Feedforward Neural Network for Classification

Assignment - 4 | Pattern Recognition

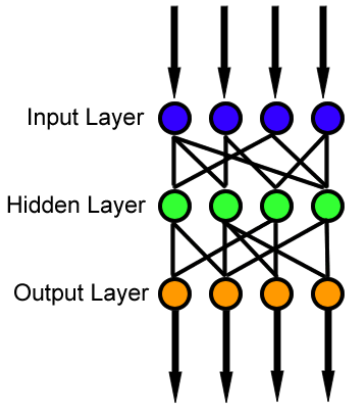
RAMIT NANDI || MD2211

2023-11-14

Contents

INTRODUCTION

A feedforward neural network (FNN) is an artificial neural network, where the information in the model flows in only one direction — from the input nodes, through the hidden nodes (if any) and to the output nodes, without any cycles or loops.



Description

Here we will create a FNN , having

- required number of input & output nodes for a dataset
- two hidden layer with ReLU activation function for non-linearity
- softmax activation in output layer for classification problem

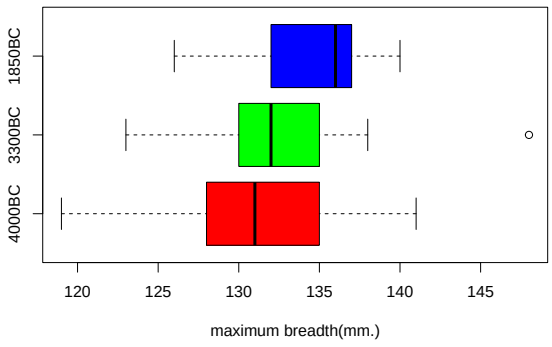
```
import tensorflow as tf

class My_NeuralNetwork(tf.Module):
    def __init__(self,num_Inputs,num_Outputs,num_nodesHidden=None):
        # some rule of thumb regarding the width of hidden layer ...
        if num_nodesHidden is None:
          num_nodesHidden = round(2*num_Inputs/3) + num_Outputs
        # connecting input layer to hidden layer 1...
        self.W0 = tf.Variable(tf.random.normal([num_Inputs,num_nodesHidden]))
        self.b0 = tf.Variable(tf.zeros([num_nodesHidden]))
        # connecting hidden layers 1 & 2
```

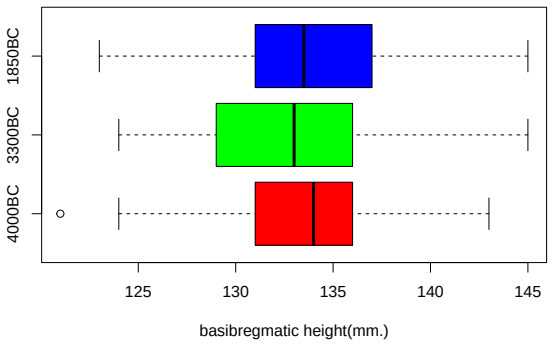
```
        self.W1 = tf.Variable(tf.random.normal([num_nodesHidden,num_nodesHidden]))
        self.b1 = tf.Variable(tf.zeros([num_nodesHidden]))
        # connecting hidden layer 2 to output layer ...
        self.W2 = tf.Variable(tf.random.normal([num_nodesHidden,num_Outputs]))
        self.b2 = tf.Variable(tf.zeros([num_Outputs]))

    def __call__(self, x):
        # ReLU activation for the hidden layers ...
        x = tf.nn.relu(tf.matmul(x, self.W0) + self.b0)
        x = tf.nn.relu(tf.matmul(x, self.W1) + self.b1)
        # SoftMax activation for the output layer ...
        return tf.nn.softmax(tf.matmul(x, self.W2) + self.b2)
```

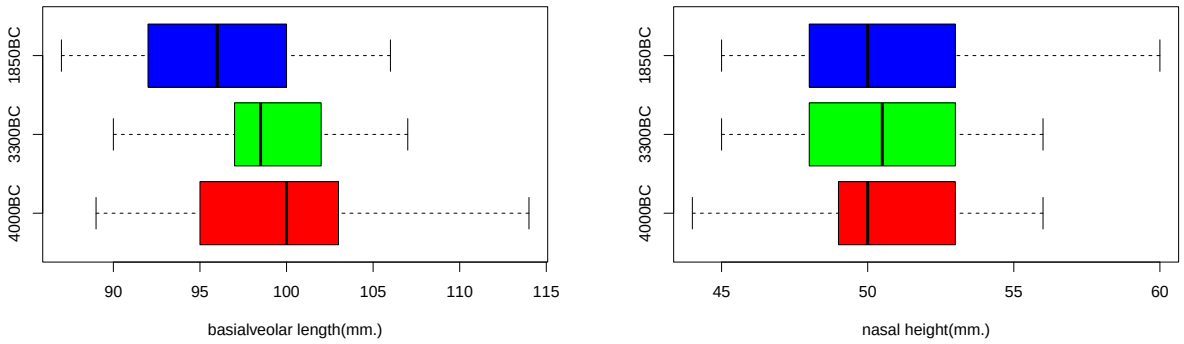
DATA

Here we will use the Egyptian-skull dataset, consists of anthropometric measurements on 90 male Egyptian skulls from three different periods(30 skulls/group). The data looks like -

	mb	bh	bl	nh	period
1	131	138	89	49	4000BC
30	124	138	101	46	4000BC
31	124	138	101	48	3300BC
60	130	128	101	51	3300BC
61	137	141	96	52	1850BC
90	138	133	91	46	1850BC







We have already seen earlier that, due to the overlapping nature of the data, it is very hard to classify them using methods like- LDA, QDA, RDA, SVM, Logistic Regression etc. Lets try Neural Network here.

Fitting the Model

We will split the data in 3-folds ,

- Based on 2-folds of data the Neural Network will be fitted by minimizing 'categorical crossentropy loss' i.e. maximizing the multinomial log-likelihood.
- To prevent overfitting we add L2 penalty
- The loss function will be evaluated on the remaining holdout fold of the data also, to get idea about the performance on new data.

Then we will repeat the entire process 3-times , finally we will compute the average accuracy.

```
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import accuracy_score

folds = RepeatedStratifiedKFold(n_repeats=3,n_splits=5,random_state=101)
folds = list(folds.split(X_data,y_data))

def fit_FNN(MODEL,
            X_train,y_train,X_validation,y_validation,
            lr=0.01,num_epochs=100,L2_lambda=0.01):
    optimizer = tf.optimizers.Adam(learning_rate=lr)
    def total_loss(X,y):
```

```
        loss_type = tf.losses.SparseCategoricalCrossentropy(from_logits=False)
        y_hat = MODEL(X)
        L2W0 = tf.reduce_sum(tf.square(MODEL.W0))
        L2W1 = tf.reduce_sum(tf.square(MODEL.W1))
        L2W2 = tf.reduce_sum(tf.square(MODEL.W2))
        penaltyL2 = L2_lambda*(L2W0+L2W1+L2W2)
        return loss_type(y,y_hat) + penaltyL2
    loss_train = []
    loss_validation = []

    # training loop ....
    for epoch in range(num_epochs):
        with tf.GradientTape() as tape: loss = total_loss(X_train,y_train)
        loss_train += [loss.numpy()]
        loss_validation += [(total_loss(X_validation,y_validation)).numpy()]
        gradients = tape.gradient(loss, MODEL.trainable_variables)
        optimizer.apply_gradients(zip(gradients, MODEL.trainable_variables))
    loss_train += [(total_loss(X_train,y_train)).numpy()]
    loss_validation += [(total_loss(X_validation,y_validation)).numpy()]

    # accuracy scores ....
    y_train_hat = np.argmax(MODEL(X_train).numpy(),axis=1)
    train_accuracy = accuracy_score(y_train,y_train_hat)
    y_validation_hat = np.argmax(MODEL(X_validation).numpy(),axis=1)
    validation_accuracy = accuracy_score(y_validation,y_validation_hat)


    return {"loss_train_history": np.array(loss_train),
            "loss_validation_history": np.array(loss_validation),
            "train_accuracy": train_accuracy,
            "validation_accuracy": validation_accuracy}
```

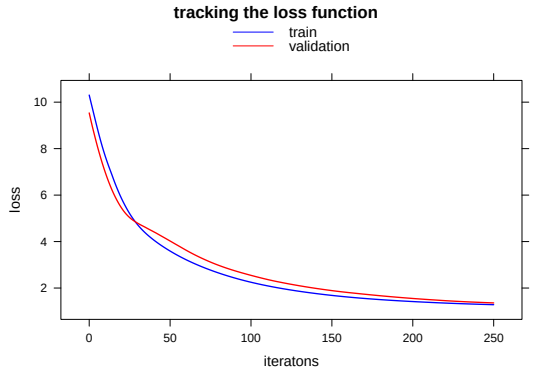
case: default number of nodes in hidden layer

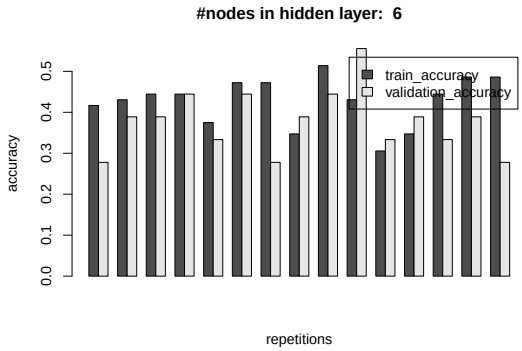
Start with the case , where we have default number of nodes in the hidden layer.

```
## Number of nodes: input= 4 | output= 3 | hidden= 6
```

```
accu_train = []
accu_validation = []

for ix_train,ix_validation in folds:
    current_Model = My_NeuralNetwork(num_Inputs,num_Outputs)
    OUT = fit_FNN(current_Model,
                  X_data[ix_train],y_data[ix_train],
```

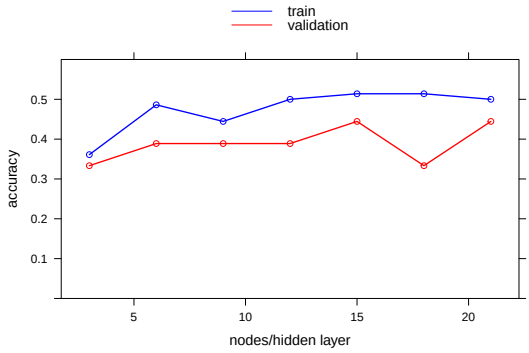
```
                X_data[ix_validation],y_data[ix_validation],
                num_epochs=250,L2_lambda=0.095)
    accu_train += [OUT["train_accuracy"]]
    accu_validation += [OUT["validation_accuracy"]]
```





optimum model

We will try with some different number of nodes in hidden layers , and try to choose the one that gives maximum accuracy on average , for validation set.



But don't see very much improvement there.

Conclusion:

This dataset is hard to classify for its overlapping nature.