Northeastern University
College of Professional Studies

Module 4 Assignment

Shivam Chauhan

January 23, 2020

Class: ALY 6020 – Predictive Analytics

Professor: Dr. Marco Montes de Oca

Topic: Feedforward neural networks for BCD adder.

Introduction

Feedforward neural networks are artificial neural networks where the connections between units do not form a cycle. Feedforward neural networks were the first type of artificial neural network invented and are simpler than other counterpart neural networks. They are called feedforward because information only travels forward in the network (no loops), first through the input nodes, then through the hidden nodes (if present), and finally through the output nodes. (Brilliant.org,2020)

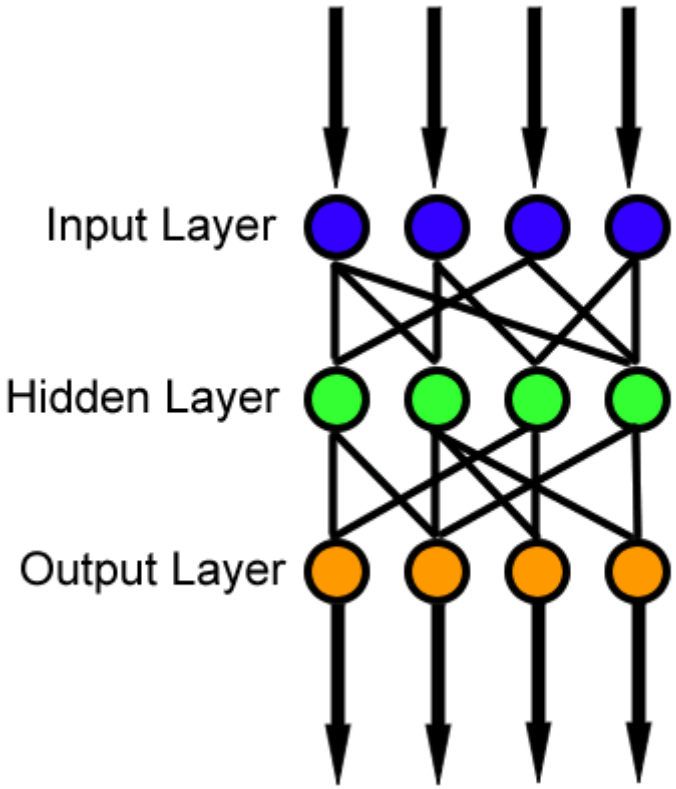
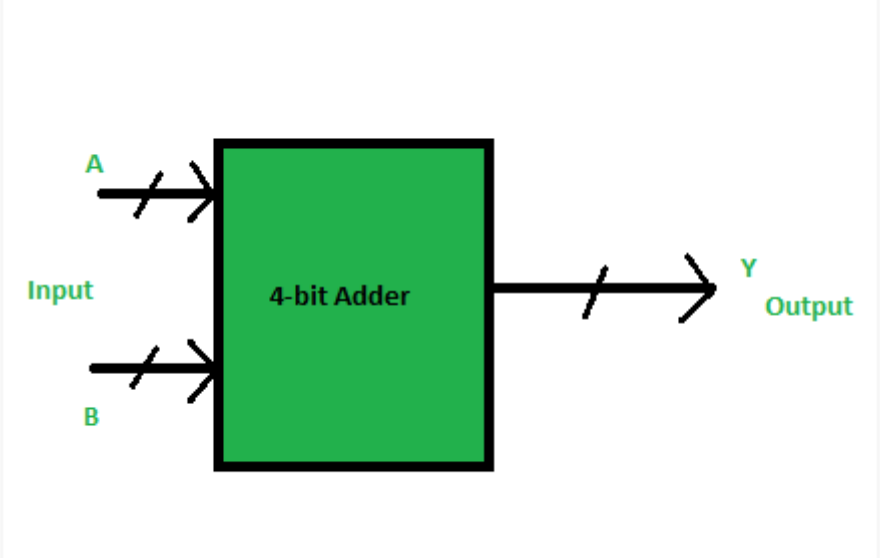


Fig: Feed Forward Neural Networks

The concept of Neural Network is that the algorithm will learn when the information is passed from the neurons. The input variables will pass through neurons, in other words it will pass through a function called activation function. Input for this activation functions is the product of randomly generated weights and the value of input variable. These activation function will convert the product value into a value between 0 to 1(in this case) using sigmoid function and will activate the neuron if the threshold is crossed. The output of this layer will be multiplied with respective random weights, and that product will be the input for the next hidden or output layer. There is no rule how many hidden layer and number of neurons should be there. Then the cost/loss function will calculate the square difference of the output and actual prediction. With this value of cost/loss function, we will update the random weights to a value in such a way that it will minimize the value of loss function.

Goal

The objective of this assignment is to train a feedforward neural network to implement a binary BCD (binary coded decimal) adder. This model will take 8 bits as input (4 bits per digit) and transform it into a 5-bit output.



The value of A and B can vary from 0(0000 in binary) to 9(1001 in binary) because we are considering decimal numbers. The output will vary from 0 to 18.

Digits	Binary Sum (Input)					BCD Sum (Output)				
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	1	0	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0
9	0	1	0	0	1	0	1	0	0	1
10						1	0	0	0	0
11						1	0	0	0	1
12						1	0	0	1	0
13						1	0	0	1	1
14						1	0	1	0	0
15						1	0	1	0	1
16						1	0	1	1	0
17						1	0	1	1	1
18						1	1	0	0	0

We will create 2 Neural Network without any library and 1 neural network using TensorFlow library:

1. Simple neural Network with 2 layers
2. Simple Neural Network with 3 layers
3. Neural Network with multiple hidden layers using tensorflow.keras.

Data Preparation

First, I have created a truth table that represents the mapping between all the possible inputs and desired outputs which we want our neural network to learn.

```
#library used
import pandas as pd
import numpy as np


""" loading the data and seperating the input and output data. Saving it as numpy array """
df_og = pd.read_excel("bcd.xlsx", set_index = 0,axis=1,header = None )
df_sub = df_og.drop(0,axis =1)
df_bcd = df_sub.iloc[:10,:5]
df_out = df_sub.iloc[:,6:]
df_bcd2 = df_bcd.copy(deep=True)
np_bcd = np.array(df_bcd)
np_bcd2 = np.array(df_bcd2)
np_out = np.array(df_out)

""" loop that will create required truth table """
final = []
for i in range(len(np_bcd)):
    for j in range(len(np_bcd2)):
        k=0
        k =i+j
        final.append(np_bcd[i,:])
        final.append(np_bcd2[j,:])
        final.append(np_out[k,:])


 """converting the truth table in np array """
final_np = np.array(final)
df_np = np.reshape(final_np,1500)
df = np.reshape(df_np, (-1,15)
df_tt = pd.DataFrame(df)
df_tt = df_tt.drop([0, 5], axis=1)

 """converting the truth table in np array """
final_np = np.array(final)
df_np = np.reshape(final_np,1500)
df = np.reshape(df_np, (-1,15)
df_tt = pd.DataFrame(df)
df_tt = df_tt.drop([0, 5], axis=1)

""" splitting the data into inpit and output as numpy array """
X = df_tt.iloc[:,:8]
Y = df_tt.iloc[:,8:]

x_np = np.array(X)

y_np = np.array(Y)


# input dataset
X_input = x_np
# output dataset
y_output = y_np
```

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1

Fig: Input variable with 8 bits. First 4 columns are for first digit and next 4 columns are for the second digits. The output of addition of these two numbers are below table in y_output.

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1

Fig: Output variable with 5 bits. These are out put of the addition of two numbers in a row in X_input

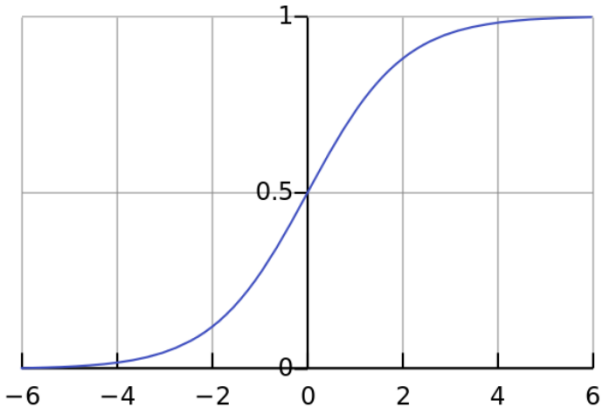
We have **100** possible inputs and desired outputs which we want.

Creating the activation function (Sigmoid).

```
# sigmoid function
def nonlin(x,deriv=False):
    if(deriv==True):
        return x*(1-x)
    return 1/(1+np.exp(-x))
```

(I am trask, 2020) This function will map the values between 0 and 1. We use it to convert numbers to probabilities. It also has several other desirable properties for training neural networks. This function can also generate the derivative of a sigmoid (when deriv=True). One of the desirable properties of a sigmoid function is that its output can be used to create its derivative. If the sigmoid output is a

variable "out", then the derivative is simply out * (1-out). This will help us to know the slope of a sigmoid function at a given point.



2 layers Neural Network

Input Layer (8 node) → Output layer (5 node)

```
# seed random numbers to make calculation
# deterministic (just a good practice)
np.random.seed(1)

# initialize weights randomly with mean 0
syn0 = 2*np.random.random((8,5)) - 1
```

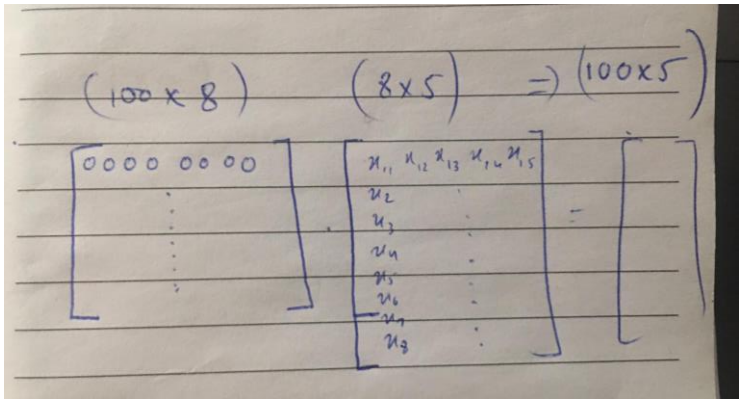
Since we only have 2 layers (input and output), we only need one matrix of weights to connect them. Its dimension is (8,5) because we have 8 inputs and 5 output.

```
for iter in range(10000):

    # forward propagation
    l0 = X_input
    l1 = nonlin(np.dot(l0,syn0))

    # how much did we miss?
    l1_error = y_output - l1

    # multiply how much we missed by the
    # slope of the sigmoid at the values in l1
    l1_delta = l1_error * nonlin(l1,True)

    # update weights
    syn0 += np.dot(l0.T,l1_delta)

print("Output After Training:")
print(l1)
```

This iteration will begin to train our neural network.

First layer (l0) takes the 8-bit input which is array with each row containing 8 values.

Second layer converts the Dot product of the randomly generated weights into 0 to 1 value using the function nonlin(sigmoid function).



(100X8) matrix is input array, (8X5) matrix is our weights and (100X5) matrix is our output.

Then we have calculated the cost function for every input and output row.

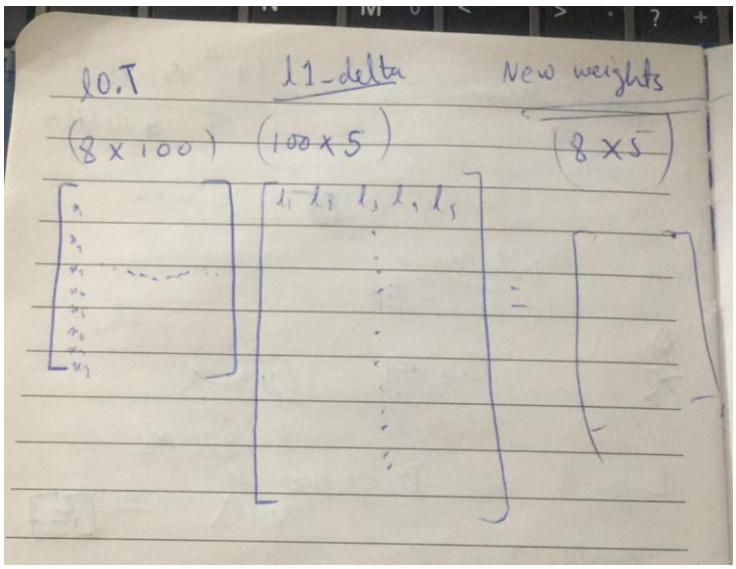
l1_error = y_output - l1

Now we will calculate the slope of the points generated by sigmoid function by using the nonlin(l1,True) function. Now we will multiplied this by the error we have got for that point. Now we will get a tiny number li_delta which is the average.

We have got the derivate multiplied by the error. We can add that value to the existing weights. IF l1_delta is negative, it will make the existing weight smaller. This means it wants that neuron to be less significant.

If the l1_delta value is positive, then it will make existing weight greater. This means we want neuron to be more significant.

This will be repeated for 10000 times, data will be moving from 8 input to 5 output and weights will be updated.



The new (8X5) matrix will be added to syn0 matrix for every iteration.

Results of this trained neural network are L1.

So, after this network updates, we look back at our training data and reflect. When both an input and output are 1, we increase the weight between them. When an input is 1 and an output is 0, we decrease the weight between them.

0.5	0.5	0.5	0.5	0.5

For example, if we look at the first out which is sum of digit 0 and digit 0. The answer should be 00000. We can see the weights are 0.5.

1	0.241693	0.39573	0.199222	0.0859151	0.759571

If we look at the first out which is sum of digit 0 and digit 1. The answer should be 00001. We can see the weights of all the first 4 binary digits are close to 0 except for the last binary digit. Last binary digit is close to 1. This shows that model has learned.

	0	1	2	3	4
0	0.5	0.5	0.5	0.5	0.5
1	0.241693	0.39573	0.199222	0.0859151	0.759571
2	0.316973	0.316312	0.290873	0.329491	0.57207
3	0.128854	0.232533	0.0925984	0.0441482	0.808552
4	0.653322	0.402766	0.414032	0.23317	0.57207
5	0.375254	0.30635	0.149506	0.0277856	0.808552
6	0.466539	0.23781	0.224702	0.129997	0.641207
7	0.217983	0.169663	0.0672555	0.0138497	0.849532
8	0.994845	0.413512	0.443421	0.495978	0.451653
9	0.984003	0.315883	0.165418	0.0846599	0.722387
10	0.241693	0.39573	0.199222	0.0859151	0.759571
11	0.0922192	0.300151	0.0582867	0.0087568	0.908931
12	0.128854	0.232533	0.0925984	0.0441482	0.808552
13	0.0450214	0.16557	0.0247595	0.00432239	0.930277
14	0.375254	0.30635	0.149506	0.0277856	0.808552
15	0.160682	0.224344	0.0419008	0.00267902	0.930277

Fig: L1 showing the weights. The results of Neural Networks.

3 Layer Neural Network

input Layer (8 node) → Hidden layer (5 node) → output layer (5 node)

```
np.random.seed(1)

# initialize weights randomly with mean 0
syn0 = 2*np.random.random((8,5)) - 1
syn1 = 2*np.random.random((5,5)) - 1

for iter in range(100000):

    # forward propagation
    l0 = X_input
    l1 = nonlin(np.dot(l0,syn0))

    l2 = nonlin(np.dot(l1,syn1))

    # how much did we miss the target value?
    l2_error = y_output - l2

    if (iter % 10000) == 0:
        print("Error:" + str(np.mean(np.abs(l2_error))))

    # in what direction is the target value?
    l2_delta = l2_error*nonlin(l2,deriv=True)

    # how much did each l1 value contribute to the l2 error
    l1_error = l2_delta.dot(syn1.T)

    # in what direction is the target l1?
    l1_delta = l1_error * nonlin(l1,deriv=True)

    syn1 += l1.T.dot(l2_delta)
    syn0 += l0.T.dot(l1_delta)
```

In this case we are uses the "confidence weighted error" from l2 to establish an error for l1. To do this, it simply sends the error across the weights from l2 to l1.

This gives what you could call a "contribution weighted error" because we learn how much each node value in l1 "contributed" to the error in l2. This step is called "backpropagating".

We then update syn0 using the same steps we did in the 2 layers implementation.

We have also calculated the error for every 10000 loop.

```
Error:0.5055254194908523
Error:0.2716479383936254
Error:0.2690949914222323
Error:0.26825714366660697
Error:0.2674809884659815
Error:0.2670960994974095
Error:0.26673728199793095
Error:0.26659056721557645
Error:0.26631508076157673
Error:0.26614813788016406
```

We can observe that at first the error rate decreased continuously but after looping 20000 times the loss function is expected to reach near local minimum and error is not decreasing in a faster rate. Still we can see little improvement in the error till the last iteration.

	0	1	2	3	4
0	9.11926e-42	1.6082e-33	0.151092	0.00241212	7.7317e-15
1	4.64963e-66	7.68365e-41	0.0643606	4.66424e-06	0.991673
2	3.38771e-06	0.00268661	0.480192	0.445658	5.73122e-21
3	8.98748e-29	0.0410983	0.66268	0.619626	1
4	0.00179071	0.05166	0.490394	0.472838	8.89249e-11
5	8.98744e-29	0.0410986	0.66268	0.619626	1
6	0.0488791	0.203265	0.49549	0.48724	1.91382e-05
7	9.78841e-29	0.0412631	0.662481	0.619474	1
8	0.0398789	5.30657e-26	0.279745	0.537475	3.56858e-40
9	1.55488e-23	6.47509e-54	0.0244245	1.79306e-05	0.993216
10	2.33683e-36	1.79509e-56	0.02044	7.4395e-06	0.991558
11	1.06694e-22	5.27512e-46	0.0210425	3.55074e-06	2.21051e-35

Fig: L2 which are the results and weight of the 3 layers neural networks.

We can see some improvement in these results when compared to 2 layers neural networks. The one which are expected to be 0 are shrink nearly to 0.

The one which should be 1, have higher weights than other ranging from 0.4 approx. to 0.99 approx.

Neural Network using Keras

```
#library used
import keras
from keras.models import Sequential
from keras.layers import Dense
#from keras.layers import LeakyReLU,PReLU,ELU
from keras.layers import Dropout


NN_model = Sequential()
# The Input Layer :
NN_model.add(Dense(8, activation='sigmoid'))
# The Hidden Layers :
NN_model.add(Dense(7,activation='sigmoid'))
#NN_model.add(Dropout(0.2))
# The Output Layer :
NN_model.add(Dense(5,activation='sigmoid'))

# Compile the network :
NN_model.compile(loss='mean_absolute_error', optimizer='adam', metrics=['accuracy'])
NN_model.summary()

# Fitting the ANN to the Training set
model_history=NN_model.fit(X_input, y_output, nb_epoch = 5000)
```

Initially, we declare the variable and assign it to the type of architecture we'll be declaring, which is a “Sequential()” architecture in this case.

Next, we directly add layers the NN_model.add(). The type of layer can be imported from tf.keras.layers.

We use adam function as an optimizer and mean absolute error function as parameters for our architecture.

Once the model is defined, the next step is to start the training process for which we will be using the NN_model.fit() method.

The summary will be done on the test dataset which can be called using NN_model.summary() method.

```
100/100 [==============================] - 0s 120us/step - loss: 0.1570 - acc: 0.7100
Epoch 4998/5000
100/100 [==============================] - 0s 80us/step - loss: 0.1569 - acc: 0.7100
Epoch 4999/5000
100/100 [==============================] - 0s 80us/step - loss: 0.1569 - acc: 0.7100
Epoch 5000/5000
100/100 [==============================] - 0s 80us/step - loss: 0.1569 - acc: 0.7100

In [315]: NN_model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_47 (Dense)             (None, 8)                 72
_________________________________________________________________
dense_48 (Dense)             (None, 7)                 63
_________________________________________________________________
dense_49 (Dense)             (None, 5)                 40
=================================================================
Total params: 175
Trainable params: 175
Non-trainable params: 0
```

After creating a neural network using the keras library, I have observed minimum “mean square error” for this neural network above with shape of 8 X 7 X 5. Accuracy is 71% after 5000 epochs. Optimization function ‘adam’ is also used in this neural network.

Conclusion

Deep learning is an area of computer science with a huge scope of research. There are no general rules that dictate how to design a network for a task. We have seen how Neural Network feedforward works. Later we saw how backpropagation can improve the learning by updating the weights within the networks. We experimented a powerful library tensordlow.keras to build different neural networks and with the use of optimization techniques we saw improvements in the results. We saw how these neural networks learned the patterns to train this model to implement a binary BCD (binary coded decimal) adder.

Reference:

A Neural Network in 11 lines of Python (Part 1) - i am trask. (2020). Iamtrask.github.io. Retrieved 8 February 2020, from https://iamtrask.github.io/2015/07/12/basic-python-network/

A Quick Primer on Feedforward Neural Networks. (2019). Built In. Retrieved 8 February 2020, from https://builtin.com/data-science/feedforward-neural-network-intro

Feedforward Neural Networks | Brilliant Math & Science Wiki. (2020). Brilliant.org. Retrieved 8 February 2020, from https://brilliant.org/wiki/feedforward-neural-networks/

Attachments:


3 layer NN.py


2 layer NN.py


week 4 assignment ANN 02062020.py