**Deep Context-Attentive Transformer Transfer Learning for Financial Forecasting**

Ling Feng1, Ananta Sinchai1

1 School of Integrated Innovative Technology, King Mongkut’s Institute of Technology, Ladkrabang, Bangkok, Thailand

Corresponding Author:

Ananta Sinchai2

Chalong Krung Road, Ladkrabang, Bangkok, 10520, Thailand

Email address: ananta.sin@kmitl.ac.th

Contributing Author:

Ling Feng1

Chalong Krung Road, Ladkrabang, Bangkok, 10520, Thailand

Email address: 64612007@kmitl.ac.th

**SUPPLEMENTARY INFORMATION**

This supplementary information document provides a comprehensive guide for reproducing the proposed Deep Context-Attentive Transformer Transfer Learning model for financial forecasting. Reproducing includes detailed instructions on setting up the environment, required libraries, dataset structure, and code organization.

Key sections:

* **Reproduction Procedures:** Instructions on setting up Python-based environments and dependencies for running the model.
* **Organizational Structure:** Explanation of folder and file hierarchies within the project, ensuring reproducibility.
* **Code Overview:** Description of the core components, including data handling scripts (data\_loader.py, data\_factory.py), experimental setup (exp\_basic.py, exp\_main.py), and utilities (masking.py, metrics.py, tools.py).
* **Dataset Structure:** Guidelines for formatting data input, particularly financial time-series, including Open, High, Low, and Closing (OT) prices.
* **Model Implementation:** Details of the CNNCorrelationBasedTransformer model, highlighting its CNN correlation-based attention mechanism for time-series forecasting regarding context-attentive Transformer.

This document serves as a technical reference for researchers and developers aiming to replicate or extend the study.

**1. REPRODUCTION PROCEDURES**

In this section, we describe how to reproduce the proposed method using the modified code based on a Python interpreter. The original code developed by Haixu Wu, available on [GitHub](https://github.com/thuml/Autoformer), has been adapted to serve the requirements of our experiments.

All modifications to the original code were made in accordance with the MIT License below:

MIT License

Copyright (c) 2021 THUML @ Tsinghua University

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.1 Prerequisite requirements

This subsection outlines the prerequisites for setting up a development environment for machine learning projects using PyTorch. It details the required Python version and necessary libraries, ensuring you can successfully run your code and experiment with the proposed model. Follow the steps to configure the environment effectively.

1.1.1 Python: Ensure Python 3.6 or higher is installed.

1.1.2 Jupyter: Ensure JupyterLab 4.2 or higher is installed.

1.1.3 Install Required Libraries:

pip install torch torchvision torchaudio

pip install numpy pandas matplotlib scikit-learn argparse

1.1.4 Custom Utilities: Ensure that any custom utility modules (utils/timefeatures.py, utils/tools.py, etc.) are available in your project directory.

1.2 Organizational structure of folders and files

 To reproduce the experiment, a specific set of folders and files organized in a hierarchical data structure is required (see Figure S1). Developers—including users, scholars, or researchers are suggested to make sure that these folders and files are produced properly. Upon running Execute.ipynb, additional folders and files will be automatically created in the MyProject root directory. These automatically generated items will include at least “checkpoints,” “test\_results,” and “result.txt.” The “checkpoints” folder is specifically designed to collect a series of parameters, storing the obtained parameter settings in files with a “.pth” extension for each experiment. This setup supports transfer learning by allowing pre-trained models to be fine-tuned on new tasks, enhancing their performance and adaptability. Additionally, other folders and files may be generated by Jupyter that are not explicitly mentioned here.

/MyProject

├── /data\_provider

│ ├── \_\_init\_\_.py

│ ├── data\_factory.py

│ └── data\_loader.py

├── /data

│ ├── dataset\_instrument01.csv

│ ├── …

│ ├── …

│ ├── …

│ └── dataset\_instrumentNN.csv

├── /exp

│ ├── \_\_init\_\_.py

│ ├── exp\_basic.py

│ └── exp\_main.py

├── /utils

│ ├── \_\_init\_\_.py

│ ├── masking.py

│ ├── metrics.py

│ ├── timefeatures.py

│ └── tools.py

├── /results

├── CNNCorrelationBasedTransformer.py

└── Execute.ipynb

**Figure S1** Hierarchical structure of the MyProject directory.

This diagram depicts the hierarchical organization of the MyProject directory. It includes essential subfolders like /data\_provider, /data, /exp, and /utils, each containing necessary scripts and datasets. Following the execution of Execute.ipynb, additional folders such as “checkpoints,” “test\_results,” and “result.txt” are automatically generated in the root directory, along with other files potentially created by Jupyter. This structure facilitates efficient data management and reproducibility of the experiment.

1.3 Overview of code structure, functionality, and file preparation

 In this section, we provide a detailed overview of the code contained within each file of the MyProject directory. This organized structure, illustrated in Figure S1, facilitates efficient data management and experimentation. Each file and folder are designed to serve a specific purpose within this work, from data handling to experimental execution.

 The /data\_provider folder contains scripts essential for loading and preparing datasets, while the /data folder houses the various datasets utilized in our experiments. The /exp directory is dedicated to the core experimental code, including the main execution script, Execute.ipynb. The /utils folder comprises utility functions that enhance the overall functionality of the proposed method.

 A key component of this work is the CNNCorrelationBasedTransformer.py file, which includes significant classes that play a crucial role in the experimental framework. By breaking down the code for each file, we aim to clarify the role and functionality of every component, ensuring that users can easily navigate and reproduce the methods outlined in the following parts of this study.

1.3.1 Dataset structure in a CSV file format

To create a dataset consisting of five attributes to be consistent to the use in this work, which includes financial metrics, follow these steps to collect and structure your data:

**Date**: Ensure that dates are in a consistent format (e.g., YYYY-MM-DD). These dates typically represent trading days, as weekends and holidays might not have data. Such data can be obtained from historical financial data providers, stock market APIs, or other financial data sources.

**Open, High, Low, Closing (OT)**: These values represent the daily trading range:

**Open**: The opening price at the start of the trading day

**High**: The highest price recorded during the trading day

**Low**: The lowest price recorded during the trading day

**Closing (OT)**: The price at which the instrument closed at the end of the trading day

Each row in the CSV file should represent one trading day, maintaining consistency in format and completeness across all entries. Collect data over a range that matches your intended analysis, whether it is daily, weekly, or at other intervals. Please keep in mind that this is not limited to financial time series data; other types of time series data can also be used. Additionally, ensure that the last column (OT) represents the objective target feature to predict.

1.3.1.1 Example CSV Structure

An example CSV structure is shown in Figure S2. This illustration captures how each of the metrics is arranged within the CSV. Ensure that the created file follows this example for uniformity and compatibility with existing datasets or analysis tools.

To maintain data quality, verify each entry against reliable sources. For financial datasets, cross-referencing with multiple data providers can help maintain accuracy, especially for historical values. Proper formatting, as shown in Figure S2, also ensures that your dataset remains compatible with statistical software and machine learning algorithms.



**Figure S2** Example of a CSV dataset structure for time series forecasting in finance, showing columns for date, Open, High, Low, and closing price (OT). Each row represents a trading day, capturing essential daily price metrics, with OT as the closing value. This format supports consistent data input for forecasting and financial modeling.

1.3.2 Generation of \_\_init\_\_.py

 The \_\_init\_\_.py file is a crucial component in Python package creation, indicating to the interpreter that a directory should be treated as a package. When a new package is generated, many development environments and frameworks automatically create a blank \_\_init\_\_.py file. This blank file serves as a placeholder, simplifying the package structure without requiring any additional code. Developers—including users, scholars, or researchers—must include this file if creating a package manually to ensure proper functionality and avoid import issues, even though it does not contain any code by default.

1.3.3 Generation of data\_factory.py

This code (see below) implements a data loading system primarily designed for time series datasets. The core functionality revolves around the data\_provider function, which creates data loaders for different types of financial market data (stocks, cryptocurrencies, commodities, etc.).

The code uses a dictionary data\_dict to map various dataset identifiers (like 'btc' for Bitcoin, 'sp' for S&P 500, etc.) to a custom dataset class called Dataset\_Custom. The data\_provider function accepts arguments for configuration and a flag parameter ('test', 'pred', or other) that determines how the data will be loaded. For testing, it disables shuffling and keeps all batches. For prediction, it sets batch size to 1. For training (default), it enables shuffling and drops incomplete final batches. Developers—including users, scholars, or researchers may copy the given code and paste it in a file named data\_factory.py and save it.

#-----------------------------------------------------------data\_factory.py--------------------------------------#

from data\_provider.data\_loader import Dataset\_Custom, Dataset\_Pred

from torch.utils.data import DataLoader

data\_dict = {

 'custom': Dataset\_Custom,

 'dj': Dataset\_Custom,

 'sp': Dataset\_Custom,

 'nq': Dataset\_Custom,

 'nk': Dataset\_Custom,

 'dax': Dataset\_Custom,

 'sp1': Dataset\_Custom,

 'nk1': Dataset\_Custom,

 'dax1': Dataset\_Custom,

 'gold': Dataset\_Custom,

 'gold1': Dataset\_Custom,

 'btc': Dataset\_Custom,

 'btc1': Dataset\_Custom,

 'eth': Dataset\_Custom,

 'eth1': Dataset\_Custom,

 'eth2': Dataset\_Custom,

 'set': Dataset\_Custom,

 'set1': Dataset\_Custom,

 'dj2': Dataset\_Custom,

 'oil': Dataset\_Custom,

 'oil1': Dataset\_Custom,

 'ibo': Dataset\_Custom,

 'ibo1': Dataset\_Custom,

 'sensex': Dataset\_Custom,

 'sensex1': Dataset\_Custom,

 'ltc': Dataset\_Custom,

 'ltc1': Dataset\_Custom,

 'nq1': Dataset\_Custom,

}

def data\_provider(args, flag):

 Data = data\_dict[args.data]

 timeenc = 0 if args.embed != 'timeF' else 1

 if flag == 'test':

 shuffle\_flag = False

 drop\_last = False

 batch\_size = args.batch\_size

 freq = args.freq

 elif flag == 'pred':

 shuffle\_flag = False

 drop\_last = False

 batch\_size = 1

 freq = args.freq

 Data = Dataset\_Pred

 else:

 shuffle\_flag = True

 drop\_last = True

 batch\_size = args.batch\_size

 freq = args.freq

 data\_set = Data(

 root\_path=args.root\_path,

 data\_path=args.data\_path,

 flag=flag,

 size=[args.seq\_len, args.label\_len, args.pred\_len],

 features=args.features,

 target=args.target,

 timeenc=timeenc,

 freq=freq

 )

 print(flag, len(data\_set))

 data\_loader = DataLoader(

 data\_set,

 batch\_size=batch\_size,

 shuffle=shuffle\_flag,

 num\_workers=args.num\_workers,

 drop\_last=drop\_last)

 return data\_set, data\_loader

#----------------------------------------------------------data\_factory.py---------------------------------------#

1.3.4 Generation of data\_loader.py

This implementation provides a PyTorch-based data loading system specifically designed for time series forecasting. The system consists of two main components:

1. Dataset\_Custom: The primary dataset class for training/testing, which handles:
	* Data splitting (70% train, 20% test, 10% validation)
	* Time feature encoding (date features or custom time encodings)
	* Data normalization using StandardScaler
	* Flexible sequence length configuration for input, label, and prediction windows
	* Support for both single-feature (S) and multi-feature (M/MS) time series
2. Dataset\_Pred: A specialized version for prediction that:
	* Uses the last available sequence for forecasting.
	* Generates future timestamp features for prediction period.
	* Supports inverse transformation of scaled data.

The data factory (data\_factory.py) acts as a wrapper that:

* Maps dataset names to appropriate dataset classes.
* Configures DataLoader with correct parameters for training, testing, or prediction.
* Handles batch size, shuffling, and worker settings based on the usage mode.

To use this system:

1. Prepare your time series data in CSV format with a 'date' column and target variable.
2. Configure parameters (sequence length, prediction length, features type).
3. Use data\_provider() to get appropriate data loaders for training/testing/prediction.

The system is particularly optimized for financial time series like stocks, cryptocurrencies, and commodities, as evidenced by the dataset mappings in data\_factory.py. Developers—including users, scholars, or researchers may copy the given code and paste it in a file named data\_loader.py and save it.

#----------------------------------------------------------data\_loader.py----------------------------------------#

import os

import numpy as np

import pandas as pd

import os

import torch

from torch.utils.data import Dataset, DataLoader

from sklearn.preprocessing import StandardScaler

from utils.timefeatures import time\_features

import warnings

warnings.filterwarnings('ignore')

class Dataset\_Custom(Dataset):

 def \_\_init\_\_(self, root\_path, flag='train', size=None,

 features='MS', data\_path='ibo.csv',

 target='OT', scale=True, timeenc=0, freq='d'):

 # size [seq\_len, label\_len, pred\_len]

 # info

 if size == None:

 self.seq\_len = 24 \* 4 \* 4

 self.label\_len = 24 \* 4

 self.pred\_len = 24 \* 4

 else:

 self.seq\_len = size[0]

 self.label\_len = size[1]

 self.pred\_len = size[2]

 # init

 assert flag in ['train', 'test', 'val']

 type\_map = {'train': 0, 'val': 1, 'test': 2}

 self.set\_type = type\_map[flag]

 self.features = features

 self.target = target

 self.scale = scale

 self.timeenc = timeenc

 self.freq = freq

 self.root\_path = root\_path

 self.data\_path = data\_path

 self.\_\_read\_data\_\_()

 def \_\_read\_data\_\_(self):

 self.scaler = StandardScaler()

 df\_raw = pd.read\_csv(os.path.join(self.root\_path,

 self.data\_path))

 '''

 df\_raw.columns: ['date', ...(other features), target feature]

 '''

 cols = list(df\_raw.columns)

 cols.remove(self.target)

 cols.remove('date')

 df\_raw = df\_raw[['date'] + cols + [self.target]]

 # print(cols)

 num\_train = int(len(df\_raw) \* 0.7)

 num\_test = int(len(df\_raw) \* 0.2)

 num\_vali = len(df\_raw) - num\_train - num\_test

 border1s = [0, num\_train - self.seq\_len, len(df\_raw) - num\_test - self.seq\_len]

 border2s = [num\_train, num\_train + num\_vali, len(df\_raw)]

 border1 = border1s[self.set\_type]

 border2 = border2s[self.set\_type]

 if self.features == 'M' or self.features == 'MS':

 cols\_data = df\_raw.columns[1:]

 df\_data = df\_raw[cols\_data]

 elif self.features == 'S':

 df\_data = df\_raw[[self.target]]

 if self.scale:

 train\_data = df\_data[border1s[0]:border2s[0]]

 self.scaler.fit(train\_data.values)

 data = self.scaler.transform(df\_data.values)

 else:

 data = df\_data.values

 df\_stamp = df\_raw[['date']][border1:border2]

 df\_stamp['date'] = pd.to\_datetime(df\_stamp.date)

 if self.timeenc == 0:

 df\_stamp['month'] = df\_stamp.date.apply(lambda row: row.month, 1)

 df\_stamp['day'] = df\_stamp.date.apply(lambda row: row.day, 1)

 df\_stamp['weekday'] = df\_stamp.date.apply(lambda row: row.weekday(), 1)

 df\_stamp['hour'] = df\_stamp.date.apply(lambda row: row.hour, 1)

 data\_stamp = df\_stamp.drop(['date'], 1).values

 elif self.timeenc == 1:

 data\_stamp = time\_features(pd.to\_datetime(df\_stamp['date'].values), freq=self.freq)

 data\_stamp = data\_stamp.transpose(1, 0)

 self.data\_x = data[border1:border2]

 self.data\_y = data[border1:border2]

 self.data\_stamp = data\_stamp

 def \_\_getitem\_\_(self, index):

 s\_begin = index

 s\_end = s\_begin + self.seq\_len

 r\_begin = s\_end - self.label\_len

 r\_end = r\_begin + self.label\_len + self.pred\_len

 seq\_x = self.data\_x[s\_begin:s\_end]

 seq\_y = self.data\_y[r\_begin:r\_end]

 seq\_x\_mark = self.data\_stamp[s\_begin:s\_end]

 seq\_y\_mark = self.data\_stamp[r\_begin:r\_end]

 return seq\_x, seq\_y, seq\_x\_mark, seq\_y\_mark

 def \_\_len\_\_(self):

 return len(self.data\_x) - self.seq\_len - self.pred\_len + 1

 def inverse\_transform(self, data):

 return self.scaler.inverse\_transform(data)

class Dataset\_Pred(Dataset):

 def \_\_init\_\_(self, root\_path, flag='pred', size=None,

 features='S', data\_path='ETTh1.csv',

 target='OT', scale=True, inverse=False, timeenc=0, freq='15min', cols=None):

 # size [seq\_len, label\_len, pred\_len]

 # info

 if size == None:

 self.seq\_len = 24 \* 4 \* 4

 self.label\_len = 24 \* 4

 self.pred\_len = 24 \* 4

 else:

 self.seq\_len = size[0]

 self.label\_len = size[1]

 self.pred\_len = size[2]

 # init

 assert flag in ['pred']

 self.features = features

 self.target = target

 self.scale = scale

 self.inverse = inverse

 self.timeenc = timeenc

 self.freq = freq

 self.cols = cols

 self.root\_path = root\_path

 self.data\_path = data\_path

 self.\_\_read\_data\_\_()

 def \_\_read\_data\_\_(self):

 self.scaler = StandardScaler()

 df\_raw = pd.read\_csv(os.path.join(self.root\_path,

 self.data\_path))

 '''

 df\_raw.columns: ['date', ...(other features), target feature]

 '''

 if self.cols:

 cols = self.cols.copy()

 cols.remove(self.target)

 else:

 cols = list(df\_raw.columns)

 cols.remove(self.target)

 cols.remove('date')

 df\_raw = df\_raw[['date'] + cols + [self.target]]

 border1 = len(df\_raw) - self.seq\_len

 border2 = len(df\_raw)

 if self.features == 'M' or self.features == 'MS':

 cols\_data = df\_raw.columns[1:]

 df\_data = df\_raw[cols\_data]

 elif self.features == 'S':

 df\_data = df\_raw[[self.target]]

 if self.scale:

 self.scaler.fit(df\_data.values)

 data = self.scaler.transform(df\_data.values)

 else:

 data = df\_data.values

 tmp\_stamp = df\_raw[['date']][border1:border2]

 tmp\_stamp['date'] = pd.to\_datetime(tmp\_stamp.date)

 pred\_dates = pd.date\_range(tmp\_stamp.date.values[-1], periods=self.pred\_len + 1, freq=self.freq)

 df\_stamp = pd.DataFrame(columns=['date'])

 df\_stamp.date = list(tmp\_stamp.date.values) + list(pred\_dates[1:])

 if self.timeenc == 0:

 df\_stamp['month'] = df\_stamp.date.apply(lambda row: row.month, 1)

 df\_stamp['day'] = df\_stamp.date.apply(lambda row: row.day, 1)

 df\_stamp['weekday'] = df\_stamp.date.apply(lambda row: row.weekday(), 1)

 df\_stamp['hour'] = df\_stamp.date.apply(lambda row: row.hour, 1)

 df\_stamp['minute'] = df\_stamp.date.apply(lambda row: row.minute, 1)

 df\_stamp['minute'] = df\_stamp.minute.map(lambda x: x // 15)

 data\_stamp = df\_stamp.drop(['date'], 1).values

 elif self.timeenc == 1:

 data\_stamp = time\_features(pd.to\_datetime(df\_stamp['date'].values), freq=self.freq)

 data\_stamp = data\_stamp.transpose(1, 0)

 self.data\_x = data[border1:border2]

 if self.inverse:

 self.data\_y = df\_data.values[border1:border2]

 else:

 self.data\_y = data[border1:border2]

 self.data\_stamp = data\_stamp

 def \_\_getitem\_\_(self, index):

 s\_begin = index

 s\_end = s\_begin + self.seq\_len

 r\_begin = s\_end - self.label\_len

 r\_end = r\_begin + self.label\_len + self.pred\_len

 seq\_x = self.data\_x[s\_begin:s\_end]

 if self.inverse:

 seq\_y = self.data\_x[r\_begin:r\_begin + self.label\_len]

 else:

 seq\_y = self.data\_y[r\_begin:r\_begin + self.label\_len]

 seq\_x\_mark = self.data\_stamp[s\_begin:s\_end]

 seq\_y\_mark = self.data\_stamp[r\_begin:r\_end]

 return seq\_x, seq\_y, seq\_x\_mark, seq\_y\_mark

 def \_\_len\_\_(self):

 return len(self.data\_x) - self.seq\_len + 1

 def inverse\_transform(self, data):

 return self.scaler.inverse\_transform(data)

#----------------------------------------------------------data\_loader.py----------------------------------------#

1.3.5 Generation of exp\_basic.py

The code implements a basic experimental framework for time series forecasting, with an emphasis on financial data. At its core, Exp\_Basic serves as an abstract base class that provides fundamental experiment management functionality:

* Device management (CPU/GPU selection)
* Model initialization
* Template methods for training, testing, and validation

Developers—including users, scholars, or researchers may copy the given code and paste it in a file named exp\_basic.py and save it. This base class is designed to be extended by specific experiment implementations.

The data loading system consists of:

1. A data factory (data\_factory.py) that manages dataset creation and loading configurations.
2. Custom dataset classes (data\_loader.py) that handle:
	* Data preprocessing and normalization
	* Time feature engineering
	* Sequence preparation for both training and prediction
	* Train/validation/test splits (70/10/20)

This system is particularly suited for financial time series analysis, supporting various markets (stocks, crypto, commodities) with flexible input features and prediction windows. The modular design allows for easy extension to new datasets while maintaining consistent data handling across experiments.

To use this framework, users should:

1. Set up the appropriate directory structure.
2. Place their time series data in CSV format.
3. Extend the Exp\_Basic class for their specific experiment needs.
4. Configure data loading parameters through the args system.

The framework handles the complexities of data preparation and device management, allowing researchers to focus on model implementation and experimentation.

#----------------------------------------------------------exp\_basic.py------------------------------------------#

import os

import torch

import numpy as np

class Exp\_Basic(object):

 def \_\_init\_\_(self, args):

 self.args = args

 self.device = self.\_acquire\_device()

 self.model = self.\_build\_model().to(self.device)

 def \_build\_model(self):

 raise NotImplementedError

 return None

 def \_acquire\_device(self):

 if self.args.use\_gpu:

 os.environ["CUDA\_VISIBLE\_DEVICES"] = str(

 self.args.gpu) if not self.args.use\_multi\_gpu else self.args.devices

 device = torch.device('cuda:{}'.format(self.args.gpu))

 print('Use GPU: cuda:{}'.format(self.args.gpu))

 else:

 device = torch.device('cpu')

 print('Use CPU')

 return device

 def \_get\_data(self):

 pass

 def vali(self):

 pass

 def train(self):

 pass

 def test(self):

 pass

#----------------------------------------------------------exp\_basic.py------------------------------------------#

1.3.6 Generation of exp\_main.py

The code implements a complete experimental framework for time series forecasting, particularly focused on financial data. The system is structured with clear separation of concerns:

1. exp\_basic.py - The abstract base class providing fundamental experiment infrastructure:
	* Device management (CPU/GPU)
	* Model initialization
	* Template methods for core functionality
2. exp\_main.py - The main experiment implementation that extends Exp\_Basic:
	* Implements training, validation, and testing loops.
	* Handles model optimization and loss calculation.
	* Supports mixed-precision training with AMP.
	* Includes early stopping and learning rate adjustment.
	* Provides comprehensive metrics calculation and result saving.
	* Implements prediction functionality for real-world forecasting.
3. Data Management System:
	* data\_factory.py: Manages dataset creation and configuration.
	* data\_loader.py: Handles data preprocessing, normalization, and sequence preparation.

The system supports:

* Multiple financial markets (stocks, crypto, commodities)
* Flexible sequence lengths for input/prediction
* Both single and multi-feature time series
* Automated train/validation/test splits
* Comprehensive result logging and visualization
* GPU acceleration with multi-GPU support
* Mixed precision training for improved performance

This implementation particularly focuses on transformer-based models with correlation mechanisms, providing a robust framework for financial time series forecasting research and applications. Developers—including users, scholars, or researchers may copy the given code and paste it in a file named exp\_main.py and save it.

#----------------------------------------------------------exp\_main.py------------------------------------------#

from data\_provider.data\_factory import data\_provider

from exp.exp\_basic import Exp\_Basic

import CNNCorrelationBasedTransformer

from utils.tools import EarlyStopping, adjust\_learning\_rate, visual

from utils.metrics import metric

import numpy as np

import torch

import torch.nn as nn

from torch import optim

import os

import time

import warnings

import matplotlib.pyplot as plt

import numpy as np

warnings.filterwarnings('ignore')

class Exp\_Main(Exp\_Basic):

 def \_\_init\_\_(self, args):

 super(Exp\_Main, self).\_\_init\_\_(args)

 def \_build\_model(self):

 model\_dict = {

 'CNNCorrelationBasedTransformer': CNNCorrelationBasedTransformer,

 }

 model = model\_dict[self.args.model].Model(self.args).float()

 if self.args.use\_multi\_gpu and self.args.use\_gpu:

 model = nn.DataParallel(model, device\_ids=self.args.device\_ids)

 return model

 def \_get\_data(self, flag):

 data\_set, data\_loader = data\_provider(self.args, flag)

 return data\_set, data\_loader

 def \_select\_optimizer(self):

 model\_optim = optim.Adam(self.model.parameters(), lr=self.args.learning\_rate)

 return model\_optim

 def \_select\_criterion(self):

 criterion = nn.MSELoss()

 return criterion

 def vali(self, vali\_data, vali\_loader, criterion):

 total\_loss = []

 self.model.eval()

 with torch.no\_grad():

 for i, (batch\_x, batch\_y, batch\_x\_mark, batch\_y\_mark) in enumerate(vali\_loader):

 batch\_x = batch\_x.float().to(self.device)

 batch\_y = batch\_y.float()

 batch\_x\_mark = batch\_x\_mark.float().to(self.device)

 batch\_y\_mark = batch\_y\_mark.float().to(self.device)

 # decoder input

 dec\_inp = torch.zeros\_like(batch\_y[:, -self.args.pred\_len:, :]).float()

 dec\_inp = torch.cat([batch\_y[:, :self.args.label\_len, :], dec\_inp], dim=1).float().to(self.device)

 # encoder - decoder

 if self.args.use\_amp:

 with torch.cuda.amp.autocast():

 if self.args.output\_attention:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)[0]

 else:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)

 else:

 if self.args.output\_attention:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)[0]

 else:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)

 f\_dim = -1 if self.args.features == 'MS' else 0

 outputs = outputs[:, -self.args.pred\_len:, f\_dim:]

 batch\_y = batch\_y[:, -self.args.pred\_len:, f\_dim:].to(self.device)

 pred = outputs.detach().cpu()

 true = batch\_y.detach().cpu()

 loss = criterion(pred, true)

 total\_loss.append(loss)

 total\_loss = np.average(total\_loss)

 self.model.train()

 return total\_loss

 def train(self, setting):

 train\_data, train\_loader = self.\_get\_data(flag='train')

 vali\_data, vali\_loader = self.\_get\_data(flag='val')

 test\_data, test\_loader = self.\_get\_data(flag='test')

 path = os.path.join(self.args.checkpoints, setting)

 if not os.path.exists(path):

 os.makedirs(path)

 time\_now = time.time()

 train\_steps = len(train\_loader)

 early\_stopping = EarlyStopping(patience=self.args.patience, verbose=True)

 model\_optim = self.\_select\_optimizer()

 criterion = self.\_select\_criterion()

 if self.args.use\_amp:

 scaler = torch.cuda.amp.GradScaler()

 for epoch in range(self.args.train\_epochs):

 iter\_count = 0

 train\_loss = []

 self.model.train()

 epoch\_time = time.time()

 for i, (batch\_x, batch\_y, batch\_x\_mark, batch\_y\_mark) in enumerate(train\_loader):

 iter\_count += 1

 model\_optim.zero\_grad()

 batch\_x = batch\_x.float().to(self.device)

 batch\_y = batch\_y.float().to(self.device)

 batch\_x\_mark = batch\_x\_mark.float().to(self.device)

 batch\_y\_mark = batch\_y\_mark.float().to(self.device)

 # decoder input

 dec\_inp = torch.zeros\_like(batch\_y[:, -self.args.pred\_len:, :]).float()

 dec\_inp = torch.cat([batch\_y[:, :self.args.label\_len, :], dec\_inp], dim=1).float().to(self.device)

 # encoder - decoder

 if self.args.use\_amp:

 with torch.cuda.amp.autocast():

 if self.args.output\_attention:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)[0]

 else:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)

 f\_dim = -1 if self.args.features == 'MS' else 0

 outputs = outputs[:, -self.args.pred\_len:, f\_dim:]

 batch\_y = batch\_y[:, -self.args.pred\_len:, f\_dim:].to(self.device)

 loss = criterion(outputs, batch\_y)

 train\_loss.append(loss.item())

 else:

 if self.args.output\_attention:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)[0]

 else:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)

 f\_dim = -1 if self.args.features == 'MS' else 0

 outputs = outputs[:, -self.args.pred\_len:, f\_dim:]

 batch\_y = batch\_y[:, -self.args.pred\_len:, f\_dim:].to(self.device)

 loss = criterion(outputs, batch\_y)

 train\_loss.append(loss.item())

 if (i + 1) % 100 == 0:

 print("\titers: {0}, epoch: {1} | loss: {2:.7f}".format(i + 1, epoch + 1, loss.item()))

 speed = (time.time() - time\_now) / iter\_count

 left\_time = speed \* ((self.args.train\_epochs - epoch) \* train\_steps - i)

 print('\tspeed: {:.4f}s/iter; left time: {:.4f}s'.format(speed, left\_time))

 iter\_count = 0

 time\_now = time.time()

 if self.args.use\_amp:

 scaler.scale(loss).backward()

 scaler.step(model\_optim)

 scaler.update()

 else:

 loss.backward()

 model\_optim.step()

 print("Epoch: {} cost time: {}".format(epoch + 1, time.time() - epoch\_time))

 train\_loss = np.average(train\_loss)

 vali\_loss = self.vali(vali\_data, vali\_loader, criterion)

 test\_loss = self.vali(test\_data, test\_loader, criterion)

 print("Epoch: {0}, Steps: {1} | Train Loss: {2:.7f} Vali Loss: {3:.7f} Test Loss: {4:.7f}".format(

 epoch + 1, train\_steps, train\_loss, vali\_loss, test\_loss))

 early\_stopping(vali\_loss, self.model, path)

 if early\_stopping.early\_stop:

 print("Early stopping")

 break

 adjust\_learning\_rate(model\_optim, epoch + 1, self.args)

 best\_model\_path = path + '/' + 'checkpoint.pth'

 self.model.load\_state\_dict(torch.load(best\_model\_path))

 return

 def test(self, setting, test=0):

 test\_data, test\_loader = self.\_get\_data(flag='test')

 if test:

 print('loading model')

 self.model.load\_state\_dict(torch.load(os.path.join('./checkpoints/' + setting, 'checkpoint.pth')))

 preds = []

 trues = []

 folder\_path = './test\_results/' + setting + '/'

 if not os.path.exists(folder\_path):

 os.makedirs(folder\_path)

 self.model.eval()

 with torch.no\_grad():

 for i, (batch\_x, batch\_y, batch\_x\_mark, batch\_y\_mark) in enumerate(test\_loader):

 batch\_x = batch\_x.float().to(self.device)

 batch\_y = batch\_y.float().to(self.device)

 batch\_x\_mark = batch\_x\_mark.float().to(self.device)

 batch\_y\_mark = batch\_y\_mark.float().to(self.device)

 # decoder input

 dec\_inp = torch.zeros\_like(batch\_y[:, -self.args.pred\_len:, :]).float()

 dec\_inp = torch.cat([batch\_y[:, :self.args.label\_len, :], dec\_inp], dim=1).float().to(self.device)

 # encoder - decoder

 if self.args.use\_amp:

 with torch.cuda.amp.autocast():

 if self.args.output\_attention:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)[0]

 else:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)

 else:

 if self.args.output\_attention:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)[0]

 else:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)

 f\_dim = -1 if self.args.features == 'MS' else 0

 outputs = outputs[:, -self.args.pred\_len:, f\_dim:]

 batch\_y = batch\_y[:, -self.args.pred\_len:, f\_dim:].to(self.device)

 outputs = outputs.detach().cpu().numpy()

 batch\_y = batch\_y.detach().cpu().numpy()

 pred = outputs # outputs.detach().cpu().numpy() # .squeeze()

 true = batch\_y # batch\_y.detach().cpu().numpy() # .squeeze()

 preds.append(pred)

 trues.append(true)

 if i % 20 == 0:

 input = batch\_x.detach().cpu().numpy()

 gt = np.concatenate((input[0, :, -1], true[0, :, -1]), axis=0)

 pd = np.concatenate((input[0, :, -1], pred[0, :, -1]), axis=0)

 visual(gt, pd, os.path.join(folder\_path, str(i) + '.pdf'))

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

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

 print('test shape:', preds.shape, trues.shape)

 preds = preds.reshape(-1, preds.shape[-2], preds.shape[-1])

 trues = trues.reshape(-1, trues.shape[-2], trues.shape[-1])

 print('test shape:', preds.shape, trues.shape)

 # result save

 folder\_path = './results/' + setting + '/'

 if not os.path.exists(folder\_path):

 os.makedirs(folder\_path)

 mae, mse, rmse, mape, mspe, R2, rse = metric(preds, trues)

 print('mse:{}, mae:{}'.format(mse, mae))

 f = open("result.txt", 'a')

 f.write(setting + " \n")

 f.write('mse:{}, mae:{}'.format(mse, mae))

 f.write('\n')

 f.write('\n')

 f.close()

 np.save(folder\_path + 'metrics.npy', np.array([mae, mse, rmse, mape, mspe, R2, rse]))

 np.save(folder\_path + 'pred.npy', preds)

 np.save(folder\_path + 'true.npy', trues)

 return

 def predict(self, setting, load=False):

 pred\_data, pred\_loader = self.\_get\_data(flag='pred')

 if load:

 path = os.path.join(self.args.checkpoints, setting)

 best\_model\_path = path + '/' + 'checkpoint.pth'

 self.model.load\_state\_dict(torch.load(best\_model\_path))

 preds = []

 self.model.eval()

 with torch.no\_grad():

 for i, (batch\_x, batch\_y, batch\_x\_mark, batch\_y\_mark) in enumerate(pred\_loader):

 batch\_x = batch\_x.float().to(self.device)

 batch\_y = batch\_y.float()

 batch\_x\_mark = batch\_x\_mark.float().to(self.device)

 batch\_y\_mark = batch\_y\_mark.float().to(self.device)

 # decoder input

 dec\_inp = torch.zeros([batch\_y.shape[0], self.args.pred\_len, batch\_y.shape[2]]).float()

 dec\_inp = torch.cat([batch\_y[:, :self.args.label\_len, :], dec\_inp], dim=1).float().to(self.device)

 # encoder - decoder

 if self.args.use\_amp:

 with torch.cuda.amp.autocast():

 if self.args.output\_attention:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)[0]

 else:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)

 else:

 if self.args.output\_attention:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)[0]

 else:

 outputs = self.model(batch\_x, batch\_x\_mark, dec\_inp, batch\_y\_mark)

 pred = outputs.detach().cpu().numpy() # .squeeze()

 preds.append(pred)

 preds = np.array(preds)

 preds = preds.reshape(-1, preds.shape[-2], preds.shape[-1])

 # result save

 folder\_path = './results/' + setting + '/'

 if not os.path.exists(folder\_path):

 os.makedirs(folder\_path)

 np.save(folder\_path + 'real\_prediction.npy', preds)

 return

#----------------------------------------------------------exp\_main.py------------------------------------------#

1.3.7 Generation of masking.py

This code implements two essential masking classes for attention mechanisms in transformer-based models, particularly for time series forecasting:

1. TriangularCausalMask: Creates a causal (triangular) attention mask that ensures each position can only attend to previous positions and itself. This is crucial for maintaining the temporal causality in time series predictions - preventing information leakage from future to past timesteps.
2. ProbMask: Implements a probabilistic mask for attention scores, typically used in probabilistic attention mechanisms. It combines triangular masking with specific attention patterns based on provided indices and scores, allowing for more sophisticated attention patterns while maintaining causality.

Both classes use PyTorch's efficient tensor operations and support both CPU and GPU execution through the device parameter. The masks are implemented as boolean tensors where True values indicate positions that should be masked (ignored) in the attention computation.

Developers can copy this code into a file named masking.py and import these classes for implementing attention mechanisms in their transformer-based models. The classes are particularly designed for sequence modeling tasks where maintaining temporal causality is important.

#-----------------------------------------------------------masking.py-------------------------------------------#

import torch

class TriangularCausalMask():

 def \_\_init\_\_(self, B, L, device="cpu"):

 mask\_shape = [B, 1, L, L]

 with torch.no\_grad():

 self.\_mask = torch.triu(torch.ones(mask\_shape, dtype=torch.bool), diagonal=1).to(device)

 @property

 def mask(self):

 return self.\_mask

class ProbMask():

 def \_\_init\_\_(self, B, H, L, index, scores, device="cpu"):

 \_mask = torch.ones(L, scores.shape[-1], dtype=torch.bool).to(device).triu(1)

 \_mask\_ex = \_mask[None, None, :].expand(B, H, L, scores.shape[-1])

 indicator = \_mask\_ex[torch.arange(B)[:, None, None],

 torch.arange(H)[None, :, None],

 index, :].to(device)

 self.\_mask = indicator.view(scores.shape).to(device)

 @property

 def mask(self):

 return self.\_mask

#-----------------------------------------------------------masking.py-------------------------------------------#

1.3.8 Generation of metrics.py

This code provides a comprehensive collection of common evaluation metrics for comparing predicted values against true (actual) values, primarily used in regression and forecasting tasks. The file implements eight key metrics: Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Percentage Error (MAPE), Mean Squared Percentage Error (MSPE), R-squared (R²), Mean Hassanat Distance (MHD), and Relative Squared Error (RSE). Additionally, it includes a correlation coefficient (CORR) calculation.

The main function metric() serves as a convenience wrapper that returns all primary metrics in a single call. To use this code, users need to have NumPy installed and should pass their predicted and true values as NumPy arrays of matching shapes. All metrics are implemented to handle both single-dimensional and multi-dimensional arrays, making them suitable for various prediction tasks.

To reproduce this functionality, users can simply copy the provided code into a file named 'metrics.py' and import it into their projects. The only dependency required is NumPy. All functions expect two arguments: pred (predicted values) and true (actual values), both as NumPy arrays.

#------------------------------------------------------------metrics.py-------------------------------------------#

import numpy as np

def RSE(pred, true):

 return np.sqrt(np.sum((true - pred) \*\* 2)) / np.sqrt(np.sum((true - true.mean()) \*\* 2))

def CORR(pred, true):

 u = ((true - true.mean(0)) \* (pred - pred.mean(0))).sum(0)

 d = np.sqrt(((true - true.mean(0)) \*\* 2).sum(0) \* ((pred - pred.mean(0)) \*\* 2).sum(0))

 return (u / d).mean(-1)

def MAE(pred, true):

 return np.mean(np.abs(pred - true))

def MSE(pred, true):

 return np.mean((pred - true) \*\* 2)

def RMSE(pred, true):

 return np.sqrt(MSE(pred, true))

def MAPE(pred, true):

 return np.mean(np.abs((pred - true) / true))

def MSPE(pred, true):

 return np.mean(np.square((pred - true) / true))

def MHD(preds, trues):

 pred = preds.reshape(-1)

 true = trues.reshape(-1)

 N,M,P = np.shape(preds)

 total\_distance = 0

 for i in range(N):

 for j in range(M):

 pred = preds[i][j][0]

 true = trues[i][j][0]

 min\_value = min(pred, true)

 max\_value = max(pred, true)

 total\_distance += 1

 if min\_value >= 0:

 total\_distance -= (1 + min\_value) / (1 + max\_value)

 else:

 total\_distance -= 1 / (1 + max\_value + abs(min\_value))

 mean\_distance = total\_distance / (N\*M)

 return mean\_distance

def r\_squared(pred, true):

 ssr = np.sum((pred - true) \*\* 2)

 sst = np.sum((true - np.mean(true)) \*\* 2)

 return 1 - (ssr/sst)

def metric(pred, true):

 mae = MAE(pred, true)

 mse = MSE(pred, true)

 rmse = RMSE(pred, true)

 mape = MAPE(pred, true)

 mspe = MSPE(pred, true)

 R2 = r\_squared(pred, true)

 mhd = MHD(pred, true)

 rse = RSE(pred, true)

 return mae, mse, rmse, mape, mspe, R2, mhd, rse

#------------------------------------------------------------metrics.py-------------------------------------------#

1.3.9 Generation of timefeatures.py

The code provides a collection of time feature classes that inherit from a base TimeFeature class, each converting different temporal components (seconds, minutes, hours, days, weeks, months) into normalized values between -0.5 and 0.5. The main functionality is exposed through two key functions: time\_features\_from\_frequency\_str() which returns appropriate time features based on a frequency string (like "12H", "5min", "1D"), and time\_features() which converts a date index into a stacked array of these features.

To use this code, save it as 'timefeatures.py' and ensure you have pandas and numpy installed. The primary entry point is the time\_features() function, which takes a pandas DatetimeIndex and an optional frequency parameter (default='h' for hourly). The code supports various time frequencies from yearly down to secondly granularity, making it particularly useful for time series modeling tasks where cyclical temporal patterns need to be encoded as numerical features.

The supported frequencies are: yearly (Y/A), monthly (M), weekly (W), daily (D), business days (B), hourly (H), minutely (T/min), and secondly (S). Each frequency automatically includes relevant time features at that level and above in the temporal hierarchy.

#---------------------------------------------------------timefeatures.py----------------------------------------#

from typing import List

import numpy as np

import pandas as pd

from pandas.tseries import offsets

from pandas.tseries.frequencies import to\_offset

class TimeFeature:

 def \_\_init\_\_(self):

 pass

 def \_\_call\_\_(self, index: pd.DatetimeIndex) -> np.ndarray:

 pass

 def \_\_repr\_\_(self):

 return self.\_\_class\_\_.\_\_name\_\_ + "()"

class SecondOfMinute(TimeFeature):

 """Minute of hour encoded as value between [-0.5, 0.5]"""

 def \_\_call\_\_(self, index: pd.DatetimeIndex) -> np.ndarray:

 return index.second / 59.0 - 0.5

class MinuteOfHour(TimeFeature):

 """Minute of hour encoded as value between [-0.5, 0.5]"""

 def \_\_call\_\_(self, index: pd.DatetimeIndex) -> np.ndarray:

 return index.minute / 59.0 - 0.5

class HourOfDay(TimeFeature):

 """Hour of day encoded as value between [-0.5, 0.5]"""

 def \_\_call\_\_(self, index: pd.DatetimeIndex) -> np.ndarray:

 return index.hour / 23.0 - 0.5

class DayOfWeek(TimeFeature):

 """Hour of day encoded as value between [-0.5, 0.5]"""

 def \_\_call\_\_(self, index: pd.DatetimeIndex) -> np.ndarray:

 return index.dayofweek / 6.0 - 0.5

class DayOfMonth(TimeFeature):

 """Day of month encoded as value between [-0.5, 0.5]"""

 def \_\_call\_\_(self, index: pd.DatetimeIndex) -> np.ndarray:

 return (index.day - 1) / 30.0 - 0.5

class DayOfYear(TimeFeature):

 """Day of year encoded as value between [-0.5, 0.5]"""

 def \_\_call\_\_(self, index: pd.DatetimeIndex) -> np.ndarray:

 return (index.dayofyear - 1) / 365.0 - 0.5

class MonthOfYear(TimeFeature):

 """Month of year encoded as value between [-0.5, 0.5]"""

 def \_\_call\_\_(self, index: pd.DatetimeIndex) -> np.ndarray:

 return (index.month - 1) / 11.0 - 0.5

class WeekOfYear(TimeFeature):

 """Week of year encoded as value between [-0.5, 0.5]"""

 def \_\_call\_\_(self, index: pd.DatetimeIndex) -> np.ndarray:

 return (index.isocalendar().week - 1) / 52.0 - 0.5

def time\_features\_from\_frequency\_str(freq\_str: str) -> List[TimeFeature]:

 """

 Returns a list of time features that will be appropriate for the given frequency string.

 Parameters

 ----------

 freq\_str

 Frequency string of the form [multiple][granularity] such as "12H", "5min", "1D" etc.

 """

 features\_by\_offsets = {

 offsets.YearEnd: [],

 offsets.QuarterEnd: [MonthOfYear],

 offsets.MonthEnd: [MonthOfYear],

 offsets.Week: [DayOfMonth, WeekOfYear],

 offsets.Day: [DayOfWeek, DayOfMonth, DayOfYear],

 offsets.BusinessDay: [DayOfWeek, DayOfMonth, DayOfYear],

 offsets.Hour: [HourOfDay, DayOfWeek, DayOfMonth, DayOfYear],

 offsets.Minute: [

 MinuteOfHour,

 HourOfDay,

 DayOfWeek,

 DayOfMonth,

 DayOfYear,

 ],

 offsets.Second: [

 SecondOfMinute,

 MinuteOfHour,

 HourOfDay,

 DayOfWeek,

 DayOfMonth,

 DayOfYear,

 ],

 }

 offset = to\_offset(freq\_str)

 for offset\_type, feature\_classes in features\_by\_offsets.items():

 if isinstance(offset, offset\_type):

 return [cls() for cls in feature\_classes]

 supported\_freq\_msg = f"""

 Unsupported frequency {freq\_str}

 The following frequencies are supported:

 Y - yearly

 alias: A

 M - monthly

 W - weekly

 D - daily

 B - business days

 H - hourly

 T - minutely

 alias: min

 S - secondly

 """

 raise RuntimeError(supported\_freq\_msg)

def time\_features(dates, freq='h'):

 return np.vstack([feat(dates) for feat in time\_features\_from\_frequency\_str(freq)])

#---------------------------------------------------------timefeatures.py----------------------------------------#

1.3.10 Generation of tools.py

The code contains several key utilities: a learning rate adjustment function (adjust\_learning\_rate) with two adjustment strategies, an early stopping mechanism (EarlyStopping) to prevent overfitting, a standard scaler implementation (StandardScaler) for data normalization, a dictionary extension (dotdict) that allows dot notation access, and a visualization function (visual) for plotting predictions against ground truth.

To use this code, save it as 'tools.py' and ensure you have PyTorch, NumPy, and Matplotlib installed. The learning rate adjustment supports two types of schedules: 'type1' (exponential decay) and 'type2' (predefined steps). The EarlyStopping class monitors validation loss and saves the best model checkpoint, while the StandardScaler handles data normalization and denormalization. The visualization function creates simple line plots comparing true values with predictions.

Most importantly, these tools are designed to work together in a deep learning training pipeline, with each component handling a specific aspect of the training process: learning rate scheduling, model checkpointing, data normalization, and results visualization. The visualization output is configured to use the 'agg' backend, making it suitable for environments without display capabilities.

#-------------------------------------------------------------tools.py---------------------------------------------#

import numpy as np

import torch

import matplotlib.pyplot as plt

plt.switch\_backend('agg')

def adjust\_learning\_rate(optimizer, epoch, args):

 # lr = args.learning\_rate \* (0.2 \*\* (epoch // 2))

 if args.lradj == 'type1':

 lr\_adjust = {epoch: args.learning\_rate \* (0.5 \*\* ((epoch - 1) // 1))}

 elif args.lradj == 'type2':

 lr\_adjust = {

 2: 5e-5, 4: 1e-5, 6: 5e-6, 8: 1e-6,

 10: 5e-7, 15: 1e-7, 20: 5e-8

 }

 if epoch in lr\_adjust.keys():

 lr = lr\_adjust[epoch]

 for param\_group in optimizer.param\_groups:

 param\_group['lr'] = lr

 print('Updating learning rate to {}'.format(lr))

class EarlyStopping:

 def \_\_init\_\_(self, patience=7, verbose=False, delta=0):

 self.patience = patience

 self.verbose = verbose

 self.counter = 0

 self.best\_score = None

 self.early\_stop = False

 self.val\_loss\_min = np.Inf

 self.delta = delta

 def \_\_call\_\_(self, val\_loss, model, path):

 score = -val\_loss

 if self.best\_score is None:

 self.best\_score = score

 self.save\_checkpoint(val\_loss, model, path)

 elif score < self.best\_score + self.delta:

 self.counter += 1

 print(f'EarlyStopping counter: {self.counter} out of {self.patience}')

 if self.counter >= self.patience:

 self.early\_stop = True

 else:

 self.best\_score = score

 self.save\_checkpoint(val\_loss, model, path)

 self.counter = 0

 def save\_checkpoint(self, val\_loss, model, path):

 if self.verbose:

 print(f'Validation loss decreased ({self.val\_loss\_min:.6f} --> {val\_loss:.6f}). Saving model ...')

 torch.save(model.state\_dict(), path + '/' + 'checkpoint.pth')

 self.val\_loss\_min = val\_loss

class dotdict(dict):

 """dot.notation access to dictionary attributes"""

 \_\_getattr\_\_ = dict.get

 \_\_setattr\_\_ = dict.\_\_setitem\_\_

 \_\_delattr\_\_ = dict.\_\_delitem\_\_

class StandardScaler():

 def \_\_init\_\_(self, mean, std):

 self.mean = mean

 self.std = std

 def transform(self, data):

 return (data - self.mean) / self.std

 def inverse\_transform(self, data):

 return (data \* self.std) + self.mean

def visual(true, preds=None, name='./pic/test.pdf'):

 """

 Results visualization

 """

 plt.figure()

 plt.plot(true, label='GroundTruth', linewidth=2)

 if preds is not None:

 plt.plot(preds, label='Prediction', linewidth=2)

 plt.legend()

 plt.savefig(name, bbox\_inches='tight')

#-------------------------------------------------------------tools.py---------------------------------------------#

1.3.11 Generation of CNNCorrelationBasedTransformer.py

This code implements a CNN Correlation-based Transformer model designed for time series forecasting. The model builds upon the traditional Transformer architecture but introduces a novel correlation mechanism that captures periodic dependencies in time series data. The implementation consists of 17 main components, including 16 classes and 1 utility function:

1. version\_inspection (utility function for version comparison)
2. Model (main class)
3. my\_Layernorm
4. moving\_avg
5. SeriesDecomposing
6. EncoderLayer
7. Encoder
8. DecoderLayer
9. Decoder
10. Correlation (key innovation)
11. CorrelationLayer
12. TokenEmbedding
13. FixedEncoding
14. TemporalEncoding
15. TimeFeatureEncoding
16. DataEncoding

The key innovation lies in the Correlation class, which replaces traditional attention mechanisms with a CNN correlation-based approach. It uses Fast Fourier Transform (FFT) to identify periodic patterns and implements a Time Shift and Concatenation (TSC) mechanism that aggregates information across different time shifts. The version\_inspection function is a utility that compares version numbers (like "1.5.0" vs "1.4.0") to ensure compatibility with different PyTorch versions, particularly used in the TokenEmbedding class for padding calculations.

To reproduce this model:

1. Ensure you have PyTorch installed with FFT support.
2. The model requires configuration parameters including sequence length (seq\_len), label length (label\_len), prediction length (pred\_len), and model dimensions (d\_model).
3. Input data should include both the time series values and temporal markers (like hour, day, month).
4. The model processes data through an encoder-decoder architecture, with CNN correlation-based attention mechanisms in both components.

Developers—including users, scholars, or researchers may copy the given code and paste it in a file CNNCorrelationBasedTransformer.py and save it.

#-----------------------------------------CNNCorrelationBasedTransformer.py--------------------------------#

import torch

import torch.nn as nn

import torch.nn.functional as F

import numpy as np

import matplotlib.pyplot as plt

import math

from math import sqrt

import os

from torch.nn.utils import weight\_norm

def version\_inspection(ver1, ver2):

 """

 :param ver1

 :param ver2

 :return: ver1< = >ver2 False/True

 """

 list1 = str(ver1).split(".")

 list2 = str(ver2).split(".")

 for i in range(len(list1)) if len(list1) < len(list2) else range(len(list2)):

 if int(list1[i]) == int(list2[i]):

 pass

 elif int(list1[i]) < int(list2[i]):

 return -1

 else:

 return 1

 if len(list1) == len(list2):

 return True

 elif len(list1) < len(list2):

 return False

 else:

 return True

class Model(nn.Module):

 def \_\_init\_\_(self, configs):

 super(Model, self).\_\_init\_\_()

 self.seq\_len = configs.seq\_len

 self.label\_len = configs.label\_len

 self.pred\_len = configs.pred\_len

 self.output\_attention = configs.output\_attention

 # Series Decomposing

 kernel\_size = configs.moving\_avg

 self.decomp = SeriesDecomposing(kernel\_size)

 # Embedding

 # The series-wise connection inherently contains the sequential information.

 # Thus, we can discard the position embedding of transformers.

 self.enc\_embedding = DataEmbedding(configs.enc\_in, configs.d\_model, configs.embed, configs.freq,

 configs.dropout)

 self.dec\_embedding = DataEmbedding(configs.dec\_in, configs.d\_model, configs.embed, configs.freq,

 configs.dropout)

 # Encoder

 self.encoder = Encoder(

 [

 EncoderLayer(

 CorrelationLayer(

 Correlation(False, configs.factor, attention\_dropout=configs.dropout,

 output\_attention=configs.output\_attention),

 configs.d\_model, configs.cnn\_out\_channels, configs.n\_heads),

 configs.d\_model,

 configs.d\_ff,

 moving\_avg=configs.moving\_avg,

 dropout=configs.dropout,

 activation=configs.activation

 ) for l in range(configs.e\_layers)

 ],

 norm\_layer=my\_Layernorm(configs.d\_model)

 )

 # Decoder

 self.decoder = Decoder(

 [

 DecoderLayer(

 CorrelationLayer(

 Correlation(True, configs.factor, attention\_dropout=configs.dropout,

 output\_attention=False),

 configs.d\_model, configs.cnn\_out\_channels, configs.n\_heads),

 CorrelationLayer(

 Correlation(False, configs.factor, attention\_dropout=configs.dropout,

 output\_attention=False),

 configs.d\_model, configs.cnn\_out\_channels, configs.n\_heads),

 configs.d\_model,

 configs.c\_out,

 configs.d\_ff,

 moving\_avg=configs.moving\_avg,

 dropout=configs.dropout,

 activation=configs.activation,

 )

 for l in range(configs.d\_layers)

 ],

 norm\_layer=my\_Layernorm(configs.d\_model),

 projection=nn.Linear(configs.d\_model, configs.c\_out, bias=True)

 )

 def forward(self, x\_enc, x\_mark\_enc, x\_dec, x\_mark\_dec,

 enc\_self\_mask=None, dec\_self\_mask=None, dec\_enc\_mask=None):

 # decomp init

 mean = torch.mean(x\_enc, dim=1).unsqueeze(1).repeat(1, self.pred\_len, 1)

 zeros = torch.zeros([x\_dec.shape[0], self.pred\_len, x\_dec.shape[2]], device=x\_enc.device)

 seasonal\_init, trend\_init = self.decomp(x\_enc)

 # decoder input

 trend\_init = torch.cat([trend\_init[:, -self.label\_len:, :], mean], dim=1)

 seasonal\_init = torch.cat([seasonal\_init[:, -self.label\_len:, :], zeros], dim=1)

 # enc

 enc\_out = self.enc\_embedding(x\_enc, x\_mark\_enc)

 enc\_out, attns = self.encoder(enc\_out, attn\_mask=enc\_self\_mask)

 # dec

 dec\_out = self.dec\_embedding(seasonal\_init, x\_mark\_dec)

 seasonal\_part, trend\_part = self.decoder(dec\_out, enc\_out, x\_mask=dec\_self\_mask, cross\_mask=dec\_enc\_mask,

 trend=trend\_init)

 # final

 dec\_out = trend\_part + seasonal\_part

 if self.output\_attention:

 return dec\_out[:, -self.pred\_len:, :], attns

 else:

 return dec\_out[:, -self.pred\_len:, :] # [B, L, D]

class my\_Layernorm(nn.Module):

 """

 Special designed layernorm for the seasonal part

 """

 def \_\_init\_\_(self, channels):

 super(my\_Layernorm, self).\_\_init\_\_()

 self.layernorm = nn.LayerNorm(channels)

 def forward(self, x):

 x\_hat = self.layernorm(x)

 bias = torch.mean(x\_hat, dim=1).unsqueeze(1).repeat(1, x.shape[1], 1)

 return x\_hat - bias

class moving\_avg(nn.Module):

 def \_\_init\_\_(self, kernel\_size, stride):

 super(moving\_avg, self).\_\_init\_\_()

 self.kernel\_size = kernel\_size

 self.avg = nn.AvgPool1d(kernel\_size=kernel\_size, stride=stride, padding=0)

 def forward(self, x):

 # padding on the both ends of time series

 front = x[:, 0:1, :].repeat(1, (self.kernel\_size - 1) // 2, 1)

 end = x[:, -1:, :].repeat(1, (self.kernel\_size - 1) // 2, 1)

 x = torch.cat([front, x, end], dim=1)

 x = self.avg(x.permute(0, 2, 1))

 x = x.permute(0, 2, 1)

 return x

class SeriesDecomposing(nn.Module):

 def \_\_init\_\_(self, kernel\_size):

 super(SeriesDecomposing, self).\_\_init\_\_()

 self.moving\_avg = moving\_avg(kernel\_size, stride=1)

 def forward(self, x):

 moving\_mean = self.moving\_avg(x)

 res = x - moving\_mean

 return res, moving\_mean

class EncoderLayer(nn.Module):

 def \_\_init\_\_(self, attention, d\_model, d\_ff=None, moving\_avg=25, dropout=0.1, activation="relu"):

 super(EncoderLayer, self).\_\_init\_\_()

 d\_ff = d\_ff or 4 \* d\_model

 self.attention = attention

 self.conv1 = nn.Conv1d(in\_channels=d\_model, out\_channels=d\_ff, kernel\_size=1, bias=False)

 self.conv2 = nn.Conv1d(in\_channels=d\_ff, out\_channels=d\_model, kernel\_size=1, bias=False)

 self.decomp1 = SeriesDecomposing(moving\_avg)

 self.decomp2 = SeriesDecomposing(moving\_avg)

 self.dropout = nn.Dropout(dropout)

 self.activation = F.relu if activation == "relu" else F.gelu

 def forward(self, x, attn\_mask=None):

 new\_x, attn = self.attention(

 x, x, x,

 attn\_mask=attn\_mask

 )

 x = x + self.dropout(new\_x)

 x, \_ = self.decomp1(x)

 y = x

 y = self.dropout(self.activation(self.conv1(y.transpose(-1, 1))))

 y = self.dropout(self.conv2(y).transpose(-1, 1))

 res, \_ = self.decomp2(x + y)

 return res, attn

class Encoder(nn.Module):

 def \_\_init\_\_(self, attn\_layers, conv\_layers=None, norm\_layer=None):

 super(Encoder, self).\_\_init\_\_()

 self.attn\_layers = nn.ModuleList(attn\_layers)

 self.conv\_layers = nn.ModuleList(conv\_layers) if conv\_layers is not None else None

 self.norm = norm\_layer

 def forward(self, x, attn\_mask=None):

 attns = []

 if self.conv\_layers is not None:

 for attn\_layer, conv\_layer in zip(self.attn\_layers, self.conv\_layers):

 x, attn = attn\_layer(x, attn\_mask=attn\_mask)

 x = conv\_layer(x)

 attns.append(attn)

 x, attn = self.attn\_layers[-1](x)

 attns.append(attn)

 else:

 for attn\_layer in self.attn\_layers:

 x, attn = attn\_layer(x, attn\_mask=attn\_mask)

 attns.append(attn)

 if self.norm is not None:

 x = self.norm(x)

 return x, attns

class DecoderLayer(nn.Module):

 def \_\_init\_\_(self, self\_attention, cross\_attention, d\_model, c\_out, d\_ff=None,

 moving\_avg=25, dropout=0.1, activation="relu"):

 super(DecoderLayer, self).\_\_init\_\_()

 d\_ff = d\_ff or 4 \* d\_model

 self.self\_attention = self\_attention

 self.cross\_attention = cross\_attention

 self.conv1 = nn.Conv1d(in\_channels=d\_model, out\_channels=d\_ff, kernel\_size=1, bias=False)

 self.conv2 = nn.Conv1d(in\_channels=d\_ff, out\_channels=d\_model, kernel\_size=1, bias=False)

 self.decomp1 = SeriesDecomposing(moving\_avg)

 self.decomp2 = SeriesDecomposing(moving\_avg)

 self.decomp3 = SeriesDecomposing(moving\_avg)

 self.dropout = nn.Dropout(dropout)

 self.projection = nn.Conv1d(in\_channels=d\_model, out\_channels=c\_out, kernel\_size=3, stride=1, padding=1,

 padding\_mode='circular', bias=False)

 self.activation = F.relu if activation == "relu" else F.gelu

 def forward(self, x, cross, x\_mask=None, cross\_mask=None):

 x = x + self.dropout(self.self\_attention(

 x, x, x,

 attn\_mask=x\_mask

 )[0])

 x, trend1 = self.decomp1(x)

 x = x + self.dropout(self.cross\_attention(

 x, cross, cross,

 attn\_mask=cross\_mask

 )[0])

 x, trend2 = self.decomp2(x)

 y = x

 y = self.dropout(self.activation(self.conv1(y.transpose(-1, 1))))

 y = self.dropout(self.conv2(y).transpose(-1, 1))

 x, trend3 = self.decomp3(x + y)

 residual\_trend = trend1 + trend2 + trend3

 residual\_trend = self.projection(residual\_trend.permute(0, 2, 1)).transpose(1, 2)

 return x, residual\_trend

class Decoder(nn.Module):

 def \_\_init\_\_(self, layers, norm\_layer=None, projection=None):

 super(Decoder, self).\_\_init\_\_()

 self.layers = nn.ModuleList(layers)

 self.norm = norm\_layer

 self.projection = projection

 def forward(self, x, cross, x\_mask=None, cross\_mask=None, trend=None):

 for layer in self.layers:

 x, residual\_trend = layer(x, cross, x\_mask=x\_mask, cross\_mask=cross\_mask)

 trend = trend + residual\_trend

 if self.norm is not None:

 x = self.norm(x)

 if self.projection is not None:

 x = self.projection(x)

 return x, trend

class Correlation(nn.Module):

 def \_\_init\_\_(self, mask\_flag=True, factor=1, scale=None, attention\_dropout=0.1, output\_attention=False):

 super(Correlation, self).\_\_init\_\_()

 self.factor = factor

 self.scale = scale

 self.mask\_flag = mask\_flag

 self.output\_attention = output\_attention

 self.dropout = nn.Dropout(attention\_dropout)

 #TSC stands for Time Shift and Concatenation

 def TSC\_training(self, values, corr):

 head = values.shape[1]

 channel = values.shape[2]

 length = values.shape[3]

 # find top k

 top\_k = int(self.factor \* math.log(length))

 mean\_value = torch.mean(torch.mean(corr, dim=1), dim=1)

 index = torch.topk(torch.mean(mean\_value, dim=0), top\_k, dim=-1)[1]

 weights = torch.stack([mean\_value[:, index[i]] for i in range(top\_k)], dim=-1)

 # update corr

 tmp\_corr = torch.softmax(weights, dim=-1)

 # aggregation

 tmp\_values = values

 delays\_agg = torch.zeros\_like(values).float()

 for i in range(top\_k):

 pattern = torch.roll(tmp\_values, -int(index[i]), -1)

 delays\_agg = delays\_agg + pattern \* \

 (tmp\_corr[:, i].unsqueeze(1).unsqueeze(1).unsqueeze(1).repeat(1, head, channel, length))

 return delays\_agg

 def TSC\_inference(self, values, corr):

 batch = values.shape[0]

 head = values.shape[1]

 channel = values.shape[2]

 length = values.shape[3]

 # index init

 init\_index = torch.arange(length).unsqueeze(0).unsqueeze(0).unsqueeze(0)\

 .repeat(batch, head, channel, 1).to(values.device)

 # find top k

 top\_k = int(self.factor \* math.log(length))

 mean\_value = torch.mean(torch.mean(corr, dim=1), dim=1)

 weights, delay = torch.topk(mean\_value, top\_k, dim=-1)

 # update corr

 tmp\_corr = torch.softmax(weights, dim=-1)

 # aggregation

 tmp\_values = values.repeat(1, 1, 1, 2)

 delays\_agg = torch.zeros\_like(values).float()

 for i in range(top\_k):

 tmp\_delay = init\_index + delay[:, i].unsqueeze(1).unsqueeze(1).unsqueeze(1).repeat(1, head, channel, length)

 pattern = torch.gather(tmp\_values, dim=-1, index=tmp\_delay)

 delays\_agg = delays\_agg + pattern \* \

 (tmp\_corr[:, i].unsqueeze(1).unsqueeze(1).unsqueeze(1).repeat(1, head, channel, length))

 return delays\_agg

 def TSC\_full(self, values, corr):

 batch = values.shape[0]

 head = values.shape[1]

 channel = values.shape[2]

 length = values.shape[3]

 # index init

 init\_index = torch.arange(length).unsqueeze(0).unsqueeze(0).unsqueeze(0)\

 .repeat(batch, head, channel, 1).to(values.device)

 # find top k

 top\_k = int(self.factor \* math.log(length))

 weights, delay = torch.topk(corr, top\_k, dim=-1)

 # update corr

 tmp\_corr = torch.softmax(weights, dim=-1)

 # aggregation

 tmp\_values = values.repeat(1, 1, 1, 2)

 delays\_agg = torch.zeros\_like(values).float()

 for i in range(top\_k):

 tmp\_delay = init\_index + delay[..., i].unsqueeze(-1)

 pattern = torch.gather(tmp\_values, dim=-1, index=tmp\_delay)

 delays\_agg = delays\_agg + pattern \* (tmp\_corr[..., i].unsqueeze(-1))

 return delays\_agg

 def forward(self, queries, keys, values, attn\_mask):

 B, L, H, E = queries.shape

 \_, S, \_, D = values.shape

 if L > S:

 zeros = torch.zeros\_like(queries[:, :(L - S), :]).float()

 values = torch.cat([values, zeros], dim=1)

 keys = torch.cat([keys, zeros], dim=1)

 else:

 values = values[:, :L, :, :]

 keys = keys[:, :L, :, :]

 # period-based dependencies

 q\_fft = torch.fft.rfft(queries.permute(0, 2, 3, 1).contiguous(), dim=-1)

 k\_fft = torch.fft.rfft(keys.permute(0, 2, 3, 1).contiguous(), dim=-1)

 res = q\_fft \* torch.conj(k\_fft)

 corr = torch.fft.irfft(res, n=L, dim=-1)

 # time delay agg

 if self.training:

 V = self.TSC\_training(values.permute(0, 2, 3, 1).contiguous(), corr).permute(0, 3, 1, 2)

 else:

 V = self.TSC\_inference(values.permute(0, 2, 3, 1).contiguous(), corr).permute(0, 3, 1, 2)

 if self.output\_attention:

 return (V.contiguous(), corr.permute(0, 3, 1, 2))

 else:

 return (V.contiguous(), None)

class CorrelationLayer(nn.Module):

 def \_\_init\_\_(self, correlation, d\_model, n\_heads, d\_keys=None,

 d\_values=None, cnn\_out\_channels=128, kernel\_size=1):

 super(CorrelationLayer, self).\_\_init\_\_()

 d\_keys = d\_keys or (d\_model // n\_heads)

 d\_values = d\_values or (d\_model // n\_heads)

 self.inner\_correlation = correlation

 #Convert time-series data to CNN information for queries, keys, and values

 self.cnn\_queries = nn.Conv1d(in\_channels=d\_model, out\_channels=cnn\_out\_channels, kernel\_size=kernel\_size, padding=kernel\_size // 2)

 self.cnn\_keys = nn.Conv1d(in\_channels=d\_model, out\_channels=cnn\_out\_channels, kernel\_size=kernel\_size, padding=kernel\_size // 2)

 self.cnn\_values = nn.Conv1d(in\_channels=d\_model, out\_channels=cnn\_out\_channels, kernel\_size=kernel\_size, padding=kernel\_size // 2)

 #Change projection layers to align with the input size of each CNN information

 self.query\_projection = nn.Linear(cnn\_out\_channels, d\_keys \* n\_heads)

 self.key\_projection = nn.Linear(cnn\_out\_channels, d\_keys \* n\_heads)

 self.value\_projection = nn.Linear(cnn\_out\_channels, d\_values \* n\_heads)

 #

 self.out\_projection = nn.Linear(d\_values \* n\_heads, d\_model)

 self.n\_heads = n\_heads

 def forward(self, queries, keys, values, attn\_mask):

 B, L, \_ = queries.shape

 \_, S, \_ = keys.shape

 H = self.n\_heads

 #Perform input transposing to align with CNN attributions namely, batch, channels, length

 queries = queries.transpose(1, 2)

 keys = keys.transpose(1, 2)

 values = values.transpose(1, 2)

 queries = self.cnn\_queries(queries).transpose(1, 2)

 keys = self.cnn\_keys(keys).transpose(1, 2)

 values = self.cnn\_values(values).transpose(1, 2)

 queries = self.query\_projection(queries).view(B, L, H, -1)

 keys = self.key\_projection(keys).view(B, S, H, -1)

 values = self.value\_projection(values).view(B, S, H, -1)

 #

 out, attn = self.inner\_correlation(

 queries,

 keys,

 values,

 attn\_mask

 )

 out = out.view(B, L, -1)

 return self.out\_projection(out), attn

class TokenEmbedding(nn.Module):

 def \_\_init\_\_(self, c\_in, d\_model):

 super(TokenEmbedding, self).\_\_init\_\_()

 padding = 1 if version\_inspection(torch.\_\_version\_\_, '1.5.0') else 2

 self.tokenConv = nn.Conv1d(in\_channels=c\_in, out\_channels=d\_model,

 kernel\_size=3, padding=padding, padding\_mode='circular', bias=False)

 for m in self.modules():

 if isinstance(m, nn.Conv1d):

 nn.init.kaiming\_normal\_(m.weight, mode='fan\_in', nonlinearity='leaky\_relu')

 def forward(self, x):

 x = self.tokenConv(x.permute(0, 2, 1)).transpose(1, 2)

 return x

class FixedEmbedding(nn.Module):

 def \_\_init\_\_(self, c\_in, d\_model):

 super(FixedEmbedding, self).\_\_init\_\_()

 w = torch.zeros(c\_in, d\_model).float()

 w.require\_grad = False

 position = torch.arange(0, c\_in).float().unsqueeze(1)

 div\_term = (torch.arange(0, d\_model, 2).float() \* -(math.log(10000.0) / d\_model)).exp()

 w[:, 0::2] = torch.sin(position \* div\_term)

 w[:, 1::2] = torch.cos(position \* div\_term)

 self.emb = nn.Embedding(c\_in, d\_model)

 self.emb.weight = nn.Parameter(w, requires\_grad=False)

 def forward(self, x):

 return self.emb(x).detach()

class TemporalEmbedding(nn.Module):

 def \_\_init\_\_(self, d\_model, embed\_type='fixed', freq='h'):

 super(TemporalEmbedding, self).\_\_init\_\_()

 minute\_size = 4

 hour\_size = 24

 weekday\_size = 7

 day\_size = 32

 month\_size = 13

 Embed = FixedEmbedding if embed\_type == 'fixed' else nn.Embedding

 if freq == 't':

 self.minute\_embed = Embed(minute\_size, d\_model)

 self.hour\_embed = Embed(hour\_size, d\_model)

 self.weekday\_embed = Embed(weekday\_size, d\_model)

 self.day\_embed = Embed(day\_size, d\_model)

 self.month\_embed = Embed(month\_size, d\_model)

 def forward(self, x):

 x = x.long()

 minute\_x = self.minute\_embed(x[:, :, 4]) if hasattr(self, 'minute\_embed') else 0.

 hour\_x = self.hour\_embed(x[:, :, 3])

 weekday\_x = self.weekday\_embed(x[:, :, 2])

 day\_x = self.day\_embed(x[:, :, 1])

 month\_x = self.month\_embed(x[:, :, 0])

 return hour\_x + weekday\_x + day\_x + month\_x + minute\_x

class TimeFeatureEmbedding(nn.Module):

 def \_\_init\_\_(self, d\_model, embed\_type='timeF', freq='h'):

 super(TimeFeatureEmbedding, self).\_\_init\_\_()

 freq\_map = {'h': 4, 't': 5, 's': 6, 'm': 1, 'a': 1, 'w': 2, 'd': 3, 'b': 3}

 d\_inp = freq\_map[freq]

 self.embed = nn.Linear(d\_inp, d\_model, bias=False)

 def forward(self, x):

 return self.embed(x)

class DataEmbedding(nn.Module):

 def \_\_init\_\_(self, c\_in, d\_model, embed\_type='fixed', freq='h', dropout=0.1):

 super(DataEmbedding, self).\_\_init\_\_()

 self.value\_embedding = TokenEmbedding(c\_in=c\_in, d\_model=d\_model)

 self.temporal\_embedding = TemporalEmbedding(d\_model=d\_model, embed\_type=embed\_type,

 freq=freq) if embed\_type != 'timeF' else TimeFeatureEmbedding(

 d\_model=d\_model, embed\_type=embed\_type, freq=freq)

 self.dropout = nn.Dropout(p=dropout)

 def forward(self, x, x\_mark):

 x = self.value\_embedding(x) + self.temporal\_embedding(x\_mark)

 return self.dropout(x)

#-----------------------------------------CNNCorrelationBasedTransformer.py--------------------------------#

1.3.12 Generation of Execute.ipynb

This code implements a Time Series Forecasting system using a CNN Correlation-based Transformer model. The code is structured in two main parts that should be copied into separate cells in your Jupyter notebook:

Cell [1]: Contains declarations.

Cell [2]: Contains the necessary imports including PyTorch, NumPy, and custom utilities. This section sets up the basic framework for the implementation.

Cell [3]: Contains the complete configuration setup, including argument parsing and parameter initialization. It defines all hyperparameters for the model architecture, data loading, training process, and GPU usage. The code uses argparse to create a flexible command-line interface, though parameters are pre-set in the script.

Key Points about Transfer Learning Implementation:

1. The code includes commented-out sections (marked with #) that implement transfer learning.
2. To activate transfer learning:
	* Uncomment the lines starting with #import CNNCorrelationBasedTransformer.
	* Uncomment the model loading section to load a pre-trained model from checkpoints.
	* Uncomment the encoder freezing section to prevent updates to the encoder parameters.

Storage Organization:

* Checkpoints folder (./checkpoints/): Stores model parameters and configuration settings in .pth files.
* Results folder (./results/): Contains the output metrics, predictions, and performance evaluations.

To reproduce:

1. Create a new Jupyter notebook.
2. Copy everything from the import section (Cell [1]) into your first cell.
3. Copy the entire configuration and execution code (Cell [2]) into your second cell.
4. Ensure you have the required data files in the specified ./data/ directory.
5. Save the Jupyter notebook as **“Execute.ipynb.”**
6. Run the cells in order.

The model will automatically handle training, testing, and prediction phases. Training configurations and model states will be saved in the checkpoints folder, while all performance metrics and prediction results will be stored in the results folder for further analysis.

#--------------------------------------------------------Execute.ipynb-------------------------------------------#

Cell [1]

#The original code developed by Haixu Wu, available on GitHub, was adapted by the authors, with Ling Feng leading the adaptation, to meet the requirements of the experiments.

#All modifications to the original code were made in accordance with the MIT License below:

#MIT License

#Copyright (c) 2021 THUML @ Tsinghua University

#Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated

#documentation files (the \"Software\"), to deal in the Software without restriction, including without limitation

#the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software,

#and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

#The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software

#THE SOFTWARE IS PROVIDED \"AS IS\", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO

#THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS

#OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR

#OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE."

Cell [2]

import argparse

import random

import os

import matplotlib.pyplot as plt

import numpy as np

import torch

from utils.tools import dotdict

from exp.exp\_main import Exp\_Main

import sys

Cell [3]

fix\_seed = 2021

random.seed(fix\_seed)

torch.manual\_seed(fix\_seed)

np.random.seed(fix\_seed)

parser = argparse.ArgumentParser(description='Time Series Forecasting Using CNN Correlation-based Transformer')

# basic config

parser.add\_argument('--is\_training', type=int, required=True, default=1, help='status')

parser.add\_argument('--model\_id', type=str, required=True, default='test', help='model id')

parser.add\_argument('--model', type=str, required=True, default='CNNCorrelationBasedTransformer',

 help='model name, there is no option for this, only CNNCorrelationBasedTransformer available')

# data loader

parser.add\_argument('--data', type=str, required=True, default='custom', help='dataset type')

parser.add\_argument('--root\_path', type=str, default='./data/', help='root path of the data file')

parser.add\_argument('--data\_path', type=str, default='set2024.csv', help='data file')

parser.add\_argument('--features', type=str, default='MS',

 help='forecasting task, options:[M, S, MS]; M:multivariate predict multivariate, S:univariate predict univariate, MS:multivariate predict univariate')

parser.add\_argument('--target', type=str, default='OT', help='target feature in S or MS task')

parser.add\_argument('--freq', type=str, default='d',

 help='freq for time features encoding, options:[s:secondly, t:minutely, h:hourly, d:daily, b:business days, w:weekly, m:monthly], you can also use more detailed freq like 15min or 3h')

parser.add\_argument('--checkpoints', type=str, default='./checkpoints/', help='location of model checkpoints')

# forecasting task

parser.add\_argument('--seq\_len', type=int, default=10, help='input sequence length')

parser.add\_argument('--label\_len', type=int, default=10, help='start token length')

parser.add\_argument('--pred\_len', type=int, default=5, help='prediction sequence length')

# model define

parser.add\_argument('--enc\_in', type=int, default=4, help='encoder input size')

parser.add\_argument('--dec\_in', type=int, default=4, help='decoder input size')

parser.add\_argument('--c\_out', type=int, default=1, help='output size')

parser.add\_argument('--cnn\_out\_channels', type=int, default=512, help='dimension of CNN')

parser.add\_argument('--d\_model', type=int, default=512, help='dimension of model')

parser.add\_argument('--n\_heads', type=int, default=4, help='num of heads')

parser.add\_argument('--e\_layers', type=int, default=2, help='num of encoder layers')

parser.add\_argument('--d\_layers', type=int, default=1, help='num of decoder layers')

parser.add\_argument('--d\_ff', type=int, default=2048, help='dimension of fcn')

parser.add\_argument('--moving\_avg', type=int, default=35, help='window size of moving average')

parser.add\_argument('--factor', type=int, default=4, help='attn factor')

parser.add\_argument('--distil', action='store\_false',

 help='whether to use distilling in encoder, using this argument means not using distilling',

 default=True)

parser.add\_argument('--dropout', type=float, default=0.05, help='dropout')

parser.add\_argument('--embed', type=str, default='timeF',

 help='time features encoding, options:[timeF, fixed, learned]')

parser.add\_argument('--activation', type=str, default='gelu', help='activation')

parser.add\_argument('--output\_attention', action='store\_true', help='whether to output attention in encoder')

parser.add\_argument('--do\_predict', action='store\_true', help='whether to predict unseen future data')

# optimization

parser.add\_argument('--num\_workers', type=int, default=0, help='data loader num workers')

parser.add\_argument('--itr', type=int, default=2, help='experiments times')

parser.add\_argument('--train\_epochs', type=int, default=30, help='train epochs')

parser.add\_argument('--batch\_size', type=int, default=8, help='batch size of train input data')

parser.add\_argument('--patience', type=int, default=20, help='early stopping patience')

parser.add\_argument('--learning\_rate', type=float, default=0.0001, help='optimizer learning rate')

parser.add\_argument('--des', type=str, default='test', help='exp description')

parser.add\_argument('--loss', type=str, default='mse', help='loss function')

parser.add\_argument('--lradj', type=str, default='type1', help='adjust learning rate')

parser.add\_argument('--use\_amp', action='store\_true', help='use automatic mixed precision training', default=False)

# GPU

parser.add\_argument('--use\_gpu', type=bool, default=True, help='use gpu')

parser.add\_argument('--gpu', type=int, default=0, help='gpu')

parser.add\_argument('--use\_multi\_gpu', action='store\_true', help='use multiple gpus', default=False)

parser.add\_argument('--devices', type=str, default='0,1,2,3', help='device ids of multile gpus')

sys.argv = ['TSF Using CBT', '--is\_training', '1', '--model\_id', 'test', '--model', 'CNNCorrelationBasedTransformer',

 '--data', 'custom',

 '--root\_path', './data/',

 '--data\_path', 'set2024.csv',

 '--features', 'MS',

 '--target', 'OT',

 '--freq', 'd',

 '--checkpoints', './checkpoints/',

 '--seq\_len', '10',

 '--label\_len', '10',

 '--pred\_len', '5',

 '--enc\_in', '4',

 '--dec\_in', '4',

 '--c\_out', '1',

 '--cnn\_out\_channels', '128',

 '--d\_model', '512',

 '--n\_heads', '4',

 '--e\_layers', '2',

 '--d\_layers', '1',

 '--d\_ff', '2048',

 '--moving\_avg', '35',

 '--factor', '4',

 '--distil',

 '--dropout', '0.05',

 '--embed', 'timeF',

 '--activation', 'gelu',

 '--output\_attention',

 '--do\_predict',

 '--num\_workers', '0',

 '--itr', '2',

 '--train\_epochs', '30',

 '--batch\_size', '8',

 '--patience', '20',

 '--learning\_rate', '0.0001',

 '--des', 'test',

 '--loss', 'mse',

 '--lradj', 'type1',

 '--use\_amp',

 '--use\_gpu', False, #True,

 '--gpu', '0',

 '--use\_multi\_gpu',

 '--devices', '0'

 ]

args = parser.parse\_args()

args.use\_gpu = True if torch.cuda.is\_available() and args.use\_gpu else False

if args.use\_gpu and args.use\_multi\_gpu:

 args.dvices = args.devices.replace(' ', '')

 device\_ids = args.devices.split(',')

 args.device\_ids = [int(id\_) for id\_ in device\_ids]

 args.gpu = args.device\_ids[0]

print('Args in experiment:')

print(args)

### This is where the proposed method of transfer learning takes place. ###

### Once the model is trained/validated/tested to satisfy the requirements, ###

### just remove the symbol "#" in the following lines as indicated below ###

### to perform transfer learning. ###

### The parameter settings of each training/validating/testing are saved in a file ended with ".pth" ###

### in the "checkpoints" folder. Also, the filename is defined the parser section up above. ###

#import CNNCorrelationBasedTransformer ## Remove the pound sign at front ##

# Load pre-trained model

#model = CNNCorrelationBasedTransformer.Model(args) ## Remove the pound sign at front ##

#checkpoint\_path = './checkpoints/**---Folder of Trained Parameter---/**checkpoint.pth' ## Remove the pound sign at front ##

#model.load\_state\_dict(torch.load(checkpoint\_path)) ## Remove the pound sign at front ##

# Freeze encoder layers

#for param in model.encoder.parameters(): ## Remove the pound sign at front ##

# param.requires\_grad = False ## Remove the pound sign at front ##

#for name, param in model.named\_parameters(): ## Remove the pound sign at front ##

# if 'EncoderLayer.0' in name: ## Remove the pound sign at front ##

# param.requires\_grad = False ## Remove the pound sign at front ##

Exp = Exp\_Main

if args.is\_training:

 for ii in range(args.itr):

 # setting record of experiments

 setting = '{}\_{}\_{}\_ft{}\_sl{}\_ll{}\_pl{}\_dcc{}\_dm{}\_nh{}\_el{}\_dl{}\_df{}\_fc{}\_eb{}\_dt{}\_{}\_{}'.format(

 args.model\_id,

 args.model,

 args.data,

 args.features,

 args.seq\_len,

 args.label\_len,

 args.pred\_len,

 args.cnn\_out\_channels,

 args.d\_model,

 args.n\_heads,

 args.e\_layers,

 args.d\_layers,

 args.d\_ff,

 args.factor,

 args.embed,

 args.distil,

 args.des, ii)

 exp = Exp(args) # set experiments

 print('>>>>>>>start training : {}>>>>>>>>>>>>>>>>>>>>>>>>>>'.format(setting))

 exp.train(setting)

 print('>>>>>>>testing : {}<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<'.format(setting))

 exp.test(setting)

 if args.do\_predict:

 print('>>>>>>>predicting : {}<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<'.format(setting))

 exp.predict(setting, True)

 torch.cuda.empty\_cache()

else:

 ii = 0

 setting = '{}\_{}\_{}\_ft{}\_sl{}\_ll{}\_pl{}\_dcc{}\_dm{}\_nh{}\_el{}\_dl{}\_df{}\_fc{}\_eb{}\_dt{}\_{}\_{}'.format(

args.model\_id,

 args.model,

 args.data,

 args.features,

 args.seq\_len,

 args.label\_len,

 args.pred\_len,

 args.cnn\_out\_channels,

 args.d\_model,

 args.n\_heads,

 args.e\_layers,

 args.d\_layers,

 args.d\_ff,

 args.factor,

 args.embed,

 args.distil,

 args.des, ii)

 exp = Exp(args) # set experiments

 print('>>>>>>>testing : {}<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<'.format(setting))

 exp.test(setting, test=1)

 torch.cuda.empty\_cache()

#--------------------------------------------------------Execute.ipynb-------------------------------------------#

**2. IMPLEMENTATION AND EXECUTION OF THE PROPOSED METHOD**

 The proposed transfer learning method consists of two main phases: source domain training and target domain adaptation.

First, in the source domain phase:

1. Input data is fed through both encoder and decoder.
2. After training, validation, and testing, the model parameters are saved as a pre-trained model (.pth file) in the "checkpoints" directory.

Then, for the target domain phase:

1. Load the pre-trained model.
2. Freeze the encoder parameters while keeping the decoder parameters trainable.
3. Process target domain data through both encoder and decoder.
4. Train, validate, and test the model until desired performance is achieved.
5. Save the final target domain model parameters.

To make predictions using either model:

1. Load the corresponding model parameter file.
2. Set the --is\_training parameter to "0" in sys.argv within "Execute.ipynb."
3. To utilize GPU acceleration, set the --use\_gpu parameter to "True."
4. Run "Execute.ipynb."