Report: Deep Learning Model for Charity Success Prediction

Overview of the Analysis

The purpose of this analysis was to build a binary classification model using a deep neural network (implemented in TensorFlow/Keras) to predict whether an organization funded by Alphabet Soup will be successful. The input data is a CSV file (`charity_data.csv`) containing various features about organizations, including their application type, affiliation, classification, use case, organization type, income, and a special consideration flag. The target variable is 'IS_SUCCESSFUL', indicating whether the organization was successful (1) or not (0).

Results

Data Preprocessing

- **Target Variable:** The target variable for the model is `IS_SUCCESSFUL`. This is the binary outcome we want to predict.
- **Feature Variables:** The features for the model are all the remaining columns in the dataset *after* preprocessing, specifically:
  - `APPLICATION_TYPE` (binned)
  - `AFFILIATION`
  - `CLASSIFICATION` (binned)
  - `USE_CASE`
  - `ORGANIZATION`
  - `INCOME_AMT`
  - `ASK_AMT`

  These were all converted to numerical representations using one-hot encoding.
- **Removed Variables:** The following columns were removed:
  - `EIN`: Identification column, not relevant for prediction.
  - `NAME`: Identification column, not relevant for prediction.
  - `STATUS`: Almost all values are the same. It has almost zero variance and little to no informational value.
  - `SPECIAL_CONSIDERATIONS`: This column is potentially redundant with other features or contains noisy data, which might be the reason for the original model's limited accuracy. Dropping such columns simplifies the model and can help.

Compiling, Training, and Evaluating the Model

- **Model Architecture:** The final optimized model has the following architecture:
  - **Input Layer:** Implicitly defined by the number of features (39 after one-hot encoding).
  - **Hidden Layer 1:** 100 neurons, 'relu' activation function.
  - **Hidden Layer 2:** 80 neurons, 'relu' activation function.

    - **Hidden Layer 3:** 30 neurons, 'relu' activation function.
    - **Hidden Layer 4:** 10 neurons, 'relu' activation function.
    - **Output Layer:** 1 neuron, 'sigmoid' activation function.

  The rationale for this architecture was to create a relatively deep network with a decreasing number of neurons in each subsequent layer. The 'relu' activation function is a standard choice for hidden layers, and 'sigmoid' is appropriate for the binary classification output. The increased number of layers and neurons, compared to the original model, gives the network more capacity to learn complex relationships.

- **Target Model Performance:** The target model performance of at least 75% accuracy was *not* achieved. The final optimized model achieved an accuracy of approximately 72.93% on the test dataset, and the model experienced loss: 0.5873922109603882.

- **Optimization Steps:**

  a. **Data Preprocessing:**
     - Dropped the `STATUS` and `SPECIAL_CONSIDERATIONS` columns, which were likely adding noise to the model.
     - Increased the binning cutoffs for `APPLICATION_TYPE` (to 700) and `CLASSIFICATION` (to 1800) to reduce the number of unique categorical values and, therefore, the number of input features after one-hot encoding. This addresses the issue of having too many input features relative to the number of data points.

  b. **Model Architecture:**
     - Added a third and a fourth hidden layers. More layers allow the network to learn more hierarchical representations of the data.
     - Increased the number of neurons in the first two hidden layers. This gives the model more capacity.

  c. **Training:**
     - Increased the number of epochs to 200, giving the model more opportunities to learn.
     - Added a callback (`SaveEveryNepochs`) to save the best model weights every 5 epochs, which is a good practice, although that alone wouldn't improve accuracy.

  d. **Stratified Splitting:**
     - Added `stratify=y` to the `train_test_split` function. This ensures that the *proportions* of the target variable (`IS_SUCCESSFUL`, which is 0 or 1) are the *same* in both the training and testing sets.

Summary and Recommendations

The optimized deep learning model, while improved over the starter code, did not achieve the 75% accuracy target. The final accuracy was approximately 72.93%. The loss was approximately 0.5874. This indicates that the model is still struggling to learn the underlying patterns in the data effectively. The model's performance is better than random guessing (which would be 50%

accuracy for a balanced binary classification), but it's not yet at a level that would be considered highly reliable for real-world use.

Recommendations for a Different Model (and Further Optimization):

Since the deep learning model did not meet the target, I strongly recommend exploring other, *non-neural network* machine learning models. Neural networks are powerful, but they are not always the best choice, especially when dealing with tabular data that may have many categorical features. Here's a recommended approach, along with further neural network tuning options:

1. **Try Gradient Boosted Trees (Best Chance of Improved Results):**

    - **Gradient Boosted Decision Trees (GBDTs):** Models like XGBoost, LightGBM, and CatBoost are often *highly effective* on structured/tabular data (like this CSV file). They often outperform neural networks and require *far less* hyperparameter tuning. They also handle categorical features directly, without one-hot encoding (though one-hot encoding still usually helps).

```
import xgboost as xgb
from sklearn.model_selection import train_test_split,
cross_val_score, StratifiedKFold
from sklearn.metrics import accuracy_score
import pandas as pd

# Load data (same as before)
application_df =
pd.read_csv("https://static.bc-edx.com/data/dl-1-2/m21/lms/
starter/charity_data.csv")
application_df = application_df.drop(columns = ['EIN',
'NAME', 'STATUS', 'SPECIAL_CONSIDERATIONS'])

# Binning (same as before)
application_counts =
application_df['APPLICATION_TYPE'].value_counts()
application_types_to_replace =
list(application_counts[application_counts<700].index)
for app in application_types_to_replace:
    application_df['APPLICATION_TYPE'] =
application_df['APPLICATION_TYPE'].replace(app,"Other")

classification_counts =
application_df['CLASSIFICATION'].value_counts()
classifications_to_replace =
list(classification_counts[classification_counts <
1800].index)
for cls in classifications_to_replace:
     application_df['CLASSIFICATION'] =
application_df['CLASSIFICATION'].replace(cls,"Other")
```

```
# One-Hot Encode (same as before)
application_df = pd.get_dummies(application_df)

# Split data
y = application_df['IS_SUCCESSFUL'].values
X = application_df.drop(['IS_SUCCESSFUL'], axis=1).values
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=78, stratify=y)


# XGBoost Model
xgb_model = xgb.XGBClassifier(objective="binary:logistic",
random_state=42, eval_metric="logloss",
use_label_encoder=False)

# Cross-validation (important for reliable evaluation)
cv = StratifiedKFold(n_splits=5, shuffle=True,
random_state=42)  # Stratified K-Fold
cv_scores = cross_val_score(xgb_model, X_train, y_train,
cv=cv, scoring='accuracy')
print(f"Cross-Validation Accuracy: {cv_scores.mean():.4f}
(+/- {cv_scores.std() * 2:.4f})")

# Train on the full training set (for final model)
xgb_model.fit(X_train, y_train)

# Evaluate on the test set
y_pred = xgb_model.predict(X_test)
test_accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {test_accuracy:.4f}")

#You can save the XGBoost model as follows:
# import joblib
# joblib.dump(xgb_model,
'AlphabetSoupCharity_XGBoost.model')
```

- **Key Advantages of GBDTs:**
  - **Handles Categorical Features Well:** GBDTs are designed to work well with categorical features, even without one-hot encoding (though, as mentioned, one-hot encoding is generally a good idea to perform, even with these models).
  - **Less Prone to Overfitting:** They are less prone to overfitting than neural networks, especially with limited data.
  - **Faster Training:** They typically train much faster than deep neural networks.
  - **Easier Tuning:** They have fewer hyperparameters to tune.
  - **Interpretability:** It's easier to understand *why* a GBDT model made a particular prediction.

- **XGBoost:** A very popular and high-performance implementation of GBDTs. The code above provides a basic example, but you should explore its many hyperparameters for optimal performance. The `objective="binary:logistic"` is crucial for binary classification. `eval_metric="logloss"` is a good evaluation metric to use during training. `use_label_encoder=False` suppresses a warning.
- **Cross Validation** Provides a robust way to test the model's accuracy.
- **LightGBM:** Another excellent GBDT library, often faster than XGBoost.
- **CatBoost:** Another strong option, particularly good at handling categorical features.

2. **Try Other Machine Learning Models (Good Idea):**

   – **Logistic Regression:** A simple but often surprisingly effective baseline model for binary classification.
   – **Support Vector Machines (SVMs):** Another powerful classification algorithm.
   – **Random Forests:** An ensemble method (like GBDTs, but using a different approach) that can be very effective.
   – **K Nearest Neighbors (KNN)**

3. **Further Neural Network Optimization (If you want to stick with Neural Networks):**

   There are more advanced techniques to try, *in addition* to the ones already implemented:

   – **Learning Rate Scheduling:** Instead of a fixed learning rate, use a learning rate scheduler (e.g., `tf.keras.callbacks.ReduceLROnPlateau`, `tf.keras.optimizers.schedules.ExponentialDecay`). This can help the model converge more effectively.
   – **Batch Size Tuning:** Experiment with the `batch_size` argument in `nn.fit()`. Smaller batch sizes can sometimes improve generalization, but they also make training slower.
   – **Different Optimizers/Loss Functions:** While Adam is often a good choice, one can explore other optimizers like SGD, RMSprop, or Adagrad. One could also use a different activation function such as "tanh" or "elu".
   – **Early Stopping:** Use the `tf.keras.callbacks.EarlyStopping` callback. This will automatically stop training when the validation loss stops improving, preventing overfitting and saving training time. This is *highly* recommended.

4. **Feature Engineering**

~~~
*  **Interaction terms:** Combine two related features into a
single, new feature, which may hold more information than either
feature alone.
 ```python
 #Example
 application_df['ASK_AMT_Category'] =
pd.cut(application_df['ASK_AMT'], bins=[0, 5000, 100000, 1000000,
float('inf')], labels=['Low', 'Medium', 'High', 'Very High'],
~~~

```
    include_lowest=True)
     application_df = pd.get_dummies(application_df,
    columns=['ASK_AMT_Category'])

     ```
     * **Domain expertise**: Using domain knowledge to make different
    decisions in preprocessing.

Complete, Improved Code (Neural Network, Further Optimized):

```python
# Import our dependencies
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
import pandas as pd
import tensorflow as tf
import os

# Import pandas and read the charity_data.csv from the provided cloud
# URL.
import pandas as pd
application_df =
pd.read_csv("https://static.bc-edx.com/data/dl-1-2/m21/lms/starter/
charity_data.csv")
# Display the first few rows of the DataFrame
application_df.head()

# Drop the non-beneficial ID columns, 'EIN' and 'NAME'.  Also drop
STATUS and SPECIAL_CONSIDERATIONS.
application_df = application_df.drop(columns = ['EIN', 'NAME',
'STATUS', 'SPECIAL_CONSIDERATIONS'])

# Determine the number of unique values for each column.
print(application_df.nunique())

# Look at APPLICATION_TYPE value counts for binning
application_counts = application_df['APPLICATION_TYPE'].value_counts()
print(application_counts)

# Choose a cutoff value and create a list of application types to be
replaced
# use the variable name `application_types_to_replace`
# Increased cutoff for more binning.
application_types_to_replace =
list(application_counts[application_counts<700].index)

# Replace in dataframe
for app in application_types_to_replace:
    application_df['APPLICATION_TYPE'] =
application_df['APPLICATION_TYPE'].replace(app,"Other")
```

```
# Check to make sure binning was successful
print(application_df['APPLICATION_TYPE'].value_counts())

# Look at CLASSIFICATION value counts for binning
classification_counts =
application_df['CLASSIFICATION'].value_counts()
print(classification_counts)

# You may find it helpful to look at CLASSIFICATION value counts >1
print(classification_counts[classification_counts > 1])

# Choose a cutoff value and create a list of classifications to be
replaced
# use the variable name `classifications_to_replace`
# Increased cutoff for more binning
classifications_to_replace =
list(classification_counts[classification_counts < 1800].index)

# Replace in dataframe
for cls in classifications_to_replace:
    application_df['CLASSIFICATION'] =
application_df['CLASSIFICATION'].replace(cls,"Other")

# Check to make sure binning was successful
print(application_df['CLASSIFICATION'].value_counts())

# Convert categorical data to numeric with `pd.get_dummies`
# One-hot encode all categorical columns.  No need for a separate
list.
application_df = pd.get_dummies(application_df)
print(application_df.head())

# Split our preprocessed data into our features and target arrays
y = application_df['IS_SUCCESSFUL'].values
X = application_df.drop(['IS_SUCCESSFUL'], axis=1).values

# 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=78, stratify=y)

# Create a StandardScaler instances
scaler = StandardScaler()

# Fit the StandardScaler
X_scaler = scaler.fit(X_train)

# Scale the data
X_train_scaled = X_scaler.transform(X_train)
X_test_scaled = X_scaler.transform(X_test)
```

```
# Step 2: Compile, Train and Evaluate 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[0])
hidden_nodes_layer1 = 128  # Increased neurons
hidden_nodes_layer2 = 64   # Increased neurons
hidden_nodes_layer3 = 32   # Increased neurons
hidden_nodes_layer4 = 16
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",
             kernel_regularizer=tf.keras.regularizers.l2(0.001)))  #
Add L2 regularization
nn.add(tf.keras.layers.Dropout(0.3))

# Second hidden layer.
nn.add(tf.keras.layers.Dense(units=hidden_nodes_layer2,
activation="relu",
             kernel_regularizer=tf.keras.regularizers.l2(0.001)))  #
Add L2 regularization
nn.add(tf.keras.layers.Dropout(0.3))

# Third hidden layer
nn.add(tf.keras.layers.Dense(units=hidden_nodes_layer3,
activation="relu",
        kernel_regularizer=tf.keras.regularizers.l2(0.001)))  # Add L2
regularization
nn.add(tf.keras.layers.Dropout(0.3))

# Fourth Hidden Layer
nn.add(tf.keras.layers.Dense(units=hidden_nodes_layer4,
activation="relu",
        kernel_regularizer=tf.keras.regularizers.l2(0.001)))  # Add L2
regularization
nn.add(tf.keras.layers.Dropout(0.3))


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

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

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

```
# Create a directory for checkpoints
checkpoint_dir = 'checkpoints'
if not os.path.exists(checkpoint_dir):
    os.makedirs(checkpoint_dir)

# Create a custom callback to save the model's weights every 5 epochs
class SaveEveryNepochs(tf.keras.callbacks.Callback):
    def __init__(self, filepath, n_epochs=5):
        super().__init__()
        self.filepath = filepath
        self.n_epochs = n_epochs

    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % self.n_epochs == 0:  # Save every n_epochs
            path = self.filepath.format(epoch=epoch + 1)
            self.model.save_weights(path)
            print(f"Checkpoint saved at epoch {epoch + 1} to {path}")

checkpoint_callback = SaveEveryNepochs(
    filepath='checkpoints/weights.{epoch:02d}.weights.h5',  #
Corrected extension
    n_epochs=5
)

# Added early stopping callback
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',  # Monitor validation loss
    patience=20,         # Stop after 20 epochs with no improvement
    restore_best_weights=True  # Restore the best weights
)

# Train the model
# Increased epochs for longer training.  Use the custom callback.
fit_model = nn.fit(X_train_scaled, y_train, epochs=200,
callbacks=[checkpoint_callback, early_stopping], validation_split=0.2)

# 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}")

# Save the entire model to an HDF5 file
nn.save("AlphabetSoupCharity_Optimization_Final.h5")
```

This improved code incorporates additional optimizations for the neural networks, and also includes code for the use of the XGBoost algorithm.