به نام خدا

عنوان:

گزارش پروژه دوم تکلیف چهارم شبکه عصبی

استاد:

دکتر منصوری

دانشجو:

محمدعلی مجتهدسلیمانی-٤٠٣٣٩٠٤٥٠٤

تاریخ:

١٤٠٣/١٠/٢٠

Table of Contents

اضافه کردن کتابخانه‌ها 3

بارگذاری و پیش پردازش داده‌ها 4

ایجاد dataset و DataLoader 6

تعریف مدل RNN 8

آموزش مدل 10

ارزیابی مدل RNN 12

محاسبه MAE و MSE 14

تعریف مدل LSTM 16

آموزش مدل LSTM 17

ارزیابی مدل LSTM 18

گزارش MAE و MSE 18

اضافه کردن کتابخانه‌ها

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
```

از کتابخانه **numpy** برای انجام عملیات هایی عددی با ماتریس ها و آرایه ها استفاده کرده ایم.

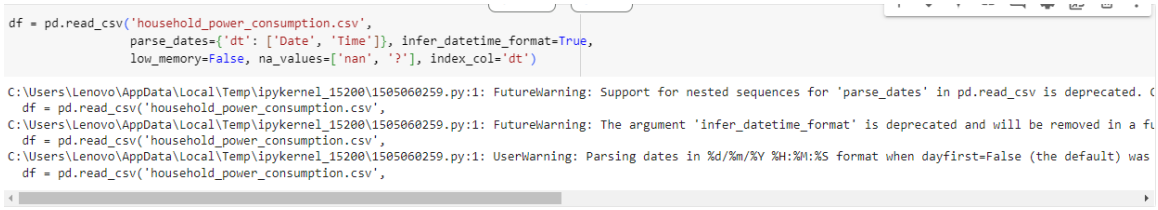
از **pandas** برای انجام عملیات های با **data frame** ها یا داده های جدولی مخصوصا برای تحلیل داده ها استفاده کرده ایم.

با استفاده از **matplotlib** نمودار هایی را که میخواهیم نمایش میدهیم.

کتابخانه **torch** و زیر مجموعه آن برای کار با شبکه های عصبی و مدل های آن مورد استفاده قرار میگیرد همچنین موارد **utils** را با استفاده از همین کتابخانه اضافه کرده ایم به عنوان مثال **dataloader** برای ساختن **dataset** ها و وارد کردن داده های خودمان در هر کدام از **batch**.

از کتاب خانه **sklearn** نیز استفاده کرده ایم. از **MinMaxScaler** برای مقیاس دادن به داده های خودمان برای اینکه مقادیری بین ۰ و ۱ بگیرند. همچنین **MSE** و **MAE** که **absolute** هست را از همین کتابخانه کمک گرفته ایم.

بارگذاری و پیش پردازش داده‌ها



با استفاده از **read_csv** داده های از فایل اکسل به **data frame** پانداز وارد میکنیم.

سپس با استفاده از **parse_dates** دو ستون **Date** و **Time** به یک ستون **datetime** تبدیل میکنیم به نام **dt**.

infer_datetime_format=True این خط کمک میکند پانداز خودکار **format**، **date, time** را بفهمد.

low_memory=False این خط اجازه میدهد حافظه بیشتری برای پیش پردازش استفاده شود و اجازه بدهد پانداز کل فایل را وارد حافظه بکند.

na_values=['nan', '?'] برای برخورد با داده های از دست رفته یا گمشده احتمالی از این خط استفاده میکنیم.

index_col='dt از این ستون به عنوان یک اندیس برای **data frame** استفاده میکنیم.

```
# We check if there are any missing values in the dataset.

df.isnull().sum()
```

```
Global_active_power      25979
Global_reactive_power    25979
Voltage                  25979
Global_intensity         25979
Sub_metering_1           25979
Sub_metering_2           25979
Sub_metering_3           25979
dtype: int64
```

بررسی میکنیم که آیا داده هایی که مقدار آنها را نداریم در مجموعه خودمان وجود دارند یا نه.

```
for col in df.columns:
    df[col] = df[col].fillna(df[col].mean())
```

روی هر ستون **data frame** حرکت میکنیم و هر کدام از **value** هایی که نداریم را با میانگین آن جایگزین میکنیم.

```
df_resampled = df.resample('h').mean()
```

resample فرکانس داده ها به ساعتی تغییر میدهیم و داده ها را یکبار دیگر میکنیم. این کد داده ها را بر اساس بازه های ساعتی گروه بندی میکند و میانگین را برای هر ستون در یک ساعت مشخص محاسبه میکند.

```
scaler = MinMaxScaler()
data_scaled = scaler.fit_transform(df_resampled)
```

در این قسمت یک **object** از **MinMax** میسازیم تا داده ها را مقیاس دهی کنیم بین بازه ۰ تا ۱.

ایجاد dataset و DataLoader

```
class TimeSeriesDataset(Dataset):
    def __init__(self, data, seq_len):
        self.data = data
        self.seq_len = seq_len

    def __len__(self):
        return len(self.data) - self.seq_len

    def __getitem__(self, index):
        x = self.data[index:index + self.seq_len]
        y = self.data[index + self.seq_len, 0]
        return torch.tensor(x).float(), torch.tensor(y).float()
```

با استفاده از **TimeSeriesDataset(Dataset)** یک کلاس برای **dataset** اختصاصی خودمان تعریف میکنیم که از **pytorch** ارث بری میکند.

در تابع **___init___** یا تابع **constructor** کلاس ما است که صرفا مقدار دهی کردیم.

در تابع **___len___(self)** تعداد کل نمونه ها را بر میگردانیم.

len(self.data) - self.seq_len این کد برای این است که یک نمونه را بسازیم زیرا به **seq_len** در گام های زمانی نیاز داریم.

__getitem__(self, index) با استفاده از این کد از دیتاست خودمان نمونه برداری میکنیم.

x = self.data[index:index + self.seq_len] یک توالی به طول seq_lenاز داده‌ها به عنوان ورودی (x) استخراج می‌کند.

y = self.data[index + self.seq_len, 0] مقدار مرحله زمانی بعدی (پس از دنباله) را به عنوان هدف (y) استخراج می‌کند. فرض می‌کنیم می‌خواهید ویژگی اول (اندیس ۰) که 'Global_active_power' است را پیش‌بینی کنید.

torch.tensor(y).float() دنباله ورودی و مقدار هدف را به صورت تنسورهای PyTorchبرمی‌گرداند (و آنها را به اعداد اعشاری تبدیل می‌کند).

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

train_size = int(len(data_scaled) * 0.8) اندازه مجموعه آموزشی (۸۰ درصد داده‌ها) را بدست میاورد.

train_data = data_scaled[:train_size]داده‌های آموزشی را با گرفتن اولین train_size ایجاد می‌کند.

test_data = data_scaled[train_size:]داده‌های تست را با گرفتن سطرهای باقی‌مانده پس از train_size ایجاد می‌کند.

```
seq_len = 24  # Sequence length

train_dataset = TimeSeriesDataset(train_data, seq_len)
test_dataset = TimeSeriesDataset(test_data, seq_len)

batch_size = 64

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

train_dataset = TimeSeriesDataset یک object برای داده های آموزشی ایجاد میکند.

test_dataset = TimeSeriesDataset یک object برای داده های تست ایجاد میکند.

train_loader=DataLoader(): یک DataLoader برای داده‌های آموزشی ایجاد می‌کند. با shuffle=True داده‌های آموزشی در هر دوره تصادفی یا درهم یا مخلوط میشوند.

test_loader = DataLoader(): یک DataLoader برای داده‌های آزمایشی ایجاد می‌کند. استفاده از shuffle=False برای مجموعه آزمایشی معمول است.

تعریف مدل RNN

```
class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(RNNModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out
```

کلاس RNNModel از nn.module ارث بری میکند و کلاس مدل را ساختیم.
__init__ تابع constructor کلاس ما است.

self.hidden_size = hidden_size: تعداد واحدهای مخفی در لایه‌ی RNN را ذخیره می‌کند.

self.num_layers = num_layers: تعداد لایه‌های RNN را ذخیره می‌کند.

self.rnn = nn.RNN(): لایه‌ی RNN را ایجاد می‌کند.

self.fc = nn.Linear(): یک لایه‌ی کاملاً متصل (خطی) ایجاد می‌کند تا خروجی نهایی را تولید کند.

h0 = torch.zeros(): یک وضعیت اولیه برای واحد های مخفی ایجاد میکند.

out, _ = self.rnn(): مسیر forward را از طریق لایه‌ی RNN انجام می‌دهد . شامل وضعیت‌های واحدهای مخفی برای هر مرحله‌ی زمانی در دنباله است. _ برای نادیده گرفتن وضعیت واحد مخفی نهایی استفاده می‌شود (فقط خروجی مرحله‌ی زمانی آخر مورد نیاز است).

out = self.fc(): خروجی مرحله زمانی آخر را میگیرد و از لایه کاملا متصل عبور میدهد تا پیشبینی نهایی بدست بیاد.

در نهایت پیشبینی را برمیگردانیم.

آموزش مدل

```
input_size = 7  # Number of features
hidden_size = 64
num_layers = 2
output_size = 1
learning_rate = 0.001
num_epochs = 10

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = RNNModel(input_size, hidden_size, num_layers, output_size).to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

در این قسمت صرفا پارامتر ها را مشخص کرده ایم تعداد ویژگی ها و تعداد واحد های مخفی و تعداد لایه ها و اندازه خروجی و نرخ یادگیری و تعداد **epoch** ها را مشخص کرده ایم. همچنین از **optimizer**، **Adam** استفاده کرده ایم.

```
for epoch in range(num_epochs):
    for batch_x, batch_y in train_loader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)

        outputs = model(batch_x)
        loss = criterion(outputs, batch_y.unsqueeze(1))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')
```

```
Epoch [1/10], Loss: 0.0068
Epoch [2/10], Loss: 0.0081
Epoch [3/10], Loss: 0.0083
Epoch [4/10], Loss: 0.0072
Epoch [5/10], Loss: 0.0076
Epoch [6/10], Loss: 0.0059
Epoch [7/10], Loss: 0.0085
Epoch [8/10], Loss: 0.0091
Epoch [9/10], Loss: 0.0052
Epoch [10/10], Loss: 0.0074
```

for epoch in range(num_epochs): این حلقه به تعداد مشخص‌شده‌ی دوره‌ها **(num_epochs)** تکرار می‌شود.

for batch_x, batch_y in train_loader: این حلقه روی دسته‌های داده**(batch)**، تکرار می‌شود. در هر تکرار، یک دسته را به **batch** (ویژگی‌های ورودی) و **batch** (مقادیر هدف) تقسیم می‌کند.

train_loaderیک **object** از **DataLoader** که از کتابخانه **PyTorch**است برای وارد کردن داده ها درون دسته ها و درهم کردن داده استفاده میشود.

batch_x = batch_x.to(device): دسته‌ی ورودی **(batch_x)** را به دستگاه مشخص‌شده منتقل می‌کند ، که توسط متغیر **device**تعیین می‌شود.

outputs = model(batch_x): این بخش، **forward pass**مدل را انجام می‌دهد. این بخش، دسته‌ی ورودی **(batch_x)** را می‌گیرد و آن را از طریق مدل عبور می‌دهد تا پیش‌بینی‌ها **(outputs)** را تولید کند.

loss = criterion(): این خط **loss** را محاسبه می کند، که تفاوت بین پیش بینی های مدل **(outputs)** و مقادیر واقعی هدف **(batch_y)** را اندازه گیری می کند.

batch_y.unsqueeze(1): این یک بعد به **batch_y** اضافه می کند. این اغلب ضروری است زیرا خروجی مدل ممکن است شکلی متفاوت از **y** داشته باشد. در این حالت، یک بعد به اندازه ۱ در موقعیت ۱ اضافه می کنیم. یعنی مثلا اگر به شکل **(64,)** باشد به **(64,1)** تبدیل میشود.

در خط بعدی **optimizer.step()** را استفاده کرده ایم برای محاسبه گرادیان.

ارزیابی مدل RNN

در صفحه بعد*

```
model.eval()
predictions = []
with torch.no_grad():
    for batch_x, batch_y in test_loader:
        batch_x = batch_x.to(device)
        outputs = model(batch_x)
        predictions.append(outputs.cpu().numpy())

predictions = np.concatenate(predictions, axis=0)
```

model.eval(): مدل را در حالت ارزیابی قرار می‌دهد. این کار مواردی مانند **dropout** و نرمال‌سازی دسته‌ای را که فقط در طول آموزش استفاده می‌شوند، خاموش می‌کند.

predictions = []: یک لیست خالی برای ذخیره پیش‌بینی‌ها ایجاد می‌کند.

with torch.no_grad(): محاسبات گرادیان را در طول ارزیابی غیرفعال می‌کند (این کار باعث صرفه‌جویی در حافظه و افزایش سرعت محاسبات می‌شود).

for batch_x, batch_y in test_loader: روی دسته‌های داده در **test_loader** تکرار می‌شود.

batch_x = batch_x.to(device): دسته‌ی ورودی را به دستگاه مشخص‌شده منتقل می‌کند. در اینجا **GPU** بنده است.

outputs = model(batch_x): خروجی پیشبینی را در **output** قرار میدهد.

predictions.append(): پیش‌بینی‌ها تبدیل‌شده به آرایه‌های **(NumPy)** را به لیست **predictions** اضافه می‌کند.

predictions = np.concatenate(): پیش‌بینی‌های همه دسته‌ها را به یک آرایه **NumPy** واحد متصل می‌کند.

```
dummy = np.zeros((len(predictions), data_scaled.shape[1]))

dummy[:, 0] = predictions.ravel()

predictions_original_scale = scaler.inverse_transform(dummy)[:, 0]
dummy_test = np.zeros((len(test_data) - seq_len, data_scaled.shape[1]))
dummy_test[:, 0] = test_data[seq_len:, 0]
true_values_original_scale = scaler.inverse_transform(dummy_test)[:, 0]
```

این قسمت برای معکوس کردن عملیات نرمال‌سازی که قبلاً روی داده‌ها اعمال شده بود، طراحی شده است. این کد پیش‌بینی‌های مدل (که در فضای مقیاس‌بندی‌شده هستند) و مقادیر واقعی متناظر از مجموعه تستی (همچنین در فضای مقیاس‌بندی‌شده) را می‌گیرد و آن‌ها را به مقیاس اصلی خود تبدیل می‌کند.

scaler.inverse_transform(du): این تبدیل مقیاس‌بندی معکوس را به **dummy_test** اعمال می‌کند تا مقادیر واقعی را به مقیاس اصلی خود بازگرداند.

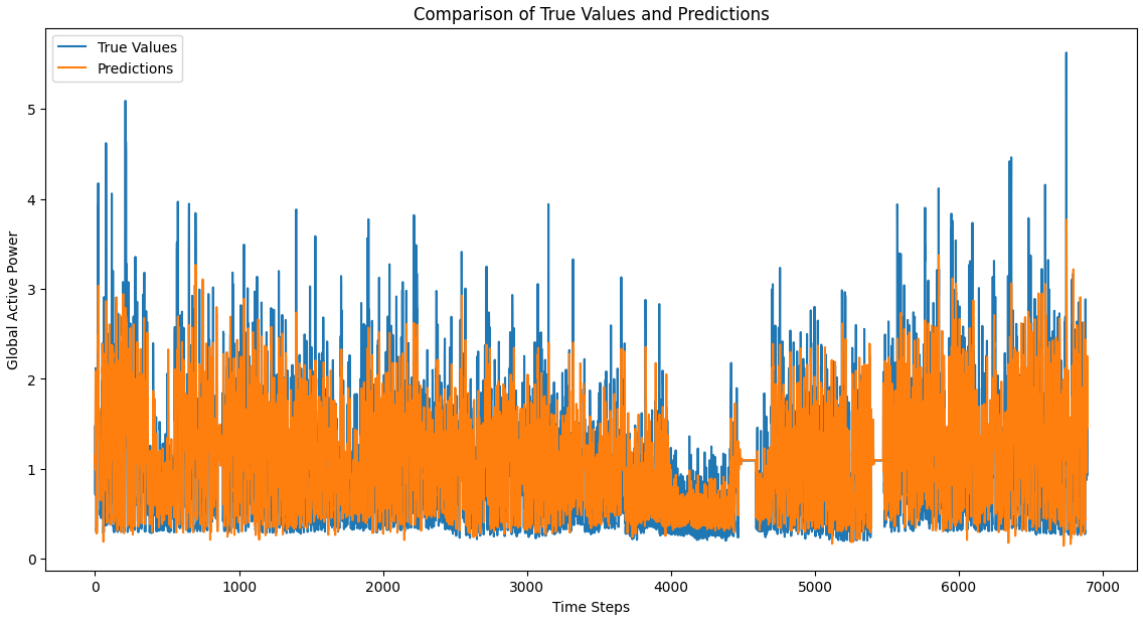
[:, 0]: اولین ستون را انتخاب می‌کند که شامل مقادیر واقعی تبدیل معکوس‌شده است. این مقادیر در متغیر **true_values_original_scale** ذخیره می‌شوند.

محاسبه MAE و MSE

```
mae = mean_absolute_error(true_values_original_scale, predictions_original_scale)
mse = mean_squared_error(true_values_original_scale, predictions_original_scale)

print(f'MAE: {mae:.4f}')
print(f'MSE: {mse:.4f}')
```

```
MAE: 0.3532
MSE: 0.2403
```



در این دیاگرام یک مقایسه بین مقادیر واقعی و مقادیر پیشبینی شده انجام داده ایم. همچنین در تیکه کد قبلی **Mean Absolute Error** و **Mean Squared Error** را گزارش داده ایم بین مقادیر واقعی و مقادیر پیشبینی شده.

برای بررسی بیشتر **LSTM** را نیز پیاده سازی کرده‌ایم.

تعریف مدل LSTM

```
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        return out
```

مدل کلاس **LSTM** را تعریف کرده‌ایم. همچنین تابع **contructor** این کلاس را هم پیاده سازی کردیم که یک مقدار دهی اولیه به متغیر ها انجام شده است.

تابع **forward pass** مدل **LSTM** در ادامه تعریف شده است. در **H0** واحد های مخفی را مقدار اولیه کرده ایم. **C0** در **cell state**، که برای **LSTM** ها است را مقدار دهی اولیه کرده ایم. در ادامه تابع را از لایه های **LSTM** عبور میدهیم. در نهایت خروجی را از آخرین گام زمانی میگیریم و به لایه تمام متصل میدهیم تا پیشبینی تولید شود.

آموزش مدل LSTM

```python
model_lstm = LSTMModel(input_size, hidden_size, num_layers, output_size).to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model_lstm.parameters(), lr=learning_rate)

for epoch in range(num_epochs):
    for batch_x, batch_y in train_loader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)

        outputs = model_lstm(batch_x)
        loss = criterion(outputs, batch_y.unsqueeze(1))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')
```

```
Epoch [1/10], Loss: 0.0065
Epoch [2/10], Loss: 0.0077
Epoch [3/10], Loss: 0.0110
Epoch [4/10], Loss: 0.0117
Epoch [5/10], Loss: 0.0100
Epoch [6/10], Loss: 0.0077
Epoch [7/10], Loss: 0.0116
Epoch [8/10], Loss: 0.0065
Epoch [9/10], Loss: 0.0077
Epoch [10/10], Loss: 0.0082
```

مانند RNN است این قسمت.

ارزیابی مدل LSTM

```
model_lstm.eval()
predictions_lstm = []
with torch.no_grad():
    for batch_x, batch_y in test_loader:
        batch_x = batch_x.to(device)
        outputs = model_lstm(batch_x)
        predictions_lstm.append(outputs.cpu().numpy())

predictions_lstm = np.concatenate(predictions_lstm, axis=0)

dummy_lstm = np.zeros((len(predictions_lstm), data_scaled.shape[1]))
dummy_lstm[:, 0] = predictions_lstm.ravel()
predictions_lstm_original_scale = scaler.inverse_transform(dummy_lstm)[:, 0]
```

مانند RNN مدل را آماده ارزیابی کردیم

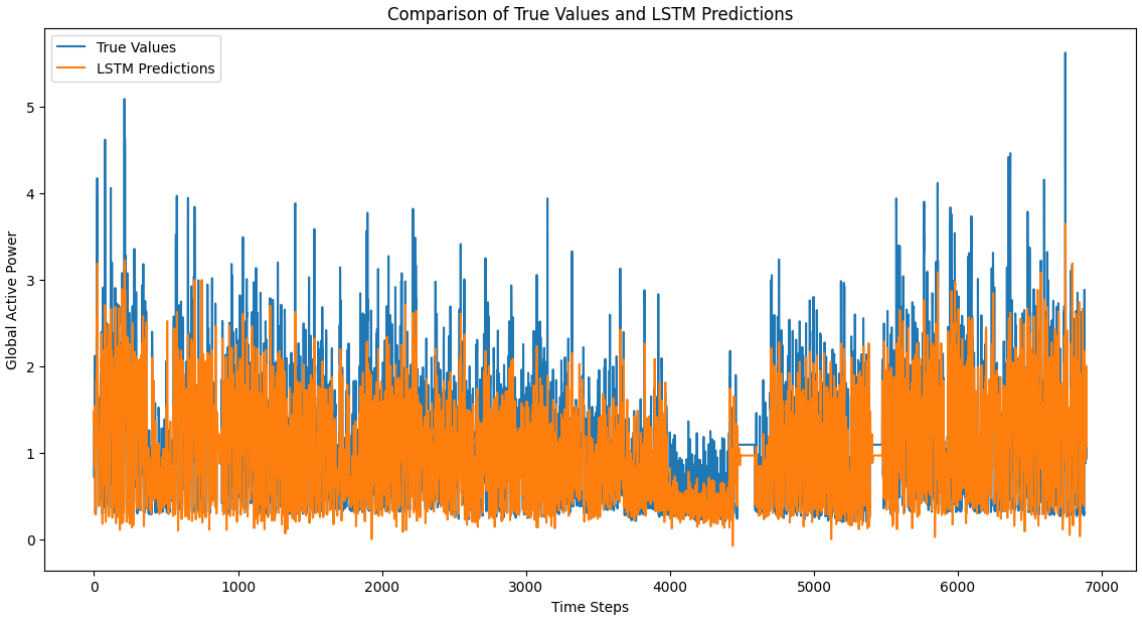
گزارش MAE و MSE

```
print(f'LSTM MAE: {mae_lstm:.4f}')
print(f'LSTM MSE: {mse_lstm:.4f}')

plt.figure(figsize=(14, 7))
plt.plot(true_values_original_scale, label='True Values')
plt.plot(predictions_lstm_original_scale, label='LSTM Predictions')
plt.title('Comparison of True Values and LSTM Predictions')
plt.xlabel('Time Steps')
plt.ylabel('Global Active Power')
plt.legend()
plt.show()
```

```
LSTM MAE: 0.3329
LSTM MSE: 0.2331
```

نمودار در صفحه بعد قرار دارد.



در این دیاگرام یک مقایسه بین مقادیر واقعی و مقادیر پیشبینی شده انجام داده ایم. همچنین در تیکه کد قبلی **Mean Absolute Error** و **Mean Squared Error** را گزارش داده ایم بین مقادیر واقعی و مقادیر پیشبینی شده.