Report for exercise 6 from group E

Tasks addressed:	5
Authors:	Minxuan He (03764584)
	Çağatay Gültekin (03775999)
	Vatsal Sharma (03784922)
	Zhitao Xu (03750803)
	Gaurav Vaibhav (0366416)
Last compiled:	2024–02–08
Source code:	https://gitlab.lrz.de/mlcms-ex-group-e/mlcms-ex-group-e

The work on tasks was divided in the following way:

Minxuan He (03764584)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Çağatay Gültekin (03775999)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Vatsal Sharma (03784922)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Zhitao Xu (03750803)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Gaurav Vaibhav (0366416)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%

Report on task 1, Background Description

We often encounter many difficulties in crowd modeling, such as complex spatial structures, interactions between people, and pedestrian diversity. Among them, the type of facilities has the greatest impact on pedestrian behavior. Pedestrians will adjust their behavior according to the conditions of the facilities. For example, the flow of people tends to increase locally at bottlenecks. The types of facilities are diverse and not precisely defined, which makes system identification ambiguous. For complex spatial structures, we adopt a neural network model to predict the speed of pedestrians. This is because the parameters of neural networks have no fixed and direct physical meaning and are very flexible. Classic decision-based, speed-based or force-based models are difficult to make accurate predictions in such fuzzy and complex situations.

The purpose of this paper is to evaluate whether neural networks can more accurately describe pedestrian behavior in different spatial structures. We trained and tested the forward feedback neural network to predict the speed of pedestrians. We compared the neural network with the Weidmann model in corridor and bottleneck experiments. Judging from the results of the mean square error, the neural network is a better model for predicting pedestrian speed.

1.1 Models

For the two models that predict continuous speeds, we take the nearest neighbor $K = 10$, denote v as the velocity, and denote $((x_i, y_i), i = 1, ..., K)$ as the locations of K nearest neighbors.

1.1.1 Weidmann Model

The Weidmann model is a basic fundamental diagram model that explores the relationship between speed and density of pedestrian movement. The relationship between the density and speed of pedestrian movement has not yet been fully understood. According to Armin Seyfried et al. [1] experimental observations show that there is a linear relationship between speed and the reciprocal of density. From equation 1 we know that the velocity v is equal to the product of the desired velocity v_0 and a term that scales it by several factors, where T is the time gap, l is the pedestrian size, v_0 the desired speed and s_K is the mean spacing. The mean spacing is defined as equation 2.

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

$$\bar{s}_K = \frac{1}{K} \sum_{i=1}^{K} \sqrt{(x - x_i)^2 + (y - y_i)^2} \tag{2}$$

1.1.2 Neural Network Model

Construct a feedforward artificial neural network (ANN) with hidden layer H.

$$v = NN(H, \bar{s}_K, (x_i - x, y_i - y, 1 \leq i \leq K)) \tag{3}$$

In the first network, the inputs are the relative positions to the K closest neighbors ($2K$ inputs). In the second network, the speed is predicted as a function of the relative positions and the mean distance spacing s_K to the K closest neighbors ($2K + 1$ inputs).

1.2 Data

Data come from an experiment [2] conducted in 2009 at the Messe Hall in Düsseldorf, Germany, with up to 400 participants. The experiment analyzed pedestrian behavior in various scenarios: straight and ring corridor, T-junction, and bottleneck. We select two data sets: the ring corridor and the bottleneck experiment (data sets are downloaded from [3])

Ring/Corridor For a closed-loop geometry of 30m long and 1.8m wide, pedestrian densities range from 0.25 to 2 ped/m^2, corresponding to several participants from 15 to 230. The measured area is 6m long and lies on a straight line segment. We obtained 8 experimental results.

Bottleneck The system width in front of the bottleneck is 1.8 m, while the width of the bottleneck varies, from 0.70, 0.95, 1.20 to 1.80 m. The number of pedestrians is fixed at 150. This experiment can be seen as a corridor with a sudden reduction in width. By changing the width of the bottleneck width, we obtained the results of four experiments.

Figure 1 shows that for the same average spacing, the speeds of the two scenarios are different, and the speed for a given average spacing is on average higher in the bottleneck than in the corridor experiment. As mentioned above, geometry affects pedestrian behavior to a certain extent.

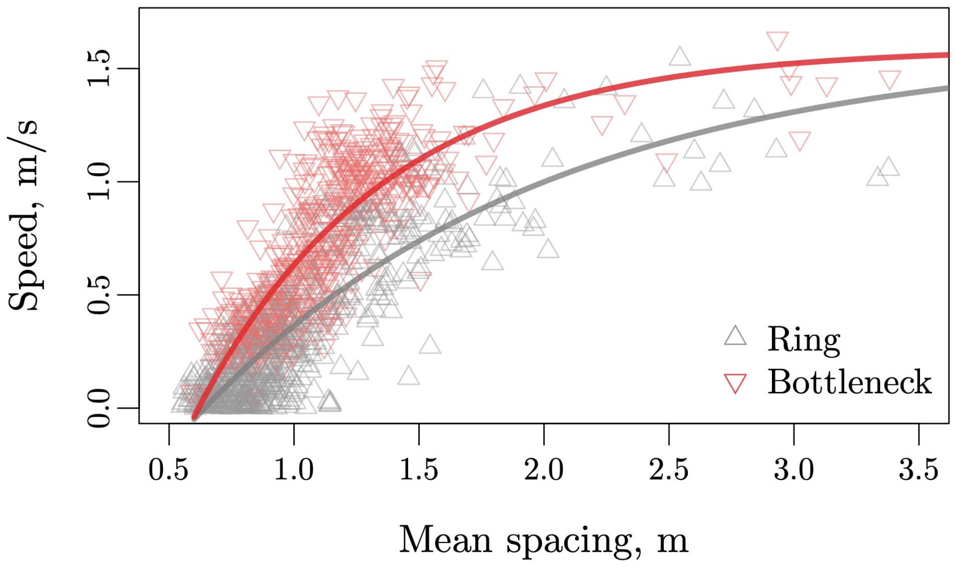


Figure 1: Fitting of the Weidmann model to corridor and bottleneck experiments, showing the relationship between pedestrian speed and mean distance spacing.

1.3 Fitting Model

Fit the neural network by minimizing the mean square error(MSE), where v_i is the actual speed and $\tilde{v}_i$ is the speed predicted by the NN.

$$\text{MSE} = \frac{1}{N}\sum_i (v_i - \tilde{v}_i)^2 \tag{4}$$

In this paper, 8 different neural network architectures were constructed. The corresponding mean square error of each type is shown in the figure 2. As the network complexity increases, the error continues to decrease, and the test error presents a minimum value before overfitting. We can see that the architecture of (3) exhibits the minimum.

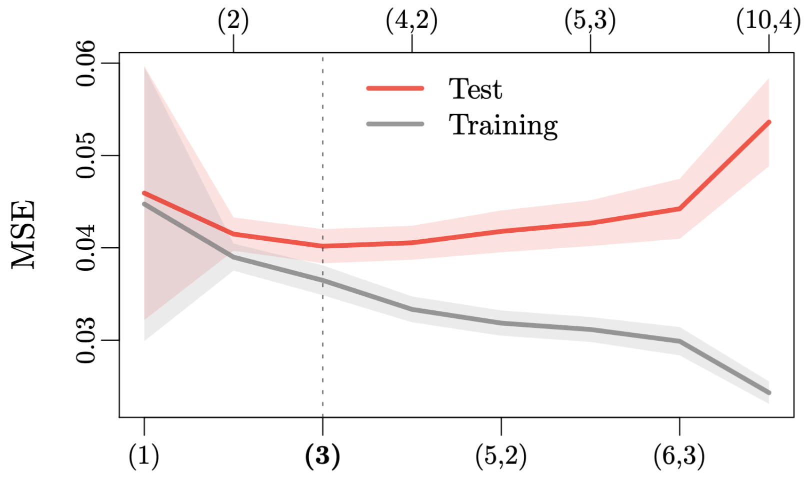


Figure 2: Minimum mean square error for training and testing of different NN architectures

1.4 Conclusion

Compare the Weidmann model and the best-performing neural network on different data combinations, as shown in Figure 3. ("./." The former represents the training phase, and the latter represents the testing phase.) When training and testing only on ring data (R), the difference between the FD model and the NN model is not obvious. When training and testing on bottleneck data, the MSE decreases by only 5%. When R/B and B/R, the MSE is reduced by 15 %, and the improvement is obvious. When the loop experiment and the bottleneck experiment are combined, the best effect can be obtained, which can increase the speed estimate by about 20%.

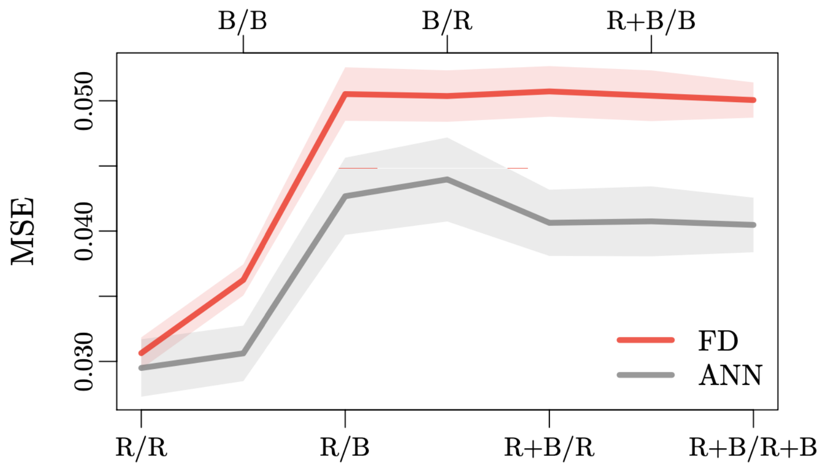


Figure 3: Comparison of MSE of NN relative to FD model for different combinations of scenarios.

Finally, as to the question of whether artificial neural networks can predict pedestrian behavior in complex geometric shapes, the answer is yes. The effect of NN is better than that of the classical physics-based model on these two data sets.

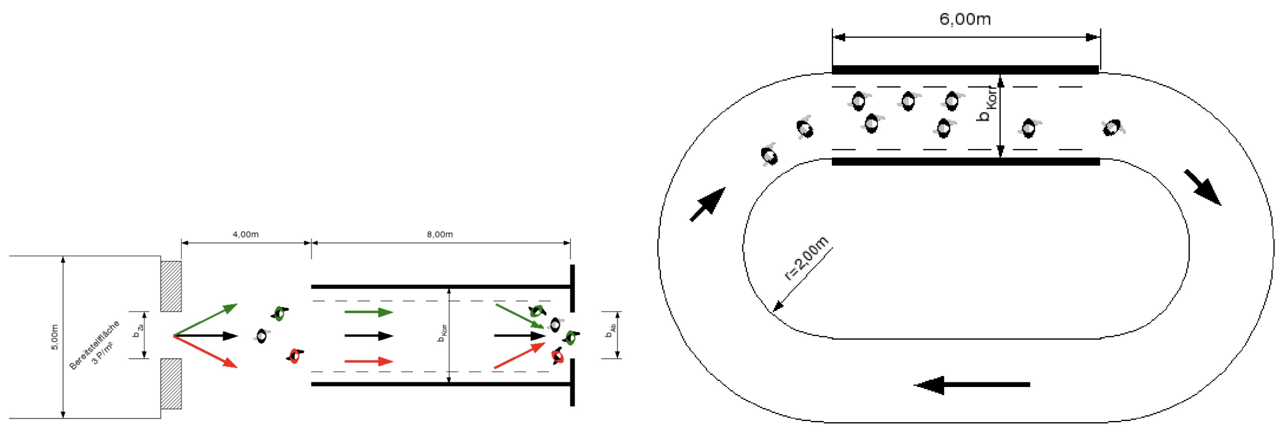
Report on task 2, Data Processing

The process of data processing is essential to comprehending and evaluating pedestrian behavior in the context of pedestrian dynamics and movement prediction. Reliable data processing techniques are essential for producing high-quality datasets that support precise analysis and model building. By systematically parsing, cleaning, and deriving meaningful insights from raw pedestrian movement data, researchers and practitioners can gain valuable insights into crowd dynamics and pave the way for the development of robust predictive models. The methods and strategies used to transform the raw dataset into a structured dataset fit for analysis and model building are explained in this section.

2.1 Raw Dataset Overview

The data originate from an experiment [2] that involved up to 400 participants and was carried out in 2009 at the Messe Hall in Düsseldorf, Germany. The ring corridor and the bottleneck experiment datasets are the two that we have chosen.

The unprocessed dataset consists of pedestrian movement records taken in a variety of complex environments, such as corridor and bottleneck situations. Raw Dataset is constructed with frame numbers, pedestrian identities, and three-dimensional coordinates that show the whereabouts of pedestrians across time are all included in each record. These are the situations 4 that data structure from:



(a) Bottleneck: The paths of pedestrians within an 8-meter-long and 1.8-meter-wide bottleneck are known as bottleneck data. 150 people participate in the experiments at bottleneck widths of 0.7, 0.95, 1.2, and 1.8 meters.

(b) Corridor(Ring): Footpaths in a closed corridor of 30 meters in length and 1.8 meters in width are called corridor data. A 6-meter segment is used to measure the trajectories. N = 15, 30, 60, 85, 95, 110, 140, and 230 people participate in the experiments.

Figure 4: The complex environments

2.2 Data Preprocessing

In this section, we will analyze how we have done the data preprocessing with the functions of the code.

2.2.1 Generate Dataset Function

The main purpose of this function is to orchestrate the entire process of dataset generation from raw data files. Its parameters are as follows:

- **exp**: A string identifier for the experiment, used for logging.
- **filepaths**: A list of paths to raw data files.
- **K**: The number of nearest neighbors to find for each point.

- **data-dir**: The directory to save processed data files.
- **data-filenames**: Names for the processed data files, corresponding to each raw file.

Its process flow is as follows:

1. **Initialization**: Logs the start of the dataset generation process.
2. **Iterating through raw data files**: For each file in filepaths, the code reads the data into a DataFrame with columns ['ID', 'FRAME', 'X', 'Y'], representing an object's ID, the frame number, and its X and Y coordinates.
3. **Adding Speed**: The code computes the speed for each point by calculating the Euclidean distance it moved between consecutive frames and multiplying by a factor to convert to m/s. This is done individually for each object (identified by 'ID') and the calculated speed is added as a new column.
4. **Grouping by Frame**: The DataFrame, now including speeds, is grouped by the frame number. This organizes the data such that all points in the same frame are processed together.
5. **Finding Nearest Neighbors and Computing Features**: For each frame group, the code finds the K nearest neighbors for each point using Euclidean distance. It calculates the difference in coordinates between each point and its neighbors, flattening these differences into a single array per point. It also calculates the mean spacing (sk) for each point as the average distance to its K nearest neighbors. These differences and the mean spacing form the feature set for each point, along with the previously computed speed.
6. **Saving Processed Data**: The features for all points in all frames are combined and saved into a CSV file specified by data-filenames. The CSV contains columns for mean spacing (sk), coordinate differences with nearest neighbors (as multiple 'feature' columns), and the speed.
7. **Completion Log**: Logs the successful generation of the dataset and its location.

2.2.2 Save Data Function

The main purpose of this function is to save the processed data to a CSV file. It takes the numpy array of samples, converts it into a DataFrame with appropriate column names (mean spacing, features, and speed), and saves it to the specified directory and file.

2.2.3 Key concepts of the code

In this subsection, we will talk about what are the key concepts of the code, how it functions, and how it affects the raw data. These are the main key concepts:

- **Nearest Neighbors**: This method is crucial for understanding the spatial context of each point by identifying its closest counterparts in the same frame.
- **Feature Engineering**: The process involves creating meaningful variables (mean spacing, relative positions of neighbors, and speed) that can help in predictive modeling.
- **Data Preprocessing**: Involves cleaning and transforming raw data (like computing speeds and organizing by frames) to make it suitable for analysis.

To conclude, this code effectively turns spatial-temporal point data into a structured form that's ready for machine learning tasks, focusing on the spatial relations (via nearest neighbors) and dynamics (via speed) of the points.

Report on task 3, Fundamental Diagram based Models

We begin with the first approach for modeling crowd behavior under different scenarios. As stated earlier, since the very first studies of pedestrian streams have been conducted, a notion has been associated with some hidden relation between the two parameters, pedestrian velocity and local density around a pedestrian.
This dependency between these 2 parameters, is termed as "fundamental diagram" [1] and models exploiting this relation are called FD-based models.

3.1 Model Description

We choose the Wiedmann model as the FD-based model for our analysis here. The model tries fitting an exponential relationship between the speed and mean spacing (a measure of local density).

$$v = FD(\bar{s}_K, v_0, T, l) = v_0 \left(1 - e^{\frac{l-\bar{s}_K}{v_0 T}}\right) \tag{5}$$

The mean spacing can be given as -

$$\bar{s}_K = \frac{1}{K} \sum_{i=1}^{K} \sqrt{(x - x_i)^2 + (y - y_i)^2} \tag{6}$$

Here v_0 is the desired velocity, T is the time gap and l is the pedestrian size. These are the three parameters that we need to find out and later feed to the above model. Additionally, we choose mean squared error to be our loss function.

3.1 Underlying algorithm for Calculating Parameters

Given a loss function, and a set of parameters that we wish to optimize, the first approach that strikes our mind is the "Gradient Descent" approach, where samples are repeatedly fed, and with each sample, our estimated parameters are slowly tuned in the direction of the optimal parameters. But this algorithm faces a major setback in cases where we have more than one minimum for the loss curve. While iterating it may get stuck in the local minimum and may fail to reach the global optimal set of weights, depending upon the initial weight initialization. We thus use a different technique called differential evolution(DE) to evaluate and find the global minimum set. Key features of the algorithm are -

- This algorithm is used for calculating global critical points in multidimensional real-valued functions and is evolutionary in nature. It means this method tries to optimize a problem by iteratively trying to improve over a candidate solution for a given measure of quality.
- The working of this algorithm does not require the calculation of derivatives at any point.
- This is the reason this algorithm works well irrespective of the nature of the objective function. The function to be optimized could be discontinuous, non-differentiable, or noisy.

3.1.1 Flow of the Algorithm

As already stated, DE is an evolutionary algorithm that is loosely based on nature's laws of genetics. Such algorithms aim to improve the overall quality of a particular population, by introducing mutations in every generation, allowing the fit individuals to be replaced with their fitter offspring, thus gradually converging to a population with the best characteristics. The fitness of an individual can be estimated by the error it produces when used in the model. Lower the model error, the fitter is the individual.
In this section, we try to provide high-level insights into the working of the differential evolution method, by laying out the different steps associated with it.

- **Initialization:**
  We first intend to create a family of solutions. In our case, a solution is a three-member vector containing the values of l,v_0, and T. We achieve this by creating n random samples of such solutions, thus creating our initial population. Additionally, numerical bounds are specified for every parameter in the solution set. Each solution is now referred to as an "individual".

- **Mutation:**
  This is an important step, as here we wish to disturb our population, by introducing a mutation. Here for every individual in the population, three random individuals are chosen, without replacement (say *A, B* and *C*). Now we wish to create mutant individuals/vectors by subtracting B and C vectors, scaling them by a mutation factor α, and adding it to A.

- **Crossover:**
  Here we expose each individual of our parent population to this mutant vector to create *n trial vectors.*

- **Selection:**
  The fitness of these trial vectors/individuals is compared with that of their parent individual. If the trial vector is fitter or produces a lower error when deployed in the model it replaces its parent, otherwise it is discarded. This is the most crucial point as **selection step ensures that at every step, fitter individuals are propagated to the next generation, thus creating an overall fitter population.**

These three steps are repeated several times, till we achieve a considerably fit population or in other words a set of optimal solution vectors. See Fig 5.
A convergence has been observed towards the best solution set over successive iterations. Why? Well, we really don't have answers when nature herself is behind an algorithm.

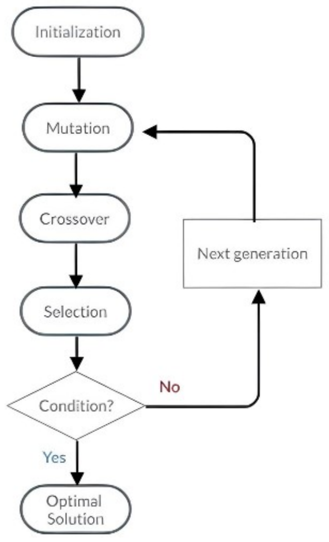


Figure 5: Flow of the Differential Evolution Algorithm

We now have a high-level understanding of how the DE algorithm works. The entire code for this question can be found in the FD_model.ipynb notebook and the associated functions in utils.py file.

3.2 Visualizations and Results

To begin with, we choose the corridor data with N = 230 participants, denoted as R (R for Ring), and the bottleneck data with bottleneck width w = 0.95m, denoted with B, as our datasets.
Also, a rather simple approach is adopted for calculating the performances associated with a set of parameters in the A/B configuration. Here the best parameters (l,v_0, and T) are calculated by using all data of the A dataset, and then using all data of B dataset for evaluating the error. Note that A and B could be same or even composite datasets like $R + B$. We first try visualizing the trajectory of two random pedestrians.

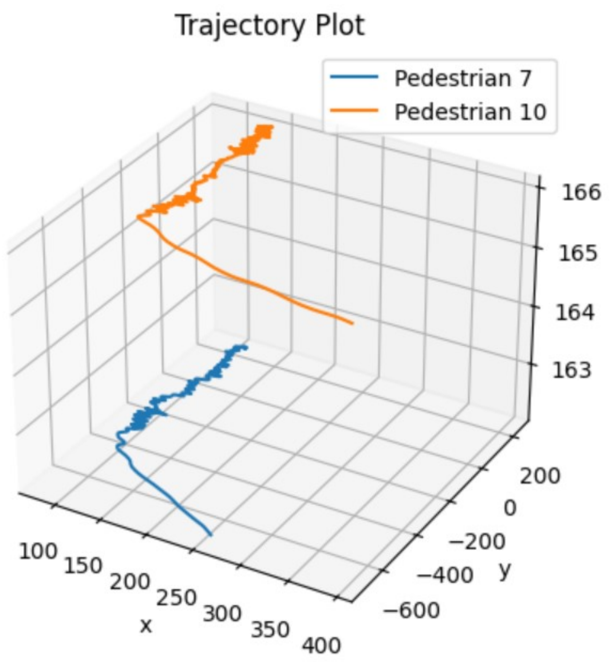


Figure 6: Trajectories of 2 pedestrians in R dataset

To implement differential evolution algorithm we use Python's `scipy.optimize.differential_evolution()` function, which when fed the loss function, bounds vector, input X and ground truth y, tries finding the best set of parameters.

Configuration	$l(m)$	$v_0(m/sec)$	$T(sec)$
R	0.613	1.576	1.226
B	0.396	2.028	0.969
R+B	0.603	1.502	1.323

Table 1: Optimal Parameters found for Different Configurations (Wiedmann Model)

These parameters when fitted to data, produce the following fitted curves. See Fig 7.
We now lay down the errors for each of the model both in tabular and graphical manner, see Fig 8.

Table 2: Errors for Different Configurations (Wiedmann Model)

Experiments	Error (MSE)
R/R	0.04632941793885659
B/B	0.03530815783956502
R/B	0.04749979348493665
B/R	0.06692780225226701
R+B/R	0.047082287725988006
R+B/B	0.04222987021644648
R+B/R+B	0.04581644418560177

Thus, the Wiedmann model was a simple approach to model this crowd behavior with only 3 parameters. We further advance to neural networks for more complex modelings and more accurate predictions.

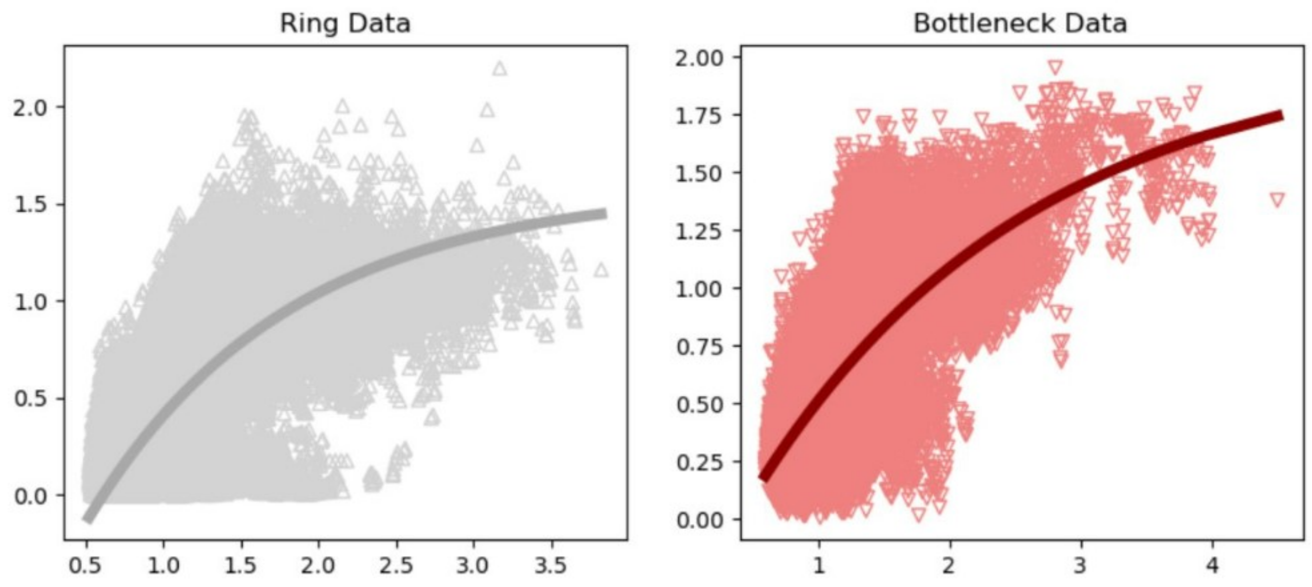


Figure 7: Fitted Curves (Wiedmann Model)

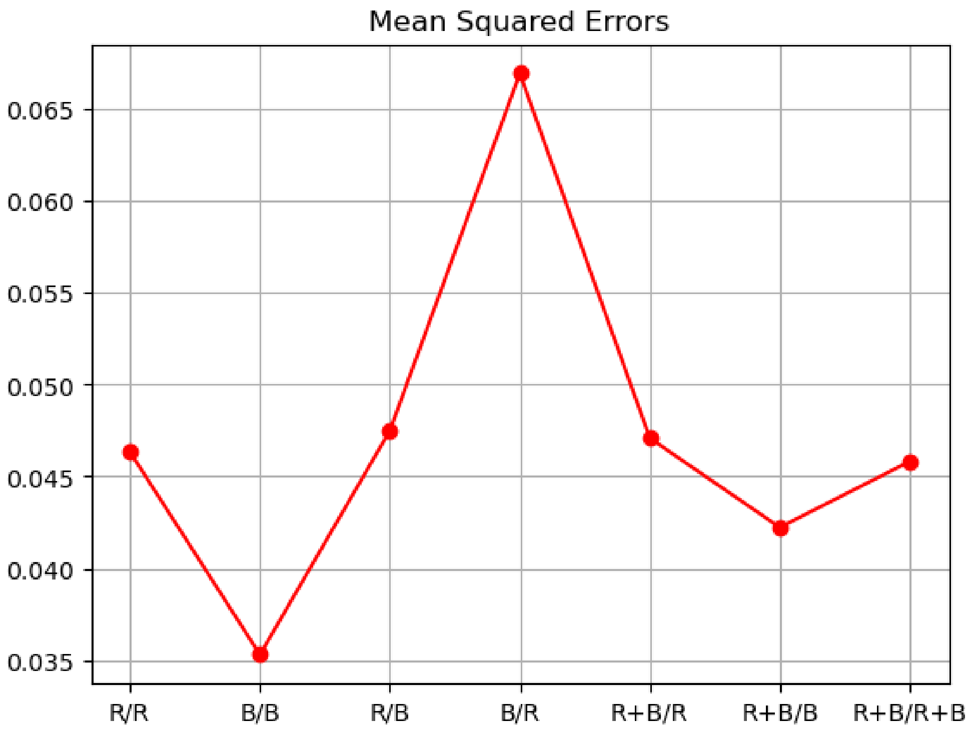


Figure 8: Errors for Different Configurations (Wiedmann Model)

Report on task 4, Artificial Neural Networks

In the previous part, we have seen that the fundamental diagram based model has very few parameters, actually only 3 parameters (the time gap, the pedestrian size, and the desired speed), to adjust the model, which limits the model's capability. We know that the neural network (NN) model can have much more parameters, which provides us an alternative method for this task. In this part, we will talk about a feed-forward neural network model which predicts the speeds of pedestrians in different scenarios, i.e. the ring and the bottleneck, and validate its feasibility. We will first introduce the model in detail, and then discuss how do we implement it, and finally give the test results of this model and compare to the FD model to see if NN can improve the performance.

4.1 Neural Network Model Introduction

In this section, we will sequentially provide the model description, introduce the model architecture, and explain the loss function.

4.1.1 Model Description

For a single pedestrian, the NN model takes the positions of its K nearest neighbors and the mean spacing as inputs, and outputs a continuous real number as its speed, so this model is a regression model. The following gives the mathematical expression:

$$v = \mathrm{NN}\left(H, \bar{s}_K, (x_i - x, y_i - y, 1 \leq i \leq K)\right), \tag{7}$$

where H denotes the number of hidden layers, and likewise $\bar{s}_K$ is the mean spacing, and the coordinates (x_i, y_i) are positions of the nearest neighbors. Here we set $K = 10$, so we have $2K + 1 = 21$ inputs.

4.1.2 Model Architecture

Tordeux's paper [4] suggests that the architecture with a single hidden layer with 3 nodes is appropriate for this task, while more complex architecture will lead to overfitting. Therefore, we only implement the same architecture, which is shown in Fig. 9.

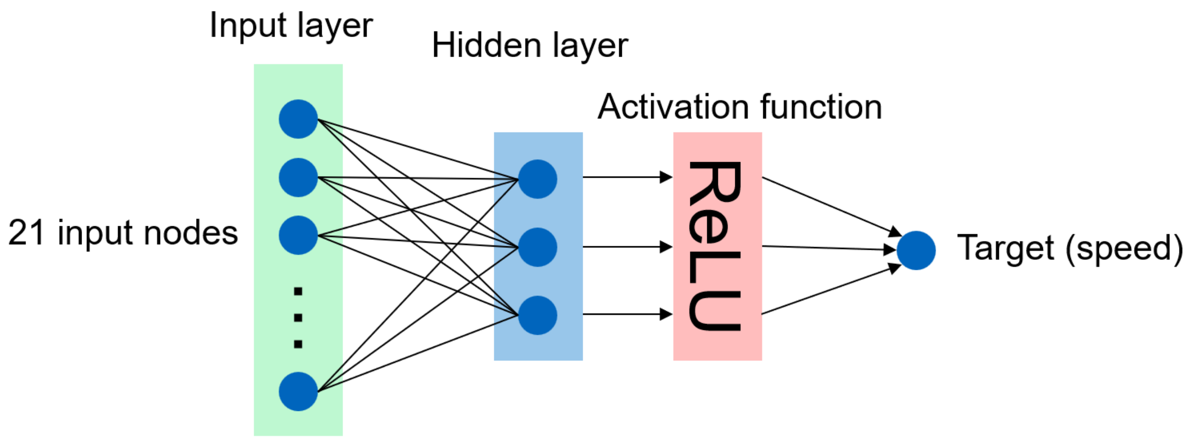


Figure 9: The architecture of the neural network.

We can see that, the 21 inputs first go from the input layer to the hidden layer. Then after the only hidden layer, we choose Rectified Linear Unit (ReLU) as our activation function, which plays a key role in the architecture, because it makes the neural network nonlinear. Finally, the speed is calculated as a scalar.

Now let's compute how many parameters are there in this model:

1. **From the input layer to the hidden layer**: Weights: $21 \times 3 = 63$, and biases: 3. Total: $63 + 3 = 66$.

2. **From the hidden layer to the output layer**: Weights: $3 \times 1 = 3$, and biases: 1. Total: $3 + 1 = 4$.

In total, there are $66 + 4 = 70$ parameters in this single-layer NN, which is much more than 3 parameters in FD model. As a result, this NN model is supposed to capture more complex patterns in the motion of crowds, and thus achieve better performance when predicting the speeds of pedestrians. It should be noted that, this model architecture is very simple, and only has one hidden layer. This implies an advantage of NN, which is that if we have more training data, we can simply add more layers to let the model have more parameters, so that the model won't be underfitting. The reason why we can directly add more parameters is that these parameters do not have explicit physical meanings, and they are just to help add the flexibility to fit the function. However we cannot easily do this to the FD model, because all the 3 parameters have physical meanings. If we have more data, 3 parameters are not sufficient to capture the intrinsic patterns. But if we want to add more parameters, we need to find new parameters with explicit physical meanings which may affect the speed, and try to incorporate them in the formula to calculate the speed, and meanwhile ensure the validity of the formula, which might be challenging.

4.1.3 Loss Function

Now we have established the model, and we know this model has 70 parameters. These parameters will be optimized by minimizing some loss function. Here we choose the mean square error (MSE) as our loss defined as below:

$$MSE = \frac{1}{N}\sum_{i}(v_i - \tilde{v}_i)^2, \tag{8}$$

where v_i is the observed speed, and $\tilde{v}_i$ is the predicted speed, and N is the number of observations. MSE measures the average of the squares of the differences between the predicted and the actual target values. Using MSE as the loss function for a feed-forward regression network has several advantages:

- **Ease of Understanding and Implementation**: MSE has a straightforward form, representing the magnitude of prediction errors intuitively;
- **Quadratic Punishment**: MSE penalizes larger errors more severely due to the squaring term, meaning the model will try harder to reduce larger errors, which can enhance the predictive accuracy of the model;
- **Differentiability**: The MSE function is differentiable across its entire domain, which is an important property for optimization using gradient descent algorithms.

4.2 Code Implementation

In this section, we will explain how do we organize the workflow of the code, including the preparation of the Dataloader, the model implementation, and the training loop implementation.

To start with, in order to obtain convenient maintainability and readability, we store the configuration parameters in a `config.yaml` file which can make the configuration more clear and readable. Besides, this will also allow us to quickly modify the hyperparameters without manually changing the code.

4.2.1 Preparation of Dataloader

As mentioned before, for the 2 scenarios corridor and bottleneck, they both have multiple experiments with different settings. In order to have a fair comparison with the FD model, we also choose the same dataset, i.e. the corridor data with $N = 230$ participants, denoted as R (R for Ring), and the bottleneck data with bottleneck width $w = 0.95m$, denoted as B. Then we split each dataset into half, and use one half for training, the other half for testing.

Next, we define the `PedestrianDataset` class that inherits from PyTorch Dataset, and it converts the features and targets into PyTorch tensors. We instantiate the `PedestrianDataset` to create the training and testing set. Then we create the `train_loader` and the `test_loader` based on respective datasets. We allow users to decide if to use mini-batch through the item `use_batch` and set the `batch_size` in the configuration file. Here, however, we set `use_batch` to `False`, meaning we use the whole epoch as one batch, and thus we only go one step by running one epoch during optimization.

4.2.2 Model Implementation

We build a simple feed-forward neural network with PyTorch. The input layer expects a fixed size of 21 features, since we have set $K = 10$, and the output layer consists of a single neuron, representing the predicted speed. We construct the network dynamically, allowing users to specify the number of layers and nodes per layer, which enables easily performing experiments with different architectures. Here we only conduct the experiment with the architecture shown in Fig. 9.

4.2.3 Training Implementation

To obtain an overall evaluation of the performance of the model, we decide to first perform 10-fold cross validation. More specifically, we split the training set into 10 folds with roughly same size. Then for each run, we select one fold as the test set, and train on the remaining. Since we don't use mini-batch, we set the number of epochs to a relatively large number, i.e. 2000 here. After training the model, we test it on the one-fold test set, getting the MSE loss and log it. We need to perform this process 10 times, and thus we get 10 losses. We take the average of them as the overall performance.

The training of the cross validation only uses 90% of the training data, and it would be better if we make use of all the training data. Therefore, we decide to perform another training after the cross validation using the whole training set. After training, we test the model on different test sets. The experiments we have performed are listed as (training set/test set): R/R, B/B, R/B, B/R, R+B/R, R+B/B, R+B/R+B. So basically we test on the same and the other dataset. For the model trained on the mixed dataset, we test it on each single test set, and their combination.

4.3 Results of the experiments

We use Weights & Biases to log all kinds of loss values during training. In this section, we will show all the relevant results.

4.3.1 Cross Validation

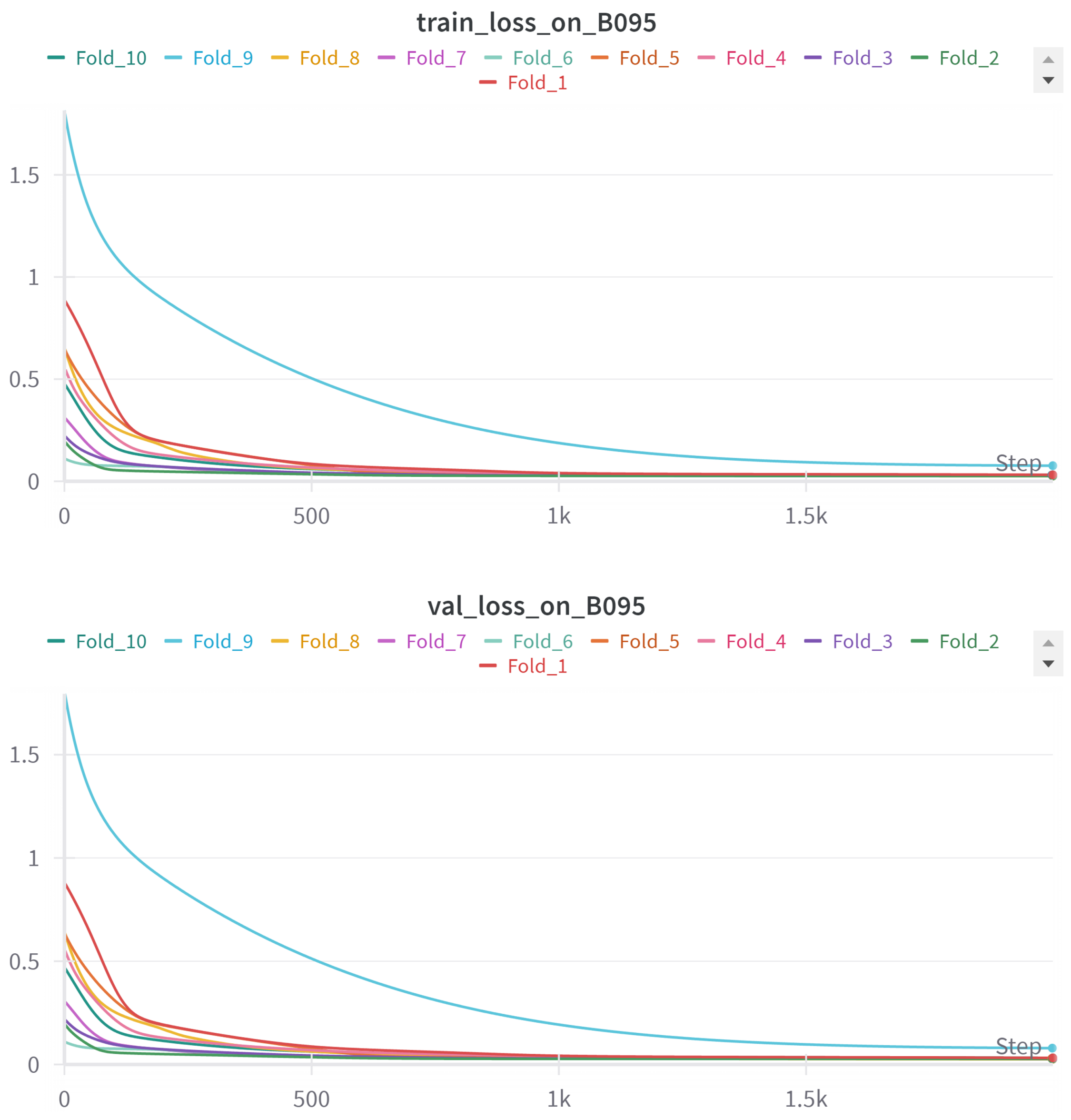


Figure 10: The training and validation loss curves of all runs in cross validation.

Fig. 10 shows the training and validation loss curves of all runs in the cross validation experiment training on dataset B. We can see that, under most circumstances, the training works well. But for `Fold_9`, the training and validation loss are noticeably higher than the others. This may be due to bad initialization, which causes the training got stuck in some larger local minimum. As for the average losses of the cross validation, they are shown in Table 3. We can find that, the losses of the model trained on the mixed dataset are comparably higher. One possible reason is that, the model tries to fit the data from two different experiments, which may follow various patterns, so the model has to make some trade-off. Nevertheless, we can still try to train the models multiple times when we train on the full training set, because there may be some initialization which will end up with lower error.

Table 3: Cross validation results

Training set	Average training loss of all folds	Average validation loss of all folds
R	0.031218636967241763	0.03142208959907293
B	0.03319365195930004	0.03388847634196281
R+B	0.05610250122845173	0.056185453338511467

4.3.2 Training on the full datasets

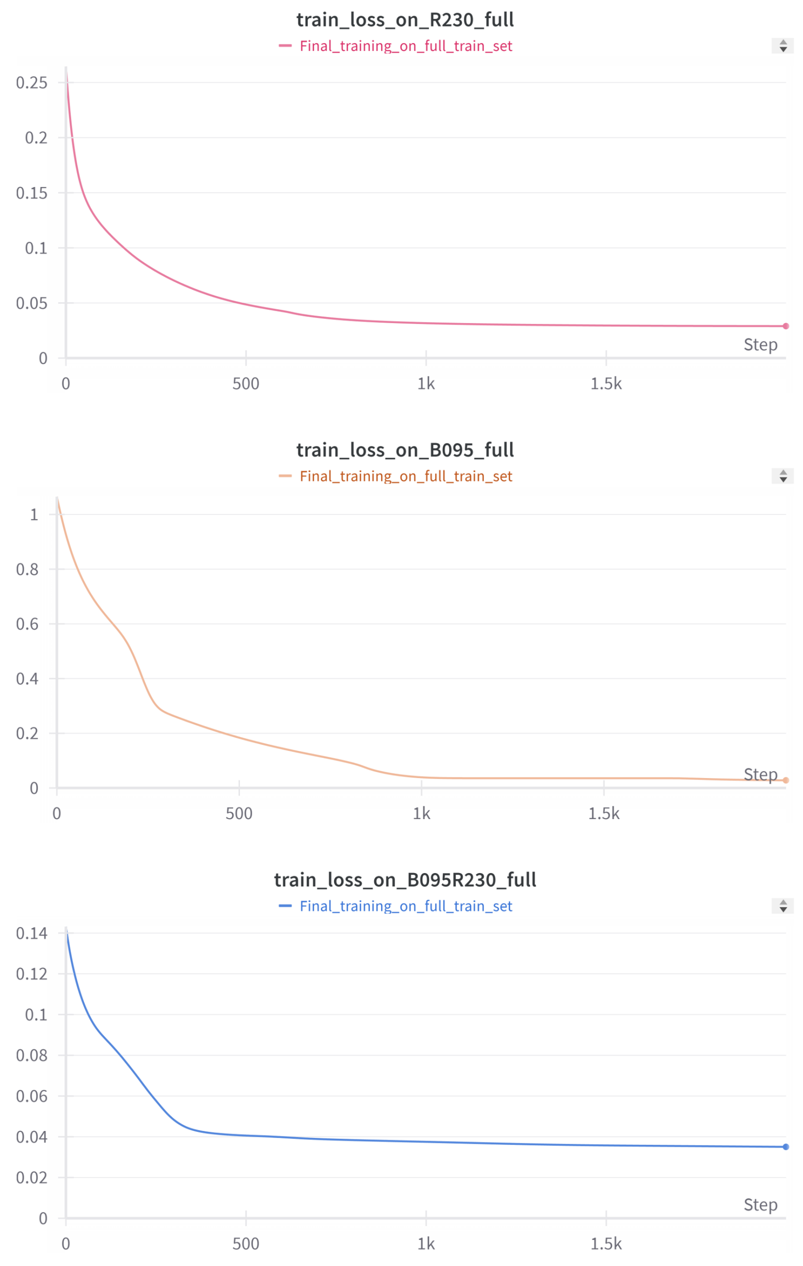


Figure 11: The training loss curves of training on dataset R, B, and R+B.

After we finish the cross validation, we perform the training on the full dataset to make use of all the data. We have tried multiple times to obtain models with relatively good results. Fig. 11 shows the training loss curves of experiments with dataset R, B, and R+B, respectively, and the test error of all the experiments is shown in Table 4.

Table 4: Test error of all the experiments

Experiments	Test error (MSE)
R/R	0.02871980145573616
B/B	0.027995677664875984
R/B	0.04080035910010338
B/R	0.056587666273117065
R+B/R	0.032848604023456573
R+B/B	0.040615957230329514
R+B/R+B	0.034875381737947464

4.4 Comparison and Discussion

The test error of the FD and NN from the paper is shown in Fig. 12. We also visualize the test error of our own implemented models as shown in Fig. 13.

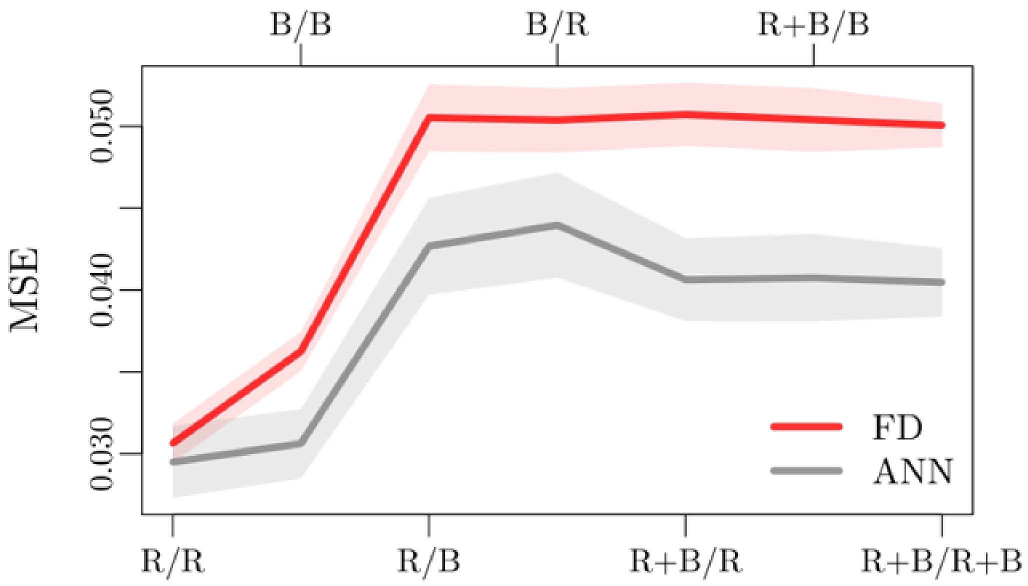


Figure 12: The test error of FD and NN from Tordeux's paper [4].

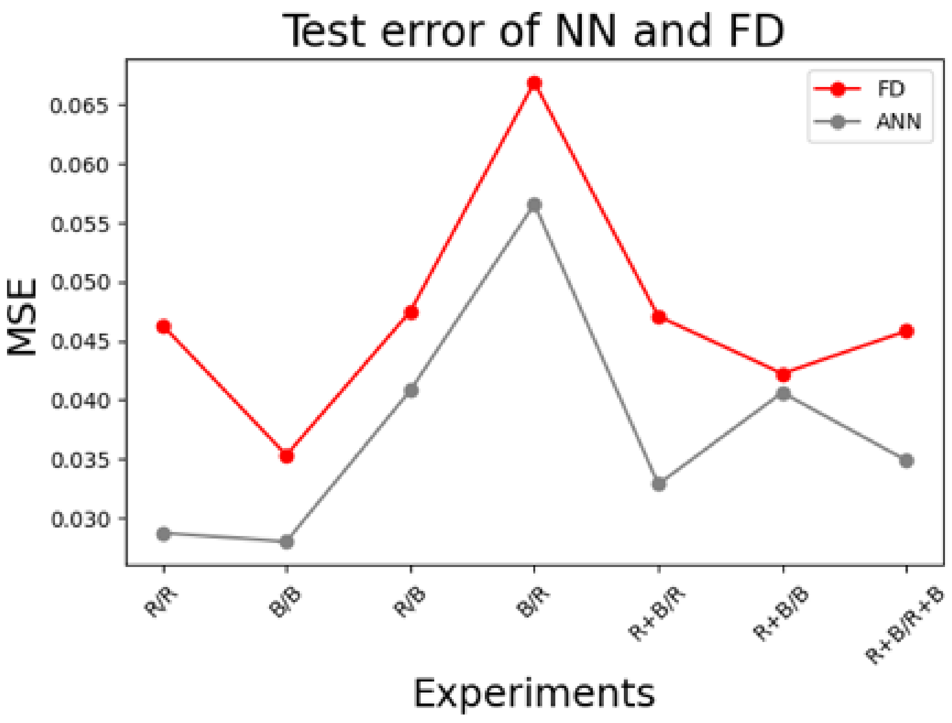


Figure 13: The test error of our implemented FD and NN.

We can see that, although the test error values are not exactly the same, we still can draw a similar conclusion, since in both figures, the ANN achieves lower error in all the experiments. So we can say, the ANN model outperforms FD model in predicting speeds of pedestrians, and is able to distinguish two different geometries, i.e. ring and bottleneck.

Compared to the FD model, ANN has the following advantages:

1. Generally, ANN can have more parameters, thus is more likely to capture more complex patterns or deeper level of crowd dynamics;
2. ANN is more ready to further enhancements. We can simply add more layers to help ANN fit more data. For the FD model, however, we need to modify the formula to let the model have more parameters, which might be challenging. Even if we can modify the formula successfully, the number of newly added parameters could hardly be as many as ANN has.

Report on task 5, Modified Neural Networks

5.1 Modified NN architecture description

The Neural Network architecture was modified to produce slightly better results in terms of accuracy. As in previous section, this implementation was also done using Python in pytorch. The input Dataset Files-Bottleneck with 0.95m width, Ring corridor with 230 participants are used. The Dataset Loading with Shuffling, 50%-50% Training-Test data split is done. The salient features of the model architecture are:

- Fully Connected Network (Input dim. 21, 1 hidden layer with 3 nodes, 1 output node)
- Target points, predicted velocity
- Activation function: Relu
- Mini-batches, Batch normalization - Achieve stable gradients, better learning

- Adam optimizer with exponential weight decay: weights of the model are decayed exponentially over time during training. It helps prevent overfitting and provides better generalization
- Hyperparameters: ["batch_size" : 10,"learning_rate" : 0.005,"decay" : 0.0001, "epochs" : 10]
- Loss func : MSE loss as mentioned in the paper

5.2 Results

5.2.1 Comparison of Models

The Testing errors of the FD-model, Neural Network (described in previous section) and the Modified Neural Network have been shown in fig. 15 for different dataset combinations R/R, B/B, R/B, B/R, R+B/R, R+B/B, R+B/R+B. The trend of testing errors of the Modified NN matches quite well with that of given in the paper (fig. 14).

As clearly evident from the fig 15, both the neural networks outperform the FD model for all experiments. Moreover, the Modified NN performs slightly better (or atleast somewhat similar) than our previous NN model. The testing errors for R/R and B/B were found to be the least which is convincing as the training and testing datasets are from the same distribution. Hence, this also explains why the testing errors for dataset combination R/B and B/R is somewhat greater than R/R or B/B. Finally, the improvement of the speed is bit significant by using the network when the experiments are mixed (i.e. R+B).

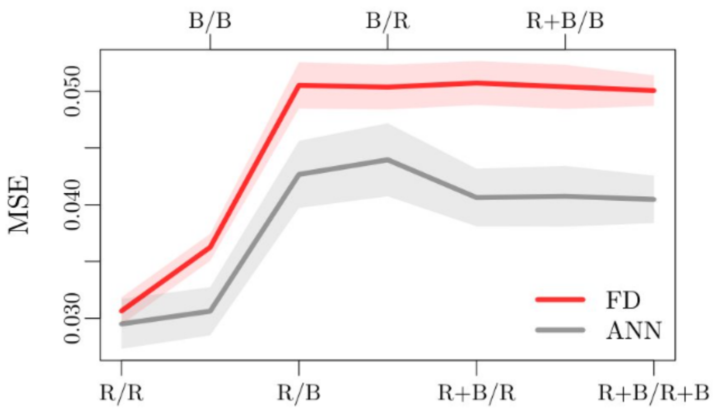


Figure 14: Testing errors for FD, NN from paper

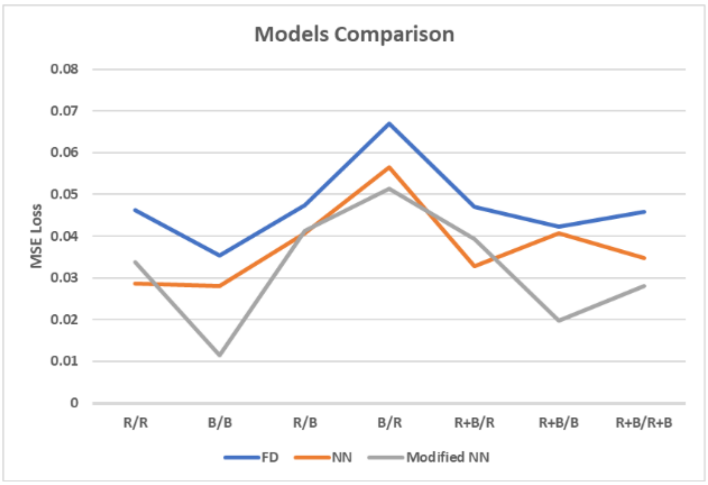


Figure 15: Testing errors for FD, NN, Modifed NN

5.2.2 Pred. Speed comparison for Ring and Bottleneck

In the fig 16, the speed predictions by the modified neural network of the pedestrians for the R+B/R+B training and testing datasets have been presented. As observed in this figure, the speed for a given mean spacing is in average in the bottleneck (fitted red curve) higher than the flow in the corridor(fitted blue curve) for congested situations. It shows that this network is able to, at least partially, identify the two geometries. As observed in the real data, the speed for a given mean spacing is in average in the bottleneck higher than the flow in the corridor for congested situations.

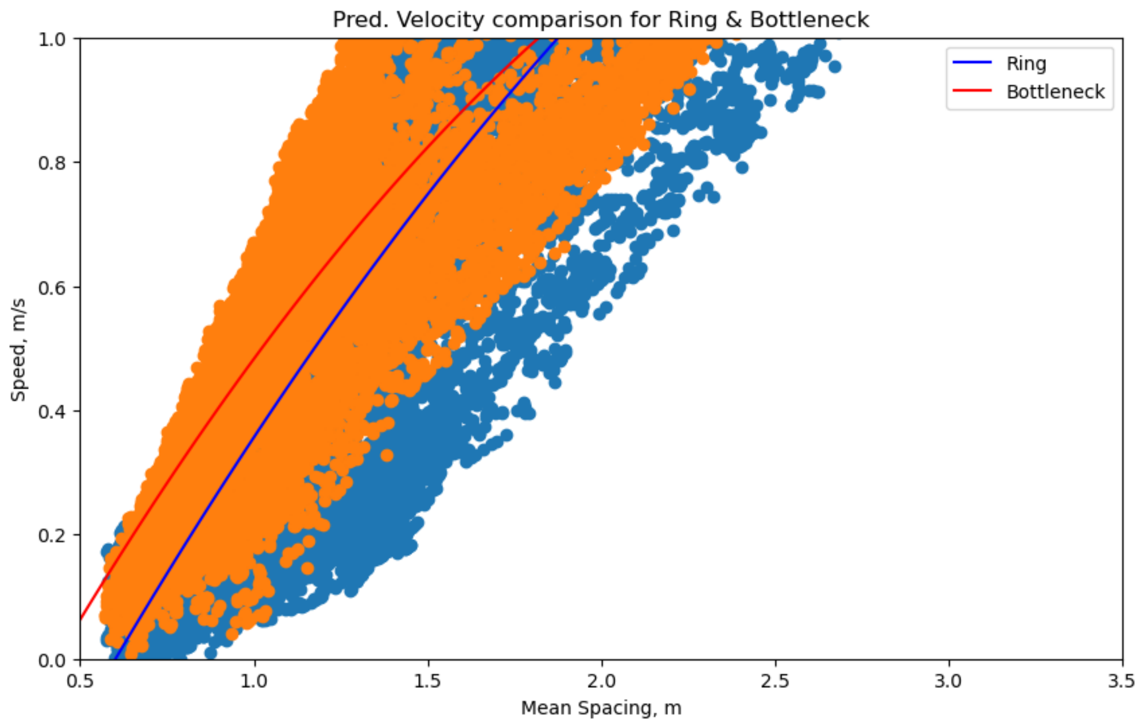


Figure 16: Prediction of Speed for Ring and Bottleneck

References

[1] Armin Seyfried, Bernhard Steffen, Wolfram Klingsch, and Maik Boltes. The fundamental diagram of pedestrian movement revisited. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(10):P10002, oct 2005.

[2] Keip C, Ries K. Dokumentation von Versuchen zur Personenstromdynamik “[J]. Project Hermes, Bergische Universität Wuppertal, Tech. Rep, 2009.

[3] Tordeux, A., Chraibi, M., Seyfried, A., Schadschneider, A.: Data from: Prediction of Pedestrian Speed with Artificial Neural Networks (2017). DOI 10.5281/zenodo.1054017. URL `https://doi.org/10.5281/zenodo.1054017`

[4] A. Tordeux, M. Chraibi, A. Seyfried, and A. Schadschneider,“Prediction of pedestrian speed with artificial neural networks,” in Proc. Int. Conf. Traffic Granular Flow, Jul. 2017, pp. 327–335.