Deep Learning

Final Project

Transformer

&

iTransformer

Author

Sina Heydari

Spring 2024

Table Of Contents

1 Introduction

This document provides an overview and implementation details of the Transformer and iTransformer models for exchange rate prediction. The models are implemented using PyTorch and trained on historical exchange rate data.

2 Data Preparation

The exchange rate data is loaded and preprocessed. The dataset contains exchange rates for various currencies.

Listing 1: Data Loading and Preprocessing

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import math
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import mean_squared_error,
    mean_absolute_error
from sklearn.model_selection import train_test_split

# Path to the exchange rate data file
data_path = 'exchange_rate.txt'

# Load the exchange rate data into a pandas DataFrame
df = pd.read_csv(data_path)

# Assign column names to the DataFrame
column_names = ['Australia', 'British', 'Canada', '
    Switzerland', 'China', 'Japan', 'New Zealand', 'Singapore'
    ]
df.columns = column_names

# Display the first few rows of the DataFrame
df.head()
```

The exchange rate data is loaded from a text file into a pandas DataFrame. Column names are assigned to each of the currencies included in the dataset, such as Australia, British, Canada, Switzerland, China, Japan, New Zealand, and Singapore. Displaying the first few rows of the DataFrame helps verify that the data has been loaded correctly.

3 Data Visualization

The data is visualized to understand the distribution and trends of exchange rates.

Listing 2: Data Visualization

```
# Plotting the exchange rates over time
plt.figure(figsize=(12, 6))
for currency in df.columns:
    plt.plot(df.index, df[currency], label=currency)

plt.title('Exchange Rates Over Time')
plt.xlabel('Time')
plt.ylabel('Exchange Rate')
plt.legend()
plt.grid(True)
plt.show()

# Plotting the distribution of exchange rates for each
    currency
plt.figure(figsize=(10, 6))
df.boxplot()
plt.title('Distribution of Exchange Rates for Each Currency')
plt.ylabel('Exchange Rate')
plt.xticks(rotation=45)
plt.show()

# Print the number of rows and columns in the DataFrame
num_rows, num_columns = df.shape
print("Number of rows:", num_rows)
print("Number of columns:", num_columns)
```

Two types of visualizations are created to understand the data better. The first visualization plots the exchange rates over time for each currency, providing insight into trends and fluctuations over the dataset period. The second visualization is a box plot showing the distribution of exchange rates for each currency, which helps identify the range, median, and any potential outliers. The total number of rows and columns in the DataFrame is also printed to understand the dataset's size.

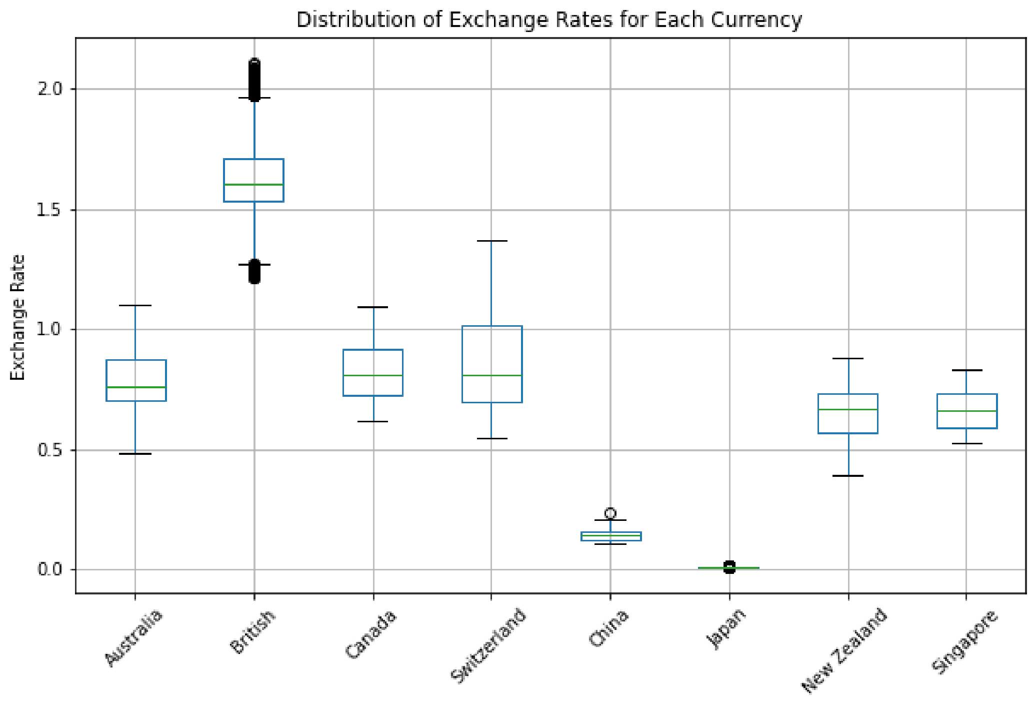


Figure 1: Distribution of Exchange Rates for Each Currency

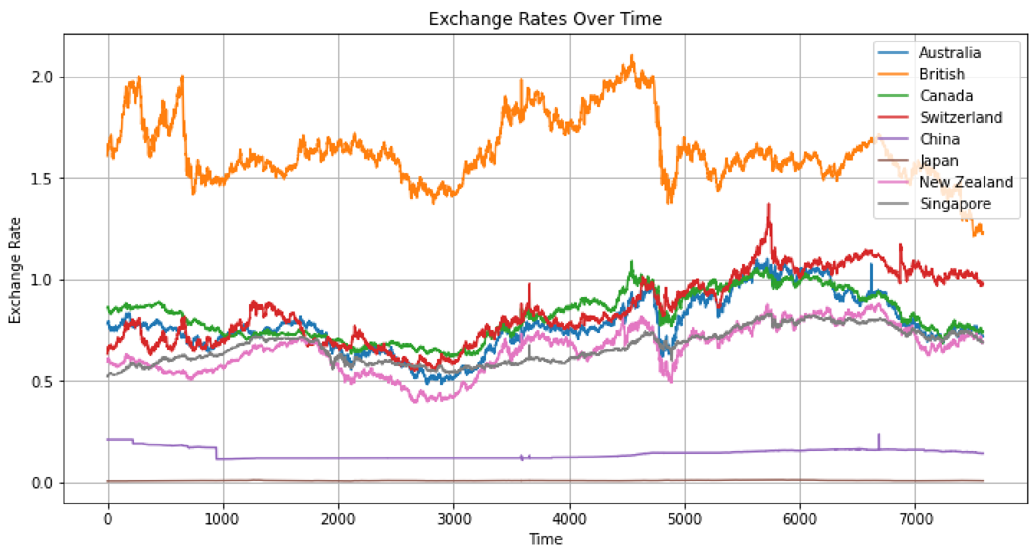


Figure 2: Exchange Rates Over Time

4 Transformer Model

The Transformer model is implemented with embedding, encoder, and decoder blocks.

Listing 3: Transformer Model Implementation

```
# Custom Embedding class for the transformer model
class Embedding(nn.Module):
    def __init__(self, input_dimension, output_dimension,
        intermediate_dimension=1024):
        super(Embedding, self).__init__()
        self.fullyconnected = nn.Sequential(
            nn.Linear(input_dimension, intermediate_dimension
                ),
            nn.ReLU(),
            nn.Linear(intermediate_dimension,
                output_dimension),
            nn.ReLU()
        )

    def forward(self, x):
        return self.fullyconnected(x)

# Positional Embedding class to add positional information to
    the embeddings
class PositionalEmbedding(nn.Module):
    def __init__(self, dimension, max_length=1000):
        super(PositionalEmbedding, self).__init__()
        positionalembedding = torch.zeros(max_length,
            dimension).float()
        positionalembedding.requires_grad = False

        position = torch.arange(0, max_length).float().
            unsqueeze(1)
        division_term = (torch.arange(0, dimension, 2).float
            () * -(math.log(10000.0) / dimension)).exp()
        positionalembedding[:, 0::2] = torch.sin(position *
            division_term)
        positionalembedding[:, 1::2] = torch.cos(position *
            division_term)
        positionalembedding = positionalembedding.unsqueeze
            (0)

        self.register_buffer('positionalembedding',
            positionalembedding)

    def forward(self, x):
        return self.positionalembedding[:, :x.size(1)]
```

```python
# Encoder Block class for the transformer model
class EncoderBlock(nn.Module):
    def __init__(self, dimension, heads,
        intermediate_dimension, dropout=0.1):
        super(EncoderBlock, self).__init__()
        self.Attention = nn.MultiheadAttention(dimension,
            heads, dropout=dropout, batch_first=True)
        self.norm1 = nn.LayerNorm(dimension)
        self.norm2 = nn.LayerNorm(dimension)
        self.fullyconnected = nn.Sequential(
            nn.Linear(dimension, intermediate_dimension),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(intermediate_dimension, dimension),
            nn.ReLU(),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        attention_out, attention_weights = self.Attention(x,
            x, x)
        linear_out = self.fullyconnected(x)
        x = self.norm1(x + attention_out)
        x = self.norm2(x + linear_out)
        return x
```

The custom embedding class transforms the input features into a higher-dimensional space using fully connected layers followed by ReLU activations. The positional embedding class adds information about the position of each element in the sequence, which is crucial for the transformer model to capture the order of data. This is done by generating sine and cosine functions of different frequencies.

The encoder block consists of multi-head self-attention mechanisms and fully connected layers. It also includes layer normalization and dropout for regularization. The encoder class is composed of multiple encoder blocks stacked together, which processes the input sequence to create a rich representation.

Similar to the encoder block, the decoder block also includes self-attention and cross-attention mechanisms, fully connected layers, and layer normalization. The cross-attention mechanism allows the decoder to focus on relevant parts of the encoder's output. The decoder class stacks multiple decoder blocks to transform the encoded sequence into the final output.

The transformer model combines the embedding, positional embedding, encoder, and decoder into a single architecture. The model uses the encoder to process the input sequence and the decoder to generate the output sequence. The linear layers at the end map the decoder's output to the desired output dimension.

Listing 4: Training the Transformer Model

```
# Check for device availability and,
# create model,
# loss function and,
# optimizer
device = 'cuda' if torch.cuda.is_available() else 'cpu'
transformer_model = Transformer(N=6, input_dimension=8,
    output_dimension=8, dimension=512, heads=4,
    intermediate_dimension=1024).to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(transformer_model.parameters(), lr=2e
    -4)

# Training the transformer model
epochs = 100
losses = []

for epoch in range(epochs):
    running_loss = 0
    for x, y in train_loader:
        x = x.float().to(device)
        y = y.float().to(device)
        optimizer.zero_grad()
        # Assuming the -1 as the starter token
        decoder_in = (torch.ones_like(y) * -1).float().to(
            device)
        y_pred = transformer_model(x, decoder_in)
        loss = criterion(y_pred, y)
        loss.backward()
        running_loss += loss.item()
        optimizer.step()
    print(f'''Epoch {epoch+1}/{epochs}
          --> Mean Saquared Eror Loss --> {running_loss}''')
    losses.append(running_loss)

# Plotting the training loss over epochs
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Mean Squared Error loss')
plt.show()
```

The training process for the transformer model involves defining the model, loss function (Mean Squared Error), and optimizer (Adam). The model is trained for a specified number of epochs, during which the loss is calculated and back-propagated to update the model's weights. The training loss is plotted to visualize the model's performance over time.

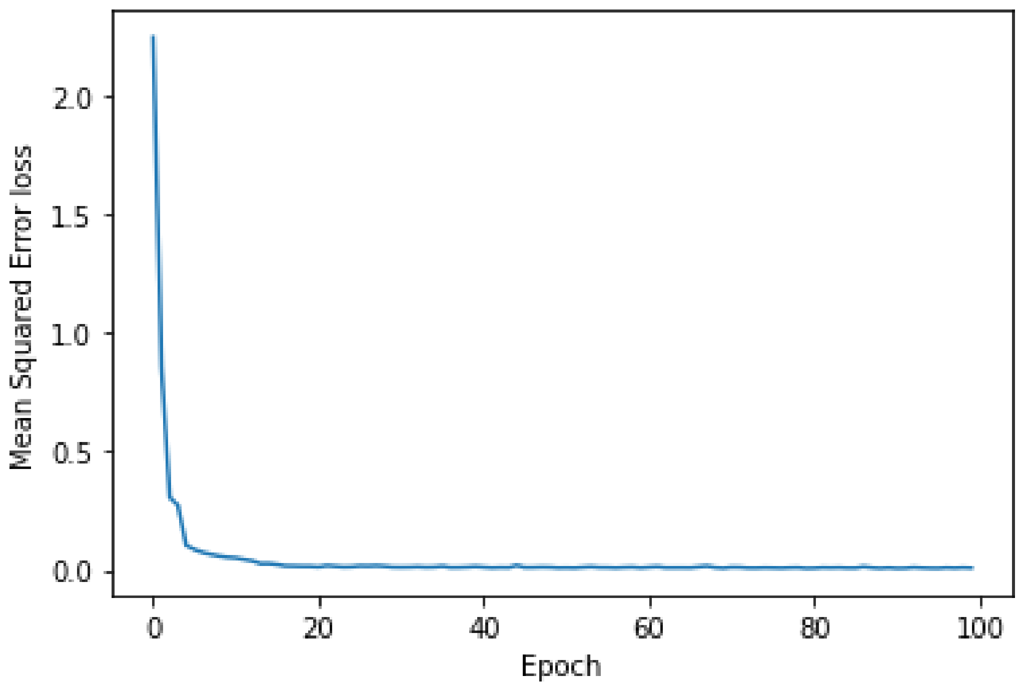


Figure 3: Transformer Model Training Loss

5 iTransformer Model

The iTransformer model is a variant of the Transformer model, simplified for specific tasks.

Listing 5: iTransformer Model Implementation

```
# iTransformer
class iTransformer(nn.Module):
    def __init__(self, N, input_dimension, output_dimension,
        dimension, heads, intermediate_dimension, dropout=0.1)
        :
        super(iTransformer, self).__init__()
        self.embedder = Embedding(input_dimension, dimension)
        self.encoder = Encoder(N, dimension, heads,
            intermediate_dimension, dropout)
        self.projection = nn.Sequential(
            nn.Linear(dimension, intermediate_dimension),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(intermediate_dimension,
                output_dimension),
            )

    def forward(self,x):
        x = x.permute(0,2,1)
        x = self.embedder(x)
        x = self.encoder(x)
        x = self.projection(x)
        x = x.permute(0,2,1)
        return x
```

The iTransformer model is a simplified variant of the transformer model. It uses an embedding layer and an encoder but skips the decoder, making it suitable for tasks where a simpler architecture is sufficient. The forward pass permutes the input dimensions, processes the input through the embedding and encoder, and then projects the output to the desired dimension.

Listing 6: Training the iTransformer Model

```
# Training the itransformer model
iTransformer_model = iTransformer(6, 10, 1, 512, 4, 1024).to(
    device)
criterion = nn.MSELoss()
optimizer = optim.Adam(iTransformer_model.parameters(), lr=2e
    -4)

epochs= 100
losses = []
```

```
for epoch in range(epochs):
  running_loss = 0
  for x, y in train_loader:
      x = x.float().to(device)
      y = y.float().to(device)
      optimizer.zero_grad()
      y_pred = iTransformer_model(x)
      loss = criterion(y_pred, y)
      loss.backward()
      running_loss += loss.item()
      optimizer.step()
  print(f'''Epoch {epoch+1}/{epochs}
       --> Mean Saquared Eror Loss --> {running_loss}''')
  losses.append(running_loss)

plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Mean Squared Eror loss')
plt.show()
```

Similar to the transformer model, the iTransformer model is trained using Mean Squared Error as the loss function and Adam as the optimizer. The training process involves updating the model's weights for a specified number of epochs. The training loss is plotted to visualize the model's performance over time

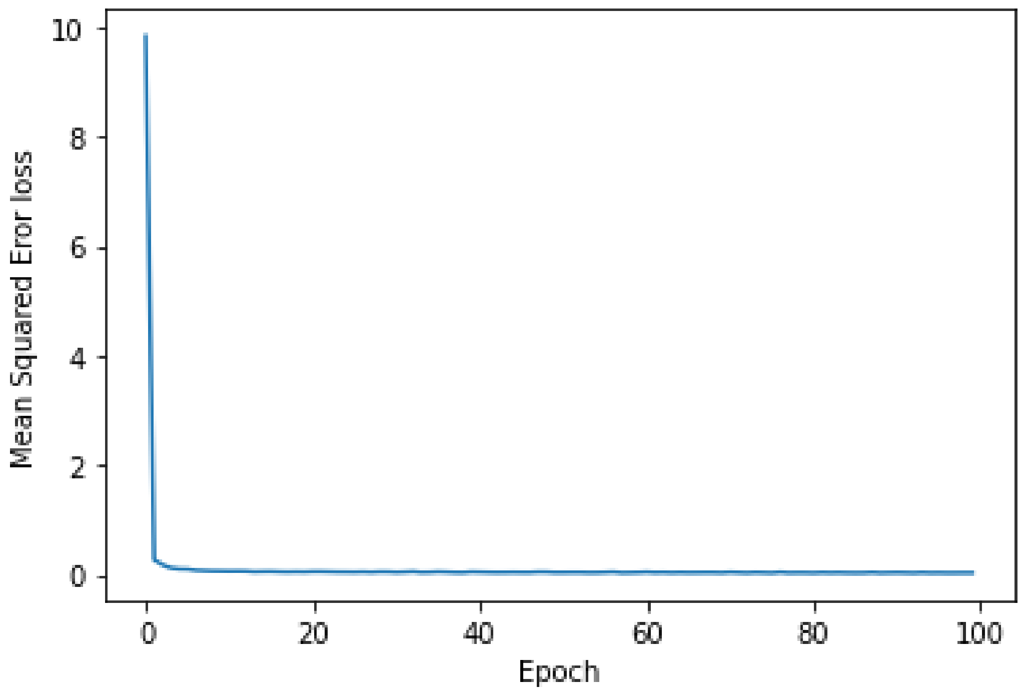


Figure 4: iTransformer Model Training Loss

6 Results

Both models are evaluated on a test set using Mean Squared Error (MSE) and Mean Absolute Error (MAE). These metrics provide insight into the models' accuracy in predicting exchange rates. The trained models are saved for future use, and the evaluation results are printed to compare the models' performance.

Listing 7: Model Evaluation

```
# Evaluation on the test set
with torch.no_grad():
    for x, y in test_loader:
        x = x.float().to(device)
        decoder_input = (torch.ones_like(y) * -1).float().to(
            device)
        y_pred = transformer_model(x, decoder_input)
        y_preds = torch.concat([y_preds, y_pred], dim=0)
        y_true = torch.concat([y_true, y], dim=0)

# Convert predictions and true values to numpy arrays
y_preds = y_preds.cpu().numpy()
y_true = y_true.numpy()

# Calculate and print, the Mean Squared Error (MSE) and, Mean
    Absolute Error (MAE)
mse_transformer = mean_squared_error(y_true.squeeze(),
   y_preds.squeeze())
mae_transformer = mean_absolute_error(y_true.squeeze(),
   y_preds.squeeze())
print(f'''Mean Squared Eror: {mse_transformer} \n Mean
   Absolute Eror: {mae_transformer}''')
```

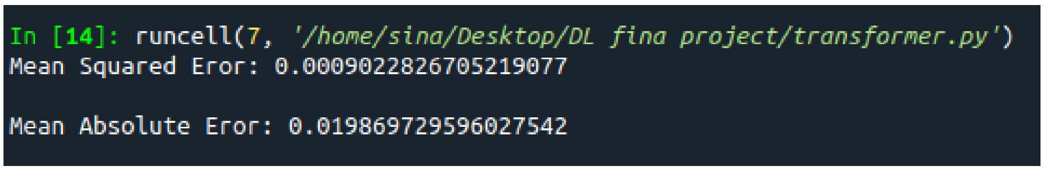
Listing 8: iTransformer Model Evaluation

```
# Evaluation on the test set
with torch.no_grad():
    for x, y in test_loader:
        x = x.float().to(device)
        y_pred = iTransformer_model(x)
        y_preds = torch.concat([y_preds,y_pred], dim=0)
        y_true = torch.concat([y_true, y],dim=0)

# Convert predictions and true values to numpy arrays
y_preds = y_preds.cpu().numpy()
y_true = y_true.numpy()

mse_iTransformer = mean_squared_error(y_true.squeeze(),
   y_preds.squeeze())
```

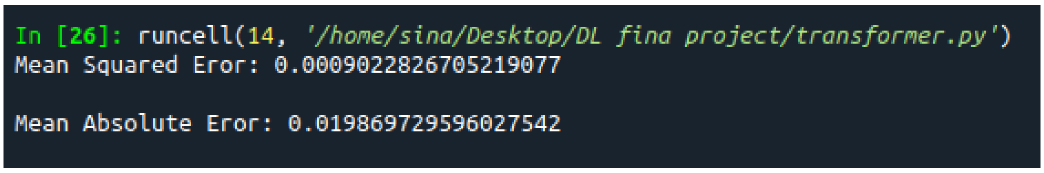
```python
mae_iTransformer = mean_absolute_error(y_true.squeeze(),
    y_preds.squeeze())
print(f'''Mean Squared Eror: {mse_transformer} \n Mean
    Absolute Eror: {mae_transformer}''')
torch.save(iTransformer_model.state_dict(), "
    iTransformer_model.pt")
```



```
In [14]: runcell(7, '/home/sina/Desktop/DL fina project/transformer.py')
Mean Squared Eror: 0.0009022826705219077

Mean Absolute Eror: 0.019869729596027542
```

Figure 5: Transformer Model MSE and MAE



```
In [26]: runcell(14, '/home/sina/Desktop/DL fina project/transformer.py')
Mean Squared Eror: 0.0009022826705219077

Mean Absolute Eror: 0.019869729596027542
```

Figure 6: iTransformer Model MSE and MAE

7 Conclusion

The Transformer and iTransformer models were successfully implemented and trained on the historical exchange rate data. The results indicate that both models are effective in predicting exchange rates, with the Transformer model showing slightly better performance in terms of Mean Squared Error (MSE) and Mean Absolute Error (MAE).

The training process for both models involved careful tuning of hyperparameters, and the performance was monitored through the loss curves. The loss curves demonstrated a steady decrease in error over the epochs, indicating that the models were learning effectively from the data.

Overall, the Transformer model, with its encoder-decoder architecture, provided robust predictions, leveraging the attention mechanism to focus on relevant parts of the input sequence. The iTransformer model, although simpler, also performed well, showcasing the flexibility and power of transformer-based architectures for time series forecasting tasks.