experiment_9

October 11, 2025

Name:	Tufan Kundu
Registration no:	24MDT0184
Course Name:	Deep Learning Lab
Course Code:	PMDS603P
Experiment:	9
Date:	25 September,2025

0.1 Question 1: First, we will try to load the dataset and do the pre-processing part. From Github i have taken this data set which has information of monthly gold price till a latest date

0.1.1 Importing necessary libraries

[1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import mean_squared_error, r2_score
from tensorflow import keras
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense, TimeDistributed
from keras.callbacks import EarlyStopping

import warnings
warnings.filterwarnings('ignore')
```

[2]:
```python
np.random.seed(42)

### Loading the dataset
df = pd.read_csv("gold_price.csv")
df
```

[2]:
```
          Date     Price
0      1833-01     18.93
1      1833-02     18.93
2      1833-03     18.93
3      1833-04     18.93
```

```
4     1833-05    18.93
…       …        …
2306  2025-03  2983.25
2307  2025-04  3217.64
2308  2025-05  3309.49
2309  2025-06  3352.66
2310  2025-07  3340.15

[2311 rows x 2 columns]
```

[3]:
```
data = df['Price'].values.reshape(-1,1)
data
```

[3]:
```
array([[  18.93],
       [  18.93],
       [  18.93],
       …,
       [3309.49],
       [3352.66],
       [3340.15]])
```

[4]:
```
scaler = MinMaxScaler()
data_scaled = scaler.fit_transform(data)
```

0.1.2 Preparing the data

[6]:
```
train_size = int(len(data_scaled)*0.8)
train_data = data_scaled[:train_size]
test_data = data_scaled[train_size:]
```

[7]:
```
def create_sequences(data,seq_length):
    sequences = []
    for i in range(len(data) - seq_length+1):
        sequences.append(data[i:i+seq_length])
    return np.array(sequences)

seq_length = 10
train_sequences = create_sequences(train_data,seq_length)
test_sequences = create_sequences(test_data, seq_length)
```

0.1.3 Splitting the sequences into input and targets

[12]:
```
x_train_full, y_train_full = train_sequences[:,:-1], train_sequences[:, -1]
x_test, y_test = test_sequences[:,:-1], test_sequences[:, -1]
```

```
val_fraction = 0.2
val_size = int(len(x_train_full)*val_fraction)

x_val = x_train_full[-val_size:]
y_val = y_train_full[-val_size:]

x_train = x_train_full[:-val_size]
y_train = y_train_full[:-val_size]
```

0.1.4 Defining and compiling the RNN Model

[14]:
```
model = Sequential()
model.add(SimpleRNN(64, activation = 'tanh', input_shape = (seq_length-1,1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss = 'mean_squared_error')
model.summary()

early_stop = EarlyStopping(monitor = 'val_loss', patience = 5,␣
 ↪restore_best_weights=True, verbose=1)

history = model.fit(x_train,y_train,
                    epochs = 500,
                    batch_size = 16,
                    validation_data=(x_val,y_val),
                    verbose = 1,
                    callbacks= [early_stop])
```

```
c:\Users\TUFAN\.conda\envs\tf_env\lib\site-
packages\keras\src\layers\rnn\rnn.py:199: 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)
```

```
Model: "sequential"


 Layer (type)                      Output Shape                 Param #

 simple_rnn (SimpleRNN)            (None, 64)                     4,224

 dense (Dense)                     (None, 1)                         65


 Total params: 4,289 (16.75 KB)


 Trainable params: 4,289 (16.75 KB)
 Non-trainable params: 0 (0.00 B)


Epoch 1/500
92/92            2s 5ms/step - loss:
9.3772e-05 - val_loss: 3.9152e-04
Epoch 2/500
92/92            0s 3ms/step - loss:
9.6166e-08 - val_loss: 3.6820e-04
Epoch 3/500
92/92            0s 3ms/step - loss:
8.7216e-08 - val_loss: 3.5442e-04
Epoch 4/500
92/92            0s 3ms/step - loss:
1.0011e-07 - val_loss: 3.3771e-04
Epoch 5/500
92/92            0s 3ms/step - loss:
1.0102e-07 - val_loss: 3.1484e-04
Epoch 6/500
92/92            0s 3ms/step - loss:
7.3213e-08 - val_loss: 2.9836e-04
Epoch 7/500
92/92            0s 3ms/step - loss:
8.1495e-08 - val_loss: 2.7874e-04
Epoch 8/500
92/92            0s 3ms/step - loss:
1.0150e-07 - val_loss: 2.7002e-04
Epoch 9/500
92/92            0s 3ms/step - loss:
5.9958e-08 - val_loss: 2.4461e-04
Epoch 10/500
92/92            0s 3ms/step - loss:
6.4128e-08 - val_loss: 2.3065e-04
Epoch 11/500
92/92            0s 3ms/step - loss:
8.2368e-08 - val_loss: 2.2065e-04
Epoch 12/500
92/92            0s 2ms/step - loss:
5.7914e-08 - val_loss: 2.0032e-04
Epoch 13/500
92/92            0s 2ms/step - loss:
7.0208e-08 - val_loss: 1.8862e-04
Epoch 14/500
92/92            0s 2ms/step - loss:
6.3733e-08 - val_loss: 1.8109e-04
Epoch 15/500
92/92            0s 2ms/step - loss:
6.2409e-08 - val_loss: 1.6689e-04
```

```
Epoch 16/500
92/92            0s 2ms/step - loss:
5.4545e-08 - val_loss: 1.5538e-04
Epoch 17/500
92/92            0s 2ms/step - loss:
5.1540e-08 - val_loss: 1.4894e-04
Epoch 18/500
92/92            0s 2ms/step - loss:
3.9574e-08 - val_loss: 1.3078e-04
Epoch 19/500
92/92            0s 2ms/step - loss:
4.1837e-08 - val_loss: 1.2377e-04
Epoch 20/500
92/92            0s 2ms/step - loss:
3.7922e-08 - val_loss: 1.1526e-04
Epoch 21/500
92/92            0s 2ms/step - loss:
3.9879e-08 - val_loss: 1.1101e-04
Epoch 22/500
92/92            0s 2ms/step - loss:
9.6580e-08 - val_loss: 1.0964e-04
Epoch 23/500
92/92            0s 3ms/step - loss:
3.6806e-08 - val_loss: 1.0487e-04
Epoch 24/500
92/92            0s 2ms/step - loss:
7.0121e-08 - val_loss: 9.7452e-05
Epoch 25/500
92/92            0s 2ms/step - loss:
3.3860e-08 - val_loss: 9.3796e-05
Epoch 26/500
92/92            0s 2ms/step - loss:
2.6423e-08 - val_loss: 8.9613e-05
Epoch 27/500
92/92            0s 2ms/step - loss:
3.2746e-08 - val_loss: 8.3791e-05
Epoch 28/500
92/92            0s 3ms/step - loss:
2.2390e-08 - val_loss: 8.0220e-05
Epoch 29/500
92/92            0s 2ms/step - loss:
2.9187e-08 - val_loss: 7.8521e-05
Epoch 30/500
92/92            0s 2ms/step - loss:
2.0004e-08 - val_loss: 7.6565e-05
Epoch 31/500
92/92            0s 2ms/step - loss:
1.9323e-08 - val_loss: 7.3813e-05
```

```
Epoch 32/500
92/92            0s 3ms/step - loss:
3.1451e-08 - val_loss: 7.0650e-05
Epoch 33/500
92/92            0s 2ms/step - loss:
2.7166e-08 - val_loss: 6.9395e-05
Epoch 34/500
92/92            0s 3ms/step - loss:
2.0559e-08 - val_loss: 6.8895e-05
Epoch 35/500
92/92            0s 3ms/step - loss:
2.7681e-08 - val_loss: 6.6662e-05
Epoch 36/500
92/92            0s 4ms/step - loss:
2.2744e-08 - val_loss: 6.5504e-05
Epoch 37/500
92/92            0s 3ms/step - loss:
2.5234e-08 - val_loss: 6.6564e-05
Epoch 38/500
92/92            0s 3ms/step - loss:
3.1124e-08 - val_loss: 6.6440e-05
Epoch 39/500
92/92            0s 3ms/step - loss:
2.0425e-08 - val_loss: 6.5000e-05
Epoch 40/500
92/92            0s 3ms/step - loss:
2.0832e-08 - val_loss: 6.0271e-05
Epoch 41/500
92/92            0s 2ms/step - loss:
3.9688e-08 - val_loss: 5.9527e-05
Epoch 42/500
92/92            0s 3ms/step - loss:
3.2011e-08 - val_loss: 5.8785e-05
Epoch 43/500
92/92            0s 2ms/step - loss:
2.1576e-08 - val_loss: 5.7750e-05
Epoch 44/500
92/92            0s 3ms/step - loss:
2.3649e-08 - val_loss: 5.6891e-05
Epoch 45/500
92/92            0s 2ms/step - loss:
2.4557e-08 - val_loss: 5.6333e-05
Epoch 46/500
92/92            0s 2ms/step - loss:
2.9139e-08 - val_loss: 5.4708e-05
Epoch 47/500
92/92            0s 3ms/step - loss:
2.1123e-08 - val_loss: 5.3470e-05
```

```
Epoch 48/500
92/92             0s 3ms/step - loss:
4.6100e-08 - val_loss: 5.3588e-05
Epoch 49/500
92/92             0s 3ms/step - loss:
1.9368e-08 - val_loss: 5.7409e-05
Epoch 50/500
92/92             0s 3ms/step - loss:
3.2647e-08 - val_loss: 5.2319e-05
Epoch 51/500
92/92             0s 4ms/step - loss:
1.7981e-08 - val_loss: 5.3704e-05
Epoch 52/500
92/92             0s 3ms/step - loss:
2.5447e-08 - val_loss: 5.1381e-05
Epoch 53/500
92/92             0s 3ms/step - loss:
3.5196e-08 - val_loss: 5.0996e-05
Epoch 54/500
92/92             0s 3ms/step - loss:
2.8622e-08 - val_loss: 5.1440e-05
Epoch 55/500
92/92             0s 3ms/step - loss:
1.3115e-08 - val_loss: 5.3297e-05
Epoch 56/500
92/92             0s 3ms/step - loss:
2.6399e-08 - val_loss: 4.8762e-05
Epoch 57/500
92/92             0s 4ms/step - loss:
1.4915e-08 - val_loss: 4.9283e-05
Epoch 58/500
92/92             0s 3ms/step - loss:
3.0933e-08 - val_loss: 4.8661e-05
Epoch 59/500
92/92             0s 3ms/step - loss:
1.4864e-08 - val_loss: 4.6791e-05
Epoch 60/500
92/92             0s 3ms/step - loss:
2.6216e-08 - val_loss: 4.6163e-05
Epoch 61/500
92/92             0s 2ms/step - loss:
2.4084e-08 - val_loss: 4.9848e-05
Epoch 62/500
92/92             0s 3ms/step - loss:
3.5370e-08 - val_loss: 4.5251e-05
Epoch 63/500
92/92             0s 2ms/step - loss:
2.4244e-08 - val_loss: 4.7410e-05
```

```
Epoch 64/500
92/92            0s 3ms/step - loss:
1.3788e-08 - val_loss: 4.5628e-05
Epoch 65/500
92/92            0s 3ms/step - loss:
3.7181e-08 - val_loss: 4.4495e-05
Epoch 66/500
92/92            0s 3ms/step - loss:
3.4376e-08 - val_loss: 4.3944e-05
Epoch 67/500
92/92            0s 3ms/step - loss:
3.0196e-08 - val_loss: 4.4322e-05
Epoch 68/500
92/92            0s 3ms/step - loss:
3.2877e-08 - val_loss: 4.4298e-05
Epoch 69/500
92/92            0s 4ms/step - loss:
1.8189e-08 - val_loss: 4.2800e-05
Epoch 70/500
92/92            0s 2ms/step - loss:
1.3442e-08 - val_loss: 4.2432e-05
Epoch 71/500
92/92            0s 3ms/step - loss:
2.3120e-08 - val_loss: 4.2184e-05
Epoch 72/500
92/92            0s 3ms/step - loss:
7.6539e-08 - val_loss: 4.1896e-05
Epoch 73/500
92/92            0s 2ms/step - loss:
2.2185e-08 - val_loss: 4.1677e-05
Epoch 74/500
92/92            0s 3ms/step - loss:
2.2031e-08 - val_loss: 4.3872e-05
Epoch 75/500
92/92            0s 3ms/step - loss:
2.3024e-08 - val_loss: 4.0901e-05
Epoch 76/500
92/92            0s 3ms/step - loss:
3.9058e-08 - val_loss: 4.2225e-05
Epoch 77/500
92/92            0s 3ms/step - loss:
3.5648e-08 - val_loss: 4.0394e-05
Epoch 78/500
92/92            0s 3ms/step - loss:
2.0412e-08 - val_loss: 4.6721e-05
Epoch 79/500
92/92            0s 3ms/step - loss:
5.7803e-08 - val_loss: 4.3706e-05
```

```
Epoch 80/500
92/92            0s 3ms/step - loss:
2.4857e-08 - val_loss: 3.9752e-05
Epoch 81/500
92/92            0s 3ms/step - loss:
1.0573e-07 - val_loss: 4.3228e-05
Epoch 82/500
92/92            0s 3ms/step - loss:
2.7223e-08 - val_loss: 4.0952e-05
Epoch 83/500
92/92            0s 3ms/step - loss:
1.2786e-08 - val_loss: 3.9536e-05
Epoch 84/500
92/92            0s 3ms/step - loss:
2.3293e-08 - val_loss: 3.8747e-05
Epoch 85/500
92/92            0s 3ms/step - loss:
3.3372e-08 - val_loss: 3.7808e-05
Epoch 86/500
92/92            0s 3ms/step - loss:
3.4545e-08 - val_loss: 3.8056e-05
Epoch 87/500
92/92            0s 3ms/step - loss:
1.3372e-08 - val_loss: 3.7394e-05
Epoch 88/500
92/92            0s 3ms/step - loss:
3.2702e-08 - val_loss: 3.9638e-05
Epoch 89/500
92/92            0s 4ms/step - loss:
2.3807e-08 - val_loss: 3.6876e-05
Epoch 90/500
92/92            0s 3ms/step - loss:
3.2728e-08 - val_loss: 3.7225e-05
Epoch 91/500
92/92            0s 3ms/step - loss:
4.8509e-08 - val_loss: 3.7245e-05
Epoch 92/500
92/92            0s 3ms/step - loss:
2.4564e-08 - val_loss: 3.6274e-05
Epoch 93/500
92/92            0s 3ms/step - loss:
3.7587e-08 - val_loss: 3.6736e-05
Epoch 94/500
92/92            1s 3ms/step - loss:
2.6198e-08 - val_loss: 3.9486e-05
Epoch 95/500
92/92            0s 4ms/step - loss:
2.0085e-08 - val_loss: 3.7497e-05
```

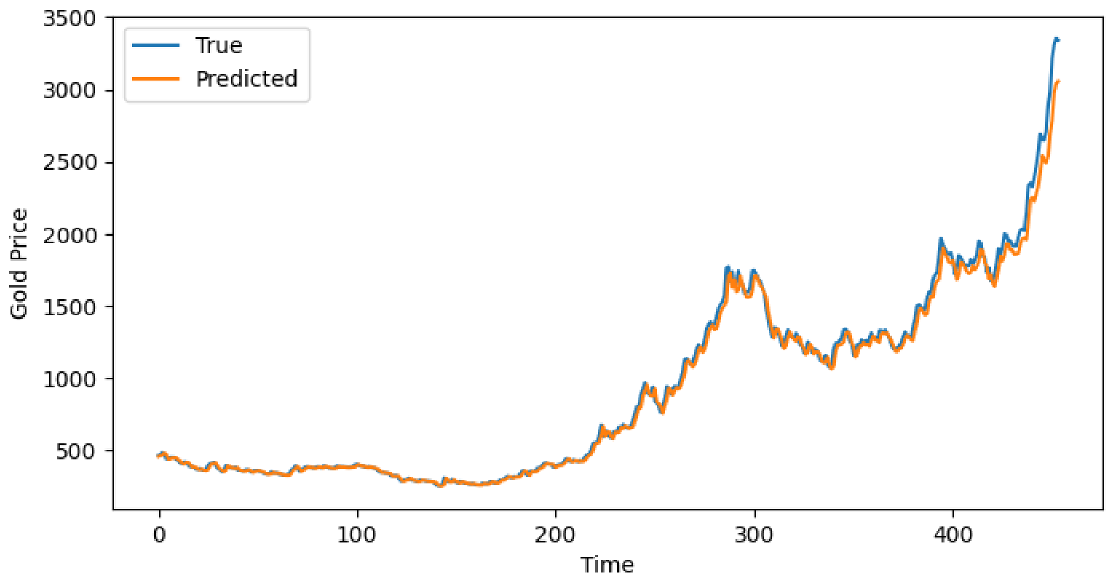
```
Epoch 96/500
92/92            0s 3ms/step - loss:
3.8519e-08 - val_loss: 3.5977e-05
Epoch 97/500
92/92            0s 3ms/step - loss:
2.5974e-08 - val_loss: 3.7319e-05
Epoch 98/500
92/92            0s 3ms/step - loss:
5.5575e-08 - val_loss: 3.8587e-05
Epoch 99/500
92/92            0s 3ms/step - loss:
2.4031e-08 - val_loss: 3.6936e-05
Epoch 100/500
92/92            0s 3ms/step - loss:
6.5016e-08 - val_loss: 3.5248e-05
Epoch 101/500
92/92            0s 3ms/step - loss:
3.5700e-08 - val_loss: 3.5832e-05
Epoch 102/500
92/92            0s 3ms/step - loss:
1.1026e-08 - val_loss: 3.8457e-05
Epoch 103/500
92/92            0s 3ms/step - loss:
1.9037e-08 - val_loss: 3.5675e-05
Epoch 104/500
92/92            0s 3ms/step - loss:
3.2910e-08 - val_loss: 3.7895e-05
Epoch 105/500
92/92            0s 3ms/step - loss:
3.5495e-08 - val_loss: 3.5003e-05
Epoch 106/500
92/92            0s 4ms/step - loss:
3.1266e-08 - val_loss: 3.5079e-05
Epoch 107/500
92/92            0s 4ms/step - loss:
1.1774e-08 - val_loss: 4.1049e-05
Epoch 108/500
92/92            0s 3ms/step - loss:
2.3825e-08 - val_loss: 3.4174e-05
Epoch 109/500
92/92            0s 3ms/step - loss:
3.2877e-08 - val_loss: 3.4035e-05
Epoch 110/500
92/92            0s 4ms/step - loss:
1.7255e-08 - val_loss: 3.3995e-05
Epoch 111/500
92/92            0s 3ms/step - loss:
5.9085e-08 - val_loss: 3.4152e-05
```

```
Epoch 112/500
92/92              0s 3ms/step - loss:
4.6817e-08 - val_loss: 3.8359e-05
Epoch 113/500
92/92              0s 3ms/step - loss:
2.2454e-08 - val_loss: 3.4733e-05
Epoch 114/500
92/92              0s 3ms/step - loss:
6.5081e-08 - val_loss: 3.4348e-05
Epoch 115/500
92/92              0s 3ms/step - loss:
1.9619e-08 - val_loss: 3.4303e-05
Epoch 115: early stopping
Restoring model weights from the end of the best epoch: 110.
```

[15]:
```
y_pred = model.predict(x_test)
y_test_orig = scaler.inverse_transform(y_test)
y_pred_orig = scaler.inverse_transform(y_pred)
rootmse = np.sqrt(mean_squared_error(y_test_orig,y_pred_orig))
r2 = r2_score(y_test_orig,y_pred_orig)
print("Root Mean Squared error:", rootmse)
print("R2 score:", r2)

##plot
plt.figure(figsize = (8,4))
plt.plot(y_test_orig, label = 'True')
plt.plot(y_pred_orig, label = 'Predicted')
plt.xlabel("Time")
plt.ylabel("Gold Price")
plt.legend()
plt.show()
```

```
15/15              0s 9ms/step
Root Mean Squared error: 63.98662536813604
R2 score: 0.9906292642115201
```



0.2 Question 2: Now the previous model we have seen takes an input sequence of length 9 and predicts the next days gold price. if you want to have a model which takes 10 days of data and predict the next ten days of gold price (lets say), then the same model can be modified to get output at every time step by introducing a timedistributed dense layer and also keeping the return sequences parameter = True. But here in order to attempt this problem you need to prepare your input and output sequences accordingly to the model so we can perform these to prepare the sequence initially. First you can import necessary layers, use from keras.layers import SimpleRNN, Dense, TimeDistributed

[16]:
```
def create_sequences(data, input_length = 10, output_length = 10 ):
    x,y = [], []
    for i in range(len(data) - input_length - output_length + 1):
        x.append(data[i:i+input_length])
        y.append(data[i+input_length:i+input_length+output_length])
    return np.array(x), np.array(y)

input_length = 10
output_length = 10
x_train_full,y_train_full = create_sequences(train_data,␣
 ↪input_length,output_length)
x_test,y_test = create_sequences(test_data, input_length,output_length)
```

[17]:
```
val_fraction = 0.2
val_size = int(len(x_train_full)*val_fraction)
```

```
x_val = x_train_full[-val_size:]
y_val = y_train_full[-val_size:]

x_train = x_train_full[:-val_size]
y_train = y_train_full[:-val_size]
```

[18]:
```
model = Sequential()
model.add(SimpleRNN(64,activation = 'tanh', return_sequences=True, input_shape␣
 ↪= (seq_length-1,1)))
model.add(TimeDistributed(Dense(1)))

model.compile(optimizer = 'adam', loss = 'mean_squared_error')
model.summary()

early_stop = EarlyStopping(monitor = 'val_loss', patience = 10,␣
 ↪restore_best_weights=True, verbose=1)
history = model.fit(
x_train, y_train,
epochs = 500,
batch_size = 16,
validation_data = (x_val,y_val),
verbose = 1,
callbacks = [early_stop]
)
```

```
c:\Users\TUFAN\.conda\envs\tf_env\lib\site-
packages\keras\src\layers\rnn\rnn.py:199: 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)
```

```
Model: "sequential_1"


 Layer (type)                      Output Shape              Param #

 simple_rnn_1 (SimpleRNN)          (None, 9, 64)               4,224

 time_distributed                  (None, 9, 1)                   65
 (TimeDistributed)



 Total params: 4,289 (16.75 KB)


 Trainable params: 4,289 (16.75 KB)
 Non-trainable params: 0 (0.00 B)


Epoch 1/500
92/92            2s 6ms/step - loss:
3.2812e-05 - val_loss: 0.0013
Epoch 2/500
92/92            0s 3ms/step - loss:
8.5257e-07 - val_loss: 9.6952e-04
Epoch 3/500
92/92            0s 3ms/step - loss:
5.6306e-07 - val_loss: 7.7712e-04
Epoch 4/500
92/92            0s 3ms/step - loss:
3.5947e-07 - val_loss: 6.6458e-04
Epoch 5/500
92/92            0s 3ms/step - loss:
3.0991e-07 - val_loss: 5.9694e-04
Epoch 6/500
92/92            0s 3ms/step - loss:
2.8305e-07 - val_loss: 5.5276e-04
Epoch 7/500
92/92            0s 3ms/step - loss:
2.2143e-07 - val_loss: 5.2047e-04
Epoch 8/500
92/92            0s 4ms/step - loss:
1.9736e-07 - val_loss: 4.9274e-04
Epoch 9/500
92/92            0s 3ms/step - loss:
1.8151e-07 - val_loss: 4.7383e-04
Epoch 10/500
92/92            0s 3ms/step - loss:
1.7538e-07 - val_loss: 4.5736e-04
Epoch 11/500
92/92            0s 3ms/step - loss:
1.3656e-07 - val_loss: 4.4680e-04
Epoch 12/500
92/92            0s 3ms/step - loss:
1.5518e-07 - val_loss: 4.4225e-04
Epoch 13/500
92/92            0s 3ms/step - loss:
1.4275e-07 - val_loss: 4.3232e-04
Epoch 14/500
92/92            0s 3ms/step - loss:
2.1070e-07 - val_loss: 4.2938e-04
Epoch 15/500
92/92            0s 3ms/step - loss:
1.4059e-07 - val_loss: 4.2990e-04
```

```
Epoch 16/500
92/92            0s 3ms/step - loss:
2.0753e-07 - val_loss: 4.2194e-04
Epoch 17/500
92/92            0s 3ms/step - loss:
1.6027e-07 - val_loss: 4.2971e-04
Epoch 18/500
92/92            0s 3ms/step - loss:
1.9676e-07 - val_loss: 4.2259e-04
Epoch 19/500
92/92            0s 3ms/step - loss:
1.2095e-07 - val_loss: 4.1887e-04
Epoch 20/500
92/92            0s 3ms/step - loss:
3.4036e-07 - val_loss: 4.1965e-04
Epoch 21/500
92/92            0s 4ms/step - loss:
1.3150e-07 - val_loss: 4.2277e-04
Epoch 22/500
92/92            0s 3ms/step - loss:
1.1441e-07 - val_loss: 4.2228e-04
Epoch 23/500
92/92            0s 3ms/step - loss:
1.4067e-07 - val_loss: 4.2440e-04
Epoch 24/500
92/92            0s 3ms/step - loss:
1.2333e-07 - val_loss: 4.2536e-04
Epoch 25/500
92/92            0s 3ms/step - loss:
1.7509e-07 - val_loss: 4.2114e-04
Epoch 26/500
92/92            0s 4ms/step - loss:
1.7088e-07 - val_loss: 4.2046e-04
Epoch 27/500
92/92            0s 3ms/step - loss:
1.4359e-07 - val_loss: 4.2075e-04
Epoch 28/500
92/92            0s 3ms/step - loss:
1.2858e-07 - val_loss: 4.2166e-04
Epoch 29/500
92/92            0s 4ms/step - loss:
1.8276e-07 - val_loss: 4.2113e-04
Epoch 29: early stopping
Restoring model weights from the end of the best epoch: 19.
```

[19]:
```
# Prediction
y_pred = model.predict(x_test)
```

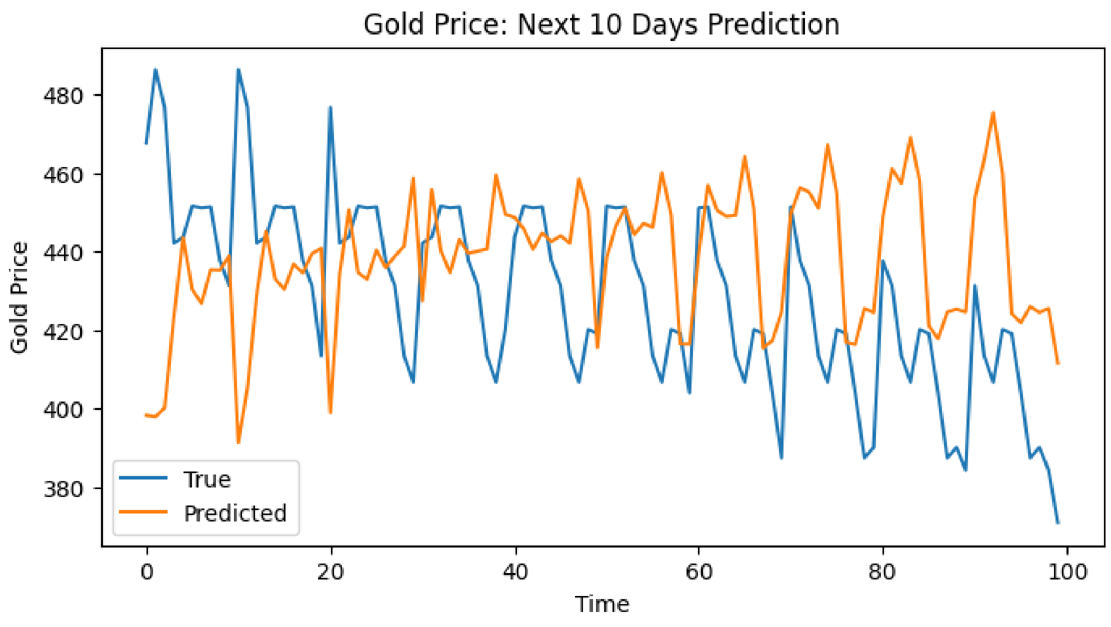
```
y_pred_orig = scaler.inverse_transform(y_pred.reshape(-1,1))
y_test_orig = scaler.inverse_transform(y_test.reshape(-1,1))

# Evaluation
rmse = np.sqrt(mean_squared_error(y_test_orig, y_pred_orig))
r2 = r2_score(y_test_orig, y_pred_orig)

print("RMSE:", rmse)
print("R² Score:", r2)

# Plot results
plt.figure(figsize=(8,4))
plt.plot(y_test_orig[:100], label='True')
plt.plot(y_pred_orig[:100], label='Predicted')
plt.title("Gold Price: Next 10 Days Prediction")
plt.xlabel("Time")
plt.ylabel("Gold Price")
plt.legend()
plt.show()
```

```
14/14                 1s 15ms/step
RMSE: 200.3901869726053
R² Score: 0.8977128957634519
```



0.3 Question 3. Next if we can check the same problem with a deep RNN. So we will introduce more layers and see how we can proceed. We will try for a many to one RNN model, where we will take 10 days continuous data and predict the next day data using deep RNN model. That is we are going to add few layers in the model.

Now this is an example of an RNN model with enough layers with other dense layers attached. try to fit this model and see the error and performance

0.3.1 Preparing the data

[20]:
```
def create_sequences(data,seq_length):
    sequences = []
    for i in range(len(data) - seq_length+1):
        sequences.append(data[i:i+seq_length])
    return np.array(sequences)

seq_length = 10
train_sequences = create_sequences(train_data,seq_length)
test_sequences = create_sequences(test_data, seq_length)


x_train_full, y_train_full = train_sequences[:,:-1], train_sequences[:, -1]
x_test, y_test = test_sequences[:,:-1], test_sequences[:, -1]

val_fraction = 0.2
val_size = int(len(x_train_full)*val_fraction)

x_val = x_train_full[-val_size:]
y_val = y_train_full[-val_size:]

x_train = x_train_full[:-val_size]
y_train = y_train_full[:-val_size]
```

[21]:
```
model = Sequential()
model.add(SimpleRNN(128, activation = 'tanh', return_sequences=True,␣
 ↪input_shape = (seq_length-1,1)))
model.add(SimpleRNN(64,activation='tanh', return_sequences=True))
model.add(SimpleRNN(32, activation='tanh'))
model.add(Dense(1))
model.compile(optimizer = 'adam', loss = 'mean_squared_error')
model.summary()

early_stop = EarlyStopping(monitor = 'val_loss', patience = 10,␣
 ↪restore_best_weights=True, verbose=1)
history = model.fit(
x_train, y_train,
epochs = 500,
batch_size = 16,
validation_data = (x_val,y_val),
verbose = 1,
callbacks = [early_stop]
)
```

```
c:\Users\TUFAN\.conda\envs\tf_env\lib\site-
packages\keras\src\layers\rnn\rnn.py:199: 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)
```

```
Model: "sequential_2"


 Layer (type)                      Output Shape                  Param #

 simple_rnn_2 (SimpleRNN)          (None, 9, 128)                 16,640

 simple_rnn_3 (SimpleRNN)          (None, 9, 64)                  12,352

 simple_rnn_4 (SimpleRNN)          (None, 32)                      3,104

 dense_2 (Dense)                   (None, 1)                          33


 Total params: 32,129 (125.50 KB)

 Trainable params: 32,129 (125.50 KB)

 Non-trainable params: 0 (0.00 B)
```

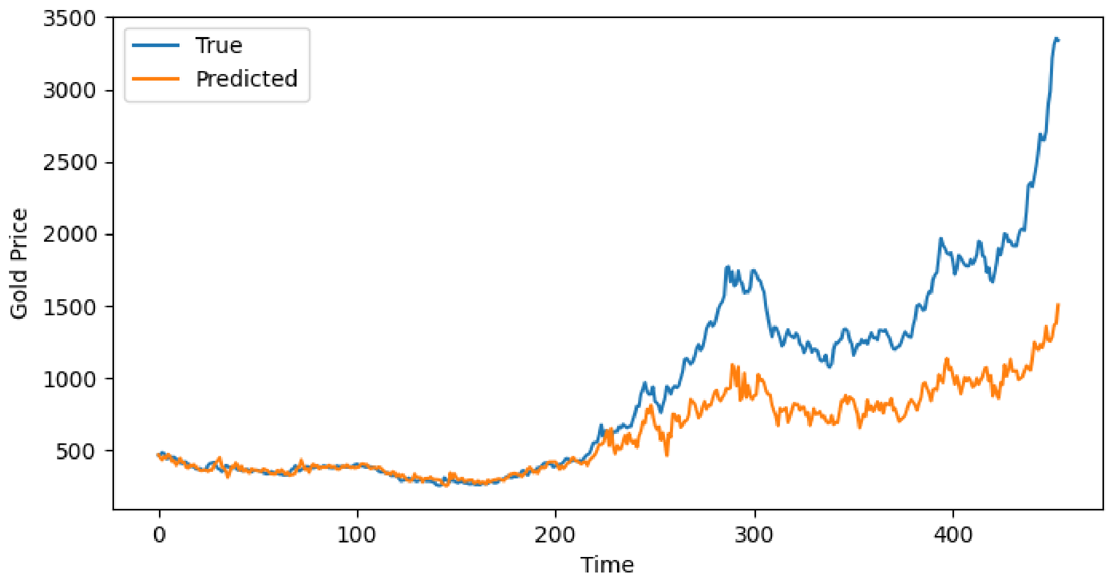
```
Epoch 1/500
92/92             3s 9ms/step - loss:
0.0064 - val_loss: 1.4657e-04
Epoch 2/500
92/92             0s 5ms/step - loss:
2.5777e-07 - val_loss: 2.5384e-04
Epoch 3/500
92/92             0s 4ms/step - loss:
5.1542e-08 - val_loss: 2.4952e-04
Epoch 4/500
92/92             0s 5ms/step - loss:
3.3795e-08 - val_loss: 2.4884e-04
Epoch 5/500
92/92             0s 5ms/step - loss:
```

```
4.5798e-08 - val_loss: 2.5312e-04
Epoch 6/500
92/92            0s 4ms/step - loss:
6.5413e-08 - val_loss: 2.4651e-04
Epoch 7/500
92/92            0s 5ms/step - loss:
4.8357e-08 - val_loss: 2.5806e-04
Epoch 8/500
92/92            0s 4ms/step - loss:
6.0504e-08 - val_loss: 2.4808e-04
Epoch 9/500
92/92            0s 5ms/step - loss:
4.2549e-08 - val_loss: 2.4556e-04
Epoch 10/500
92/92            0s 5ms/step - loss:
3.8251e-08 - val_loss: 2.2903e-04
Epoch 11/500
92/92            0s 4ms/step - loss:
6.8488e-08 - val_loss: 2.3935e-04
Epoch 11: early stopping
Restoring model weights from the end of the best epoch: 1.
```

[22]:
```
y_pred = model.predict(x_test)
y_test_orig = scaler.inverse_transform(y_test)
y_pred_orig = scaler.inverse_transform(y_pred)
rootmse = np.sqrt(mean_squared_error(y_test_orig,y_pred_orig))
r2 = r2_score(y_test_orig,y_pred_orig)
print("Root Mean Squared error:", rootmse)
print("R2 score:", r2)

##plot
plt.figure(figsize = (8,4))
plt.plot(y_test_orig, label = 'True')
plt.plot(y_pred_orig, label = 'Predicted')
plt.xlabel("Time")
plt.ylabel("Gold Price")
plt.legend()
plt.show()
```

```
15/15            0s 18ms/step
Root Mean Squared error: 498.8896169875025
R2 score: 0.4303551029635413
```


True
Predicted
Gold Price
Time
3500
3000
2500
2000
1500
1000
500
0
100
200
300
400