In [1]:
```
import torch
import torch.nn as nn
from torch import nn, optim
from torch.utils.data import DataLoader, TensorDataset
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
```

Traditional time series models like ARIMA rely on the assumption that future values are dependent on past values.

Transformers, initially designed for natural language processing, have recently been adapted for time series forecasting as they work well for time series patterns, exceling at capturing long-range dependencies and complex patterns.

How do the transformers work? see class TimeSeriesTransformer(nn.Module):

__init__ Method: This sets up our magic box with all the parts it needs.
forward Method: This is how the toy goes through the magic box, gets transformed, and comes out the other side.
Input Embedding: changes the data into a shape that the magic box can understand.
Positional Encoding: imagine we put a sticker on each datapoint to remember where it came from. This helps the magic box know the order of the data.
Transformer: the magic box has many little wizards (layers) that work together to transform the data. They look at the data from different angles (heads) and make it better.
Fully Connected Output: after the data go through the magic box, they come out of the output door. This part makes sure the data are in the right shape before they leave the box.

Training the model. see # Initialize and train the model Imagine We're Training a Smart Robot. Setting it up: Model: This is our smart robot. We tell it how to look at the data and learn from it. Criterion: This is like a teacher who tells the robot how well it's doing. If the robot makes a mistake, the teacher gives it a score. Optimizer: This is like a coach who helps the robot get better. The coach tells the robot how to adjust itself to improve.
Training the Robot.
Epochs are like rounds of practice. Each round, the robot gets better at its task.

Training Loop:
Train Mode: The robot goes into training mode, ready to learn.
Zero Gradients: The robot clears its mind of any previous mistakes.
Forward Pass: The robot looks at the training data and makes a prediction.
Calculate Loss: The teacher (criterion) tells the robot how far off its prediction was from the correct answer.
Backward Pass: The robot learns from its mistakes. It figures out how to adjust itself to do better next time.
Optimizer Step: The coach (optimizer) helps the robot make those adjustments.

Evaluation Loop:
Eval Mode: The robot goes into evaluation mode, where it doesn't learn but just checks how well it's doing.
No Gradient Calculation: The robot doesn't need to learn here, so it doesn't calculate gradients.

Test Prediction: The robot looks at the test data and makes a prediction.
Calculate Test Loss: The teacher (criterion) tells the robot how well it did on the test data.

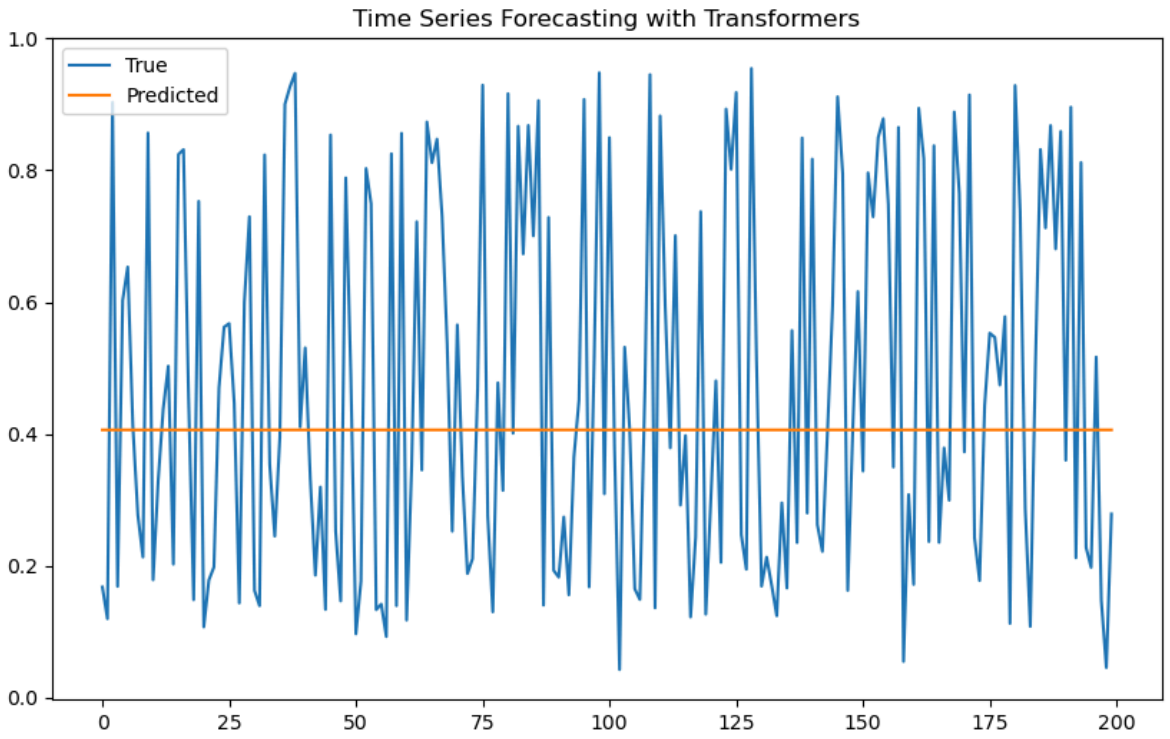
Print Progress:
After each round (epoch), we print out how well the robot did during training and testing.

In [2]:

```
# Generate fake time series data or a bunch of wavy lines
def generate_time_series(seq_length=50, n_samples=1000):   # This means each wavy line wi
    x = np.linspace(0, 4 * np.pi, seq_length)              # This sets up the points for
    y = np.sin(x) + 0.1 * np.random.normal(size=x.shape)   # This creates the base wave (
    data = []
    for _ in range(n_samples):        # We use a loop to make 1000 wavy lines. Each time,
        noise = 0.1 * np.random.normal(size=x.shape)       # Shift the sine wave slightly
        sample = np.sin(x + np.random.uniform(0, 2 * np.pi)) + noise
        data.append(sample)
    return np.array(data)

# Data preparation
seq_length = 50
data = generate_time_series(seq_length=seq_length, n_samples=1000)  # A collection of 100
scaler = MinMaxScaler()
data = scaler.fit_transform(data)

X = data[:, :-1]
y = data[:, -1]

# Convert to PyTorch tensors
train_size = int(len(X) * 0.8)
X_train, y_train = torch.tensor(X[:train_size], dtype=torch.float32), torch.tensor(y[:tra
X_test, y_test = torch.tensor(X[train_size:], dtype=torch.float32), torch.tensor(y[train_

# Transformer model for time series forecasting
class TimeSeriesTransformer(nn.Module):
    def __init__(self, input_dim, model_dim, n_heads, n_layers, dropout=0.1):
        super(TimeSeriesTransformer, self).__init__()
        self.input_embedding = nn.Linear(input_dim, model_dim)
        self.model_dim = model_dim
        self.positional_encoding = nn.Parameter(torch.zeros(1, seq_length, model_dim))
        self.transformer = nn.Transformer(d_model=model_dim, nhead=n_heads, num_encoder_la
        self.fc_out = nn.Linear(model_dim, 1)

    def forward(self, x):
        # Adjust positional encoding dynamically based on input sequence length
        seq_len = x.size(1)
        pos_encoding = self.positional_encoding[:, :seq_len, :]
        x = self.input_embedding(x) + pos_encoding

        # Transformer expects (sequence_length, batch_size, model_dim)
        x = x.permute(1, 0, 2)
        x = self.transformer(x, x)

        # Back to (batch_size, sequence_length, model_dim)
        x = x.permute(1, 0, 2)
        x = self.fc_out(x[:, -1, :])  # Predict only the last timestep
        return x


# Initialize and train the model
model = TimeSeriesTransformer(input_dim=1, model_dim=64, n_heads=4, n_layers=2)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

epochs = 20
for epoch in range(epochs):
```

```
    model.train()
    optimizer.zero_grad()  # Start fresh. Telling the robot to forget about the old mistal
    output = model(X_train.unsqueeze(-1))  # Add a feature or like a box for each datapoi
    loss = criterion(output.squeeze(), y_train) # Squeezes to flatten the robot's predict
    loss.backward()  # If the robot made a mistake, this step is the robot thinking about
    optimizer.step() # The robot actually makes the adjustments. The coach (optimizer) he

    model.eval()
    with torch.no_grad():  # Evaluate only, telling the robot to take a break from learni
        test_output = model(X_test.unsqueeze(-1))
        test_loss = criterion(test_output.squeeze(), y_test)

    print(f"Epoch {epoch + 1}, Train Loss: {loss.item():.4f}, Test Loss: {test_loss.item(

# Plot predictions
model.eval()
with torch.no_grad():
    preds = model(X_test.unsqueeze(-1)).squeeze().numpy()

plt.figure(figsize=(10, 6))
plt.plot(y_test.numpy(), label="True")
plt.plot(preds, label="Predicted")
plt.legend()
plt.title("Time Series Forecasting with Transformers")
plt.show()
```

```
C:\Users\thesk\anaconda3\lib\site-packages\torch\nn\modules\transformer.py:307: UserWarnin
g: enable_nested_tensor is True, but self.use_nested_tensor is False because encoder_laye
r.self_attn.batch_first was not True(use batch_first for better inference performance)
  warnings.warn(f"enable_nested_tensor is True, but self.use_nested_tensor is False becaus
e {why_not_sparsity_fast_path}")
Epoch 1, Train Loss: 0.1374, Test Loss: 5.2973
Epoch 2, Train Loss: 4.7981, Test Loss: 0.7983
Epoch 3, Train Loss: 0.6876, Test Loss: 0.2749
Epoch 4, Train Loss: 0.3288, Test Loss: 0.6104
Epoch 5, Train Loss: 0.6673, Test Loss: 0.3990
Epoch 6, Train Loss: 0.4492, Test Loss: 0.1595
Epoch 7, Train Loss: 0.1991, Test Loss: 0.0799
Epoch 8, Train Loss: 0.0887, Test Loss: 0.1440
Epoch 9, Train Loss: 0.1256, Test Loss: 0.2221
Epoch 10, Train Loss: 0.1822, Test Loss: 0.2347
Epoch 11, Train Loss: 0.1938, Test Loss: 0.1965
Epoch 12, Train Loss: 0.1616, Test Loss: 0.1431
Epoch 13, Train Loss: 0.1245, Test Loss: 0.1017
Epoch 14, Train Loss: 0.0978, Test Loss: 0.0821
Epoch 15, Train Loss: 0.0907, Test Loss: 0.0811
Epoch 16, Train Loss: 0.0992, Test Loss: 0.0884
Epoch 17, Train Loss: 0.1132, Test Loss: 0.0947
Epoch 18, Train Loss: 0.1206, Test Loss: 0.0953
Epoch 19, Train Loss: 0.1183, Test Loss: 0.0908
Epoch 20, Train Loss: 0.1145, Test Loss: 0.0845
```



Real life data

In [10]:
```
# Load the supermarket sales dataset to inspect its structure
file_path = 'supermarket_sales.csv'
supermarket_data = pd.read_csv(file_path)

# Display the first few rows of the dataset
supermarket_data.head(), supermarket_data.info()

# Convert the 'Date' column to datetime format and aggregate sales by day
supermarket_data['Date'] = pd.to_datetime(supermarket_data['Date'])
daily_sales = supermarket_data.groupby('Date')['Total'].sum().reset_index()

# Sort the data by date for consistency
daily_sales = daily_sales.sort_values(by='Date').reset_index(drop=True)

# Display the first few rows of the processed time-series data
daily_sales.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 17 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   Invoice ID               1000 non-null   object
 1   Branch                   1000 non-null   object
 2   City                     1000 non-null   object
 3   Customer type            1000 non-null   object
 4   Gender                   1000 non-null   object
 5   Product line             1000 non-null   object
 6   Unit price               1000 non-null   float64
 7   Quantity                 1000 non-null   int64
 8   Tax 5%                   1000 non-null   float64
 9   Total                    1000 non-null   float64
 10  Date                     1000 non-null   object
 11  Time                     1000 non-null   object
 12  Payment                  1000 non-null   object
 13  cogs                     1000 non-null   float64
 14  gross margin percentage  1000 non-null   float64
 15  gross income             1000 non-null   float64
 16  Rating                   1000 non-null   float64
dtypes: float64(7), int64(1), object(9)
memory usage: 132.9+ KB
```

Out[10]:

	Date	Total
0	2019-01-01	4745.1810
1	2019-01-02	1545.5030
2	2019-01-03	2078.1285
3	2019-01-04	1623.6885
4	2019-01-05	3539.6835

10 epochs

In [6]:

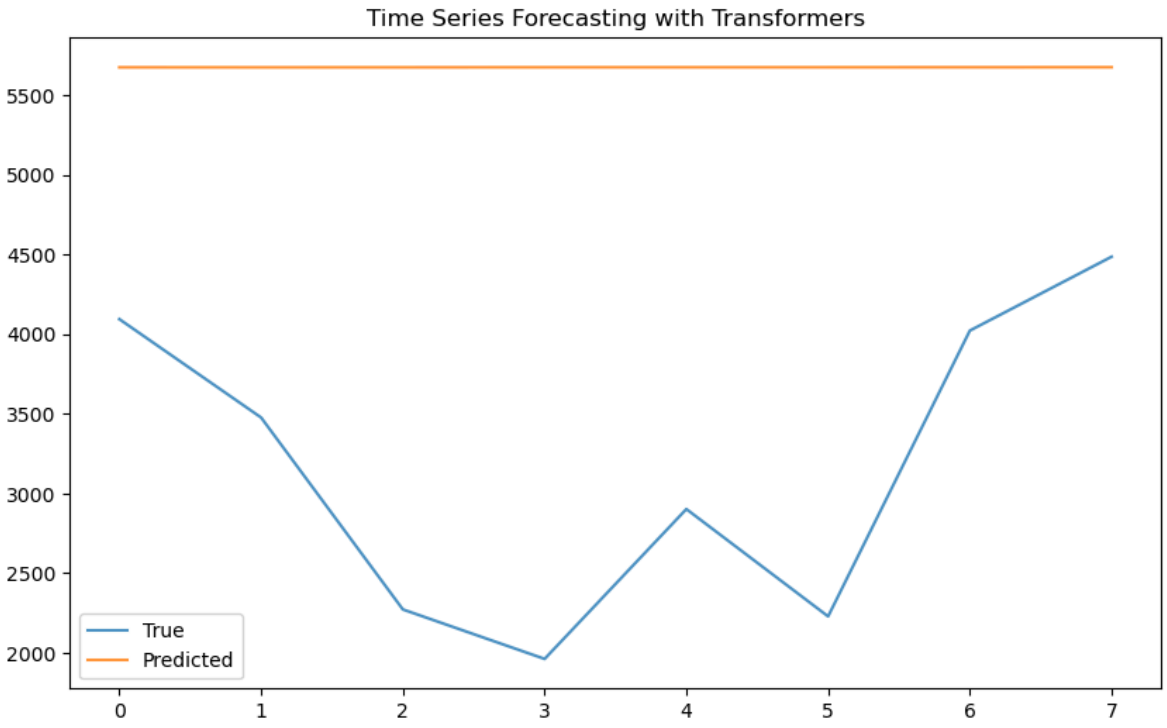
```python
# Load the supermarket sales dataset to inspect its structure
file_path = 'supermarket_sales.csv'
supermarket_data = pd.read_csv(file_path)
supermarket_data.head(), supermarket_data.info()

# Convert the 'Date' column to datetime format and aggregate sales by day
supermarket_data['Date'] = pd.to_datetime(supermarket_data['Date'])
daily_sales = supermarket_data.groupby('Date')['Total'].sum().reset_index()

# Sort the data by date for consistency
daily_sales = daily_sales.sort_values(by='Date').reset_index(drop=True)
daily_sales.head()

# Prepare the time-series data for the Transformer
sequence_length = 50

# Scale the data
scaler = MinMaxScaler()
scaled_sales = scaler.fit_transform(daily_sales['Total'].values.reshape(-1, 1))

# Create sequences and targets
sequences, targets = [], []
for i in range(len(scaled_sales) - sequence_length):
    sequences.append(scaled_sales[i:i + sequence_length])
    targets.append(scaled_sales[i + sequence_length])

sequences = np.array(sequences)
```

```
targets = np.array(targets)

# Split into training and testing sets (80% train, 20% test)
train_size = int(0.8 * len(sequences))
X_train, y_train = sequences[:train_size], targets[:train_size]
X_test, y_test = sequences[train_size:], targets[train_size:]

# Convert to PyTorch tensors
X_train, y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dtyp
X_test, y_test = torch.tensor(X_test, dtype=torch.float32), torch.tensor(y_test, dtype=to

# Create data loaders
train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=True)
test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=32)

# Verify the shapes
X_train.shape, y_train.shape, X_test.shape, y_test.shape

# Define the Transformer Model
class TimeSeriesTransformer(nn.Module):
    def __init__(self, input_dim, model_dim, num_heads, num_layers, dropout=0.1):
        super(TimeSeriesTransformer, self).__init__()
        self.input_embedding = nn.Linear(input_dim, model_dim)
        self.positional_encoding = nn.Parameter(torch.zeros(1, sequence_length, model_dim
        self.transformer = nn.Transformer(
            d_model=model_dim, nhead=num_heads, num_encoder_layers=num_layers, dropout=dr
        )
        self.fc_out = nn.Linear(model_dim, 1)

    def forward(self, x):
        x = self.input_embedding(x.squeeze(-1))  # Ensure the input has shape (batch_size,
        x = x + self.positional_encoding
        x = x.permute(1, 0, 2)  # Switch to (sequence_length, batch_size, model_dim)
        x = self.transformer(x, x)  # Transformer expects src and tgt, using the same for
        x = x.permute(1, 0, 2)  # Switch back to (batch_size, sequence_length, model_dim)
        x = self.fc_out(x[:, -1, :])  # Use the output of the last sequence step
        return x

# Initialize model, loss, and optimizer
model = TimeSeriesTransformer(input_dim=1, model_dim=64, num_heads=4, num_layers=2)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training the model
epochs = 10
for epoch in range(epochs):
    model.train()
    total_loss = 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        output = model(X_batch.unsqueeze(-1))  # Add feature dimension
        loss = criterion(output.squeeze(), y_batch)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss / len(train_loader)}")

# Evaluate and plot
model.eval()
predictions, true_values = [], []

with torch.no_grad():
    for X_batch, y_batch in test_loader:
        output = model(X_batch.unsqueeze(-1))
        predictions.extend(output.squeeze().tolist())
        true_values.extend(y_batch.tolist())
```

```python
# Rescale predictions and true values to original scale
predictions = scaler.inverse_transform(np.array(predictions).reshape(-1, 1))
true_values = scaler.inverse_transform(np.array(true_values).reshape(-1, 1))

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(true_values, label="True", alpha=0.8)
plt.plot(predictions, label="Predicted", alpha=0.8)
plt.title("Time Series Forecasting with Transformers")
plt.legend()
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 17 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   Invoice ID               1000 non-null   object 
 1   Branch                   1000 non-null   object 
 2   City                     1000 non-null   object 
 3   Customer type            1000 non-null   object 
 4   Gender                   1000 non-null   object 
 5   Product line             1000 non-null   object 
 6   Unit price               1000 non-null   float64
 7   Quantity                 1000 non-null   int64  
 8   Tax 5%                   1000 non-null   float64
 9   Total                    1000 non-null   float64
 10  Date                     1000 non-null   object 
 11  Time                     1000 non-null   object 
 12  Payment                  1000 non-null   object 
 13  cogs                     1000 non-null   float64
 14  gross margin percentage  1000 non-null   float64
 15  gross income             1000 non-null   float64
 16  Rating                   1000 non-null   float64
dtypes: float64(7), int64(1), object(9)
memory usage: 132.9+ KB
```

```
C:\Users\thesk\anaconda3\lib\site-packages\torch\nn\modules\transformer.py:307: UserWarnin
g: enable_nested_tensor is True, but self.use_nested_tensor is False because encoder_laye
r.self_attn.batch_first was not True(use batch_first for better inference performance)
  warnings.warn(f"enable_nested_tensor is True, but self.use_nested_tensor is False becaus
e {why_not_sparsity_fast_path}")
C:\Users\thesk\anaconda3\lib\site-packages\torch\nn\modules\loss.py:538: UserWarning: Usin
g a target size (torch.Size([31, 1])) that is different to the input size (torch.Size([3
1])). This will likely lead to incorrect results due to broadcasting. Please ensure they h
ave the same size.
  return F.mse_loss(input, target, reduction=self.reduction)
Epoch 1/10, Loss: 0.10171861201524734
Epoch 2/10, Loss: 7.260791301727295
Epoch 3/10, Loss: 1.177998661994934
Epoch 4/10, Loss: 0.1875714361667633
Epoch 5/10, Loss: 0.6118249297142029
Epoch 6/10, Loss: 0.4349763095378876
Epoch 7/10, Loss: 0.190816730260849
Epoch 8/10, Loss: 0.08038369566202164
Epoch 9/10, Loss: 0.11761417239904404
Epoch 10/10, Loss: 0.18173150718212128
```



50 epochs

In [5]:
```
# Load the supermarket sales dataset to inspect its structure
file_path = 'supermarket_sales.csv'
supermarket_data = pd.read_csv(file_path)
supermarket_data.head(), supermarket_data.info()

# Convert the 'Date' column to datetime format and aggregate sales by day
supermarket_data['Date'] = pd.to_datetime(supermarket_data['Date'])
daily_sales = supermarket_data.groupby('Date')['Total'].sum().reset_index()

# Sort the data by date for consistency
daily_sales = daily_sales.sort_values(by='Date').reset_index(drop=True)
daily_sales.head()

# Prepare the time-series data for the Transformer
sequence_length = 50

# Scale the data
scaler = MinMaxScaler()
scaled_sales = scaler.fit_transform(daily_sales['Total'].values.reshape(-1, 1))

# Create sequences and targets
sequences, targets = [], []
for i in range(len(scaled_sales) - sequence_length):
    sequences.append(scaled_sales[i:i + sequence_length])
    targets.append(scaled_sales[i + sequence_length])

sequences = np.array(sequences)
targets = np.array(targets)

# Split into training and testing sets (80% train, 20% test)
train_size = int(0.8 * len(sequences))
X_train, y_train = sequences[:train_size], targets[:train_size]
X_test, y_test = sequences[train_size:], targets[train_size:]

# Convert to PyTorch tensors
X_train, y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dtyp
X_test, y_test = torch.tensor(X_test, dtype=torch.float32), torch.tensor(y_test, dtype=to
```

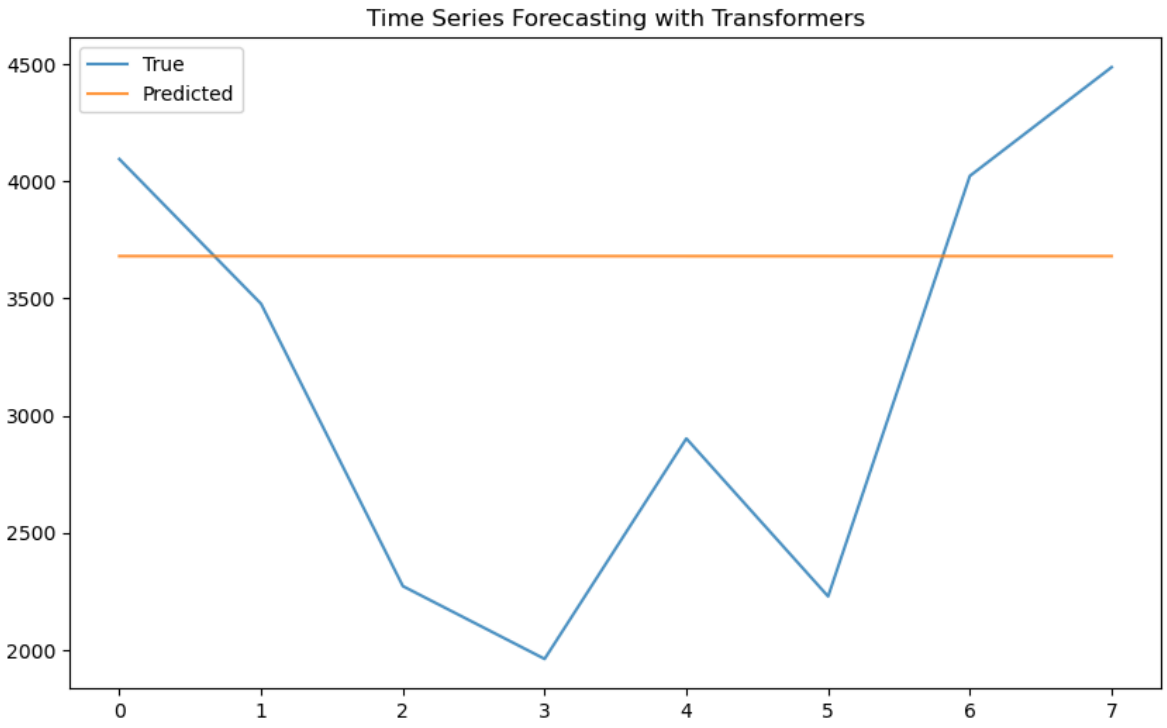
```python
# Create data loaders
train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=True)
test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=32)

# Verify the shapes
X_train.shape, y_train.shape, X_test.shape, y_test.shape

# Define the Transformer Model
class TimeSeriesTransformer(nn.Module):
    def __init__(self, input_dim, model_dim, num_heads, num_layers, dropout=0.1):
        super(TimeSeriesTransformer, self).__init__()
        self.input_embedding = nn.Linear(input_dim, model_dim)
        self.positional_encoding = nn.Parameter(torch.zeros(1, sequence_length, model_dim
        self.transformer = nn.Transformer(
            d_model=model_dim, nhead=num_heads, num_encoder_layers=num_layers, dropout=dr
        )
        self.fc_out = nn.Linear(model_dim, 1)

    def forward(self, x):
        x = self.input_embedding(x.squeeze(-1))  # Ensure the input has shape (batch_size,
        x = x + self.positional_encoding
        x = x.permute(1, 0, 2)  # Switch to (sequence_length, batch_size, model_dim)
        x = self.transformer(x, x)  # Transformer expects src and tgt, using the same for
        x = x.permute(1, 0, 2)  # Switch back to (batch_size, sequence_length, model_dim)
        x = self.fc_out(x[:, -1, :])  # Use the output of the last sequence step
        return x

# Initialize model, loss, and optimizer
model = TimeSeriesTransformer(input_dim=1, model_dim=64, num_heads=4, num_layers=2)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training the model
epochs = 50
for epoch in range(epochs):
    model.train()
    total_loss = 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        output = model(X_batch.unsqueeze(-1))  # Add feature dimension
        loss = criterion(output.squeeze(), y_batch)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss / len(train_loader)}")

# Evaluate and plot
model.eval()
predictions, true_values = [], []

with torch.no_grad():
    for X_batch, y_batch in test_loader:
        output = model(X_batch.unsqueeze(-1))
        predictions.extend(output.squeeze().tolist())
        true_values.extend(y_batch.tolist())

# Rescale predictions and true values to original scale
predictions = scaler.inverse_transform(np.array(predictions).reshape(-1, 1))
true_values = scaler.inverse_transform(np.array(true_values).reshape(-1, 1))

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(true_values, label="True", alpha=0.8)
plt.plot(predictions, label="Predicted", alpha=0.8)
plt.title("Time Series Forecasting with Transformers")
```

```
plt.legend()
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 17 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   Invoice ID               1000 non-null   object
 1   Branch                   1000 non-null   object
 2   City                     1000 non-null   object
 3   Customer type            1000 non-null   object
 4   Gender                   1000 non-null   object
 5   Product line             1000 non-null   object
 6   Unit price               1000 non-null   float64
 7   Quantity                 1000 non-null   int64
 8   Tax 5%                   1000 non-null   float64
 9   Total                    1000 non-null   float64
 10  Date                     1000 non-null   object
 11  Time                     1000 non-null   object
 12  Payment                  1000 non-null   object
 13  cogs                     1000 non-null   float64
 14  gross margin percentage  1000 non-null   float64
 15  gross income             1000 non-null   float64
 16  Rating                   1000 non-null   float64
dtypes: float64(7), int64(1), object(9)
memory usage: 132.9+ KB
```

```
C:\Users\thesk\anaconda3\lib\site-packages\torch\nn\modules\transformer.py:307: UserWarnin
g: enable_nested_tensor is True, but self.use_nested_tensor is False because encoder_laye
r.self_attn.batch_first was not True(use batch_first for better inference performance)
  warnings.warn(f"enable_nested_tensor is True, but self.use_nested_tensor is False becaus
e {why_not_sparsity_fast_path}")
C:\Users\thesk\anaconda3\lib\site-packages\torch\nn\modules\loss.py:538: UserWarning: Usin
g a target size (torch.Size([31, 1])) that is different to the input size (torch.Size([3
1])). This will likely lead to incorrect results due to broadcasting. Please ensure they h
ave the same size.
  return F.mse_loss(input, target, reduction=self.reduction)
```

```
Epoch 1/50, Loss: 0.4354504644870758
Epoch 2/50, Loss: 8.003783226013184
Epoch 3/50, Loss: 1.2868690490722656
Epoch 4/50, Loss: 0.2025356888771057
Epoch 5/50, Loss: 0.7075115442276001
Epoch 6/50, Loss: 0.4719478487968445
Epoch 7/50, Loss: 0.17422634363174438
Epoch 8/50, Loss: 0.0815255418419838
Epoch 9/50, Loss: 0.12688380479812622
Epoch 10/50, Loss: 0.2193678617477417
Epoch 11/50, Loss: 0.2188984602689743
Epoch 12/50, Loss: 0.15632259845733643
Epoch 13/50, Loss: 0.12006177753210068
Epoch 14/50, Loss: 0.0812406837940216
Epoch 15/50, Loss: 0.0771125853061676
Epoch 16/50, Loss: 0.1044146716594696
Epoch 17/50, Loss: 0.1220683678984642
Epoch 18/50, Loss: 0.13628429174423218
Epoch 19/50, Loss: 0.10992175340652466
Epoch 20/50, Loss: 0.10208961367607117
Epoch 21/50, Loss: 0.0895366296172142
Epoch 22/50, Loss: 0.08261831104755402
Epoch 23/50, Loss: 0.07649055868387222
Epoch 24/50, Loss: 0.07688829302787781
Epoch 25/50, Loss: 0.08509210497140884
Epoch 26/50, Loss: 0.08511574566364288
Epoch 27/50, Loss: 0.09601689875125885
Epoch 28/50, Loss: 0.09335649013519287
Epoch 29/50, Loss: 0.08709117770195007
Epoch 30/50, Loss: 0.08291696757078171
Epoch 31/50, Loss: 0.07679128646850586
Epoch 32/50, Loss: 0.07631276547908783
Epoch 33/50, Loss: 0.07467085868120193
Epoch 34/50, Loss: 0.08010407537221909
Epoch 35/50, Loss: 0.08552033454179764
Epoch 36/50, Loss: 0.0802275538444519
Epoch 37/50, Loss: 0.0863845944404602
Epoch 38/50, Loss: 0.07794713973999023
Epoch 39/50, Loss: 0.07976798713207245
Epoch 40/50, Loss: 0.07695821672677994
Epoch 41/50, Loss: 0.07769594341516495
Epoch 42/50, Loss: 0.0758724957704544
Epoch 43/50, Loss: 0.07566829770803452
Epoch 44/50, Loss: 0.07831770181655884
Epoch 45/50, Loss: 0.07505082339048386
Epoch 46/50, Loss: 0.07959333807229996
Epoch 47/50, Loss: 0.07515168935060501
Epoch 48/50, Loss: 0.07918087393045425
Epoch 49/50, Loss: 0.07528778165578842
Epoch 50/50, Loss: 0.0753898024559021
```



In [4]:
```
print(X_batch.shape)
```

```
torch.Size([8, 50, 1])
```

100 epochs

In [7]:
```
# Load the supermarket sales dataset to inspect its structure
file_path = 'supermarket_sales.csv'
supermarket_data = pd.read_csv(file_path)
supermarket_data.head(), supermarket_data.info()

# Convert the 'Date' column to datetime format and aggregate sales by day
supermarket_data['Date'] = pd.to_datetime(supermarket_data['Date'])
daily_sales = supermarket_data.groupby('Date')['Total'].sum().reset_index()

# Sort the data by date for consistency
daily_sales = daily_sales.sort_values(by='Date').reset_index(drop=True)
daily_sales.head()

# Prepare the time-series data for the Transformer
sequence_length = 50

# Scale the data
scaler = MinMaxScaler()
scaled_sales = scaler.fit_transform(daily_sales['Total'].values.reshape(-1, 1))

# Create sequences and targets
sequences, targets = [], []
for i in range(len(scaled_sales) - sequence_length):
    sequences.append(scaled_sales[i:i + sequence_length])
    targets.append(scaled_sales[i + sequence_length])

sequences = np.array(sequences)
targets = np.array(targets)

# Split into training and testing sets (80% train, 20% test)
train_size = int(0.8 * len(sequences))
X_train, y_train = sequences[:train_size], targets[:train_size]
X_test, y_test = sequences[train_size:], targets[train_size:]

# Convert to PyTorch tensors
```

```
X_train, y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dtyp
X_test, y_test = torch.tensor(X_test, dtype=torch.float32), torch.tensor(y_test, dtype=to

# Creates data loaders to efficiently load the training and testing data in batches
train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=True)
test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=32)

# Verify the shapes
X_train.shape, y_train.shape, X_test.shape, y_test.shape

# Define the Transformer Model
class TimeSeriesTransformer(nn.Module):
    def __init__(self, input_dim, model_dim, num_heads, num_layers, dropout=0.1):
        super(TimeSeriesTransformer, self).__init__()
        self.input_embedding = nn.Linear(input_dim, model_dim)
        self.positional_encoding = nn.Parameter(torch.zeros(1, sequence_length, model_dim
        self.transformer = nn.Transformer(
            d_model=model_dim, nhead=num_heads, num_encoder_layers=num_layers, dropout=dr
        )
        self.fc_out = nn.Linear(model_dim, 1)

    def forward(self, x):
        x = self.input_embedding(x.squeeze(-1))  # Ensure the input has shape (batch_size
        x = x + self.positional_encoding
        x = x.permute(1, 0, 2)  # Switch to (sequence_length, batch_size, model_dim)
        x = self.transformer(x, x)  # Transformer expects src and tgt, using the same for
        x = x.permute(1, 0, 2)  # Switch back to (batch_size, sequence_length, model_dim)
        x = self.fc_out(x[:, -1, :])  # Use the output of the last sequence step
        return x

# Initialize model, loss, and optimizer
model = TimeSeriesTransformer(input_dim=1, model_dim=64, num_heads=4, num_layers=2)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training the model
epochs = 100
for epoch in range(epochs):
    model.train()
    total_loss = 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        output = model(X_batch.unsqueeze(-1))  # Add feature dimension
        loss = criterion(output.squeeze(), y_batch)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss / len(train_loader)}")

# Evaluate and plot
model.eval()
predictions, true_values = [], []

with torch.no_grad():
    for X_batch, y_batch in test_loader:
        output = model(X_batch.unsqueeze(-1))
        predictions.extend(output.squeeze().tolist())
        true_values.extend(y_batch.tolist())

# Rescale predictions and true values to original scale
predictions = scaler.inverse_transform(np.array(predictions).reshape(-1, 1))
true_values = scaler.inverse_transform(np.array(true_values).reshape(-1, 1))

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(true_values, label="True", alpha=0.8)
plt.plot(predictions, label="Predicted", alpha=0.8)
```

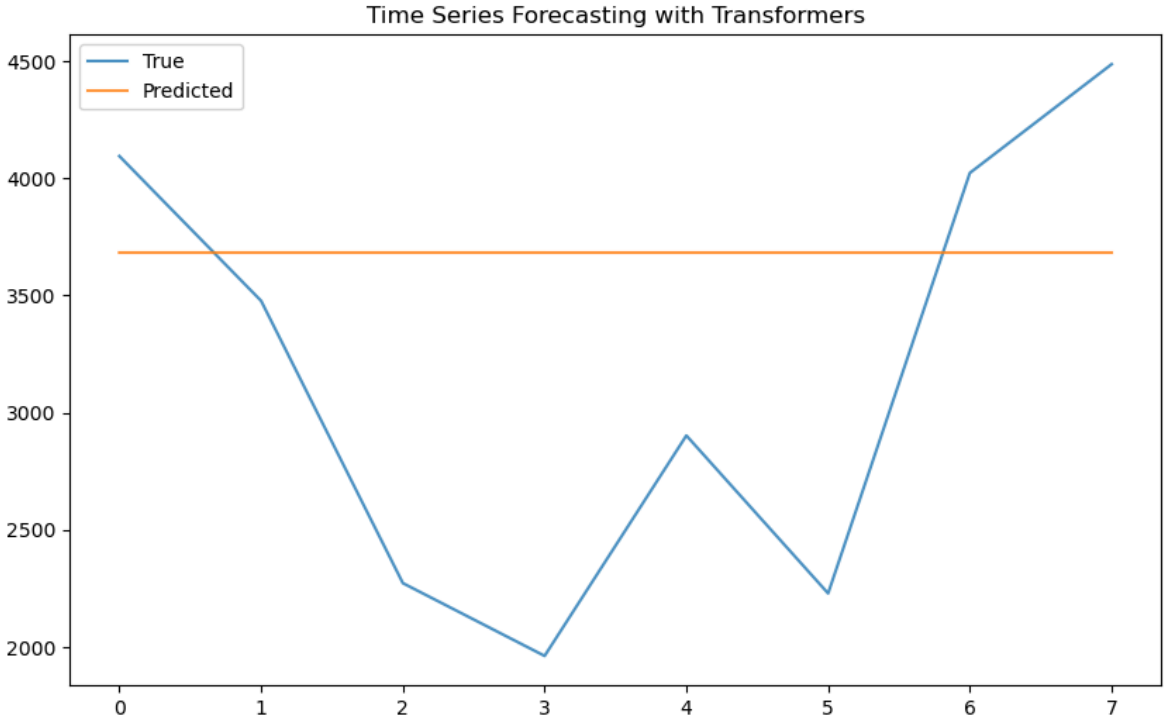
```
plt.title("Time Series Forecasting with Transformers")
plt.legend()
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 17 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   Invoice ID               1000 non-null   object 
 1   Branch                   1000 non-null   object 
 2   City                     1000 non-null   object 
 3   Customer type            1000 non-null   object 
 4   Gender                   1000 non-null   object 
 5   Product line             1000 non-null   object 
 6   Unit price               1000 non-null   float64
 7   Quantity                 1000 non-null   int64  
 8   Tax 5%                   1000 non-null   float64
 9   Total                    1000 non-null   float64
 10  Date                     1000 non-null   object 
 11  Time                     1000 non-null   object 
 12  Payment                  1000 non-null   object 
 13  cogs                     1000 non-null   float64
 14  gross margin percentage  1000 non-null   float64
 15  gross income             1000 non-null   float64
 16  Rating                   1000 non-null   float64
dtypes: float64(7), int64(1), object(9)
memory usage: 132.9+ KB
```

```
C:\Users\thesk\anaconda3\lib\site-packages\torch\nn\modules\transformer.py:307: UserWarnin
g: enable_nested_tensor is True, but self.use_nested_tensor is False because encoder_laye
r.self_attn.batch_first was not True(use batch_first for better inference performance)
  warnings.warn(f"enable_nested_tensor is True, but self.use_nested_tensor is False becaus
e {why_not_sparsity_fast_path}")
C:\Users\thesk\anaconda3\lib\site-packages\torch\nn\modules\loss.py:538: UserWarning: Usin
g a target size (torch.Size([31, 1])) that is different to the input size (torch.Size([3
1])). This will likely lead to incorrect results due to broadcasting. Please ensure they h
ave the same size.
  return F.mse_loss(input, target, reduction=self.reduction)
```

```
Epoch 1/100, Loss: 1.0127911567687988
Epoch 2/100, Loss: 6.748830318450928
Epoch 3/100, Loss: 2.0468225479125977
Epoch 4/100, Loss: 0.13288745284080505
Epoch 5/100, Loss: 0.34944358468055725
Epoch 6/100, Loss: 0.7091167569160461
Epoch 7/100, Loss: 0.5169772505760193
Epoch 8/100, Loss: 0.23349213600158691
Epoch 9/100, Loss: 0.08730469644069672
Epoch 10/100, Loss: 0.10717874020338058
Epoch 11/100, Loss: 0.20853640139102936
Epoch 12/100, Loss: 0.24381765723228455
Epoch 13/100, Loss: 0.2193818837404251
Epoch 14/100, Loss: 0.15545552968978882
Epoch 15/100, Loss: 0.1193750724196434
Epoch 16/100, Loss: 0.08029663562774658
Epoch 17/100, Loss: 0.08232636749744415
Epoch 18/100, Loss: 0.08972978591918945
Epoch 19/100, Loss: 0.11034460365772247
Epoch 20/100, Loss: 0.12024503946304321
Epoch 21/100, Loss: 0.11286495625972748
Epoch 22/100, Loss: 0.10967258363962173
Epoch 23/100, Loss: 0.09742312133312225
Epoch 24/100, Loss: 0.08520285785198212
Epoch 25/100, Loss: 0.08964531123638153
Epoch 26/100, Loss: 0.07400774955749512
Epoch 27/100, Loss: 0.07773391902446747
Epoch 28/100, Loss: 0.08264531940221786
Epoch 29/100, Loss: 0.08740779757499695
Epoch 30/100, Loss: 0.08762872964143753
Epoch 31/100, Loss: 0.08725845813751221
Epoch 32/100, Loss: 0.08343788236379623
Epoch 33/100, Loss: 0.0881424993276596
Epoch 34/100, Loss: 0.08382187783718109
Epoch 35/100, Loss: 0.07957571744918823
Epoch 36/100, Loss: 0.07971730828285217
Epoch 37/100, Loss: 0.07627003639936447
Epoch 38/100, Loss: 0.0800226703286171
Epoch 39/100, Loss: 0.08527135848999023
Epoch 40/100, Loss: 0.0783003643155098
Epoch 41/100, Loss: 0.08345404267311096
Epoch 42/100, Loss: 0.0820423811674118
Epoch 43/100, Loss: 0.08243689686059952
Epoch 44/100, Loss: 0.07825823128223419
Epoch 45/100, Loss: 0.07952045649290085
Epoch 46/100, Loss: 0.08044349402189255
Epoch 47/100, Loss: 0.07946986705064774
Epoch 48/100, Loss: 0.07573408633470535
Epoch 49/100, Loss: 0.0770542323589325
Epoch 50/100, Loss: 0.07981651276350021
Epoch 51/100, Loss: 0.07868500053882599
Epoch 52/100, Loss: 0.07939756661653519
Epoch 53/100, Loss: 0.07884439080953598
Epoch 54/100, Loss: 0.07664717733860016
Epoch 55/100, Loss: 0.07598010450601578
Epoch 56/100, Loss: 0.07478012144565582
Epoch 57/100, Loss: 0.07671282440423965
Epoch 58/100, Loss: 0.07895170897245407
Epoch 59/100, Loss: 0.07626534253358841
Epoch 60/100, Loss: 0.07394526898860931
Epoch 61/100, Loss: 0.07774597406387329
Epoch 62/100, Loss: 0.07680416107177734
Epoch 63/100, Loss: 0.07604176551103592
Epoch 64/100, Loss: 0.07905716449022293
Epoch 65/100, Loss: 0.07705195248126984
Epoch 66/100, Loss: 0.0769982859492302
Epoch 67/100, Loss: 0.07517679780721664
```

```
Epoch 68/100, Loss: 0.0756845623254776
Epoch 69/100, Loss: 0.07839549332857132
Epoch 70/100, Loss: 0.0769963264465332
Epoch 71/100, Loss: 0.07952249050140381
Epoch 72/100, Loss: 0.07602357119321823
Epoch 73/100, Loss: 0.07595216482877731
Epoch 74/100, Loss: 0.07796413451433182
Epoch 75/100, Loss: 0.0773812010884285
Epoch 76/100, Loss: 0.07489200681447983
Epoch 77/100, Loss: 0.0747077539563179
Epoch 78/100, Loss: 0.07871881127357483
Epoch 79/100, Loss: 0.07635742425918579
Epoch 80/100, Loss: 0.07674969732761383
Epoch 81/100, Loss: 0.07902040332555771
Epoch 82/100, Loss: 0.07988692075014114
Epoch 83/100, Loss: 0.07366636395454407
Epoch 84/100, Loss: 0.07590731978416443
Epoch 85/100, Loss: 0.07429580390453339
Epoch 86/100, Loss: 0.07669607549905777
Epoch 87/100, Loss: 0.0778823047876358
Epoch 88/100, Loss: 0.07657656818628311
Epoch 89/100, Loss: 0.07874392718076706
Epoch 90/100, Loss: 0.075896717607975
Epoch 91/100, Loss: 0.07575180381536484
Epoch 92/100, Loss: 0.07552846521139145
Epoch 93/100, Loss: 0.07544749975204468
Epoch 94/100, Loss: 0.0741904228925705
Epoch 95/100, Loss: 0.07763507217168808
Epoch 96/100, Loss: 0.07474330812692642
Epoch 97/100, Loss: 0.07428950816392899
Epoch 98/100, Loss: 0.07598082721233368
Epoch 99/100, Loss: 0.07430050522089005
Epoch 100/100, Loss: 0.07406464219093323
```



10 epochs were not enough to train the model.

No difference between 50 and 100 epochs.