Report for final project from group K

Tasks addressed:	4
Authors:	Oliver Beck (03685783)
	Junle Li (03748878)
	Chenqi Zhou (03734992)
Last compiled:	2021–07–14
Source code:	https://github.com/Linnore/MLCMS-Final-Project

The work on tasks was divided in the following way:

Oliver Beck (03685783)	Task 1	25,0%
	Task 2	42,5%
	Task 3	25,0%
	Task 4	33,3%
Junle Li (03748878)	Task 1	25,0%
	Task 2	37,5%
	Task 3	50,0%
	Task 4	33,3%
Chenqi Zhou (03734992)	Task 1	50,0%
	Task 2	20,0%
	Task 3	25,0%
	Task 4	33,3%

Abstract

In this project, we learned about the challenges, when applying neuronal networks to crowd modeling scenarios. We created all the tools necessary from the different steps from data processing to neuronal network designs to apply the methods of Tordeux et al. (2019) to simulation data from Vadere and also the real data sets used in the paper itself. We show our proceeding and difficulties being inexperienced in the field of machine learning.

Python	3.7.1
Jupyter Notebook	6.3.0
Visual Studio Code	1.58.0
IntelliJ IDEA	2021.1.1 (Community Edition)
Vadere Crowd Simulation	1.15
pytorch	1.7.0 + cu101
pytorch-lightning	1.3.8
tensorboard	2.4.1

Table 1: Software versions

Report on task 1, Summary of the paper Tordeux et al. (2019). (20 points)

1 Introduction Traffic engineers frequently use relative simplicity microscopic physics-based models to predict crowd dynamics, which consider physical, social or psychological factors and can describe realistic pedestrian flows and observed self-organisation phenomena.

However, accurate predictions of pedestrian dynamics in complex spatial structures remain difficult. Recent experiments have shown that the behaviour of pedestrians tends to depend on the type of facility. For instance, flows at bottlenecks often exceed the maximal rates observed in straight corridors. This makes pedestrian behaviours geometry-dependent. Yet the types of geometries are various and not precisely defined. Their systematic identification in complex buildings is ambiguous.

This paper tests the assertion that artificial neural networks (ANN) could be a suitable alternative for forecasts of pedestrian dynamics in complex architectures, which are able to identify various types of patterns without supervision. The objective of this article is to evaluate whether neural networks could accurately describe pedestrian behaviours for two different types of facilities, namely a corridor and a bottleneck. Feed-forward networks for the prediction of pedestrian speeds in corridor and bottleneck experiments are developed, trained and tested. A physics-based model as classical operational approach is used for comparison as a benchmark. The results show that neural networks distinguish the flow characteristics for the two different types of facilities and significantly improve the prediction of pedestrian speeds. The performances significantly differ according to the geometry.

2 Speed Model and Artificial Neural Networks The aim is to predict the speed of pedestrians according to the relative positions of the $K = 10$ closest neighbours. One denotes in the following (x, y) as the position of the considered pedestrian, v as its speed, and $((x_i, y_i), i = 1, ..., K)$ as the positions of the K closest neighbours.

Speed-Based Model (Weidmann's model)

The physics-based modelling approach is the Weidmann fitting model for the fundamental diagram. The Weidmann's model and its parameters are used for comparison as a benchmark. In the Weidmann's model, the speed of a pedestrian is a non-linear function of the mean spacing with the closest neighbours:

$$FD(\bar{s}_K, v_0, T, l) = v_0(1 - exp(\frac{l - \bar{s}_K}{v_0 T})).$$

Here

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

is the mean spacing distance to the K closest neighbours that used to approximate the local density. The Weidmann's model has three parameters: The time gap T, corresponding to the following time gap with the

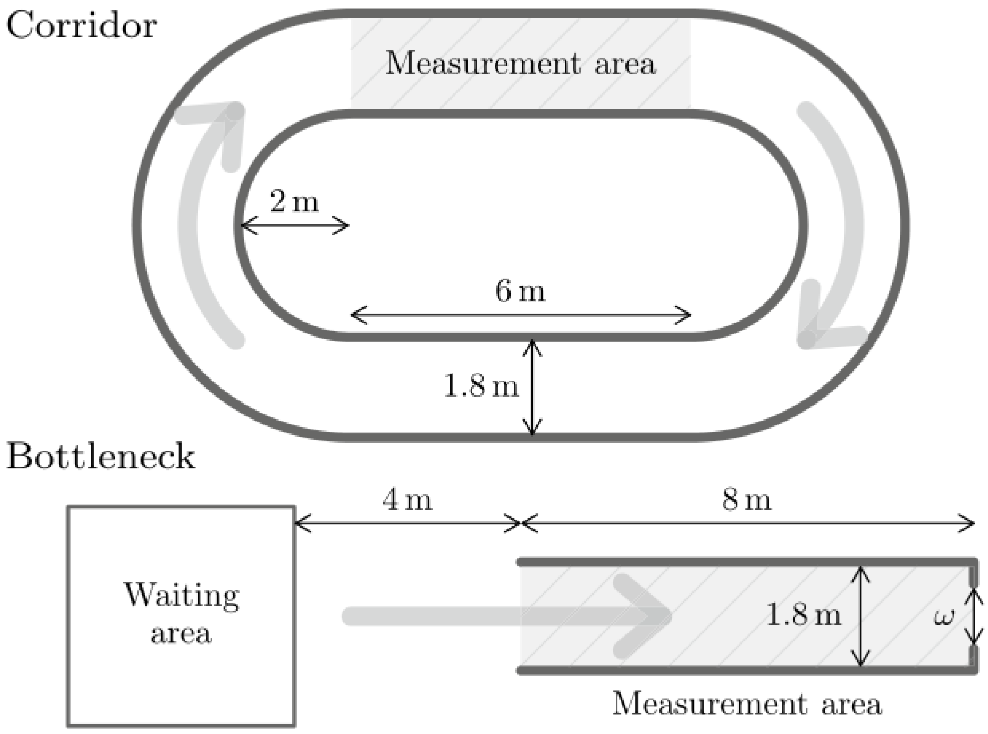


Figure 1: Top panel: Scheme for the corridor experiment(C). Bottom Panel: Scheme for the bottleneck experiment(B).

neighbour in front; The pedestrian speed in a free situation, also called the desired speed and denoted v_0; The physical size of a stopped pedestrian l.

Artificial Neural Networks

The data-based modelling approach for prediction of the pedestrian speed are feed-forward neural networks with hidden layers h. Two networks with different inputs are tested:

In the first network, the inputs are the relative positions to the K closest neighbours ($2K$ inputs)

$$NN_1 = NN_1(h, (x_i - x, y_i - y, 1 \leqslant i \leqslant K)).$$

In the second network, the speed is predicted as function of the relative positions and the mean distance spacing $\bar{s}_K$ to the K closest neighbours ($2K + 1$ inputs)

$$NN_2 = NN_2(h, \bar{s}_K, (x_i - x, y_i - y, 1 \leqslant i \leqslant K)).$$

3 Empirical Data Two experiments are used to calibrate, train, test and compare the physics-based model and the artificial neural networks. In the first experiment, the pedestrians walk through a corridor while in the second they pass a bottleneck.

Corridor and Bottleneck Experiments

The first dataset comes from a unidirectional experiment done in a corridor of length 30m and width 1.8m with periodic boundary condition (see Figure 1, top panel). The trajectories were measured on a straight section of length 6m. Eight experiments were carried out with different numbers of participants for different density levels. The second dataset is an experiment at bottlenecks (see Figure 1, bottom panel). The width of the corridor in front of the bottleneck is 1.8m with different bottleneck widths in 4 distinct experiments involving 150 participants each.

Data Analysis

The speed/mean spacing data sets in the corridor and at the bottleneck describe two slightly different interaction behaviours (see Figure 2). The speed for a given mean spacing is in average higher in the bottleneck

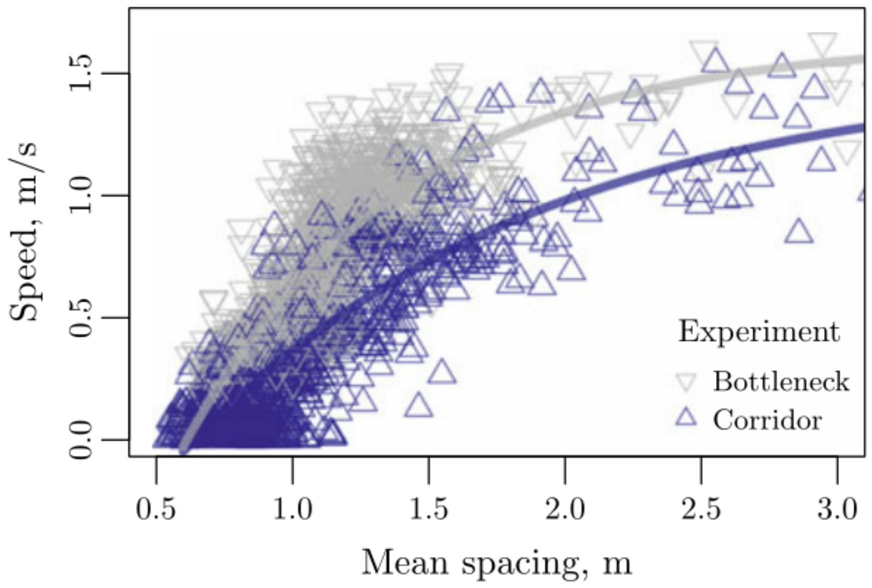
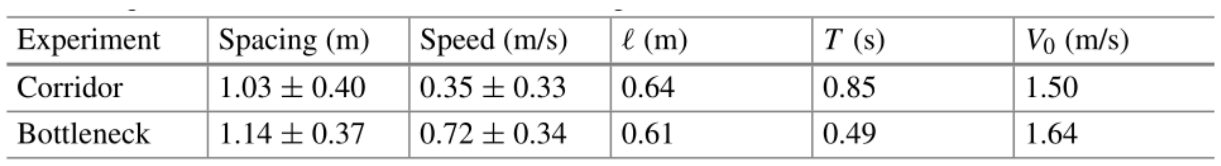


Figure 2: Pedestrian speeds as function of the mean distance spacing with the $K = 10$ closest pedestrian neighbours for the corridor and bottleneck experiments and their respective fitting with Weidmann's model.



Experiment	Spacing (m)	Speed (m/s)	ℓ (m)	T (s)	V_0 (m/s)
Corridor	1.03 ± 0.40	0.35 ± 0.33	0.64	0.85	1.50
Bottleneck	1.14 ± 0.37	0.72 ± 0.34	0.61	0.49	1.64

Figure 3: Mean value and standard deviation for the speed and the spacing, and least squares estimations for the pedestrian size l, the time gap T, and the desired speed v_0 parameters of Weidmann's model for the corridor and bottleneck experiments

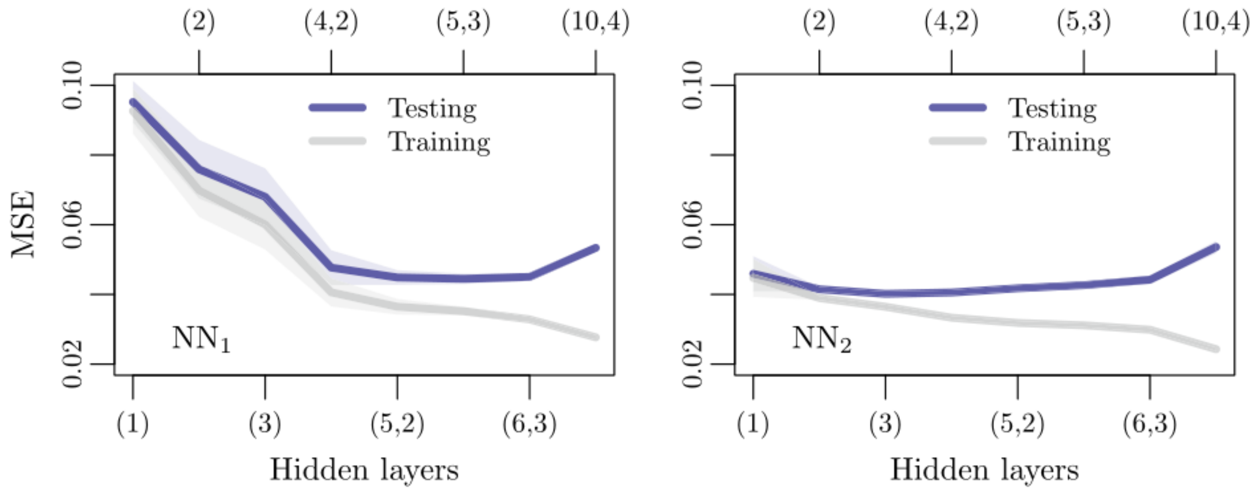


Figure 4: Training and testing errors according to different hidden layers in the networks.

than in the corridor experiment. Estimations by least squares of the time gap T and the desired speed v_0 for Weidmann's model significantly differ according to the experiment (see Figure 3). The pedestrian size l remains approximately constant. Note that the mean spacing is around 10% smaller in the corridor. However, the mean speed is more than two times larger in the bottleneck.

4 Predictions for the Speed The coefficients of the neural networks and the three parameters of the physics-based model are estimated by minimising the mean square error

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(v_i - \widetilde{v}_i)^2.$$

Here v_i are the observed speeds, while $\widetilde{v}_i$ are the predicted speeds and N is the number of observations.

Setting the Network Complexity

The complexity (hidden layer h) of the neural networks is determined through training and testing phases (cross-validation). Eight different hidden h are tested. The training and testing MSE for the full dataset combining the corridor and bottleneck experiments are presented in Figure 4. As expected, the training error systematically decreases as the complexity of the network increases, while the testing error presents a minimum before overfitting. This minimum is reached for the single hidden layer $h = (3)$ for the network NN_2 based on mean distance spacing and relative positions. While it is reached for $h = (5, 2)$ for the networks NN_1 solely based on the relative positions.

5 Conclusion Artificial neural networks for the prediction of pedestrian dynamics in two different walking situations, namely a corridor and a bottleneck, are developed. The data-driven approach is able to distinguish pedestrian behaviours according to the facility. The predictions for mixed data combining both the corridor and bottleneck experiments are improved by a factor up to 20% compared to a classical physics-based model. Furthermore, predictions in case of new situations are also significantly improved by a factor up to 15%, attesting for the robustness of the networks. Adding the mean spacing in the input of the networks, even if it is calculated by the relative positions, significantly increases the quality of the prediction.

Report on task 2, Generate crowd data from experimental data and Vadere (40 points)

Required data for the neuronal network For the Neuronal Network we need certain input that can't be generated by Vadere's default output processors. For the first Neuronal Networks described in summary of the paper we need the relative positions $(x_i - x, y_i - y)$ to the K closest neighbors. For the second network we need the relative positions $(x_i - x, y_i - y)$ and the mean distance spacing $\overline{s}_k$ to predict the pedestrian speed.

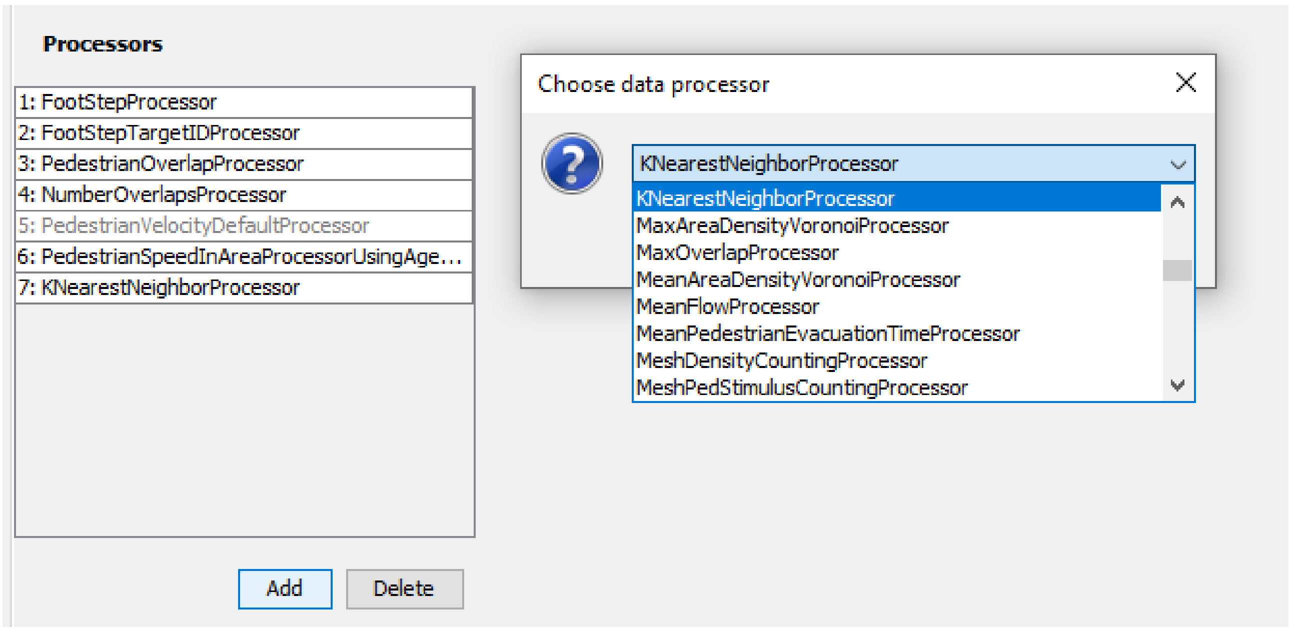

Processors
1: FootStepProcessor
2: FootStepTargetIDProcessor
3: PedestrianOverlapProcessor
4: NumberOverlapsProcessor
5: PedestrianVelocityDefaultProcessor
6: PedestrianSpeedInAreaProcessorUsingAge…
7: KNearestNeighborProcessor
Add
Delete
Choose data processor
KNearestNeighborProcessor
KNearestNeighborProcessor
MaxAreaDensityVoronoiProcessor
MaxOverlapProcessor
MeanAreaDensityVoronoiProcessor
MeanFlowProcessor
MeanPedestrianEvacuationTimeProcessor
MeshDensityCountingProcessor
MeshPedStimulusCountingProcessor

Figure 5: The kNearestNeighborProcessor we introduced with the default setting for our outputs

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

Vadere data

Existing processors in Vadere While the crowd simulation software Vadere has various output processors, that seem similar to the k-nearest neighborhood we need to create our own processor. The PedestriansNearbyProcessor for example comes close, but defines neighborhood as being within a certain radius, which absolutely makes sense in, for example a pandemic scenario, but isn't applicable for the input described in the paper being the k nearest neighbors.

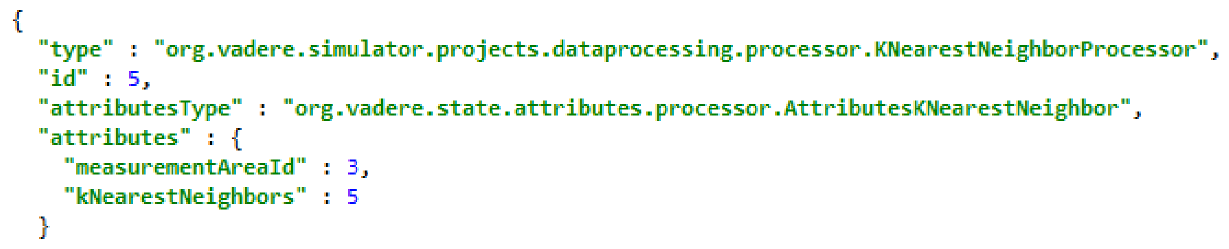
In a similar way the various density processors touch upon the subject, but don't deliver the required inputs.

We can use the existing PedestrianSpeedInAreaProcessorUsingAgentVelocity that uses the PedestrianVelocityDefaultProcessor for the Pedestrian Velocity in the Measurement area. Since it uses the same datakey TimestepPedestrianIdKey as we are using in our own processors we can connect these two into the same output file in Vadere.

Added processor in Vadere Since we experienced that there is no suitable preexisting processor in Vadere we created our own processor. We added the kNearestNeighborProcessor into Vadere. It takes the measurementAreaID, that defines the corresponding measurement area, and the number of neighbors kNearestNeighbor as inputs for the Vadere console.

Based on the requirements for the output data it would have been possible to generate all the data just by post processing on the output files. Creating an integrated processor has many advantages. First and foremost we have full access to all the computed simulation steps, that aren't represented in the less detailed nature of the output files. Also we can profit from the predefined structure that is existing for processors, their data formats and attributes and make us e of already existing functionality.

When being concerned with the k nearest neighbours for a pedestrian, we would like the pedestrian himself to be within the measurement area, but some of the k nearest neighbours will be outside of it. When solely working on the output files, we would only see measured values within the measurement areas, and therefore not correct or not entire neighborhoods. Lastly setting up the simulation run in Vadere itself, selecting the according self-defined processor alongside with the scenario creates a very satisfying workflow compared to the alternative of post processing. The only drawback is in order harvest all of these advantages, we first of all have to understand many preexisting underlying structures in the Vadere programming.



```
{
  "type" : "org.vadere.simulator.projects.dataprocessing.processor.KNearestNeighborProcessor",
  "id" : 5,
  "attributesType" : "org.vadere.state.attributes.processor.AttributesKNearestNeighbor",
  "attributes" : {
    "measurementAreaId" : 3,
    "kNearestNeighbors" : 5
  }
```

Figure 6: Json representation of the input for the k-nearest neighbor output processor

The processor uses a priority queue to store the nearest k neighbors, calculates the corresponding relative coordinates and the mean spacing distance.

Additionally to the kNearestNeighborProcessor.java we added and the AttributesProcessor AttributesKNearestNeighbor.java in order to retrieve the desired input. The last java file we added, PedestriansKNearestNeighborData, was basically the value, to fit the $Dataprocessor\langle key, value\rangle >$ type of our kNearestNeighborProcessor.

In an independent post processing step we cleared out the data points which weren't in the measurement area, had invalid speed attributes or didn't have a sufficient amount of 10 nearest neighbors.

Scenarios in the Vadere Simulation The scenarios we created in Vadere are similar to the core features of the experiments the paper used. There are some differences between the scenarios of the experimental data and the vadere generated data, we will explain in the following.

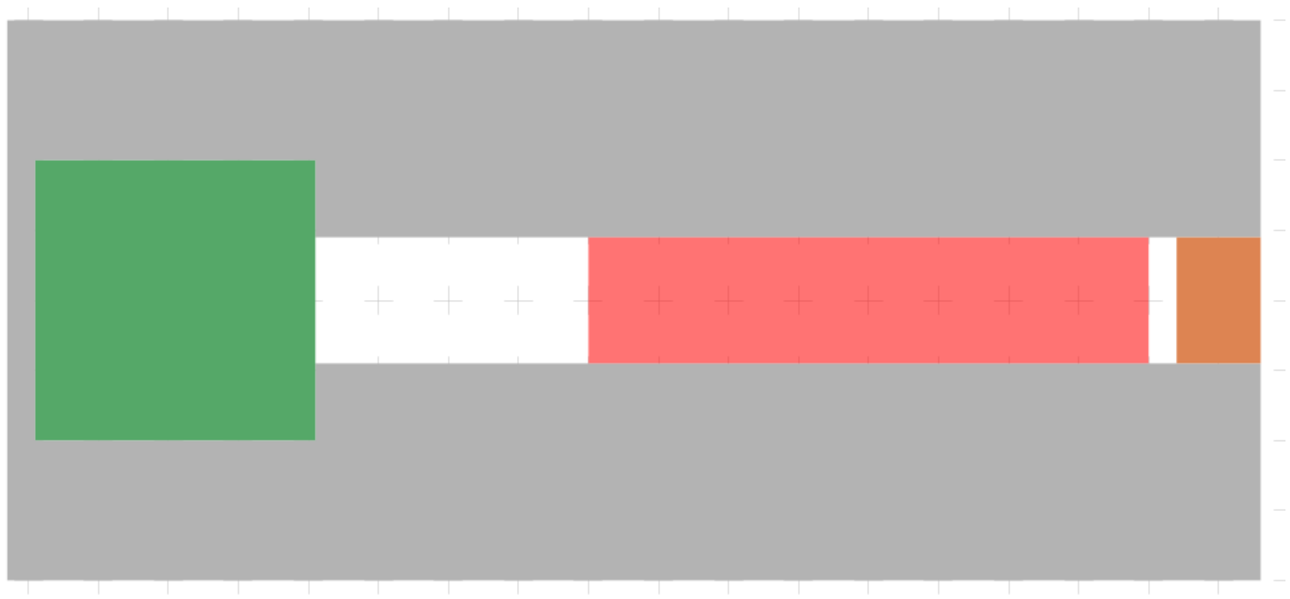


Figure 7: Scenario Bottleneck Vadere

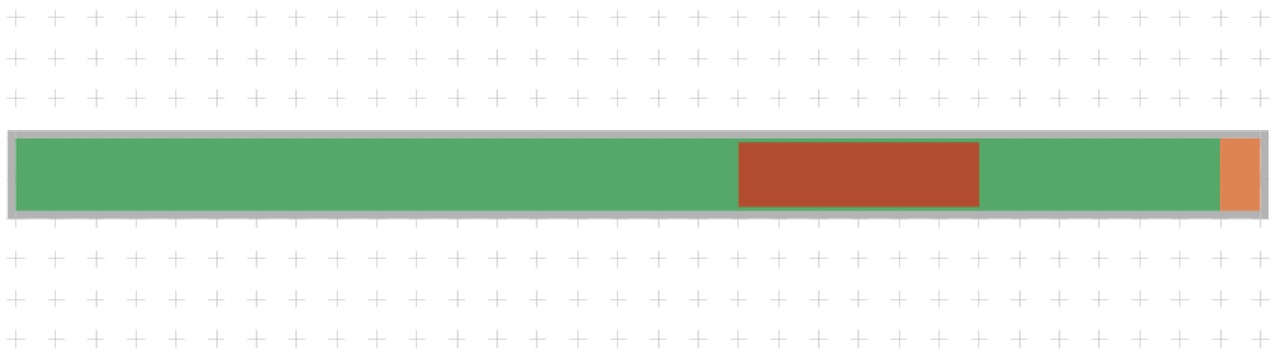


Figure 8: Scenario Corridor Vadere

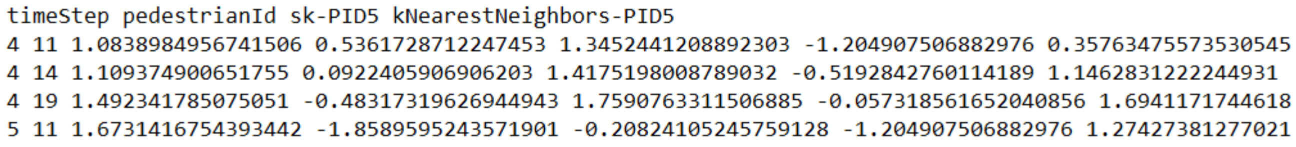
If you compare the scenarios from Vadere in figure 7 and 8 to the scenarios used in the paper 1 and 2. The most appararent difference is that we use a straight corridor example versus the paper using a loop in which the pedestrians move.

The experiment uses the loop to guarantee the respective density's of 0.25 and 2 Ped/m^2 by letting the desired amount of pedestrians into the closed pedestrian area. Vadere doesn't allow truly circular plotting, the desired measurements are in a flat, non-rounded corridor, and we can easily initialize densities to our desire. Therefore we basically "rolled the circle flat". Goal wasn't to mimic the experiment run exactly, but to produce comparable scenarios for the Simulation.

For the bottleneck scenario we placed a bottleneck behind the measurement are and before the target. It follows the structure of the experiments fairly well. Interestingly enough we had to "thicken" the walls to prevent pedestrians from escaping and taking a short cut.

Parameters that are more difficult to set revolve around the pedestrian size. In the real world this varies alot and is hard to perceive accurately, since the radius of a pedestrian isn't circular and there are various social rules around the proper distance. Vadere has different settings to accomodate for that. We choose the OptimalStepsModel with the default initialization, since think that the differenting pedestrian size and the preset personal space parameters are set reasonable.

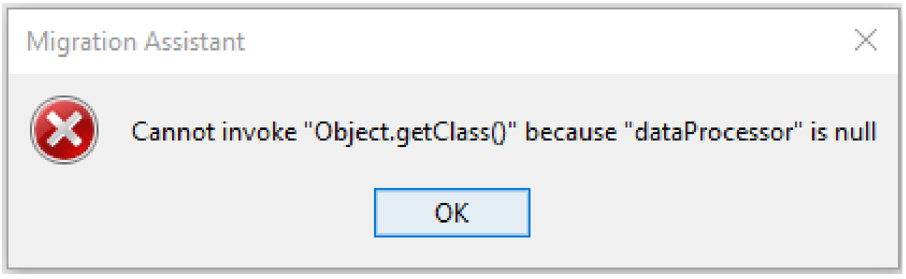
Structure of the generated data The kNearestNeighborProcessor creates a output file with the following format: The first line contains the headlines (timeStep pedestrianId sk-PID kNearestNeighbors-PID) separated by a space. All following lines are the respective datavalues, only containing the pedestrians in the measurement area at the respecting timesteps. All k-nearest neighbors relative x and y values are added at the end of the line.



```
timeStep pedestrianId sk-PID5 kNearestNeighbors-PID5
4 11 1.0838984956741506 0.5361728712247453 1.3452441208892303 -1.204907506882976 0.35763475573530545
4 14 1.109374900651755 0.0922405906906203 1.4175198008789032 -0.5192842760114189 1.1462831222244931
4 19 1.492341785075051 -0.48317319626944943 1.7590763311506885 -0.057318561652040856 1.6941171744618
5 11 1.6731416754393442 -1.8589595243571901 -0.20824105245759128 -1.204907506882976 1.27427381277021
```

Figure 9: Example output with k=2

Issues Overall understanding how to integrate a new processor takes time, even though there is a very brief documentation regarding that topic, most of the knowledge was gained by looking at preexisting processors and by trial and error. One issue that remained unresolved and really impacted our workflow negatively was that the an error that appeared every single time we changed our processor, so very often. We suspect it might be due to the ProcessorFactory class, but it stays unsolved. Since rebuilding didn't help, we figured out that removing the main part of the processor, running Vadere, and copying it back into the class and running it again makes it possible to run the changes.


Migration Assistant

Cannot invoke "Object.getClass()" because "dataProcessor" is null

Figure 10: frequent and time consuming error

Experimental data

Scenarios in the Experiments In the first paragraph related to the summary of the paper Tordeux et al. (2019) it was explained at length how the experiment results were obtained. To summarize very shortly: two different experiments for a corridor scenario and a bottleneck scenario with varying parameters for pedestrians(15, 30, 60, 85, 95, 110, 140 and 230) in the corridor(1.8m width) and pedestrians(150) in a bottleneck scenario with varying width(0.70, 0.95, 1.20 and 1.80m) were analyzed.

Structure of the experimental data The data obtained by camera in a frame rate of 16/s have the format of 13. The first column is the Pedestrian ID, the second column is the frame(basically the timestep) the remaining columns are the coordinates, with the z coordinate being the height of the pedestrian. Note that the data is sorted according to the PedestrianID.

```
1 0 125.174 115.38 166.284
1 1 125.144 115.317 166.284
1 2 125.438 115.181 166.284
2 20 115.134 311.497 167.794
```

Figure 11: format of the website data

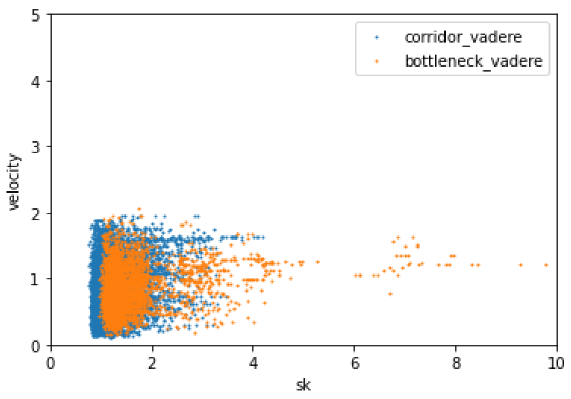
Processing of the experimental data As mentioned several times above for the neuronal networks we need values of velocity, mean spacing distance(s_k) and relative pedestrians coordinates. To use the data in a similar way we process the data to the same format as the resulting Vadere output data seen in figure 9. It is possible to obtain the desired information just from the 4 columns from the experimental data set.

For the velocity we simply take two data points and calculate the speed, through the distance of the data point coordinates and the time passed between the intervals. So we computed the velocity v of time step t, with the passed time step coordinates (x_{t-1}, y_{t-1}) and the frame rate f, which was $16/s$. Additionally we scaled the units from centimeters to meters.

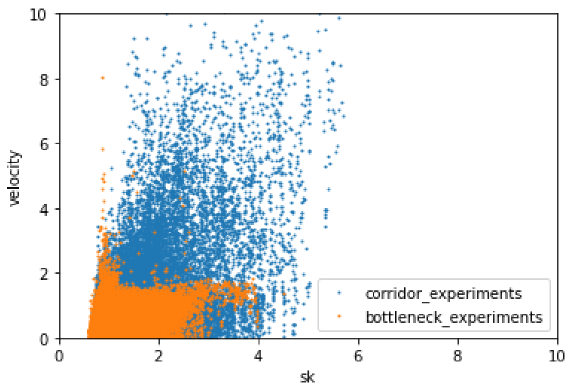
$$v_t = \sqrt{(x_t - x_{t-1})^2 + (y_t - y_{t-1})^2}.f$$

For the nearest neighbors we first grouped the data set by the time frame, then we looped over every single pedestrian in that frame, looping over all the possible neighbors from that time frame, to check weather they are contained in the k nearest neighbors. During that loop every pedestrian has a priority queue ensuring the k nearest neighbours are stored fairly efficiently. All this can be seen in the corresponding documented codes.

Resulting relation between velocity and the mean distance spacing One main goal observation from Tordeux et al. (2019) was that there is a strong correlation between the mean distance spacing(s_k) and the velocity. For the neuronal network this is exactly the relationship, that is to be learned. If there is or isn't a dependency can decide whether the neuronal network performs, -or not. The following graphs show the plots we obtained, when plotting these to variables together.



(a) vadere data, all simulation runs plotted together



(b) experimental data, from the bottleneck with 0.95cm width and the corridor with 95 pedestrians

Figure 12: Unsatisfying data

What is immediately noticeable is that the corridor case in figure 12(a) has extremely high velocity. Unless Usain Bolt wasn't part of these experiments, the data set we obtained,our assumptions, our logic or our programming is flawed. Since the Tordeux et al. (2019) achieves reasonable results, the problem most likely lies in our processing. What makes this especially hard to understand is that the generation of bottleneck and corridor scenario where produced with the same program, but only the corridor has these extreme high velocity values for the experimental results. For the figure 12(a) the velocity values are in a more sensible area, which makes sence since the values are introduced by vadere and capped to realistic values and we don't process on them.

Issues A major issue is that the observed difference between the bottleneck scenario and the corridor scenario depicted in figure 1 doesn't translate into the Vadere simulation results at all. Basically the testing results for the bottleneck and the corridor scenario in Vadere are identical considering the relation between the mean spacing distance and the velocity. Since we especially mimiced the bottleneck scenario identically we are unsure whether this property doesn't translate into Vadere at all, whether it doesn't translate in the Optimal Steps Model, if an adaption of the scenario to capture this notion not only in the experiment, but also in the simulation would have been needed or the pedestrians in the experiment had an extraordinary properties.

For the Vadere simulation we think, that a scenario differing more from the experimental setup might be possible to produce. For both figures the relationship that is learnable between mean spacing and speed that was shown from the paper in figure 2 couldn't be reproduced. This input therefore is not sufficient for the training of the neuronal network, yet we are positive that the installed pipeline has potential, when the underlying issue of bad inputs is resolved.

Another problem we briefly touched upon, during the advantages of integrating the processor for vadere instead of only using the output files is that the k Neighborhood is incomplete, when only considering the neighbors inside the measurement area.

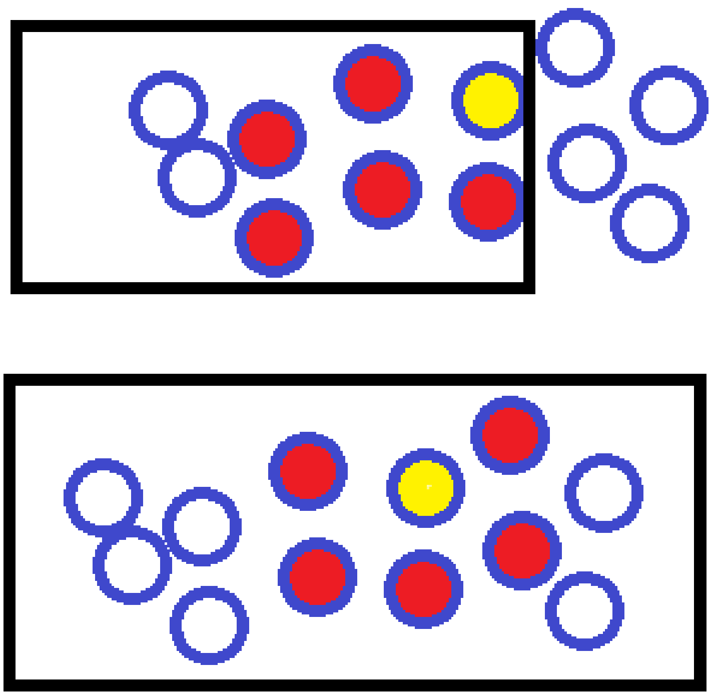


Figure 13: Explanation concerning k nearest Neighbors within measurement areas

In this example it can be seen that the k neighborhood that is set to 5 is completely different at the boundary situation than in the center area if we only have data from the measurement area, we are limited by this. The pedestrians outside of the area , that aren't in the observed neighborhood(red), influence the observed pedestrian(yellow) just as much as any other neighbor, but we simply can't see them. This major problem was omitted by using our Vadere output processor, but is a limitation on the experimental data set.

Report on task 3, Neural Network Implementation and Training. (35 points)

Neural Network Design

As shown previously, we considered two categories of neural network as follows:

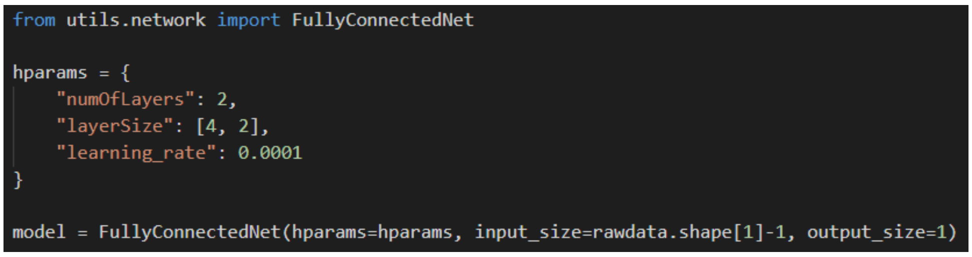
$$NN_1 = NN_1(h, (x_i - x, y_i - y, 1 \leq i \leq K)$$

$$NN_2 = NN_2(h, \bar{s_K}, (x_i - x, y_i - y, 1 \leq i \leq K)$$

.

Since we have the knowledge that Weidmann's model can fit the crowd data and is parametric model with 3 parameters, the neural network we considered here does not need to be very complex. Shown by Tordeux et al, fully connected neural network with one to two layers suffices the modelling.

We implemented a pipeline to train the fully connected network in the notebook “NN.ipynb” at the root of our repo. The fully connected network is defined as a class “FullyConnectedNet” in “utils.network” and is implemented under the framework of pytorch-lightning. By default, ReLU is used as the activation function; Adam algorithm (optimizer) is invoked to train the network. As shown in figure 14, we design the class of fully connected network to be able to initialize an network object with different number of layers and size of each layer by passing such information as hyperparameters. This design helped us to do the model selection in the future steps more efficiently.



```
from utils.network import FullyConnectedNet

hparams = {
    "numOfLayers": 2,
    "layerSize": [4, 2],
    "learning_rate": 0.0001
}

model = FullyConnectedNet(hparams=hparams, input_size=rawdata.shape[1]-1, output_size=1)
```

Figure 14: Control network architecture by passing hyperparameters.

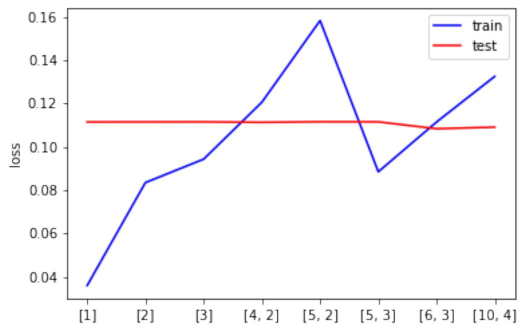
Neural Network Training Before training the neural network, we first need to load the pre-processed data into a “dataset” class that works with Dataloader in pytorch. This dataset class is the ”crowdDataset” class we wrote in the “utils/crowdDataset.py” file. After the entire dataset is successfully loaded, it would be spit in to training dataset, validation dataset and testing dataset with the proportion 60%:20%:20%, and thus corresponding pytorch Dataloaders are created.

The fully connected network class is implemented under the framework of pytorch-lightning, therefore, we also adopted the training framework of pytorch-lightning—train the network using pytorch-lightning.Trainer(). Provided the network as a model, the pytoch-lightning.Trainer() can conduct the training process automatically against the validation loss , which can be further controlled by specifying arguments like “max_epoches”. In particular, we introduced an early stopping callback to the Trainer with patience = 10 to save redundant training and avoid overfitting.

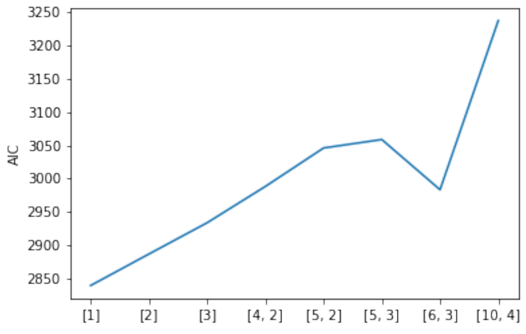
To visualize the training loss and validation loss during the training process, the tool tensorboard is used incorporate with Trainer of pytorch-lightning. In tensorboard, each training process of the model is called a run, and in our design, we would name each run according to the architecture of the network and the starting time of training. For instance, a run with name ”2_4_2_20210714-1300” saved under the folder “lightning_logs” corresponds to the training logs of a network created at 13:00, July 14th 2021, with 2 layers whose sizes are 4 and 2 respectively. This enabled us to save the information of training loss and validation loss of each model in an organized manner.

Training Results and Model Selection Same as the tentative configurations of the neural network used in the paper, we train the different networks on the merged dataset for bottleneck and corridor scenarios generated in Task 2 from Vadere. The configurations of the network we explored are: [1], [2], [3], [4,2], [5,2], [5,3], [6,3] and [10,4]. The values insides the brackets are the size of each hidden layers of the network.

Figure 15(a) shows the training loss and testing loss of NN2 model with the above configurations. The testing loss only varies very slightly between different configurations, i.e. one neuron already learned enough information for prediction. This is an unexpected result, and it shows that the dataset we created using Vadere is not reasonable enough in terms of describing crowd dynamics in real life as discussed at the end of report on task2. Furthermore, due to the stable testing error along different configurations, from 15(b) we can see that models with higher complexity are just worse in terms of AIC.



(a) NN2 model loss.

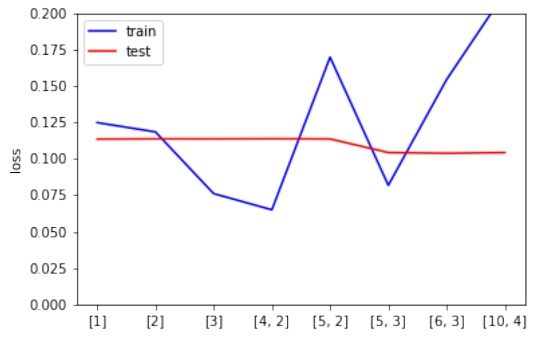


(b) NN2 model AIC.

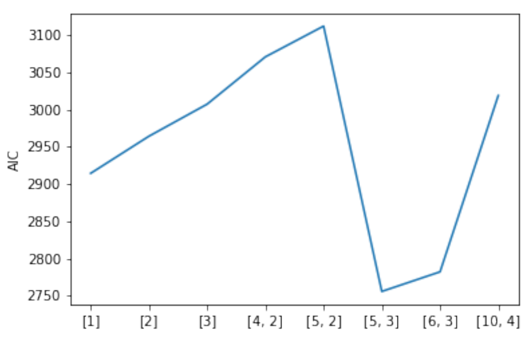
Figure 15: NN2 Performance on the merged corridor and bottleneck dataset generated in Task2 by Vadere.

The performance of NN1 on the merged corridor and bottleneck generated in task2 from Vadere is similar to the one of NN2. Still the reason is that the dataset we feed the network is violating the real-life situation. But surprisingly, AIC selects the configuration of [5,3] as the best for NN1.

The above training loss, testing loss and AIC values for the models we trained, are summarized by our implementation of the network. The summary files are stored as "NN1_model_summary/model_summary.txt" and "NN2_model_summary/model_summary.txt" respectively."



(a) NN1 model loss.



(b) NN1 model AIC.

Figure 16: NN1 Performance on the merged corridor and bottleneck dataset generated in Task2 by Vadere.

From Task2, we have discussed the issues of the the flawed dataset generated from Vadere. The dataset was not good, however, our implemented pipeline of training a neural network is fine. Figure 17 shows the curve of training loss and validation loss of NN2 with configuration [3], and figure 18 shows the curve of training loss and validation loss of NN1 with configuration [4,2]. Both of them are plotted by tensorbord during the training progress and can be considered as good training curves. Since we used an early stopping callback, all these training curves would not show overfitting. For each model we trained, we preserve the logs of training information in the folder "lightning-logs" by pytorch-lightning automatically. Load the directory "./lightning-logs" of our repo by tensorboard, one should be able to see all the training curves of the model mentioned in this section. These nice training curves are the evidence that our pipeline of training a fully connected neural network should be fine.

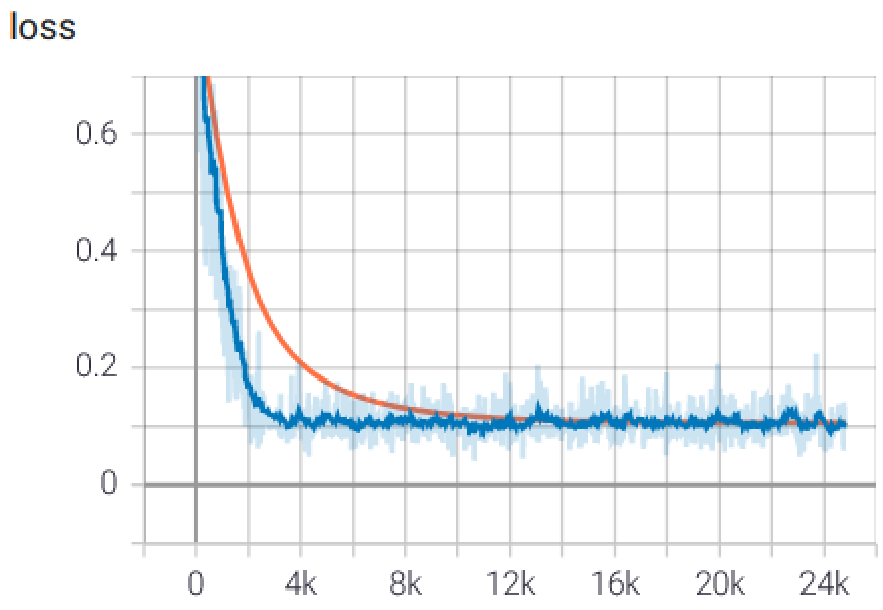


Figure 17: Training loss and validation loss for NN2 with configuration [3]

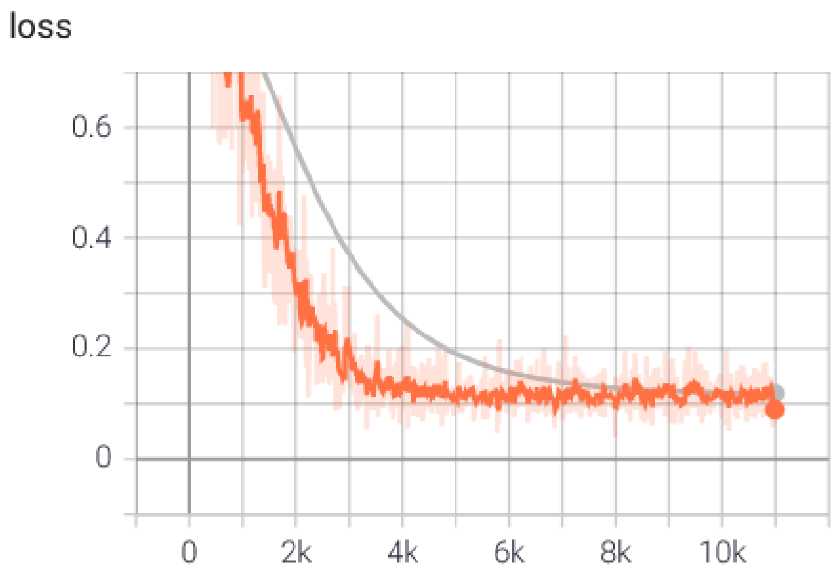
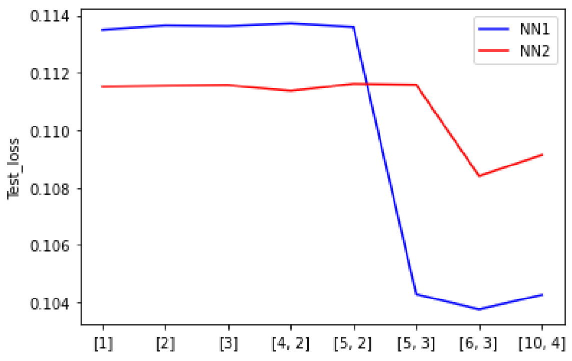
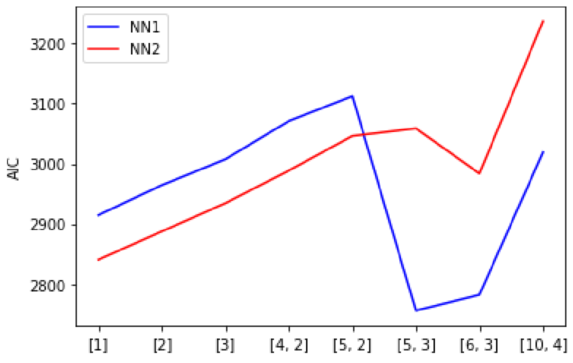


Figure 18: Training loss and validation loss for NN1 with configuration [4,2]

Based on the above analysis, for our flawed dataset generated from Vadere, the best configuration for NN1 and NN2 are [5,3] and [1]. We are aware that this result is non-sense, but base on this result we still could see that introducing the mean spacing distance s_K is useful to predict speed of pedestrians. As figure 19 shown, for model that are with less complexity, NN2 performs better than NN1, which probably due to introducing the information of mean spacing distance and mean spacing distance has dependency with velocity even in our flawed datasets generated from Vadere.



(a) Testing loss.



(b) AIC

Figure 19: NN1 and NN2 comparison.

Report on task 4, Summary and Reflection. (5 points)

In general, this project is not a successful one.

We learned from Tordeux et al.(2019) that neural networks can work as alternatives of physical-based Weidmann's model to describe crowd dynamics. We did implement a processor that can output the relative coordinates of K-nearest-neighbors into Vadere, which enabled us to better understand how this powerful crowd simulation software generates information of the crowd dynamics during its simulation. We also managed to create a pipeline of training a neural network in a modular way in python. However, our experimentations on generating reasonable crowd data from Vadere failed, which led us to a dilemma where tools for modelling are ready but lacking valid datasets,

It is a worthy experience though, for our group as neural network beginners, to develop a complete pipeline of neural network training with aids of popular python libraries and tools (pytorch, pytorch-lightning, tensorboad, etc.).

References

Tordeux, A., Chraibi, M., Seyfried, A., Schadschneider, A. (2019, March). Artificial neural networks predicting pedestrian dynamics in complex buildings. In Workshop on Stochastic Models, Statistics and their Application (pp. 363-372). Springer, Cham.