Report for exercise Final from group B

Tasks addressed:	5
Authors:	HORIA TURCUMAN (03752553) GEORGI HRUSANOV (03714895) UPPILI SRINIVASAN (03734253)
Last compiled:	2023–02–09
Source code:	https://gitlab.com/turcumanhoria/crowd-modeling

The work on tasks was divided in the following way:

HORIA TURCUMAN (03752553)	Task 1	33.3%
	Task 2	33.3%
	Task 3	33.3%
	Task 4	33.3%
	Task 5	33.3%
GEORGI HRUSANOV (03714895)	Task 1	33.3%
	Task 2	33.3%
	Task 3	33.3%
	Task 4	33.3%
	Task 5	33.3%
UPPILI SRINIVASAN (03734253)	Task 1	33.3%
	Task 2	33.3%
	Task 3	33.3%
	Task 4	33.3%
	Task 5	33.3%

Report on task 1, Description of the example and summarizing the paper

The forecast of pedestrian speed may have important implications for the design of various transportation systems, as well as the planning of city areas and other problems. The safety of pedestrians in modern cities is a key challenge. The article "Prediction of Pedestrian Speed using Artificial Neural Networks" presents research on the application of neural networks to the challenge of forecasting pedestrian speed. Because real-world data is used, the neural network approach is evaluated, there are opportunities to experiment with the data and the model, and the topic is intriguing and applicable to real-world applications. A work in this are has important practical applications and the potential to improve public space administration and design in ways that benefit society as a whole as well as individuals.

Our final project within the scope of this practical course is based on the paper "Prediction of Pedestrian Speed with Artificial Neural Networks" [3] which is a paper written in the year 2018 which is adressing this topic. The authors of this paper are claiming that the type of facility usually has a considerable impact on pedestrian behavior. As a result, generating solid estimates of pedestrian movements in complex geometries using classic models with few parameters, such as corridors, bottlenecks, or crossings, is difficult. Artificial neural networks can recognize a wide range of patterns and have a variety of parameters to pick from. They have the potential to be an acceptable substitute for predictions. The main goal in this work is to provide the first testing procedures for this methodology. The authors estimate pedestrian speed using both a standard model and a neural network, and then compare the results to those obtained from various combinations of corridor and bottleneck studies. The results show that the neural network can distinguish between the two geometries and improve pedestrian speed prediction even when the geometries are mixed together.

The fundamental diagram (FD) connects the speed and the distance between neighbors or barriers in the immediate area. Humans are viewed in microscopic models as points of a specific size, and their limited physical characteristics can be deduced from the FD. Microscopic models can accurately describe a range of pedestrian motions and occurrences, such as the creation of lanes, despite their simplicity; nonetheless, it is challenging to make predictions in complex spatial structures. The goal of this research is to determine whether neural networks can accurately describe pedestrian behavior in respect to various geographical factors. These are compared using data from trials conducted in corridor/ring and bottleneck scenarios with an FD-based model (Weidmann's Fundamental Diagram).

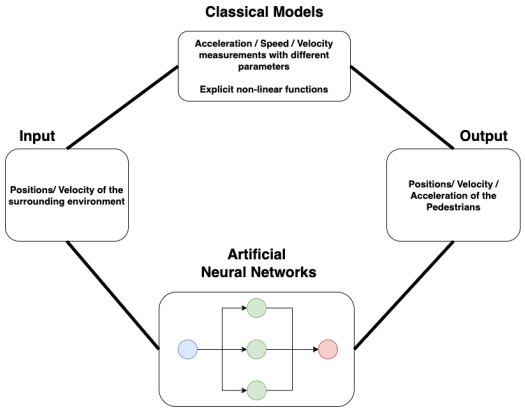

Figure 1: Observations of the pedestrian speeds in original paper [3]

Models The first modeling approach uses the Weidmann Fundamental Diagram, which shows the speed as a continuous scalar value that depends on the mean spacing (that is, the average distance that separates each pedestrian from its K nearest neighbors). Weidmann created this modeling approach. Equation 1 has this information.

$$v = FD(\overline{s}_K, v_0, T, l) = v_0 \left(1 - e^{\frac{l-\overline{s}_K}{v_0 T}}\right) \tag{1}$$

where T is the time gap, l is the pedestrian size, v_0 the desired speed and s_K is the mean spacing, defined as $s_K = \frac{1}{K}\sum_{i=1}^{K}\sqrt{(x-x_i)^2+(y-y_i)^2}$, with (x_i, y_i) denoting the i-th nearest neighbor's position.

The second approach is a fully-connected feed-forward neural network. It takes $2K + 1$ inputs, namely the mean spacing s_K and the K relative positions $(x - x_i, y - y_i)$.

Data To compare the FD-based and ANN modeling methodologies, two datasets collected under laboratory settings are employed. Internet users can access the information [2]. They are a component of the online pedestrian experiment database.

- **The Ring/Corridor (R)**: this geometry is a closed circle that has a length of 30 meters and a width of 1.8 meters. The pedestrian density level in this geometry ranges from 0.25 to 2 ped/m2, which corresponds to a number of participants that may range anywhere from 15 to 230. Because this scenario takes place along a straight line, it is sometimes referred to as a "corridor." The length of the measuring area is 6 meters, and it is located on a straight section.
- **Bottleneck (B)**: this is the next geometry simulated by the dataset, and it may be seen as a corridor whose width abruptly decreases. The width at the beginning is set at 1.8 meters, however the real bottleneck may assume widths of 0.70 meters, 0.95 meters, 1.20 meters, or 1.80 meters (the last number actually nullifies the bottleneck effect due to the fact that the width does not fluctuate). In this particular case, the number of pedestrians will always be 150.

Two different interaction behaviors are described by the two data sets . In a congested system, the speed for a given mean spacing is more likely to be higher in the bottleneck than on the ring. As a result, depending on the geometry, the assessment of the time gap greatly differs.

Fitting the Data With a total of fifty distinct subsamples, the neural network is trained using cross-validation and bootstrap. The network is trained using half of the information while the other half of the data is used to evaluate the network. The back-propagation approach, which involves minimizing the mean square error starting at the top of the network and working your way down, is used to train on the normalized data.

$$\text{MSE} = \frac{1}{N} \sum_i (v_i - \tilde{v}_i)^2 \tag{2}$$

where v_i is the actual speed and $\tilde{v}_i$ is the speed predicted by the NN.

Investigated from the authors of the paper are the several hidden layers, including (1), (2), (3), (4,2), (5,2), (5,3), (6,3), (10,4), and (12,5). The testing error shows a minimum before overfitting begins, although the training error tends to decrease as the network complexity increases, which is expected. The single hidden layer with three nodes, represented by H = (3), achieves this level of reduction.

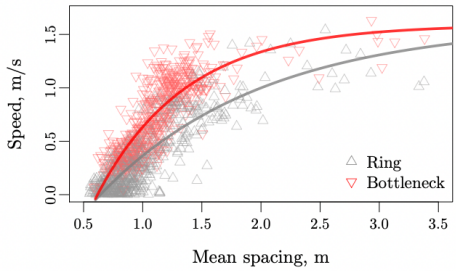

Figure 2: Observations of the pedestrian speeds in original paper [3]

The authors of the paper took the trained neural network with H = 3 and the calibrated FD-model are compared using a variety of data combinations resulting from the ring R and bottleneck B tests. The first argument in the notation ". /." in this context refers to the training phase, whereas the second argument refers to the testing phase. First, identical data sets R/R and B/B were used from the authors to evaluate the modeling approaches. In this aspect, the network performs marginally better than the FD-model. When it comes to the ring experiment, the results are rather comparable, and the utilization of the network helps to

reduce the MSE by about 5%. In the paper the observation was made that the performances, however, are more prone to variation when it comes to the bottleneck, although the benefits are more perceptible (by roughly 15%). The improvement is also significant when dealing with unseen situations, i.e. for the datasets R/B and B/R (around 15%). The best results the authors obtain when training the models on ring and bottleneck trials together, namely the scenarios R/R+B, B/R+B, and R+B/R+B. When faced with a situation like this, the network is able to partially discriminate between the two geometries. The authors compare their findings with some other approaches like the social LSTM neural network and the social force pedestrian model.

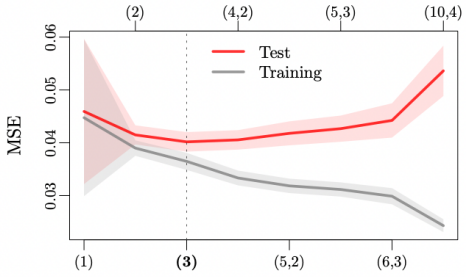

Figure 3: Training and testing errors according to different hidden in the network of original paper [3]

Report on task 2, Create scenarios and gather data

In order to separate our study from the work that has been done previously on predicting the speed of pedestrians, the generation of fake data through the use of the VADERE program is a smart concept. An open-source simulation platform called VADERE offers a setting in which to model and analyze pedestrian movement in a variety of real-world settings. Its full name is the Virtual Environment for the Development of Research in Evolutionary and Evolutionary Algorithms. Working and getting to know VADERE was the main scope of our second exercise sheet within the scope of this pracitical course and we gathered some important knowledge how to utilize and work with VADERE.

VADERE gives us the opportunity to execute large-scale simulations using the main tool it provides. Also it enables the examination of a wide variety of problems connected to pedestrian behavior and crowd behavior. The program is created as a tool for researchers to use in order to evaluate complicated pedestrian dynamics.All of these make VADERE an excellent instrument for creating false data for our final project also, the primary emphasis of which is estimating the pace at which pedestrians move. It also makes the scope of our work more comprehensive cause we also apply skills obtained during the practical.

The primary benefit of utilizing VADERE for the purpose of producing fake data is that it enables us to mimic realistic and complicated circumstances that are either challenging to see in the actual world or are just not feasible to observe there. Because of this, we are able to conduct an analysis of the behavior of pedestrians in a variety of scenarios, including those involving emergency evacuations, high-density crowds, and other conditions that would be difficult to monitor in real life.

In addition, VADERE enables us to control a variety of parameters, such as the number of pedestrians, the geometry of the environment, and the behavior of the pedestrians; this gives us a high degree of control over the data that we generate. This makes it possible for us to construct a huge dataset that is reflective of the actual world and to investigate another scope, different from the scenarios from the original paper our final project is based on. This helps us establish the resilience of our model as well as check the accuracy of our forecasts.

To summarize, we opted to work with VADERE so that we can generate fabricated data for the purpose of our investigation into the prediction of pedestrian speed has furnished us with a potent instrument for simulating realistic yet complex situations, the likes of which are difficult to observe in the real world. This enables us to evaluate the accuracy and robustness of our predictions as well as study the behavior of pedestrians in a variety of scenarios to see how they react. It also makes our work more unique than the one from the original paper of Tourdeux.

For this task we use Vadare to create 3 different scenarios show in figures below.

1. Left turn scenario: Left turn with 100 pedestrians on a 20 x 20 meters grid. The hallway is 7 meters wide.

2. Corridor Scenario: 2-way corridor consists of 50 pedestrians going right and 50 pedestrians going left. The corridor is 20 x 7 meters.

3. Supermarket scenario : A 50 x 50 meters supermarket. A total of 120 clients enter the supermarket and move around from one place to another. Towards the end they exit the shop, not before make sure they have payed.

We use the Optimal Step Model as the engine for pedestrians' movement. The first two are simple scenarios and aim to test whether the architecture given in the original paper is also the best for these other scenarios. The last one is a complex scenario through which we try to understand if there is any architecture that can capture the dynamics in such a complex setting. To gather the data we use a post-processor which outputs for every time step and every a pedestrian a line containing the pedestrian's position: exactly what our next data processing step requires.

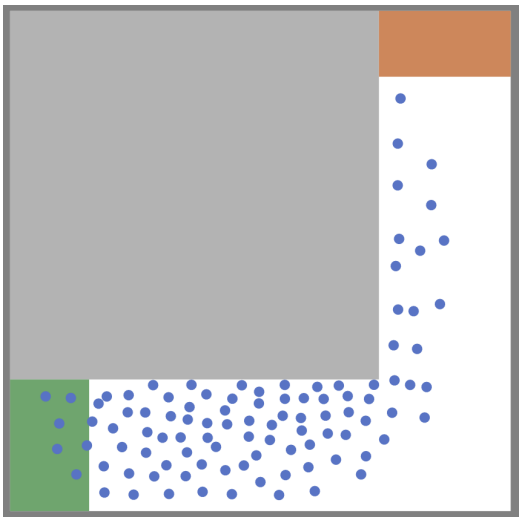

Figure 4: Left turn scenario

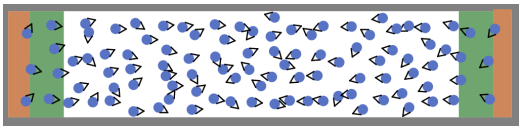

Figure 5: Corridor scenario

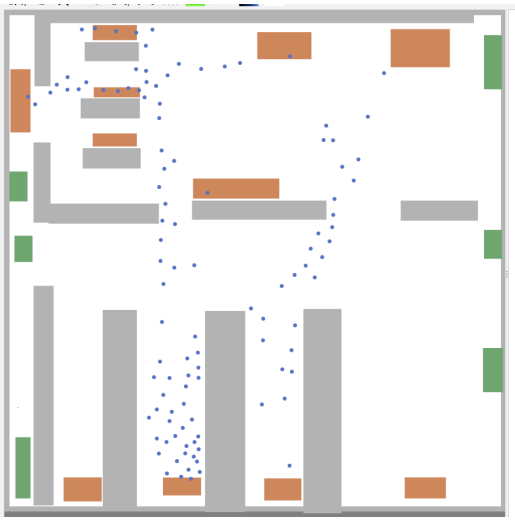

Figure 6: Supermarket scenario

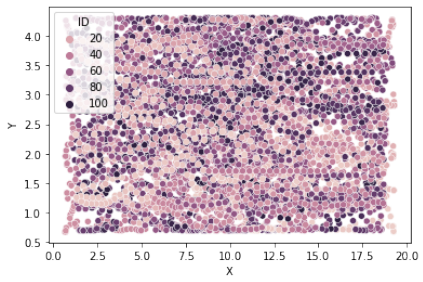

Figure 7: Scatterplot of the Corridor Scenario

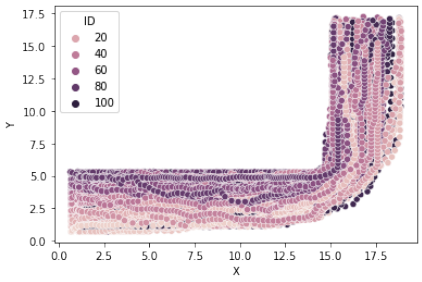

Figure 8: Scatterplot of the Left Turn Scenario

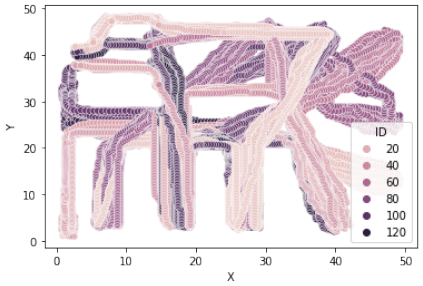

Figure 9: Scatterplot of the Supermarket Scenario

Report on task 3, Preprocessing the data

The data used in the experiment is obtained from scenarios run on Vadere software. According the scenario, the software provides the data in the form of {timestep, pedestrian_ ID, X-co-ordinate, Y-co-ordinate}. The timestep in the data depends on the parameter "simTimeStepLength" in the "attributesSimulation" part of the scenario. The same can be observed in the scenario json file obtained from Vadere as shown in the image below. This parameter is set to 0.4 for all scenarios. This parameter is required to calculate pedestrian speed to create the dataset. Also we can set a maximum speed (2.0 m/s for all scenarios) to the pedestrian in the scenario file itself. This is verified when we preprocess the data to create the dataset for training the model.

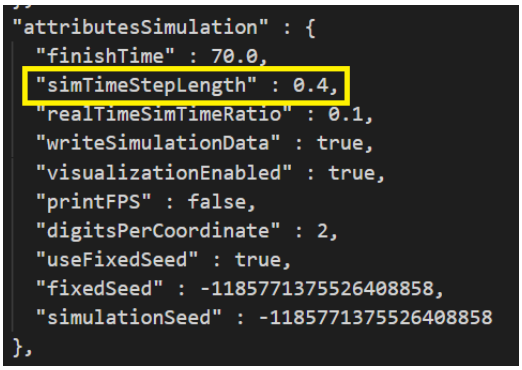

Figure 10: simTimeStepLength Parameter from scenario file

From these data we are asked to create the dataset needed for training the models. The parameters that we need for the training respectively are:

1. mean spacing as input and speed as output regarding the FD model. 1 float input, 1 float output.
2. mean spacing and knn positions (relative) as input and speed as output regarding the speed predictor model. 2k+1 float input, 1 float output.

In order to create the dataset, the following are the preprocessing steps of the data obtained from Vadere:

- Read the data and convert it to a dataframe
- Consider the current frame or timestep to find the K nearest neighbors (in our case this is 5) and calculate their distances.

```
frame = frame[frame['pedestrianId'].isin(next_frame['pedestrianId'])]
next_frame = next_frame[next_frame['pedestrianId'].isin(frame['pedestrianId'])]

if len(frame) <= NO_NEIGHBORS:
    continue

pos = frame[['x', 'y']].to_numpy()
next_pos = next_frame[['x', 'y']].to_numpy()

dist = distance.squareform(distance.pdist(pos))
dist = remove_diagonal(dist)[:, :NO_NEIGHBORS]

knn = np.argsort(dist, axis=1)
```

Figure 11: Code snippet to find distances of K nearest neighbors

- Use the current frame, the next frame and timestep length or frame rate to find the pedestrian speed. Add the calculated pedestrian speeds and the relative positions of K nearest neighbors to the dataframe to obtain the dataset as shown in figures below.

```
dic['timestep'] += frame.timeStep.to_list()
dic['pid'] += frame.pedestrianId.to_list()
dic['pos'] += pos.tolist()
dic['mean_spacing'] += dist.mean(axis = 1).tolist()
dic['speed'] += (np.linalg.norm(next_pos - pos, axis=1) / TIMESTEP_LENGTH).tolist()
dic['knn'] += (pos[knn] - pos[:, np.newaxis]).tolist()
```

Figure 12: Code snippet to find pedestrian speeds and relative positions of K nearest neighbors

	timestep	pid	pos	mean_spacing	speed	knn
0	1	1	[17.701, 0.7010000000000001]	0.615083	1.558207	[[0.8039999999999985, 0.0], [0.0, 0.0], [0.0, ...
1	1	2	[18.103, 0.7010000000000001]	0.468606	1.118237	[[0.0, 0.0], [-0.402000000000001, 0.4020000000...
2	1	3	[18.505, 0.7010000000000001]	0.615083	1.225062	[[-0.40199999999999747, 0.0], [-0.4019999999999...
3	1	4	[17.701, 1.1030000000000002]	0.615083	1.406499	[[0.0, -0.40200000000000014], [0.0, 0.0], [0.4...
4	1	5	[18.103, 1.1030000000000002]	0.468606	0.597641	[[0.0, 0.0], [0.0, -0.40200000000000014], [-0....
...	...	...	...	...	...	...
6806	97	21	[5.181136793745895, 3.3850734558817044]	4.644860	0.936756	[[0.0, 0.0], [-3.667156217732756, 0.37490439922...
6807	97	24	[4.720948195807343, 2.2659927824467654]	4.390301	1.348784	[[0.4601885979385525, 1.1190806734349388], [10...
6808	97	41	[15.330413479996816, 1.7052568064425038]	12.504181	0.960692	[[-10.149276686250921, 1.6798166494392004], [-...
6809	97	61	[1.4479074099845006, 2.480854747643503]	4.722096	0.599436	[[0.066073166028638 44, 1.2791231074759701], [0...
6810	97	85	[18.033312914121325, 3.381025639235303]	12.498176	1.057322	[[-2.702899434124509, -1.6757688327927993], [-...

6811 rows × 6 columns

Figure 13: Output snippet of dataset including speed and relative positions of K nearest neighbors

In this way we obtain the dataset with the parameters required to train our model for all scenarios.

Also following images show the scatterplot to visualize the trajectory and movement of pedestrians for the real world data used in the original paper for the Bottleneck and Corridor scenarios.

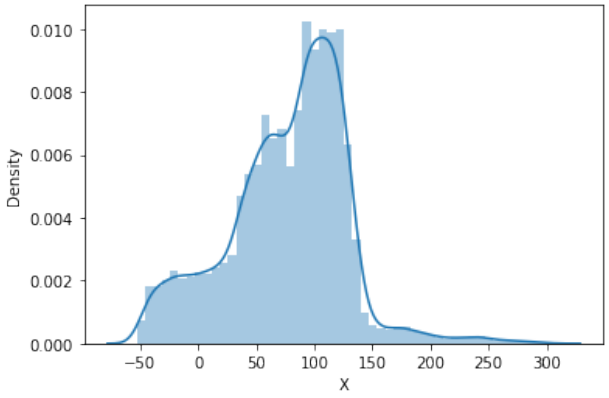

Figure 14: Histogram that shows the distribution of the **x** values of corridor data

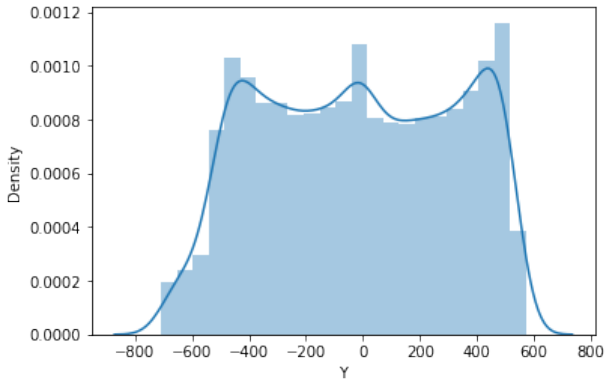

Figure 15: Histogram that shows the distribution of the **y** values of corridor data

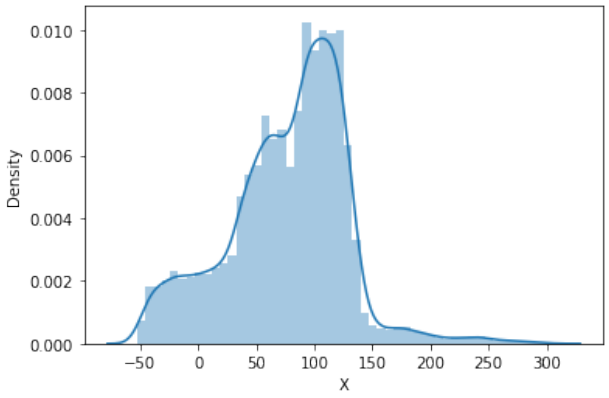

Figure 16: Histogram that shows the distribution of the **x** values of bottleneck data

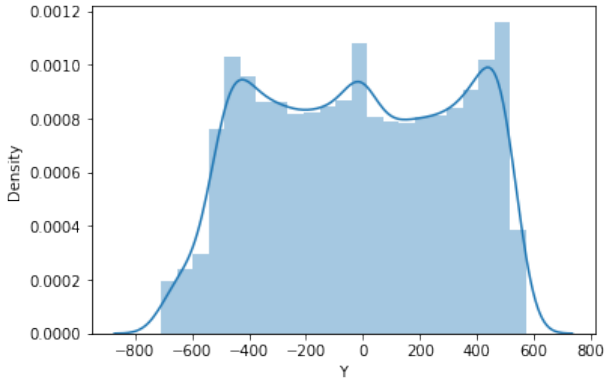

Figure 17: Histogram that shows the distribution of the **y** values of bottleneck data

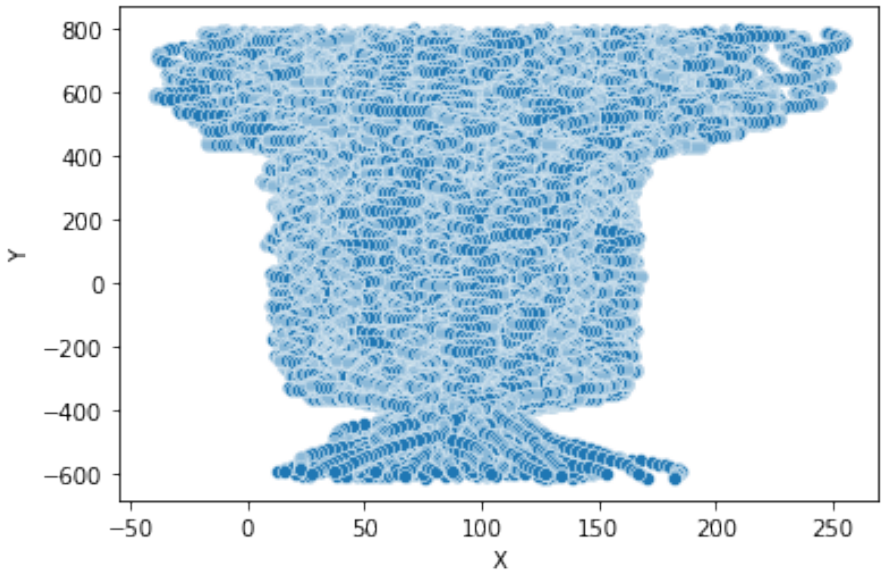

Figure 18: Scatterplot that visualizes the trajectory and the movement of the pedestrians in the real world data that is also used in the original paper in the Bottleneck Scenario

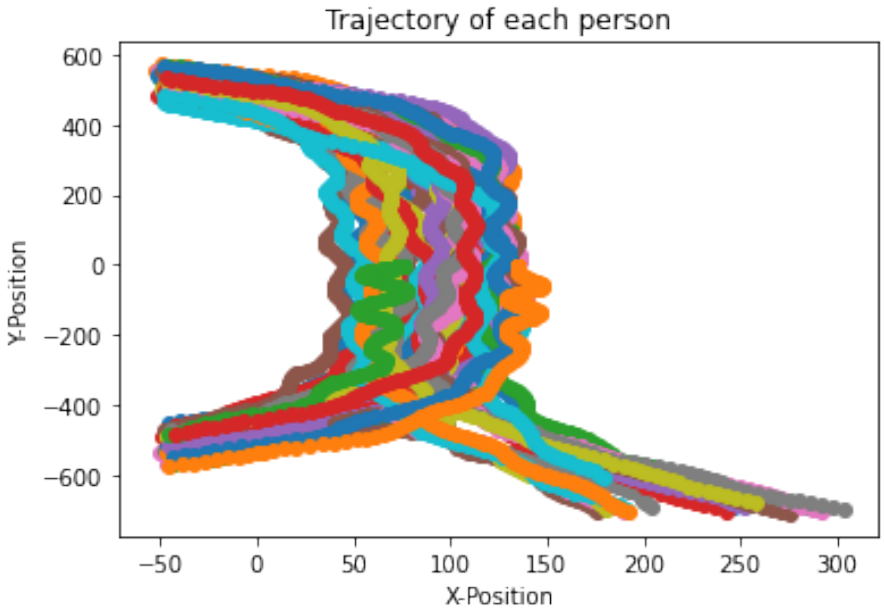

Figure 19: Scatterplot that visualizes the trajectory and the movement of the pedestrians in the real world data that is also used in the original paper in the Corridor Scenario

---

Report on task 4, Creating, Setting Up and Training of the Neural Network

Weidmann FD Model: We defined a custom neural network model that we called FD Network in order to create and simulate the Classical Weidmann approach described in the paper, which uses the fundamental diagram (FD) as its baseline. A single hidden layer, a dropout layer, and three output layers—one for the desired speed, pedestrian size, and time gap—make up this model, which is a simple feedforward network. The `softplus` activation function, which is used to activate the hidden layer as well as all of the output layers, offers a smooth approximation of the `ReLU` activation. This choice of activation functions was in no way arbitrary. These activation functions make sure that each layer's output values are finite, non-negative, and have distinct gradients, which helps to prevent NaN loss during training. They also make sure that each layer's output values have distinct gradients. Since we initially tried using `relu` and received NaN values for our validation loss, we had to learn this lesson the hard way. For us, this was a discouraging discovery.

The network is trained using the crowd's mean spacing values as input and the corresponding pedestrian speeds as training objectives. Throughout the model's training process, the Adam optimizer and mean squared error (MSE) loss function are both used. The `EarlyStopping` callback can be used to stop the training process if the model does not exhibit any signs of improvement after 10 iterations. We will discuss more thoroughly the use of `EarlyStopping` in the next subsection

The model can be used to predict pedestrian speeds based on the mean spacing value after being trained. The `visualize_pedestrian_speed_prediction_model` function generates a plot that contrasts the pedestrian speed prediction model's predictions with the actual speeds that were recorded.

On Figure 20 the implementation of the FD Model could be observed more comprehensively.

```python
def __init__(self, dropout_rate=0.1):
    """
    initialize the network, very simple feed forward network with 3 parameter outputs
    :param dropout_rate: rate of dropout to be used
    """
    super(FD_Network, self).__init__()
    self.hidden_layer = tf.keras.layers.Dense(10, activation='softplus')
    self.dropout = tf.keras.layers.Dropout(dropout_rate)
    # output layers producing the 3 parameters of the FD
    self.desired_speed = tf.keras.layers.Dense(1, activation='softplus')
    self.pedestrian_size = tf.keras.layers.Dense(1, activation='softplus')
    self.time_gap = tf.keras.layers.Dense(1, activation='softplus')
    self.FD_model_parameters = {'t': [], 'l': [], 'v0': []}
    self.mse = -1

def call(self, mean_spacing):
    """
    execute the feedforward, create the fd function from the parameters, return the predicted speed
    """
    x = self.hidden_layer(mean_spacing)
    x = self.dropout(x)
    v0 = self.desired_speed(x)
    l = self.pedestrian_size(x)
    t = self.time_gap(x)
    self.FD_model_parameters['t'].append(tf.math.reduce_mean(t, 0))
    self.FD_model_parameters['l'].append(tf.math.reduce_mean(l, 0))
    self.FD_model_parameters['v0'].append(tf.math.reduce_mean(v0, 0))
    return v0 * (1 - tf.exp((1 - mean_spacing) / (v0 * t)))
```

Figure 20: Implementation of the Weidmann Model Architecture

Neural Network Regression Model: First of all we should mention that the neural network is going to be a regression model. This model, which takes into account a number of variables, can be used to predict the speed of a pedestrian in different setting. The input data is stored in an array called "X" and has 11 attributes: one property describing the mean spacing and ten numeric attributes with values between 0 and 9. The output data, which is kept in the array y, contains the speed. The input to our network is equivalent to $2 * K + 1$, because our project is based on the research article "Prediction of Pedestrian Speed using Artificial Neural Networks." This k denotes how many people are walking right next to the pedestrian in question. The mean spacing value for each pedestrian is calculated using the first ten values, which are the X and Y coordinates of the closest neighbors. These numbers are listed in ascending order by value. The `StandardScaler` object is first used to standardize the data before the training and splitting of the neural network. Pre-processing using `StandardScaler` is used in data science and machine learning for normalization. To get a dataset with a mean of 0 and a standard deviation of 1, the process subtracts the mean and divides by the standard deviation. This makes the 'learning' of the model easier, because the data used is similar to each other. We decided to use a Standard Scaler, which is easily accesible with the `sklearn` library. As already mentioned we normalize the input data in our model, so that the model treats them identically. This is essential because features might have a wide variety of sizes and ranges, and neural networks can be sensitive to input scale. The inputs are normalized to guarantee that each feature contributes equally and that the model's predictions are unaffected by scale. Another benefit that normalization brings is that it is a good measure to reduce overfitting. When a complex model learns the noise in the training data, overfitting occurs, which has a negative impact on the generalization capabilites of a neural network. That is why the inputs are made smaller with normalization, which simplifies the model and lowers this danger.

The data are then split with a ratio of 50% each into a training set and a testing set.(Just like the paper suggests which is different to the most common 80/10/10 split). The bootstrapping and cross validation approaches are used to again for the purpose to prevent the model from becoming overfit and to obtain a more precise evaluation of the model's performance. The training data is resampled a total of thirty or fifty times to obtain 30 or 50 subsamples. The original paper from Tordeux [3] suggested that the resampling and bootstraping should be done 50 times on each scenario. We faced a sizable hurdle in terms of training time while using real world data to train the neural network, particularly in the bottleneck case. Given that the dataset had more than 75,000 data points, using 5-fold cross validation and training each subsample was a time-consuming procedure. 11 hours were spent on training as a result of this. Nevertheless, it was found that even though the training procedure took a long time, adding 50 bootstrapped subsamples had no appreciable impact on the regression model's performance. This shows that the extra time and effort put into making more subsamples was pointless and did not result in any appreciable advantages. Therefore, it can be concluded that future research should examine the use of fewer subsamples or a different strategy to improve the performance of the regression model and drastically lower the time used for training. Each subsample's data is divided into five folds for the purpose of cross validation. Two dense layers, the ReLU activation function for the first layer, and no activation function for the second layer make up a sequential neural network model. The first layer of the model also has a ReLU activation function. Throughout the model's training process, the Adam optimizer and mean squared error loss function are both utilized.
In the implementation of our neural network we use another very common concept in Machine and Deep Learn-

ing: Early stopping. The use of early stopping in regression models for the prediction of speed is driven jet again by the need to avoid overfitting just like the other things we included and to enhance the generalization performance of the model. Early stopping can be useful to achieve this goals. This concept is accessible and easy to use as it is supported by `Keras`. By keeping an eye on the performance on a validation set and stopping the training when the performance begins to decline, the model is able to strike a balance between good performance on the training data and good generalization to new data. As a result, the model is able to perform well on the training set of data and generalize well to new sets of data. This approach in our involves tracking the validation loss and pausing the training if the loss does not improve after 10 iterations.

The mean squared error, which is calculated using the validation data and the testing data that are used to evaluate the model, is used to determine the loss. This is the main metric for evaluation we use. The mean and standard deviation of the training loss, validation loss, and testing loss are calculated, and some plots are created for visualization purposes. The mean validation loss and mean testing loss provide an evaluation of the model's overall performance, while the standard deviation of the losses provides an indication of the variability of the model's performance.

On Figures 21,22,23 the whole model implementation could be observed how it is executed step by step.

```
X = preprocessed_data[["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "mean_spacing"]]
y = preprocessed_data["speed"]

# Preprocess the data
scaler_X = StandardScaler()
scaler_X.fit(X)
X = scaler_X.transform(X)

scaler_y = StandardScaler()
scaler_y.fit(y.values.reshape(-1, 1))
y = scaler_y.transform(y.values.reshape(-1, 1))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=42)

training_losses = []
validation_losses = []
testing_losses = []

kf = KFold(n_folds)
```

Figure 21: The implementation of the neural network we created

```
for i in range(n_subsamples):
    X_train_sub, y_train_sub = resample(X_train, y_train, random_state=i)

    for train_index, val_index in kf.split(X_train_sub):
        X_train_cv, X_val = X_train_sub[train_index], X_train_sub[val_index]
        y_train_cv, y_val = y_train_sub[train_index], y_train_sub[val_index]

        with tf.device('/GPU:0'):
            model = Sequential()
            model.add(Dense(units=3, activation='relu', input_shape=(X_train_cv.shape[1],)))
            model.add(Dense(1))
            model.compile(optimizer='adam', loss='mean_squared_error')

        early_stopping = EarlyStopping(monitor='val_loss', patience=10, verbose=1)

        with tf.device('/GPU:0'):
            history = model.fit(X_train_cv, y_train_cv, epochs=100, verbose=1, validation_data=(X_val, y_val),
                                callbacks=[early_stopping])

        with tf.device('/GPU:0'):
            val_loss = model.evaluate(X_val, y_val, verbose=1)
            validation_losses.append(val_loss)

        with tf.device('/GPU:0'):
            test_loss = model.evaluate(X_test, y_test, verbose=1)
            testing_losses.append(test_loss)
```

Figure 22: The implementation of the neural network we created

```
mean_training_loss = np.mean(history.history['loss'])
std_training_loss = np.std(history.history['loss'])
training_losses.append((mean_training_loss, std_training_loss))

# Calculate the mean and standard deviation of the validation losses

mean_validation_loss = np.mean(validation_losses)
std_validation_loss = np.std(validation_losses)

# Calculate the mean and standard deviation of the testing losses
mean_testing_loss = np.mean(testing_losses)
std_testing_loss = np.std(testing_losses)

print("Mean training loss: {:.4f} +/- {:.4f}".format(mean_training_loss, std_training_loss))
print("Mean validation loss: {:.4f} +/- {:.4f}".format(mean_validation_loss, std_validation_loss))
print("Mean testing loss: {:.4f} +/- {:.4f}".format(mean_testing_loss, std_testing_loss))
```

Figure 23: The implementation of the neural network we created

There was one more thought that crossed our minds as we were working on the project, and that was to create a regression model as a baseline in order to be able to make comparisons with the Neural Network. To implement it was quite easy because the `scikit-learn` library already supports this functionality. In our opinion it was a wise choice to use a general polynomial regression as baseline, because that type of model is useful for solving regression-related problems because it can capture non-linear correlations between the independent and dependent variables. The prediction of continuous variables like speed is another application in which polynomial regression models are frequently used. In order to make the characteristics (X) and goal (Y) variables in our study measure on a scale that is comparable to one another, we again used a Standard Scale, just like in the Neural Network. Next, we run the data through a pipeline that combines normalization and regression to adapt the polynomial regression model to the data. We can use cross-validation to assess the model's performance on the validation data after fitting the model to the training data quickly and efficiently thanks to the pipeline. Once again we opted for using the MSE as an evaluation metric for the performance on both the validation data and the test data (MSE). As mentioned before a lower MSE indicates that the model fits the data more accurately. By plotting the projected values against the actual values, we have also produced a visual representation of the regression's results.

On Figure 24 you can see the implementaition of the Polynomial Regression

```python
# Split the data into features (X) and target (y)

X = dataset[[ "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "mean_spacing"]]
y = dataset["speed"]

# Load the data
scaler_X = StandardScaler()

# Fit the scaler on X
scaler_X.fit(X)

# Transform X
X = scaler_X.transform(X)

# Initialize the scaler for y
scaler_y = StandardScaler()

# Fit the scaler on y
scaler_y.fit(y.values.reshape(-1, 1))

# Transform y
y = scaler_y.transform(y.values.reshape(-1, 1))

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=101)

# create a pipeline that combines normalization and regression
model = Pipeline([
  ('poly', PolynomialFeatures(degree=2)),
  ('regression', LinearRegression())
])

# use KFold cross-validation to evaluate the model
kfold = KFold(n_splits=5, shuffle=True, random_state=101)
scores = []
for train_index, val_index in kfold.split(X_train):
  X_train_fold, X_val = X_train[train_index], X_train[val_index]
  y_train_fold, y_val = y_train[train_index], y_train[val_index]

  # fit the model to the training data
  model.fit(X_train_fold, y_train_fold)

  # evaluate the model on the validation data
  y_val_pred = model.predict(X_val)
  score = mean_squared_error(y_val, y_val_pred)
  scores.append(score)

# print the average mean squared error across the 5 folds
print("Average MSE:", np.mean(scores))

# evaluate the model on the test data
y_test_pred = model.predict(X_test)
test_score = mean_squared_error(y_test, y_test_pred)
print("Test MSE:", test_score)
```

Figure 24: The implementation of the polynomial regression

Finally, using a general polynomial regression model gives us a useful point of comparison for our neural network model. Additionally, it enables us to assess the model's effectiveness and the precision of its predictions, which aids in our decision-making.

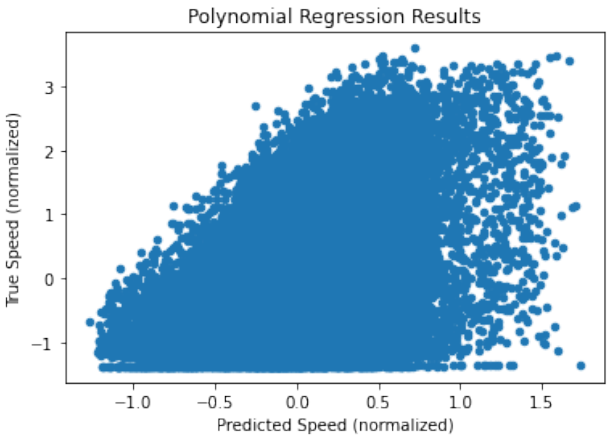

Figure 25: Polynomial Regression baseline for the Bottleneck Scenario

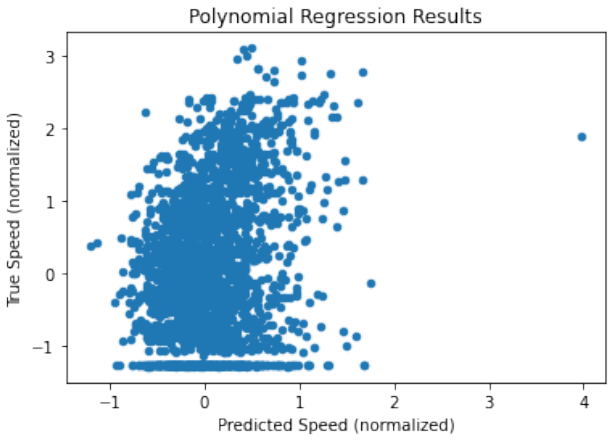

Figure 26: Polynomial Regression baseline for the Corridor Scenario of the real data

Data augmentation is a common tactic that is employed in various Machine and Deep Learning models; it entails creating more training examples based on the data that currently exists. We were able to generate our own data thanks to the open-source Vadere simulation software [1] as we described previously in Section and

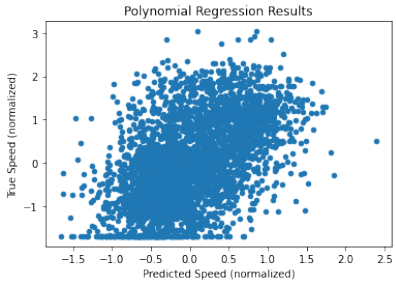

Figure 27: Polynomial Regression baseline for the Corridor Scenario Vadere data

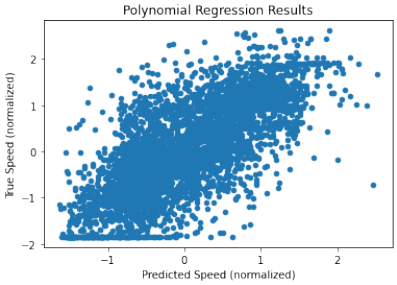

Figure 28: Polynomial Regression baseline for the Corridor Scenario Vadere data

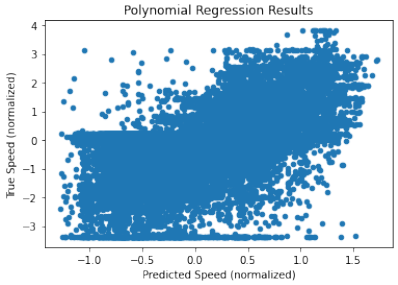

Figure 29: Polynomial Regression baseline for the Corridor Scenario Vadere data

Task 2, which we then utilized to anticipate the speed at which pedestrians walk. This allowed us to train our regression model with a large enough set of data. This is why we did not find it essential to implement any specific data augmentation procedures, because we have access to a large volume of data and also VADERE. Instead, given the information at hand, we focused our efforts on creating a precise regression model based on the research paper. In cases when there is a dearth of data, data augmentation may be a valuable tactic as it may help to increase the amount and variety of the dataset, which in turn improves model performance. This is something that has to be always thought about. On the other hand, in our case, the ability to generate data via Vadere gave us the advantage of having sufficient data without the need to use data augmentation techniques.

HyperParameter Tuning: "Hyperparameter tuning" is the process of figuring out the ideal values for a model's hyperparameters in order to achieve optimal performance in response to a particular challenge. The following code, which can be found on Figure 30, shows how hyperparameter tuning for a neural network regressor using `GridSearchCV` works. The network design is determined by the `build_regressor` function. The `optimizer, units_1, and units_2` are the three arguments that this function accepts. The optimizer argument specifies the optimization strategy to be used while the model is being trained, while the units 1 and units 2 parameters specify the number of nodes that should be present in the first and second hidden layers, respectively. Three thick layers make up the network's design; the first two of these layers each have an activation function known as a rectified linear unit `(ReLU)`, while the third layer lacks an activation function (i.e., a linear activation function). The loss function used during training is mean squared error `(MSE)`. A `KerasRegressor` class instance, which can be found in the scikit-learn library, houses the `build_regressor` method. The `param_grid` dictionary, which also contains batch size and epochs, contains definitions for the hyperparameters that coudl possibly benegit of some adjusting. The `GridSearchCV` object is applied to the training data by the function run hyperparameter tuning, which then displays the best hyperparameters and the mean squared error for each of them. The `GridSearchCV` object performs a 5-fold cross-validation to assess the relative merits of the various hyperparameter configurations.

```python
def build_regressor(units_1, units_2, optimizer):
    model = Sequential()
    model.add(Dense(units=units_1, activation='relu', input_shape=(X_train_cv.shape[1],)))
    model.add(Dense(units=units_2, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer=optimizer, loss='mean_squared_error')
    return model

# Initialize the KerasRegressor
regressor = KerasRegressor(build_fn=build_regressor)

# Define the hyperparameters to tune
param_grid = {'units_1': [3, 5, 10],
              'units_2': [3, 5, 10],
              'optimizer': ['adam', 'rmsprop'],
              'batch_size': [32, 64, 128],
              'epochs': [50, 100, 200]}


def run_hyperparameter_tuning(X_train,y_train):
  # Initialize the GridSearchCV object
  grid_search = GridSearchCV(estimator=regressor, param_grid=param_grid, scoring='neg_mean_squared_error', cv=5)

  # Fit the GridSearchCV object on the training data
  grid_search.fit(X_train, y_train)

  # Print the best hyperparameters and the corresponding mean squared error
  print("Best hyperparameters: ", grid_search.best_params_)
  print("Mean squared error: {:.4f}".format(-grid_search.best_score_))
```

Figure 30: Implementation of the script that could be used for hyperparameter tuning

It is important to say that we implemented this functionality and our project takes this also in consideration and supports it, but we did not have sufficient time to conduct comprehensive analysis of HyperParameter tuning. Given that there are so many variables that can affect how long `grid_search.fit(X_train, y_train)` will take, it is challenging to give an exact time estimate. The size of the training data, the number of grid search combinations, the model's level of complexity, the state of our computing resources, and many others. On the other hand, grid search, which trains multiple models and must look for the best values for its hyperparameters, can frequently take a lot longer than training a single model. Given the fact that the model was training 11 Hours on the Bottleneck Scenario of the real data we decided not to perform Hyperparameter tuning ourselves.

Discussion on the paper It is important to say that the research our work is based on is using really small neural networks, which is quite uncommon for Deep Learning architectures.

We also found it very strange and were surprised to learn that the authors never actually acknowledged and described their hyperparameters. The only things they discuss are the fact that they use an activation function and three input layers. They made no mention of regularization, data normalization, or anything else that might have helped other researchers replicate and apply their findings.

Since the data used for this investigation was time-series data, it is reasonable and logical to be concerned not only with finding the best match for our predictions but also with extrapolating based on historical trends. This is a very crucial factor to consider when estimating the speed of pedestrians because we must consider both the immediate environment and any other external factors that might have an impact on the pace. That subject was never really explored by the report's authors, either in terms of whether they addressed it or how they did.

Nowadays, a sizeable portion of research is conducted using recurrent neural networks, which is significant when taking into account the current state of the art. In further research, the effectiveness of these various models for predicting pedestrian speed may be compared in order to determine the best approach to solving this problem.

Report on task 5, Evaluation and Discussion

Weidmann FD Model: The model can be used to make predictions about pedestrian speeds after it has been trained, given the mean distance between each pedestrian. The speeds that were observed and the speeds that the model predicted are then shown in a scatterplot created with the `seaborn` library. For visualization this scatterplot with "Mean Spacings" plotted along the x-axis and "Speeds" plotted along the y-axis is created.

Another scatterplot is used to visually represent the relationship between the model's predicted speeds and the average distances between pedestrians. These scatterplots we generated give us a visual overview to evaluate the effectiveness of the model and identify relationships between mean spacings and speeds. It can reveal information about how well the model predicts the future and highlight any areas that could want improvement.

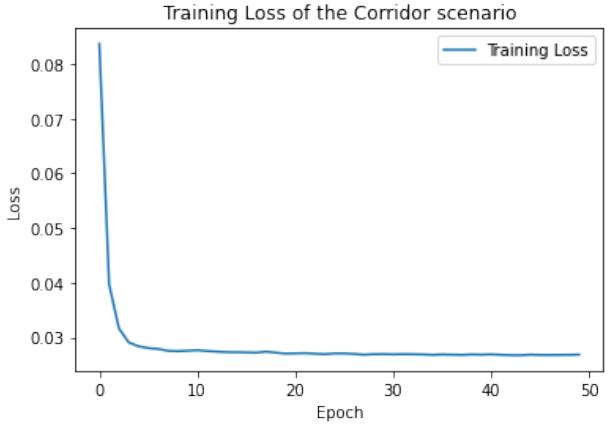

Figure 31: The training loss of the Weidmann Model for Corridor

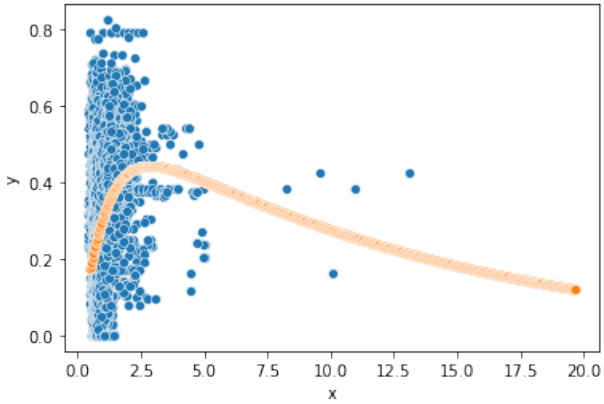

Figure 32: Scatter Plot Real vs Prediction Weidmann Model for Corridor

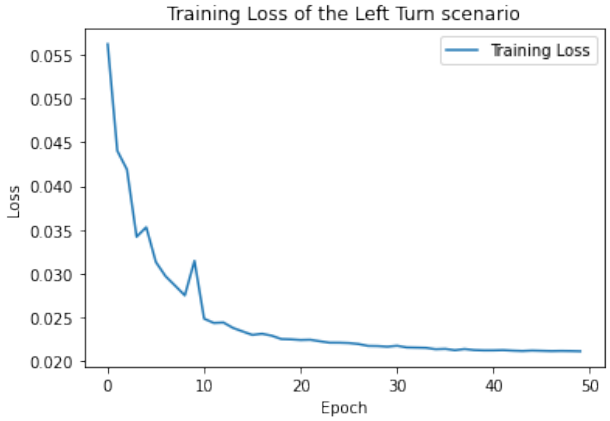

Figure 33: The training loss of the Weidmann Model for Left Turn

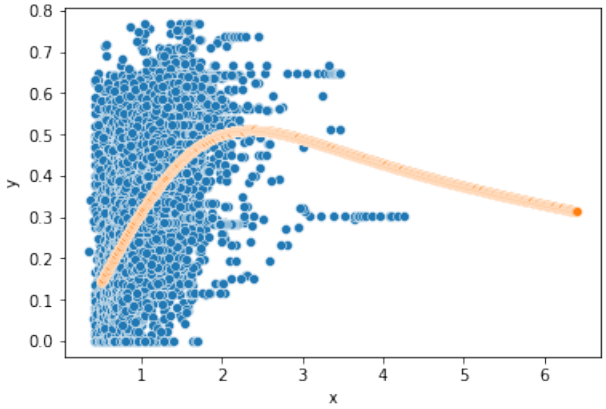

Figure 34: Scatter Plot Real vs Prediction Weidmann Model for Left Turn

Neural Network Regression Model: In the regression neural network that is used to predict the speed of pedestrians, it is typical to see a little discrepancy between the validation loss and the training loss. When the difference is between the range of 0.01 to 0.09 standard deviations, it may be assumed that the model has not been overfit to the training set in the current situation. As we described previously in the neural network section overfitting occurs when a model has absorbed the training data too well and finds it challenging to generalize to new data that it has never seen before. It's likely that the result we got for our regression model were influenced by the small size of the neural network in question. The fact that the data were set up in a time-series format may also have have had some of an impact on the results. When working with time-series data, extrapolation is frequently necessary as opposed to interpolation, which entails generating predictions outside the bounds of the training data. Using interpolation is a simpler technique. It is imperative in this situation to have a model that generalizes effectively to new inputs.

A tiny, generalizing model may be particularly helpful for the two situations under investigation, the bottleneck scenario and the corridor scenario which the original paper is also based on. It is difficult to create precise forecasts under these circumstances because there are complex interactions between pedestrians and

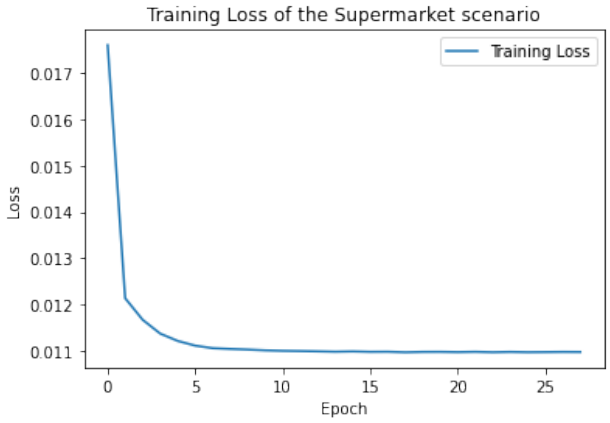

Figure 35: The training loss of the Weidmann Model for Supermarket

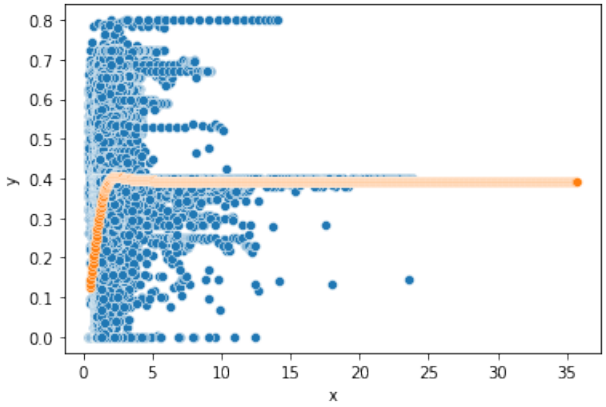

Figure 36: Scatter Plot Real vs Prediction Weidmann Model for Supermarket

their environment. With a model that generalizes effectively, the neural network is better able to manage these intricate connections and produce precise predictions.

When considering the model as a whole, the fact that there isn't much of a difference between the training loss and the validation loss in our regression neural network for the prediction of pedestrians' speed is a positive indication that the model is trustworthy and readily adjusts to new data. It's important to remember, though, that in these sorts of challenging situations, other cutting-edge techniques, such the usage of recurrent neural networks or long short-term memory networks, might also be used to estimate the pace of pedestrians.

Results:

- **Weidmann Classical Approach**: Weidmann's classical model was evaluated using pedestrian speed data from the Vadere simulation program, and the results were overwhelmingly positive, which we found to be very encouraging. In all three of the tested scenarios, the model obtained training loss values that were incredibly low. For instance, the model lost 0.0268 in the scenario involving the bidirectional corridor, 0.0212 in the scenario involving a left turn, and 0.011 in the scenario involving a supermarket. Due to the lower mean speeds in the data produced by the Vadere, these results suggest that the Weidmann classical model is well suited for them, allowing it to perform incredibly well and produce forecasts that are extremely accurate.

  Figure 31- Figure36 show plots for the training loss of the three scenarios respectively.

- **Neural Network / Paper Data**: The regression neural network's results show that it performed admirably on both the training data and the validation data after being trained on the bottleneck scenario. The validation loss is marginally reduced by roughly 0.0349 compared to the mean training loss which has a value of 0.8719, and the mean validation loss, which is 0.8370. This was good news for us while training the model because it indicates that the network is not overfitting on the training data which our main goal was. The validation loss should be less than the training loss since it shows how well the network generalizes to data it has never seen before. Further evidence that the network is not overfitting to the data it has seen and is able to generalize well to data it has not seen comes from the fact that the validation loss and the mean testing loss are both equal 0.8360. These results suggest that the network is able to forecast accurately the speed at which pedestrians travel in situations with bottlenecks.
  The regression neural network's results show that it performed well on both the training and validation sets of data after being trained also on the corridor scenario. The validation loss is lower by about 0.066 than the training loss, which is relatively close to one with a mean of 0.9382 and a mean of 0.8718. This shows that the network is once again not overfitting on the training data and that, as was the case with the bottleneck example, it can generalize well to data that it has never seen before. On the other hand, the validation loss is marginally less than the mean testing loss of 0.9455, which raises the possibility that the validation data may have overfit. This might be the case because the dataset for the corridor is much smaller than the dataset for the bottleneck (approx 10 times smaller). If there was really a case

of overfitting in the corridor scenario, the outcomes might not be as trustworthy as the results of the bottleneck one. To find a solution to this issue more investigation and testing using different network designs and training methods. To fully analyze the network's performance and put improvements into place where they are needed, more work must be done.

Figures 37 till 45 give better visualization overview of the results we obtained.

- **Neural Network / Vadere Data :** It is necessary to take into account the circumstances surrounding the gathering of these data if one is to successfully examine and understand them. The speed values in this case were first normalized using a standard scaler before being split into the training, validation, and testing datasets. In addition to removing any potential biases or outliers from the data, this normalization technique makes the data easier for the neural network to handle.

  Based on the results, it can be concluded that the Left Turns scenario's average training loss is 0.7222, with a standard deviation of 0.1385. With a standard deviation of 0.0271, the mean validation loss is 0.6583 and the mean testing loss is 0.6607.The mean validation loss is less than the mean testing loss by a value of 0.0271. These numbers show the model's ability to adapt and predict pedestrian speeds in the Left Turns scenario with pretty high accuracy. This is the scenario in which our architecture perfomed the best

  The next scenario we will discuss is the Corridor scenario's training experience loss averages 0.8486, with a standard deviation of 0.1430. The difference between the mean validation loss is 0.8387 and the mean testing loss is 0.9145 is around 0.0184. The testing loss's standard deviation is 0.0184, while the validation loss's standard deviation is 0.0478. Given that the validation and testing losses in the Corridor scenario are significantly higher than they were in the Left Turns scenario, these findings suggest that the model is likely not performing as well in this geometric setting.

  Last but not least we discuss the setting in a Supermarket. The Supermarket scenario's mean training loss is 0.7925, and the standard deviation is 0.0413. The mean validation loss is 0.7920 and the mean testing loss is 0.8042. The differnce between the mean validation loss is less than the mean testing loss in this case is 0.0385. These results suggest that the model might perform better in the supermarket scenario than the corridor scenario, though perhaps not as well as it did in the left turns scenario.

  The outcomes demonstrated that the model predicts the speed of pedestrians in the left Turns scenario better than both the Supermarket and Corridor scenarios. It's likely that the Left Turns scenario did better since it was less complicated than the other two. The Left Turns scenario only involves pedestrians making left turns, whereas the Supermarket scenario involves pedestrian behavior that is more complex and varied, and the Corridor scenario involves pedestrian flow in both directions, which makes it more difficult for the neural network to accurately predict pedestrian speed. In addition to this, the Left Turns scenario had 9k data points whereas the Corridor scenario had only 6k.

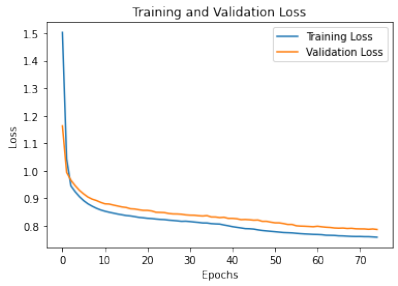

Figure 37: Train-Valid Loss of the Corridor Scenario

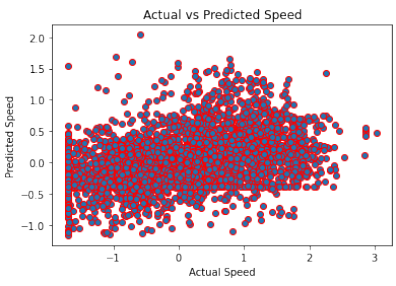

Figure 38: Actual vs Predicted Values of the Corridor

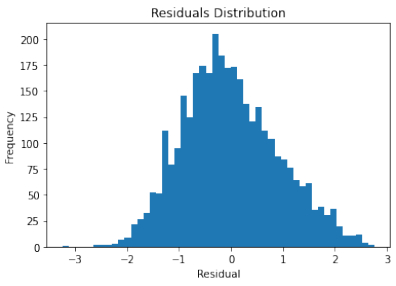

Figure 39: Residual Graph of the Corridor Scenario

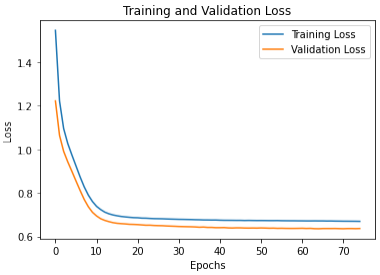

Figure 40: Train-Valid Loss of the Left Turn Scenario

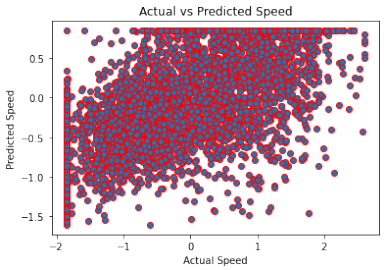

Figure 41: Actual vs Predicted Values of Left Turn Scenario

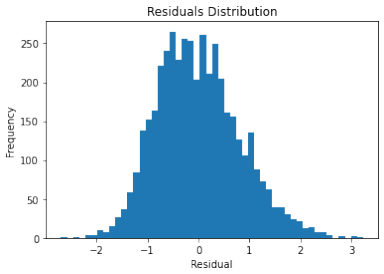

Figure 42: Residual Graph of the Left Turn Scenario

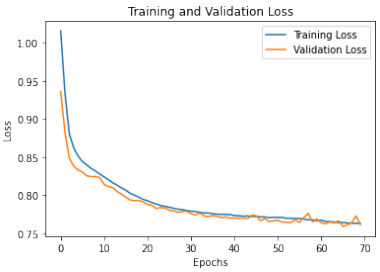

Figure 43: Train-Valid Loss of the Supermarket Scenario

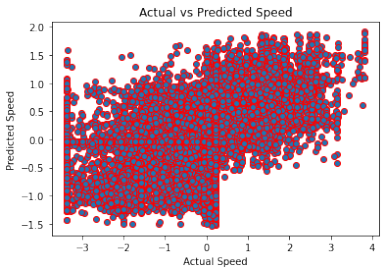

Figure 44: Actual vs Predicted Values Supermarket Scenario

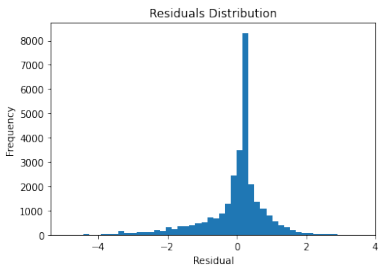

Figure 45: Residual Graph of the Supermarket Scenario

References

[1] Benedikt Kleinmeier, Benedikt Zönnchen, Marion Gödel, and Gerta Köster. Vadere: An open-source simulation framework to promote interdisciplinary understanding. *Collective Dynamics*, 4, 2019.

[2] Antoine Tordeux, Mohcine Chraibi, Armin Seyfried, and Andreas Schadschneider. Data from: Prediction of Pedestrian Speed with Artificial Neural Networks, November 2017.

[3] Antoine Tordeux, Mohcine Chraibi, Armin Seyfried, and Andreas Schadschneider. Prediction of pedestrian speed with artificial neural networks, 2018.