In [1]:
```
import tensorflow as tf
print("TF version:", tf.__version__)
print("GPU available:", tf.config.list_physical_devices('GPU'))
```

```
TF version: 2.19.0
GPU available: []
```

In [2]:
```
import pandas as pd
```

In [3]:
```
from tensorflow.keras import layers, models
```

In [4]:
```
# simple model
model = models.Sequential([
    layers.Dense(10, activation='relu', input_shape=(5,)),
    layers.Dense(1)
])
```

```
C:\Users\user\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87: User
Warning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using
Sequential models, prefer using an `Input(shape)` object as the first layer in th
e model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [5]:
```
model.compile(optimizer='adam', loss='mse')
print("Model built successfully!")
```

```
Model built successfully!
```

Constant - here two square brackets are very important orelse it wont be able to convert arg dtype to tf dtype

In [22]:
```
A= tf.constant([[2,3],[4,5]])
A
```

Out[22]:
```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[2, 3],
       [4, 5]])>
```

Variable- V should be capital here

In [26]:
```
B=tf.Variable([[3,3],[4,6]])
B
```

Out[26]:
```
<tf.Variable 'Variable:0' shape=(2, 2) dtype=int32, numpy=
array([[3, 3],
       [4, 6]])>
```

Concatenation

In [29]:
```
AB= tf.concat(values=[A,B],axis=1)
AB
```

Out[29]:
```
<tf.Tensor: shape=(2, 4), dtype=int32, numpy=
array([[2, 3, 3, 3],
       [4, 5, 4, 6]])>
```

In [33]:
```
AB_row= tf.concat(values=[A,B],axis=0)
AB_row
```

Out[33]:
```
<tf.Tensor: shape=(4, 2), dtype=int32, numpy=
array([[2, 3],
       [4, 5],
       [3, 3],
       [4, 6]])>
```

Masking - can fill matrices with ones and zeros

In [35]:
```
tensor=tf.ones(shape=[3,4],dtype=tf.float32)
tensor
```

Out[35]:
```
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]], dtype=float32)>
```

In [46]:
```
te=tf.zeros(shape=[3,4],dtype=tf.int32)
te
```

Out[46]:
```
<tf.Tensor: shape=(3, 4), dtype=int32, numpy=
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])>
```

In [75]:
```
random_v=tf.random.uniform(shape=[4,4],dtype=tf.float32)
print(random_v)
#random_v=tf.random.uniform(shape=[4,4],dtype=tf.int32) here i have to specify m
ran_v=tf.random.uniform(shape=[4,4],minval=20,maxval=100,dtype=tf.int32)
ran_v
```

```
tf.Tensor(
[[0.97025144 0.3443786  0.06935751 0.20907426]
 [0.35617268 0.05145383 0.4771732  0.3762691 ]
 [0.94333076 0.3781333  0.24258304 0.58892846]
 [0.91413796 0.56065786 0.5900793  0.41488802]], shape=(4, 4), dtype=float32)
```

Out[75]:
```
<tf.Tensor: shape=(4, 4), dtype=int32, numpy=
array([[76, 93, 35, 21],
       [74, 84, 58, 41],
       [44, 74, 36, 58],
       [41, 28, 43, 91]])>
```

Reshaping- cnt reshape more than the actual metrix coz it doesnt add extra values

In [51]:
```
reshaped_te=tf.reshape(tensor=te,shape=[4,3])
reshaped_te
```

Out[51]:
```
<tf.Tensor: shape=(4, 3), dtype=int32, numpy=
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])>
```

In [57]:
```
#InvalidArgumentError:Input to reshape is a tensor with 12 values, but the reque
#reshape=tf.reshape(tensor=tensor,shape=[2,2])
#reshape
```

Type Casting

In [80]:
```
ran_value=tf.cast(ran_v,tf.float32)
ran_value
```

Out[80]:
```
<tf.Tensor: shape=(4, 4), dtype=float32, numpy=
array([[76., 93., 35., 21.],
       [74., 84., 58., 41.],
       [44., 74., 36., 58.],
       [41., 28., 43., 91.]], dtype=float32)>
```

Multiplication

In [87]:
```
x=tf.constant([[3,2],[2,5]])
y=tf.constant([[4,7],[6,9]])
xy=tf.matmul(x,y)
xy
```

Out[87]:
```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[24, 39],
       [38, 59]])>
```

In [91]:
```
#element wise multiplication
xy=tf.multiply(x,y)
xy
```

Out[91]:
```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[12, 14],
       [12, 45]])>
```

In [3]:
```
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler,StandardScaler
from sklearn.metrics import mean_squared_error
```

In [ ]:
```
# math=When you need common mathematical constants (π, e) or complex calculation
# sequential= sequential neural network
# dense= fully connected neural network
# lstm=long short term memory a type of Recurrent Neural Network, Specifically d
# When working with time-series data (like stock prices, weather), sequences of
# MinMaxScaler, StandardScaler=Scales/Normalizes data to a small range (usually
# mean_squared_error=Calculates the Mean Squared Error (MSE), which measures the
```

LSTM Google Stock Price Prediction

In [8]:
```
# ggl_stock_exg.csv
import matplotlib.pyplot as plt
import pandas as pd
df = pd.read_csv("ggl_stock_exg.csv")
```

In [10]:
```
print(df.shape)
print(df.head())
```

```
(4431, 7)
         Date       Open       High        Low      Close  Adj Close    Volume
0  2004-08-19  50.050049  52.082081  48.028027  50.220219  50.220219  44659096
1  2004-08-20  50.555557  54.594597  50.300301  54.209209  54.209209  22834343
2  2004-08-23  55.430431  56.796799  54.579578  54.754753  54.754753  18256126
3  2004-08-24  55.675674  55.855858  51.836838  52.487488  52.487488  15247337
4  2004-08-25  52.532532  54.054054  51.991993  53.053055  53.053055   9188602
```

In [ ]:
```
# Our dataset ggl_stock_exg.csv has 4431 rows and 7 columns:Date,Open,High,Low,C
# For an LSTM stock prediction project, we’ll typically use Close or Adj Close a
```

In [12]:
```
data = df.filter(['Adj Close'])
dataset = data.values
```

In [14]:
```
training_data_len = math.ceil(len(dataset) * 0.8)
```

In [16]:
```
scaler = MinMaxScaler(feature_range=(0,1))
scaled_data = scaler.fit_transform(dataset)
```

In [18]:
```
train_data = scaled_data[0:training_data_len, :]
```

In [20]:
```
x_train = []
y_train = []

# Use past 60 days → predict next day
for i in range(60, len(train_data)):
    x_train.append(train_data[i-60:i, 0])
    y_train.append(train_data[i, 0])
```

In [28]:
```
import numpy as np
x_train, y_train = np.array(x_train), np.array(y_train)
# Reshape for LSTM (samples, time steps, features)
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
```

In [30]:
```
model = Sequential()
model.add(LSTM(50, return_sequences=True, input_shape=(x_train.shape[1], 1)))
model.add(LSTM(50, return_sequences=False))
model.add(Dense(25))
model.add(Dense(1))

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

```
C:\Users\user\anaconda3\Lib\site-packages\keras\src\layers\rnn\rnn.py:200: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

In [32]:
```python
model.fit(x_train, y_train, batch_size=32, epochs=10)
```

```
Epoch 1/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 6s 21ms/step - loss: 0.0022
Epoch 2/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - loss: 3.8412e-05
Epoch 3/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - loss: 4.1170e-05
Epoch 4/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - loss: 3.8176e-05
Epoch 5/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 2s 22ms/step - loss: 4.2667e-05
Epoch 6/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 2s 22ms/step - loss: 4.6641e-05
Epoch 7/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - loss: 3.3588e-05
Epoch 8/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - loss: 4.2039e-05
Epoch 9/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - loss: 3.3135e-05
Epoch 10/10
109/109 ━━━━━━━━━━━━━━━━━━━━ 2s 22ms/step - loss: 3.8978e-05
```

Out[32]: <keras.src.callbacks.history.History at 0x1ee93fcb810>

In [34]:
```python
# Test data (remaining 20%)
test_data = scaled_data[training_data_len - 60:, :]

x_test = []
y_test = dataset[training_data_len:, :]

for i in range(60, len(test_data)):
    x_test.append(test_data[i-60:i, 0])

x_test = np.array(x_test)
x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))

# Predictions
predictions = model.predict(x_test)
predictions = scaler.inverse_transform(predictions)
```

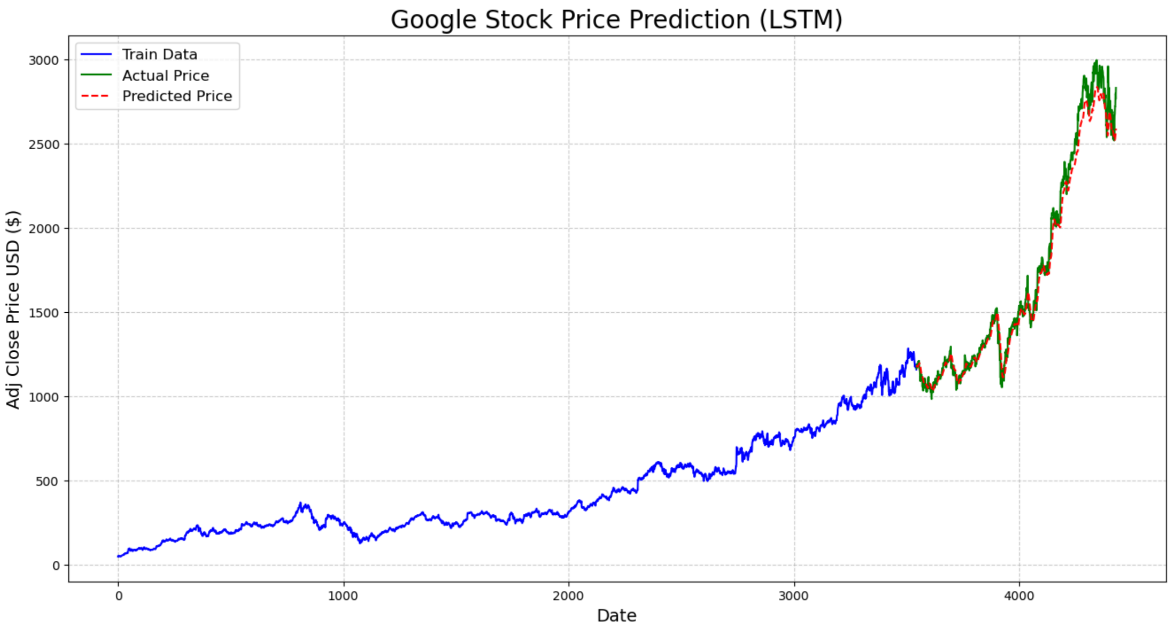
```
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step
```

In [36]:
```python
rmse = np.sqrt(mean_squared_error(y_test, predictions))
print("RMSE:", rmse)
```

```
RMSE: 78.22104922145199
```

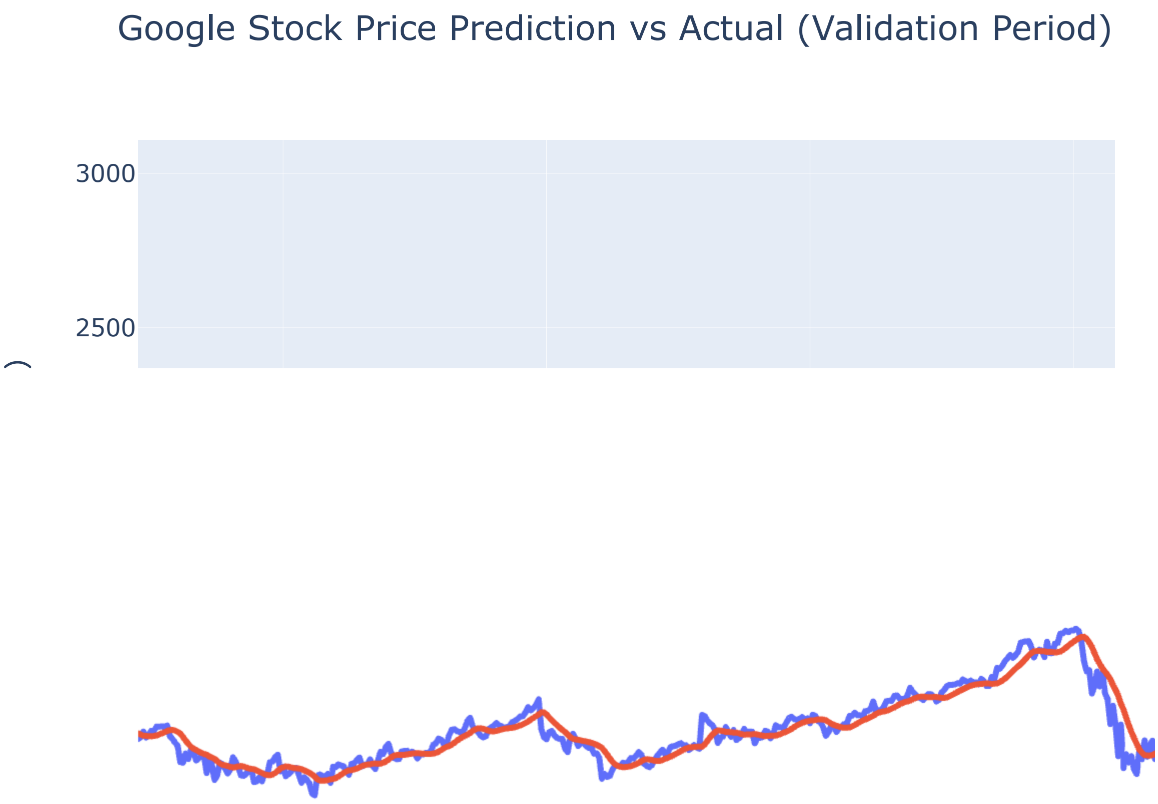
In [40]:
```python
plt.figure(figsize=(16,8))
plt.title("Google Stock Price Prediction (LSTM)", fontsize=20)
plt.xlabel("Date", fontsize=14)
plt.ylabel("Adj Close Price USD ($)", fontsize=14)

# Plot train data
plt.plot(train['Adj Close'], label="Train Data", color="blue")
```

```
# Plot actual validation data
plt.plot(valid['Adj Close'], label="Actual Price", color="green")

# Plot predictions
plt.plot(valid['Predictions'], label="Predicted Price", color="red", linestyle="

# Grid + legend
plt.grid(True, linestyle="--", alpha=0.6)
plt.legend(fontsize=12)
plt.show()
```



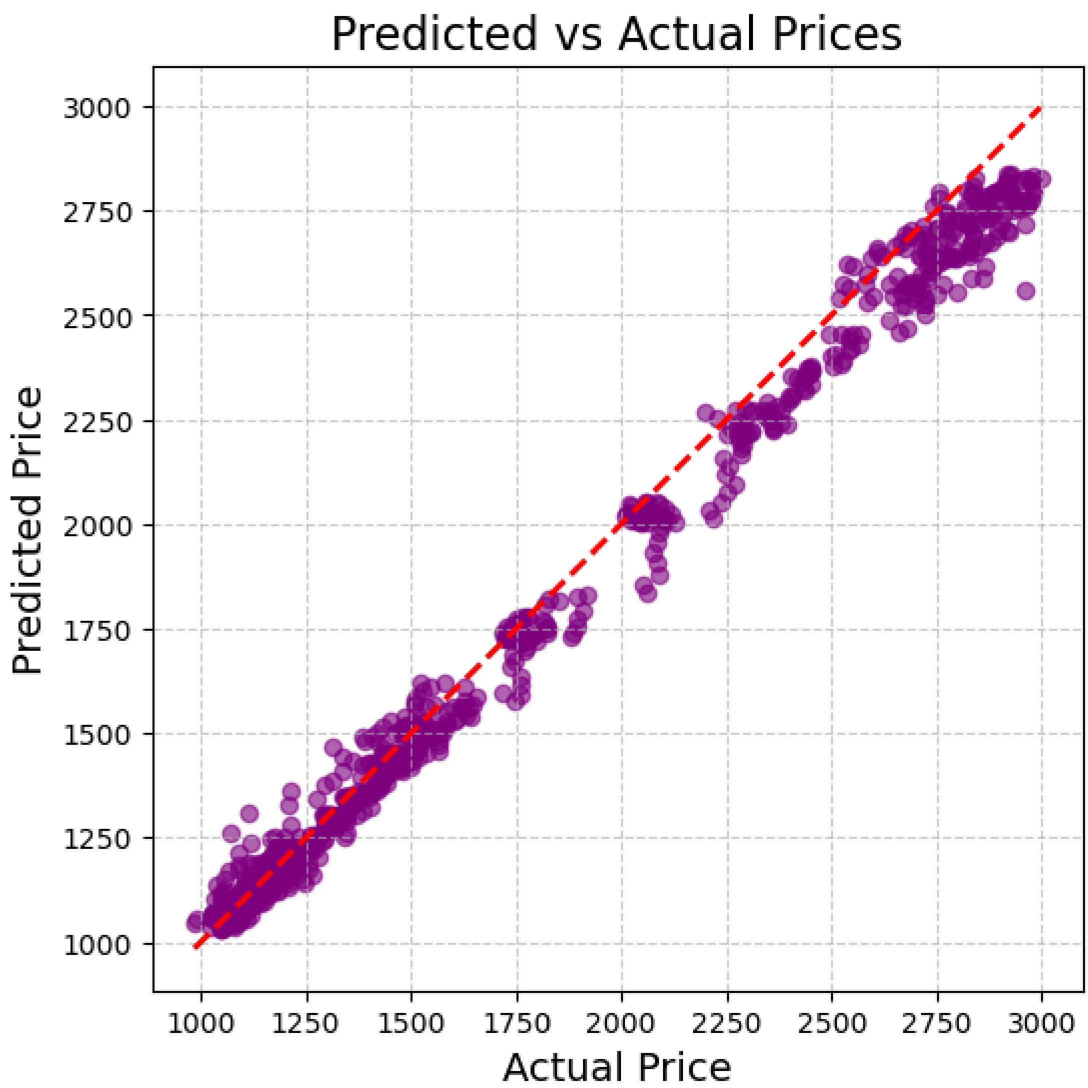
In [42]:

```
import plotly.express as px

fig = px.line(valid, y=["Adj Close", "Predictions"],
              labels={"value": "Stock Price (USD)", "index": "Time"},
              title="Google Stock Price Prediction vs Actual (Validation Period)

fig.update_traces(line=dict(width=3))
fig.show()
```



In [46]:

```
plt.figure(figsize=(6,6))
plt.scatter(valid['Adj Close'], valid['Predictions'], color="purple", alpha=0.6)
plt.plot([valid['Adj Close'].min(), valid['Adj Close'].max()],
         [valid['Adj Close'].min(), valid['Adj Close'].max()],
         color="red", linestyle="--", linewidth=2)

plt.title("Predicted vs Actual Prices", fontsize=16)
plt.xlabel("Actual Price", fontsize=14)
plt.ylabel("Predicted Price", fontsize=14)
plt.grid(True, linestyle="--", alpha=0.6)
plt.show()
```



In [52]:
```
def predict_next_day(model, scaler, recent_data):
    import numpy as np
    # Convert to numpy array
    recent_data = np.array(recent_data).reshape(-1, 1)
    # Scale the data
    scaled_data = scaler.transform(recent_data)
    # Reshape for LSTM [samples, time_steps, features]
    X_input = np.reshape(scaled_data, (1, scaled_data.shape[0], 1))
    # Predict
    pred_scaled = model.predict(X_input)
    # Inverse transform to get actual price
    predicted_price = scaler.inverse_transform(pred_scaled)
    return float(predicted_price[0][0])
```

In [54]:
```
# Take last 60 days from your dataset
recent_60_days = data['Adj Close'][-60:].values
# Predict next day
next_price = predict_next_day(model, scaler, recent_60_days)
print(f"Predicted next day price: ${next_price:.2f}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step
Predicted next day price: $2611.61
```

In [ ]: