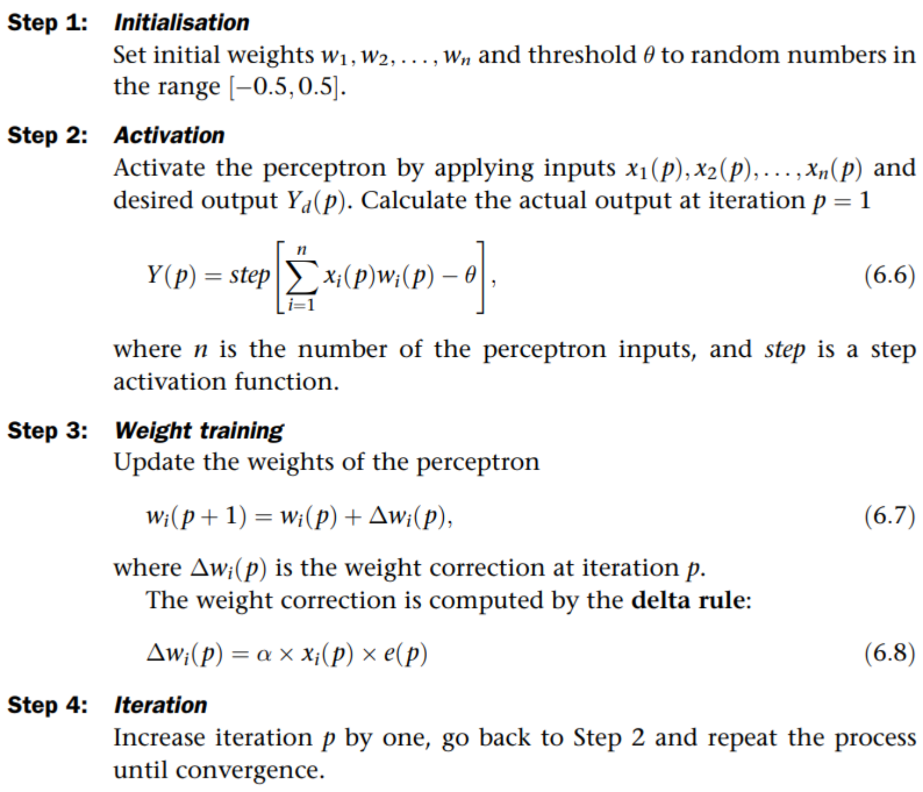
Now learn how a dot product is used to calculate the output before it goes through the activation function of the perceptron.

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

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

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

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



Step 1: ***Initialisation***
Set initial weights $w_1, w_2, \ldots, w_n$ and threshold θ to random numbers in the range $[-0.5, 0.5]$.

Step 2: ***Activation***
Activate the perceptron by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired output $Y_d(p)$. Calculate the actual output at iteration $p = 1$

$$Y(p) = step\left[\sum_{i=1}^{n} x_i(p) w_i(p) - \theta\right], \qquad (6.6)$$

where n is the number of the perceptron inputs, and *step* is a step activation function.

Step 3: ***Weight training***
Update the weights of the perceptron

$$w_i(p+1) = w_i(p) + \Delta w_i(p), \qquad (6.7)$$

where $\Delta w_i(p)$ is the weight correction at iteration p.
The weight correction is computed by the **delta rule**:

$$\Delta w_i(p) = \alpha \times x_i(p) \times e(p) \qquad (6.8)$$

Step 4: ***Iteration***
Increase iteration p by one, go back to Step 2 and repeat the process until convergence.

Source: Negnevitsky, M. (2005). *Artificial intelligence: a guide to intelligent systems*. Pearson education. (Chapter 6: Artificial Neural Networks)

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.

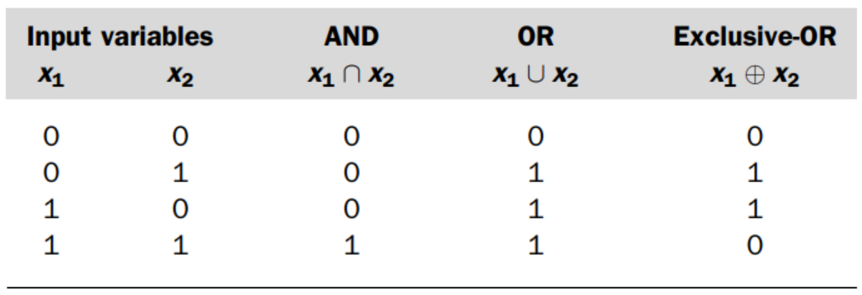
Click on 'Run'.

```
#https://gist.githubusercontent.com/Thomascountz/77670d1fd621364bc41a7094563a7b9c/raw/39091aea4beb7
# Copyright (c) 2018 Thomas Countz

import numpy as np
```

```
class Perceptron(object):

    def __init__(self, no_of_inputs, threshold, learning_rate):
        self.threshold = threshold
        self.learning_rate = learning_rate
        self.weights = np.zeros(no_of_inputs + 1)

    def predict(self, inputs):
        summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
        if summation > 0:
          activation = 1
        else:
          activation = 0
        return activation

    def train(self, training_inputs, labels):
        for _ in range(self.threshold):

            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                #print(prediction, label)
                self.weights[1:] += self.learning_rate * (label - prediction) * inputs
                self.weights[0] += self.learning_rate * (label - prediction) #bias


# set the dataset inputs
# AND gate

training_inputs = []
training_inputs.append(np.array([1, 1]))
training_inputs.append(np.array([1, 0]))
training_inputs.append(np.array([0, 1]))
training_inputs.append(np.array([0, 0]))

# set the dataset class labels
labels = np.array([1, 0, 0, 0])# AND
#labels = np.array([1, 0, 0, 1]) #XOR
#labels = np.array([1, 1, 1, 0]) # OR

no_of_inputs = 2
threshold=100
learning_rate=0.01

# set the class and train
perceptron = Perceptron(no_of_inputs, threshold, learning_rate)
perceptron.train(training_inputs, labels)

# now test trained percepton
inputs = np.array([1, 1])
output = perceptron.predict(inputs)
print(output, ' for input [1, 1]' )
```

```
inputs = np.array([0, 1])
output = perceptron.predict(inputs)
print(output, ' for input [0, 1]' )
```

Table 6.2 Truth tables for the basic logical operations



Input variables		AND	OR	Exclusive-OR
x_1	x_2	$x_1 \cap x_2$	$x_1 \cup x_2$	$x_1 \oplus x_2$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

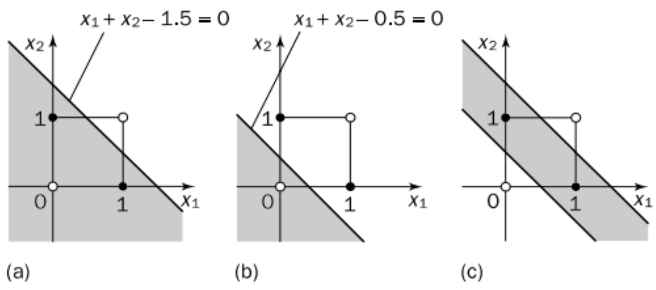
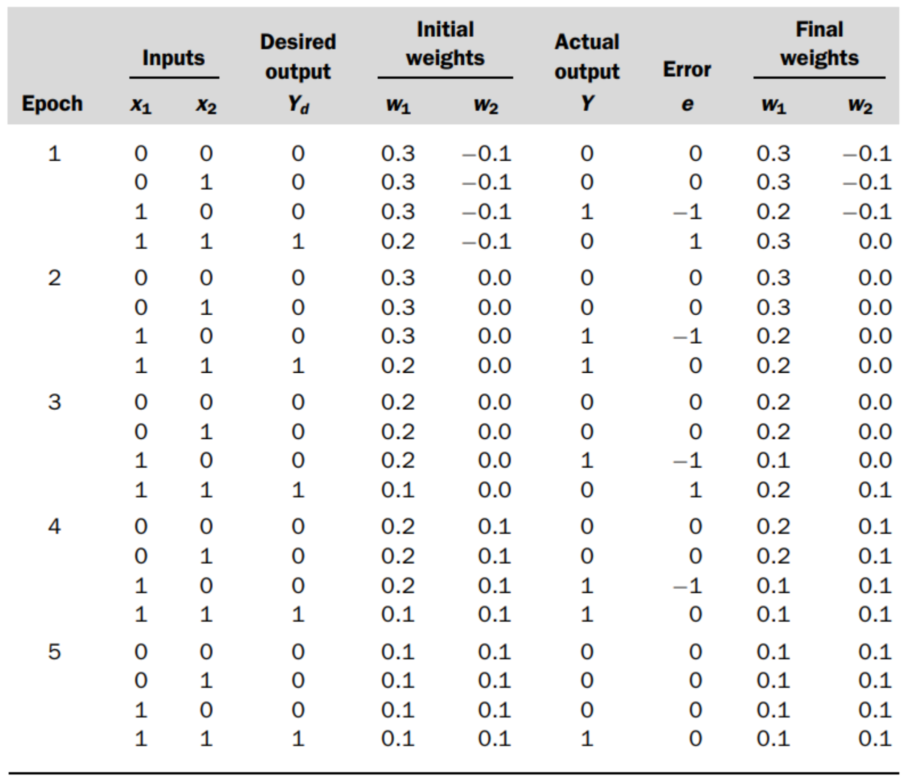


Figure 6.13 (a) Decision boundary constructed by hidden neuron 3 of the network in Figure 6.12; (b) decision boundary constructed by hidden neuron 4; (c) decision boundaries constructed by the complete three-layer network

Source: Negnevitsky, M. (2005). *Artificial intelligence: a guide to intelligent systems*. Pearson education. (Chapter 6: Artificial Neural Networks)

Below, we have an example of how the perception (step function) is trained for the AND gate. Note that the perception cannot be trained for the XOR gate.

Table 6.3 Example of perceptron learning: the logical operation AND



	Inputs		Desired output	Initial weights		Actual output	Error	Final weights	
Epoch	x_1	x_2	Y_d	w_1	w_2	Y	e	w_1	w_2
1	0	0	0	0.3	−0.1	0	0	0.3	−0.1
	0	1	0	0.3	−0.1	0	0	0.3	−0.1
	1	0	0	0.3	−0.1	1	−1	0.2	−0.1
	1	1	1	0.2	−0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	−1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	−1	0.1	0.0
	1	1	1	0.1	0.0	0	1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	−1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1

Threshold: $\theta = 0.2$; learning rate: $\alpha = 0.1$.

Source: Negnevitsky, M. (2005). *Artificial intelligence: a guide to intelligent systems*. Pearson education. (Chapter 6: Artificial Neural Networks)

The proof of convergence for the perceptron learning rule is given here.

References

1. Negnevitsky, M. (2005). *Artificial intelligence: a guide to intelligent systems*. Pearson education. (Chapter 6: Artificial Neural Networks)
2. Michael Collins, "Convergence Proof for the Perceptron Algorithm", http://www.cs.columbia.edu/~mcollins/courses/6998-2012/notes/perc.converge.pdf

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.

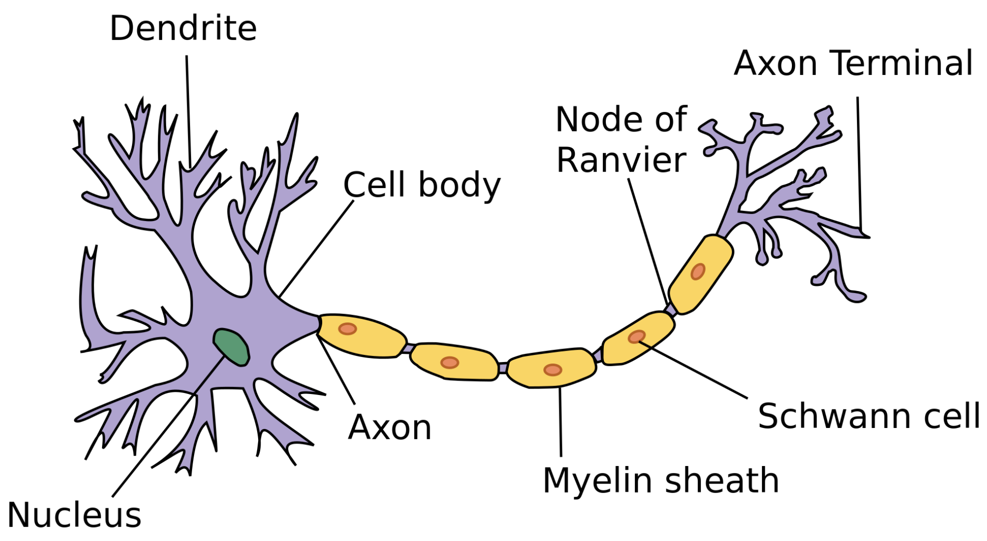


Figure: Neuron connection. Adapted from "Neuron" by Wikipedia, 2020. Retrieved from https://simple.wikipedia.org/wiki/Neuron.

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.

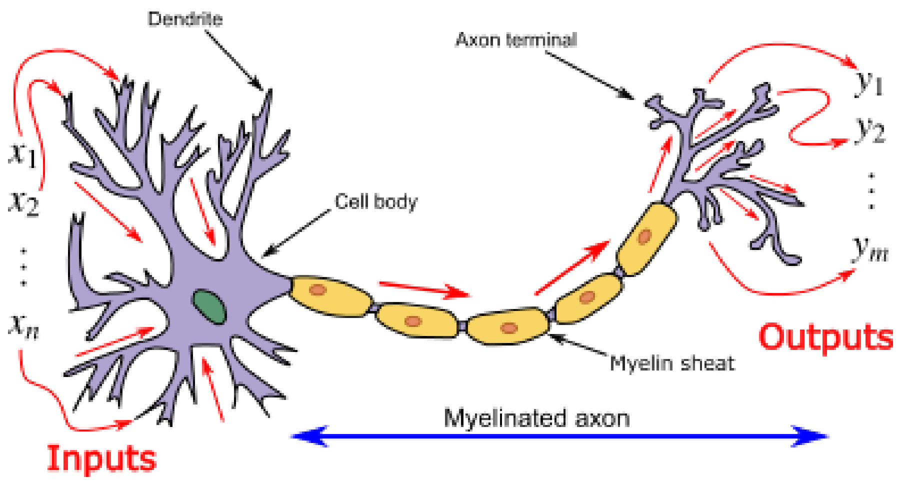


Figure: Neuron and myelinated axon, with the signal flow from inputs at dendrites to outputs at axon terminals. Adapted from "Artificial neural network" by Wikipedia, 2020. Retrieved from https://en.wikipedia.org/wiki/Artificial_neural_network.

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.

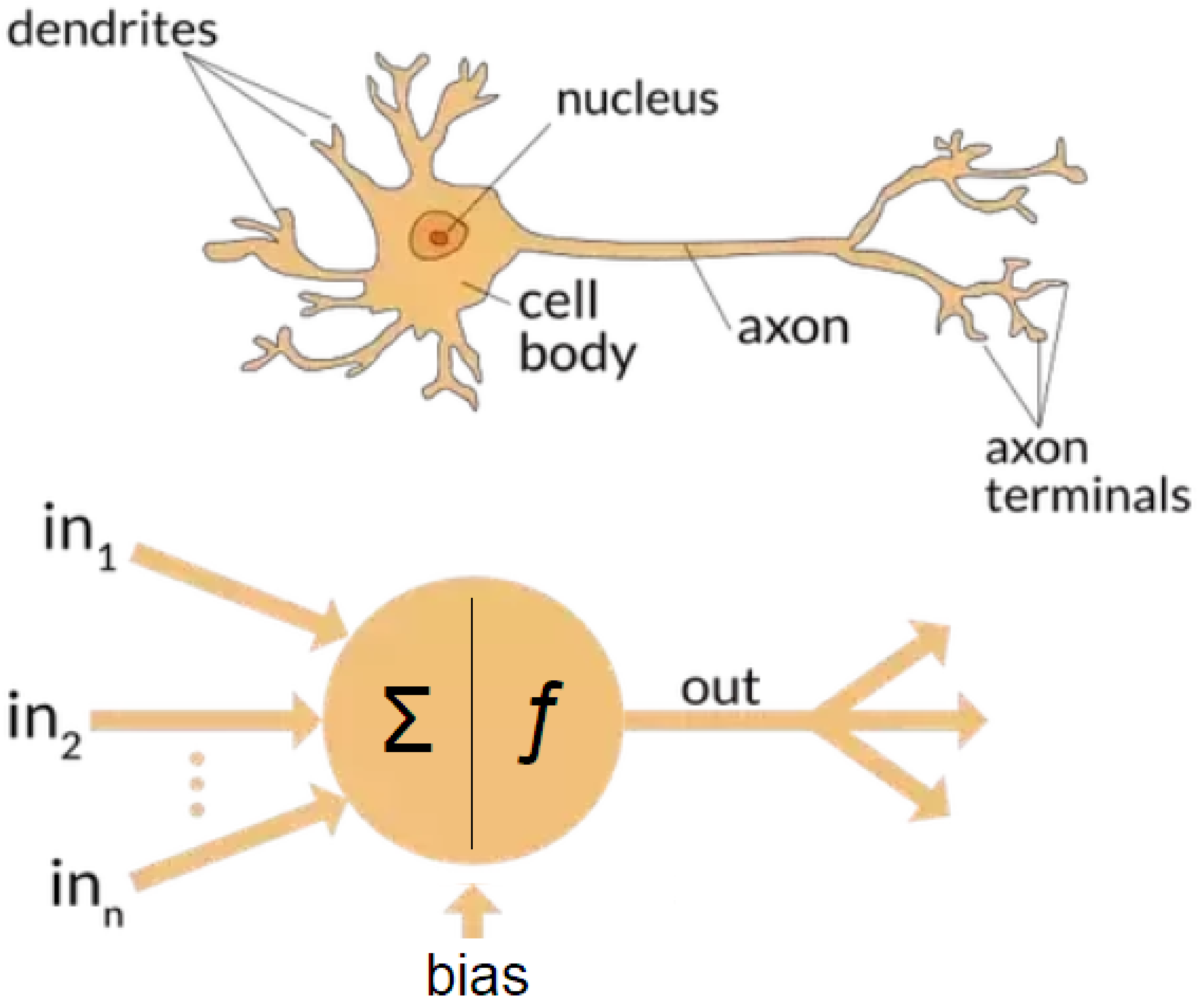


Figure: A biological and an artificial neural with inputs $(in_1, in_2, ...in_n)$. Adapted from "The differences between Artificial and Biological Neural Networks" by R. Nagfyi, 2018. Retrieved from https://towardsdatascience.com/the-differences-between-artificial-and-biological-neural-networks-a8b46db828b7.

The figure above shows inputs $(in_1, in_2, ...in_n)$ 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

Learn about neural network 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.

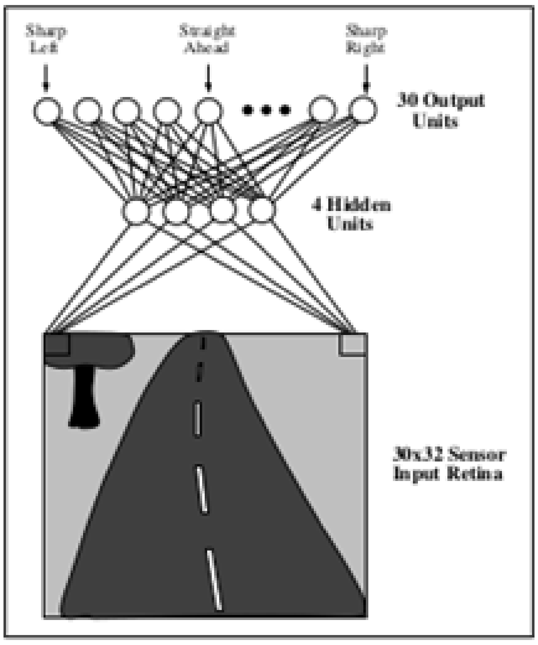
Let's learn some details of one of the earliest attempts to use neural networks for an autonomous driving system. Essentially an input retina sensor (a special camera) is used for input where the video is unpacked into images. This means it is transformed into greyscale at a lower resolution so that 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). Alvinn: 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-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf.

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

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



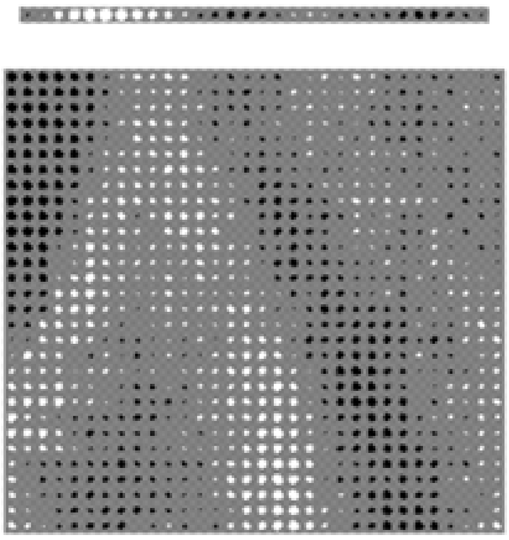


Figure: Neural network to simulate Alvinn driving at speed of 70 mph. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

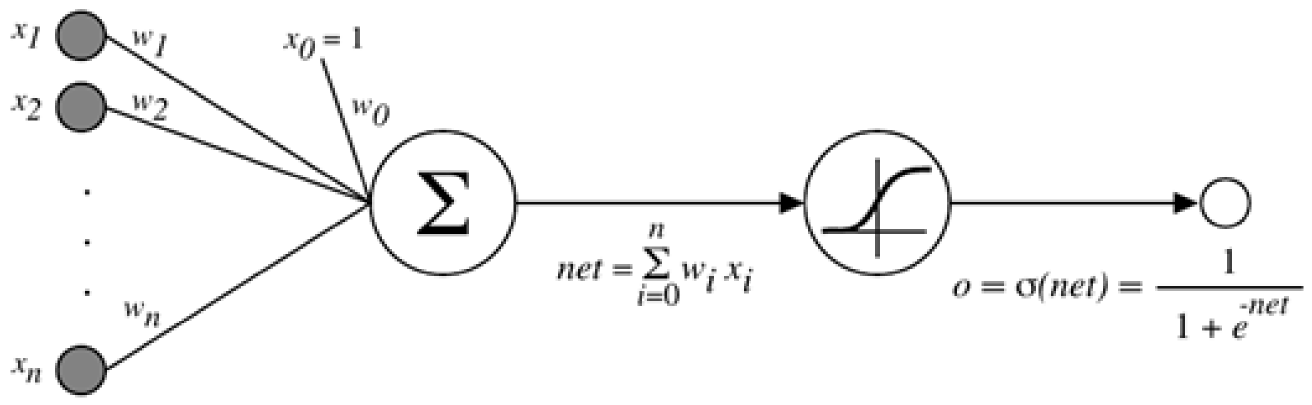
What are some other applications of neural networks and deep learning? There are many examples from healthcare, entertainment, robotics, advertising, and earthquake prediction.

Introduction to a feedforward neural network: Multilayer perceptron

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).



x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

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

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.

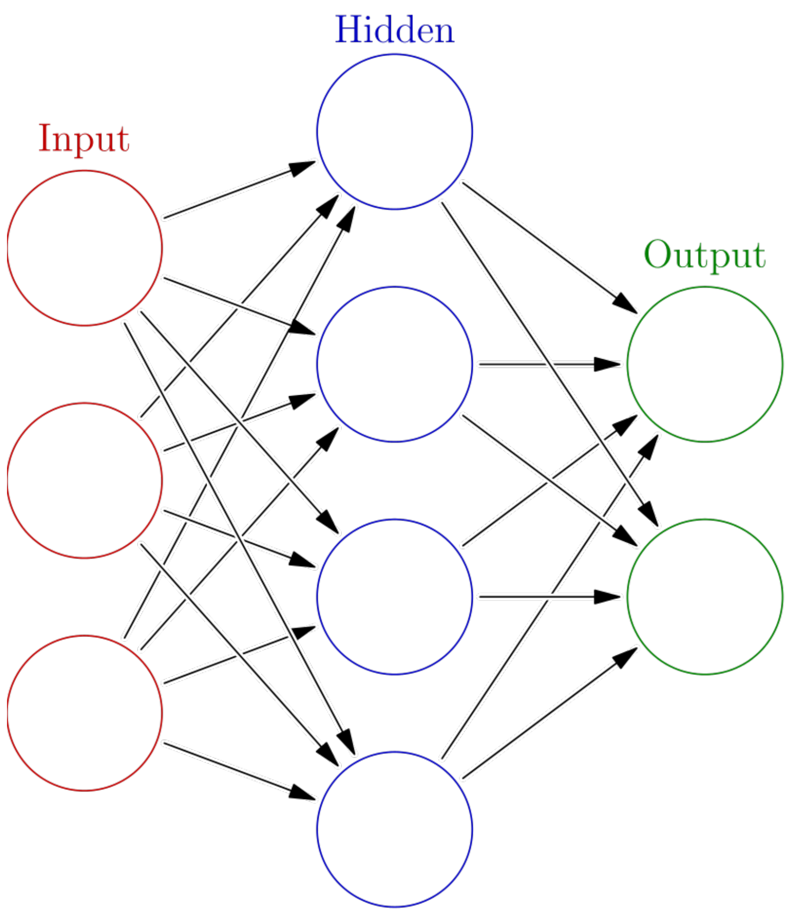


Figure: Neuron connection, Adapted from "Artificial neural network" by Wikipedia, 2020. Retrieved from https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Colored_neural_network.svg.

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 code 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).

```
# Topo = [3,4, 2]

import matplotlib.pyplot as plt
import numpy as np
import random
import time

class Network:

    def __init__(self, Topo, Train, Test, MaxTime, Samples, MinPer, learnRate):
                self.Top  = Topo  # NN topology [input, hidden, output]
                self.Max = MaxTime # max epocs
                self.TrainData = Train
                self.TestData = Test
                self.NumSamples = Samples

                self.learn_rate  = learnRate


                self.minPerf = MinPer

                #initialize weights ( W1 W2 ) and bias ( b1 b2 ) of the network
                np.random.seed()
                self.W1 = np.random.uniform(-0.5, 0.5, (self.Top[0] , self.Top[1]))
                #print(self.W1,  ' self.W1')
                self.B1 = np.random.uniform(-0.5,0.5, (1, self.Top[1])  ) # bias first layer
                #print(self.B1, ' self.B1')
                self.BestB1 = self.B1
                self.BestW1 = self.W1
                self.W2 = np.random.uniform(-0.5, 0.5, (self.Top[1] , self.Top[2]))
                self.B2 = np.random.uniform(-0.5,0.5, (1,self.Top[2]))  # bias second layer
                self.BestB2 = self.B2
                self.BestW2 = self.W2
                self.hidout = np.zeros(self.Top[1] ) # output of first hidden layer
                self.out = np.zeros(self.Top[2]) #  output last layer

                self.hid_delta = np.zeros(self.Top[1] ) # output of first hidden layer
                self.out_delta = np.zeros(self.Top[2]) #  output last layer
```

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.

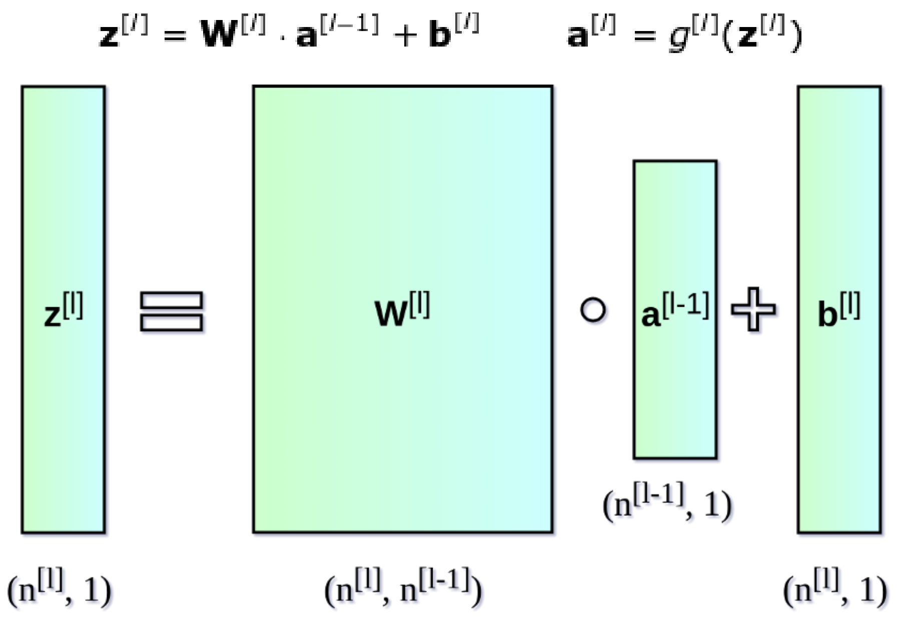


Figure: Forward propagation. Adapted from "Deep Dive into Math Behind Deep Networks" by P. Skalski, 2019. Retrieved from https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba.

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.

```
#FNN: Version Two

    def ForwardPass(self, X ):
                z1 = X.dot(self.W1) - self.B1
                self.hidout = self.sigmoid(z1) # output of first hidden layer
                z2 = self.hidout.dot(self.W2)  - self.B2
                self.out = self.sigmoid(z2)  # output second hidden layer
```

```
#FNN: Version One


    def ForwardPass_Simple(self, input_vec ):  # Alternative implementation of ForwardPass(self, X
                layer = 0 # input to hidden layer
                weightsum_first = 0

                # [3,4,2]

                for y in range(0, self.Top[layer+1]): # 4  - y  0, 1, 2, 3
                        for x in range(0, self.Top[layer]): # 3 - x 0, 1, 2
                                weightsum_first   +=   input_vec[x] * self.W1[x,y]
                        self.hidout[y] = self.sigmoid(weightsum_first - self.B1[y])
                        weightsum_first  = 0

                layer = 1 #   hidden layer to output
                weightsum_second = 0 # output of second layer (class outputs)
                for y in range(0, self.Top[layer+1]): #2 - y 0, 1
                        for x in range(0, self.Top[layer]): # 4 - x 0, 1, 2, 3
                                weightsum_second  +=   self.hidout[x] * self.W2[x,y]
                        self.out[y] = self.sigmoid(weightsum_second - self.B2[y])
                        weightsum_second = 0
```

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.

```
    def sampleEr(self,actualout):
            error = np.subtract(self.out, actualout)
            sqerror= np.sum(np.square(error))/self.Top[2]

            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.

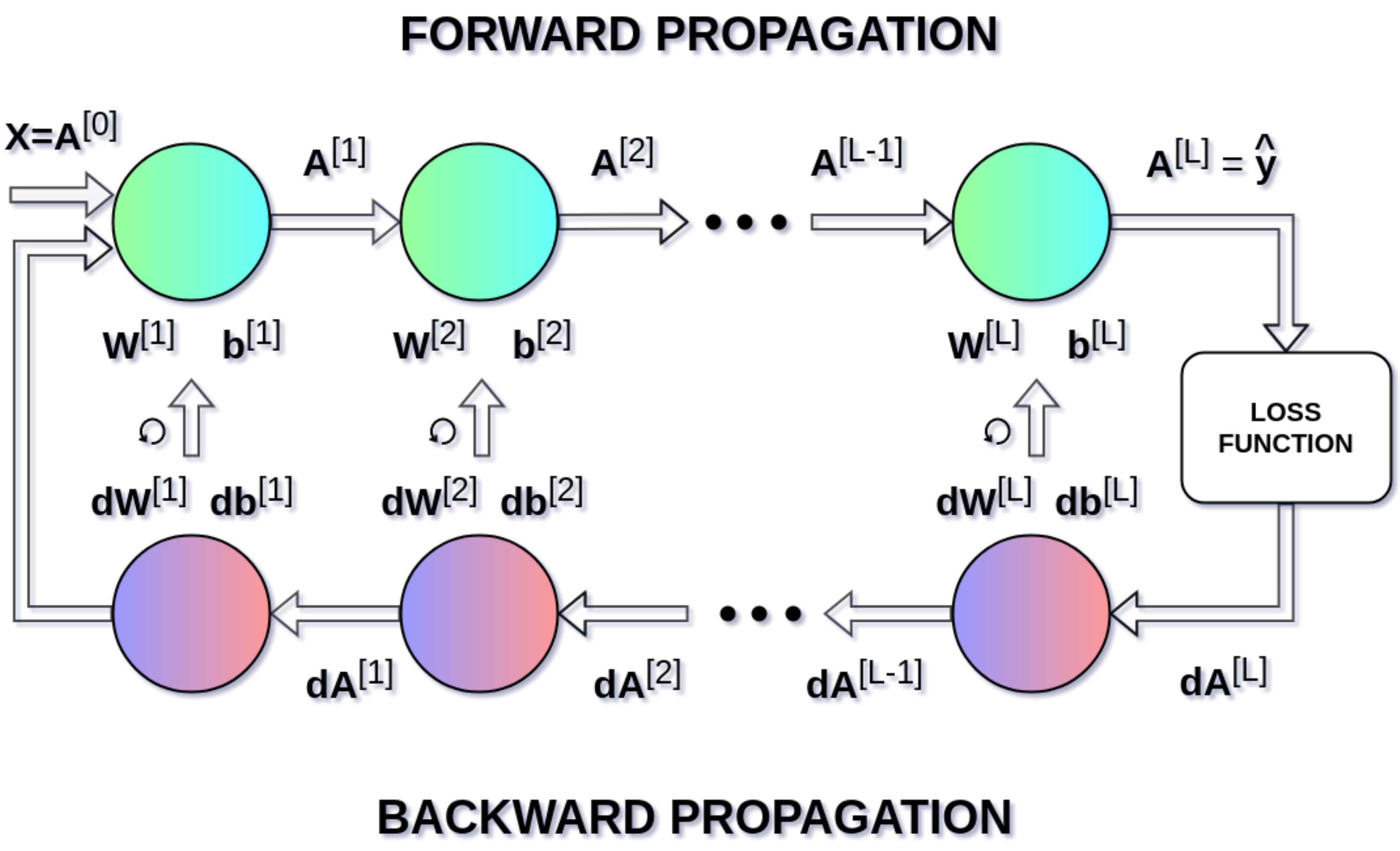


Figure: Feedforward and backward propagation. Adapted from "Deep Dive into Math Behind Deep Networks" by P. Skalski, 2019. Retrieved from https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba.

Now lets revisit the XOR gate problem:

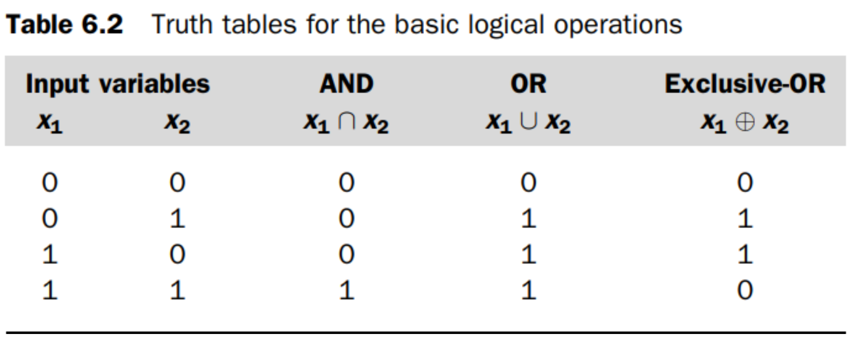


Table 6.2 Truth tables for the basic logical operations

Input variables		AND	OR	Exclusive-OR
x_1	x_2	$x_1 \cap x_2$	$x_1 \cup x_2$	$x_1 \oplus x_2$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

and create a simple neural network (with one hidden layer) for the XOR gate problem.

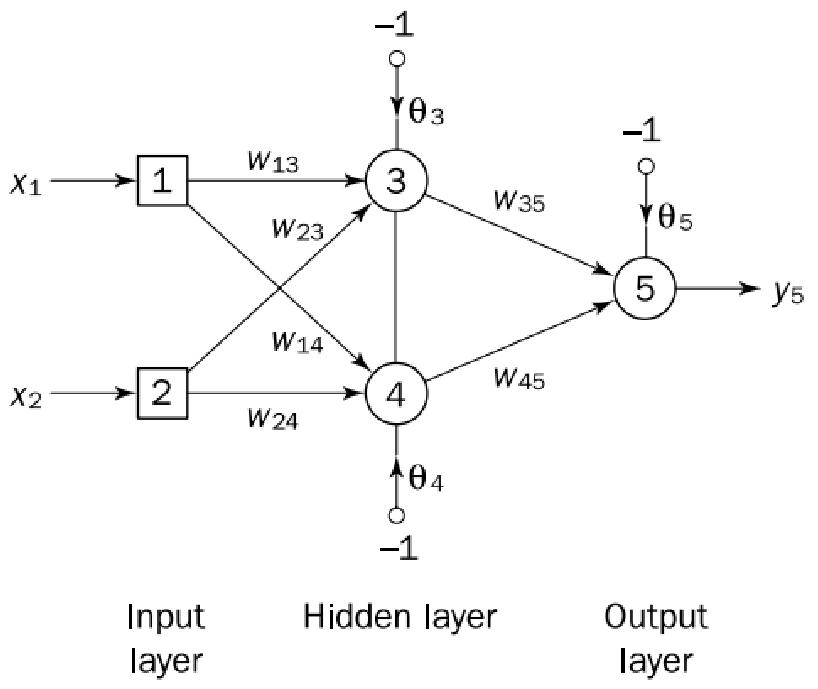


Figure 6.10 Three-layer network for solving the Exclusive-OR operation

Source: Negnevitsky, M. (2005). *Artificial intelligence: a guide to intelligent systems*. Pearson education. (Chapter 6: Artificial Neural Networks)

Let's place some values of weights, to begin with:

$w_{13} = 0.5$, $w_{14} = 0.9$, $w_{23} = 0.4$, $w_{24} = 1.0$, $w_{35} = -1.2$, $w_{45} = 1.1$,
$\theta_3 = 0.8$, $\theta_4 = -0.1$ and $\theta_5 = 0.3$.

Consider a training set where inputs x_1 and x_2 are equal to 1 and desired output $y_{d,5}$ is 0. The actual outputs of neurons 3 and 4 in the hidden layer are calculated as

$$y_3 = sigmoid\,(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1/[1 + e^{-(1\times 0.5 + 1\times 0.4 - 1\times 0.8)}] = 0.5250$$

$$y_4 = sigmoid\,(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1/[1 + e^{-(1\times 0.9 + 1\times 1.0 + 1\times 0.1)}] = 0.8808$$

Now the actual output of neuron 5 in the output layer is determined as

$$y_5 = sigmoid\,(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1/[1 + e^{-(-0.5250\times 1.2 + 0.8808\times 1.1 - 1\times 0.3)}] = 0.5097$$

Thus, the following error is obtained:

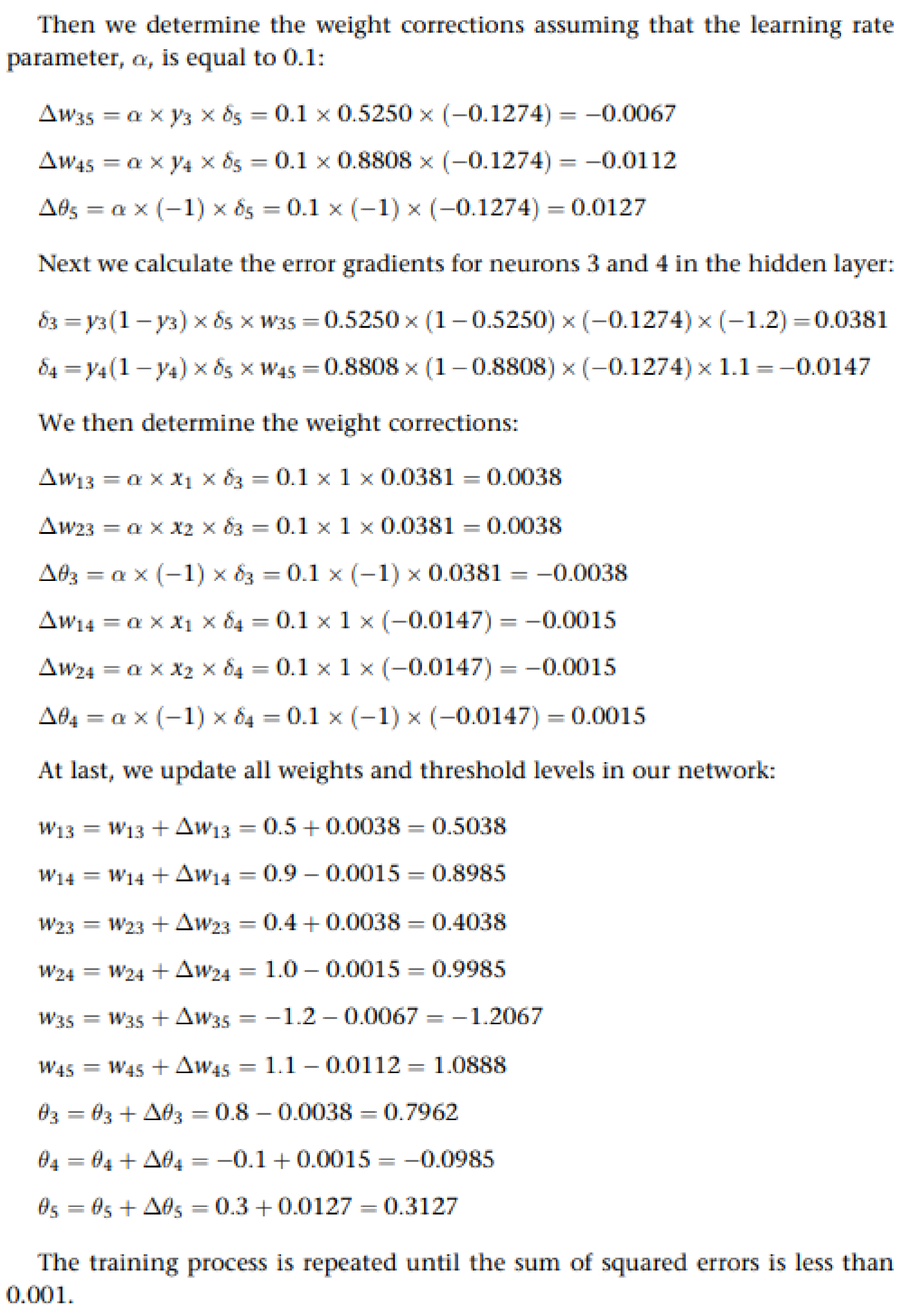
$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

The next step is weight training. To update the weights and threshold levels in our network, we propagate the error, e, from the output layer backward to the input layer.

First, we calculate the error gradient for neuron 5 in the output layer:

$$\delta_5 = y_5(1 - y_5)e = 0.5097 \times (1 - 0.5097) \times (-0.5097) = -0.1274$$

Source: Negnevitsky, M. (2005). *Artificial intelligence: a guide to intelligent systems*. Pearson education. (Chapter



Then we determine the weight corrections assuming that the learning rate parameter, α, is equal to 0.1:

$$\Delta w_{35} = \alpha \times y_3 \times \delta_5 = 0.1 \times 0.5250 \times (-0.1274) = -0.0067$$

$$\Delta w_{45} = \alpha \times y_4 \times \delta_5 = 0.1 \times 0.8808 \times (-0.1274) = -0.0112$$

$$\Delta \theta_5 = \alpha \times (-1) \times \delta_5 = 0.1 \times (-1) \times (-0.1274) = 0.0127$$

Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:

$$\delta_3 = y_3(1-y_3) \times \delta_5 \times w_{35} = 0.5250 \times (1-0.5250) \times (-0.1274) \times (-1.2) = 0.0381$$

$$\delta_4 = y_4(1-y_4) \times \delta_5 \times w_{45} = 0.8808 \times (1-0.8808) \times (-0.1274) \times 1.1 = -0.0147$$

We then determine the weight corrections:

$$\Delta w_{13} = \alpha \times x_1 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta w_{23} = \alpha \times x_2 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta \theta_3 = \alpha \times (-1) \times \delta_3 = 0.1 \times (-1) \times 0.0381 = -0.0038$$

$$\Delta w_{14} = \alpha \times x_1 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta w_{24} = \alpha \times x_2 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta \theta_4 = \alpha \times (-1) \times \delta_4 = 0.1 \times (-1) \times (-0.0147) = 0.0015$$

At last, we update all weights and threshold levels in our network:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$

$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$

$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$

$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$

$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$

$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$

$$\theta_3 = \theta_3 + \Delta \theta_3 = 0.8 - 0.0038 = 0.7962$$

$$\theta_4 = \theta_4 + \Delta \theta_4 = -0.1 + 0.0015 = -0.0985$$

$$\theta_5 = \theta_5 + \Delta \theta_5 = 0.3 + 0.0127 = 0.3127$$

The training process is repeated until the sum of squared errors is less than 0.001.

Code below tries to replicate the XOR gate neural network for a single epoch.

```
import numpy as np

np.random.seed()

Top = [2,2,1] # NN topology for XOR gate problem
# lets set the weights and biases
#W1 = np.random.uniform(-0.5, 0.5, (Top[0] , Top[1]))
W1 = np.array([[0.5,0.9 ], [0.4,1.0]]) # overwritten
#B1 = np.random.uniform(-0.5,0.5, (1, Top[1])  ) # bias first layer
B1 = np.array([[0.8,-0.1]]) # overwritten bias of hidden layer

#W2 = np.random.uniform(-0.5, 0.5, (Top[1] , Top[2]))
W2 = np.array([[-1.2],[1.1]]) # overwritten
#B2 = np.random.uniform(-0.5,0.5, (1,Top[2]))
B2 = np.array([[0.3]]) # overwritten bias of hidden layer

print(W1,  ' W1')
print(B1, ' B1')
print(W2,  ' W2')
print(B2, ' B2')


def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def ForwardPass(X):
    z1 = X.dot(W1) - B1
    print(z1, ' z1')
    hidout = sigmoid(z1) # output of first hidden layer
    print(hidout, ' hidout')
    z2 = hidout.dot(W2)  - B2
    print(z2, ' z2')
    out = sigmoid(z2)  # output second hidden layer
    print(out, ' out')
    return out, hidout

def BackwardPass(input_vec, learn_rate, prediction, hidout, desired):
    out = prediction
    out_delta =   (desired - out)*(out*(1-out))
    print(out_delta, ' is gradient at output')
    #hid_delta = out_delta.dot(W2) * (hidout * (1-hidout))

    #W2+= hidout.T.dot(out_delta) * learn_rate
    #B2+=  (-1 * learn_rate * out_delta)
    #W1 += (input_vec.T.dot(hid_delta) * learn_rate)
    #B1+=  (-1 * learn_rate * hid_delta)

#main


training_inputs = []
training_inputs.append(np.array([1, 1]))
training_inputs.append(np.array([1, 0]))
training_inputs.append(np.array([0, 1]))
training_inputs.append(np.array([0, 0]))

# set the dataset class labels
labels = np.array([0, 1, 1, 0])

print(training_inputs, ' list of training features')

learn_rate = 0.1

max_epochs = 1

for epochs in range(max_epochs):
    for X, y in zip(training_inputs, labels):
        print(X,y, ' instance')
        prediction, hidout = ForwardPass(X)
        BackwardPass(X, learn_rate, prediction, hidout, y)
```

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.

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

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

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

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

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

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

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.

```
# FNN: Version One
# self.Top = [3,4,2]
    def BackwardPass_Simple(self, input_vec, desired ):  # Alternative implementation of BackwardPa

                # compute gradients for each layer (output and hidden layer)

                layer = 2 #output layer
                for x in range(0, self.Top[layer]):
                        self.out_delta[x] =  (desired[x] - self.out[x])*(self.out[x]*(1-self.out[x]

                layer = 1 # hidden layer
                temp = 0
                for x in range(0, self.Top[layer]):
                        for y in range(0, self.Top[layer+1]):
                                temp += ( self.out_delta[y] * self.W2[x,y]);
                        self.hid_delta[x] =  (self.hidout[x] * (1 - self.hidout[x])) * temp
                        temp = 0

                                # update weights and bias
                layer = 1 # hidden to output

                for x in range(0, self.Top[layer]):
                        for y in range(0, self.Top[layer+1]):
                                self.W2[x,y] += self.learn_rate * self.out_delta[y] * self.hidout[x
                        #print self.W2
                for y in range(0, self.Top[layer+1]):
                        self.B2[y] += -1 * self.learn_rate * self.out_delta[y]

                layer = 0 # Input to Hidden

                for x in range(0, self.Top[layer]):
                        for y in range(0, self.Top[layer+1]):
                                self.W1[x,y] += self.learn_rate * self.hid_delta[y] * input_vec[x]

                for y in range(0, self.Top[layer+1]):
                        self.B1[y] += -1 * self.learn_rate * self.hid_delta[y]
```

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.

```
#FNN Version Two



    def BackwardPass(self, input_vec, desired):
                out_delta =   (desired - self.out)*(self.out*(1-self.out))
                hid_delta = out_delta.dot(self.W2.T) * (self.hidout * (1-self.hidout)) #https://www

                self.W2+= self.hidout.T.dot(out_delta) * self.learn_rate
                self.B2+=  (-1 * self.learn_rate * out_delta)

                self.W1 += (input_vec.T.dot(hid_delta) * self.learn_rate)
                self.B1+=  (-1 * self.learn_rate * hid_delta)
```

Next we revisit the example for the XOR gate problem and compute the gradients and weight update:

Source: Negnevitsky, M. (2005). *Artificial intelligence: a guide to intelligent systems*. Pearson education. (Chapter 6: Artificial Neural Networks)

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

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

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

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

where

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

```
#FNN: Version Two

    def BP_GD(self):



                Input = np.zeros((1, self.Top[0])) # temp hold input
                Desired = np.zeros((1, self.Top[2]))



                Er = []
                epoch = 0
                bestmse = 10000 # assign a large number in begining to maintain best (lowest RMSE)
                bestTrain = 0
                while  epoch < self.Max and bestTrain < self.minPerf :
                        sse = 0
                        for s in range(0, self.NumSamples):

                                Input[:]  =  self.TrainData[s,0:self.Top[0]]
                                Desired[:]  = self.TrainData[s,self.Top[0]:]

                                self.ForwardPass(Input)
                                self.BackwardPass(Input ,Desired)
                                sse = sse+ self.sampleEr(Desired)

                        mse = np.sqrt(sse/self.NumSamples*self.Top[2])

                        if mse < bestmse:
                                 bestmse = mse
                                 self.saveKnowledge()
                                 (x,bestTrain) = self.TestNetwork(self.TrainData, self.NumSamples,

                        Er = np.append(Er, mse)

                        epoch=epoch+1

                return (Er,bestmse, bestTrain, epoch)
```

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. Below see how the error changes over time (epochs) for the XOR gate problem.

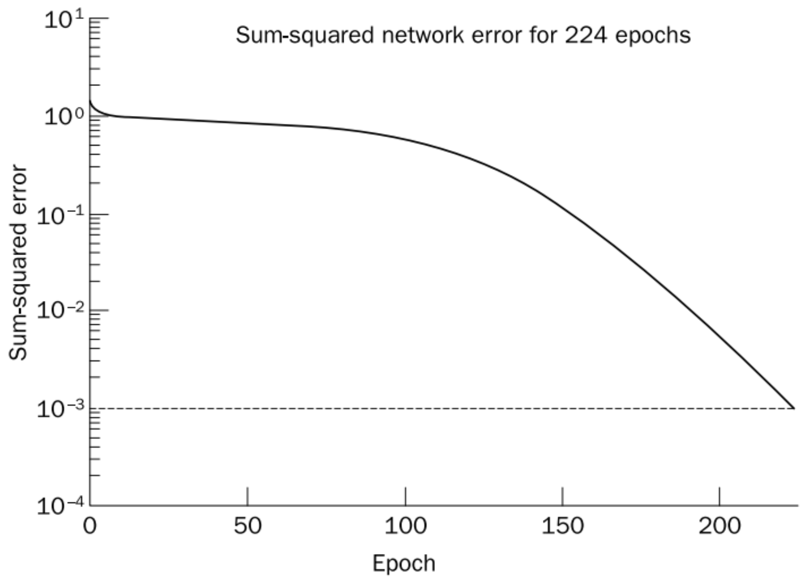


Figure 6.11 Learning curve for operation Exclusive-OR

Source: Negnevitsky, M. (2005). *Artificial intelligence: a guide to intelligent systems*. Pearson education. (Chapter 6: Artificial Neural Networks)

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.

```
def TestNetwork(self, Data, testSize, erTolerance):
            Input = np.zeros((1, self.Top[0])) # temp hold input
            Desired = np.zeros((1, self.Top[2]))
            nOutput = np.zeros((1, self.Top[2]))
            clasPerf = 0
            sse = 0
            self.W1 = self.BestW1
            self.W2 = self.BestW2 #load best knowledge
            self.B1 = self.BestB1
            self.B2 = self.BestB2 #load best knowledge

            for s in range(0, testSize):

                    Input[:]   =   Data[s,0:self.Top[0]]
                    Desired[:] =  Data[s,self.Top[0]:]
```

```
                        self.ForwardPass(Input )
                        sse = sse+ self.sampleEr(Desired)

                        if(np.isclose(self.out, Desired, atol=erTolerance).any()):
                                clasPerf =  clasPerf +1

                return ( sse/testSize, float(clasPerf)/testSize * 100 )
```

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.

References

- Negnevitsky, M. (2005). *Artificial intelligence: a guide to intelligent systems*. Pearson education. (Chapter 6: Artificial Neural Networks)
- Mitchell, T. (1997). Machine learning. Prentice Hall.
- Friedman, J. H. (2017). *The elements of statistical learning: Data mining, inference, and prediction*. springer open. https://web.stanford.edu/~hastie/Papers/ESLII.pdf (open book)

FNN: Version one

Practise the code on a feedforward neural network.

FNN: Version one shows a non-vectorised version of the forward pass that uses nested for loops rather than dot products for computing weighted sum and calculating the gradients in the backward pass. Note that sigmoid units are used in hidden and output layers. Sample training problems are given so you can test the different problems and note both performances for different values of the learning rate and also the effect of choosing a different number of hidden neurons.

You need to save your results so you can test with FNN: Version two in the next lesson.

Presently the code cannot handle more than one hidden layer. You can try implementing it but that will take a lot of time and it is not a focus of the course.

FNN: Version two

Practise the code on a feedforward neural network.

FNN: Version two shows a vectorised version of the forward pass that uses dot products for computing weighted sum and calculating the gradients in the backward pass. Note that sigmoid units used in hidden layers and output. Sample training problems are given so you can test the different problems, note performance for different values of the learning rate, and test the effect of a different number of hidden neurons.

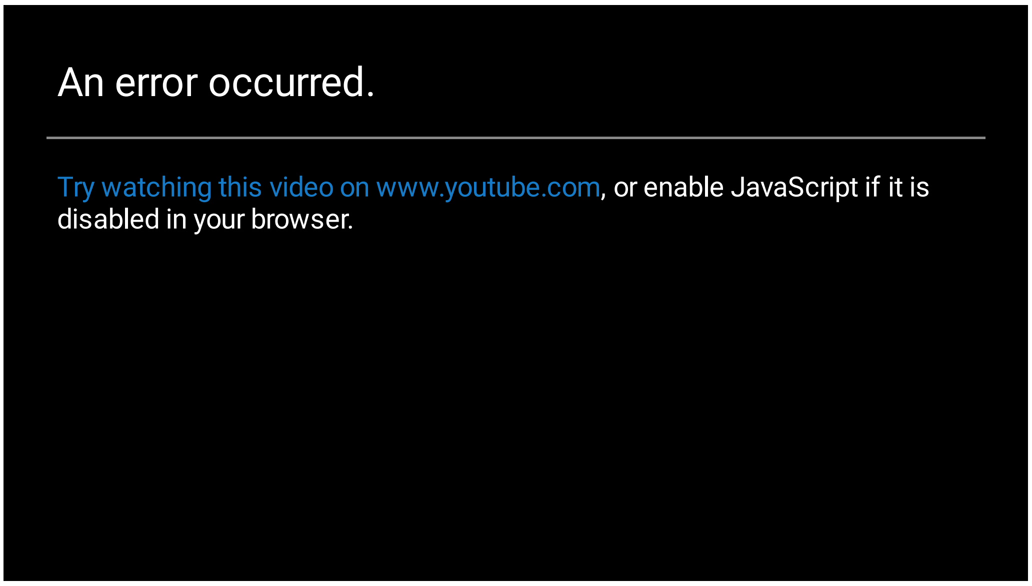
You need to save your results so you can test with FNN: Version one in the previous lesson. Consider: What are the similarities and differences? Have you done multiple experimental runs to compare mean and standard training and classification performance?

Presently the code cannot handle more than one hidden layer, but you can try implementing it but that could take a lot of time which is not a focus of the course, but the tutorial below video titled FNN: Version one discusses how that can be done.

Training or learning?

But what is a neural network?

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


An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

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?

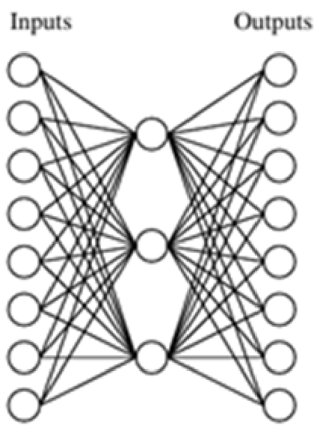
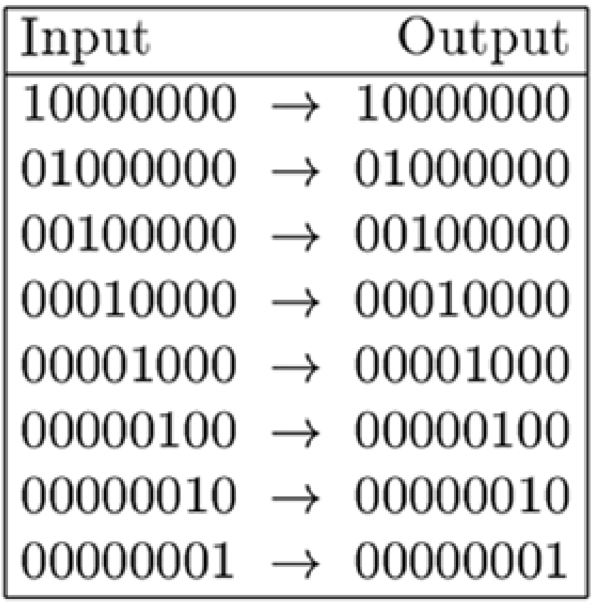


Figure: A multi-layer network. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K:

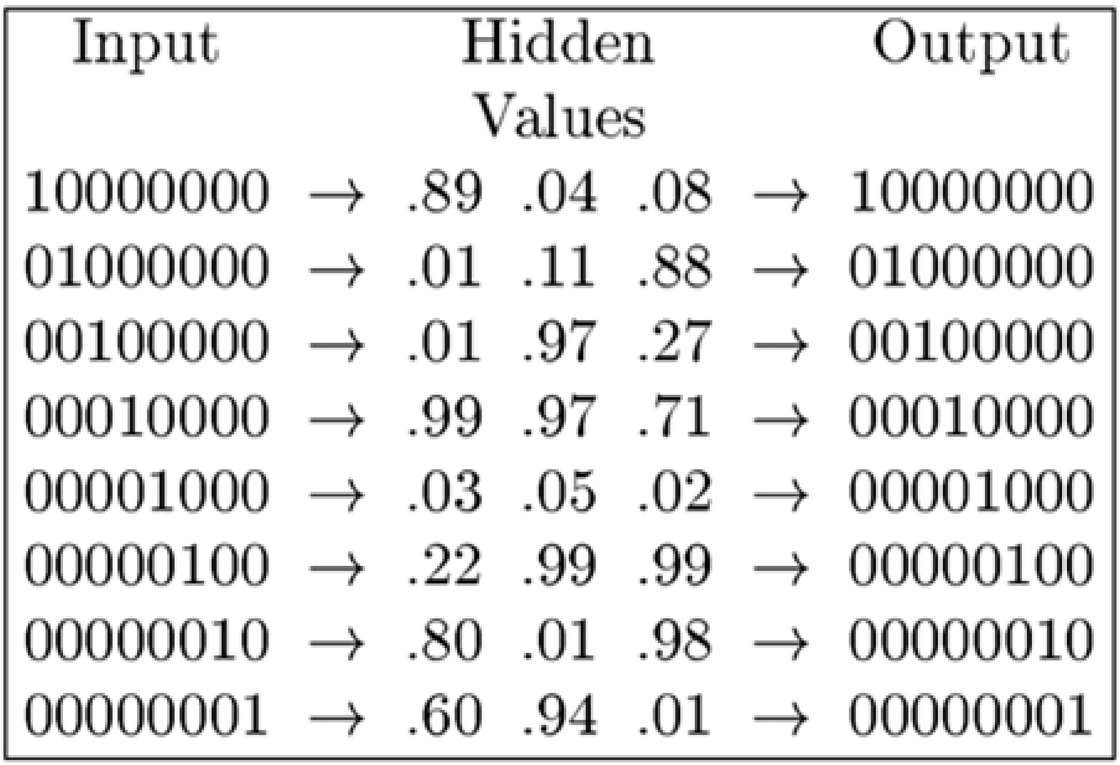
McGraw Hill.



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

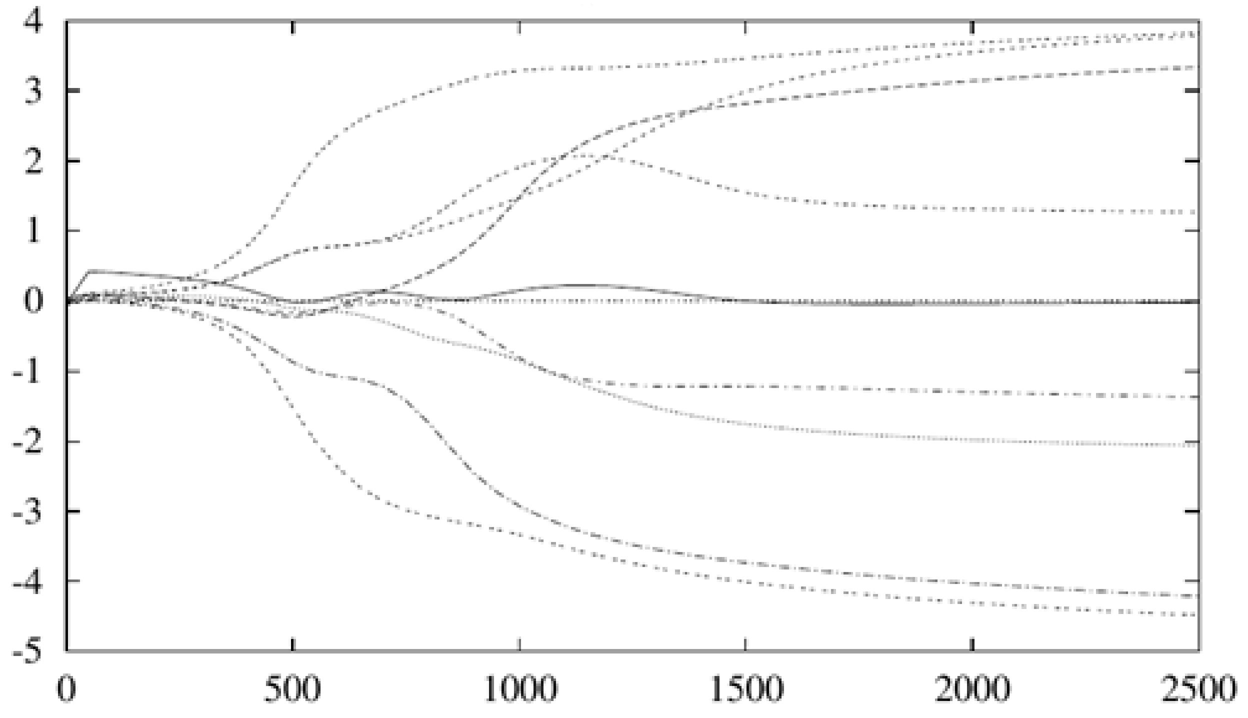


Figure: Weights from inputs to one hidden unit. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

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.

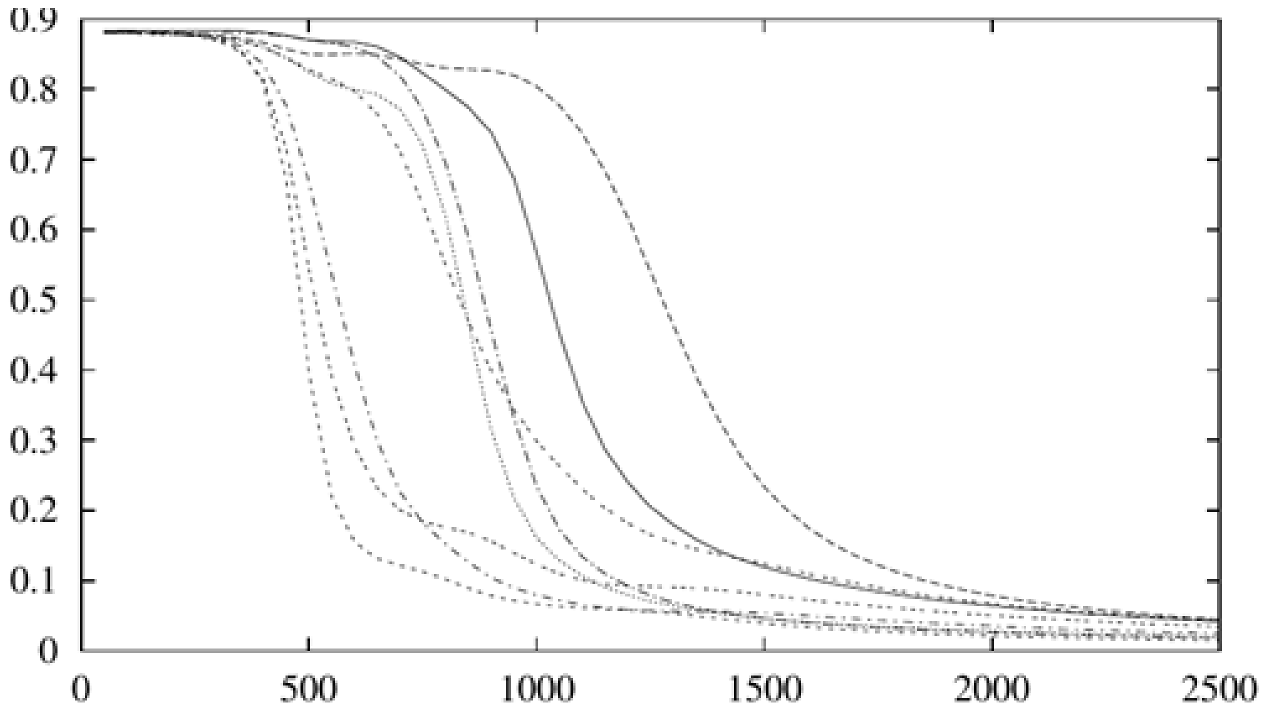


Figure: Sum of squared errors for each output unit. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

Try and implement the above with 1. scikit learn 2. FNN version one, 3. FNN version two as a

Neural networks in Sklearn

Now that you have learned the basics of neural networks by covering fundamental issues such as gradient descent, weight update, and training procedure, it is good to know which approaches are being used most commonly in the research sector and industry.

The code that you worked on earlier was an implementation by R. Chandra which has been well-tested in several research projects but it has some limitations such as extending the number of hidden layers. It is important to learn existing libraries so you can test other functionalities. There are several different variations of multilayer perceptron in terms of different gradient-based algorithms and topologies.

Scikit-learn

Scikit-learn is a machine learning library based on Python that features neural networks and associated training algorithms. It has been very popular in the industry, and it's good to learn to apply it to the problems studied previously and also to some new problems.

The below code demonstrates the use of scikit-learn for a multilayer perceptron. Note how the scikit-learn library is utilised.

```
#Source: https://www.python-course.eu/neural_networks_with_scikit.php

from sklearn.datasets import load_iris
iris = load_iris()
# splitting into train and test datasets
from sklearn.model_selection import train_test_split
datasets = train_test_split(iris.data, iris.target,test_size=0.2)

train_data, test_data, train_labels, test_labels = datasets
# scaling the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# we fit the train data
scaler.fit(train_data)
# scaling the train data
train_data = scaler.transform(train_data)
test_data = scaler.transform(test_data)
print(train_data[:3])
# Training the Model
from sklearn.neural_network import MLPClassifier
# creating an classifier from the model:
mlp = MLPClassifier(hidden_layer_sizes=(10, 5), max_iter=1000)
# let's fit the training data to our model
mlp.fit(train_data, train_labels)
```

```
from sklearn.metrics import accuracy_score

predictions_train = mlp.predict(train_data)
print(accuracy_score(predictions_train, train_labels))
predictions_test = mlp.predict(test_data)
print(accuracy_score(predictions_test, test_labels))

from sklearn.metrics import confusion_matrix
cm_train = confusion_matrix(predictions_train, train_labels)
cm_test = confusion_matrix(predictions_test, test_labels)
print(cm_train, 'cm train')
print(cm_train, 'cm test')


from sklearn.metrics import classification_report
print(classification_report(predictions_test, test_labels))
```

Further reading:

1. Supervised neural networks

2. Scikit-learn for machine learning

3. Machine learning-Diabetes

Exercise 3.1

Python Challenge

- Try training the FNN model with two selected classification datasets from the UCI machine learning repository. Evaluate the effect of number of hidden neurons, and learning rate [0.1, 0.2 ...1]. Try 10 experiments with different random state in data split (60/40 train/test) or different initial weights for each run and report the mean and standard deviation for each experiment and plot your results.

Optional (this will require more time and not part of the course assessment)

- Try to add another hidden layer and update the forward and backward pass.
- Can you generalise to any user-selected number of hidden layers? Discuss how you will do this and implement it if possible.

R Challenge

- Use either Caret or Keras in R. Try training the FNN model with two selected classification datasets from the UCI machine learning repository. Evaluate the effect of the number of hidden neurons, and learning rate [0.1, 0.2 ...1]. Try 10 experiments with different initial positions in weight space for each case and report mean and standard deviation for each experiment and plot your results. Ref: https://cran.r-project.org/web/packages/caret/vignettes/caret.html

Resources

Example datasets http://archive.ics.uci.edu/ml/datasets

Exercise 3.1 Solution

Python Challenge

- Try training the FNN model with two selected classification datasets from the UCI machine learning repository. Evaluate the effect of number of hidden neurons, and learning rate [0.1, 0.2 ...1]. Try 10 experiments with different random state in data split (60/40 train/test) or different initial weights for each run and report the mean and standard deviation for each experiment and plot your results.

Optional (this will require more time and not part of the course assessment)

- Try to add another hidden later and update the forward and backward pass.
- Can you generalise to any user-selected number of hidden layers? Discuss how you will do this and implement it if possible.

R Challenge

- Use either Caret or Keras in R. Try training the FNN model with two selected classification datasets from the UCI machine learning repository. Evaluate the effect of the number of hidden neurons, and learning rate [0.1, 0.2 ...1]. Try 10 experiments with different initial positions in weight space for each case and report mean and standard deviation for each experiment and plot your results. Ref: https://cran.r-project.org/web/packages/caret/vignettes/caret.html

Resources

Example datasets http://archive.ics.uci.edu/ml/datasets

Activation functions and layers

Learn about activation functions and layers.

We covered the basics of activation function in the previous lessons. Here we cover mode 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.

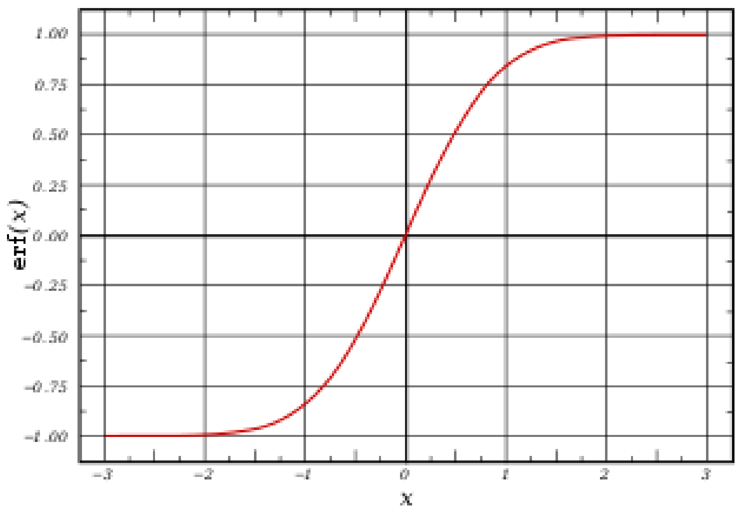


Figure: Sigmoid function w.r.t x

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.

```
#source: https://medium.com/@omkar.nallagoni/activation-functions-with-derivative-and-python-code-s

import matplotlib.pyplot as plt
import numpy as np

def sigmoid(x):
    s=1/(1+np.exp(-x)) # this implements the sigmoid function
    ds=s*(1-s)  # this gives the derivative
    return s,ds
x=np.arange(-6,6,0.01)
sigmoid(x)
# Setup centered axes
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# Create and show plot
ax.plot(x,sigmoid(x)[0], color="#307EC7", linewidth=3, label="sigmoid")
ax.plot(x,sigmoid(x)[1], color="#9621E2", linewidth=3, label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.savefig('sig.png')
```

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.

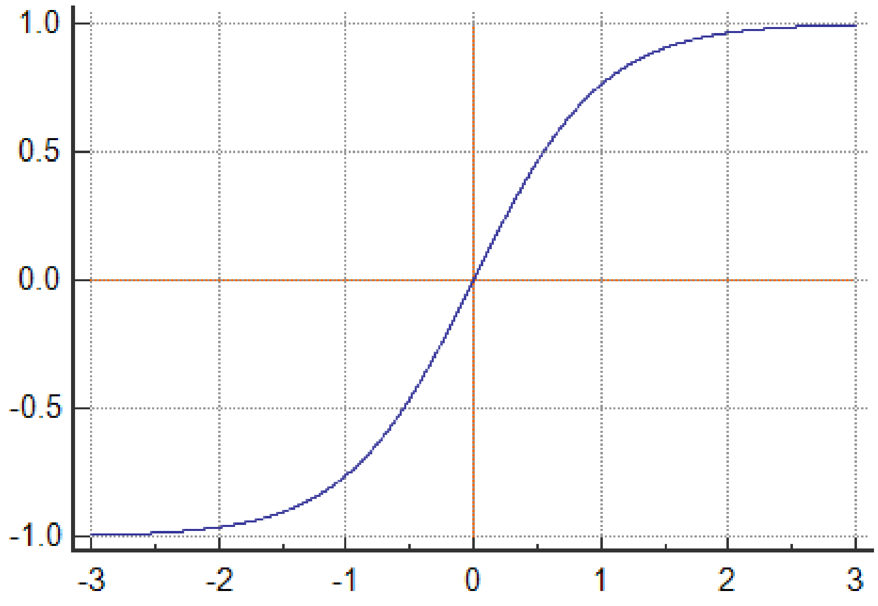


Figure: $tanh$ function w.r.t x

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.

```
# source: https://medium.com/@omkar.nallagoni/activation-functions-with-derivative-and-python-code-
import matplotlib.pyplot as plt
import numpy as np

def tanh(x):
    t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
    dt=1-t**2 # this gives the derivative
    return t,dt
z=np.arange(-4,4,0.01)
tanh(z)[0].size,tanh(z)[1].size
# Setup centered axes
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# Create and show plot
ax.plot(z,tanh(z)[0], color="#307EC7", linewidth=3, label="tanh")
ax.plot(z,tanh(z)[1], color="#9621E2", linewidth=3, label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.savefig('tanh.png')
```

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:

$$f\left(x\right) = x^{+} = \max\left(0, x\right)$$

where x is the input.

Softplus

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

$$f\left(x\right) = \ln\left(1 + e^{x}\right)$$

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

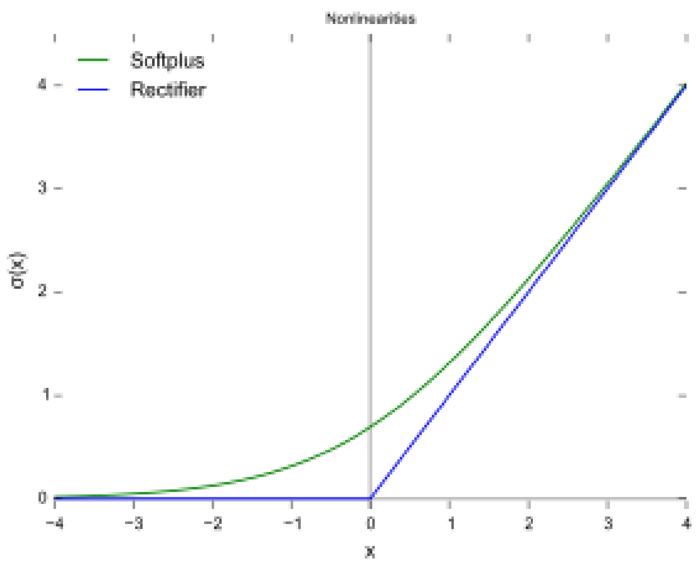


Figure: Relu and Softplus functions w.r.t x

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.

$$\sigma\left(x\right) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_i}}\ for\ i = 1, 2....K\ and\ z\ =\ z_1, z_2...z_K\ \in \Re^K$$

Run the codes.

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

```
import numpy as np
a = [1.0, 2.0, 3.0, 4.0, 1.0, 2.0, 3.0]
print(np.exp(a))
print(np.sum(np.exp(a)))

soft_max = np.exp(a) / np.sum(np.exp(a))
print(soft_max)
```

Run the below code to understand softmax in R.

```
z <- c(1.0, 2.0, 3.0, 4.0, 1.0, 2.0, 3.0)
softmax <- exp(z)/sum(exp(z))
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.

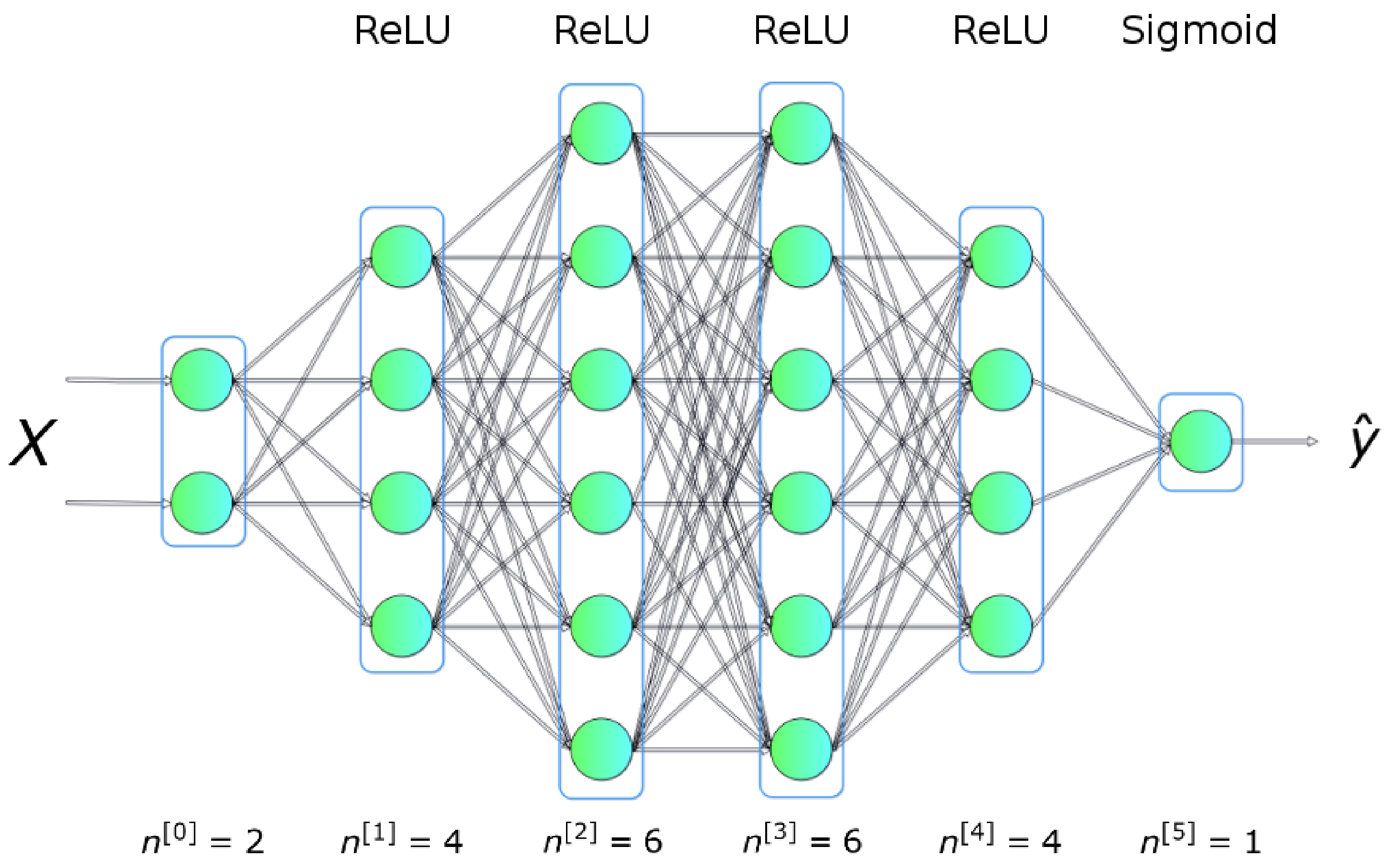


Figure: Sigmoid unit in the output layer used with Relu in 4 hidden layers. Adapted from "Deep Dive into Math Behind Deep Networks" by P. Skalski, 2019 (https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba).

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

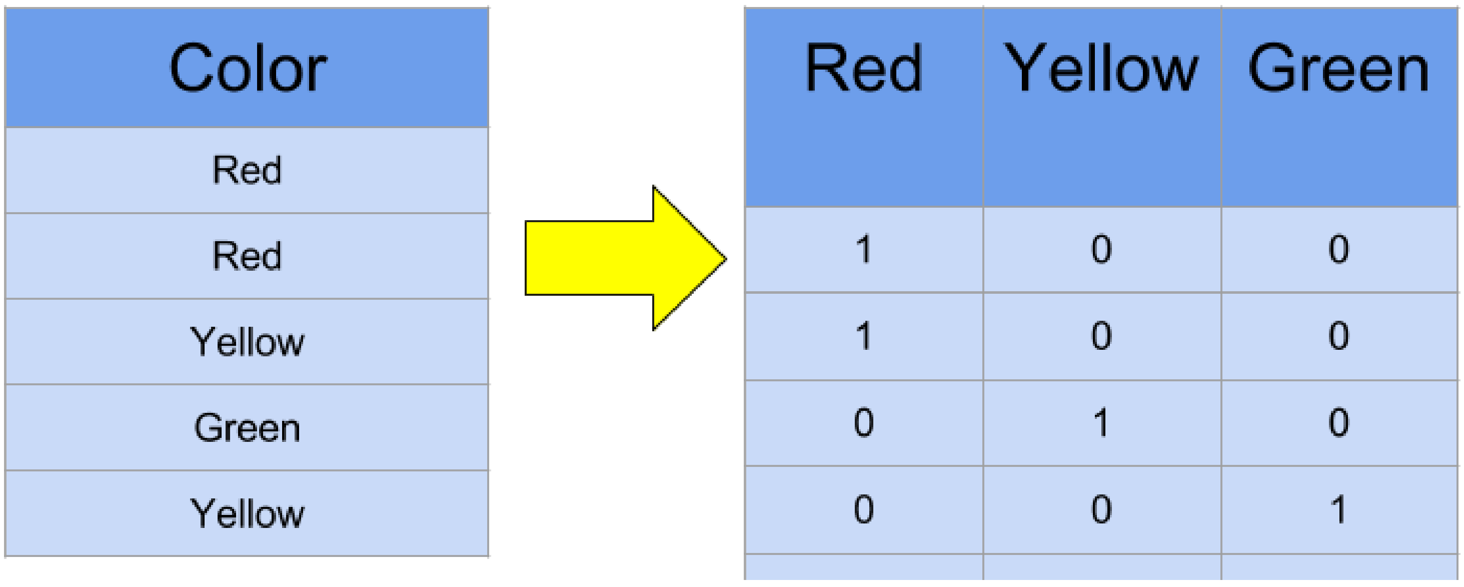
Activation functions in Keras

Data processing: One hot encoding

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
Red
Yellow
Green
Yellow

Red	Yellow	Green
1	0	0
1	0	0
0	1	0
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.

Run the below code on one hot encoding.

```
#source: https://machinelearningmastery.com/how-to-one-hot-encode-sequence-data-in-python/

from numpy import array
from numpy import argmax
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
# define example
data = ['cold', 'cold', 'warm', 'cold', 'hot', 'hot', 'warm', 'cold', 'warm', 'hot']
values = array(data)
```

```
print(values)
# integer encode
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(values)
print(integer_encoded)
# binary encode
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
onehot_encoded = onehot_encoder.fit_transform(integer_encoded)
print(onehot_encoded)
# invert first example
inverted = label_encoder.inverse_transform([argmax(onehot_encoded[0, :])])
print(inverted)
```

Next, we do the same with Caret in R.

```
library(caret)
# check the help file for more details
#?dummyVars

customers <- data.frame(
  id=c(10, 20, 30, 40, 50),
  gender=c('male', 'female', 'female', 'male', 'female'),
  mood=c('happy', 'sad', 'happy', 'sad','happy'),
  outcome=c(1, 1, 0, 0, 0))
customers

dmy <- dummyVars(" ~ .", data = customers)
trsf <- data.frame(predict(dmy, newdata = customers))
trsf
```

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.

Run the code to transform input between 0 and 1.

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

```python
# Python code to Rescale data (between 0 and 1)
import pandas
import scipy
import numpy
from sklearn.preprocessing import MinMaxScaler
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width',  'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
print(array[0:4,:])

# separate array into input and output components
X = array[:,0:4]
Y = array[:,4]
scaler = MinMaxScaler(feature_range=(0, 1))
rescaledX = scaler.fit_transform(X)

# summarize transformed data
numpy.set_printoptions(precision=3)
print(rescaledX[0:4,:])
print(Y[0:4])
```

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

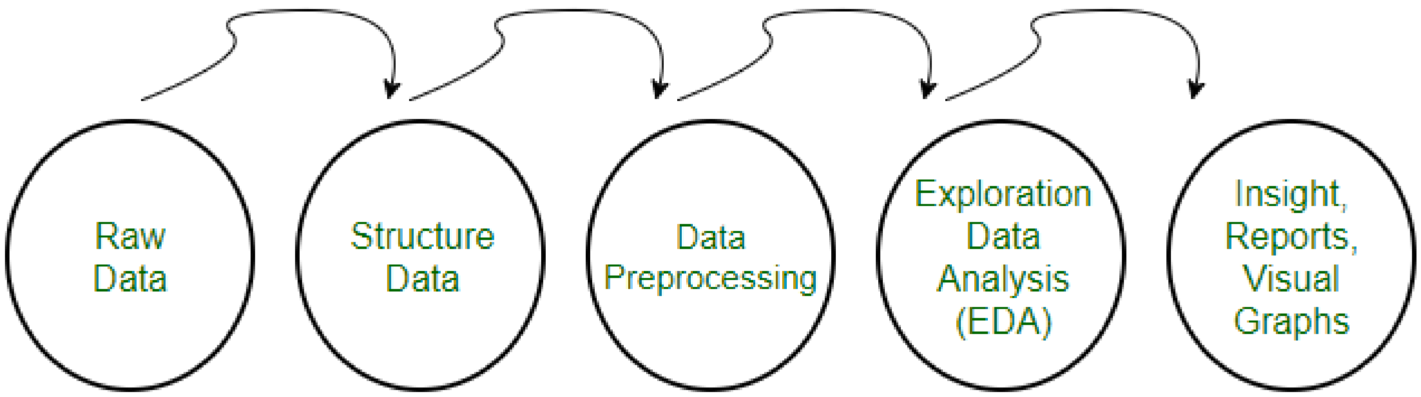

Raw Data
Structure Data
Data Preprocessing
Exploration Data Analysis (EDA)
Insight, Reports, Visual Graphs

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/.

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.

```
#https://analyticsindiamag.com/data-pre-processing-in-python/

from sklearn.preprocessing import Binarizer
import pandas
import numpy
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width',  'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values

# separate array into input and output components
X = array[:,0:4]
Y = array[:,4]
binarizer = Binarizer(threshold=0.0).fit(X)
binaryX = binarizer.transform(X)

# summarize transformed data
numpy.set_printoptions(precision=3)
print(binaryX[0:20,:])
```

Number of hidden layers

Understand the relationship between the number of hidden layers and the performance of a neural network.

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.

Bayesian optimisation package can be helpful. https://scikit-optimize.github.io/stable/

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

Neural Networks for Regression - sklearn

```
#Source: "Linear Regression on Boston Housing Dataset"
# https://towardsdatascience.com/linear-regression-on-boston-housing-dataset-f409b7e4a155

# adapted for MLP by R. Chandra

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn.datasets import load_boston
boston_dataset = load_boston()
from sklearn.preprocessing import StandardScaler

print(boston_dataset.keys())
boston = pd.DataFrame(boston_dataset.data, columns=boston_dataset.feature_names)
boston.head()
boston['MEDV'] = boston_dataset.target

correlation_matrix = boston.corr().round(2)
print(correlation_matrix)
# annot = True to print the values inside the square
sns.heatmap(data=correlation_matrix, annot=True)
plt.savefig('corr.png')
#plt.clr()

plt.figure(figsize=(20, 5))

features = ['LSTAT', 'RM']
target = boston['MEDV']

for i, col in enumerate(features):
    plt.subplot(1, len(features) , i+1)
    x = boston[col]
    y = target
    plt.scatter(x, y, marker='o')
    plt.title(col)
    plt.xlabel(col)
    plt.ylabel('MEDV')
    plt.savefig('feature.png')

  # next we prepare data
X = pd.DataFrame(np.c_[boston['LSTAT'], boston['RM']], columns = ['LSTAT','RM'])
Y = boston['MEDV']
from sklearn.model_selection import train_test_split



scaler = StandardScaler()
```

```
X = scaler.fit(X).transform(X)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.4, random_state=5)
print(X_train.shape)
print(X_test.shape)
print(Y_train.shape)
print(Y_test.shape)

####

from sklearn.neural_network import MLPRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

mlp_model = MLPRegressor(hidden_layer_sizes=(100, ))
lin_model = LinearRegression()

mlp_model.fit(X_train, Y_train)
lin_model.fit(X_train, Y_train)

# model evaluation for training set
y_train_predict = lin_model.predict(X_train)
rmse = (np.sqrt(mean_squared_error(Y_train, y_train_predict)))
r2 = r2_score(Y_train, y_train_predict)

# model evaluation for training set
y_train_predict = mlp_model.predict(X_train)
rmsemlp = (np.sqrt(mean_squared_error(Y_train, y_train_predict)))
r2mlp = r2_score(Y_train, y_train_predict)

print("The model performance for training set")
print("--------------------------------------")
print('RMSE LM is {}'.format(rmse))
print('RMSE MLP is {}'.format(rmsemlp))
print('R2 LM score is {}'.format(r2))
print('R2 MLP score is {}'.format(r2mlp))
print("\n")

# model evaluation for testing set
y_test_predict = lin_model.predict(X_test)
y_test_predict = mlp_model.predict(X_test)
rmse = (np.sqrt(mean_squared_error(Y_test, y_test_predict)))
r2 = r2_score(Y_test, y_test_predict)

rmsemlp = (np.sqrt(mean_squared_error(Y_test, y_test_predict)))
r2mlp = r2_score(Y_test, y_test_predict)

print("The model performance for testing set")
print("--------------------------------------")
print('RMSE LM is {}'.format(rmse))
print('RMSE MLP is {}'.format(rmsemlp))
print('R2 MLP score is {}'.format(r2mlp))
print('R2 LM score is {}'.format(r2))
```

✓ More examples: https://www.python-course.eu/regression_introduction.php

Exercise 3.2

Neural networks for regression problems:

1. Use the dataset from exercise 1.4 and run the same experiments using neural networks and compare with linear regression for the heating load.
   https://edstem.org/au/courses/6212/lessons/13871/slides/111781
2. Extend the model in Part 1 so that it can predict both the heating and cooling load. Therefore, you need to have 2 output neurons with linear outputs.
3. Ensure that you do 30 experiments in all the above tasks. In each experiment, you can have a different train/test split.

Exercise 3.2 Solution

This code slide does not have a description.