Neural Network Model Analysis

Lucas Barber

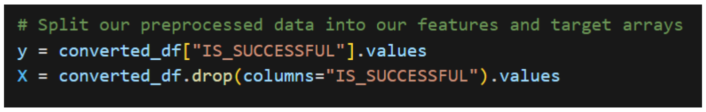
1) Overview

In this project I have built a neural network model to help a nonprofit foundation (Alphabet Soup) select which applicants it should fund, based on the best chance of success. The purpose of this analysis is to examine the efficacy of the neural network model, as well as document iterations to optimize it and compare results.

2) Results

Data Preprocessing

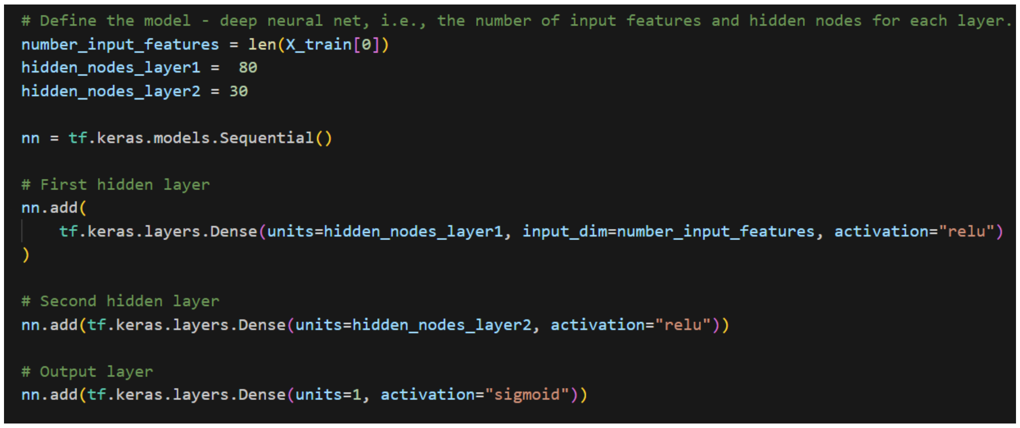
- **Target:** We want to identify which funding applicants have the greatest chance of success, so we're using whether or not past ventures have been successful: the "IS_SUCCESSFUL" column of our dataframe.
- **Features**: The features of this model are all columns other than "IS_SUCCESSFUL."
- We have removed the "EIN" and "NAME" columns from the input data, as they don't contribute to targets or features.



```
# Split our preprocessed data into our features and target arrays
y = converted_df["IS_SUCCESSFUL"].values
X = converted_df.drop(columns="IS_SUCCESSFUL").values
```

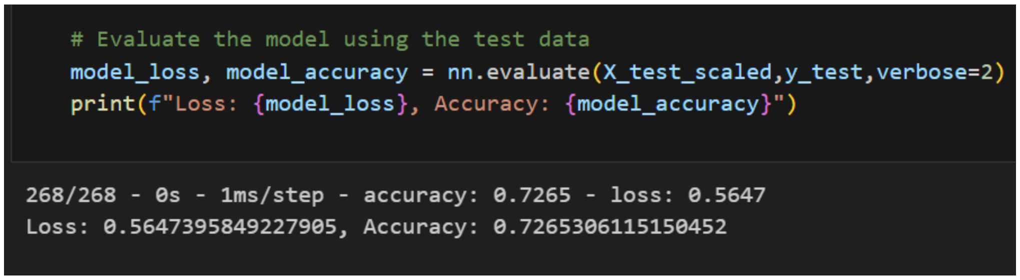
Compiling, Training, and Evaluation

- Initial Model:
    - Two hidden layers
        - Layer1: 80 nodes
        - Layer2: 30 nodes
        - Total: 110 neurons
        - Both used "relu" activation function
    - Output layer
        - Used "sigmoid" activation function



```python
# Define the model - deep neural net, i.e., the number of input features and hidden nodes for each layer.
number_input_features = len(X_train[0])
hidden_nodes_layer1 =  80
hidden_nodes_layer2 = 30

nn = tf.keras.models.Sequential()

# First hidden layer
nn.add(
    tf.keras.layers.Dense(units=hidden_nodes_layer1, input_dim=number_input_features, activation="relu")
)

# Second hidden layer
nn.add(tf.keras.layers.Dense(units=hidden_nodes_layer2, activation="relu"))

# Output layer
nn.add(tf.keras.layers.Dense(units=1, activation="sigmoid"))
```

This first model wasn't able to meet the accuracy threshold of 75%, as show below.



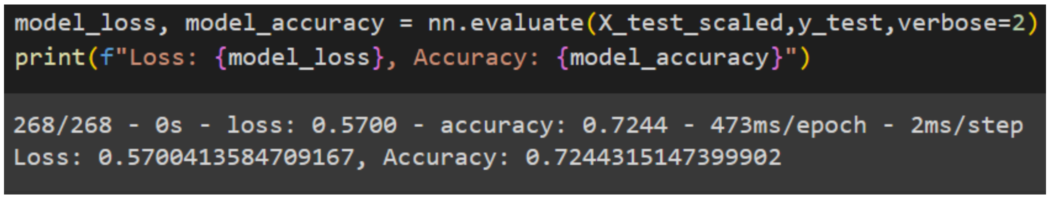
```python
# Evaluate the model using the test data
model_loss, model_accuracy = nn.evaluate(X_test_scaled,y_test,verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

```
268/268 - 0s - 1ms/step - accuracy: 0.7265 - loss: 0.5647
Loss: 0.5647395849227905, Accuracy: 0.7265306115150452
```

Therefore, I attempted the following optimization techniques:

- Changed the threshold of binning "CLASSIFICATION" value counts as "Other" from 1883 to 200
- Increased the number of nodes on hidden layers 1 and 2
    - Increase was done proportionally by 25%
    - Layer 1: 80 → 100
    - Layer 2: 30 → 38 (rounded up)
- Added an additional hidden layer
    - The number of neurons on the new node follows the proportional difference between layers 1 and 2. Layer 2 is 38% of layer 1, so layer 3 was made to be 38% of layer 2, being 14 nodes (rounded down).

Each of these optimization techniques failed to meet the accuracy threshold of 75%. In fact, their combination led to a marginally less accurate model, with a drop of 0.0021% accuracy.



```
model_loss, model_accuracy = nn.evaluate(X_test_scaled,y_test,verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

```
268/268 - 0s - loss: 0.5700 - accuracy: 0.7244 - 473ms/epoch - 2ms/step
Loss: 0.5700413584709167, Accuracy: 0.7244315147399902
```

3) Summary

The results of this model leave much to be desired. Neither the initial or tuned models were able to meet the minimum accuracy score of 75%. I would recommend building a separate model, perhaps using a support vector machine. While SVMs and neural networks perform similar functions of classification, they differ in their structure and approach – with the added benefit that SVMs take less time to train.