Charity Funding Predictor Report

1. Overview

The non-profit foundation Alphabet Soup wants to select applicants for its charity funding program, and for this, the foundation needs to find a system that will help it choose the right ventures. Therefore, with my knowledge of machine learning and neural network, I will target some features on the dataset provided by the foundation by using the appropriate machine learning model to create a binary classifier that can predict whether the selected ventures will succeed after receiving the fund.
I started analysing historical funding data about the 34,000 organisations that have received funding from Alphabet Soup over the years. The dataset contains 12 columns that capture metadata about each beneficiary.

2. Results

- **Data pre-processing**

I started my data pre-processing by removing irrelevant dataset features like EIN and NAME, which dropped the columns to 10.

The target variable of the model was IS_SUCCESSFUL which has a value of 1 for yes; the funding was used effectively, and 0 for no; it wasn't. Also, the same variable was the feature column chosen for the model (represented by y, the dependent variable)

- **Compiling, Training, and Evaluating the Model**

```
# 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_scaled[0])

nn = tf.keras.models.Sequential()

# First hidden layer
nn.add(tf.keras.layers.Dense(units=80, activation="relu", input_dim = number_input_features))

# Second hidden layer
nn.add(tf.keras.layers.Dense(units=30, 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, 80)                3520

 dense_1 (Dense)             (None, 30)                2430

 dense_2 (Dense)             (None, 1)                 31

=================================================================
Total params: 5,981
Trainable params: 5,981
Non-trainable params: 0
_________________________________________________________________
```

I chose those hyperparameters to increase the neural network model's performance and speed.
However, the figure below shows that the model didn't perform well.

```
# Compile the model
nn.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
```

```
# Train the model
fit_model = nn.fit(X_train_scaled, y_train, epochs=100)
```

```
804/804 [==============================] - 1s 1ms/step - loss: 0.5372 - accuracy: 0.7386
Epoch 61/100
804/804 [==============================] - 1s 1ms/step - loss: 0.5375 - accuracy: 0.7389
Epoch 62/100
804/804 [==============================] - 1s 1ms/step - loss: 0.5372 - accuracy: 0.7390
Epoch 63/100
804/804 [==============================] - 1s 1ms/step - loss: 0.5364 - accuracy: 0.7390
Epoch 64/100
804/804 [==============================] - 1s 1ms/step - loss: 0.5367 - accuracy: 0.7391
Epoch 65/100
804/804 [==============================] - 1s 1ms/step - loss: 0.5367 - accuracy: 0.7399
Epoch 66/100
804/804 [==============================] - 1s 1ms/step - loss: 0.5365 - accuracy: 0.7387
Epoch 67/100
804/804 [==============================] - 1s 1ms/step - loss: 0.5362 - accuracy: 0.7393
Epoch 68/100
804/804 [==============================] - 1s 1ms/step - loss: 0.5366 - accuracy: 0.7393
Epoch 69/100
804/804 [==============================] - 1s 1ms/step - loss: 0.5360 - accuracy: 0.7397
Epoch 70/100
```

```
# 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 - loss: 0.5616 - accuracy: 0.7298 - 187ms/epoch - 699us/step
Loss: 0.5615804195404053, Accuracy: 0.7297959327697754
```

The model achieves an accuracy of 73%, far below the 85% accuracy required for a basic model. Another factor to point out is the loss, which value is high.
To improve the performance, I built another model starting by dropping four columns from the initial dataset: EIN, NAME, SPECIAL_CONSIDERATION and USE_CASE and increased the number of neurons in the two hidden layers, as you can see in the following figure.

```
# 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_scaled[0])

nn = tf.keras.models.Sequential()

# First hidden layer
nn.add(tf.keras.layers.Dense(units=90, activation="relu", input_dim = number_input_features))

# Second hidden layer
nn.add(tf.keras.layers.Dense(units=50, 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, 90)                3330

 dense_1 (Dense)             (None, 50)                4550

 dense_2 (Dense)             (None, 1)                 51

=================================================================
Total params: 7,931
Trainable params: 7,931
Non-trainable params: 0
_________________________________________________________________
```

However, the performance has not improved; the loss and accuracy values worsened slightly.

```
# Train the model
fit_model = nn.fit(X_train_scaled, y_train, epochs=100)
```

```
Epoch 91/100
804/804 [==============================] - 3s 4ms/step - loss: 0.5414 - accuracy: 0.7371
Epoch 92/100
804/804 [==============================] - 3s 4ms/step - loss: 0.5404 - accuracy: 0.7373
Epoch 93/100
804/804 [==============================] - 3s 3ms/step - loss: 0.5406 - accuracy: 0.7373
Epoch 94/100
804/804 [==============================] - 3s 3ms/step - loss: 0.5408 - accuracy: 0.7376
Epoch 95/100
804/804 [==============================] - 3s 3ms/step - loss: 0.5407 - accuracy: 0.7373
Epoch 96/100
804/804 [==============================] - 3s 3ms/step - loss: 0.5405 - accuracy: 0.7368
Epoch 97/100
804/804 [==============================] - 3s 3ms/step - loss: 0.5406 - accuracy: 0.7366
Epoch 98/100
804/804 [==============================] - 3s 3ms/step - loss: 0.5404 - accuracy: 0.7374
Epoch 99/100
804/804 [==============================] - 3s 3ms/step - loss: 0.5406 - accuracy: 0.7369
Epoch 100/100
804/804 [==============================] - 3s 4ms/step - loss: 0.5399 - accuracy: 0.7370
```

```
# 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 - 1s - loss: 0.5659 - accuracy: 0.7264 - 961ms/epoch - 4ms/step
Loss: 0.5659346580505371, Accuracy: 0.7264139652252197
```

3. Summary

The model's best result employing the different numbers of neurons and layers was a 73% accuracy for the ReLU and sigmoid. The value could be improved by considering other processes, such as using a different model, increasing the number of layers, or dropping more columns in the dataset. Concerning the recommendation, I would suggest accessing a better dataset.