Boot Camp on

Creating & Training a Artificial Neural Network

For quick boot up

Objectives of this Boot Camp

Open up Deep Learning area for learners

- What is Neural Network and different applications
- How to create a neural network from scratch
- Implementing Neural network from scratch
- Implementing Neural net with Scikit
- Implementing Neural net with Keras
- How to save model and load it
- Implementing Neural net on Google colab

What is a boot camp?

- Introducing importance of a given field briefly and to give the big picture
- To “enable” participants to enter the area quickly
- Learning by “hands-on” methodology
- To encourage talent on innovation in the chosen area

What a Boot Camp IS NOT

- An arena to learn coding
- To ask theoretical questions in the given area
- To become “experts” in the area after boot camp
- To do better in exams and projects after the boot camp

Prerequisites for the boot camp

The participants need to complete the following steps in order to attend the bootcamp

- Basic understanding of **Python** and **Neural network concepts**
- Laptops - must carry at least one laptop with latest Operating system, wireless and Ethernet cards working properly for internet.
- Good speed internet access
- Peaceful and bright environment to participate in the bootcamp

Creating and Training an Artificial Neural Network

What is a neural network?

The basic idea behind a neural network is to *simulate* lots of densely interconnected brain cells inside a computer so you can get it to learn things, recognize patterns, and make decisions in a humanlike way.

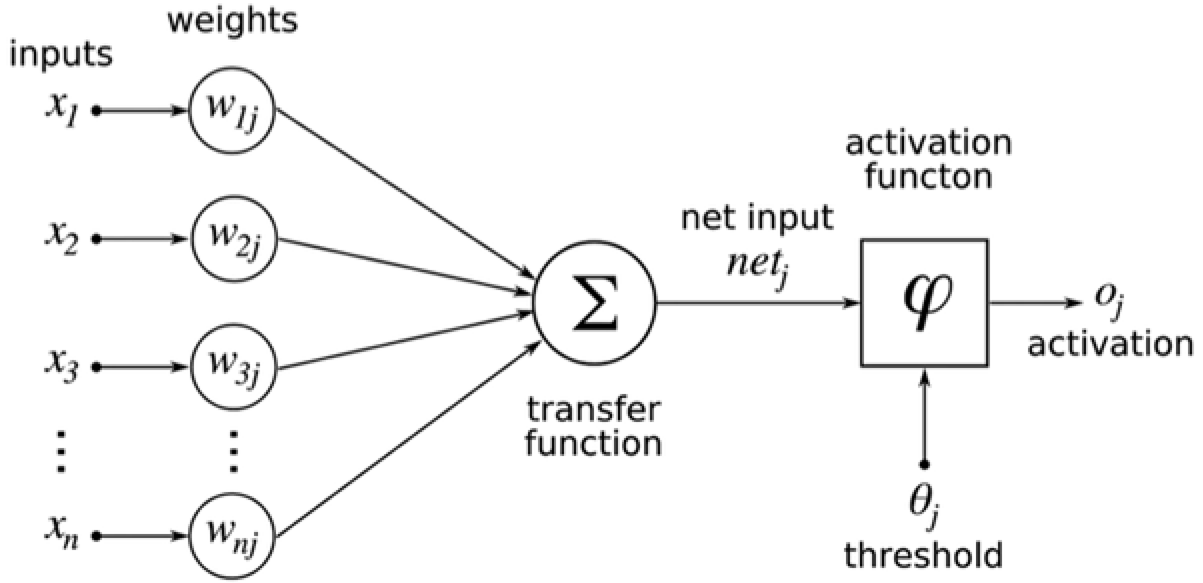
The amazing thing about a neural network is that you don't have to program it to learn explicitly: ***it learns all by itself, just like a brain!***

Application of neural networks

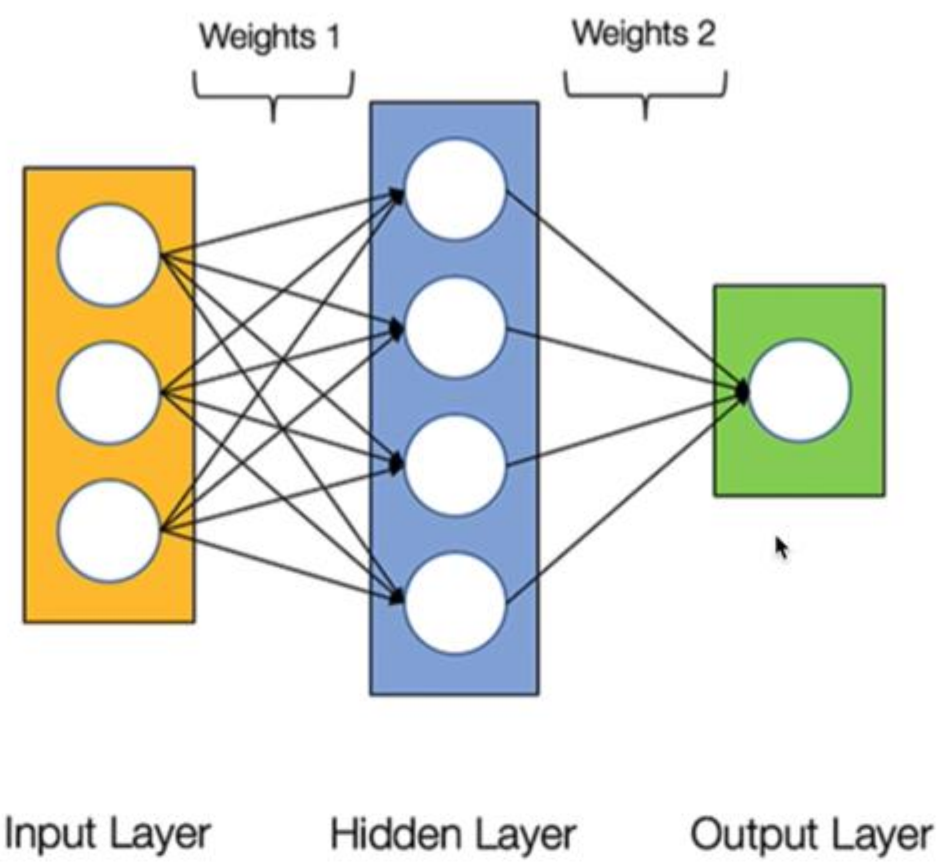
- Process modelling and control
- Machine Diagnostics
- Portfolio Management
- Target Recognition
- Medical Diagnosis
- Credit Rating
- Targeted Marketing
- Voice recognition
- Face recognition
- Financial Forecasting
- Intelligent searching
- Fraud detection

Components

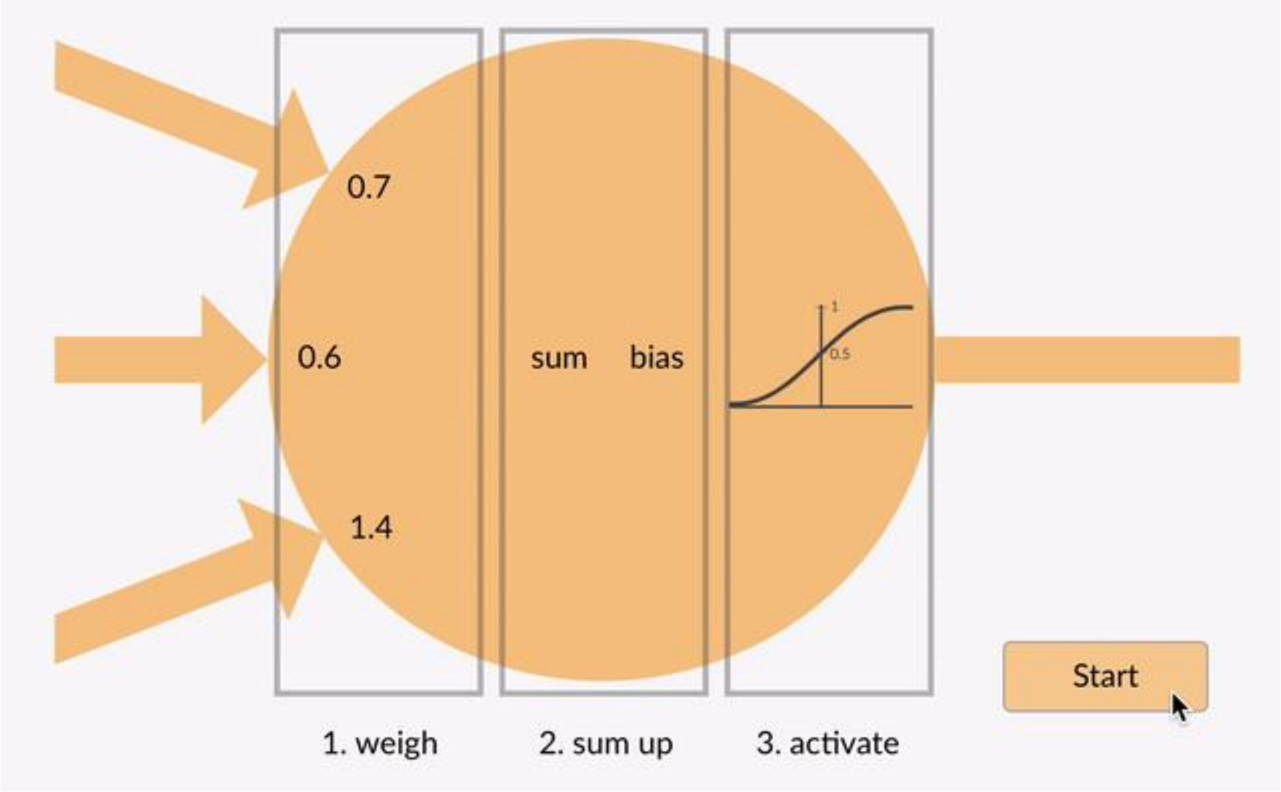
- Input Layer
- Hidden Layer
- Output Layer
- Weights and Biases between Layers
- Activation Function



2 Layer Architecture



Single Neuron



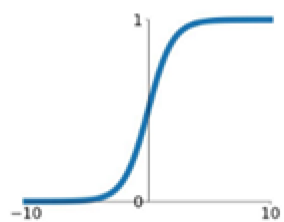
https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/

Activation Function

Activation Functions

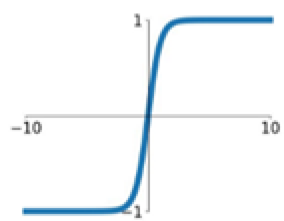
Sigmoid

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



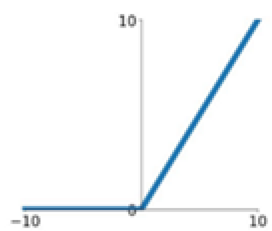
tanh

$$\tanh(x)$$



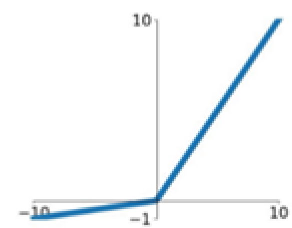
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

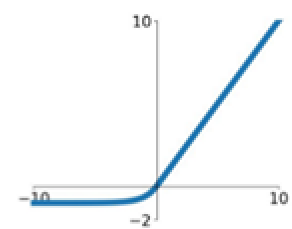


Maxout

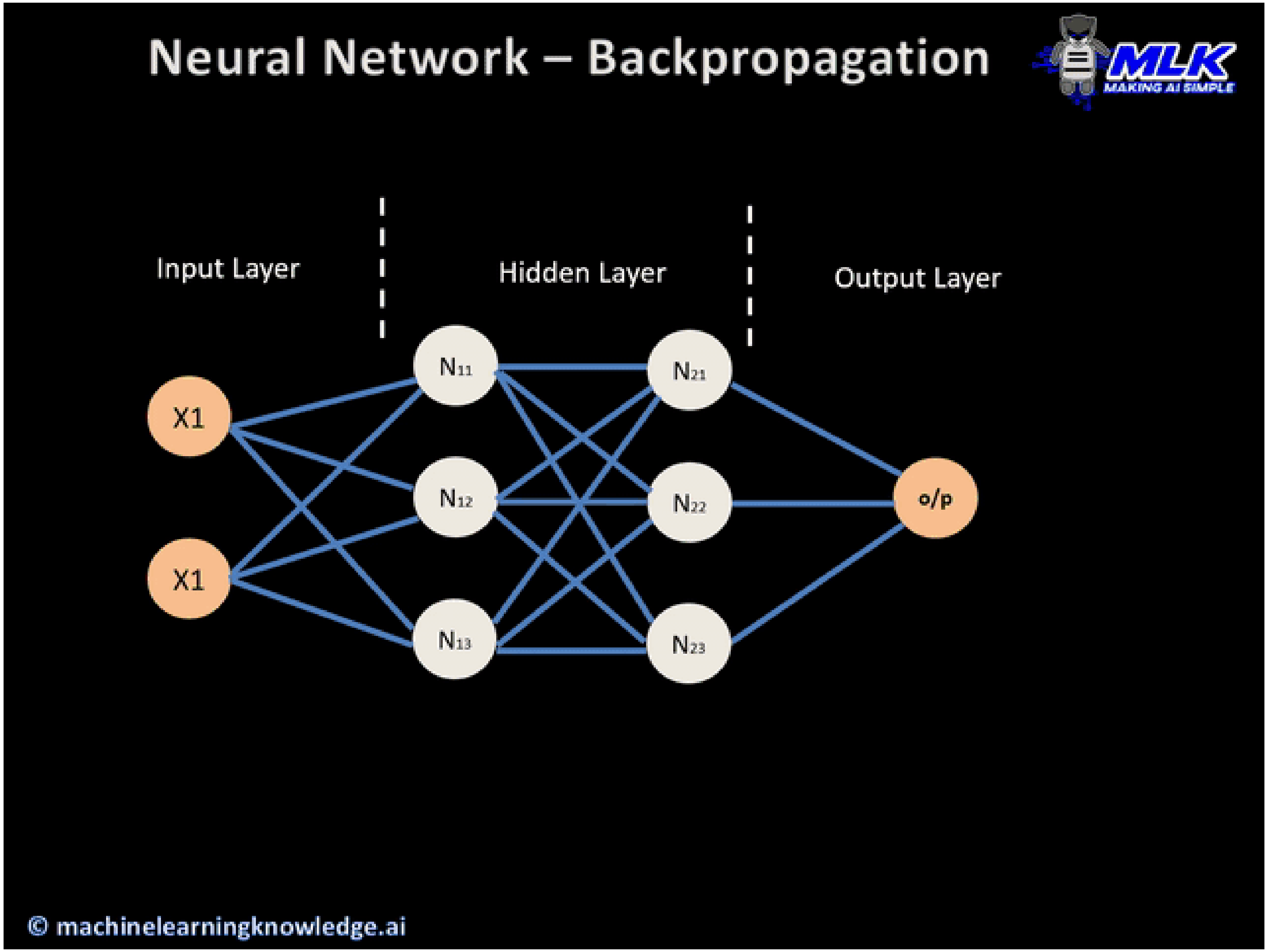
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

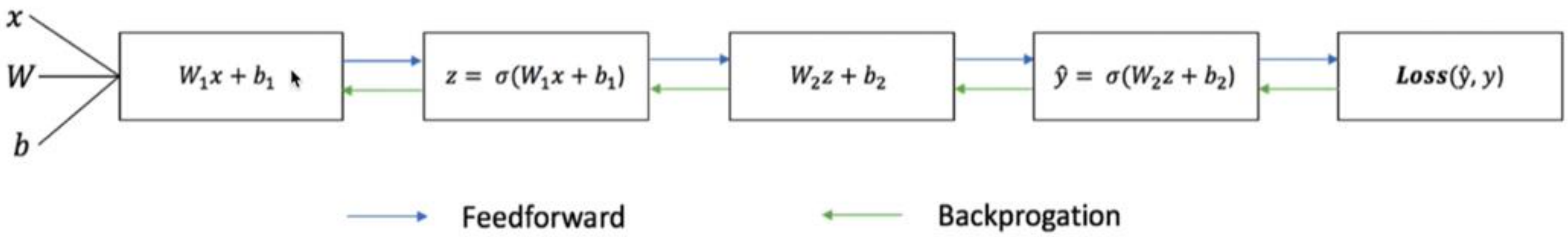


Working of ANN



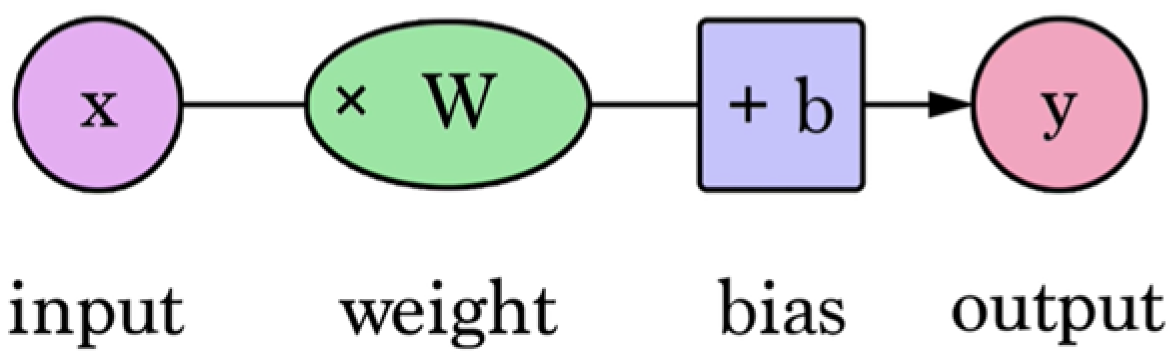
Ref: Machinelearningknowledge.ai

Sequential Graph



Training a Neural Network

- Output is designated by the following function:
  - $Y = \sigma(W_2\sigma(W_1x+b_1)+b_2)$
    - Weights are represented by W_1 and W_2
    - Biases represented by b_1 and b_2
- Two Steps in Training:
  - Feedforward
  - Backpropagation



Loss Functions

A loss function, that can be used to estimate the loss of the model so that the weights can be updated to reduce the loss on the next evaluation.

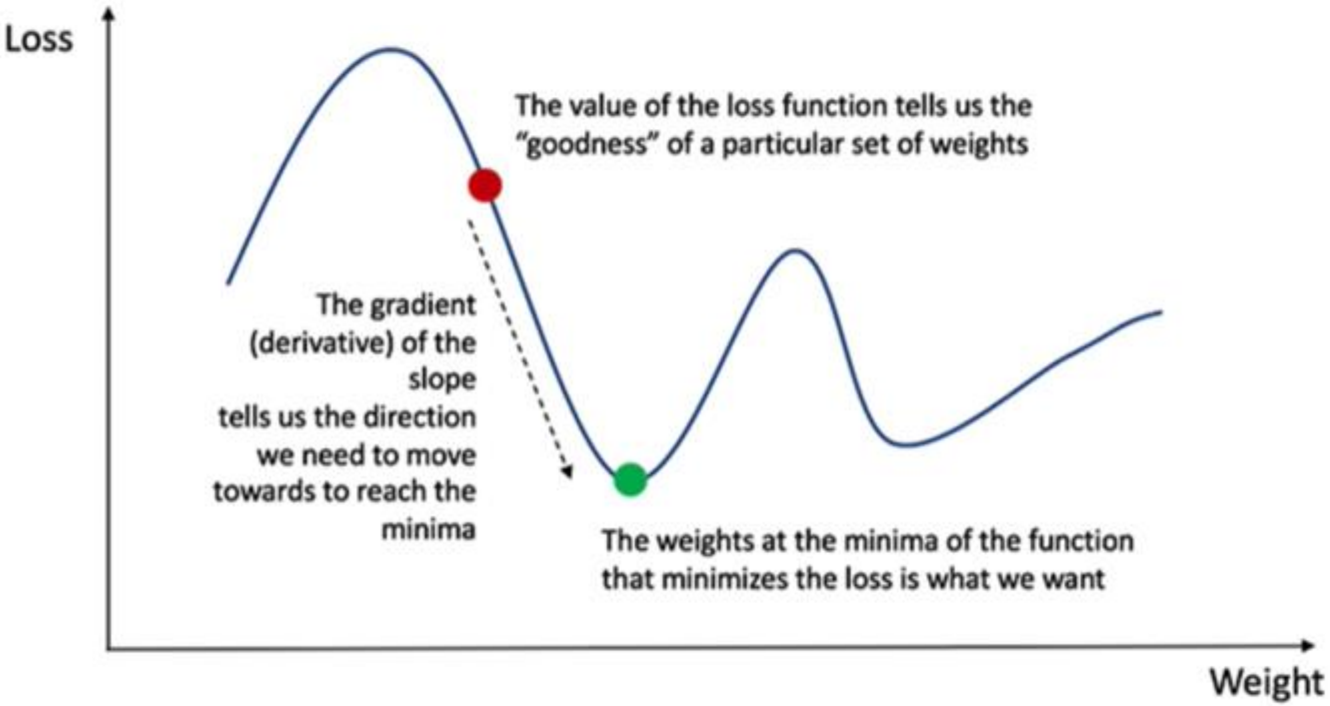
1. Regression Loss Functions
    1. Mean Squared Error Loss
    2. Mean Squared Logarithmic Error Loss
    3. Mean Absolute Error Loss
2. Binary Classification Loss Functions
    1. Binary Cross-Entropy
    2. Hinge Loss
    3. Squared Hinge Loss
3. Multi-Class Classification Loss Functions
    1. Multi-Class Cross-Entropy Loss
    2. Sparse Multiclass Cross-Entropy Loss
    3. Kullback Leibler Divergence Loss

Gradient Descent

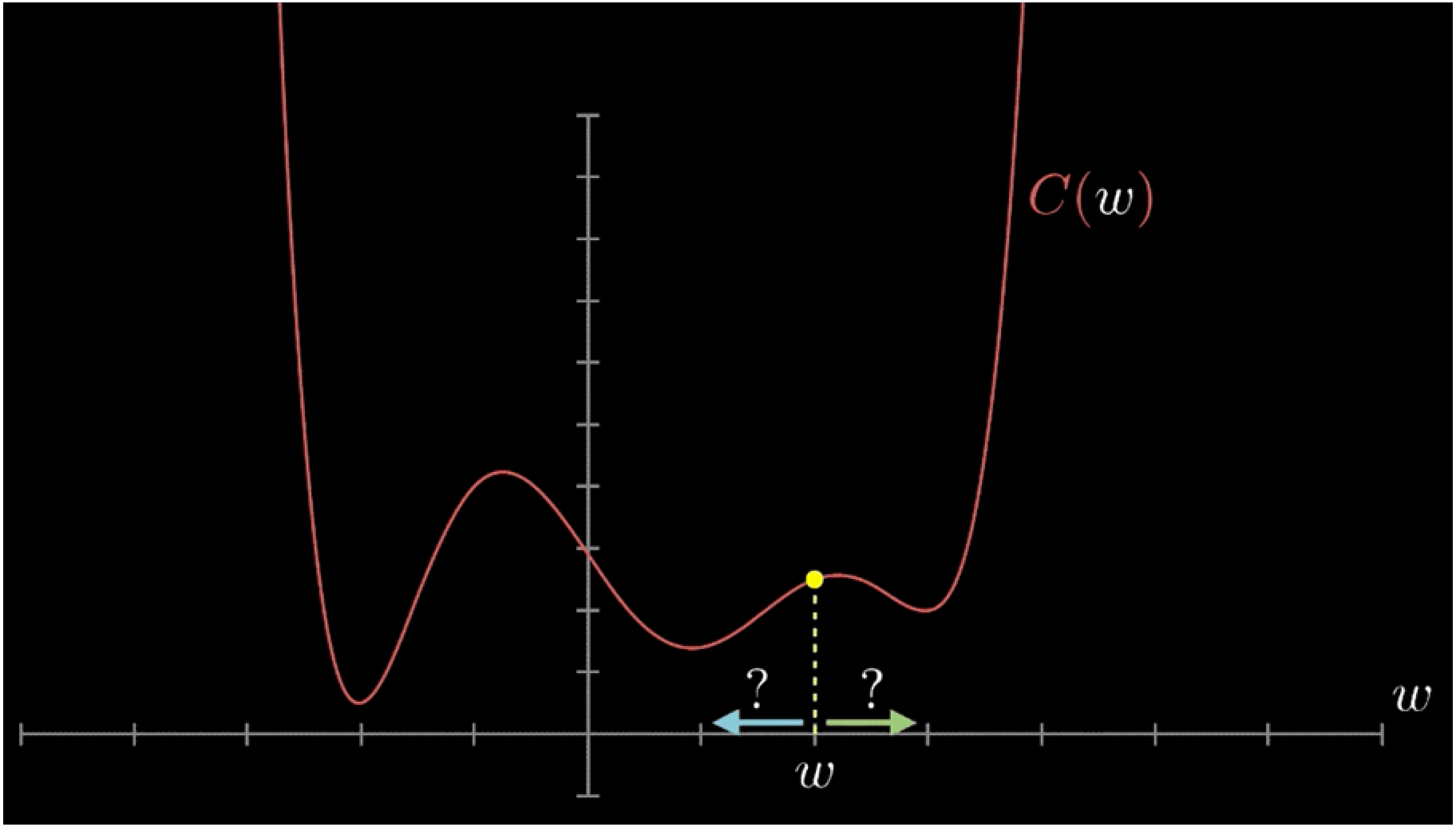
- Derivative of the Loss Function with respect to weights and biases

$$W_1 = W_1 - \alpha \frac{dL}{dW_1}$$

$$W_2 = W_2 - \alpha \frac{dL}{dW_2}$$



Gradient Descent



Ref: mlfromscratch.com

Build ANN from Scratch

What Are You Trying to Solve?

We are going to create a neural network that predicts if a person will have heart disease or not

Heart Dataset

UCI

Machine Learning Repository

Center for Machine Learning and Intelligent Systems

About Citation Policy Donate a Data Set Contact

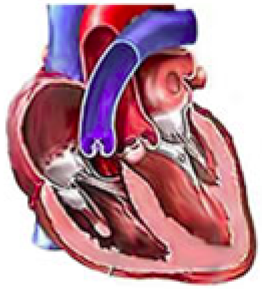
Search

Repository Web Google™

View ALL Data Sets

Statlog (Heart) Data Set

Download: Data Folder, Data Set Description



Abstract: This dataset is a heart disease database similar to a database already present in the repository (Heart Disease databases) but in a slightly different form

Data Set Characteristics:	Multivariate	**Number of Instances:**	270	**Area:**	Life
Attribute Characteristics:	Categorical, Real	**Number of Attributes:**	13	**Date Donated**	N/A
Associated Tasks:	Classification	**Missing Values?**	No	**Number of Web Hits:**	209551

Download heart.dat

Import Data

```
#prepare data downloaded from UCL

import csv
import pandas as pd

# add header names
headers =  ['age', 'sex','chest_pain','resting_blood_pressure',
        'serum_cholestoral', 'fasting_blood_sugar', 'resting_ecg_results',
        'max_heart_rate_achieved', 'exercise_induced_angina', 'oldpeak',"slope of the peak",
        'num_of_major_vessels','thal', 'heart_disease']

heart_df = pd.read_csv('heart.dat', sep=' ', names=headers)
```

Features

The features present in the dataset are:

- Age
- sex
- chest pain type (4 values)
- resting blood pressure
- serum cholesterol in mg/dl
- fasting blood sugar > 120 mg/dl
- resting electrocardiographic results (values 0,1,2)
- maximum heart rate achieved
- exercise-induced angina
- oldpeak (ST depression induced by exercise relative to rest)
- the slope of the peak exercise ST segment
- number of major vessels (0–3) colored by fluoroscopy
- thal (3 = normal; 6 = fixed defect; 7 = reversible defect)
- **heart_disease: absence (1) or presence (2) of heart disease**

Replace target class to binary

```
#replace target class with 0 and 1
#1 means "have heart disease" and 0 means "do not have heart disease"
heart_df['heart_disease'] = heart_df['heart_disease'].replace(1, 0)
heart_df['heart_disease'] = heart_df['heart_disease'].replace(2, 1)
```

Train – Test Data Split and Standardize

```
#Importing essential packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings("ignore")

#Splitting data into independant and depedant variables

X = heart_df.drop(columns=['heart_disease']) #Independant data variables

y_label = heart_df['heart_disease'].values.reshape(X.shape[0], 1) #Dependant or target variable
#print(y_label)
#Split data into train and test set
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y_label, test_size=0.2, random_state=2)

#Standardize the dataset
sc = StandardScaler()
sc.fit(Xtrain)
Xtrain = sc.transform(Xtrain)
Xtest = sc.transform(Xtest)

print(f"Shape of train set is {Xtrain.shape}")
print(f"Shape of test set is {Xtest.shape}")
print(f"Shape of train label is {ytrain.shape}")
print(f"Shape of test labels is {ytest.shape}")
```

Create a 2-layer Neural Network Class

```
class NeuralNet():
    '''
    A two layer neural network
    '''

    def __init__(self, layers=[13,8,1], learning_rate=0.001, iterations=100):
        self.params = {}
        self.learning_rate = learning_rate
        self.iterations = iterations
        self.loss = []
        self.sample_size = None
        self.layers = layers
        self.X = None
        self.y = None

    def init_weights(self):
        '''
        Initialize the weights from a random normal distribution
        '''
        np.random.seed(1) # Seed the random number generator
        self.params["W1"] = np.random.randn(self.layers[0], self.layers[1])
        self.params['b1']  =np.random.randn(self.layers[1],)
        self.params['W2'] = np.random.randn(self.layers[1],self.layers[2])
        self.params['b2'] = np.random.randn(self.layers[2],)
```

2-layer Neural Network Class

- We created a **neural network class**, and then during initialization, you created some variables to hold intermediate calculations.
- The argument layers is a list that stores your network's architecture. You can see that it accepts 13 input features, uses 8 nodes in the hidden layer and finally uses 1 node in the output layer.
- Moving on to the next code section, you created a function (***init_weights***) to initialize the weights and biases as random numbers. These weights are initialized from a uniform random distribution and saved to a dictionary called **params**.
- The first weight array (***W1***) will have dimensions of 13 by 8—this is because you have 13 input features and 8 hidden nodes,while the first bias (***b1***) will be a vector of size 8 because you have 8 hidden nodes.
- The second weight array (***W2***) will be a 1-dimensional array because you have 8 hidden nodes and 1 output node, and finally, the second bias (***b2***) will be a vector of size because you have just 1 output.

Defining Activation Function

- We are using relu and sigmoid

```
def relu(self,Z):
        '''
        The ReLu activation function is to performs a threshold
        operation to each input element where values less
        than zero are set to zero.
        '''
        return np.maximum(0,Z)


def sigmoid(self,Z):
        '''
        The sigmoid function takes in real numbers in any range and
        squashes it to a real-valued output between 0 and 1.
        '''
        return 1.0/(1.0+np.exp(-Z))
```

Defining Activation Function

- We are using relu and sigmoid

```
def relu(self,Z):
        '''
        The ReLu activation function is to performs a threshold
        operation to each input element where values less
        than zero are set to zero.
        '''
        return np.maximum(0,Z)


def sigmoid(self,Z):
        '''
        The sigmoid function takes in real numbers in any range and
        squashes it to a real-valued output between 0 and 1.
        '''
        return 1.0/(1.0+np.exp(-Z))
```

Loss function - Cross-entropy

$$CE = -\sum_{i}^{C} y_i log(\hat{y}_i)$$

Where C is the number of classes, y is the true value and y_hat is the predicted value.

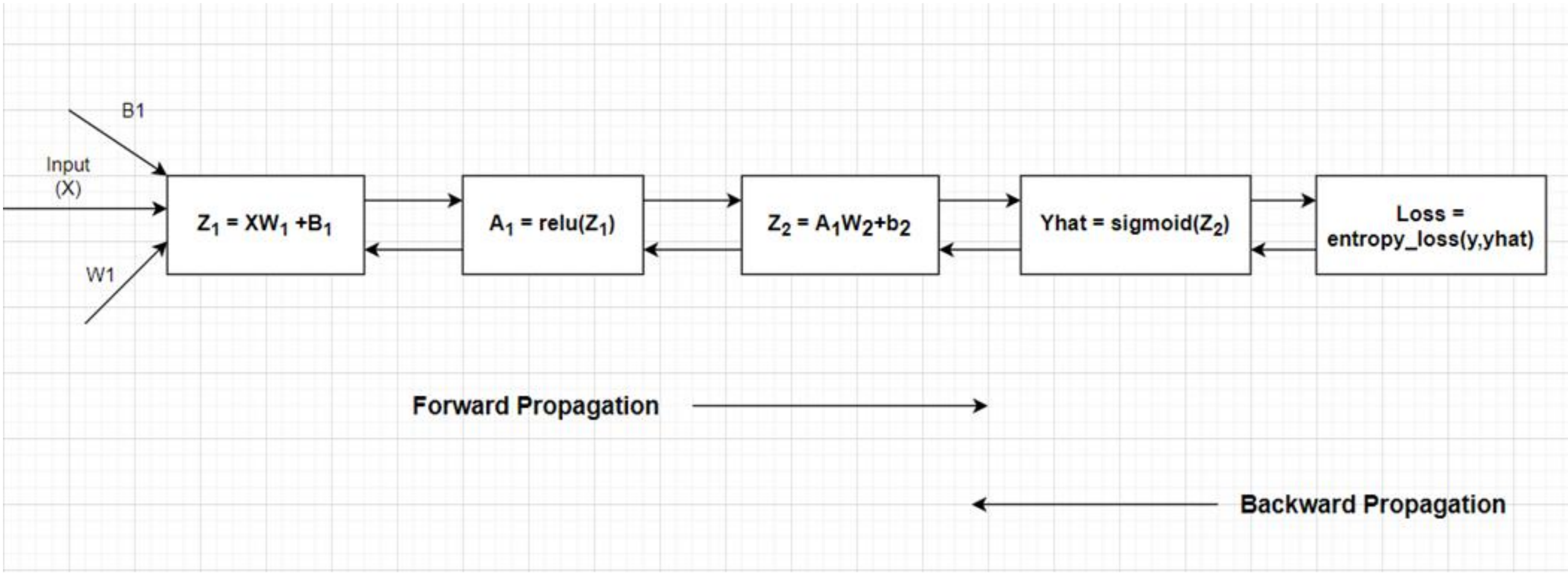
For a binary classification task (i.e. C=2), the cross-entropy loss function becomes:

$$CE = -\sum_{i=1}^{2} y_1 log(\hat{y}) = -y_1 log(\hat{y}_1) - (1 - y_1) log(1 - \hat{y}_1)$$

Defining Loss function

```
def entropy_loss(self,y, yhat):
    nsample = len(y)
    loss = -1/nsample * (np.sum(np.multiply(np.log(yhat), y) + np.multiply((1 - y), np.log(1 - yhat))))
    return loss
```

Sequential Graph



Forward Propagation

In this two-layer network, we have to perform the following computation for forward propagation:

- Compute the weighted sum between the input and the first layer's weights and then add the bias: **Z1 = (W1 * X) + b**
- Pass the result through the ReLU activation function: **A1 = Relu(Z1)**
- Compute the weighted sum between the output (A1) of the previous step and the second layer's weights—also add the bias: **Z2 = (W2 * A1) + b2**
- Compute the output function by passing the result through a sigmoid function: **A2 = sigmoid(Z2)**
- And finally, compute the loss between the predicted output and the true labels: **loss(A2, Y)**

Forward Propagation Definition

```
def forward_propagation(self):
    '''
    Performs the forward propagation
    '''

    Z1 = self.X.dot(self.params['W1']) + self.params['b1']
    A1 = self.relu(Z1)
    Z2 = A1.dot(self.params['W2']) + self.params['b2']
    yhat = self.sigmoid(Z2)
    loss = self.entropy_loss(self.y,yhat)

    # save calculated parameters
    self.params['Z1'] = Z1
    self.params['Z2'] = Z2
    self.params['A1'] = A1

    return yhat,loss
```

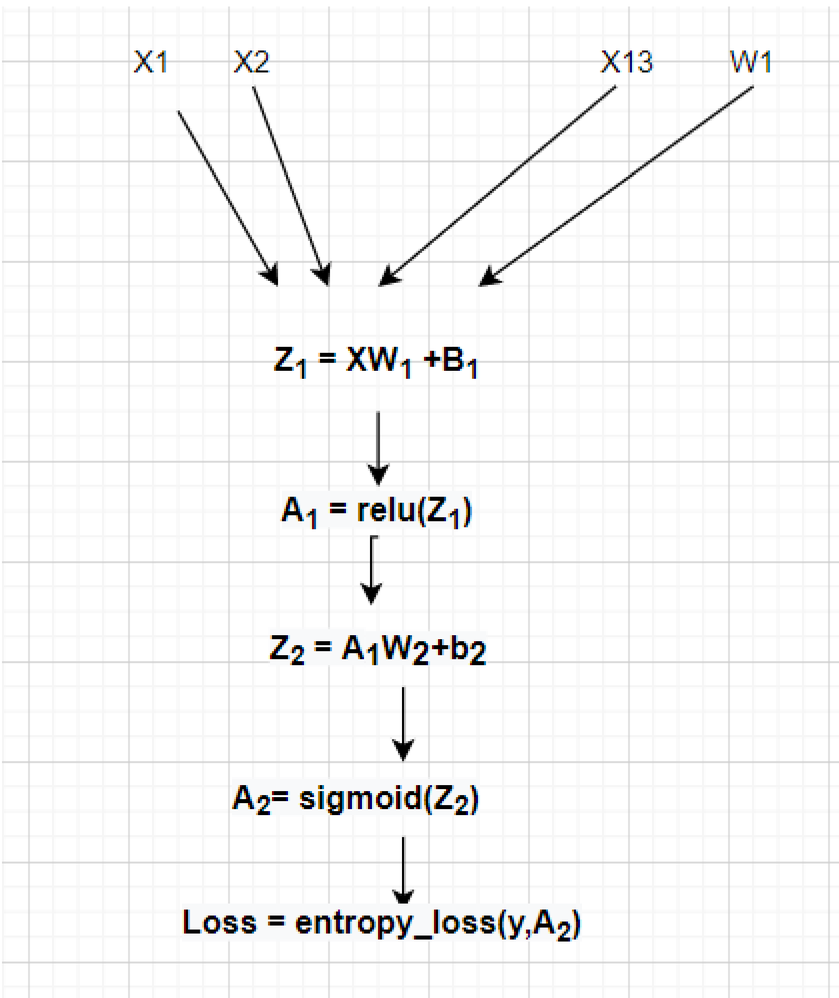
Backward Propagation

After computing the output and loss in the forward propagation layer, you'll move to the backpropagation phase, where you calculate the derivatives backward, from the loss all the way up to the first weight and bias.

$$W_1 = W_1 - \alpha \frac{dL}{dW_1}$$

$$W_2 = W_2 - \alpha \frac{dL}{dW_2}$$

Backward Propagation



$$\frac{dL}{dW_2} = \frac{dL}{dA_2}\frac{dA_2}{dZ_2}\frac{dZ_2}{dW_2}$$

$$\frac{dL}{dW_1} = \frac{dL}{dA_2}\frac{dA_2}{dZ_2}\frac{dZ_2}{dA_1}\frac{dA_1}{dZ_1}\frac{dZ_1}{dW_1}$$

Defining Backward Propagation

```
def back_propagation(self,yhat):
        '''
        Computes the derivatives and update weights and bias according.
        '''
        def dRelu(x):
            x[x<=0] = 0
            x[x>0] = 1
            return x

        dl_wrt_yhat = -(np.divide(self.y,yhat) - np.divide((1 - self.y),(1-yhat)))
        dl_wrt_sig = yhat * (1-yhat)
        dl_wrt_z2 = dl_wrt_yhat * dl_wrt_sig

        dl_wrt_A1 = dl_wrt_z2.dot(self.params['W2'].T)
        dl_wrt_w2 = self.params['A1'].T.dot(dl_wrt_z2)
        dl_wrt_b2 = np.sum(dl_wrt_z2, axis=0)

        dl_wrt_z1 = dl_wrt_A1 * dRelu(self.params['Z1'])
        dl_wrt_w1 = self.X.T.dot(dl_wrt_z1)
        dl_wrt_b1 = np.sum(dl_wrt_z1, axis=0)
```

Updating weights

```
#update the weights and bias
self.params['W1'] = self.params['W1'] - self.learning_rate * dl_wrt_w1
self.params['W2'] = self.params['W2'] - self.learning_rate * dl_wrt_w2
self.params['b1'] = self.params['b1'] - self.learning_rate * dl_wrt_b1
self.params['b2'] = self.params['b2'] - self.learning_rate * dl_wrt_b2
```

Training of the Neural Network

```
def fit(self, X, y):
        '''
        Trains the neural network using the specified data and labels
        '''
        self.X = X
        self.y = y
        self.init_weights() #initialize weights and bias


        for i in range(self.iterations):
            yhat, loss = self.forward_propagation()
            self.back_propagation(yhat)
            self.loss.append(loss)
```

Making Predictions

```
def predict(self, X):
        '''
        Predicts on a test data
        '''
        Z1 = X.dot(self.params['W1']) + self.params['b1']
        A1 = self.relu(Z1)
        Z2 = A1.dot(self.params['W2']) + self.params['b2']
        pred = self.sigmoid(Z2)
        return np.round(pred)
```

Build ANN Using Scikit-learn

Extensions to SciPy (Scientific Python) are called SciKits. SciKit-Learn provides machine learning algorithms.

- Algorithms for supervised & unsupervised learning
- Built on SciPy and Numpy
- Standard Python API interface
- Sits on top of c libraries, LAPACK, LibSVM, and Cython
- Open Source: BSD License (part of Linux)

Building Model using Scikit Package

Applying same dataset over MLPClassifier under Scikit Package

```
#With scikit learn - with multilayer perceptron classifier
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

sknet = MLPClassifier(hidden_layer_sizes=(8),learning_rate_init=0.001, max_iter=100)
```

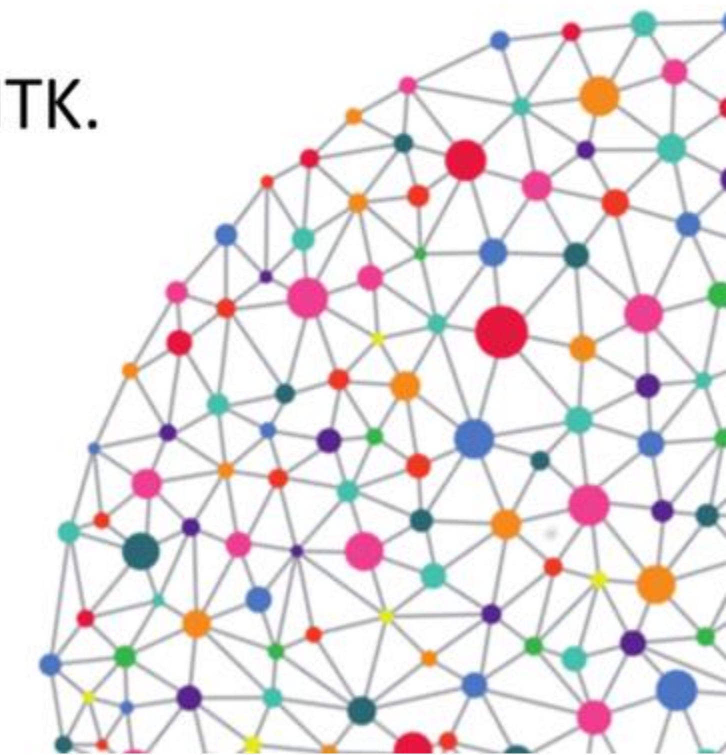
```
#Fit the data to the classifier model
sknet.fit(Xtrain, ytrain)
preds_train = sknet.predict(Xtrain)
preds_test = sknet.predict(Xtest)

#Print the accuracy of the train and test datasets
print("Train accuracy of sklearn neural network: {}".format(round(accuracy_score(preds_train, ytrain),2)*100))
print("Test accuracy of sklearn neural network: {}".format(round(accuracy_score(preds_test, ytest),2)*100))
```

Build ANN Using Keras

What is Keras

- High Level neural network API
- Written in Python
- Integration with TensorFlow, Theano & CNTK.
  - (MXNet backend for Keras on the way!)



Why Keras

- Fast prototyping
- Supports CNN, RNN & combination of both
- Modularity
- Easy extensibility
- Simple to get started, simple to keep going
- Deep enough to build serious models.
- Well-written document.
- Runs seamlessly on CPU and GPU.

Keras Pipeline

1. Define Network
2. Compile Network
3. Fit Network
4. Evaluate Network
5. Make Predictions

Building Models using Keras

```
#With Keras

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

#Define the model
model = Sequential()
model.add(Dense(8,input_shape=(13,)))
model.add(Dense(1, activation='sigmoid'))
#model.summary()
```

```
# compile the model
opt = Adam(learning_rate=0.001)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
```

```
#Fitting the model to data - Training
model.fit(Xtrain, ytrain, epochs=10, verbose=1)
```

References -

1. Neuron - https://commons.wikimedia.org/wiki/File:Neuron.svg
2. Artificial Neuron - https://commons.wikimedia.org/wiki/File:Artificial_neural_network.png
3. Multilayer Neural Network - https://commons.wikimedia.org/wiki/File:MultiLayerNeuralNetworkBigger_english.png
4. Single Neuron - https://heartbeat.fritz.ai/building-a-neural-network-from-scratch-using-python-part-1-6d399df8d432
5. Activation Function - https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092
6. Forward and backpropagation - https://commons.wikimedia.org/wiki/File:Backprogation_neural_networks.png
7. Gradient Descent - https://www.oreilly.com/library/view/learn-arcore-/9781788830409/e24a657a-a5c6-4ff2-b9ea-9418a7a5d24c.xhtml