index

May 12, 2022

1 Tuning and Optimizing Neural Networks - Lab

1.1 Introduction

Now that you've practiced regularization, initialization, and optimization techniques, its time to synthesize these concepts into a cohesive modeling pipeline.

With this pipeline, you will not only fit an initial model but also attempt to improve it. Your final model selection will pertain to the test metrics across these models. This will more naturally simulate a problem you might be faced with in practice, and the various modeling decisions you are apt to encounter along the way.

Recall that our end objective is to achieve a balance between overfitting and underfitting. You've seen the bias variance trade-off, and the role of regularization in order to reduce overfitting on training data and improving generalization to new cases. Common frameworks for such a procedure include train/validate/test methodology when data is plentiful, and K-folds cross-validation for smaller, more limited datasets. In this lab, you'll perform the latter, as the dataset in question is fairly limited.

1.2 Objectives

You will be able to:

- Apply normalization as a preprocessing technique
- Implement a K-folds cross validation modeling pipeline for deep learning models
- Apply regularization techniques to improve your model's performance

1.3 Load the data

First, run the following cell to import all the neccessary libraries and classes you will need in this lab.

[1]:
```
# Necessary libraries and classes
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_predict
from keras import models
```

```
from keras import layers
from keras import regularizers
from keras.wrappers.scikit_learn import KerasRegressor
```

In this lab you'll be working with the *The Lending Club* data.

- Import the data available in the file `'loan_final.csv'`
- Drop rows with missing values in the `'total_pymnt'` column (this is your target column)
- Print the first five rows of the data
- Print the dimensions of the data

[8]:
```
# Import the data
data = pd.read_csv("loan_final.csv")

# Drop rows with no target value
data.dropna(subset = ["total_pymnt"], inplace = True)

# Print the first five rows
data["total_pymnt"].isna().sum()
```

[8]:
```
0
```

[9]:
```
# Print the dimensions of data
data.shape
```

[9]:
```
(42535, 16)
```

1.4 Generating a Hold Out Test Set

While we will be using K-fold cross validation to select an optimal model, we still want a final hold out test set that is completely independent of any modeling decisions. As such, pull out a sample of 30% of the total available data. For consistency of results, use random seed 42.

[11]:
```
# Features to build the model
features = ['loan_amnt', 'funded_amnt_inv', 'installment', 'annual_inc',
            'home_ownership', 'verification_status', 'emp_length']

X = data[features]
y = data[['total_pymnt']]

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
                                                    random_state = 42)
```

1.5 Preprocessing (Numerical features)

- Fill all missing values in numeric features with their respective means

- Standardize all the numeric features
- Convert the final results into DataFrames

[49]:
```
# Select continuous features
cont_features = ['loan_amnt', 'funded_amnt_inv', 'installment', 'annual_inc']

X_train_cont = X_train.loc[:, cont_features]
X_test_cont = X_test.loc[:, cont_features]

# Instantiate SimpleImputer - fill the missing values with the mean
si = SimpleImputer(strategy = 'mean')

# Fit and transform the training data
X_train_imputed = si.fit_transform(X_train_cont)

# Transform test data
X_test_imputed = si.transform(X_test_cont)

# Instantiate StandardScaler
ss_X = StandardScaler()

# Fit and transform the training data
X_train_scaled = pd.DataFrame(ss_X.fit_transform(X_train_imputed), columns =␣
 ↪cont_features)

# Transform test data
X_test_scaled = pd.DataFrame(ss_X.transform(X_test_imputed), columns =␣
 ↪cont_features)
```

1.6 Preprocessing (Categorical features)

- Fill all missing values in categorical features with the string `'missing'`
- One-hot encode all categorical features
- Convert the final results into DataFrames

[50]:
```
# Select only the categorical features
cat_features = ['home_ownership', 'verification_status', 'emp_length']
X_train_cat = X_train.loc[:, cat_features]
X_test_cat = X_test.loc[:, cat_features]

# Fill missing values with the string 'missing'

X_train_cat.fillna(value='missing', inplace=True)
X_test_cat.fillna(value='missing', inplace=True)


# OneHotEncode categorical variables
```

```
ohe = OneHotEncoder(handle_unknown='ignore')


# Transform training and test sets
X_train_ohe = ohe.fit_transform(X_train_cat)
X_test_ohe  = ohe.transform(X_test_cat)

# Get all categorical feature names
cat_columns = ohe.get_feature_names(input_features=X_train_cat.columns)

# Fit and transform the training data
X_train_categorical = pd.DataFrame(X_train_ohe.todense(), columns=cat_columns)

# Transform test data
X_test_categorical = pd.DataFrame(X_test_ohe.todense(), columns=cat_columns)
```

```
/opt/anaconda3/envs/learn-env/lib/python3.8/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning: Function
get_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will
be removed in 1.2. Please use get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

Run the below cell to combine the numeric and categorical features.

[51]:
```
# Combine continuous and categorical feature DataFrames
X_train_all = pd.concat([X_train_scaled, X_train_categorical], axis=1)
X_test_all = pd.concat([X_test_scaled, X_test_categorical], axis=1)

# Number of input features
n_features = X_train_all.shape[1]
```

- Standardize the target DataFrames (`y_train` and `y_test`)

[52]:
```
# Instantiate StandardScaler
ss_y = StandardScaler()

# Fit and transform Y (train)
y_train_scaled = ss_y.fit_transform(y_train)

# Transform test Y (test)
y_test_scaled = ss_y.transform(y_test)
```

1.7 Define a K-fold Cross Validation Methodology

Now that your have a complete holdout test set, you will perform k-fold cross-validation using the following steps:

- Create a function that returns a compiled deep learning model
- Use the wrapper function `KerasRegressor()` that defines how these folds are trained

- Call the `cross_val_predict()` function to perform k-fold cross-validation

In the cell below, we've defined a baseline model that returns a compiled Keras models.

[53]:
```
# Define a function that returns a compiled Keras model
def create_baseline_model():

    # Initialize model
    model = models.Sequential()

    # First hidden layer
    model.add(layers.Dense(10, activation='relu', input_shape=(n_features,)))

    # Second hidden layer
    model.add(layers.Dense(5, activation='relu'))

    # Output layer
    model.add(layers.Dense(1, activation='linear'))

    # Compile the model
    model.compile(optimizer='SGD',
                  loss='mse',
                  metrics=['mse'])

    # Return the compiled model
    return model
```

Wrap `create_baseline_model` inside a call to `KerasRegressor()`, and:

- Train for 150 epochs
- Set the batch size to 256

  NOTE: Refer to the documentation to learn about `KerasRegressor()`.

[54]:
```
# Wrap the above function for use in cross-validation
keras_wrapper_1 = KerasRegressor(create_baseline_model,
                                 epochs = 150,
                                 batch_size = 256, verbose = 0)
```

Use `cross_val_predict()` to generate cross-validated predictions with: - 5-fold cv - scaled input (`X_train_all`) and output (`y_train_scaled`)

[63]:
```
#   This cell may take several mintes to run
# Generate cross-validated predictions
np.random.seed(123)

cv_baseline_preds = cross_val_predict(keras_wrapper_1,
                                      X_train_all, y_train_scaled,
                                      cv = 5)
```

- Find the RMSE on train data

[64]:
```
# RMSE on train data (scaled)
mean_squared_error(y_train_scaled, cv_baseline_preds, squared = False)
```

[64]:
```
0.4406958527621424
```

- Convert the scaled predictions back to original scale
- Calculate RMSE in the original units with `y_train` and `baseline_preds`

[66]:
```
# Convert the predictions back to original scale
baseline_preds = ss_y.inverse_transform(cv_baseline_preds.reshape(-1,1))

# RMSE on train data (original scale)
np.sqrt(mean_squared_error(y_train, baseline_preds))
```

[66]:
```
4014.095588894171
```

1.8 Intentionally Overfitting a Model

Now that you've developed a baseline model, its time to intentionally overfit a model. To overfit a model, you can: * Add layers * Make the layers bigger * Increase the number of training epochs

Again, be careful here. Think about the limitations of your resources, both in terms of your computers specs and how much time and patience you have to let the process run. Also keep in mind that you will then be regularizing these overfit models, meaning another round of experiments and more time and resources.

[70]:
```
# Define a function that returns a compiled Keras model
def create_bigger_model():
    # Initialize model
    model = models.Sequential()

    # First hidden layer
    model.add(layers.Dense(10, activation='relu', input_shape=(n_features,)))

    # Second hidden layer
    model.add(layers.Dense(10, activation='relu'))

    # Third hidden layer
    model.add(layers.Dense(10, activation='relu'))

    # Forth hidden layer
    model.add(layers.Dense(10, activation='relu'))

    # Fifth hidden layer
    model.add(layers.Dense(10, activation='relu'))

    # Output layer
```

```
    model.add(layers.Dense(1, activation='linear'))

    # Compile the model
    model.compile(optimizer='SGD',
                  loss='mse',
                  metrics=['mse'])

    # Return the compiled model
    return model
```

[71]:
```
# Wrap the above function for use in cross-validation
keras_wrapper_2 = KerasRegressor(create_bigger_model,
                                 epochs = 150,
                                 batch_size = 256,
                                 verbose = 0)
```

[72]:
```
#  This cell may take several mintes to run
# Generate cross-validated predictions
np.random.seed(123)
cv_bigger_model_preds = cross_val_predict(keras_wrapper_2,
                                          X_train_all, y_train_scaled,
                                          cv = 5)
```

[73]:
```
# RMSE on train data (scaled)
mean_squared_error(y_train_scaled, cv_bigger_model_preds, squared = False)
```

[73]:
```
0.441422151900384
```

1.9 Regularizing the Model to Achieve Balance

Now that you have a powerful model (albeit an overfit one), we can now increase the generalization of the model by using some of the regularization techniques we discussed. Some options you have to try include:
* Adding dropout * Adding L1/L2 regularization * Altering the layer architecture (add or remove layers similar to above)

This process will be constrained by time and resources. Be sure to test at least two different methodologies, such as dropout and L2 regularization. If you have the time, feel free to continue experimenting.

[74]:
```
# Define a function that returns a compiled Keras model
def create_regularized_model():

    # Initialize model
    model = models.Sequential()

    # First hidden layer
    model.add(layers.Dropout(0.3, input_shape=(n_features,)))
```

```
    # Second hidden layer
    model.add(layers.Dense(10, kernel_regularizer = regularizers.l2(0.005),␣
↪activation='relu'))
    model.add(layers.Dropout(0.3))

    # Third hidden layer
    model.add(layers.Dense(10, kernel_regularizer = regularizers.l2(0.005),␣
↪activation='relu'))
    model.add(layers.Dropout(0.3))

    # Forth hidden layer
    model.add(layers.Dense(10, kernel_regularizer = regularizers.l2(0.005),␣
↪activation='relu'))
    model.add(layers.Dropout(0.3))


    # Output layer
    model.add(layers.Dense(1, activation='linear'))

    # Compile the model
    model.compile(optimizer='SGD',
                  loss='mse',
                  metrics=['mse'])

    # Return the compiled model
    return model
```

[75]:
```
# Wrap the above function for use in cross-validation
keras_wrapper_3 = KerasRegressor(create_regularized_model,
                                 epochs = 150,
                                 batch_size = 256,
                                 verbose = 0)
```

[76]:
```
#   This cell may take several mintes to run
# Generate cross-validated predictions
np.random.seed(123)
cv_dropout_preds = cross_val_predict(keras_wrapper_3,
                                     X_train_all, y_train_scaled,
                                     cv = 5)
```

[77]:
```
# RMSE on train data (scaled)
mean_squared_error(y_train_scaled, cv_dropout_preds, squared = False)
```

[77]:
```
0.6029538270870766
```

1.10 Final Evaluation

Now that you have selected a network architecture, tested various regularization procedures and tuned hyperparameters via a validation methodology, it is time to evaluate your final model on the test set. Fit the model using all of the training data using the architecture and hyperparameters that were most effective in your experiments above. Afterwards, measure the overall performance on the hold-out test data which has been left untouched (and hasn't leaked any data into the modeling process)!

[79]:
```
#  This cell may take several mintes to run

## The best model is `create_baseline_model` so we will use this

# Initialize model
model = models.Sequential()

# First hidden layer
model.add(layers.Dense(10, activation='relu', input_shape=(n_features,)))

# Second hidden layer
model.add(layers.Dense(5, activation='relu'))

# Output layer
model.add(layers.Dense(1, activation='linear'))

# Compile the model
model.compile(optimizer='SGD',
                  loss='mse',
                  metrics=['mse'])
model.fit(X_train_all, y_train_scaled, epochs = 150, batch_size = 256, verbose␣
 ↪= 0)
```

[79]:
```
<tensorflow.python.keras.callbacks.History at 0x7f9ad9b1bf70>
```

[80]:
```
y_test_pred_scaled = model.predict(X_test_all)
y_test_pred = ss_y.inverse_transform(y_test_pred_scaled)

mean_squared_error(y_test, y_test_pred, squared = False)
```

[80]:
```
3851.5775714586002
```

1.11 Summary

In this lab, you investigated some data from *The Lending Club* in a complete data science pipeline to build neural networks with good performance. You began with reserving a hold-out set for testing which never was touched during the modeling phase. From there, you implemented a k-fold cross validation methodology in order to assess an initial baseline model and various regularization methods.