Appendix

```
Load_data.py
Import os
Import numpy as np
Import csv
Import pandas as pd
Import matplotlib.pyplot as plt
From matplotlib.dates import DateFormatter

Import math
Import warnings
Import warnings

warnings.filterwarnings("ignore")
warnings.filterwarnings("ignore")
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)3
pd.set_option('display.width', 1000)

## import data
# stockName = 'zgpa'   #  Ping An of China 601318
# stockName = 'zgtb'   # China Pacific Insurance 601601
# stockName = 'xhbx'
# New China Insurance 601336
stockName = 'zgrs' # China Life Insurance 601628

df = pd.read_csv('stock/' + stockName + '/' + stockName + '.csv', parse_dates=['date'])
print(df.head())
print(df.tail())
print(df.shape)
print(df.columns)

# Create Apple stock price plot
fig, ax = plt.subplots(figsize=(7, 5), dpi=600)
ax.plot(df['date'], df['close'], label=stockName + 'stock')
ax.set(xlabel="Trade Date",
                    ylabel="Close")
# title=stockName + " Stock Price")
# date_form = DateFormatter("%Y")
# ax.xaxis.set_major_formatter(date_form)
plt.savefig('stock/' + stockName + '/test/' + "Stock Close Price" + '.png', bbox_inches="tight",  pad_inches=0.02)
```

```
plt.show()

# Calculate technical indicators
def get_technical_indicators(data):
“””

“””
print(data.iloc[:, 5],data.iloc[:, 4])
# Create 7 and 21 days Moving Average
data[‘MA7’] = data.iloc[:, 5].rolling(window=7).mean()
data[‘MA21’] = data.iloc[:, 5].rolling(window=21).mean()

# Create MACD
data[‘MACD’] = data.iloc[:, 5].ewm(span=26).mean()- data.iloc[:, 2].ewm(span=12, adjust=False).mean()

# Create Bollinger Bands
data[‘20SD’] = data.iloc[:, 5].rolling(20).std() data[‘upper_band’] = data[‘MA21’] + (data[‘20SD’] * 2) data[‘lower_band’] = data[‘MA21’] - (data[‘20SD’] * 2)

# Create Exponential moving average
data[‘EMA’] = data.iloc[:, 4].ewm(com=0.5).mean()

# Create LogMomentum
data[‘logmomentum’] = np.log(data.iloc[:, 4] - 1)

return data

T_df = get_technical_indicators(df)

# Drop the first 21 rows
# For doing the fourier
dataset = T_df.iloc[20:, :].reset_index(drop=True)

# Getting the Fourier transform features
def get_fourier_transfer(dataset):
# Get the columns for doing fourier

data_FT = dataset[[‘date’, ‘close’]]

close_fft = np.fft.fft(np.asarray(data_FT[‘close’].tolist()))
fft_df = pd.DataFrame({‘fft’: close_fft})
fft_df[‘absolute’] = fft_df[‘fft’].apply(lambda x: np.abs(x))
# Find the modular length of a complex number
```

```
fft_df['angle'] = fft_df['fft'].apply(lambda x: np.angle(x))
# Find the angle corresponding to a complex number

fft_list = np.asarray(fft_df['fft'].tolist())
fft_com_df = pd.DataFrame()
for num_ in [3, 6, 9]:
        fft_list_m10 = np.copy(fft_list);
        fft_list_m10[num_:-num_] = 0
fft_ = np.fft.ifft(fft_list_m10) # Perform the inverse Fourier transform on the Fourier transformed data
        fft_com = pd.DataFrame({'fft': fft_})
        fft_com['absolute of ' + str(num_) + ' comp'] = fft_com['fft'].apply(lambda x: np.abs(x)) #
Find the module length after the inverse Fourier transform
        fft_com['angle of ' + str(num_) + ' comp'] = fft_com['fft'].apply(lambda x: np.angle(x)) #
Find the angle corresponding to the complex number after the inverse Fourier transform
        fft_com = fft_com.drop(columns='fft')
        fft_com_df = pd.concat([fft_com_df, fft_com], axis=1)

return fft_com_df

# Get Fourier features
dataset_F = get_fourier_transfer(dataset)
Final_data = pd.concat([dataset, dataset_F], axis=1)

print(Final_data.head())

Final_data.to_csv("stock/" + stockName + "/" + stockName + "_with_Fourier.csv", index=False)

def plot_technical_indicators(dataset, last_days):
plt.figure(figsize=(16, 10), dpi=100)
shape_0 = dataset.shape[0]
xmacd_ = shape_0 - last_days

dataset = dataset.iloc[-last_days:, :]
x_ = range(3, dataset.shape[0])
x_ = list(dataset.index)

# Plot first subplot
plt.subplot(2, 1, 1)
plt.plot(dataset['MA7'], label='MA 7', color='g', linestyle='--')
plt.plot(dataset['close'], label='Closing Price', color='b')
plt.plot(dataset['MA21'], label='MA 21', color='r', linestyle='--')
plt.plot(dataset['upper_band'], label='Upper Band', color='c')
plt.plot(dataset['lower_band'], label='Lower Band', color='c')
```

```
plt.fill_between(x_, dataset['lower_band'], dataset['upper_band'], alpha=0.35)
plt.title('Technical indicators for Apple - last {} days.'.format(last_days)) plt.ylabel('USD')
plt.legend()

# Plot second subplot
 plt.subplot(2, 1, 2) plt.title('MACD')
plt.plot(dataset['MACD'], label='MACD', linestyle='-.')
plt.hlines(15, xmacd_, shape_0, colors='g', linestyles='--')
plt.hlines(-15, xmacd_, shape_0, colors='g', linestyles='--')
plt.plot(dataset['logmomentum'], label='Momentum', color='b', linestyle='-')

plt.legend()
plt.show()

plot_technical_indicators(T_df, 400)

def plot_Fourier(dataset):
        data_FT = dataset[['date', 'close']]

        close_fft = np.fft.fft(np.asarray(data_FT['close'].tolist()))
        fft_df = pd.DataFrame({'fft': close_fft})
        fft_df['absolute'] = fft_df['fft'].apply(lambda x: np.abs(x))
        fft_df['angle'] = fft_df['fft'].apply(lambda x: np.angle(x))

        fft_list = np.asarray(fft_df['fft'].tolist())
        plt.figure(figsize=(14, 7), dpi=100)
        fft_list = np.asarray(fft_df['fft'].tolist())
        for num_ in [3, 6, 9]:
                fft_list_m10 = np.copy(fft_list);
                fft_list_m10[num_:-num_] = 0
                plt.plot(np.fft.ifft(fft_list_m10), label='Fourier transform with {}
                components'.format(num_)) plt.plot(data_FT['close'], label='Real')
        plt.xlabel('Days') plt.ylabel('Stock Price')
        plt.title(stockName + ' (close) stock prices & Fourier transforms') plt.legend()
        plt.show()

plot_Fourier(dataset)
data_preprocessing.py
import os
import pandas as pd
import numpy as np
import pandas as pd
import statsmodels.api as sm
```

```
from numpy import *
from math import sqrt
from pandas import *
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
from sklearn.preprocessing
import MinMaxScaler from pickle
import dump
import warnings

warnings.filterwarnings("ignore")

pd.set_option('display.max_columns', None)

pd.set_option('display.max_rows', None)
pd.set_option('display.width', 160)

# stockName = 'zgpa' # Ping An of China 601318
# stockName = 'zgtb'   # China Pacific Insurance 601601
# stockName = 'xhbx' # New China Insurance 601336 stockName = 'zgrs'
# China Life Insurance 601628

#          %%      ----------------------------------------      Load      Data

dataset  = pd.read_csv('stock/' + stockName + '/' + stockName +       '_with_Fourier.csv', parse_dates=['date'])
print('Number of data items in the original data set :  {}'.format(len(dataset)))

#          %%      ----------------------------------------      Data      Pre-processing

# Replace 0 by NA
dataset.replace(0, np.nan, inplace=True)
dataset.fillna(0, inplace=True)
# Check NA and fill them
dataset.isnull().sum()
dataset.iloc[:, 2:]  = pd.concat([dataset.iloc[:, 2:].ffill(), dataset.iloc[:, 2:].bfill()]).groupby(level=0).mean()
print(dataset.columns)

# Set the date to datetime data
datetime_series = pd.to_datetime(dataset['date'])
datetime_index =   pd.DatetimeIndex(datetime_series.values)
dataset = dataset.set_index(datetime_index)
dataset = dataset.sort_values(by='date')
dataset = dataset.drop(columns='date')
```

```
dataset = dataset.drop(columns='code')
dataset = dataset.drop(columns='isST')
dataset.to_csv("stock/" + stockName + '/' + stockName + "_dataset.csv", index=False)

# Get features and target
X_value = pd.DataFrame(dataset.iloc[:, :])
y_value = pd.DataFrame(dataset.iloc[:, 3])

# Autocorrelation Check sm.graphics.tsa.plot_acf(y_value.squeeze(), lags=100)
plt.show()

# Normalized the data
X_scaler = MinMaxScaler(feature_range=(-1, 1))
y_scaler = MinMaxScaler(feature_range=(-1, 1))
X_scaler.fit(X_value)
y_scaler.fit(y_value)

X_scale_dataset = X_scaler.fit_transform(X_value) y_scale_dataset = y_scaler.fit_transform(y_value)

dump(X_scaler, open('stock/' + stockName + '/X_scaler.pkl', 'wb'))
dump(y_scaler, open('stock/' + stockName + '/y_scaler.pkl', 'wb'))

# Reshape the data
'''Set the data input steps and output steps,

        we use 30 days data to predict 1 day price here,
        reshape it to (None, input_step, number of features) used for LSTM input'''
n_steps_in = 30
n_features = X_value.shape[1]
n_steps_out = 3

# Get X/y dataset
def get_X_y(X_data, y_data):
        X = list()
        y = list()
        yc = list()

        length = len(X_data)
        for i in range(0, length, 1):
                X_value = X_data[i: i + n_steps_in][:, :]
                y_value = y_data[i + n_steps_in: i + (n_steps_in + n_steps_out)][:, 0]
                yc_value = y_data[i: i + n_steps_in][:, :]
                if len(X_value) == 30 and len(y_value) == 3:
```

```
                X.append(X_value)
                y.append(y_value)
                yc.append(yc_value)
print('After the original data is processed by sliding window : {}, {}, {}'.format(np.array(X).shape, np.array(y).shape, np.array(yc).shape))
return np.array(X), np.array(y), np.array(yc)

# get the train test predict index
def predict_index(dataset, X_train, n_steps_in, n_steps_out):
        # get the predict data (remove the in_steps days)
        # train_predict_index = dataset.iloc[n_steps_in: X_train.shape[0] + n_steps_in + n_steps_out - 1, :].index
        train_predict_index = dataset.iloc[0: X_train.shape[0] + n_steps_out - 1, :].index
        # test_predict_index = dataset.iloc[X_train.shape[0] + n_steps_in:, :].index
        test_predict_index = dataset.iloc[X_train.shape[0] + n_steps_in:, :].index
        print(train_predict_index, test_predict_index)        # 1835+30
        return train_predict_index, test_predict_index

# Split train/test dataset
def split_train_test(data):
        train_size = round(len(X) * 0.7)
        data_train = data[0:train_size]
        data_test = data[train_size:]
        return data_train, data_test

# Get data and check shape
X, y, yc = get_X_y(X_scale_dataset, y_scale_dataset)
X_train, X_test, = split_train_test(X)
y_train, y_test, = split_train_test(y)
yc_train, yc_test, = split_train_test(yc)
train_predict_index, test_predict_index, = predict_index(dataset, X_train, n_steps_in, n_steps_out)

#       %%      ----------------------------------------      Save      dataset

print('X shape: ', X.shape)
print('y shape: ', y.shape)

print('X_train shape: ', X_train.shape)
print('y_train shape: ', y_train.shape)
print('y_c_train shape: ', yc_train.shape)
print('X_test shape: ', X_test.shape)
print('y_test shape: ', y_test.shape)
print('y_c_test shape: ', yc_test.shape)
print('train_predict_index shape:', train_predict_index.shape)
```

```
print('test_predict_index shape:', test_predict_index.shape)

np.save("stock/" + stockName + "/X_train.npy", X_train)
np.save("stock/" + stockName + "/y_train.npy", y_train)
np.save("stock/" + stockName + "/X_test.npy", X_test)
np.save("stock/" + stockName + "/y_test.npy", y_test)
np.save("stock/" + stockName + "/yc_train.npy", yc_train)
np.save("stock/" + stockName + "/yc_test.npy", yc_test)
np.save('stock/' + stockName + '/train_predict_index.npy', train_predict_index)
np.save('stock/' + stockName + '/test_predict_index.npy', test_predict_index)
CAL_WGAN_GP.py
import os
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from sklearn import metrics from pickle
import load
from tensorflow.keras.losses import
mean_squared_error
from tensorflow.keras.layers
import GRU, Dense, Flatten, Dropout, Conv1D, BatchNormalization,
LeakyReLU, ELU, ReLU, \
        Bidirectional, AveragePooling1D
from tensorflow.keras import Sequential, regularizers
from tensorflow.python.client import device_lib import matplotlib
from tensorflow.keras import Input
from tensorflow.keras.layers import Dense, LSTM
from tensorflow.python.keras.layers import Permute, multiply, RepeatVector, Multiply, Reshape,
GlobalAveragePooling1D

from tensorflow.keras.models import load_model, Model
from tensorflow.keras import backend as K
from tensorflow.keras.layers
import Dense, Lambda, Dot, Activation, Concatenate, Layer from tensorflow.keras.utils
import plot_modelfrom IPython.display
import Image # Draw the topology diagram of the network model
from tensorflow.keras.utilsimport plot_model
from IPython.display import Image # Draw the topology diagram of the network model

stockName = 'zgpa'

matplotlib.rc("font", family='Microsoft YaHei')
```

```
gpus = tf.config.experimental.list_physical_devices("GPU")
if gpus:
        tf.config.experimental.set_memory_growth(gpus[0], True)
        tf.config.set_visible_devices([gpus[0]], "GPU")

# Load data
X_train = np.load("stock/" + stockName + "/X_train.npy", allow_pickle=True)
y_train = np.load("stock/" + stockName + "/y_train.npy", allow_pickle=True)
X_test = np.load("stock/" + stockName + "/X_test.npy", allow_pickle=True)

y_test = np.load("stock/" + stockName + "/y_test.npy", allow_pickle=True)
 yc_train = np.load("stock/" + stockName + "/yc_train.npy", allow_pickle=True)
 yc_test = np.load("stock/" + stockName + "/yc_test.npy", allow_pickle=True)

def evaluation_metric(y_test, y_hat):
        MSE = metrics.mean_squared_error(y_test, y_hat)
        MAPE = np.mean(np.abs((y_hat - y_test) / y_test)) * 100 RMSE = MSE ** 0.5
        MAE = metrics.mean_absolute_error(y_test, y_hat)
        R2 = metrics.r2_score(y_test, y_hat)
        print('MAPE: %.5f' % MAPE)
        print('RMSE: %.5f' % RMSE)
        print('MAE: %.5f' % MAE)
        print('R2: %.5f' % R2)

def attention_3d_block(inputs, single_attention_vector=False):
        # inputs.shape = (batch_size, time_steps, input_dim) time_steps = K.int_shape(inputs)[1]
        input_dim = K.int_shape(inputs)[2] a = Permute((2, 1))(inputs)
        a = Reshape((input_dim, time_steps))(a)
        # a = Dense(time_steps)(a)
        a = Dense(time_steps, activation='softmax')(a) a = Dropout(0.01)(a)
        # a = Dense(time_steps, activation='softmax')(a)
if single_attention_vector:
        a = Lambda(lambda x: K.mean(x, axis
        =1))(a)
        a = RepeatVector(input_dim)(a)

a_probs = Permute((2, 1))(a)
# element-wise
output_attention_mul = Multiply()([inputs, a_probs])
return output_attention_mul


# Define the generator
```

```
def Generator(input_dim, output_dim, feature_size):
model_input = Input(shape=(input_dim, feature_size))
x = Conv1D(filters=64, kernel_size=6, activation=LeakyReLU(alpha=0.01), padding='same')( model_input)  # padding = 'same'
x = AveragePooling1D(2, 1)(x)
x = BatchNormalization()(x)
x = attention_3d_block(x) x = Bidirectional(
        LSTM(150, return_sequences=True, recurrent_regularizer=regularizers.l2(1e-3)))( x)
x = Dropout(0.22)(x)
x = Dense(101)(x)
x = Flatten()(x)
x = Dense(output_dim, activation='linear')(x)
model = Model(model_input, x)
return model

# Define the discriminator

def Discriminator():
        model = tf.keras.Sequential(name='discriminator')
        model.add(
                Conv1D(64, input_shape=(33, 1), kernel_size=3, strides=2, padding="same", activation=LeakyReLU(alpha=0.01)))
        model.add(AveragePooling1D(2, 1))
        model.add(BatchNormalization())
        model.add(Conv1D(128, kernel_size=3, strides=2, padding="same", activation=LeakyReLU(alpha=0.01)))
        model.add(AveragePooling1D(2, 1))
        model.add(BatchNormalization())
        model.add(Conv1D(256, kernel_size=3, strides=2, padding="same", activation=LeakyReLU(alpha=0.01)))
        model.add(AveragePooling1D(2, 1))
        model.add(BatchNormalization())
        model.add(Flatten())
        model.add(Dense(220, use_bias=True, activation=LeakyReLU()))
        model.add(Dense(220, use_bias=True, activation=ReLU()))
        # model.add(GlobalAveragePooling1D())
        #model.add(Activation('softmax'))
        model.add(Dense(1))
        return model

# Train WGAN-GP model
class GAN():
def init (self, generator, discriminator):
        super(GAN, self). init ()
```

```
        self.d_optimizer = tf.keras.optimizers.Adam(0.0004)
        self.g_optimizer = tf.keras.optimizers.Adam(0.0001)
        self.generator = generator
        self.discriminator = discriminator
        self.batch_size = 128
        checkpoint_dir = '../training_checkpoints'
        self.checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
        self.checkpoint = tf.train.Checkpoint(generator_optimizer=
        self.g_optimizer,
                                              discriminator_optimizer=self.d_optimizer,
                                              generator=self.generator,
                                              discriminator=self.discriminator)
```

Used to constrain the weight of the discriminator, the gradient penalty can not only ensure that the 1-L condition is met, but also ensure that the weight changes are not so drastic (gradient disappearance or gradient explosion)

```
        def gradient_penalty(self, batch_size, real_output, fake_output):
        """ Calculates the gradient penalty.

        This loss is calculated on an interpolated image
        and added to the discriminator loss.
        """
```

get the interpolated data

```
        alpha = tf.random.normal([batch_size, 33, 1], 0.0, 1.0)
        diff = fake_output - tf.cast(real_output, tf.float32)
        interpolated = tf.cast(real_output, tf.float32) + alpha * diff
```

```
with tf.GradientTape() as gp_tape:
        gp_tape.watch(interpolated)
        # 1. Get the discriminator output for this interpolated image.
        pred = self.discriminator(interpolated, training=True)
```

```
# 2. Calculate the gradients w.r.t to this interpolated image.
grads = gp_tape.gradient(pred, [interpolated])[0]
```

```
# 3. Calcuate the norm of the gradients
norm = tf.sqrt(tf.reduce_sum(tf.square(grads), axis=[1, 2]))
```

```
gp = tf.reduce_mean((norm - 1.0) ** 2) return gp
```

```
def train_step(self, data):
        real_input, real_price, yc = data
        batch_size = tf.shape(real_input)[0]
```

```
        for _ in range(1):
                with tf.GradientTape() as d_tape:
                        # Train the discriminator
                        # generate fake output
                        generated_data = self.generator(real_input, training=True)
                        # reshape the data
                        generated_data_reshape = tf.reshape(generated_data,
                                                        [generated_data.shape[0],
generated_data.shape[1], 1])
                        fake_output = tf.concat([generated_data_reshape, tf.cast(yc, tf.float32)], axis=1) real_y_reshape =
                        tf.reshape(real_price, [real_price.shape[0], real_price.shape[1], 1])
                        real_output = tf.concat([tf.cast(real_y_reshape, tf.float32), tf.cast(yc, tf.float32)],
 axis=1)

                        # Get the logits for the fake images
                        D_real = self.discriminator(real_output, training=True)
                        # Get the logits for real images
                        D_fake = self.discriminator(fake_output, training=True)
                        # Calculate discriminator loss using fake and real logits
                        real_loss = tf.cast(tf.reduce_mean(D_real), tf.float32)
                        fake_loss = tf.cast(tf.reduce_mean(D_fake), tf.float32)
                        d_cost = fake_loss - real_loss
                        # Calculate the gradientjiu penalty
                        gp = self.gradient_penalty(batch_size, real_output, fake_output)
                        # Add the gradient penalty to the original discriminator loss
                        d_loss = d_cost + gp * 10.0

        d_grads = d_tape.gradient(d_loss, self.discriminator.trainable_variables)
        self.d_optimizer.apply_gradients(zip(d_grads, self.discriminator.trainable_variables))
for _ in range(3):
        with tf.GradientTape() as g_tape:
                # Train the generator
                # generate fake output
                generated_data = self.generator(real_input, training=True)
                # reshape the data
                generated_data_reshape = tf.reshape(generated_data,
                                                        [generated_data.shape[0],

generated_data.shape[1], 1])
                fake_output = tf.concat([generated_data_reshape, tf.cast(yc, tf.float32)], axis=1)
                # Get the discriminator logits for fake images
                G_fake = self.discriminator(fake_output, training=True)
                # Calculate the generator loss
```

```
            g_loss = -tf.reduce_mean(G_fake)
            g_grads = g_tape.gradient(g_loss, self.generator.trainable_variables)
            self.g_optimizer.apply_gradients(zip(g_grads, self.generator.trainable_variables))
            return real_price, generated_data, {'d_loss': d_loss, 'g_loss': g_loss}

def train(self, X_train, y_train, yc, epochs):
data = X_train, y_train, yc
train_hist = {}
train_hist['D_losses'] = []
train_hist['G_losses'] = []

for epoch in range(epochs):

        real_price, fake_price, loss = self.train_step(data)

        G_losses = []
        D_losses = []

Real_price = [] Predicted_price = []

        D_losses.append(loss['d_loss'].numpy())
        G_losses.append(loss['g_loss'].numpy())

        Predicted_price.append(fake_price)
        Real_price.append(real_price)

# Save the model every 15 epochs
if (epoch + 1) % 10 == 0:
        tf.keras.models.save_model(generator,
'stock/model/final/ACL-WGAN-GP_%d.h5' % epoch)
        # tf.keras.models.save_model(generator, 'gen_GRU_model_%d.h5' % epoch)
        # self.checkpoint.save(file_prefix=self.checkpoint_prefix)
        print('epoch', epoch + 1, 'd_loss', loss['d_loss'].numpy(), 'g_loss', loss['g_loss'].numpy())

        # For printing loss
            train_hist['D_losses'].append(D_losses)
            train_hist['G_losses'].append(G_losses)

        # Reshape the predicted result & real
        Predicted_price = np.array(Predicted_price)
        Predicted_price  = Predicted_price.reshape(Predicted_price.shape[1],
        Predicted_price.shape[2])
        Real_price = np.array(Real_price)
```

```
        Real_price = Real_price.reshape(Real_price.shape[1], Real_price.shape[2])

        # Plot the loss
        plt.plot(train_hist['D_losses'], label='D_loss')
        plt.plot(train_hist['G_losses'], label='G_loss')
        plt.xlabel('training times')
        plt.ylabel('loss') plt.legend()
        # plt.title("ACL-WGAN-GP " + stockName + "training loss", fontsize=20)
        plt.savefig('stock/' + stockName + '/loss/ACL-WGAN-GP_train_loss2.png',
bbox_inches="tight", pad_inches=0.02) plt.show()

        print("Real stock price data shape", Real_price.shape)
        print(Real_price)
        print("Predicting stock price data shapes", Predicted_price.shape)
        print(Predicted_price)

        return Predicted_price, Real_price, np.sqrt(mean_squared_error(Real_price, Predicted_price))
/ np.mean(
        Real_price)

if  name  == ' main ':
        input_dim = X_train.shape[1] feature_size = X_train.shape[2]
        output_dim = y_train.shape[1]
        epoch = 230

        generator = Generator(X_train.shape[1], output_dim, X_train.shape[2])
        print(generator.summary())
        plot_model(generator, to_file="modelPng/CAL_WGAN_GP_generator.png") Image("modelPng/CAL_WGAN_GP_gen-
erator.png")
        discriminator = Discriminator()
        print(discriminator.summary())
        plot_model(discriminator, to_file="modelPng/CAL_WGAN_GP_discriminator.png") Image("modelPng/CAL_
WGAN_GP_discriminator.png")
        gan = GAN(generator, discriminator)
        Predicted_price, Real_price, RMSPE = gan.train(X_train, y_train, yc_train, epoch) Predicted_price_test, Real_price_test,
        RMSPE_test = gan.train(X_test, y_test, yc_test, epoch)

# %%----------------------------------------------Plot the result ----------------------------------------------------

# Rescale back the real dataset
X_scaler = load(open('stock/' + stockName + '/X_scaler.pkl', 'rb'))
y_scaler = load(open('stock/' + stockName + '/y_scaler.pkl', 'rb'))
train_predict_index = np.load("stock/" + stockName + "/train_predict_index.npy", allow_pickle=True) test_predict_index =
```

```
np.load("stock/" + stockName + "/test_predict_index.npy", allow_pickle=True)

rescaled_Real_price = y_scaler.inverse_transform(Real_price)
rescaled_Real_price_test = y_scaler.inverse_transform(Real_price_test)
rescaled_Predicted_price = y_scaler.inverse_transform(Predicted_price)
rescaled_Predicted_price_test = y_scaler.inverse_transform(Predicted_price_test)

# print("----- rescaled predicted price        ", rescaled_Predicted_price)
# print("----- rescaled predicted price test ", rescaled_Predicted_price_test)
# print("----- SHAPE rescaled predicted price          ", rescaled_Predicted_price.shape)
# print("----- SHAPE rescaled predicted price test   ", rescaled_Predicted_price_test.shape)

#          %%       ---------------------------------------- Plot the TRAIN result----------------------------

predict_result_train = pd.DataFrame()
for i in range(rescaled_Predicted_price.shape[0]):
        y_predict_train = pd.DataFrame(rescaled_Predicted_price[i], columns=["predicted_price"],
                                       index=train_predict_index[i:i + output_dim])
        predict_result_train = pd.concat([predict_result_train, y_predict_train], axis=1, sort=False)

real_price = pd.DataFrame()
for i in range(rescaled_Real_price.shape[0]):
        y_train   =          pd.DataFrame(rescaled_Real_price[i],       columns=["real_price"], index=train_predict_index[i:i + out-
put_dim])
        real_price = pd.concat([real_price, y_train], axis=1, sort=False)

predict_result_train['predicted_train_mean'] = predict_result_train.mean(axis=1)
real_price['real_train_mean'] = real_price.mean(axis=1)

# Plot the predicted result
plt.figure(figsize=(16, 8), dpi=600)
plt.plot(real_price["real_train_mean"])
plt.plot(predict_result_train["predicted_train_mean"], color='r')
plt.xlabel("date")
plt.ylabel(stockName + "stock price")
plt.legend(("real stock price", "Predict stock price"), loc="upper left", fontsize=16) plt.title("ACL_WGAN_GP" + stockName +
"Training set results", fontsize=20)
plt.savefig('stock/' + stockName + '/train' + "/ACL_WGAN_GP model training set prediction result 2"
+ '.png', bbox_inches="tight",
        pad_inches=0.02)
plt.show()
print('ACL_WGAN_GP' + stockName + 'Training set error index') evaluation_metric(predict_result_train["predicted_train_mean"],
real_price["real_train_mean"])
```

```
# %% ---------------------------------------- Plot the TEST result

predict_result_test = pd.DataFrame()
for i in range(rescaled_Predicted_price_test.shape[0]):
    y_predict_test = pd.DataFrame(rescaled_Predicted_price_test[i], columns=["predicted_price"],
        index=test_predict_index[i:i + output_dim]) predict_result_test = pd.concat([predict_result_test, y_predict_test],
    axis=1, sort=False)
real_price_test = pd.DataFrame()
for i in range(rescaled_Real_price_test.shape[0]):
y_test = pd.DataFrame(rescaled_Real_price_test[i], columns=["real_price"],
index=test_predict_index[i:i + output_dim]) real_price_test = pd.concat([real_price_test, y_test], axis=1, sort=False)

predict_result_test['predicted_test_mean'] = predict_result_test.mean(axis=1) real_price_test['real_test_mean'] = real_price_test.
mean(axis=1)

# Plot the predicted result
plt.figure(figsize=(16, 8), dpi=600)
plt.plot(real_price_test["real_test_mean"])
plt.plot(predict_result_test["predicted_test_mean"], color='r')
plt.xlabel("time/month")
plt.ylabel(stockName + "Closing price/yuan")
plt.legend(("real stock price", "Predict stock price"), loc="upper left", fontsize=16) plt.title("ACL_WGAN_GP" + stockName + "Test
set results", fontsize=20)
plt.savefig('stock/' + stockName + '/png' + "/ACL_WGAN_GP model test set prediction result 2" + '.png', bbox_inches="tight",
pad_inches=0.02)
plt.show()
ACL_WGAN_GP = pd.merge(left=real_price_test["real_test_mean"],
        right=predict_result_test["predicted_test_mean"], left_index=True,
        right_index=True)
ACL_WGAN_GP.to_csv('stock/' + stockName + '/prediction' + '/ACL_WGAN_GP2.csv') print('ACL_WGAN_GP' + stockName + 'Test
set error index') evaluation_metric(predict_result_test["predicted_test_mean"], real_price_test["real_test_mean"])
```