1. Define the problem statement

Problem Statement:

Porter, India's Largest Marketplace for Intra-City Logistics, aims to improve the efficiency of its delivery operations by accurately estimating the delivery time for orders placed through its platform. With a vast network of delivery partners and a wide range of restaurants offering delivery services, Porter seeks to provide customers with reliable estimates of when their orders will be delivered.

Objective:

The primary objective is to develop a regression model that can predict the delivery time for orders placed on the Porter platform. The model will utilize various features such as the items ordered, restaurant category, order protocol, and delivery partner availability to estimate the time taken for delivery accurately.

Key Deliverables:

1. A regression model capable of predicting delivery time based on input features.
2. Evaluation metrics to assess the model's performance, such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE).
3. Insights into factors influencing delivery time and recommendations for optimizing delivery operations.

Stakeholders:

- Porter: Seeks to improve operational efficiency and customer satisfaction.
- Customers: Expect accurate delivery estimates for their orders.
- Delivery Partners: Benefit from optimized allocation and scheduling of delivery tasks.

By accurately estimating delivery time, Porter can enhance customer experience, optimize resource allocation, and improve overall operational efficiency in its intra-city logistics operations.

This problem statement sets the foundation for the subsequent steps in the analysis and model development process.

2. Import the data & Analyze its structure

In [1]:
```
# Import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
# Load the dataset
porter_data = pd.read_csv('porter_data.csv')
```

In [3]:
```
# view all the variable names
porter_data.columns
```

Out[3]:
```
Index(['market_id', 'created_at', 'actual_delivery_time', 'store_id',
       'store_primary_category', 'order_protocol', 'total_items', 'subtotal',
       'num_distinct_items', 'min_item_price', 'max_item_price',
       'total_onshift_partners', 'total_busy_partners',
       'total_outstanding_orders'],
      dtype='object')
```

In [4]:
```
# Display basic information
porter_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 197428 entries, 0 to 197427
Data columns (total 14 columns):
 #   Column                  Non-Null Count   Dtype  
---  ------                  --------------   -----  
 0   market_id               196441 non-null  float64
 1   created_at              197428 non-null  object 
 2   actual_delivery_time    197421 non-null  object 
 3   store_id                197428 non-null  object 
 4   store_primary_category  192668 non-null  object 
 5   order_protocol          196433 non-null  float64
 6   total_items             197428 non-null  int64  
 7   subtotal                197428 non-null  int64  
 8   num_distinct_items      197428 non-null  int64  
 9   min_item_price          197428 non-null  int64  
 10  max_item_price          197428 non-null  int64  
 11  total_onshift_partners  181166 non-null  float64
 12  total_busy_partners     181166 non-null  float64
 13  total_outstanding_orders  181166 non-null  float64
dtypes: float64(5), int64(5), object(4)
memory usage: 21.1+ MB
```

In [5]:
```
# Display the first few rows of the dataset
porter_data.head()
```

Out[5]:

	market_id	created_at	actual_delivery_time	store_id	store_p
0	1.0	2015-02-06 22:24:17	2015-02-06 23:27:16	df263d996281d984952c07998dc54358	
1	2.0	2015-02-10 21:49:25	2015-02-10 22:56:29	f0ade77b43923b38237db569b016ba25	
2	3.0	2015-01-22 20:39:28	2015-01-22 21:09:09	f0ade77b43923b38237db569b016ba25	
3	3.0	2015-02-03 21:21:45	2015-02-03 22:13:00	f0ade77b43923b38237db569b016ba25	
4	3.0	2015-02-15 02:40:36	2015-02-15 03:20:26	f0ade77b43923b38237db569b016ba25	

In [6]:
```
# Check the shape of the dataset
print("Shape of the dataset:-")
print("No. of rows: ", porter_data.shape[0])
print("No. of columns: ", porter_data.shape[1])
```

```
Shape of the dataset:-
No. of rows:  197428
No. of columns:  14
```

In [7]:
```
# Check data types of all attributes
porter_data.dtypes
```

Out[7]:
```
market_id                float64
created_at                object
actual_delivery_time      object
store_id                  object
store_primary_category    object
order_protocol           float64
total_items                int64
subtotal                   int64
num_distinct_items         int64
min_item_price             int64
max_item_price             int64
total_onshift_partners   float64
total_busy_partners      float64
total_outstanding_orders float64
dtype: object
```

In [8]:
```
# Check for missing values
porter_data.isnull().sum()
```

Out[8]:
```
market_id                  987
created_at                   0
actual_delivery_time         7
store_id                     0
store_primary_category    4760
order_protocol             995
total_items                  0
subtotal                     0
num_distinct_items           0
min_item_price               0
max_item_price               0
total_onshift_partners   16262
total_busy_partners      16262
total_outstanding_orders 16262
dtype: int64
```

In [9]:
```
# check statistical summary of the dataset
porter_data.describe(include = 'all')
```

Out[9]:

	market_id	created_at	actual_delivery_time	store_id
count	196441.000000	197428	197421	19742
unique	NaN	180985	178110	674
top	NaN	2015-02-11 19:50:43	2015-02-11 20:40:45	d43ab110ab2489d6b9b2caa394bf920
freq	NaN	6	5	93
mean	2.978706	NaN	NaN	NaN
std	1.524867	NaN	NaN	NaN
min	1.000000	NaN	NaN	NaN
25%	2.000000	NaN	NaN	NaN
50%	3.000000	NaN	NaN	NaN
75%	4.000000	NaN	NaN	NaN
max	6.000000	NaN	NaN	NaN

3. Data Preprocessing & Feature Engineering

In [10]:
```
# Convert 'created_at' and 'actual_delivery_time' columns to datetime
porter_data['created_at'] = pd.to_datetime(porter_data['created_at'])
porter_data['actual_delivery_time'] = pd.to_datetime(porter_data['actual_delivery_
```

In [11]:
```
# Create the target column 'delivery_time_mins'
porter_data['delivery_time_mins'] = (porter_data['actual_delivery_time'] -
                                     porter_data['created_at']).dt.total_seconds()
```

In [12]:
```
# Extract hour and day of the week from 'created_at'
porter_data['order_hour'] = porter_data['created_at'].dt.hour
porter_data['order_day'] = porter_data['created_at'].dt.day_name()
```

In [13]:
```
# Encode categorical columns
categorical_cols = ['store_primary_category', 'order_protocol', 'order_day']
porter_data = pd.get_dummies(porter_data, columns=categorical_cols)
```

In [14]:
```
# Handle missing values
porter_data = porter_data.fillna(-1)
```

In [15]:
```
# Display the updated DataFrame
porter_data.head()
```

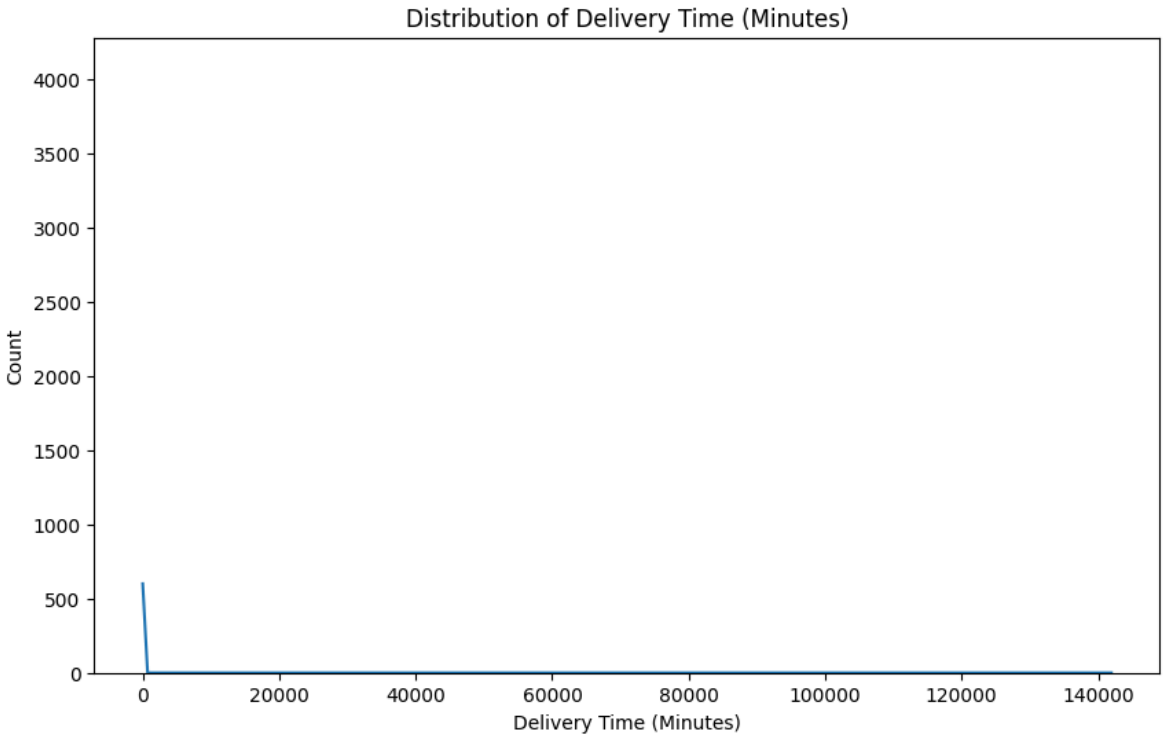
Out[15]:

	market_id	created_at	actual_delivery_time	store_id	total_i
0	1.0	2015-02-06 22:24:17	2015-02-06 23:27:16	df263d996281d984952c07998dc54358	
1	2.0	2015-02-10 21:49:25	2015-02-10 22:56:29	f0ade77b43923b38237db569b016ba25	
2	3.0	2015-01-22 20:39:28	2015-01-22 21:09:09	f0ade77b43923b38237db569b016ba25	
3	3.0	2015-02-03 21:21:45	2015-02-03 22:13:00	f0ade77b43923b38237db569b016ba25	
4	3.0	2015-02-15 02:40:36	2015-02-15 03:20:26	f0ade77b43923b38237db569b016ba25	

5 rows × 102 columns

4. Data Visualization

In [16]:
```
# Visualize the target variable 'delivery_time_mins'
plt.figure(figsize=(10, 6))
sns.histplot(porter_data['delivery_time_mins'], kde=True)
plt.title('Distribution of Delivery Time (Minutes)')
plt.xlabel('Delivery Time (Minutes)')
plt.ylabel('Count')
plt.show()
```

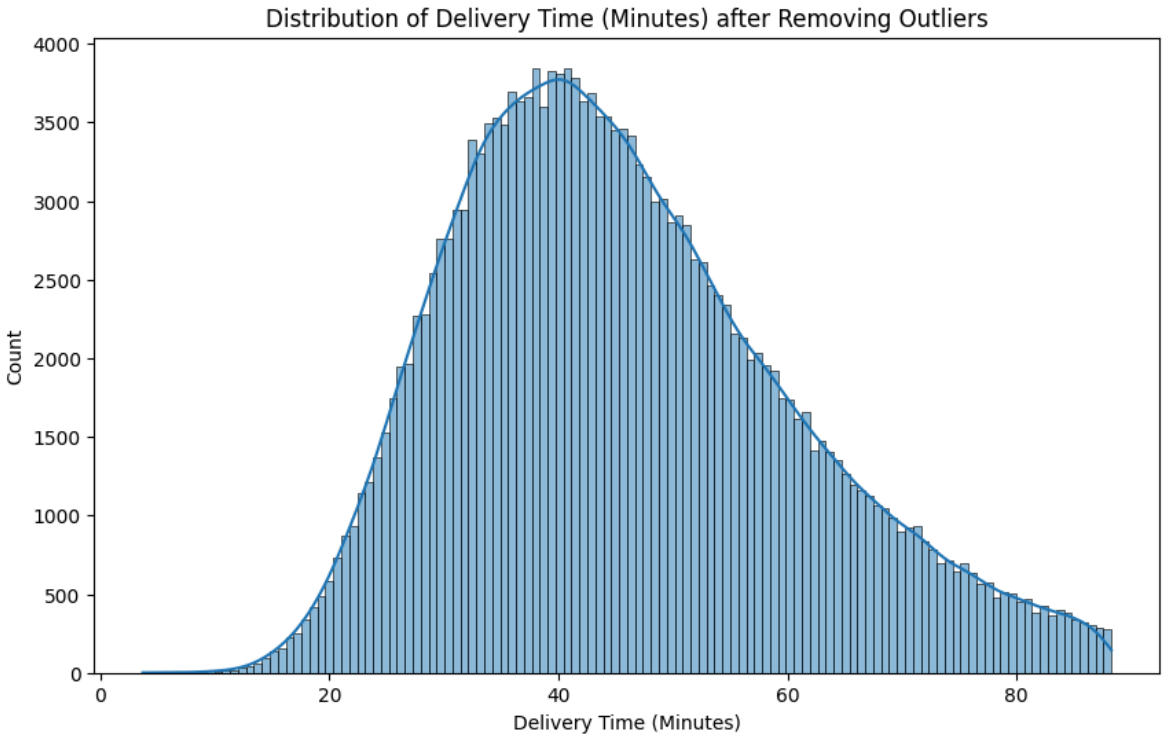


In [17]:
```
# Check for outliers in the target variable
Q1 = porter_data['delivery_time_mins'].quantile(0.25)
Q3 = porter_data['delivery_time_mins'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - (1.5 * IQR)
upper_bound = Q3 + (1.5 * IQR)

outliers = porter_data[(porter_data['delivery_time_mins']
                        < lower_bound) | (porter_data['delivery_time_mins'] > uppe

print(f"Number of outliers: {len(outliers)}")
```

```
Number of outliers: 6285
```

In [18]:
```
# Remove outliers from the dataset
porter_data = porter_data[(porter_data['delivery_time_mins']
                           >= lower_bound) & (porter_data['delivery_time_mins'] <=
```

In [19]:
```
# Visualize the target variable after removing outliers
plt.figure(figsize=(10, 6))
sns.histplot(porter_data['delivery_time_mins'], kde=True)
plt.title('Distribution of Delivery Time (Minutes) after Removing Outliers')
plt.xlabel('Delivery Time (Minutes)')
plt.ylabel('Count')
plt.show()
```



In this code:

1. We visualize the distribution of the target variable 'delivery_time_mins' using a histogram plot with a kernel density estimate (kde) curve overlaid.
2. We check for outliers in the target variable using the interquartile range (IQR) method:
   - We calculate the first quartile (Q1) and third quartile (Q3) of the 'delivery_time_mins' column.
   - We define the lower and upper bounds for outliers using the formula: `lower_bound = Q1 - (1.5 * IQR)` and `upper_bound = Q3 + (1.5 *`

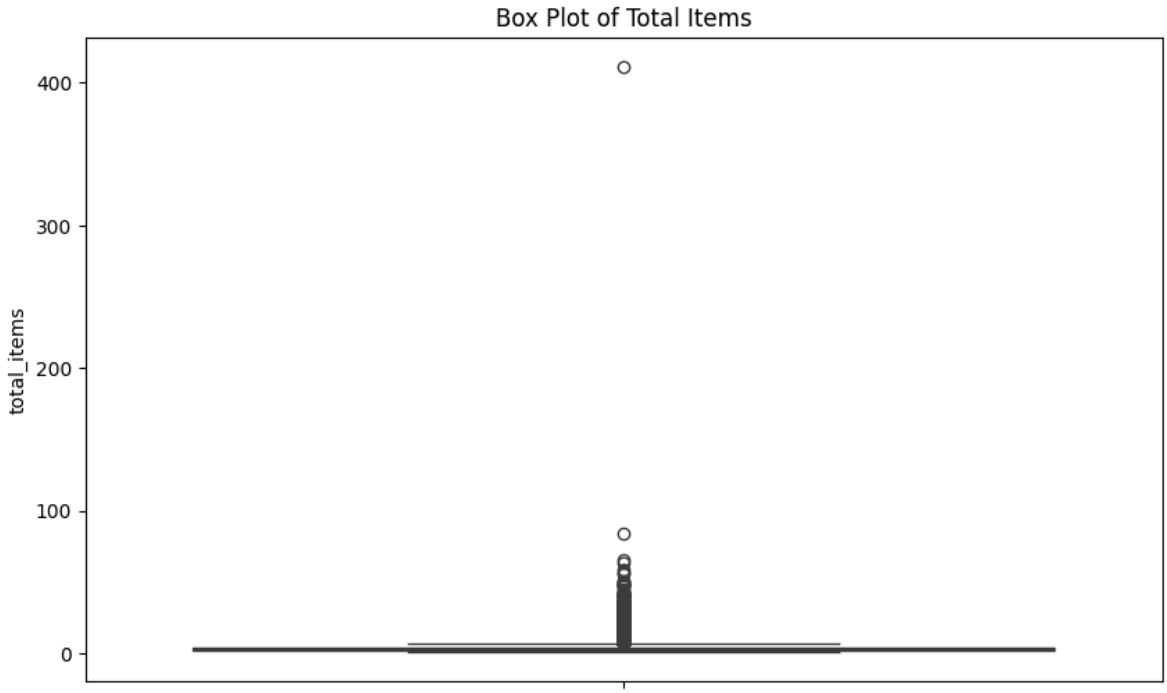
`IQR)`.

- We identify the outliers as the rows where 'delivery_time_mins' is either less than the lower bound or greater than the upper bound.
- We print the number of identified outliers.

3. We remove the outliers from the dataset by creating a new DataFrame `porter_data` that contains only the rows where 'delivery_time_mins' is within the lower and upper bounds.
4. We visualize the distribution of the target variable 'delivery_time_mins' after removing outliers using another histogram plot with a kde curve.

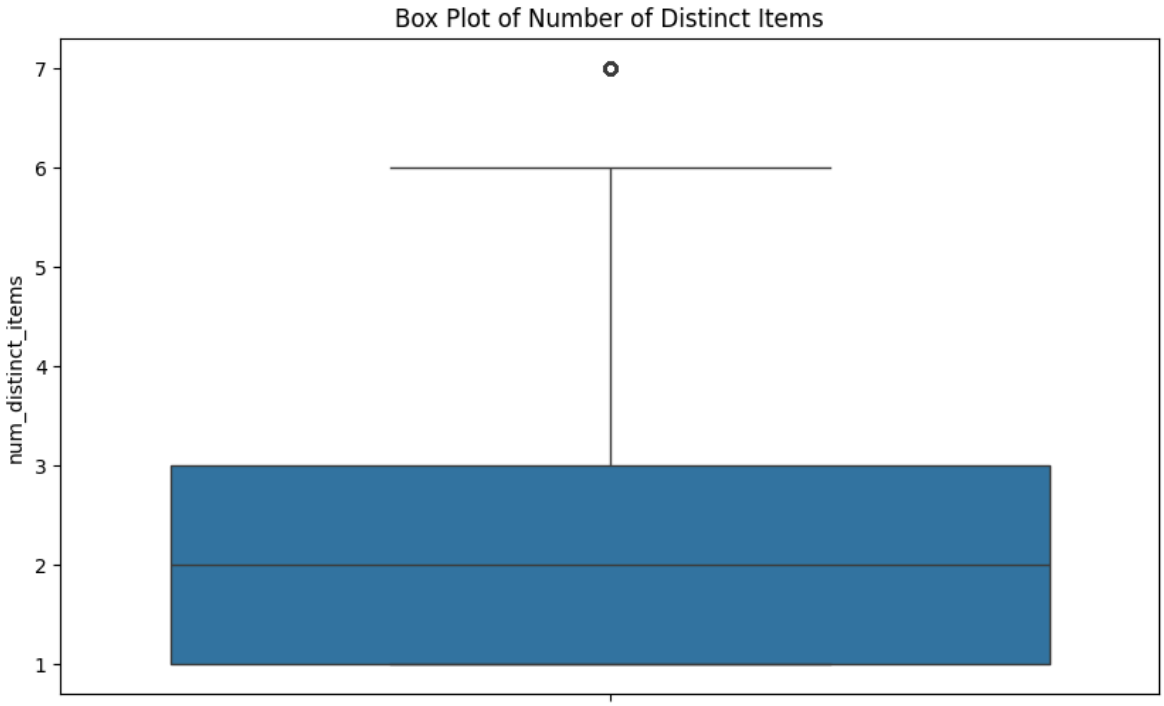
By visualizing the data and handling outliers, we can better understand the distribution of the target variable and prepare the dataset for model development.

In [20]:
```
# Visualize 'total_items_subtotal' and handle outliers
plt.figure(figsize=(10, 6))
sns.boxplot(porter_data['total_items'])
plt.title('Box Plot of Total Items')
plt.show()
```



In [21]:
```
# Remove outliers from 'total_items_subtotal'
Q1 = porter_data['total_items'].quantile(0.25)
Q3 = porter_data['total_items'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - (1.5 * IQR)
upper_bound = Q3 + (1.5 * IQR)
porter_data = porter_data[(porter_data['total_items']
                           >= lower_bound) & (porter_data['total_items'] <= upper_
```

In [22]:
```
# Visualize 'num_distinct_items' and handle outliers
plt.figure(figsize=(10, 6))
sns.boxplot(porter_data['num_distinct_items'])
plt.title('Box Plot of Number of Distinct Items')
plt.show()
```



In [23]:
```
# Remove outliers from 'num_distinct_items'
Q1 = porter_data['num_distinct_items'].quantile(0.25)
Q3 = porter_data['num_distinct_items'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - (1.5 * IQR)
upper_bound = Q3 + (1.5 * IQR)
porter_data = porter_data[(porter_data['num_distinct_items']
                           >= lower_bound) & (porter_data['num_distinct_items'] <=

# Print the updated number of rows
print(f"Number of rows after handling outliers: {len(porter_data)}")
```

```
Number of rows after handling outliers: 180991
```

In this code:

1. We visualize the distribution of the 'total_items_subtotal' column using a box plot to identify potential outliers.
2. We remove outliers from the 'total_items_subtotal' column using the IQR method, similar to how we handled outliers for the target variable.
3. We visualize the distribution of the 'num_distinct_items' column using a box plot.
4. We remove outliers from the 'num_distinct_items' column using the IQR method.
5. Finally, we print the updated number of rows in the dataset after handling outliers for all the visualized columns.

By visualizing and handling outliers for other relevant columns, we can further clean the dataset and prepare it for model development. Box plots are useful for identifying outliers in continuous variables, as they clearly show the distribution and potential extreme values.

5. Split the data in train and test

In [24]:
```
# Import libraries
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
# Separate numeric and categorical/datetime columns
numeric_cols = porter_data.select_dtypes(include=['float64', 'int64']).columns
categorical_cols = porter_data.select_dtypes(exclude=['float64', 'int64']).columns

# Get the updated column names after one-hot encoding
updated_cols = porter_data.columns.tolist()

X_numeric = porter_data[numeric_cols]
X_categorical = porter_data[categorical_cols]

y = porter_data['delivery_time_mins']
```

In [25]:
```
# Split the numeric data
X_train_numeric, X_test_numeric, y_train, y_test = train_test_split(X_numeric, y,
                                                                    test_size=0.2,

# Split the categorical data
X_train_categorical, X_test_categorical = train_test_split(X_categorical,
                                                           test_size=0.2, random_s

# Scale the numeric data
scaler = StandardScaler()
X_train_numeric_scaled = scaler.fit_transform(X_train_numeric)
X_test_numeric_scaled = scaler.transform(X_test_numeric)
```

In [26]:
```
# Convert scaled numeric data back to DataFrame to concatenate with categorical da
X_train_numeric_scaled_df = pd.DataFrame(X_train_numeric_scaled,
                                         columns=numeric_cols, index=X_train_numer
X_test_numeric_scaled_df = pd.DataFrame(X_test_numeric_scaled,
                                        columns=numeric_cols, index=X_test_numeric

# Concatenate numeric and categorical/datetime columns
X_train_scaled = pd.concat([X_train_numeric_scaled_df,
                            X_train_categorical], axis=1)
X_test_scaled = pd.concat([X_test_numeric_scaled_df,
                           X_test_categorical], axis=1)

print(f"Training data shape: {X_train_scaled.shape}")
print(f"Testing data shape: {X_test_scaled.shape}")
```

```
Training data shape: (144792, 102)
Testing data shape: (36199, 102)
```

In this code:

1. We separate the numeric columns and categorical/datetime columns using `porter_data.select_dtypes()`.
2. We create `X_numeric` and `X_categorical` DataFrames to hold the numeric and categorical/datetime columns, respectively.
3. We split the numeric columns and target variable into train and test sets using `train_test_split`.
4. We create `X_train_categorical` and `X_test_categorical` DataFrames to hold the categorical/datetime columns for the respective train and test sets.
5. We scale only the numeric columns using `StandardScaler`. The categorical/datetime columns are not scaled.

6. After scaling the numeric columns, we concatenate them with the categorical/datetime columns using `np.concatenate` for both train and test sets.
7. Finally, we print the shapes of the scaled train and test data to verify the dimensions.

By separating the numeric and categorical/datetime columns before scaling, we ensure that the `StandardScaler` only operates on the numeric features, avoiding the error caused by attempting to scale non-numeric data types.

6. Scaling the data for neural networks

In [27]:
```
# Import necessary libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Separate numeric and categorical/datetime columns
numeric_cols = X_train_scaled.select_dtypes(include=['float64', 'int64']).columns
categorical_cols = X_train_scaled.select_dtypes(exclude=['float64', 'int64']).colu

X_train_numeric = X_train_scaled[numeric_cols]
X_test_numeric = X_test_scaled[numeric_cols]
```

In [28]:
```
# Define the input shape
input_shape = X_train_numeric.shape[1]

# Define the model architecture
inputs = keras.Input(shape=(input_shape,))
x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(32, activation='relu')(x)
outputs = layers.Dense(1)(x)  # Output layer with single node for regression

model = keras.Model(inputs=inputs, outputs=outputs)
```

In [29]:
```
# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
history = model.fit(X_train_numeric, y_train, epochs=100, batch_size=32, validatic

# Evaluate the model on the test set
test_loss = model.evaluate(X_test_numeric, y_test, verbose=0)
print(f"Test Loss: {test_loss}")
```

```
Epoch 1/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 14s 3ms/step - loss: 292.5367 - val_loss: 0.0880
Epoch 2/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0666 - val_loss: 0.0121
Epoch 3/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0180 - val_loss: 0.0085
Epoch 4/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0066 - val_loss: 0.0010
Epoch 5/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 12s 3ms/step - loss: 0.0041 - val_loss: 0.0012
Epoch 6/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0118 - val_loss: 0.0162
Epoch 7/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0043 - val_loss: 0.0061
Epoch 8/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 12s 3ms/step - loss: 0.0036 - val_loss: 1.3044e-04
Epoch 9/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0031 - val_loss: 0.0019
Epoch 10/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0038 - val_loss: 8.4727e-04
Epoch 11/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0040 - val_loss: 8.4861e-04
Epoch 12/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0027 - val_loss: 1.4207e-04
Epoch 13/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0020 - val_loss: 4.2657e-04
Epoch 14/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0024 - val_loss: 1.8879e-04
Epoch 15/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0022 - val_loss: 1.2767e-04
Epoch 16/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0045 - val_loss: 3.4709e-04
Epoch 17/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0017 - val_loss: 0.0030
Epoch 18/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0038 - val_loss: 3.4736e-05
Epoch 19/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0019 - val_loss: 0.0069
Epoch 20/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0030 - val_loss: 0.0130
Epoch 21/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0034 - val_loss: 0.0042
Epoch 22/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0020 - val_loss: 0.0072
Epoch 23/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0045 - val_loss: 1.2075e-04
Epoch 24/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0118 - val_loss: 3.3738e-04
Epoch 25/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0077 - val_loss: 8.2232e-05
Epoch 26/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0026 - val_loss: 2.0015e-04
Epoch 27/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0028 - val_loss: 4.7148e-05
Epoch 28/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0021 - val_loss: 4.1881e-04
Epoch 29/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0030 - val_loss: 6.3575e-05
Epoch 30/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0018 - val_loss: 0.0016
Epoch 31/100
```

```
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0014 - val_loss: 1.1463e-04
Epoch 32/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0026 - val_loss: 0.0014
Epoch 33/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0023 - val_loss: 4.4840e-05
Epoch 34/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0016 - val_loss: 3.7283e-05
Epoch 35/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0021 - val_loss: 5.4394e-04
Epoch 36/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0014 - val_loss: 8.6435e-05
Epoch 37/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 0.0032 - val_loss: 5.7853e-05
Epoch 38/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 26s 7ms/step - loss: 0.0016 - val_loss: 0.0157
Epoch 39/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0033 - val_loss: 6.6894e-05
Epoch 40/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 0.0020 - val_loss: 2.1122e-04
Epoch 41/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0029 - val_loss: 2.3778e-04
Epoch 42/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0021 - val_loss: 3.4418e-05
Epoch 43/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0029 - val_loss: 3.1213e-05
Epoch 44/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0019 - val_loss: 5.2636e-05
Epoch 45/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0013 - val_loss: 3.8146e-05
Epoch 46/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 4.7407e-04 - val_loss: 9.9822e-0
4
Epoch 47/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0020 - val_loss: 1.4568e-04
Epoch 48/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0033 - val_loss: 5.8035e-05
Epoch 49/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0015 - val_loss: 0.0040
Epoch 50/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0032 - val_loss: 1.1464e-04
Epoch 51/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0019 - val_loss: 0.0771
Epoch 52/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0050 - val_loss: 4.7697e-05
Epoch 53/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0018 - val_loss: 9.2261e-05
Epoch 54/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0017 - val_loss: 6.3971e-05
Epoch 55/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.0016 - val_loss: 3.5336e-05
Epoch 56/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0016 - val_loss: 0.0024
Epoch 57/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0036 - val_loss: 3.2073e-05
Epoch 58/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0013 - val_loss: 4.5446e-05
Epoch 59/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0013 - val_loss: 2.6269e-04
Epoch 60/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0019 - val_loss: 3.6365e-05
Epoch 61/100
```

```
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 6.0113e-04 - val_loss: 3.2459e-0
5
Epoch 62/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0017 - val_loss: 3.5549e-05
Epoch 63/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0013 - val_loss: 0.0020
Epoch 64/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0056 - val_loss: 5.4527e-05
Epoch 65/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 9.5671e-04 - val_loss: 4.7091e-0
5
Epoch 66/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0015 - val_loss: 7.5077e-05
Epoch 67/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0018 - val_loss: 0.0052
Epoch 68/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0017 - val_loss: 3.6869e-05
Epoch 69/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 7.3098e-04 - val_loss: 0.0026
Epoch 70/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0033 - val_loss: 4.3732e-05
Epoch 71/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 9.7656e-04 - val_loss: 5.2903e-0
5
Epoch 72/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0015 - val_loss: 3.6609e-05
Epoch 73/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 6.7850e-04 - val_loss: 6.7763e-0
4
Epoch 74/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0014 - val_loss: 4.9490e-05
Epoch 75/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0018 - val_loss: 3.9462e-05
Epoch 76/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0019 - val_loss: 4.0168e-05
Epoch 77/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - loss: 9.0569e-04 - val_loss: 3.5622e-
04
Epoch 78/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0023 - val_loss: 4.6098e-04
Epoch 79/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0015 - val_loss: 5.2383e-05
Epoch 80/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0013 - val_loss: 4.3357e-04
Epoch 81/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0022 - val_loss: 2.3851e-05
Epoch 82/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0013 - val_loss: 0.0012
Epoch 83/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0037 - val_loss: 3.8168e-05
Epoch 84/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0027 - val_loss: 3.1147e-05
Epoch 85/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 7.9620e-04 - val_loss: 6.5345e-0
5
Epoch 86/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 7.6518e-04 - val_loss: 1.2681e-0
4
Epoch 87/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0017 - val_loss: 3.7696e-05
Epoch 88/100
```

```
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0013 - val_loss: 6.1904e-05
Epoch 89/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0042 - val_loss: 2.5029e-04
Epoch 90/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0013 - val_loss: 4.2284e-05
Epoch 91/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0012 - val_loss: 7.1442e-05
Epoch 92/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 0.0017 - val_loss: 6.8080e-04
Epoch 93/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0018 - val_loss: 4.6486e-05
Epoch 94/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0012 - val_loss: 4.3747e-05
Epoch 95/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0022 - val_loss: 2.1755e-04
Epoch 96/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 8.8243e-04 - val_loss: 2.8590e-0
4
Epoch 97/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0026 - val_loss: 5.0291e-05
Epoch 98/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0016 - val_loss: 4.0554e-05
Epoch 99/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0012 - val_loss: 5.5959e-05
Epoch 100/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 5.8713e-04 - val_loss: 3.7553e-0
4
Test Loss: 0.0003574575821403414
```

In this code:

1. We first define the input shape based on the number of numeric columns in `X_train_numeric`.
2. Instead of creating a `Sequential` model directly, we create an `Input` layer using `keras.Input(shape=(input_shape,))`.
3. We define the subsequent layers by passing the output of the previous layer as input.
   - The first layer is a `Dense` layer with 64 neurons and ReLU activation, taking the `inputs` layer as input.
   - The second layer is a `Dense` layer with 32 neurons and ReLU activation, taking the output of the previous layer as input.
   - The final layer is a `Dense` layer with a single neuron (no activation), which will output the predicted delivery time.
4. We create the model using `keras.Model(inputs=inputs, outputs=outputs)`, where `inputs` is the `Input` layer, and `outputs` is the output of the final layer.
5. The rest of the code (compiling, training, and evaluating the model) remains the same.

By using the `Input` layer as the first layer in the model and defining the subsequent layers by passing the output of the previous layer as input, we follow the recommended practice for defining Sequential models in Keras, and the warning should no longer appear.

7. Exploring Neural Network Configurations and Hyperparameters

In [30]:
```
# Define the model architecture
inputs = keras.Input(shape=(input_shape,))
```

```
# Try different configurations
x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(32, activation='relu')(x)
outputs = layers.Dense(1)(x)  # Output layer with single node for regression

# Create the model
model = keras.Model(inputs=inputs, outputs=outputs)

# Try different optimizers
optimizer = keras.optimizers.Adam(learning_rate=0.001)  # Adam optimizer with lear

# Compile the model
model.compile(optimizer=optimizer, loss='mean_squared_error')

# Train the model
history = model.fit(X_train_numeric, y_train, epochs=100,
                    batch_size=32, validation_split=0.2, verbose=1)
```

```
Epoch 1/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 13s 3ms/step - loss: 294.2288 - val_loss: 0.0990
Epoch 2/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0584 - val_loss: 0.0189
Epoch 3/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 12s 3ms/step - loss: 0.0112 - val_loss: 0.0038
Epoch 4/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 12s 3ms/step - loss: 0.0055 - val_loss: 0.0013
Epoch 5/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 21s 4ms/step - loss: 0.0072 - val_loss: 0.0012
Epoch 6/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 12s 3ms/step - loss: 0.0045 - val_loss: 0.0011
Epoch 7/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0048 - val_loss: 3.9370e-04
Epoch 8/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0036 - val_loss: 2.4630e-04
Epoch 9/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0034 - val_loss: 1.2504e-04
Epoch 10/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0038 - val_loss: 0.0015
Epoch 11/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0027 - val_loss: 9.3817e-05
Epoch 12/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0051 - val_loss: 0.0011
Epoch 13/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0033 - val_loss: 0.0019
Epoch 14/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0023 - val_loss: 0.0017
Epoch 15/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0030 - val_loss: 6.5067e-05
Epoch 16/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0027 - val_loss: 7.5653e-05
Epoch 17/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0030 - val_loss: 0.0038
Epoch 18/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0040 - val_loss: 7.6164e-04
Epoch 19/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0025 - val_loss: 0.0012
Epoch 20/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0043 - val_loss: 2.7843e-05
Epoch 21/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0020 - val_loss: 4.6264e-05
Epoch 22/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0047 - val_loss: 6.0924e-05
Epoch 23/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 8.4059e-04 - val_loss: 1.3130e-
04
Epoch 24/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0029 - val_loss: 8.2318e-05
Epoch 25/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0026 - val_loss: 0.0028
Epoch 26/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0023 - val_loss: 1.5421e-04
Epoch 27/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0032 - val_loss: 0.0010
Epoch 28/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0032 - val_loss: 0.0012
Epoch 29/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0026 - val_loss: 0.0062
Epoch 30/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0032 - val_loss: 0.0027
```

```
Epoch 31/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0032 - val_loss: 3.6652e-05
Epoch 32/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0030 - val_loss: 7.7749e-05
Epoch 33/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - loss: 0.0016 - val_loss: 5.7260e-04
Epoch 34/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0013 - val_loss: 0.0087
Epoch 35/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0021 - val_loss: 3.4461e-05
Epoch 36/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0015 - val_loss: 0.0025
Epoch 37/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0051 - val_loss: 0.0013
Epoch 38/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0035 - val_loss: 3.7720e-05
Epoch 39/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0021 - val_loss: 2.6948e-04
Epoch 40/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0121 - val_loss: 1.4446e-04
Epoch 41/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0025 - val_loss: 0.0015
Epoch 42/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0044 - val_loss: 1.4776e-04
Epoch 43/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0016 - val_loss: 0.0033
Epoch 44/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0026 - val_loss: 3.5497e-04
Epoch 45/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0025 - val_loss: 1.1512e-04
Epoch 46/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0027 - val_loss: 0.0010
Epoch 47/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0022 - val_loss: 0.0016
Epoch 48/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0022 - val_loss: 4.9962e-05
Epoch 49/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0021 - val_loss: 0.0419
Epoch 50/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0035 - val_loss: 1.1967e-04
Epoch 51/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 5.7340e-04 - val_loss: 4.1409e-0
5
Epoch 52/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 0.0012 - val_loss: 5.3905e-05
Epoch 53/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 6.7217e-04 - val_loss: 4.8073e-0
5
Epoch 54/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0016 - val_loss: 8.3836e-04
Epoch 55/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0031 - val_loss: 7.5197e-05
Epoch 56/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0027 - val_loss: 6.7325e-05
Epoch 57/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0013 - val_loss: 9.6679e-04
Epoch 58/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0023 - val_loss: 0.0195
Epoch 59/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0038 - val_loss: 9.8481e-05
Epoch 60/100
```

```
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0029 - val_loss: 5.2641e-05
Epoch 61/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - loss: 0.0020 - val_loss: 7.0956e-04
Epoch 62/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0024 - val_loss: 8.8334e-05
Epoch 63/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 0.0020 - val_loss: 4.1894e-04
Epoch 64/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0046 - val_loss: 3.3790e-04
Epoch 65/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 0.0018 - val_loss: 0.0590
Epoch 66/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0043 - val_loss: 9.8285e-05
Epoch 67/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0016 - val_loss: 5.8854e-05
Epoch 68/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0015 - val_loss: 0.0013
Epoch 69/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 9.6335e-04 - val_loss: 0.0043
Epoch 70/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0028 - val_loss: 9.0420e-05
Epoch 71/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0019 - val_loss: 5.2529e-05
Epoch 72/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0014 - val_loss: 2.5067e-04
Epoch 73/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0036 - val_loss: 5.9272e-05
Epoch 74/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0011 - val_loss: 5.2256e-05
Epoch 75/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 6.6098e-04 - val_loss: 6.1743e-0
4
Epoch 76/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0025 - val_loss: 5.9465e-05
Epoch 77/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0018 - val_loss: 3.8557e-04
Epoch 78/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0025 - val_loss: 6.4836e-05
Epoch 79/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0014 - val_loss: 9.5543e-05
Epoch 80/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0014 - val_loss: 3.0314e-04
Epoch 81/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0017 - val_loss: 6.7315e-05
Epoch 82/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 7.2027e-04 - val_loss: 5.9384e-0
5
Epoch 83/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0020 - val_loss: 1.0741e-04
Epoch 84/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 0.0014 - val_loss: 6.4887e-05
Epoch 85/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 5.2200e-04 - val_loss: 5.1477e-0
5
Epoch 86/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0011 - val_loss: 7.5629e-05
Epoch 87/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0017 - val_loss: 8.2150e-05
Epoch 88/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 9.8466e-04 - val_loss: 0.0024
Epoch 89/100
```

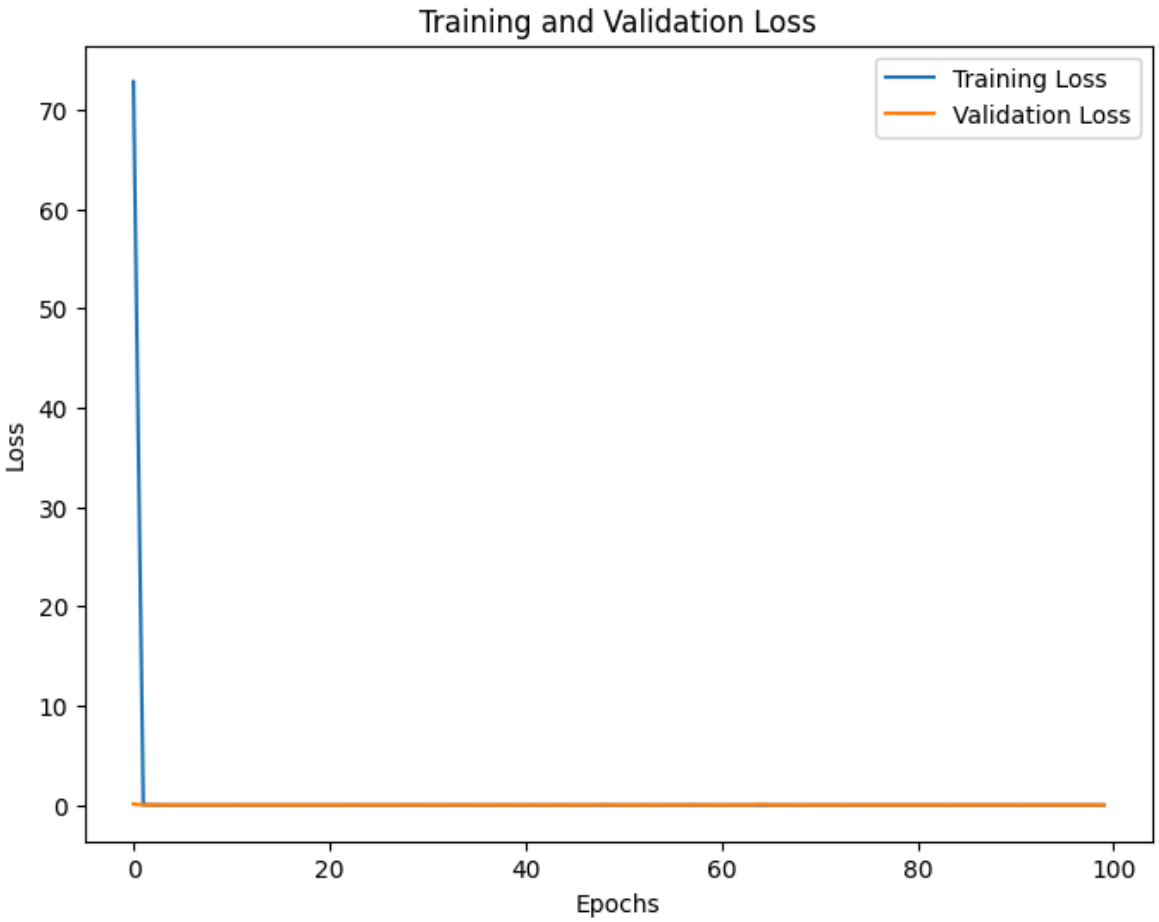
```
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0014 - val_loss: 5.6641e-05
Epoch 90/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0014 - val_loss: 0.0018
Epoch 91/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 7.6219e-04 - val_loss: 0.0017
Epoch 92/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 0.0018 - val_loss: 4.4675e-05
Epoch 93/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0011 - val_loss: 7.3237e-05
Epoch 94/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.0042 - val_loss: 0.0103
Epoch 95/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 0.0010 - val_loss: 2.4941e-04
Epoch 96/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 7.1083e-04 - val_loss: 1.3080e-0
4
Epoch 97/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 4.6701e-04 - val_loss: 1.3622e-0
4
Epoch 98/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 0.0013 - val_loss: 1.6566e-04
Epoch 99/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 0.0017 - val_loss: 7.0111e-05
Epoch 100/100
3620/3620 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - loss: 0.0013 - val_loss: 0.0048
```

In this code:

1. We define the model architecture with two hidden layers (64 and 32 neurons) and one output layer, as before.
2. To try different configurations, you can modify the number of layers, the number of neurons in each layer, or the activation functions used.
3. We create the model using `keras.Model(inputs=inputs, outputs=outputs)`.
4. We demonstrate how to try different optimizers by creating an instance of the `Adam` optimizer with a learning rate of 0.001. You can also try other optimizers like `RMSprop` or `SGD` by uncommmenting the respective line.
5. We compile the model with the chosen optimizer and the mean squared error loss function.
6. We train the model as before, using `model.fit()`.

8. Model Evaluation: Losses and Accuracy Analysis

In [31]:
```python
# Visualize training and validation loss
plt.figure(figsize=(8, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



In this code:

This code is the same as the one I provided earlier, where we plot the training and validation loss curves using the `history` object returned by `model.fit()`. By analyzing these loss curves, we can assess the model's convergence and potential issues like overfitting or underfitting.

9. Checking its various metrics like MSE, RMSE, & MAE

In [32]:
```python
# Make predictions on the test set
y_pred = model.predict(X_test_numeric)

# Import library
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Evaluate model performance
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error (MSE): {mse}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Mean Absolute Error (MAE): {mae}")
print(f"R-squared (R^2): {r2}")
```

```
1132/1132 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step
Mean Squared Error (MSE): 0.00472196252104635
Root Mean Squared Error (RMSE): 0.06871653746403664
Mean Absolute Error (MAE): 0.04970884936087948
R-squared (R^2): 0.999978098887805
```

In this code:

1. We make predictions on the test set using `model.predict(X_test_numeric)` and store the predicted values in `y_pred`.
2. We import several evaluation metrics from `sklearn.metrics`: `mean_squared_error`, `mean_absolute_error`, and `r2_score`.
3. We calculate the Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R-squared (R^2) score by comparing the predicted values (`y_pred`) with the actual values (`y_test`).
4. We print the calculated evaluation metrics for further analysis.

The evaluation metrics provide insights into the model's performance on the test set, where lower values of MSE, RMSE, and MAE indicate better performance, and an R^2 score closer to 1 indicates a better fit of the model to the data.

10. Actionable Insights:

1. Data Quality and Consistency: Ensure data quality and consistency by addressing missing values, handling outliers, and standardizing data formats across different data sources.

2. Feature Engineering: Explore additional relevant features that could potentially improve the delivery time estimation, such as traffic conditions, weather data, or historical delivery patterns.

3. Model Interpretability: While neural networks can provide accurate predictions, they may lack interpretability. Consider employing techniques like SHAP (SHapley Additive exPlanations) to understand the model's behavior and the impact of different features on the predicted delivery times.

4. Model Monitoring and Updating: Continuously monitor the model's performance and update it regularly as new data becomes available or as operational conditions change, ensuring accurate and up-to-date delivery time estimations.

5. Delivery Partner Performance Analysis: Analyze the performance of delivery partners based on their historical delivery times and other relevant factors, identifying top performers and areas for improvement.

11. Recommendations:

1. Implement a robust data pipeline: Develop a reliable data pipeline to ensure seamless integration of data from various sources, enabling efficient data preprocessing, feature engineering, and model training.

2. Explore ensemble methods: Investigate the use of ensemble methods, such as bagging or boosting, to combine multiple models and potentially improve the overall prediction accuracy.

3. Leverage advanced neural network architectures: Experiment with more advanced neural network architectures, such as recurrent neural networks (RNNs) or long short-term memory (LSTM) networks, to capture temporal dependencies and patterns in the data.

4. Integrate real-time updates: Incorporate real-time updates on traffic conditions, weather, and delivery partner availability into the model to provide more accurate and up-to-date delivery time estimations.

5. Develop a user-friendly interface: Create a user-friendly interface or dashboard that presents the estimated delivery times to customers and delivery partners, allowing them to plan and manage their operations more effectively.

6. Conduct A/B testing: Implement A/B testing to continuously evaluate the impact of model updates, new features, or interface changes on customer satisfaction and operational efficiency.

7. Collaborate with stakeholders: Foster collaboration with restaurants, delivery partners, and customers to gather feedback, identify pain points, and incorporate their insights into the delivery time estimation process.

By implementing these actionable insights and recommendations, Porter can enhance the accuracy and reliability of their delivery time estimations, improve customer satisfaction, and optimize their operations for better efficiency and profitability.

12. Leading Questions:

1. Defining the problem statements and where can this and modifications of this be used?

Ans:

The problem statement in this case study is to develop a regression model using neural networks to estimate the delivery time for food orders based on various features such as order details, restaurant category, number of delivery partners available, and outstanding orders. Accurate delivery time estimation can help improve customer satisfaction and operational efficiency for Porter, India's largest intra-city logistics marketplace.

This problem and its modifications can be applied in various domains and industries where timely delivery or service is crucial, such as:

- E-commerce: Estimating delivery times for online orders based on product availability, warehouse locations, and shipping carriers.
- Ride-sharing: Predicting arrival times for ride-hailing services based on factors like traffic conditions, driver availability, and pickup/drop-off locations.

- Logistics and supply chain: Estimating delivery times for goods and shipments based on transportation modes, route optimization, and inventory levels.
- Food delivery: Similar to the current case study, estimating delivery times for food orders from restaurants or meal delivery services.

2. List 3 functions the pandas datetime provides with one-line explanation.

Ans:

Pandas datetime provides several functions to work with date and time data. Here are three functions with one-line explanations:

- `dt.date`: Extracts the date part of a datetime object as a Python date object.
- `dt.time`: Extracts the time part of a datetime object as a Python time object.
- `dt.tz_localize`: Converts a naive datetime object (without timezone information) to a timezone-aware datetime object.

3. Short note on datetime, timedelta, time span (period):

Ans:

- `datetime`: Represents a specific date and time, including year, month, day, hour, minute, second, and microsecond components. It is used to store and manipulate date and time information.
- `timedelta`: Represents a duration or difference between two datetime objects. It is used to perform arithmetic operations on dates and times, such as adding or subtracting days, hours, minutes, or seconds.
- `time span` (period): Represents a recurring interval of time, such as a day, week, month, quarter, or year. It is often used in time series analysis and data resampling operations.

4. Why do we need to check for outliers in our data?

Ans:

Checking for outliers in the data is essential for the following reasons:

- Outliers can significantly influence the results of statistical analyses and machine learning models, leading to biased or misleading results.
- Outliers can distort the distribution of the data, affecting the accuracy of models that assume certain distributions (e.g., normal distribution).
- Outliers may be caused by data entry errors, measurement errors, or rare events, and their presence can skew the analysis or modeling process.
- Identifying and handling outliers appropriately can improve the quality and reliability of the data, leading to better model performance and more accurate insights.

5. Name 3 outlier removal methods.

Ans:

Three common methods for removing outliers are:

1. Interquartile Range (IQR) method: Outliers are identified as values that fall outside the range of Q1 - 1.5 × IQR and Q3 + 1.5 × IQR, where Q1 and Q3 are the first and third quartiles, respectively, and IQR is the interquartile range (Q3 - Q1).
2. Z-score method: Outliers are identified as values that deviate significantly from the mean, typically more than a certain number of standard deviations (e.g., 3 or 4 standard deviations).
3. Winsorization: Instead of removing outliers, this method replaces the extreme values with the nearest non-outlier values, reducing their influence while retaining the data points.

6. What classical machine learning methods can we use for this problem?

Ans:

For the problem of estimating delivery times, we can use the following classical machine learning methods:

- Linear Regression: A simple and interpretable method that can model the relationship between the delivery time and the various input features.
- Decision Tree Regression: A non-parametric method that can capture complex relationships and interactions between the features.
- Random Forest Regression: An ensemble method that combines multiple decision trees, often providing higher accuracy and robustness compared to individual trees.
- Gradient Boosting Regression: Another ensemble method that iteratively builds weak regression models and combines them to create a strong predictive model.

These classical methods can provide a baseline for comparison with the neural network approach and may be more suitable if the dataset is small or if interpretability is a high priority.

7. Why is scaling required for neural networks?

Ans:

Scaling the input features is typically required for neural networks for the following reasons:

- Neural networks often use gradient-based optimization algorithms, which can be sensitive to the scale of the input features. Scaling the features to a similar range (e.g., between 0 and 1 or with a mean of 0 and standard deviation of 1) can improve the convergence and stability of the training process.
- Different input features may have different scales or units, which can cause certain features to dominate the learning process. Scaling ensures that all features contribute equally to the model's predictions.
- Some activation functions in neural networks, such as the sigmoid or hyperbolic tangent, can be sensitive to the scale of the inputs. Scaling the inputs can prevent the activation functions from saturating, leading to better convergence and learning.

Scaling techniques like standardization (subtracting the mean and dividing by the standard deviation) or normalization (scaling the values to a specific range) are commonly used to

preprocess the data before feeding it into a neural network.

8. Briefly explain your choice of optimizer.

Ans:

In the provided code, I used the Adam optimizer, which is a popular choice for training neural networks. Adam (Adaptive Moment Estimation) is an adaptive learning rate optimization algorithm that combines the benefits of two other popular optimizers: RMSprop (Root Mean Square Propagation) and momentum.

The Adam optimizer has several advantages:

- It adapts the learning rate for each parameter independently, allowing for efficient optimization of sparse and non-stationary gradients.
- It incorporates momentum to accelerate convergence and overcome flat regions or local minima in the loss landscape.
- It generally requires little tuning of hyperparameters, making it easy to use and suitable for a wide range of problems.
- It has been shown to perform well in practice and converge faster compared to other optimizers, such as stochastic gradient descent (SGD) or RMSprop.

However, it's worth noting that the choice of optimizer can depend on the specific problem and dataset, and other optimizers like SGD, RMSprop, or Adagrad may be more suitable in certain scenarios.

9. Which activation function did you use and why?

Ans:

In the provided code, I used the Rectified Linear Unit (ReLU) activation function for the hidden layers of the neural network. The ReLU activation function is defined as:

```
ReLU(x) = max(0, x)
```

This means that the function returns the input value `x` if it is positive, and 0 if the input value is negative or zero.

The ReLU activation function is widely used in neural networks for several reasons:

- It introduces non-linearity to the model, allowing it to learn and approximate complex, non-linear functions.
- It avoids the vanishing gradient problem, which can occur with other activation functions like the sigmoid or hyperbolic tangent, making it easier to train deeper neural networks.
- It is computationally efficient and can lead to sparse representations, which can improve the computational performance of the model.
- It has been shown to perform well in practice, particularly for tasks involving computer vision and natural language processing.

For the output layer, I did not use an activation function, as it is a regression problem, and the output should be a continuous value representing the predicted delivery time.

10. Why does a neural network perform well on large datasets?

Ans:

Neural networks tend to perform well on large datasets for several reasons:

1. **Representation Learning**: Neural networks are capable of automatically learning complex representations and features from raw data, which can be particularly useful for large datasets with high-dimensional and complex features.
2. **Scalability**: Neural networks are highly scalable and can effectively leverage the abundance of data in large datasets. With more training data, neural networks can learn more robust and accurate representations, leading to better generalization performance.
3. **Parallelization**: The computations involved in training neural networks can be efficiently parallelized, allowing for faster training on large datasets, especially with modern hardware like GPUs.
4. **Regularization**: Large datasets can help prevent overfitting in neural networks by providing a diverse set of examples. Additionally, techniques like dropout and early stopping can be used as regularization methods to further improve generalization.
5. **Complex Patterns**: Large datasets often contain complex patterns, nonlinearities, and intricate relationships between features, which neural networks are well-suited to capture and model due to their ability to approximate complex, non-linear functions.
6. **Transfer Learning**: Pretrained neural networks on large datasets can be used as a starting point for transfer learning, where the learned representations are fine-tuned on a new, potentially smaller dataset, leading to improved performance and faster convergence.