MLPonRiceDataset

January 31, 2021

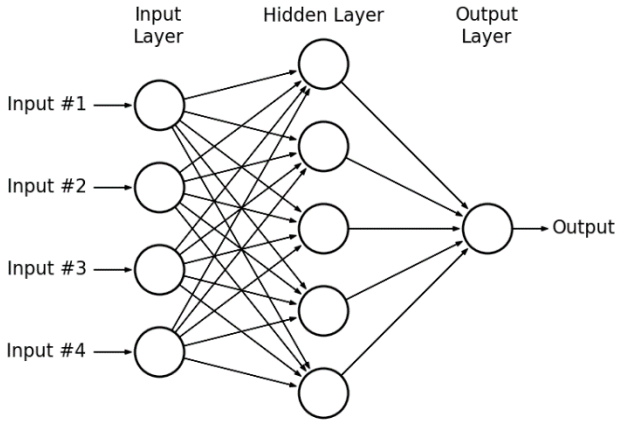
1 Multilayer Perceptron from Scratch

Details of the task:

For this homework, we ask you to implement multilayer perceptron you need to implement at least a one hidden layer(any number of hidden layers is fine) for the classification data you are given. You need to implement backpropagation and layer weight updates without using readily available library codes. You will be implementing network structure and training methods yourself.(You can use readily available functions for trying different activation functions and backpropagation methods). You can use library codes for mathematical operations and data management. You need to classify rice type using the other features given to you.(Deep neural networks are just mlp's with wider and deeper structure.)

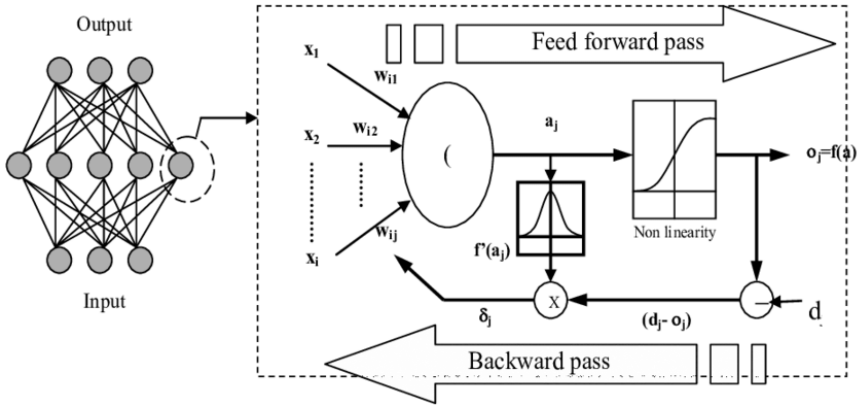
2 What is Multilayer Perceptron?

In Homework-2, we had studied on "perceptron" algorithms, which is under supervised learning mechanisms. Perceptron is just a neuron which transfers input to output with the adjusted weights. As the name suggets, multilayer perceptron does this for multiple layers and multiple neurons.



As you can see, there is an input layer, which takes the features. Then the neurons (as known as perceptrons) tries different weights and gives various outputs. Hidden layers may be 1 or more, then it gives output for the classification. However, from the image on the top, you cannot see the evaluating mechanisms for the weight updates. Because the approach is a little different from "mere perceptron" in multilayer perceptron.

What we need in multilayered approach are: 1- Forward propagation: Which also known as forward feed from previous layer to next layer 2- Back propagation: It is used for error evaluation for the weight updates from the layer to previous layer 3- Weight update protocol: In this case what we will use is gradient descent algorithm. 4- Storing the outputs: Actually, storing the activation function outputs and derivatives of the activations to use them in the back propagation stage.



These are important concepts with mathematical background, which need more and more explicit explanations. In the picture below, you can see a neuron's cycle. As you can imagine, this MLP takes much more time to train the set than the single perceptron. The complexity of it depends on the number of neurons, dataset's scale and number of layers. I will call multilayer perceptron as neural network, because it is cool in this way. So, let's take a look at the code and explain further.

[217]:
```
import numpy as np
from random import random
import pandas as pd
import math
import os
```

As you can see, we only have Numpy framework. I had imported pandas for the any case I need to do further Dataframe actions. Random is imported so that we have randomized weights as initials. Actually, the convergence of the results also depends on the weight updates as you will see. So, if you want to get the same results at each run, you can use seed function for consistency. Math is used for exponentials, which is an essential for our activation function aka sigmoid function. But, you can always change the activation function as you like.

In the code below, we had 2 datasets: Ricetrain.csv and Ricetest.csv. In the first 7 column, we had features and the last column has the binary output. There is no missing values in the dataset and since it is binary, I had encoded Osmancik to -1, and Cammeo to 1.

Also, for the performance issues, I had *roughly* normalized the columns by dividing them into respective columns' max values. However, a better would be using Max - Min to divide the value. But this approach also worked quite well. Without normalization, we have much bigger outputs to evaluate and it goes off to the limits of the integer offset. I tried, it went off limits with red warnings and took very long time to compute.

[218]:
```
os.chdir("C:\\Users\\esrac\\Downloads")
df = pd.read_csv('Ricetrain.csv', header=None)
```

```
df_shuffled = df.sample(frac=1)
y = df_shuffled.iloc[0:,7].values
y = np.where(y == "Osmancik", -1 ,1)
X = df_shuffled.iloc[0:, 0:7].values

df_test = pd.read_csv('Ricetest.csv', header=None)
df_shuffled_test = df_test.sample(frac=1)

y_test = df_shuffled_test.iloc[0:,7].values
print(y_test)
y_test = np.where(y_test == "Osmancik", -1 ,1)
X_test = df_shuffled_test.iloc[0:, 0:7].values

X_normed = X / X.max(axis=0)
X_test_normed = X_test / X_test.max(axis=0)
```

```
['Cammeo' 'Cammeo' 'Osmancik' ... 'Osmancik' 'Osmancik' 'Osmancik']
```

Starting with the explanations on the references: Videos, codes and articles I used. My main 2 source is https://www.youtube.com/c/ValerioVelardoTheSoundofAI/ and https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/ the MachineLearningMastery.com has very similar example with step by step tutorial. It is also very helpful resource to look on.

The code implementation is from the videos of Valerio Velardo on MLP. The repository can be found here: https://github.com/musikalkemist/DeepLearningForAudioWithPython The videos can be watched there: https://www.youtube.com/watch?v=0oWnheK-gGk The channel has many machine learning and deep learning videos, the 6-7-8th series are for the Multilayer Perceptron from scratch, so I had used all the explanations and codes from there and just tuned a little bit. I recommended to my friends to watch these videos. It is clear as crystal water.

Now let's being to explain all the functions with deeper knowledge.

3 How to create a Neural Network with Perceptrons?

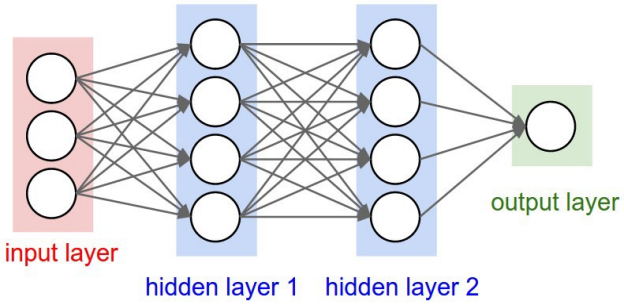
When we say multilayer perceptron, according to my researches, we do not indicate the one we are going to implement. A perceptron is always forward-feeded, where neural network has loops for evaluation, as we are going to see in details. So, what is neural network? Actually, we need to say, artificial neural networks, because they are just a mere implementation of our neural network. How does it work in our body? We get an input from outer world, then we transmit it using chemical reactions and electric signals to our brain's respective areas by evaluating the input. We have some thresholds for the check the values' significance to move forward. These are so much similar to artificial neural networks. Our neurons are perceptrons, each iteration is layers and the iterative mechanism is forward propagation. The threshold for output is activiation functions. We only do not have back propagation like action in our neural network, but back propagation is also something like humane thing: we learn from our past mistakes and know what actually the input means.

Let's look it in a different example. We are cooking and we need salt. Salt is white and in the similar jar as the sugar. So, first we need to check the vision. Our eyes takes the inputs and features by looking at it, using the neurons from eyeballs to brain and forward propagate to our brain's Occipital Lob (whose task is about seeing and vision) by activating the necessary neurons on the path. But this does not alone informative, they are all white. From past experiences, we know salt is matte where sugar is shiny. You check the shiny-ness of the grains from the older training experiences. So this is similar to back propagation. But better way is to taste the white grains to be more accurate, because your past experiences say that you may be not that much accurate and you are hungry, so you should not waste food. So tongue, taste buds activate other neurons and so on...

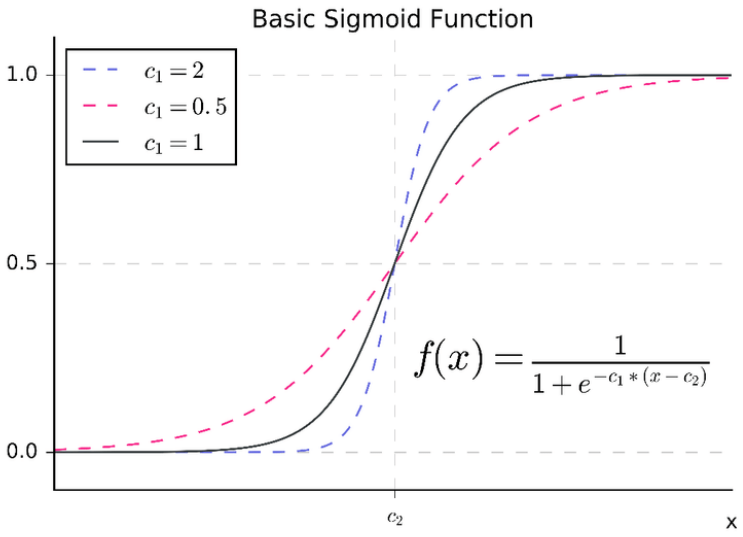
Yes, this is what artificial neural network is. As we said, each neuron is perceptron and there are layers between each neuron links. Each neuron has weights to be updated. So, we need to give some initial weights by random function.

There may be multiple neurons at each level to evaluate input and the weights. In the input layer, we need as the number of the features of the data. Then, there may be any dimensional multi layers which are going to be hidden layers for further evaluation. Then there should be an output layer to give the last evaluations.

We need a feed forward function, which is named as forwardProp at the code segment. It is used to bring the inputs from the layer i with respective weights to next layer i+1. We created a weight vector on 2d, because of this reason. We have ith layer and i+1th layer till the end.



Then we created an activations list of the neurons for each layer. We will store the weights, activations and the derivatives to back propagating reasons. Derivatives give us the direction to update the weights for the accurate classifications.



Then we create forwardProp function to transmit the values to next layer. In this function, we iterate the activations lists by using sigmoid (logistic) function to get the values between 0 and 1. It's derivative is also easy to evaluate. So, sigmoid function is very handy for these representations.

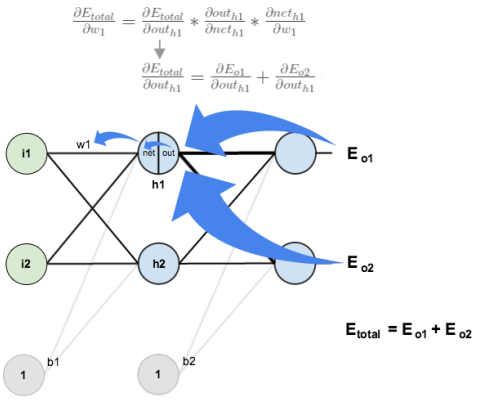
$$\frac{d}{dx}S(x) = \frac{1}{(1+e^{-x})}\left(1 - \frac{1}{1+e^{-x}}\right)$$

The most important part to understand is back propagation. In backProp function, we need to go back from right to left to evaluate the weights by considering misclassifications aka errors. We take the activations from the i+1th layer to evaluate 1 previous layer's performance iteratively. As the input, it gets errors and we need to calculate delta, which is actually errors * activation's derivative. And we reshaped the delta and activations for the matrix dot product purposes for i+1th layer's activations and ith layer activations, which is in accordance with the formulas.

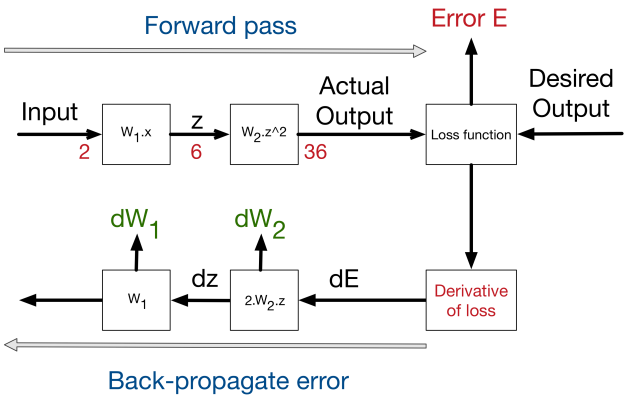
After the back propagation and error evaluation, we need to update weights according to the direction of the derivatives. This is where gradient descent algorithm comes into the stage. It is as we know, we update the weights with learning rate times the derivative of the activations from the

Output Layer Error	$E_o = (O - y) \cdot R'(Z_o)$
Hidden Layer Error	$E_h = E_o \cdot W_o \cdot R'(Z_h)$
Cost-Weights Deriv	$LayerError \cdot LayerInput$

layer which is actually dot product of activations times delta, where delta is errors * activation's derivative.



So the main part is training. For training we need a dataset which has inputs and target values. Epochs stands for number of the iteration cycle for training and we need to give some learning rate. Learning rate is actually step size for each iteration, as you can recall from the one paragraph above, we update weights by multiplying the derivatives with learning rate. Learning rate and epochs are just hyperparameters for the training. It is acceptable that learning rate being between 0.1-1.0 range. If learning rate is bigger, you can miss the minima which you try to achieve with gradient descent algorithm. And as the number of epochs gets bigger, you learn better, but there are some tradeoff, for example since our dataset is big, I let the epochs smaller because it took time to evaluate and my kernel was not really prepared for this kind of event.



I added a classification trick for the end, after training I put output to an averaging boundaries just to ensure it splits the predictions into the classes. For this purpose, I used my heuristics and created a virtual decision boundary for the rice data. This part may change according to your dataset (if you are reading this from Github, because this is where I will put my homework afterwards.) So, it may be better to optimize this classifier by the training samples, at the end of the evaluation. I will be upgrading this code before adding it to repository to make it automated.

[1]:
```
class NeuralNetwork(object):

    #Let's create our network:
    #By default 3 neurons for inputs, 2 hidden layers with 3 and 5 neurons, and␣
 ↪2 for ouputs

    def __init__(self, inputLayer=3, hiddenLayers=[3, 5], lastLayer=2):


        self.inputLayer = inputLayer
        self.hiddenLayers = hiddenLayers
        self.lastLayer = lastLayer
        layers = [inputLayer] + hiddenLayers + [lastLayer]


        #random weights initialization
        #weight list with the weights
        weights = []
        #creating w matrix with random arrays with 2 dimensions
        #2d array w with current layer and subsequent layer's neurons
        for i in range(len(layers) - 1):
            w = np.random.rand(layers[i], layers[i + 1])
            weights.append(w)
        self.weights = weights

        #adding 1 dimension for each layer
        activations = []
        for i in range(len(layers)):
```

```
            a = np.zeros(layers[i])
            activations.append(a)
        self.activations = activations


        #we need derivatives
        #but we need only layers-1 derivatives, because the last one does not␣
↪need it
        #2d derivatives array will have number of neurons of the layer and␣
↪subsequent layer's neurons
        derivatives = []
        for i in range(len(layers) - 1):
            d = np.zeros((layers[i], layers[i + 1]))
            derivatives.append(d)
        self.derivatives = derivatives

    def forwardProp(self, inputs):

        #first layer's activation is its inputs
        #activations will be stored as list to conclude derivatives afterwards

        activations = inputs

        # storing these for backpropogation purposes
        self.activations[0] = activations

        #So we need to use loop for iteration
        #we need to calculate net inputs as we had done in perceptron
        #and then calculate the activation

        for i, w in enumerate(self.weights):
            net_inputs = np.dot(activations, w)
            activations = self._sigmoid(net_inputs)
            self.activations[i + 1] = activations
        return activations

    def backProp(self, error):
        #this is where it is important.
        #we need to iterate from right to left
        #therefore, we will use reversed function for iteration
        for i in reversed(range(len(self.derivatives))):

            #activation is got from previous layer to next
            activations = self.activations[i+1]
            delta = error * self._sigmoid_derivative(activations)

            #we need to reshape activations and delta because
```

```
            #we need 2d array and get the delta's transpose
            #because of the dot product purposes
            delta_re = delta.reshape(delta.shape[0], -1).T

            # get activations for current layer
            current_activations = self.activations[i]
            current_activations = current_activations.
↪reshape(current_activations.shape[0],-1)

            #so we need to store derivatives as well
            self.derivatives[i] = np.dot(current_activations, delta_re)
            #and the error is dot product of delta and weights
            #this way it backpropagates
            error = np.dot(delta, self.weights[i].T)

    def gradientDescent(self, learningRate=0.4):
        # update the weights by stepping down the gradient
        for i in range(len(self.weights)):
            weights = self.weights[i]
            derivatives = self.derivatives[i]
            weights += derivatives * learningRate


    def train(self, inputs, targets, epochs=50, learningRate=0.2):
        #epochs and learning rate are hyperparameters
        #training is similar to perceptron but needs forward feed
        #backpropagation and gradient descent as well
        for i in range(epochs):
            sumErrors = 0
            #we will loop through the inputs and its indexes
            for j, input in enumerate(inputs):
                target = targets[j]
                #activating the layer by forward propagation
                output = self.forwardProp(input)
                #calculating the error
                error = target - output

                self.backProp(error)

                # now perform gradient descent on the derivatives
                # (this will update the weights)
                self.gradientDescent(learningRate)
                sumErrors += self._mse(target, output)
                #we can report using sumErrors at each epoch to
                #check the error, but I did not display it as the video␣
↪suggested.
```

```
        print("Trained the model.")

    #A little trick into classification, I had normalized the output
    #by average and created a virtual decision boundary for the rice data
    #this decision function may change according dataset you will try out.
    #it may be better to optimize the classifier to be automated one by the␣
↪training
    #samples, but I had used heuristics and manually set the function.

    def classification(self,output):
        return np.where(output/np.average(output) < 1, 1, -1)

    #And these are the mathematical expressions for the
    #mean squared error, sigmoid function and sigmoid's derivative

    def _mse(self, target, output):
        return np.average((target - output) ** 2)


    def _sigmoid(self, x):
        y = 1.0 / (1 + np.exp(-x))
        return y


    def _sigmoid_derivative(self, x):
        return x * (1.0 - x)
```

So after all the code segment, we need to engage our training set and test set. In this part, I recommend you to use cross validation k-fold techniques to ensure the accuracy and avoid over-fitting issues, which is explicitly explained in the Machine Learning Mastery link I just shared above. However, I just used my X_normed sample from Ricetrain.csv and X_test_normed from Ricetest.csv files.

First I tried the model with 1 hidden layer with 5 neurons inside and the trained with 30 epochs and on 0.1 learning rate. 30 epochs are very low, but my laptop's hardware was like on fire, so I just kept it between 30-100 range. Then after training, I used the model on X_test_normed and results were 84,6% accurate, which is pretty good for 1 hidden layer and 30 epochs.

[307]:
```
# Adding datasets for training the network
items = X_normed
targets = y

#Creating the 1 hidden layered network with 5 neurons inside
#we want to get the first layer's size as the number of the features
inputSize = items.shape[1]
nn = NeuralNetwork(inputsize, [5], 1)
nn.train(items, targets, 30, 0.1)
```

```
#After training, adding the test data
input = X_test_normed
target = y_test

#Predictions and classifications.
output = nn.forwardProp(input)
result = nn.classification(output).T
print(result)
```

```
Trained the model.
[[ 1  1 -1 ... -1 -1  1]]
```

[308]:
```
dataset = pd.DataFrame({'Predicted': result[len(result)-1], 'Actual': target})
dataset['Score_diff'] = dataset['Predicted'].sub(dataset['Actual'], axis = 0)
dataset["Score_diff"] = np.where(dataset["Score_diff"] == 0 , 0 , 1)
```

[309]:
```
dataset
column_sums = dataset.sum(axis=0)
accuracy = 1 - (column_sums[2] / target.shape[0])
print("Accuracy is:", accuracy*100, "%.")
```

```
Accuracy is: 84.60192475940508 %.
```

Then I tried to increase the number of epochs to try out the new accuracy. When we increase the number of epoch, we expect it to learn better. And it went OK, accuracy was increased to 85,3% as we expected.

[310]:
```
#Let's check different epochs
nn = NeuralNetwork(7, [5], 1)
nn.train(items, targets, 50, 0.1)

# create dummy data
input = X_test_normed
target = y_test

# get a prediction
output = nn.forwardProp(input)
result = nn.classification(output).T
print(result)
dataset = pd.DataFrame({'Predicted': result[len(result)-1], 'Actual': target})
dataset['Score_diff'] = dataset['Predicted'].sub(dataset['Actual'], axis = 0)
dataset["Score_diff"] = np.where(dataset["Score_diff"] == 0 , 0 , 1)
dataset
column_sums = dataset.sum(axis=0)
accuracy = 1 - (column_sums[2] / target.shape[0])
print("Accuracy is:", accuracy*100, "%.")
```

```
Trained the model.
```

```
[[ 1  1 -1 ... -1 -1 -1]]
Accuracy is: 85.30183727034121 %.
```

Then I added various differences, for example changed learning rate and accuracy was decreased, added hidden layers with more neurons, accuracy increased but take much more time. Therefore, I am just adding one more example output for the accuracy, with additional 2 hidden layers(3 in total) and accuracy increased to 86,87%, which is expected again.

As you can see, on each improvement, we had more accurate values but complexity got higher as well. So, it is better to consider these trade-offs and improve the code for efficiency. If we did not normalized the data or used sigmoid function, the outcome would be much more chaotic.

[327]:
```
#Let's check different epochs
nn = NeuralNetwork(7, [5,2,4], 1)
nn.train(items, targets, 100, 0.1)

# create dummy data
input = X_test_normed
target = y_test

# get a prediction
output = nn.forwardProp(input)
result = nn.classification(output).T
print(result)
dataset = pd.DataFrame({'Predicted': result[len(result)-1], 'Actual': target})
dataset['Score_diff'] = dataset['Predicted'].sub(dataset['Actual'], axis = 0)
dataset["Score_diff"] = np.where(dataset["Score_diff"] == 0 , 0 , 1)
dataset
column_sums = dataset.sum(axis=0)
accuracy = 1 - (column_sums[2] / target.shape[0])
print("Accuracy is:", accuracy*100, "%.")
```

```
Trained the model.
[[ 1  1 -1 ... -1 -1  1]]
Accuracy is: 86.8766404199475 %.
```

[328]:
```
dataset["Predicted"] = np.where(dataset["Predicted"] == -1, "Osmancik" ,"Cammeo")
dataset["Actual"]  = np.where(dataset["Actual"] == -1 , "Osmancik", "Cammeo")
dataset["Score_diff"] = np.where(dataset["Score_diff"] == 0 , "Correct" ,␣
 ↪"Wrong")
dataset.head(n=20)
```

[328]:

	Predicted	Actual	Score_diff
0	Cammeo	Cammeo	Correct
1	Cammeo	Cammeo	Correct
2	Osmancik	Osmancik	Correct
3	Osmancik	Osmancik	Correct
4	Osmancik	Osmancik	Correct

```
5   Osmancik  Osmancik    Correct
6   Osmancik  Osmancik    Correct
7     Cammeo    Cammeo    Correct
8   Osmancik  Osmancik    Correct
9     Cammeo    Cammeo    Correct
10    Cammeo    Cammeo    Correct
11  Osmancik  Osmancik    Correct
12    Cammeo    Cammeo    Correct
13    Cammeo    Cammeo    Correct
14    Cammeo    Cammeo    Correct
15    Cammeo    Cammeo    Correct
16    Cammeo  Osmancik      Wrong
17    Cammeo    Cammeo    Correct
18  Osmancik  Osmancik    Correct
19  Osmancik  Osmancik    Correct
```

As a conclusion, you can see the evaulation of the Predicted and Actual classifications of the rice grains. There could be better approaches, but since it is neural network from scratch which only uses numpy and math functions, we can say it is nice. There are many more efficient machine learning framework which only requires preprocessing of the data only. So, in the next repository updates besides the homework commits, I will be sharing Keras, Tensorflow applications on Iris dataset and Rice dataset on Github for the beginners of machine learning -just like me.

For now, you know what is activation function, how to iterate the values to next layers as forward propagation and how to update weights with back propagation and using gradient descent algorithm. The algorithm works on 85+% accuracy for the Rice dataset, but it can always be improved by using different data preprocessing sessions, different evaluation mechanisms, even using ReLU for activation function instead of Sigmoid may also affect the output. You can upgrade your function by setting some penalization which is also known as regularization for the features. They are really good tricks to know about and have a solid mathematical background to learn.

4 Bibliography

- Sagar Sharma, Activation Functions on Neural Networks: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6
- Avinash Sharma V, Understanding Activation Functions in Neural Network: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0
- IBM, Neural Networks: https://www.ibm.com/cloud/learn/neural-networks
- MIT, Larry Hardesty, Explained: Neural Networks:https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414
- Jason Brownlee, How to Code Neural Network with Backpropagation in Python (from scratch): https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/
- Valerio Velardo, Training a Neural Network 6-7-8: https://www.youtube.com/watch?v=Z97XGNUUx9o
- Github, mattm, Simple Neural Network: https://github.com/mattm/simple-neural-network
- Github, emilwallner, Deep Learning From Scratch: https://github.com/emilwallner/Deep-Learning-From-Scratch
- Github, KarnageKnight, Neural Network with n Hidden Layers: https://github.com/KarnageKnight/Neural-Network-with-n-hidden-layers
- Github, JGuymont, Numpy Multilayer Perceptron: https://github.com/JGuymont/numpy-multilayer-perceptron
- Github, musikalkemist, Deep Learning for Audio with Python (my main code reference):https://github.com/musikalkemist/DeepLearningForAudioWithPython

[ ]: