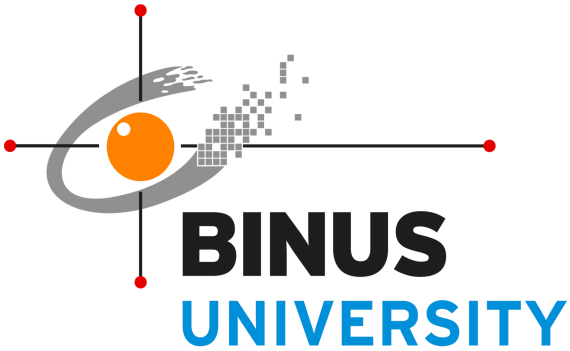
Odd Semester (2023/2024)



BINUS UNIVERSITY INTERNATIONAL

Final Project Report: Pedestrian Movement Prediction Model

Student Information:	Surname:	Given Name:	Student ID Number:
1.	Simrochelle	Chellshe Love	2502043040
2.	Alfin Rizqullah	Muhammad	2502036842
3.	Wirawan	Ian	2502009596
4.	Kelo	Anton Vili Sakari	2702383350

Course Code : COMP6065001

Course Name : Artificial Intelligence

Class : L5BC

Lecturer : Zhandos Yessenbayev, B.Sc., M.Sc., Ph.D

Type of Assignments : Final Project Report

Due Date/Time : 19 December, 2023 / 8:00 P.M.

Problem description

Among the most significant participants in the rapidly evolving field of modern transportation are pedestrians. For these new technologies to succeed, pedestrian safety must be taken into account during the design process. The detection and tracking system, which recognizes and follows pedestrians as they move, is essential to achieving this goal. Nevertheless, since pedestrians are dynamic objects that can abruptly change direction, detection and tracking alone are insufficient. This issue, known as pedestrian trajectory prediction, also arises in other domains, including social robotics and crowd surveillance. For that to succeed, one of the most important underlying technologies is computer vision and motion prediction. We can apply this technology to other things, not only cars, trucks and buses. For example, drones and ships could use motion prediction for different tasks. We decided to look into this technology and develop a model that can be used for real-time motion detection.

Solution features, e.g. algorithms used, special technologies used, user interface, etc.

The model we will be using is the Constant Velocity Model (CVM), developed using data from the ETH Zurich and University of Cyprus trajectory prediction dataset. We chose this model, although it is more complicated than other models. When used in the context of a prediction model for pedestrian movement, it would be a lot more streamlined in terms of application compared to existing models. However, whether it would be better or not is something that we do not know currently. Algorithms used in the solution tend to be quite simple. We are talking about parsers or evaluation algorithms counting displacement for example.

In terms of the framework that would be used for the code, we would decide on the usage of pytorch. The reason is that it is that PyTorch is one of the few machine learning frameworks that was specifically designed for the neural networks on which the Constant Velocity Model can be designed, trained, created, built and run. Along with that, using PyTorch should decrease the loss and increase the accuracy of the end model.

Along with that, we also used quite some Python libraries/ modules. Of course, we have PyTorch with the reasoning explained above. The OS module was also utilised to help with the Operating System level of modification and/ or access. The module named Copy is also used, this is due to the amount of classes and functions that are available and need to be duplicated in other areas of the code. Python module Glob is used to help parse and search files that have

similar file names or patterns. Json is needed to access and read Json files, we require it due to our dataset being in Json format.

Solution design architecture

One of the most important aspects of doing anything that relates to Machine Learning is the ability to find and use a dataset. Sadly, while we found a good dataset to use we had a lot of issues with pre-processing said data. Due to that, we resorted to using a processed dataset that we found on Dropbox.

Raw dataset: https://paperswithcode.com/sota/trajectory-prediction-on-ethucy

Processed dataset:
https://www.dropbox.com/scl/fo/a8ysgnvbva9i608eaiksj/h?rlkey=xt52e1mmct3mli1ykkcrssgg&dl=

The strength of CVM design is that it does not require "training", CVM can be run with or without sampling. A call to the main function generates one sample if the sample option is true. The solution includes five JSON data repositories "**eth_hotel**, **eth_univ**, ucy_univ, ucy_zara01, ucy_zara02". It also features files named; Dataset.py (defines few classes), evaluate.py (defines run confs, eval data), metrics.py (), pedestrian.py.

The PyTorch Dataset class is the ancestor of the class called PedestrianDataset, which is used to organize the program. The JSON files containing the pedestrian trajectory data are used to load the dataset. Initializing the dataset, sorting it according to timestamps, generating sequences, loading detections from files, and producing samples with corresponding timestamps and positions are all done by the program.

Slicing sequences, handling padding for sequences that are shorter than the given length, and computing masks for the sequences are all handled by the PedestrianDataset class. PyTorch tensors are used by the program to handle and manipulate data efficiently. It also specifies how to extract items from the dataset, such as masks and observed trajectories and their corresponding deltas. Reading data from a JSON file called "dataset_info.json" initializes the dataset.

In the world of predicting trajectories, we use metrics to check how well a model's guesses match the real data. Two important ones are Average Displacement Error (ADE) and Final Displacement Error (FDE). ADE calculates the average distance between each point in the predicted and real paths, showing how accurate the model is overall. A smaller ADE means the

model is good at getting the movement details right. It's like a measure of how well the predicted path lines up with the actual one over time. So, ADE helps us see if the model captures the little movements correctly.

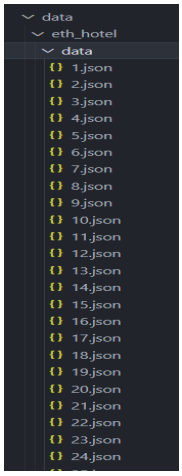
On the flip side, the Final Displacement Error (FDE) zooms in on the very end of trajectories. It calculates the distance between where the predicted path ends and where the real path ends. FDE gives us insights into how well the model predicts the last stop or final point, regardless of how it got there. If the FDE is smaller, it means the model nailed the prediction of the final position, showing it's good at figuring out the last few points of a trajectory. In summary, by looking at both Average Displacement Error (ADE) and FDE, we get a complete picture of how well models can predict trajectories, considering different aspects of accuracy.

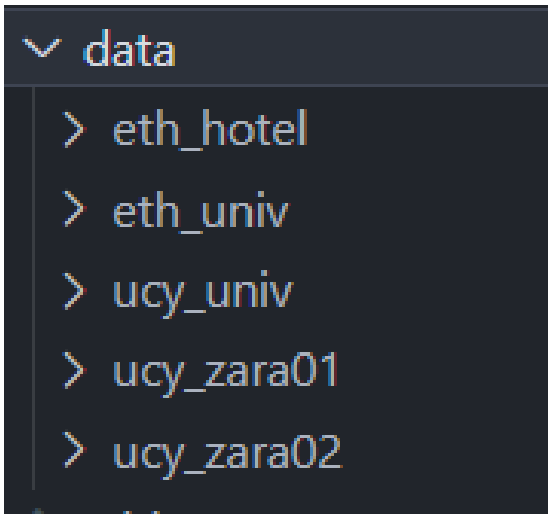
Experiments or tests that you have done.

Using the existing dataset, we have attempted to find the model loss along with finding ADE and FDE.

Program manual (with screenshots)

Datasets



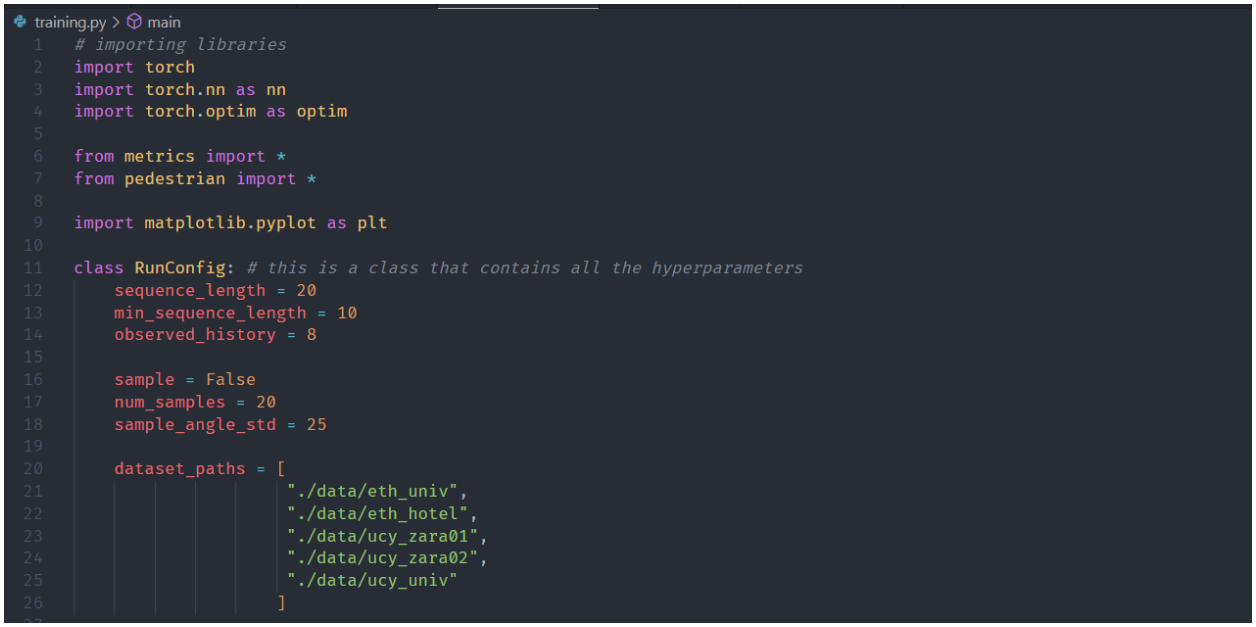


Above is the directory of where we have our dataset at. ETH and UCY are 2 different dataset.

When you proceed to look into the JSON file, you will see that it has identifier and such.

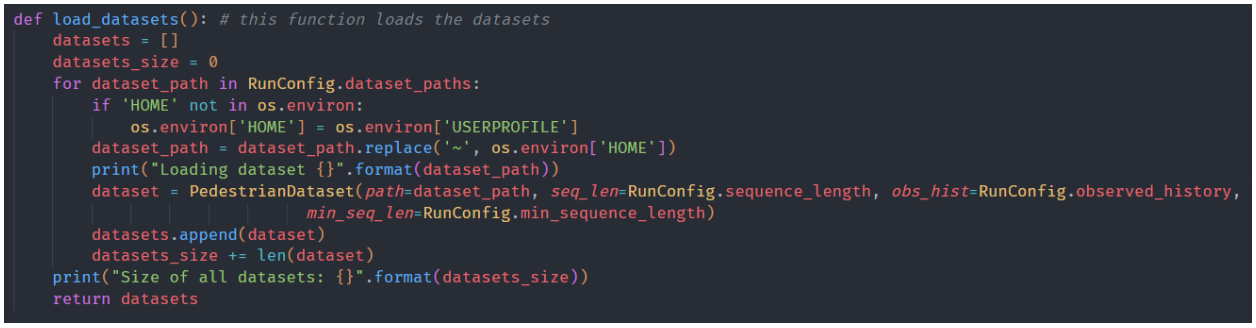
training.py

Starting with training.py, this file is mostly focused on the training and testing of the CVM model itself. This is to create an understanding of a baseline of what our model can do, and it's potential downfall.



```python
# importing libraries
import torch
import torch.nn as nn
import torch.optim as optim

from metrics import *
from pedestrian import *

import matplotlib.pyplot as plt

class RunConfig: # this is a class that contains all the hyperparameters
    sequence_length = 20
    min_sequence_length = 10
    observed_history = 8

    sample = False
    num_samples = 20
    sample_angle_std = 25

    dataset_paths = [
                     "./data/eth_univ",
                     "./data/eth_hotel",
                     "./data/ucy_zara01",
                     "./data/ucy_zara02",
                     "./data/ucy_univ"
                    ]
```

Here we have the imports that we would be using in the file. *Class RunConfig* is created to have a place where we can store the necessary configuration for our model, including the location of all our datasets.



```python
def load_datasets(): # this function loads the datasets
    datasets = []
    datasets_size = 0
    for dataset_path in RunConfig.dataset_paths:
        if 'HOME' not in os.environ:
            os.environ['HOME'] = os.environ['USERPROFILE']
        dataset_path = dataset_path.replace('~', os.environ['HOME'])
        print("Loading dataset {}".format(dataset_path))
        dataset = PedestrianDataset(path=dataset_path, seq_len=RunConfig.sequence_length, obs_hist=RunConfig.observed_history,
                                    min_seq_len=RunConfig.min_sequence_length)
        datasets.append(dataset)
        datasets_size += len(dataset)
    print("Size of all datasets: {}".format(datasets_size))
    return datasets
```

This section is designed to load the dataset into the files. Note however about that *if* statement, if you are doing this locally with only a singer user, is typically fine. However, you might run into

errors if the exception of *USERPROFILE* is not mentioned due to the different OS configuration of each machine.

```python
class ConstantVelocityModel(nn.Module): # this is a class that contains the constant velocity model
    def __init__(self, input_size, output_size):
        super(ConstantVelocityModel, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, observed): # this function computes the forward pass
        obs_rel = observed[:, :, 1:] - observed[:, :, :-1]
        deltas = obs_rel[:, :, -1].unsqueeze(1)
        y_pred_rel = deltas.repeat(1, 12, 1)
        return self.linear(y_pred_rel.view(y_pred_rel.size(0), -1))
```

Code to initially prepare the CVM to start training.

```python
def train_cvm_model(model, train_loader, criterion, optimizer, num_epochs=10): # this function trains the CVM model
    loss_val = []

    model.train()

    for epoch in range(num_epochs):
        running_loss = 0.0
        for batch_x, batch_y in train_loader:
            optimizer.zero_grad()

            observed = batch_x.permute(1, 0, 2)
            y_true_rel, masks = batch_y
            y_true_rel = y_true_rel.permute(1, 0, 2)

            y_pred_rel = model(observed)
            loss = criterion(y_pred_rel, y_true_rel)

            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        avg_loss = running_loss / len(train_loader)
        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss}")
        loss_val.append(avg_loss)
    return loss_val
```

This function is used to train the model itself, along with knowing how much loss it will gather from it.

```python
# Use this function to create a DataLoader for your training dataset
def create_train_loader(trainset, batch_size):
    return torch.utils.data.DataLoader(dataset=trainset, batch_size=batch_size, shuffle=True)

# This function plots the graphs for the results of the training
def plot_training_history(train_loss):
    epochs = range(1, len(train_loss) + 1)

    plt.figure(figsize=(15, 7))
    plt.plot(epochs, train_loss, label='Training Loss')
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()

    plt.show()
```

The code that is on top is designed to input and output data rapidly since we have such a large dataset. The one below it is to visualize the loss of the trained model.

```python
def main():
    # Load datasets
    datasets = load_datasets()

    # Choose a dataset for training (modify this part as needed)
    train_dataset = datasets[0]
    train_loader = create_train_loader(train_dataset, batch_size=1)

    #Define Input and output size
    input_size = 12
    output_size = 2
    # Initializing the CVM model, criterion, and optimizer
    cvm_model = ConstantVelocityModel(input_size, output_size)
    criterion = nn.MSELoss()  # You may need to choose an appropriate loss function
    optimizer = optim.Adam(cvm_model.parameters(), lr=0.001)  # Adjust the learning rate as needed

    # Train the CVM model and store the result in a variable
    train_loss = train_cvm_model(cvm_model, train_loader, criterion, optimizer, num_epochs=10)

    # Plots the loss graph
    plot_training_history(train_loss)
```

The code to run it all. What we found interesting was that, the more sample we give it I.E. sequence length, the more loss we endured compared to single digit. However, no matter how many epochs we did, it was pretty consistent.

dataset.py

```python
import json # for reading json files

class ObjectDetection: # class for storing object detection data
    def __init__(self, id, position):
        self.id = id
        self.position = position

    @staticmethod
    def from_json(json_obj):
        return ObjectDetection(json_obj['id'], json_obj['position'])

class DetectionData: # class for storing detection data
    def __init__(self, timestamp, size, object_list):
        self.timestamp = timestamp
        self.size = size
        self.object_list = object_list

    @staticmethod
    def from_json(json_detection):
        timestamp = json_detection['timestamp']
        size = json_detection['size']
        object_list = [ObjectDetection.from_json(obj) for obj in json_detection['object_list']]
        return DetectionData(timestamp, size, object_list)

    def objects(self):
        return self.object_list

    def __len__(self):
        return len(self.object_list)
```

The code here is for accessing and manipulating the JSON for data. More specifically data that has an identifier and position.

```python
class TrajectoryData: # class for storing trajectory data
    def __init__(self, id, start_time, positions=None):
        self.id = id
        self.start_time = start_time
        self.positions = positions if positions else []

    def add_position(self, position):
        self.positions.append(position)

    def __len__(self):
        return len(self.positions)
```

Created to store the data for possible trajectory

pedestrian.py

```python
# importing libraries
import os
import glob
import json
import copy
import numpy as np
import torch
from torch.utils.data import Dataset
from dataset import *

class PedestrianDataset(Dataset): # class for storing pedestrian dataset
    """
    Dataset of pedestrian trajectories.
    """
    def __init__(self, path, seq_len, obs_hist, min_seq_len):
        super().__init__()
        self.path = path
        self.seq_len = seq_len
        self.min_seq_len = min_seq_len
        self.obs_hist = obs_hist

        self.timestamps, self.paths = None, None
        self.dataset_name = None
        self.data_samples = []
        self.dataset_size = 0

        if self.path is not None:
```

metrics.py

```
import torch # torch 1.0.0

def avg_disp(predicted, true): # average displacement error
    """ Average displacement error. """
    true, masks = true
    sequence_lengths = masks.sum(1)
    batch_size = len(sequence_lengths)
    squared_distance = (true - predicted)**2
    distance = masks * torch.sqrt(squared_distance.sum(2))
    average_distance = (1. / batch_size) * ((1. / sequence_lengths) * distance.sum(1)).sum()
    return average_distance.item()

def final_disp(predicted, true): # final displacement error
    """ Final displacement error """
    true, masks = true
    sequence_lengths = masks.sum(1).type(torch.LongTensor) - 1
    batch_size = len(sequence_lengths)
    squared_distance = (true - predicted)**2
    distances = masks * torch.sqrt(squared_distance.sum(2))
    displacement_sum = distances[:, sequence_lengths].sum()
    average_final_displacement = (1. / batch_size) * displacement_sum
    return average_final_displacement.item()
```

evaluate.py

```
def constant_velocity_model(observed, sample=False): # constant velocity model
    obs_rel = observed[1:] - observed[:-1]
    deltas = obs_rel[-1].unsqueeze(0)
    if sample:
            sampled_angle = np.random.normal(0, RunConfig.sample_angle_std, 1)[0]
            theta = (sampled_angle * np.pi)/ 180.
            c, s = np.cos(theta), np.sin(theta)
            rotation_mat = torch.tensor([[c, s],[-s, c]])
            deltas = torch.t(rotation_mat.matmul(torch.t(deltas.squeeze(dim=0)))).unsqueeze(0)
    y_pred_rel = deltas.repeat(12, 1, 1)
    return y_pred_rel
```

This block of code is for the attempted calculation of the actual prediction model for movement of pedestrian. Calculating it by runnning through the sequence and using the sample data to create a prediction of where the angle might be at. It would then attempt to create a 2D model and should attempt to calculate the possible displacement error.

```python
def evaluate_testset(testset): # evaluate testset
    testset_loader = Data.DataLoader(dataset=testset, batch_size=1, shuffle=True)

    with torch.no_grad():

        avg_disp_list =  []
        final_disp_list = []
        for seq_id, (batch_x, batch_y) in enumerate(testset_loader):

            observed = batch_x.permute(1, 0, 2)
            y_true_rel, masks = batch_y
            y_true_rel = y_true_rel.permute(1, 0, 2)

            sample_avg_disp = []
            sample_final_disp = []
            samples_to_draw = RunConfig.num_samples if RunConfig.sample else 1
            for i in range(samples_to_draw):

                # predict and convert to absolute
                y_pred_rel = constant_velocity_model(observed, sample=RunConfig.sample)
                y_pred_abs = rel_to_abs(y_pred_rel, observed[-1])
                pred_pos = y_pred_abs.permute(1, 0, 2)

                # convert true label to absolute
                y_true_abs = rel_to_abs(y_true_rel, observed[-1])
                real_pos = y_true_abs.permute(1, 0, 2)

                # compute errors
                avg_displacement = avg_disp(pred_pos, [real_pos, masks])
                final_disp = final_disp(pred_pos, [real_pos, masks])
                sample_avg_disp.append(avg_disp)
                sample_final_disp.append(final_disp)
```

6. Link to video of the application demo (with max. length of 2 minutes)

https://drive.google.com/drive/folders/16p3mErtQLaElVdshMhvh1BTSXHe5PLwn?usp=drive_link

7. Link to your GIT repository

https://github.com/alfinrz/Artifical-intelligence-final-project.git