National College of Ireland

Con guration Manual

MSc Research Project
MSc. In Data Analytics

Rushabh Parmar
Student ID: x21187240

School of Computing
National College of Ireland

Supervisor: Dr. Ahmed Makki

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Rushabh Parmar
Student ID:	x21187240
Programme:	MSc. In Data Analytics
Year:	2023
Module:	MSc Research Project
Supervisor:	Dr. Ahmed Makki
Submission Due Date:	14/08/2023
Project Title:	Con guration Manual
Word Count:	941
Page Count:	7

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard speci ed in the report template. To use other author s written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	14th August 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not su cient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator o ce must be placed into the assignment box located outside the o ce.

O ce Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Rushabh Parmar
x21187240

1 Introduction

This manual outlines the necessary steps to replicate the proposed research and attain the intended outcomes. It comprise of the system configuration prerequisites, procedures for gathering and integrating datasets, instructions and code snippets for model implementation, and ultimately, the guidelines and code snippets for evaluating the model's performance.

2 System Configuration

This section mentions the system configuration to be installed and setup to replicate the procedure and run the model.

Operating System	Windows 10
RAM	8GB
Hard Disk	128GB+ SSD
Processor	ntel Core i5 8th gen

This experiments in the project are conducted using Python 3.10, where are the IDE (Integrated Development Environment) used was VS Studio and Google Colab.

3 Dataset Generation

For our experiment, we extract the Open, High, Low, Close and Volume prices from the Yahoo Finance Website of the top 10 Nifty 50 stocks, namely, RELIANCE.NS', 'TCS.NS', 'INFY.NS', 'HINDUNILVR.NS', 'ICICIBANK.NS', 'HDFCBANK.NS', 'BHARTIARTL.NS', 'KOTAKBANK.NS', 'WIPRO.NS', 'LT.NS'.

We fetch the data using the Yahoo Finance Python library, ‘yf’, as seen in the below code snippet

```
def fetch_from_yahoo_finance(self, ticker, start, end):
    df = yf.download(tickers=ticker,
                     start=start,
                     end=None,
                     progress=False,
                     interval='1d')
    return df.fillna(method='bfill')
```

Fig 1. Data Fetching

Further we create Gramian Angular Difference Field (GADF) images from this extracted data, the data is normalized using the Min Max normalization, and different sample range and feature variables are modified in the below code, to generate different type of data representation.

```
def make_dataset(self, df, ticker, kind):
    ohlc_columns = ['Open', 'High', 'Low', 'Close']
    df = self.normalize(df)            # normalize by max value
    w = 24                             # window size
    data = []
    gasf = GramianAngularField(image_size=w, method='difference', sample_range=(0, 1))
    for i in range(len(df)-w-1, -1, -1):
        gasf_images = [gasf.transform([df[c][i:i+w]]) for c in ohlc_columns]
        typical_prices = (df['High'][i:i+w] + df['Low'][i:i+w] + df['Close'][i:i+w]) / 3
        label = self.make_label(typical_prices=typical_prices, next_close_price=df['Close'][i+w])
        combined_gasf_image = np.sum(gasf_images, axis=0)/len(gasf_images)
        # combined_gasf_image = np.vstack(gasf_images)
        norm_close = df['norm_close'][i+w-1]
        original_close = df['original_close'][i+w-1]
        data.append((i, combined_gasf_image, label, norm_close, original_close))
```

Fig 2. GADF Image generation

The images, are then dumped into pickle files, along with their respective labels (generated using the make_label method, using bollinger bands), normalized closed price and original closed price. The data is saved based on train, valid or test, as pre-name, based on the kind.

4 Importing Libraries

We import the necessary python libraries in our system, in order to implement our model. The libraries mentioned below are imported.

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from helper_env import PortfolioEnvironment
from helper_models import BollingerClassifier, PolicyNetwork
from helper_dataset import RLDataset
from collections import deque, namedtuple
import random
import pickle
from tqdm import tqdm
import os
from sklearn.metrics import f1_score
from sklearn.preprocessing import StandardScaler
from torch.utils.data import Dataset, DataLoader
from pyts.image import GramianAngularField

# Definitions of neural network models
import warnings
warnings.filterwarnings('ignore')

# Definitions related to the portfolio environment
import torch.nn.functional as F

# Additional imports for mixed precision
from torch.cuda import amp
from torch import autocast
from torchsummary import summary
```

Fig. 3 libraries to import

6 Model Implementation

We import the nn.Module class from the Pytorch framework library, torch, which is used to implement the neural network models, Convolution Neural Network and the Policy Network. Below are three code snippets, used to implement the CNN, the loss function and forward function, which returns the loss and the logits from the CNN and the implementation of the Policy Network.

```
class BollingerClassifier(nn.Module):
    def __init__(self, num_classes=3): #
        super(BollingerClassifier, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=64, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.  (function) ReLU: Any  out_channels=16, kernel_size=3, stride=1, padding=1)
        self.relu3 = nn.ReLU()
        #self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(16 * 6 * 6, 128)
        self.relu4 = nn.ReLU()
        self.dropout1 = nn.Dropout(0)

        self.fc2 = nn.Linear(128, 32)
        self.relu5 = nn.ReLU()
        self.dropout2 = nn.Dropout(0)

        self.fc3 = nn.Linear(32, num_classes)
```

Fig. 4 Convolutional Neural Network Class

```
def loss_fn(self, outputs, y):
    criterion = nn.CrossEntropyLoss()
    loss = criterion(outputs, y)
    return loss

def forward(self, x, y):
    x = self.pool1(self.relu1(self.conv1(x)))
    x = self.pool2(self.relu2(self.conv2(x)))
    x = self.relu3(self.conv3(x))
    x = x.view(-1, 16 * 6 * 6)
    x = self.relu4(self.fc1(x))
    x = self.relu5(self.fc2(x))
    logits = self.fc3(x)
    loss = self.loss_fn(logits, y)
    return x, logits, loss
```

Fig. 5 Loss and Forward Function

```
#### POLICY NETWORK
class PolicyNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(PolicyNetwork, self).__init__()

        '''
            input_size  : last fully connnected layer x number of assets
            output_size : weights of the assets
        '''

        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size // 4)
        self.fc3 = nn.Linear(hidden_size // 4, output_size)

        self.relu1 = nn.ReLU()
        self.relu2 = nn.ReLU()

    def forward(self, x):
        x = self.relu1(self.fc1(x))
        x = self.relu2(self.fc2(x))
        logits = self.fc3(x)
        weights = torch.sigmoid(logits)
        return weights

if __name__ == "__main__":
    model = BollingerClassifier(num_classes=3)
    print(model)
```

Fig. 6 Policy Network

The model are implemented in the helper_model.py file.

7 Reinforcement Learning Environment Creation

In this section we mention about the part of code, which are used to create the trading environment. The essential code from the helper_agent.py file is being mentioned, while the whole file is overall interdependent for its successful implementation.

The code snippet below mentions the function for selecting the action

```python
def select_action(self, state, norm_state, image, label):
    # Exploration
    if np.random.rand() <= self.epsilon:
        self.explo_count += 1
        weights = np.random.uniform(0, 1, size = self.output_dim)
        trade_signals = np.random.choice([0, 1, 2], size = self.output_dim)       # 0: sell, 1: buy, 2: hold
        weights /= np.sum(weights)
        return weights, trade_signals

    # Greedy
    self.model_count += 1
    #state_tensor = torch.tensor(state, dtype = torch.float32)
    logits, inputs = self.construct_inputs_for_one_sample(norm_state, image, label)
    q_values = self.policy_network(inputs)
    weights = q_values.detach().to('cpu').numpy()
    weights /= np.sum(weights)

    # convert logits to trade signals
    _, trade_signals = torch.max(logits.to('cpu'), dim=1)
    trade_signals = trade_signals.numpy()
    return weights, trade_signals
```

Fig. 7 action_selection

The code snippet below mentions the function for updating the q-network

```python
## UDPATE
def update_q_network(self, batch):
    states = torch.tensor(batch.state, dtype = torch.float32)
    norm_states = torch.tensor(batch.norm_state, dtype = torch.float32)
    weights = torch.tensor(batch.action, dtype = torch.float32)
    next_states = torch.tensor(batch.next_state, dtype = torch.float32)
    next_norm_states = torch.tensor(batch.next_norm_state, dtype = torch.float32)
    rewards = torch.tensor(batch.reward, dtype = torch.float32)
    dones = torch.tensor(batch.done, dtype=torch.float32)

    cls_loss, inputs = self.construct_inputs(norm_states, batch.images, batch.labels)
    q_values = self.policy_network(inputs)

    _, next_inputs = self.construct_inputs(next_norm_states, batch.next_images, batch.next_labels)
    next_q_values = self.target_network(next_inputs)

    targets = rewards + self.gamma * torch.max(next_q_values.to('cpu'), dim = 1)[0]
    targets = targets.unsqueeze(1)
    q_targets = q_values.to('cpu').clone()
    q_targets.scatter_(1, torch.arange(self.output_dim).unsqueeze(0).expand(q_targets.shape[0], -1), weights)

    self.rl_optimizer.zero_grad()
    #self.cls_optimizer.zero_grad()

    loss = self.loss_function(q_values.to(self.config.device), q_targets.to(self.config.device))
    #cls_loss = 0.9 * cls_loss + 0.1 * loss

    #cls_loss.backward(retain_graph=True)
    loss.backward()

    self.rl_optimizer.step()
    #self.cls_optimizer.step()

    return loss.item()
```

Fig 8 updating q_network

Files for implementation

The two python notebooks namely, rl_trainer and trainer, are used to train the CNN and the CNN-DDQN model. The first file ‘trainer’ is a python notebook, which run the code to train the CNN and save the model. We save different models for different data representation, thus we have 18 models saved in the path mentioned.

The below code snippet shows the implementation of the trainer.ipynb file

```
from helper_config import Config
from helper_cnn_trainer import train

config = Config()
config.data_dir = '../data'
config.models_dir = '../models'
config.model_name = '2dcnn_stack_gasf_zero_centered_ohlcv_delete'
config.train_batch_size = 16
config.iters_to_accumlate = 1
config.sample_run = False
config.learning_rate = 1e-5
config.num_epochs = 200
config.save_epoch_wait = 1
config.early_stop_count = 20
config.save_checkpoint = True

config.centered_zero = True
config.model_arch = '3d'
config.train_gasf_image = True

config.cols = ['Open', 'High', 'Low', 'Close', 'Volume']

results = train(config)
```

Fig. 9 trainer

The second file, rl_trainer.ipynb, is a Python Notebook, which runs the saved models and implements the Reinforcement Learning environment. The number of episodes it runs is mentioned as the episodes variable.

The below code snippet shows the implementation of the rl_train.ipynb file

```
from helper_config import Config
from helper_env import PortfolioEnvironment
from helper_agent import DQLAgent
from helper_dataset import RLDataset
cfg = Config()
cfg.data_dir = '../data'
cfg.models_dir = '../models/rl'
cfg.base_model_path = '../models/cnn_predictor/baseline/model.pth'
target_assets = len(cfg.tickers)

### RL / POLICY NETWORK CONFIGURATION
cfg.input_dim       = len(cfg.tickers) * (4)
cfg.output_dim      = l
cfg.learning_rate   = 1
cfg.gamma           = 0.99
cfg.epsilon         = 1.0
cfg.epsilon_decay   = 0.9995
cfg.epsilon_min     = 0.01
cfg.reply_capacity  = 10000
cfg.batch_size      = 32
cfg.hidden_size     = 128
cfg.riskfree_rate   = 0.02
cfg.initial_balance = 1_000_000

prep = RLDataset(cfg)
df_norm_close, df_prices, images, labels = prep.prepare(kind = 'train')

# define environment
env = PortfolioEnvironment(cfg, df_norm_close, df_prices, images, labels, window_size = 2000)

# define agent
agent = DQLAgent(cfg, name = 'baseline')
results = agent.train(env, episodes = 20000)
```

(class) PortfolioEnvironment

Fig 10 rl_trainer

9 Evaluation

The results from the CNN trained models, will be stored, alongside models, in a pickled file named as results.

The train method from helper_cnn_trainer is responsible for saving the resulys of each data representation, making it 18 models, with 18 comparative results, saving in a pickled file.

The snapshot below shows the code of train method, to save the results.

```
def train(config):
    device = config.device

    config.dest_path = os.path.join(config.models_dir, config.model_name)
    os.makedirs(config.dest_path, exist_ok=True)

    # define model
    model = BollingerClassifier(num_classes=3)
    model.to(device)

    # optmizer and scheduler
    optimizer = torch.optim.AdamW(model.parameters(), lr = config.learning_rate, weight_decay=0.01)
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode = 'max', factor=0.5, patience=4)

    # dataloaders
    train_loader, valid_loader, _ = dataloaders(config)

    # Trainer
    results = trainer(config, model, train_loader, valid_loader, optimizer, scheduler)

    ### SAVE RESULTS
    with open(os.path.join(config.dest_path, 'results.pkl'), 'wb') as f:
        pickle.dump(results, f)
```

Fig. 10 results