Alphabet Soup Charity Analysis Report

Overview of the analysis:

The purpose of this analysis is to predict whether applicants will be successful if funded by the Alphabet Soup nonprofit organization.

Results:

Data Preprocessing

- What variable(s) are the target(s) for your model?
  - Target variable is the 'IS_SUCCESSFUL' column.
- What variable(s) are the features for your model?
  - All other columns, excluding our target variable, and dropped columns are the features for modeling.
- What variable(s) should be removed from the input data because they are neither targets nor features?
  - The 'EIN' column was removed, as was the 'NAME' column; although in optimization, it was found that keeping 'NAME' as a feature for the neural network model increases accuracy significantly.

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

# Split the preprocessed data into a training and testing dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, stratify=y)

print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

```
(25724, 44) (25724,)
(8575, 44) (8575,)
```

Compiling, Training, and Evaluating the Model

- How many neurons, layers, and activation functions did you select for your neural network model, and why?
  - My initial model is compiled of three layers, with the hidden layers containing eight, and then five, neurons respectively. Both hidden layers use the relu activation function, and the output layer uses the sigmoid activation function. These choices were made based on common, and simple, neural network building techniques.

```
# 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.columns)
hidden_nodes_layer1 =  8
hidden_nodes_layer2 = 5

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"))

# Check the structure of the model
nn.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 8)                 360

 dense_1 (Dense)             (None, 5)                 45

 dense_2 (Dense)             (None, 1)                 6

=================================================================
Total params: 411 (1.61 KB)
Trainable params: 411 (1.61 KB)
```

- Were you able to achieve the target model performance?
    - Yes, but only after some feature engineering to the database; specifically, keeping the 'NAME' column in the database.

```
# Evaluate the second optimization model using the test data
model_loss, model_accuracy = nn_3.evaluate(X1_test_scaled,y1_test,verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
# big difference!
```

```
268/268 - 0s - loss: 0.4839 - accuracy: 0.7548 - 322ms/epoch - 1ms/step
Loss: 0.483879417181015, Accuracy: 0.7547521591186523
```

- What steps did you take in your attempts to increase model performance?
    - In my first attempt at optimization, I increased the neurons in each hidden layer of my model, but did not find any significant improvement. I then tried adding an additional hidden layer to the model, but that was actually detrimental to the model's performance. For my third, and final, optimization effort, I re-imported the data, and went through all the same steps for preprocessing, with the exception of leaving in the 'NAME' column. I tuned the model back down to only two hidden layers, but did adjust the number of nodes slightly. These parameters produced the best performance, at over 75% accuracy.

```
# Evaluate the first optimization model using the test data
model_loss, model_accuracy = nn_1.evaluate(X_test_scaled,y_test,verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
# not much better
```

```
268/268 - 0s - loss: 0.5569 - accuracy: 0.7219 - 353ms/epoch - 1ms/step
Loss: 0.5569201111793518, Accuracy: 0.7218658924102783
```

Summary:

In conclusion, with optimization tuning and feature engineering, this deep learning model was able to achieve over 75% accuracy. It is my recommendation that, in terms of charity success, there really is something in a name!