ANN Classifier Assignment

Artificial Intelligence - Spring 2021

Thomas Trankle

February 26, 2021

Summary

This report describes the development of an artificial neural network to be used as a regression model for predicting the quality of various Portuguese white wines. Overall, the best model with a random selection resulted in an Mean Squared Error of 0.40 but the average Mean Squared Error produced throughout the evaluation was around 0.47. The introduction describes the modifiable parameters of this basic artificial neural network that is built using a Sequential model. Additionally, it describes the input features/columns used in predicting the quality of wine. The body of the report steps through necessary preprocessing procedures for the model and data, as well as the main adjustments made through the development and evaluation of the model. Throughout the paper, an assumption is made to chose a model with a faster run-time if it yields a similar Mean Squared Error.

Introduction

The purpose of this report is to provide a detailed explanation of creating a Artificial Neural Network to predict wine quality based on 11 features. This implementation uses a regression model since the output variable, which is the quality of the wine, is a score between 0 and 10 and we want to try produce a numerical value as close to those integer values as possible. The data set, which can be found here , contains 4,898 observations and 11 features, namely:

1. fixed acidity
2. volatile acidity
3. citric acid
4. residual sugar
5. chlorides
6. free sulfur dioxide
7. total sulfur dioxide
8. density
9. pH
10. sulphates
11. alcohol

The objective is to train and evaluate a neural network that will minimize our average prediction error of the output variable. To do this, I selected using Mean Squared Error as my objective function, otherwise known as loss, since it is very intuitive and common metric when evaluating the performance of a regression model. The formula is defined as follows,

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Additionally, I kept the Mean Absolute Error (MAE) as a metric throughout my model because it is also a very popular regression metric and I wanted to have an alternative metric to verify

the generalization of my model, that is make sure it wasn't solely focusing on optimizing the specified MSE loss function and have another metric to lookout for overfitting. When creating the model, there is no procedural method to define the specific parameters of our model to optimize our objective function. Instead, we have to experiment with the different parameters to determine what parameters, at least on average, can produce a sufficient result. To solve this black box architecture, we will have to tune the following parameters:

- Optimizer
- Number of Layers
- Number of Nodes per Layer
- Learning rate of the specified optimizer(left as default)
- Activation Function of each respective Layer
- Batch Size (specified in compile method)
- Number of Epochs

There are certainly more details and/or parameters that could be added to the above list, however, for the purpose of this exercise they were omitted since they are above the necessary scope. Additionally, a couple of common best practices and preprocessing steps were held constant when designing, tuning and evaluating the model. The first was checking to see that the data provided contained no *null* values, even though this information is specified in the data description. After this was verified, the next step was to split the data into **train** and **test** sets. Lastly, before we pass the data to out model, we center and scale the predictor variable (X features).

Model Development and Paramter Tuning

Preprocessing

The data is provided in a *.csv* file. However, the delimiter used to separate the columns is a semicolon. Therefore, the first step in the preprocessing phase was to split and clean the data to make sure it was in the appropriate format. To do this, I simply imported the data file, called *winequality-white.csv* using a very popular Python package called *pandas*. The package has a *read_csv* function that I used to load the data into my environment. Next I made use of an in-built Python tool, list comprehensions, to split the data into the appropriate index positions (as seen in the data). The predictor features are assigned to the *X* variable which is a list of list while the outcome is assigned to the *Y* variable that is just a single list.

```
X = np.array([[float(x) for x in dataframe.values[i][0].split(';')[0:11]]
                                        for i in range(len(dataframe))])
Y = [int(data.values[i][0].split(';')[-1]) for i in range(len(data))]
```

The package *keras* we use to create a neural network prefers to deal with an array data type as opposed to a list, in order to optimize its computational efficiency. So, we import and use another popular Python package called *numpy* to transform the X and Y variables into array data types and ensure they are both floating values.

```
X = np.array(X).astype('float32')
Y = np.array(Y).astype('float32')
```

Lastly, we use a common machine learning package called *sklearn* to split our data into train and test sets and import a normalizer called StandardScaler which centers and scales our input columns by removing the mean and scaling to unit variance

```
X, X_test, Y, Y_test = train_test_split(X,Y,test_size = 0.2)
scaler = StandardScaler()
X = scaler.fit_transform(X)
X_test = scaler.fit_transform(X_test)
```

Course of Action and Model Development

After all the preprocessing steps had been implemented, I decided to create an initial model to determine what range my number of nodes should be and to make sure I was able to pass all the data through in the correct format. To do so, a created a simple for loop to iterate through a range on nodes for my input layer and one hidden layer.

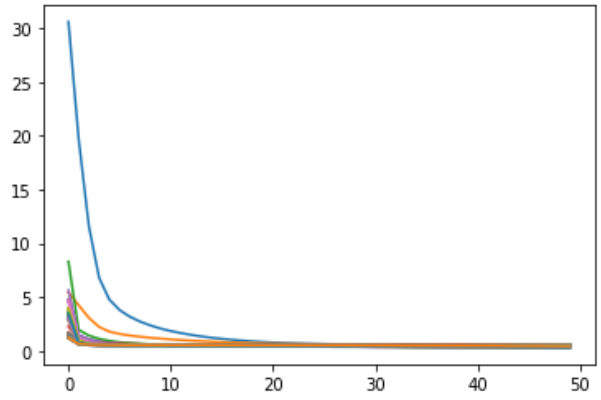
```
## All other parameters have been defined outside this code chunk ##
layer1_nodes=[32, 64, 128, 256]
layer2_nodes=[32, 64, 128, 256]

for l1n in layer1_nodes:
    for l2n in layer2_nodes:
        model = Sequential()
        model.add(Dense(l1n, input_dim=num_cols, activation = act_function))
        model.add(Dense(l2n,  activation = act_function))
        model.add(Dense(1, activation = 'linear'))
```

This gave me an average MSE of 0.593 using the optimizer RMSprop, 5 Epochs, ReLu activation function for all 3 layers, and default parameters for the remaining parameter inputs mentioned in the introduction. This loop took 1 minute and 4 seconds to run.

After creating an initial starting point, I wanted to determine what Epoch value to use going forward. So, I re-ran the above mentioned for loop with a few more node values and an the Epoch parameter set to 50. This code resulted in a average MSE 0.49 of and took 19 minutes and 10 seconds to run.

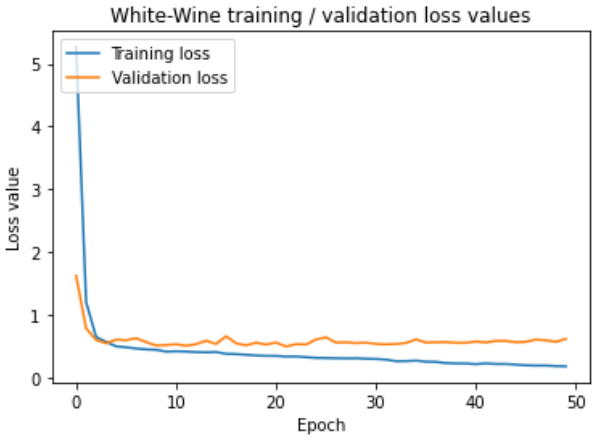
```
layer1_nodes = [11,32,64,128,254,512]
layer2_nodes = [0,32,64,128,254,512]
```



Interestingly, I noticed from this looping model that the higher nodes in a layer potentially result in a better model as I got a lower MSE as the iterations of the loop increased, that is an input and hidden layer with 512 nodes. The last iteration took 18.46 seconds to run. Additionally, I also derived from the above test that no matter the node size, it seems that most models perform well with 50 Epochs, that is, there is no obvious sign of overfitting. i continually used a similar graphical approach to validate the balance of my model between underfitting and overfitting going forward.

Next, I decided to explore the effect of adding more hidden layers to the model. To keep it simple initially, I added another four hidden layers each with 64 nodes and a ReLu activation function. This produced an MSE of 0.56901, which was an improvement in the right direction since our objective is to minimize MSE.

```
model = Sequential()
model.add(Dense(64, input_dim=num_cols, activation = 'relu'))
model.add(Dense(64,  activation = 'relu'))
model.add(Dense(64,  activation = 'relu'))
model.add(Dense(64,  activation = 'relu'))
model.add(Dense(64,  activation = 'relu'))
model.add(Dense(1, activation = 'linear'))
```



Next, I started randomly selecting different sized nodes for each layer since we saw earlier that

this could improve the model. I resulted in the following,

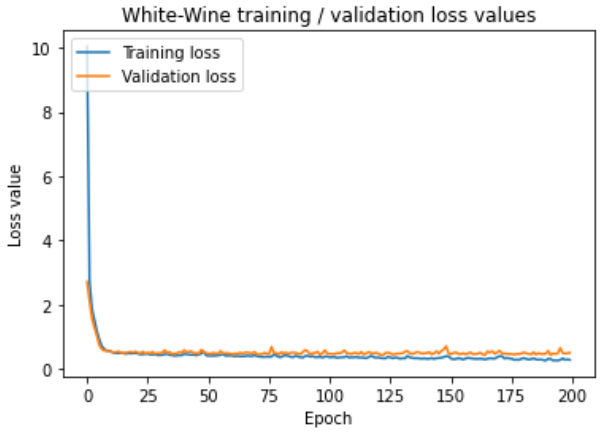
```
model = Sequential()
model.add(Dense(256, input_dim=num_cols, activation = 'relu'))
model.add(Dense(128,  activation = 'relu'))
model.add(Dense(64,  activation = 'relu'))
model.add(Dense(128,  activation = 'relu'))
model.add(Dense(256,  activation = 'relu'))
model.add(Dense(1, activation = 'linear'))
```

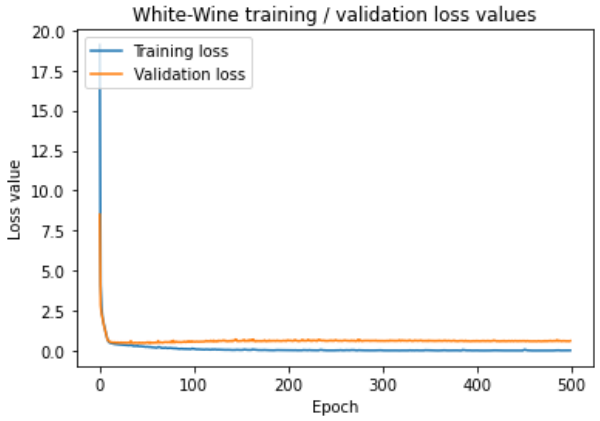
This gave me a similar MSE of 0.56. However, I noticed my validation loss and train loss started diverging from each other much earlier. After reading some of the documentation online and a comment you made about *batch_size* paramter I started to experiment with that value as it was potentially passing two small a sample and could be contributing to my current overfitting model. I then decided to go back to single hidden layer and change my batch size to 128.

```
batchSize = 128
model = Sequential()
model.add(Dense(256, input_dim=num_cols, activation = 'relu'))
model.add(Dense(128,  activation = 'relu'))
model.add(Dense(1, activation = 'linear'))
```

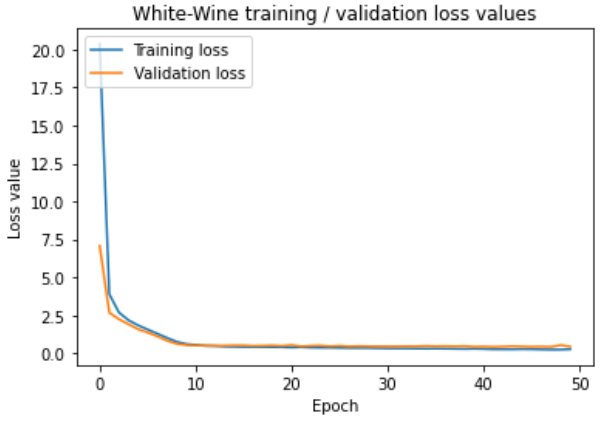
This model produced an MSE of 0.5072 and had a running time of 11.25 seconds. Next, I simply experimented with the different optimizers and settled on the Adam optimizer. Using the exact same model as abobe, except with Adam optimizer, I got an MSE of 0.4863 and it ran in 9.2 seconds.

I once more added more layers back into my model to make sure this option was still not valid. But, I actually ended up with a better model after changing the batch size once more to 256 since more nodes increases run-time. This became my best overall best model with a MSE of 0.45011 and a run-time of just over 7 seconds. Throughout my experimentation I settled on 50 epochs because at the amount my training and validation loss functions gradually decrease together. However, I decided to run a model with 200 Epochs and again with 500 Epochs to determine if I could improve the model by running it for longer. The 200 Epoch model had an MSE of 0.53 and a run-time of 25.18 seconds. The 500 Epoch model had an MSE of 0.61 and a run-time of 1 minute and 1 second. As you can see below, overfitting has become evident and begins to happen more severely after approximate 60 epochs. The scale of the graph make it seem minute but considering our output only ranges from 0 to 10; the difference matters.



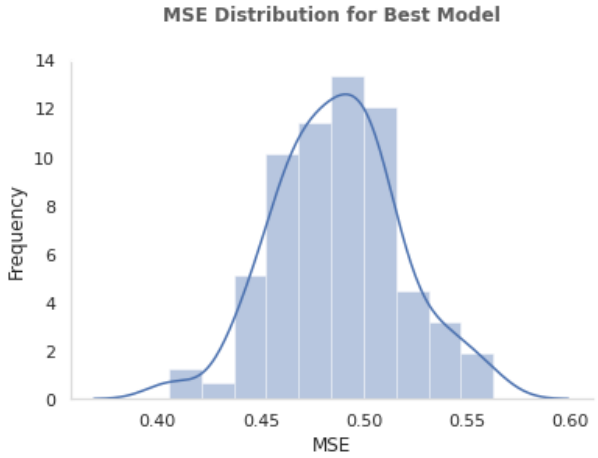


To keep it safe, but also have a decent number of Epochs, I returned it back to 50.



Conclusion

My final model had 4 hidden layers with 128, 64, 128 and 256 nodes respectively. The input and output layers had 256 and 1 layer respectfully. Note, it is necessary to only have one output node since we are trying to predict a single value. Each layer, besides the output layer, made use of the ReLu activation function. This function was chosen because it seemed to perform the best and is commonly used for regression models. The optimizer is selected was Adam, as it provided a better objective function and had less variance overall. Additionally, my final model had a batch size of 256 and 50 Epochs. To validate my model held in the general case, I ran it 100 times with complete randomization. I resulted with an average MSE of 0.48, a minimum MSE of 0.40 and a maximum MSE of 0.55. The distribution is shown in the figure below



Recommendations and Comments

I think it is possible to have different neural networks produce the same MSE. To test this hypothesis, I created another model which had only a 0.001 difference from my model on average. This model had fewer layers but a smaller batch size. However, my model ran quicker than this alternative and I think in general, it is better to have quicker model if it can produce the same distribution as a slower model.

Final Code

```
"""
Import Libraries Section
"""
import datetime
import numpy as np
import pandas as pd
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

"""
Parameters Section
"""
# first layer is # neurons and # number of inputs -- 10 in 1 out

csv_file='winequality-white.csv' #10 X 11 Y
num_cols=11
num_epochs=50
optimizer=keras.optimizers.Adam()
batchSize = 256

activation_func = 'relu'
"""
Load Data Section
"""
# Load dataset
dataframe = pd.read_csv(csv_file, header = 0)

"""
Pretreat Data Section
"""
dataframe = dataframe.replace(np.nan,0)

X = [[x for x in dataframe.values[i][0].split(';')[0:11]]
     for i in range(len(dataframe))]
Y = [dataframe.values[i][0].split(';')[-1] for i in range(len(dataframe))]
X = np.array(X).astype('float32')
Y = np.array(Y).astype('float32')


X, X_test, Y, Y_test = train_test_split(X,Y,test_size = 0.2)
scaler = StandardScaler()
```

```
X = scaler.fit_transform(X)
X_test = scaler.fit_transform(X_test)

#normalizer.adapt(np.array(X))
"""
Define Model Section
"""
start_time = datetime.datetime.now()

model = Sequential()
model.add(Dense(256, input_dim=num_cols, activation = activation_func))
model.add(Dense(128,  activation = activation_func))
model.add(Dense(64,  activation = activation_func))
model.add(Dense(128,  activation = activation_func))
model.add(Dense(256,  activation = activation_func))
model.add(Dense(1, activation = 'linear'))

#compile model
model.compile(loss = 'mse', optimizer = optimizer, metrics=['mse', 'mae'])

model.summary()
"""
Train Model Section
"""
# fix random number seed for repeatability
seed=6464
np.random.seed(seed)
history = model.fit(X, Y, epochs=num_epochs,
                      batch_size = batchSize, verbose=1 ,
                    validation_data=(X_test,Y_test))
score = model.evaluate(X_test, Y_test, verbose=1)

"""
Show output Section
"""
stop_time = datetime.datetime.now()

plt.plot(history.history['mse'])
plt.plot(history.history['mae'])
plt.show

print(score)
print ("Time required for training:",stop_time - start_time)

# Plot the loss and accuracy curves for training and validation
plt.figure()
fig, ax = plt.subplots(2,1)
```

```
ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'], color='r', label="validation loss",
           axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['mae'], color='b', label="Training MAE")
ax[1].plot(history.history['val_mae'], color='r',label="Validation MAE")
legend = ax[1].legend(loc='best', shadow=True)


# Visualize model history - diagnostic plot
plt.figure()
plt.plot(history.history['mae'], label='Training accuracy')
plt.plot(history.history['val_mae'], label='Validation MAE')
plt.title('White-Wine training / validation MAE values')
plt.ylabel('Mean Absolute Error')
plt.xlabel('Epoch')
plt.legend(loc="upper left")
plt.show()

plt.figure()
plt.plot(history.history['loss'], label='Training loss')
plt.plot(history.history['val_loss'], label='Validation loss')
plt.title('White-Wine training / validation loss values')
plt.ylabel('Loss value')
plt.xlabel('Epoch')
plt.legend(loc="upper left")
plt.show()
print(score)
```