**Crude Oil Price Forecasting**

**Data sources:**

**Overview**

The data is extracted at a day level, this is fed as a weighted average of volume. Data timeline is from 2011 to current date indicating a complex dataset

Machine learning model is built for crude price forecasting for the next 7 days based on the streaming and batch datasets in GCP.

Our main API was (St. Louis Fed's FRED API) for the below indicators.

**Commodity Prices**

We chose to include series that represent the oil market

DCOILWTICO: West Texas Intermediate (WTI) - Cushing Oklahoma

**Debt Market Indicators**

A ladder of bond market indicators are represented in the data in LIBOR rates at various maturities. Specifically, LIBOR is included at overnight, 1-month, 3-month and 12-month maturities. To (very crudely) represent the consumer and the corporate markets we also included indices representing high yield returns and prime corporate debt returns.

**Energy-Related Series**

To confirm the presence of signals in the energy sector data, we collect information on natural gas and energy sector volatility. This data is obtained from the St. Louis Fed's FRED API.

MHHNGSP: Henry Hub Natural Gas Spot Price

VXXLECLS: [CBOE Energy Sector ETF Volatility Index](https://fred.stlouisfed.org/series/VXXLECLS)

## Traditional Currencies

The CURRENCY DATABASE was accessed using archived data in Federal Government database to gather currency exchange rates of the US Dollar against the Japanese Yen, the Euro, the Chinese Yuan, the Mexican Peso, and the Australian Dollar and their past trends.

* DEXCHUS: [Chinese Yuan to USD](https://fred.stlouisfed.org/series/DEXCHUS)
* DEXJPUS: [Japanese Yen to USD](https://fred.stlouisfed.org/series/DEXJPUS)
* DEXUSEU: [USD to European Union's Euro](https://fred.stlouisfed.org/series/DEXUSEU)
* DEXMXUS: [Mexican New Pesos to USD](https://fred.stlouisfed.org/series/DEXMXUS)
* DEXUSAL: [USD to Australian Dollar](https://fred.stlouisfed.org/series/DEXUSAL)

Data pre-processing:

## Generating in Calendar Attributes for Merge

To ensure adjustment for temporal factors a function generate\_calendar is specified and used to merge in calendar attributes to the data. These attributes, such as month or weekday, can be used in a machine learning model alongside the data gathered from the FRED API

**import** numpy **as** np

**def** generate\_calendar(year, drop\_index**=False**):

'''

Simple function to generate a calendar containing

US holidays, weekdays and holiday weeks.

'''

**from** pandas.tseries.offsets **import** YearEnd

**from** pandas.tseries.holiday **import** USFederalHolidayCalendar

start\_date **=** pd**.**to\_datetime('1/1/'**+**str(year))

end\_date **=** start\_date **+** YearEnd()

DAT **=** pd**.**date\_range(str(start\_date), str(end\_date), freq**=**'D')

MO **=** [d**.**strftime('%B') **for** d **in** DAT]

holidays **=** USFederalHolidayCalendar()**.**holidays(start**=**start\_date, end**=**end\_date)

cal\_df **=** pd**.**DataFrame({'date':DAT, 'month':MO})

cal\_df['year'] **=** [format(d, '%Y') **for** d **in** DAT]

cal\_df['weekday'] **=** [format(d, '%A') **for** d **in** DAT]

cal\_df['is\_weekday'] **=** cal\_df**.**weekday**.**isin(['Monday','Tuesday','Wednesday','Thursday','Friday'])

cal\_df['is\_weekday'] **=** cal\_df['is\_weekday']**.**astype(int)

cal\_df['is\_holiday'] **=** cal\_df['date']**.**isin(holidays)

cal\_df['is\_holiday'] **=** cal\_df['is\_holiday']**.**astype(int)

cal\_df['is\_holiday\_week'] **=** cal\_df**.**is\_holiday**.**rolling(window**=**7,center**=True**,min\_periods**=**1)**.**sum()

cal\_df['is\_holiday\_week'] **=** cal\_df['is\_holiday\_week']**.**astype(int)

**if** **not** drop\_index: cal\_df**.**set\_index('date', inplace**=True**)

**return** cal\_df

**def** make\_calendars(year\_list, drop\_index):

cal\_df **=** pd**.**DataFrame()

**for** year **in** year\_list:

cal\_df **=** cal\_df**.**append(generate\_calendar(year, drop\_index**=**drop\_index))

**return** cal\_df

year\_list **=** [str(int(i)) **for** i **in** np**.**arange(2011, 2019)]

cal\_df **=** make\_calendars(year\_list, drop\_index**=False**)

cal\_df**.**head()

Output:

A screenshot of a calendar

Description automatically generated

**Integrating Calendar Data and Handling Missing Values:**

When we incorporate calendar details for each day starting from 2011 through an outer join process, it naturally leads to the emergence of missing values. This is typical in financial markets, particularly for indicators like DJIA, where we see NaN values on weekends and holidays, and for other metrics that aren't recorded daily. To address this, we employ the pd.DataFrame.fillna function from Pandas in a two-step process. Firstly, we apply the 'bfill' method, followed by the 'ffill' method. This approach is chosen for its simplicity and to steer clear of more intricate data imputation techniques, even though it might not be the most scientifically rigorous method.

econ\_df = econ\_df.join(cal\_df, how='outer')

econ\_df = econ\_df.fillna(method='bfill')

econ\_df = econ\_df.fillna(method='ffill')

**Eliminating Records Beyond Current Date**

Using Python's datetime module, we filter out any records in the dataset that fall beyond the current date. This step is crucial as the calendar function may have introduced future dates which aren't relevant for our analysis.

**from datetime import datetime as dt**

**# Remove records that are dated after the current date**

**current\_date\_filter = pd.to\_datetime(econ\_df.index.values) <= dt.now()**

**econ\_df = econ\_df.loc[current\_date\_filter]**

**Generating One-Hot Encoded Features:**

For the 'month', 'year', and 'weekday' columns in our dataset to be effectively used in neural network models, they need to be transformed into one-hot encoded vectors. This is achieved using the **pd.get\_dummies** function in Pandas, which converts these categorical columns into a format suitable for machine learning models.

# Implement one-hot encoding for specific columns

econ\_df = pd.get\_dummies(econ\_df,

columns=['month', 'year', 'weekday'],

drop\_first=True)

**Transforming Column Names to Lowercase:**

To maintain consistency and ease of access in our data handling, we'll convert all column names in the econ\_df DataFrame to lowercase. This is a simple yet effective practice in data preprocessing, especially for ensuring uniformity in column references.

# Convert all column names in econ\_df to lowercase

econ\_df.columns = [column\_name.lower() for column\_name in econ\_df.columns]

print(econ\_df.columns.tolist())

The resulting column names will be:

['sp500', 'nasdaqcom', 'djia', 'ru2000pr', 'bogmbasew', 'dexjpus', 'dexuseu', 'dexchus', 'dexusal', 'vixcls', 'usdontd156n', 'usd1mtd156n', 'usd3mtd156n', 'usd12md156n', 'bamlhyh0a0hym2triv', 'bamlcc0a1aaatriv', 'goldamgbd228nlbm', 'dcoilwtico', 'mhhngsp', 'vxxlecls', 'is\_weekday', 'is\_holiday', 'is\_holiday\_week', 'month\_august', 'month\_december', 'month\_february', 'month\_january', 'month\_july', 'month\_june', 'month\_march', 'month\_may', 'month\_november', 'month\_october', 'month\_september', 'year\_2012', 'year\_2013', 'year\_2014', 'year\_2015', 'year\_2016', 'year\_2017', 'year\_2018', 'weekday\_monday', 'weekday\_saturday', 'weekday\_sunday', 'weekday\_thursday', 'weekday\_tuesday', 'weekday\_wednesday'].

**Enhancing Data Signal and Reducing Noise through Feature Engineering**

Our approach to refining the data involves two key steps aimed at minimizing noise and amplifying the signal over time. Instead of working directly with raw price data, we proceed as follows:

Data Transformation: We reshape the data to consist of just three columns: 'date', 'variable', and 'value'. This is achieved by melting the dataset.

Signal Processing: We then apply a method known as split-apply-combine. This involves grouping the data by 'variable', computing the percent change, and then calculating a rolling window mean of this percent change.

Data Restoration: Finally, we spread the data back to its original structure, substituting the raw values with the newly calculated rolling window percent changes.

This methodology is designed to be more responsive to market fluctuations rather than focusing on absolute values at any given point in time. By taking a rolling mean, we also dilute any outlier market movements, making them more representative of general trends.

Melting the econ\_df DataFrame on the 'date' Column

To facilitate the split-apply-combine operation and streamline plotting, we first melt econ\_df on the 'date' column:

econ\_df\_melt = econ\_df.copy()

econ\_df\_melt.reset\_index(inplace=True)

econ\_df\_melt.rename(columns={'index': 'date'}, inplace=True)

econ\_df\_melt = econ\_df\_melt.melt(id\_vars=['date'])

print(econ\_df\_melt.head())

This code snippet reshapes econ\_df such that it only has three columns: 'date', 'variable', and 'value'. The first few rows of the transformed DataFrame would look like this:

date variable value

2023-01-01 sp500 1271.87

2023-01-02 sp500 1271.87

2023-01-03 sp500 1271.87

2023-01-04 sp500 1270.20

2023-01-05 sp500 1276.56

This transformation sets the stage for the subsequent steps of feature engineering to enhance the utility of the dataset for analytical purposes.

Implementing the Split-Apply-Combine Process for Feature Calculation

To extract more meaningful insights from our econ\_df\_melt dataset, we will implement a split-apply-combine strategy. This involves dividing the dataset based on the 'variable' column, applying calculations to each subset, and then recombining them. Notably, binary columns listed in onehot\_cols will not undergo this process as they don't require the same transformation.

onehot\_cols = [

'is\_weekday', 'is\_holiday', 'is\_holiday\_week',

'month\_august', 'month\_december', 'month\_february',

'month\_january', 'month\_july', 'month\_june', 'month\_march',

'month\_may', 'month\_november', 'month\_october', 'month\_september',

'year\_2011', 'year\_2012', 'year\_2013', 'year\_2014',

'year\_2015', 'year\_2016', 'year\_2017', 'year\_2018',

'weekday\_monday', 'weekday\_saturday', 'weekday\_sunday',

'weekday\_thursday', 'weekday\_tuesday', 'weekday\_wednesday'

]

We then specify a rolling window size and apply the transformation:

window = 30 # Define the rolling window size

smooth\_df = pd.DataFrame()

# Perform split-apply-combine on the 'variable' column

for variable, group\_df in econ\_df\_melt.groupby('variable'):

if variable not in onehot\_cols:

group\_df['pct\_change'] = group\_df['value'].pct\_change() # Calculate percent change

colname = 'rolling\_' + str(window) + '\_mean'

group\_df[colname] = group\_df['pct\_change'].rolling(window=window).mean() # Calculate rolling mean

else:

group\_df[colname] = group\_df['value'] # Keep onehot columns unchanged

smooth\_df = smooth\_df.append(group\_df) # Append the processed data to smooth\_df

print(smooth\_df.head())

In this code, we iterate through each group of data determined by unique 'variable' values. For non-binary columns, we compute the percent change and then a rolling window mean. Binary columns are appended as is. This refined dataset, smooth\_df, now includes features that are more indicative of trends and patterns over time, enhancing its potential usefulness in modeling and analysis

| **Index** | **Date** | **pct\_change** | **rolling\_30\_mean** | **value** | **variable** |
| --- | --- | --- | --- | --- | --- |
| 44927 | 2023-01-01 | NaN | NaN | 479.31 | bamlc0a1aaatriv |
| 44928 | 2023-01-02 | 0.00000 | NaN | 479.31 | bamlc0a1aaatriv |
| 44929 | 2023-01-03 | 0.00000 | NaN | 479.31 | bamlc0a1aaatriv |
| 44930 | 2023-01-04 | -0.000542 | NaN | 479.05 | bamlc0a1aaatriv |
| 44931 | 2023-01-05 | -0.007348 | NaN | 475.53 | bamlc0a1aaatriv |

Data Visualization and Restoration to Original Structure

The data visualization involved creating time-series plots for each continuous feature, accompanied by an inverted histogram on the right side of each plot to examine the distribution of percent changes over a given window period.

import matplotlib.pyplot as plt

from matplotlib.dates import YearLocator, MonthLocator, DateFormatter

from mpl\_toolkits.axes\_grid1 import make\_axes\_locatable

import seaborn as sns

%matplotlib inline

# Setting locators and formatter for dates

yearly\_locator = YearLocator()

monthly\_locator = MonthLocator()

yearly\_formatter = DateFormatter('%Y')

def visualize\_data\_with\_histograms(dataframe, date\_column, series\_column,

group\_column, prepend\_title='{}', labels=None,

rotate\_x=0, pad\_x\_label=60, rolling\_window=15,

columns\_to\_exclude=[]):

'''

This function plots a time series and its rolling average, including a histogram

to the right side of the time series plot.

Parameters:

dataframe: DataFrame containing the time series data

date\_column: Column name for datetime data

series\_column: Column name for the time series data

group\_column: Column name for grouping the plots

labels: Dictionary containing labels for x and y axes

prepend\_title: String to prepend to the title of each subplot

rotate\_x: Degrees to rotate x-axis labels

pad\_x\_label: Padding for the x-axis label

rolling\_window: Window size for the rolling average

columns\_to\_exclude: List of column names to exclude from plotting

'''

unique\_groups = dataframe[group\_column].unique()

num\_rows = len(unique\_groups) - len(columns\_to\_exclude)

plot\_size = (13, 6 \* num\_rows)

fig, axis\_set = plt.subplots(num\_rows, 1, figsize=plot\_size)

prepend\_title\_hist = 'Histogram of ' + prepend\_title

counter = 0

for idx, group\_value in enumerate(unique\_groups):

subset\_df = dataframe[dataframe[group\_column] == group\_value]

if group\_value not in columns\_to\_exclude:

current\_axis = axis\_set[counter]

current\_axis.plot(subset\_df[date\_column], subset\_df[series\_column], alpha=0.2, color='black')

# Plot rolling averages with different windows

for multiple in [1, 3]:

roll\_window = rolling\_window \* multiple

subset\_df[f'rolling\_{roll\_window}\_avg'] = subset\_df[series\_column].rolling(window=roll\_window, min\_periods=min(5, roll\_window)).mean()

current\_axis.plot(subset\_df[date\_column], subset\_df[f'rolling\_{roll\_window}\_avg'], label=f'{roll\_window} period rolling avg')

# Statistical lines

mean = subset\_df[series\_column].mean()

std\_dev = subset\_df[series\_column].std()

for sigma\_multiplier in [-1, 0, 1]:

current\_axis.axhline(mean + sigma\_multiplier\*std\_dev, linestyle='--', color='red' if sigma\_multiplier == 0 else 'yellow', alpha=0.3)

current\_axis.set\_title(prepend\_title.format(group\_value))

current\_axis.legend(loc='best')

current\_axis.set\_ylim(mean - 3\*std\_dev, mean + 3\*std\_dev)

if labels:

current\_axis.set\_xlabel(labels['x\_label'])

current\_axis.set\_ylabel(labels['y\_label'])

current\_axis.xaxis.set\_minor\_locator(monthly\_locator)

current\_axis.grid(alpha=0.1)

if rotate\_x != 0:

plt.setp(current\_axis.get\_xticklabels(), rotation=rotate\_x)

# Histogram on the right

divider = make\_axes\_locatable(current\_axis)

right\_axis = divider.append\_axes('right', 1.2, pad=0.1, sharey=current\_axis)

right\_axis.grid

A graph of a graph showing the growth of a number of people

Description automatically generated with medium confidence

A graph with red lines and numbers

Description automatically generated

Sample time series plots above

Data reshaping:

To revert the data to its original wide format, used for the visualization, we execute a pivot operation. This reshaping is vital for our analyses that rely on the original structure of the data.

# Pivot the melted dataframe to restore original wide format with new features

reshaped\_df = smooth\_df.pivot(index='date', columns='variable', values='rolling\_30\_mean')

reshaped\_df.dropna(inplace=True)

# Display the first few records of the reshaped dataframe

reshaped\_df.head()

A screenshot of a computer

Description automatically generated

Visualize: Heatmap

To better understand the relationships between different financial indicators, we create a scatterplot matrix for selected columns. Additionally, we prepare the data for predicting the price of crude oil by mapping actual prices to our smoothed dataset and splitting the data for model training and evaluation.

Correlation Heatmap

The plot\_correlation\_heatmap function is designed to visualize the correlation between the selected financial indicators. It calculates the Pearson correlation coefficient between each pair of variables and then plots a heatmap. This gives an immediate visual representation of how strongly each pair of variables is related. The heatmap can be filtered using a threshold parameter to only display correlations that are above a certain absolute value, which helps in focusing on the most significant relationships.

A screenshot of a computer screen

Description automatically generated

Data modelling for Predicting crude oil price

Preparing Data for the Prediction Model

Mapping Target Values

We map the actual prices of crude oil (dcoilwtico) to our dataset (smooth\_df). This is done by creating a dictionary from the econ\_df and mapping these values to the smooth\_df based on the index, which represents dates. This step is crucial to align our features with the correct target variable that we aim to predict.

Shifting Target Values

Next, the target variable (dcoilwtico) is shifted backwards by a specified window period. This is because we want to predict the future value of crude oil based on current and past information. By shifting the data, we ensure that the features correspond to the state of the economy at the time of prediction, not at the time of the actual price.

Cleaning Up Data

We remove any rows that have NaN values resulting from the shifting process with dropna. This ensures that our dataset does not have any missing values which could lead to errors during the modeling process.

Data Export

The processed dataset is then saved to a CSV file for future use or analysis. This is a common practice to ensure data integrity and to provide checkpoints for long data processing workflows.

Splitting and Scaling Data

Data Splitting

We split our dataset into training and testing sets using the train\_test\_split function from sklearn.model\_selection. A portion of the data (specified by train\_size) is used to train the model, and the rest is reserved for testing its performance.

Feature Scaling

The StandardScaler from sklearn.preprocessing is used to scale the features. This is an important step in many machine learning algorithms to standardize the range of independent variables or features of data. It helps in speeding up the convergence of stochastic gradient descent and other optimization algorithms.

Final Test Set Split

We further split the test set into a validation set and a final test set. The validation set is used during the model training to fine-tune model parameters and prevent overfitting. The final test set is kept aside to evaluate the model's performance after training has concluded, providing an unbiased assessment of the model's predictive power.

Visualizing Model Training

Plot Training and Validation Loss

The visualize\_model\_loss function is used to plot the training and validation loss over each epoch during model training. This visualization helps in diagnosing issues with the learning process, such as overfitting or underfitting, and provides insights into the model's convergence behavior.

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

import pandas as pd

import numpy as np

%matplotlib inline

# Define visualization columns

visualization\_columns = [

'bamlcc0a1aaatriv', 'bamlhyh0a0hym2triv', 'bogmbasew', 'dcoilwtico',

'dexchus', 'dexjpus', 'dexusal', 'dexuseu', 'djia', 'goldamgbd228nlbm',

'mhhngsp', 'nasdaqcom', 'ru2000pr', 'sp500', 'usd12md156n', 'usd1mtd156n',

'usd3mtd156n', 'usdontd156n', 'vixcls', 'vxxlecls'

]

# Function to plot correlation heatmap

def plot\_correlation\_heatmap(data\_frame, threshold=None, plot\_title=''):

corr\_matrix = data\_frame.corr(method='pearson')

np.fill\_diagonal(corr\_matrix.values, 0)

if threshold is not None:

corr\_matrix = corr\_matrix[corr\_matrix.abs() > threshold].fillna(0)

plt.figure(figsize=(20, 15))

sns.heatmap(corr\_matrix, cmap='coolwarm', annot=False)

plt.title(plot\_title, fontsize=18)

plt.show()

return corr\_matrix

# Set cutoff for correlation and visualize heatmap

cutoff\_threshold = 0.3

plot\_correlation\_heatmap(smooth\_df[visualization\_columns], cutoff\_threshold, 'Filtered Correlation Heatmap')

# Prepare the data for predicting crude oil prices

target\_column = 'dcoilwtico'

smooth\_df[target\_column] = smooth\_df.index.map(econ\_df[target\_column].to\_dict())

smooth\_df[target\_column] = smooth\_df[target\_column].shift(periods=-window)

smooth\_df.dropna(inplace=True)

# Export the processed data

filename = 'processed\_data.csv'

smooth\_df.to\_csv(filename)

data['processed\_data'] = smooth\_df

# Data splitting and scaling

def prepare\_data\_for\_modeling(data\_frame, features, target, scale\_columns=None, size=0.7):

X = data\_frame[features]

y = data\_frame[target]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, train\_size=size)

scaler = StandardScaler()

if scale\_columns:

X\_train[scale\_columns] = scaler.fit\_transform(X\_train[scale\_columns])

X\_test[scale\_columns] = scaler.transform(X\_test[scale\_columns])

return X\_train, y\_train, X\_test, y\_test

# Identify columns to normalize and split the data

normalize\_columns = [col for col in smooth\_df if col not in onehot\_cols + [target\_column]]

feature\_columns = [col for col in smooth\_df if col != target\_column]

X\_train, y\_train, X\_test, y\_test = prepare\_data\_for\_modeling(smooth\_df, feature\_columns, target\_column, normalize\_columns)

# Further split for final evaluation

test\_cutoff = len(X\_test) // 2

X\_validation, y\_validation = X\_test[:test\_cutoff], y\_test[:test\_cutoff]

X\_final\_test, y\_final\_test = X\_test[test\_cutoff:], y\_test[test\_cutoff:]

# Define a function to plot model loss

def visualize\_model\_loss(history, plot\_width=11):

training\_loss = history['loss']

validation\_loss = history['val\_loss']

epochs = range(1, len(training\_loss) + 1)

sns.set\_style("white")

plt.figure(figsize=(plot\_width, 6))

plt.plot(epochs, training\_loss, 'g--', label='Training Loss')

plt.plot(epochs, validation\_loss, 'b-.', label='Validation Loss')

plt.title('Training & Validation Loss Over Epochs')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

Conducting a Grid Search for Neural Network Hyperparameter Optimization

Defining the Model Generation Function

We begin by defining a function named create\_nn\_model that simplifies the construction of a feedforward neural network with dense layers. This function allows us to experiment with different hyperparameters like the number of neurons per layer, the number of hidden layers, regularization strength, dropout rate, and more. Although it's set up to use certain defaults (like mean squared error for the loss and ReLU activation), it's flexible enough to test variations in activation functions, optimizers, and loss functions if desired.

Setting Up the Hyperparameter Space

We define a hyperparameter space by creating lists or ranges of possible values for the number of neurons (neurons\_range), the number of dense layers (layers\_range), dropout rates (dropout\_options), and regularization strengths (reg\_values). These are the parameters over which we will perform our grid search to find the best combination for our predictive model.

Performing the Grid Search

Using nested loops, we iterate over every possible combination of hyperparameters defined in our search space. For each combination, we:

Construct a new model with the create\_nn\_model function using the current set of hyperparameters.

Set up an EarlyStopping callback with patience=1, which will halt training if the validation loss doesn't improve, preventing unnecessary computations and potential overfitting.

Train the model on the training data while monitoring the validation loss.

Predict and evaluate the model's performance on the training, validation, and test datasets, calculating the R-squared value for each.

Saving and Analyzing Results

Each model's performance metrics and configuration are stored in a dictionary named results\_summary. This enables post-analysis and comparison between different models to select the best performing ones based on the test R-squared values.

# Pseudocode for illustrative purposes

import tensorflow as tf

from keras.models import Sequential

from keras.layers import Dense, Dropout

from keras.regularizers import l1\_l2

from keras.callbacks import EarlyStopping

from sklearn.metrics import r2\_score

def create\_nn\_model(neurons, layers, reg\_rate, dropout\_rate):

model = Sequential()

# Add input layer

model.add(Dense(neurons, activation='relu', input\_shape=(X\_train.shape[1],),

kernel\_regularizer=l1\_l2(l1=reg\_rate, l2=reg\_rate)))

# Add hidden layers

for \_ in range(layers):

model.add(Dense(neurons, activation='relu',

kernel\_regularizer=l1\_l2(l1=reg\_rate, l2=reg\_rate)))

model.add(Dropout(dropout\_rate))

# Add output layer

model.add(Dense(1))

model.compile(optimizer='rmsprop', loss='mse')

return model

# Hyperparameter ranges

dropout\_options = [0.0, 0.1, 0.2, 0.3]

neurons\_range = [16, 22, 28, 34]

layers\_range = [1, 2]

reg\_values = [0.005, 0.001, 0.0005]

results\_summary = {}

# Grid search

for neurons in neurons\_range:

for layers in layers\_range:

for dropout\_rate in dropout\_options:

for reg\_rate in reg\_values:

model = create\_nn\_model(neurons, layers, reg\_rate, dropout\_rate)

early\_stopping = EarlyStopping(monitor='val\_loss', patience=1)

history = model.fit(X\_train, y\_train, epochs=1000, batch\_size=X\_train.shape[0] // 4,

verbose=0, validation\_data=(X\_val, y\_val), callbacks=[early\_stopping])

# Generate model ID and store results

model\_id = f"{neurons}\_{layers}\_{dropout\_rate}\_{reg\_rate}"

results\_summary[model\_id] = evaluate\_model(model, X\_train, y\_train, X\_val, y\_val, X\_test, y\_test)

# Function to evaluate the model

def evaluate\_model(model, X\_train, y\_train, X\_val, y\_val, X\_test, y\_test):

# Make predictions

train\_pred = model.predict(X\_train)

val\_pred = model.predict(X\_val)

test\_pred = model.predict(X\_test)

# Calculate R-squared for each set

train\_r2 = r2\_score(y\_train, train\_pred)

val\_r2 = r2\_score(y\_val, val\_pred)

test\_r2 = r2\_score(y\_test, test\_pred)

# Continue the function to evaluate the model

return {

'train\_r2': train\_r2,

'val\_r2': val\_r2,

'test\_r2': test\_r2,

'history': history.history,

'neurons': neurons,

'layers': layers,

'dropout': dropout\_rate,

'regularizer': reg\_rate

}

# Iterate over the results to print the performance

for model\_id, metrics in results\_summary.items():

print(f"Model ID: {model\_id}")

print(f"Train R-squared: {metrics['train\_r2']:.4f}")

print(f"Validation R-squared: {metrics['val\_r2']:.4f}")

print(f"Test R-squared: {metrics['test\_r2']:.4f}")

print("-" \* 80)

# Optionally, plot the loss curves for the best performing models

def plot\_training\_curves(history):

plt.figure(figsize=(10, 5))

plt.plot(history['loss'], label='Training Loss')

plt.plot(history['val\_loss'], label='Validation Loss')

plt.title('Training vs Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

# Identify the best model based on validation R-squared

best\_model\_id = max(results\_summary, key=lambda id: results\_summary[id]['val\_r2'])

best\_model\_metrics = results\_summary[best\_model\_id]

print(f"Best Model ID: {best\_model\_id}")

print(f"Best Model Validation R-squared: {best\_model\_metrics['val\_r2']:.4f}")

# Plotting the training and validation loss curves for the best model

plot\_training\_curves(best\_model\_metrics['history'])

Evaluate the Model

The evaluate\_model function computes the R-squared metric for the training, validation, and test predictions to assess the model's performance. It returns a dictionary containing these metrics and the history of the training process.

Print Model Performances

The script iterates over the results\_summary dictionary, printing the R-squared values for each model. This allows a quick review of how each model performed.

Plot Training Curves

The plot\_training\_curves function is provided to visually examine the training process of the models. It plots the loss across epochs for both training and validation sets, which can indicate overfitting if the validation loss increases while the training loss decreases.

Select the Best Model

The best model is determined by finding the model ID with the highest validation R-squared in the results\_summary. Identifying the best model helps in understanding which combination of hyperparameters yields the most promising results on unseen data.

Visualize the Best Model's Performance

Finally, the training and validation loss curves of the best model are plotted. This visual assessment can help confirm the model's ability to learn from the training data without overfitting to it.

Model # 1

Fully Connected Model w/ Dropout & Regularization

- Regularizer Rate: 0.0050000

- Dropout Rate: 0.000

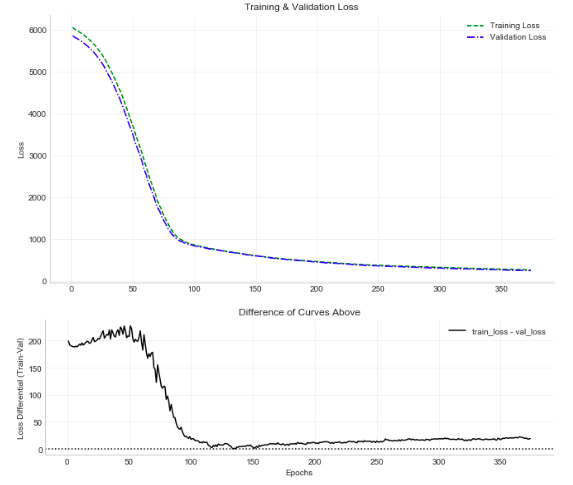
- Number Dense Layers: 1

- Neurons per Layer: 16

R-squared on training data = 0.8849

R-squared on validation data = 0.8662

R-squared on testing data = 0.8526



Model # 2

Fully Connected Model w/ Dropout & Regularization

- Regularizer Rate: 0.0010000

- Dropout Rate: 0.000

- Number Dense Layers: 1

- Neurons per Layer: 16

R-squared on training data = 0.8797

R-squared on validation data = 0.8545

R-squared on testing data = 0.8506

A graph of a training and validation loss

Description automatically generated

## Plot the Train, Test and Validation r2\_score

Visualization and Analysis of Model Performance

Visualizing R-squared Values

We'll convert the R-squared values into floating-point numbers for better precision during calculations. Then, we visualize the average R-squared values for the training, validation, and test sets, grouped by the number of neurons per layer. A horizontal bar chart is used for this visualization, allowing us to compare how the number of units affects the model's performance across different datasets.

# Convert R-squared columns to float for precision

results\_df[r2\_cols] = results\_df[r2\_cols].astype(np.float32)

# Group by the number of units and plot their average R-squared values

units\_performance = results\_df.groupby(['neurons'])[r2\_cols].mean()

units\_performance.plot(kind='barh', subplots=True, figsize=(10, 15), color=['#1f77b4', '#ff7f0e', '#2ca02c'])

plt.tight\_layout()

plt.show()

A graph with blue and white stripes

Description automatically generated

A graph with orange and white bars

Description automatically generated

A graph with green bars

Description automatically generated

Selecting the Optimum Model

To find the model with the highest R-squared value on the test set, we use the idxmax method. We then save this best-performing model to disk for future use, ensuring that the hard work of training doesn't need to be repeated.

# Identify the index of the best model based on test R-squared

optimal\_model\_index = results\_df['r2\_test'].idxmax()

# Save the best model to a file

optimal\_model = models\_archive[optimal\_model\_index]['model']

optimal\_model.save('optimal\_model.h5')

# Plot actual vs predicted values to assess model performance visually

predictions = optimal\_model.predict(X\_test)

plt.figure(figsize=(15, 7))

plt.scatter(y\_test, predictions, edgecolors='k', alpha=0.7)

plt.grid(True, which='both', linestyle='--', linewidth=0.5)

plt.xlabel('Actual Values')

plt.ylabel('Predicted Values')

plt.title('Comparison of Actual and Predicted Values', fontsize=16)

sns.despine()

plt.show()

units 34

layers 2

dropout 0

regularizer 0.005

r2\_train 0.956788

r2\_val 0.938761

r2\_test 0.945236

Examining the Best Model

We delve deeper into the architecture and performance of the best model by plotting its training history and examining its summary. Additionally, we review the specific record in our results dataframe that corresponds to the best model to understand its configuration.

# Plot the training history of the best model

optimal\_model\_history = models\_archive[optimal\_model\_index]['history']

plot\_training\_history(optimal\_model\_history)

plt.savefig('optimal\_model\_training\_history.png')

# Print a summary of the best model to understand its structure

print("Best Model Summary:")

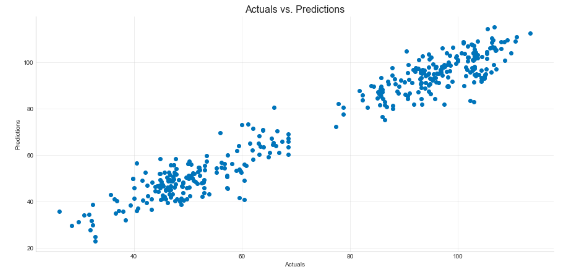
optimal\_model.summary()

# Display the best model's hyperparameters and performance metrics

optimal\_model\_info = results\_df.loc[optimal\_model\_index]

print("Best Model Info:")

print(optimal\_model\_info)



We see here that a model that uses 22 units per layer, two layers (note that the first layer is not counted in the num\_layers parameter), no dropout and an l1/l2 regularizer of 0.005. The

score for all data sets is very high, which should give cause for suspicion and further investigation.

Prediction Generation and Evaluation

Data Preprocessing Function

We define a function prepare\_data\_for\_prediction that takes a DataFrame and the names of the feature columns, target column, and any columns that require normalization. This function standardizes the feature data, aligning it with the preprocessing steps our model expects.

def prepare\_data\_for\_prediction(data, feature\_names, target\_name, columns\_to\_normalize=None):

# Extract features and target

features = data[feature\_names]

target = data[target\_name]

# Initialize standard scaler

scaler = StandardScaler()

# Normalize features if specified

if columns\_to\_normalize:

features[columns\_to\_normalize] = scaler.fit\_transform(features[columns\_to\_normalize])

else:

features = pd.DataFrame(scaler.fit\_transform(features), columns=features.columns)

# Return the prepared features and target

return features, target

# Prepare the data for prediction

prepared\_X, prepared\_Y = prepare\_data\_for\_prediction(smoothed\_dataframe, feature\_columns, target\_column, columns\_to\_normalize)

Visualization of Predictions

We generate predictions for the entire dataset using our optimal model and visualize these predictions alongside the actual target values. We also plot the errors to assess where the model's predictions diverge from reality.

# Generate predictions

predictions\_array = optimal\_model.predict(prepared\_X.to\_numpy()).flatten()

predictions\_series = pd.Series(predictions\_array, index=pd.to\_datetime(prepared\_X.index))

# Create a DataFrame to compare predictions with actuals

comparison\_df = pd.DataFrame({'predicted': predictions\_series, 'actual': prepared\_Y}).dropna()

# Plotting the predictions and actuals

plt.figure(figsize=(15, 7))

plt.plot(comparison\_df.index, comparison\_df['predicted'], label='Predicted', linestyle=':', color='green', alpha=0.9)

plt.plot(comparison\_df.index, comparison\_df['actual'], label='Actual', color='black', alpha=0.7, linewidth=0.5)

plt.xlabel('Date')

plt.ylabel('Crude Oil Price')

plt.title('Comparison of Predicted and Actual Crude Oil Prices', fontsize=16)

plt.legend()

plt.grid(visible=True, which='both', linestyle='--', linewidth=0.5)

sns.despine()

plt.show()

# Plotting the prediction errors

plt.figure(figsize=(15, 3))

prediction\_errors = comparison\_df['predicted'] - comparison\_df['actual']

plt.plot(comparison\_df.index, prediction\_errors, label='Prediction Error', linestyle='--', color='red', alpha=0.5)

plt.xlabel('Date')

plt.ylabel('Prediction Error')

plt.title('Prediction Errors Over Time', fontsize=16)

plt.legend()

plt.grid(visible=True, which='both', linestyle='--', linewidth=0.5)

sns.despine()

plt.show()

A table with numbers and letters

Description automatically generated

# Calculating and printing the R-squared value for the final predictions

final\_r2\_score = r2\_score(comparison\_df['actual'], comparison\_df['predicted'])

print(f"R-squared for the entire data

A graph of a graph of a model

Description automatically generated with medium confidence

R-squared on entire dataset: 0.9540