Report for exercise 4 from group A

Tasks addressed:	5
Authors:	ALEX PASQUALI
	NICOLA GUGOLE
Last compiled:	2022–03–10
Source code:	https://github.com/AlexPasqua/MLCMS-project

The work on tasks was divided in the following way:

ALEX PASQUALI	Task 1	50%
	Task 2	50%
	Task 3	50%
	Task 4	50%
	Task 5	50%
NICOLA GUGOLE	Task 1	50%
	Task 2	50%
	Task 3	50%
	Task 4	50%
	Task 5	50%

Report on task 1, Summing up the paper

Introduction The behavior of pedestrians changes depending also on the geometry of the facility they are in, therefore it is difficult to accurately predict their movements using simple models with few parameters that do not take into account such geometry. Neural networks (NNs) are a class of models with many adaptive parameters that do not have a fixed and direct physical meaning, for this reason they are very flexible and might be a suitable alternative for these forecasts. Specifically, the task is the one of estimating the speed of the pedestrians as a scalar value. Microscopic models consider people as points with a certain size and their few physical parameters can be inferred from the fundamental diagram (FD), which puts in relation the speed and the surrounding distance spacing to neighbors or obstacles. Despite their simplicity, microscopic models can describe quite well various pedestrian flows of phenomena such as lane formation [4, 2, 3], but the predictions in complex spatial structures remains problematic.
The aim of this paper is to asses whether NNs are able to accurately describe the pedestrians' behavior for various types of spatial structures. These are compared with an FD-based model (Weidmann's Fundamental Diagram) with data gathered through experiments in a corridor/ring and bottleneck scenarios.

Models The first modelling approach is the Weidmann Fundamental Diagram, which models the speed as a continuous scalar value that is function of the mean spacing (i.e. the average distance of each pedestrian to its K nearest neighbors). This is reported in Equation 1.

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

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 The data used in the paper is part of the online database of pedestrian experiments [1]. It is divided into two dataset (available at [5]) obtained in laboratory conditions that represents the following scenarios:

- Ring / Corridor (R): closed loop geometry of length 30m and width 1.8m with a pedestrians density level ranging from 0.25 to 2 ped/m^2, corresponding to a number of participants that goes from 15 to 230. The measurement area is 6m long and it is situated on a straight segment (that is why this scenario is also called "corridor").
- Bottleneck (B): this scenario can be seen as a corridor whose width suddenly reduces. The initial width is 1.8m, while the actual bottleneck assumes various width values, namely 0.70, 0.95, 1.20, 1.80m (this last value actually cancels the bottleneck effect because the width does not change). The number of pedestrians in this scenario is fixed to 150.

The speed for a given mean spacing differs in a scenario or the other, therefore the geometry is in part responsible for the behavior of pedestrians. Figure 1 shows this phenomenon.

Fitting the neural network The neural network is fitted on the normalized data through mean squared error MSE $= \frac{1}{N}\sum_i (v_i - \tilde{v}_i)^2$, where v_i is the actual speed and $\tilde{v}_i$ is the speed predicted by the NN. Half of the data is kept for testing, while the rest is used for training, which is carried out using cross-validation over 50 bootstrap subsamples. The authors do not specify the number of folds for the cross validation nor the size of the bootstrap subsamples. In the paper, different NN architectures are tested: they are always fully-connected feed-forward NNs, but the number and the size of the hidden layers assumes the values (1), (2), (3), (4,2), (5,2) (5,3), (6,3), (10,4) and (12,5)[1]. As shown in Figure 2, the training error continues to decrease as the complexity of the network increases, while the testing error shows a minimum corresponding to the architecture denoted by (3), before going into overfitting.

[1]The number of integers in the tuples indicates the number of hidden layers and the integers themselves represent the number of neurons in the i-th hidden layer.

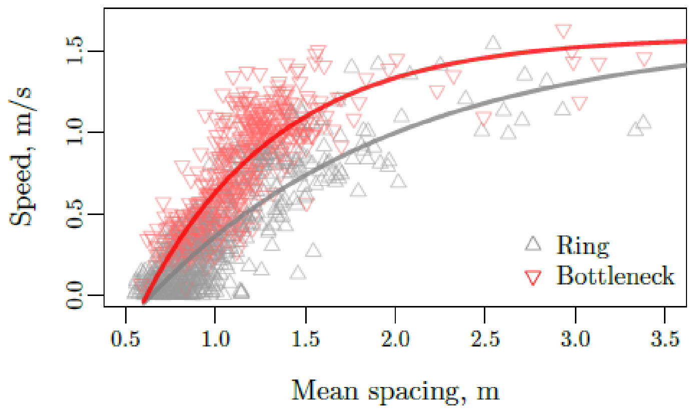


Figure 1: Weidmann FD fitted on the data, showing how the speed of pedestrians differs depending also on the geometry (corridor/bottleneck).

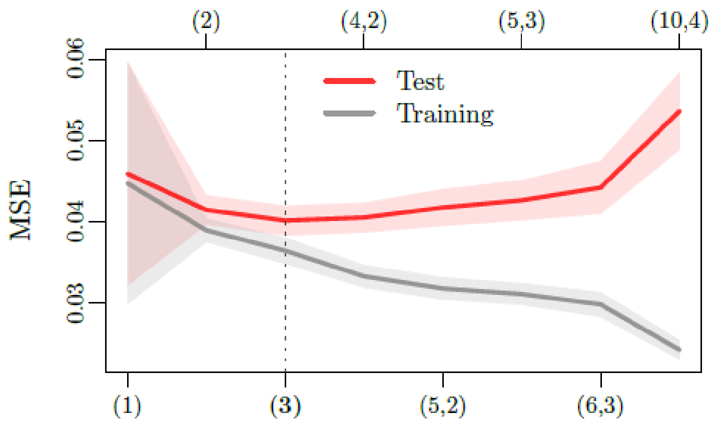


Figure 2: Training and testing MSE for different NN architectures.

Model comparison The Weidmann model and the NN that performed best (single hidden layer with 3 units) are compared on different combination of training/testing data. As it is possible to observe in Figure 3, the NN has a lower MSE for all the these combinations, but in particular the difference is more accentuated when the models are trained with a combination of the two scenarios. More precise numbers are reported in Table 1:

Training/testing scenarios	**NN's MSE improvement**
Ring (R/R)	∼ 5%
Bottleneck (B/B)	∼ 15%
Unobserved situations (R/B, B/R)	∼ 15%
Training on mixed scenarios (R+B/R, R+B/B, R+B/R+B)	∼ 20%

Table 1: MSE improvements of the NN over the FD model for different combination of scenarios in the training/testing data.

The same plot of Figure 1 is reported in Figure 4, but the FD model has been fitted on the NN's predictions. As it is possible to notice, the network is able to differenciate the type of geometry, even though the predictions are more mixed than the observations, where the two "clusters" belonging to one scenario or the other are much more distinct leading to two FD curves that are more spread apart.

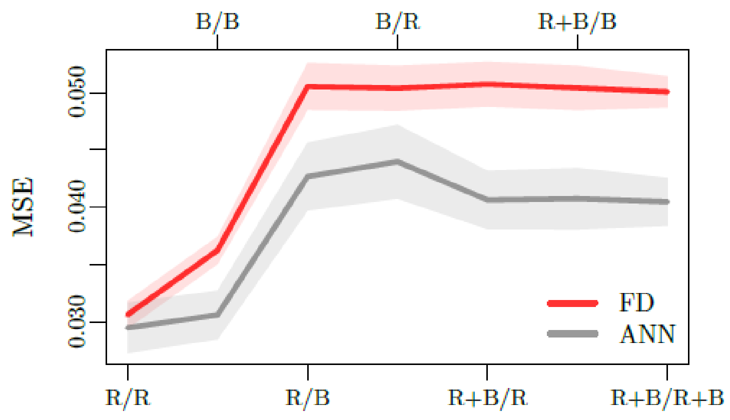


Figure 3: MSE of the FD model (in red) and the NN (in grey) with different combinations of training/testing data. R stands for "ring/corridor" while B represents the bottleneck. The letter on the left of the slash indicates the scenario(s) on which the models have been trained and the one after the slash indicates the scenario(s) used for testing.

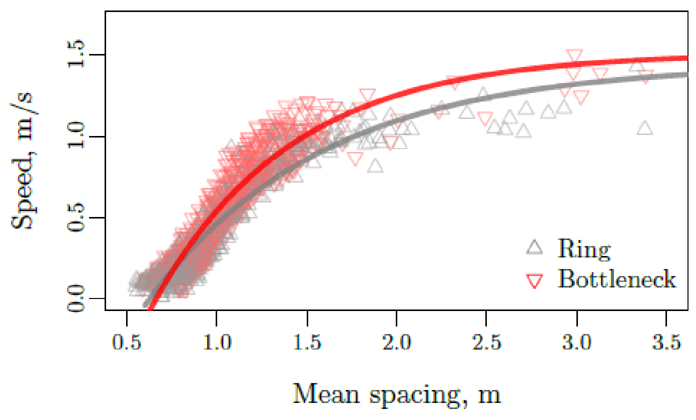


Figure 4: FD model fitted on the NN's predictions.

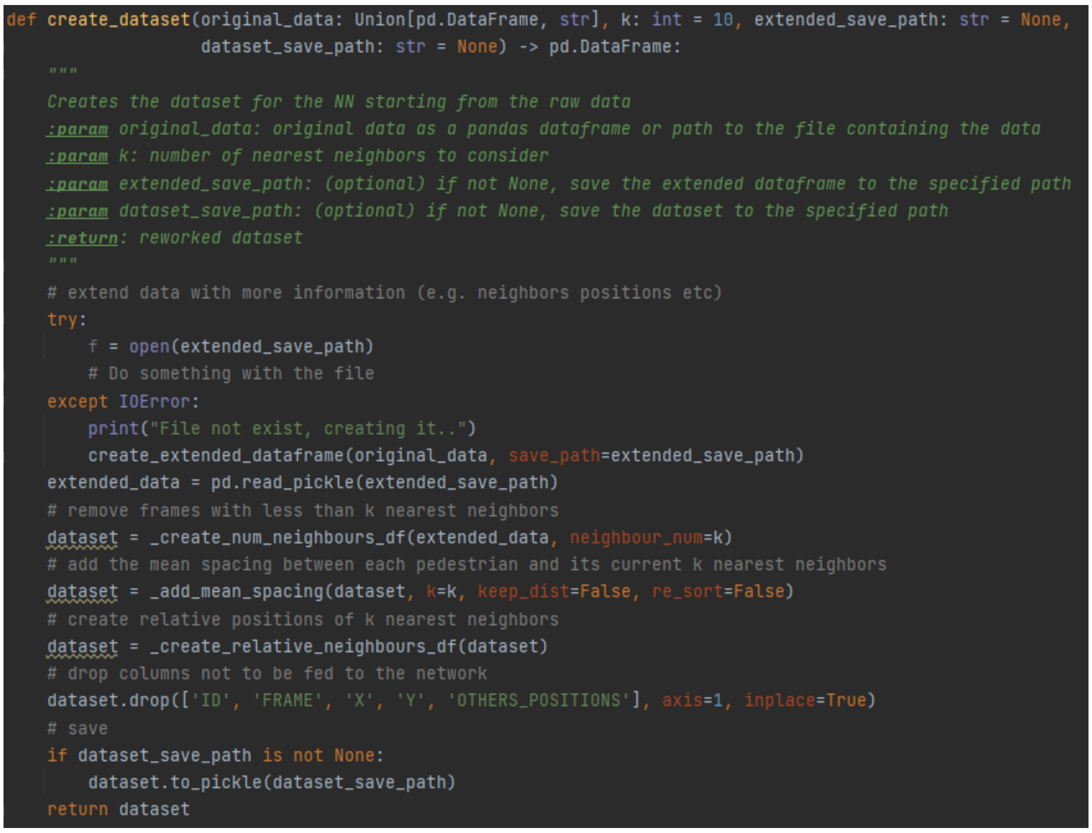
Report on task 2, Implementation of the neural network

Objective of this task is to discuss all the relevant aspects needed for the implementation of the models as well as their training. To reach a comparability between the implementation and the resulting paper we needed to construct and process the data following closely the paper's method. This proved to be non trivial because of the paper vagueness in defining their chosen methodologies.
In the following paragraphs we are going to describe how the data is preprocessed from our side, as well as how we implemented the neural network for predicting observations and finally describe how we defined the FD model through a suited neural network training.

Data Preprocessing The data provided by the paper are taken from experiments at bottlenecks and in corridors with closed boundary conditions. They are not elaborated already for the final task, needing therefore some preprocessing. We are given few fields, giving per each row the **pedestrian ID**, as well as the **time** (in frames) and the **position** of the pedestrian in X, Y, Z. From these data we are asked to create the data needed by the two models, which for the training respectively require:

- `mean_spacing` as input and `speed` as output regarding the **FD model**. 1 float input, 1 float output.
- `mean_spacing` and `knn_positions` (relative) as input and `speed` as output regarding the **speed predictor model**. 2k+1 float input, 1 float output.

To achieve this result we implemented various methods in the script `create_dataset`, which implementation is shown in **Figure 5**.



```python
def create_dataset(original_data: Union[pd.DataFrame, str], k: int = 10, extended_save_path: str = None,
                   dataset_save_path: str = None) -> pd.DataFrame:
    """
    Creates the dataset for the NN starting from the raw data
    :param original_data: original data as a pandas dataframe or path to the file containing the data
    :param k: number of nearest neighbors to consider
    :param extended_save_path: (optional) if not None, save the extended dataframe to the specified path
    :param dataset_save_path: (optional) if not None, save the dataset to the specified path
    :return: reworked dataset
    """
    # extend data with more information (e.g. neighbors positions etc)
    try:
        f = open(extended_save_path)
        # Do something with the file
    except IOError:
        print("File not exist, creating it..")
        create_extended_dataframe(original_data, save_path=extended_save_path)
    extended_data = pd.read_pickle(extended_save_path)
    # remove frames with less than k nearest neighbors
    dataset = _create_num_neighbours_df(extended_data, neighbour_num=k)
    # add the mean spacing between each pedestrian and its current k nearest neighbors
    dataset = _add_mean_spacing(dataset, k=k, keep_dist=False, re_sort=False)
    # create relative positions of k nearest neighbors
    dataset = _create_relative_neighbours_df(dataset)
    # drop columns not to be fed to the network
    dataset.drop(['ID', 'FRAME', 'X', 'Y', 'OTHERS_POSITIONS'], axis=1, inplace=True)
    # save
    if dataset_save_path is not None:
        dataset.to_pickle(dataset_save_path)
    return dataset
```

Figure 5: Most outer function to preprocess paper data.

Going through the workflow:

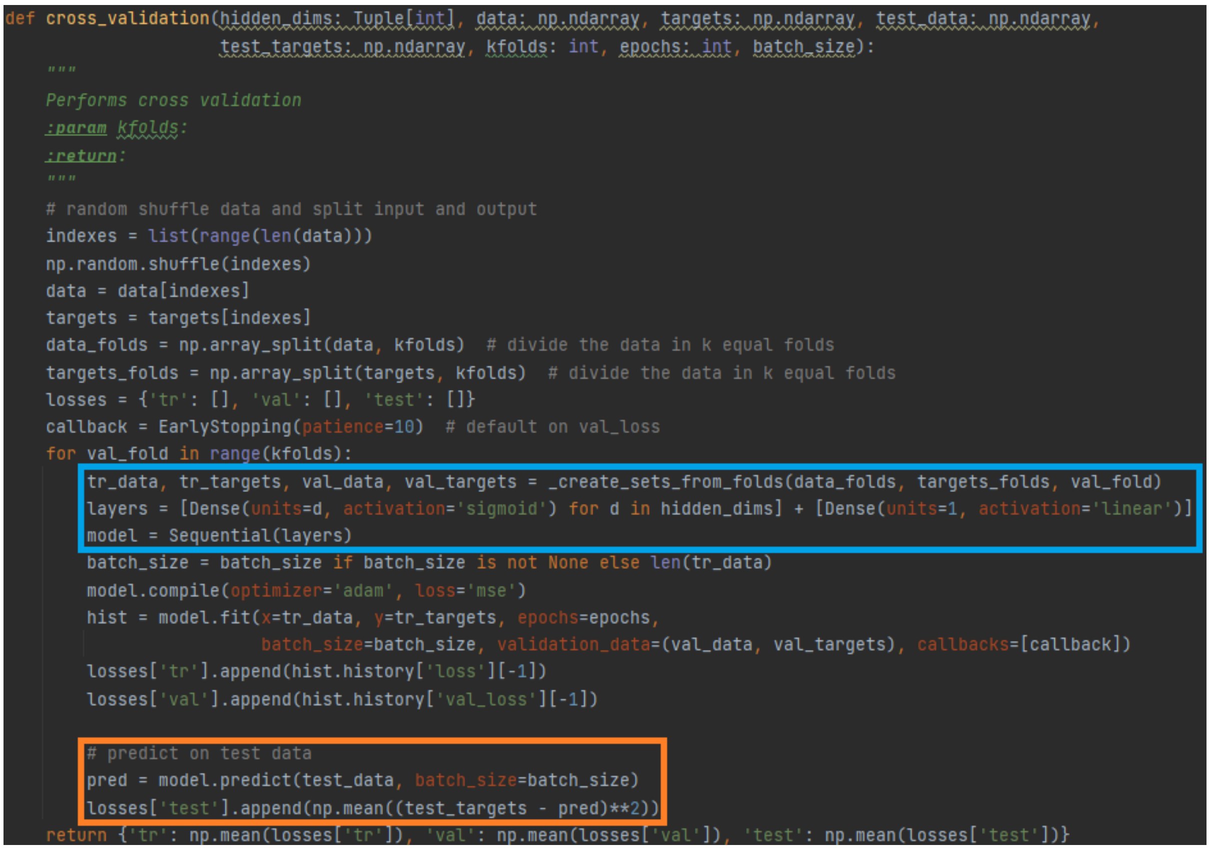
- first of all the dataset gets *extended*, if it was not already extended in the past. This means that each row (representing a certain pedestrian at a certain moment in time) is completed with ALL the neighbouring pedestrians as well as the **speed** of the pedestrian at that moment (calculated as space between two consecutive moments of that pedestrian divided by the time taken, a simple operation thanks to having frame number and frame rate). All neighbours are added to the row so that a further filtering can be done as well as to have a complete dataframe, independent from the choice of how many nearest neighbours will be considered afterwards.
- after creating the extended dataset (only done if necessary), all the rows with less than needed nearest neighbours are dropped.
- eventually the **mean_spacing** field is created by first modifying every row such that only the k nearest neighbour positions are maintained. When each row has the needed nearest neighbour data then the mean spacing can be easily calculated as mean of euclidean distance between each neighbour and the currently selected pedestrian.
- finally all nearest neighbours positions are transformed to relative nearest neighbour positions, subtracting to each neighbour the current pedestrian coordinates. At this point all data needed for training is there and all other fields (such as **frame**, *pedestrian ID*, *pedestrian coordinates*) are dropped.

To further simplify the data handling process we constructed the `read_dataset` method in `utilities.py`. The method accepts a path to access the requested data and through a flag understands if the data to return has to be adequate for training the FD model or the speed predictor NN.

Neural Network for Speed Prediction Paper describes incredibly small networks, a peculiar fact when looking at the year of publishing of the paper. The networks are also not particularly complex as previously explained, being feed-forward neural networks with small hidden layers having *sigmoid* as activation function and a one-node linear output layer.
The NN training requires *bootstrapping* over many subsamples (the paper fixes the number **50**) and actuating *cross validation* on the subsample.

To basically describe how we implemented such a process we can go over the steps:

- first of all we load the data needed for the NN training. We split the data into *training* and *testing*, following a **50/50** split as requested by the paper.
- secondly we start a cycle to train a model on every subsample requested by the bootstrapping technique. At the end of each model training the final losses regarding *training*, *validation*, and *testing* are saved to have a final mean and std operation at the end of the whole bootstrapping.
- on each subsample the cross validation technique (we opted on a **5-fold**) is performed, which implementation is shown in **Figure 6**.
- once the k-fold cross validation is terminated the overall losses are taken as the mean over the losses of the k-folds. Also the losses std is computed.
- having the results at hand allows us to select the best model, which parameters will be used to train the final model using all the training data.



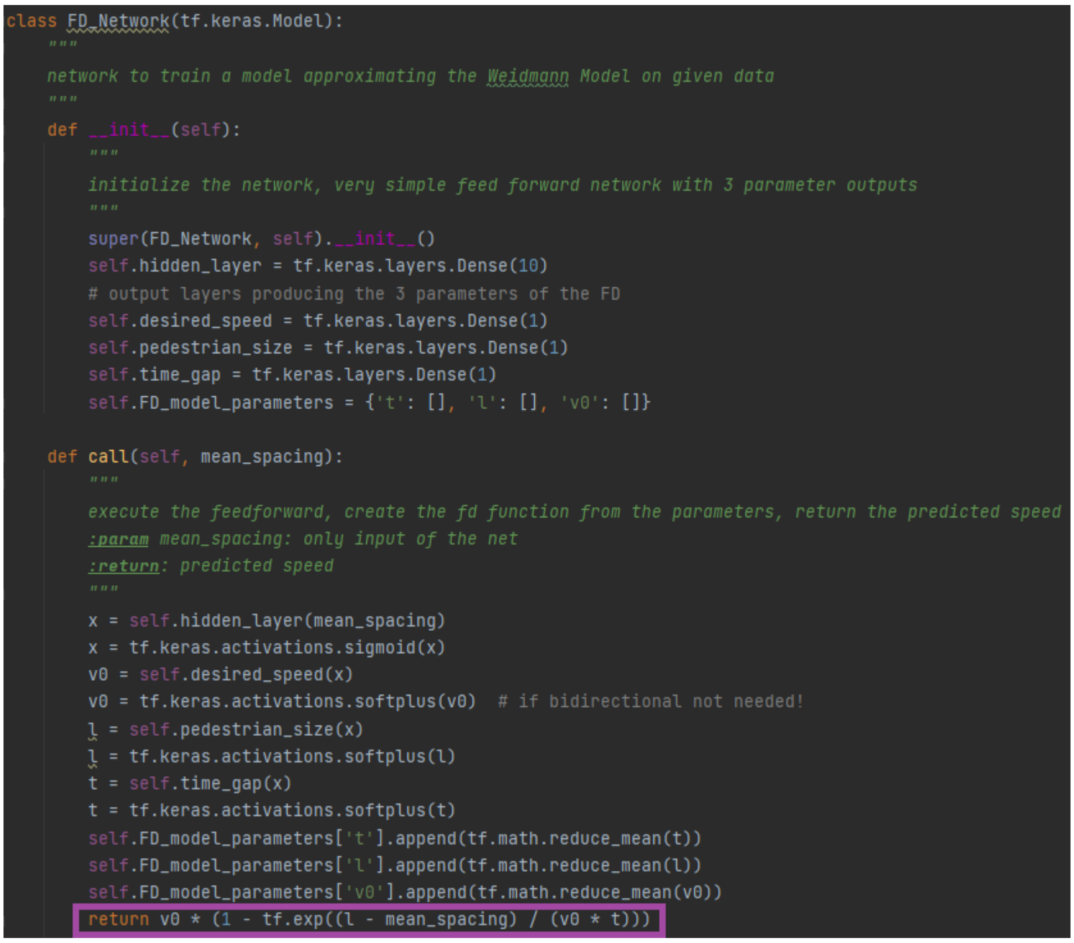
```python
def cross_validation(hidden_dims: Tuple[int], data: np.ndarray, targets: np.ndarray, test_data: np.ndarray,
                     test_targets: np.ndarray, kfolds: int, epochs: int, batch_size):
    """
    Performs cross validation
    :param kfolds:
    :return:
    """
    # random shuffle data and split input and output
    indexes = list(range(len(data)))
    np.random.shuffle(indexes)
    data = data[indexes]
    targets = targets[indexes]
    data_folds = np.array_split(data, kfolds)  # divide the data in k equal folds
    targets_folds = np.array_split(targets, kfolds)  # divide the data in k equal folds
    losses = {'tr': [], 'val': [], 'test': []}
    callback = EarlyStopping(patience=10)  # default on val_loss
    for val_fold in range(kfolds):
        tr_data, tr_targets, val_data, val_targets = _create_sets_from_folds(data_folds, targets_folds, val_fold)
        layers = [Dense(units=d, activation='sigmoid') for d in hidden_dims] + [Dense(units=1, activation='linear')]
        model = Sequential(layers)
        batch_size = batch_size if batch_size is not None else len(tr_data)
        model.compile(optimizer='adam', loss='mse')
        hist = model.fit(x=tr_data, y=tr_targets, epochs=epochs,
                         batch_size=batch_size, validation_data=(val_data, val_targets), callbacks=[callback])
        losses['tr'].append(hist.history['loss'][-1])
        losses['val'].append(hist.history['val_loss'][-1])

        # predict on test data
        pred = model.predict(test_data, batch_size=batch_size)
        losses['test'].append(np.mean((test_targets - pred)**2))
    return {'tr': np.mean(losses['tr']), 'val': np.mean(losses['val']), 'test': np.mean(losses['test'])}
```

Figure 6: Cross validation per each bootstrapped subsample.

Going more in depth at the core of the training we can discuss what is presented in **Figure 6**. Cross validation is performed on an initially random shuffle of the data to avoid possible correlation in consequent data of the original dataset. **Early stopping** is used as a simple yet efficient manner of getting the model to end the training as soon as it starts to overfit. The simplicity of the required models allowed us to automatically define them through a parameterized `Sequential` instantiation, as can be appreciated in the *light blue* box. Finally we train the models using **MSE** as metric and **Adam** as optimizer, after testing that **SGD** resulted in a too slow training. In the end (*orange* box), after fitting the model we test it on the *test data* utilizing the `predict` method, to assert the model generalization capability and save the last needed loss.

Regression on FD model **Section 1** first introduced the model required, the *Weidmann FD model*, expressed by (1). The model is rather simple, composed of four parameters. We opted therefore to create a simple NN as it can allow us to model a network with all the needed output parameters for *Weidmann.*
Since `mean_spacing` is given in input we only need a simple NN with a single node input and a three nodes layer output. The overall implementation can be appreciated in **Figure 7**. For this kind of implementation we preferred *model sub-classing* with respect to *Sequential*, since it allows for higher modelling freedom even though it is less concise code-wise.
Going into details, we decided for an architecture with a single input node and a single dense hidden layer having *sigmoid* as activation function (for non-linearity purpose) with three final distinct output nodes, one per each *Weidmann* parameter. The output activation functions should respect the parameter domain and that is why we plugged in *softplus* in all three output nodes, although one should notice how the *softplus* can become a *linear* for the *desired speed* parameter if a more complex and bidirectional scenario is taken into consideration. Eventually, looking at it from the outside, after proper training this model takes in input the `mean_spacing` and returns exactly what presented in (1), because of the formula instatiation happening in the *purple* box of **Figure 7**.



```python
class FD_Network(tf.keras.Model):
    """
    network to train a model approximating the Weidmann Model on given data
    """
    def __init__(self):
        """
        initialize the network, very simple feed forward network with 3 parameter outputs
        """
        super(FD_Network, self).__init__()
        self.hidden_layer = tf.keras.layers.Dense(10)
        # output layers producing the 3 parameters of the FD
        self.desired_speed = tf.keras.layers.Dense(1)
        self.pedestrian_size = tf.keras.layers.Dense(1)
        self.time_gap = tf.keras.layers.Dense(1)
        self.FD_model_parameters = {'t': [], 'l': [], 'v0': []}

    def call(self, mean_spacing):
        """
        execute the feedforward, create the fd function from the parameters, return the predicted speed
        :param mean_spacing: only input of the net
        :return: predicted speed
        """
        x = self.hidden_layer(mean_spacing)
        x = tf.keras.activations.sigmoid(x)
        v0 = self.desired_speed(x)
        v0 = tf.keras.activations.softplus(v0)  # if bidirectional not needed!
        l = self.pedestrian_size(x)
        l = tf.keras.activations.softplus(l)
        t = self.time_gap(x)
        t = tf.keras.activations.softplus(t)
        self.FD_model_parameters['t'].append(tf.math.reduce_mean(t))
        self.FD_model_parameters['l'].append(tf.math.reduce_mean(l))
        self.FD_model_parameters['v0'].append(tf.math.reduce_mean(v0))
        return v0 * (1 - tf.exp((l - mean_spacing) / (v0 * t)))
```

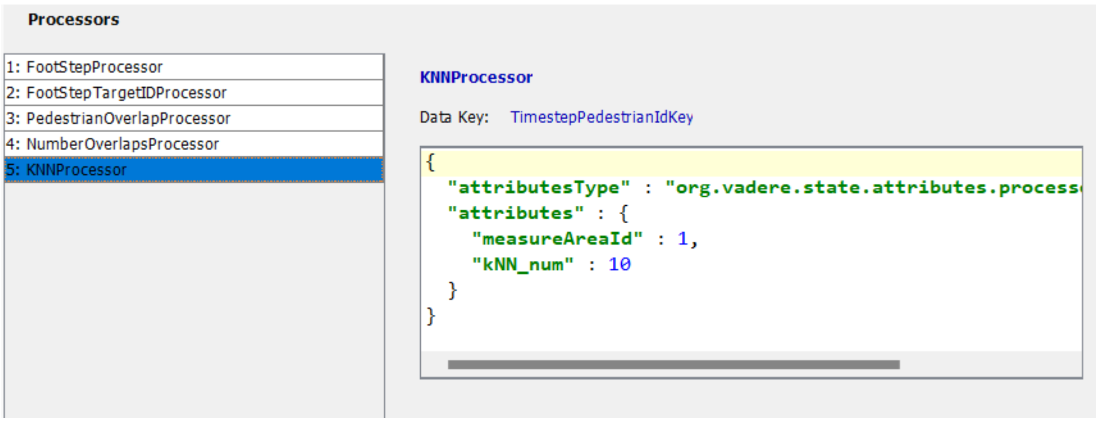
Figure 7: Weidmann model implemented as Neural Network.

As a final remark, we usually trained the FD network using once again **MSE** as metric and loss (between the *computed* and the *target speed*) with **Adam** as optimizer. Usually a small number of epochs (less than 50) was enough to reach convergence.

Report on task 3, Tests on simple examples

A first test of validity for the implementation we constructed and described in **task 2** is done through the software **Vadere**. Constructing a self-made scenario in a closed system as this software allows for a complete control over the data generation.
Constructing a scenario as we need it to be is not trivial, requiring an ad hoc *PostProcessor* implementation as well as some post processing of the output files themselves before having the data that can be fed to the constructed models.

Vadere Post Processor Implementation Implementing a new component in a large software such as Vadere proved to be rather complex at first, understanding which modules were needed to correctly append the new *PostProcessor* into the ecosystem.


Processors
1: FootStepProcessor
2: FootStepTargetIDProcessor
3: PedestrianOverlapProcessor
4: NumberOverlapsProcessor
5: KNNProcessor
KNNProcessor

Data Key: TimestepPedestrianIdKey

```
{
  "attributesType" : "org.vadere.state.attributes.process
  "attributes" : {
    "measureAreaId" : 1,
    "kNN_num" : 10
  }
}
```

Figure 8: Adding the processor in Vadere scenario.

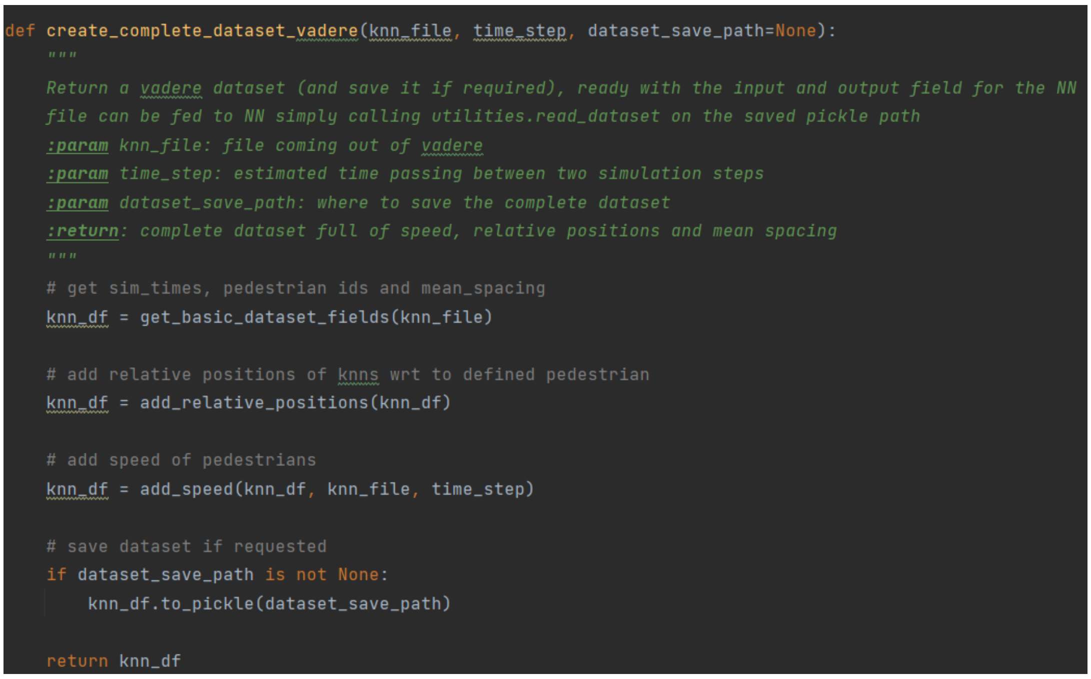
Trial and errors (and many *Migration errors*) led to the final implementation which is basically composed of three main classes:

- `KNNAttributes`: this class is extremely basic but fundamental to the whole process. It is in fact where the main needed attributes are stored, in particular the *kNN_num* of neighbours required per each pedestrian at every possible simulation step in the output file. Another important attribute here specified is *measureAreaId*, meaning the specific measurement area (there needs to be one) from where to fetch the pedestrians positions at every simulation step.
- `KNNPedestrianData`: a class useful to define how the single output line is composed. In particular this class contains all the needed output information, from *meanSpacing* (needed both as input for the FD model as well as the NN model) to *pedestrianPosition* (needed for postprocessing the speed) and finally to *kNN* (a list containing all the *kNN_num* neighbours *relative* positions, input of the NN model). In particular the function `toStrings` returns to the output file manager how to correctly order the information in the output file associated to `KNNProcessor`.
- `KNNProcessor`: the main class, updating through standard method `doUpdate` at every simulation step. When `doUpdate` is called a series of steps are performed to create the data necessary to create a new instance of `KNNPedestrianData` (new line of output):
  - the measurement area is checked to get all the pedestrians inside it at the given simulation step.
  - for each pedestrian in the measurement area a priority queue is created (to take advantage of its property of ordering).
  - for each neighbour of a pedestrian the distance between the two is calculated as well as the relative distance.
  - if the priority queue (with *kNN_num* shape) is not full or the distance of the currently selected neighbour is less than the longest distance in said priority queue, then the neighbour is added.

- each pedestrian is checked iteratively. When for a single pedestrian all neighbours have been checked then an output line is created (*timeStep*, *pID*, *meanSpacing*, *pedestrianPosition*, *kNN*), if there are enough neighbours.

Eventually, with the implementation we propose, getting the needed data is not complex. One only has to create a scenario containing a measurement area and then navigate to `Data output` in Vadere, adding a new output file with `Data Key` equal to `TimestepPedestrianIdKey`. Then one has to add the `KNNProcessor` to the processor list and fill the needed attributes (**Figure 8**), which are the number of neighbours and the measurement area id. Eventually link the output file with the processor.
The procedure is nevertheless not complete at this point, as there is still the need for postprocessing such a "`.txt`" file so that the input to NN model or FD model are created.

Post Processing of Output Files All the post processing can be done easily thanks to the utilities present in the file `data_processing_vadere.py`. The main steps are shown in **Figure 9**. First a dataframe is constructed adding the basic fields of `TIME_STEP`, `ID` and `MEAN_SPACING`. The second step is taking the relative positions and porting them from the non-convenient format of "`.txt`" to an easy to work numpy array. Eventually the only field remaining left away for execution is the `SPEED`, needed for computing the loss. Having at each row the pedestrian position and the simulation step makes the speed generation fairly easy, since what one needs is to find two following simulation steps with the same pedestrian, subtract the positions and divide by the time passed.



```python
def create_complete_dataset_vadere(knn_file, time_step, dataset_save_path=None):
    """
    Return a vadere dataset (and save it if required), ready with the input and output field for the NN
    file can be fed to NN simply calling utilities.read_dataset on the saved pickle path
    :param knn_file: file coming out of vadere
    :param time_step: estimated time passing between two simulation steps
    :param dataset_save_path: where to save the complete dataset
    :return: complete dataset full of speed, relative positions and mean spacing
    """
    # get sim_times, pedestrian ids and mean_spacing
    knn_df = get_basic_dataset_fields(knn_file)

    # add relative positions of knns wrt to defined pedestrian
    knn_df = add_relative_positions(knn_df)

    # add speed of pedestrians
    knn_df = add_speed(knn_df, knn_file, time_step)

    # save dataset if requested
    if dataset_save_path is not None:
        knn_df.to_pickle(dataset_save_path)

    return knn_df
```

Figure 9: External function to compute all needed data for training.

The aforementioned speed generation technique seems trivial, but finding the time passed between each simulation step is not. In fact one can tell to Vadere the exact time that should pass between following steps, but the execution time as well as the algorithm for calculating trajectories (we used **OSM**) tend to not allow for such a fixed time. To get a decent approximation of said timing we created a method which goes through the "`.traj`" file of the scenario and collects all the different times passing between simulation steps. Eventually the approximated `time_step` is returned as the minumum of said list. We opted for the minimum instead of the mean or the max to avoid outlier effects, as well as to get the lowest possible value, which should resemble closely what the given fixed simulation parameter was.
At the end of the procedure all the unnecessary fields are dropped, leaving only the strict necessary for training and saving the result.

Scenarios in Details This paragraph has the goal of presenting the developed scenarios before showing the results. As experiment we decided to create two different scenarios:

- a **corridor** scenario (**Figure 10**), defining a corridor of 8x2 meters. The corridor describes a situation where the pedestrians (100 people) move unidirectionally (utilizing **OSM**) from source to target, getting measure in the shown area.

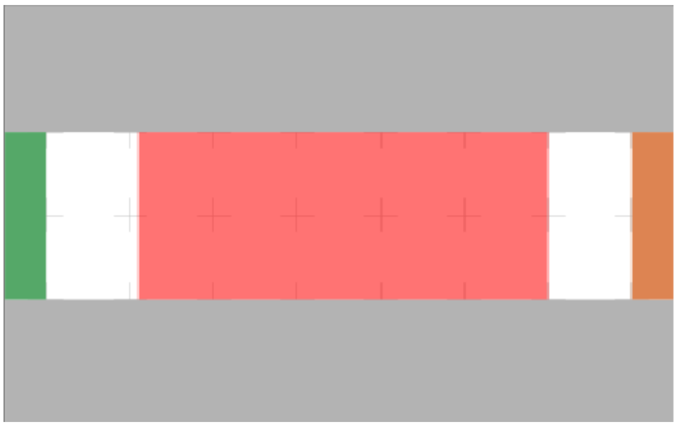


Figure 10: Vadere corridor scenario.

- a **bottleneck** scenario (**Figure 11**), defining a scenario 14 meters long with a 0.9 meters wide bottleneck. The bottleneck describes a situation where the pedestrians (again 100 people) move unidirectionally (utilizing **OSM**) from source to target, slowing down because of passing through the bottleneck.

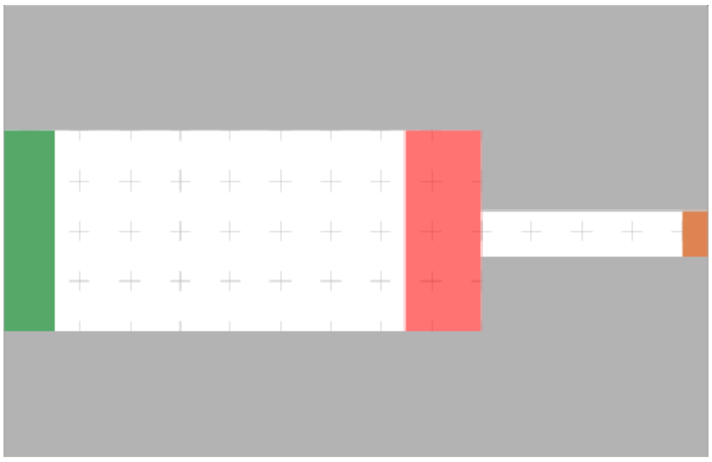


Figure 11: Vadere bottleneck scenario.

Results To validate our implementations we tested for both scenarios two different situations:

- FD model trained on observations (*in blue*).
- FD model trained on the NN's predictions (*in red*), testing both the validity of FD model as well as NN's capability of representing original observations.

We tried different NN architectures, while keeping instead for the FD model the same architecture defined in **task 2**. Moreover, since this task goes to prove the validity of the model we have not gone through validation (an important part of next task), testing the model on the training data itself. We know this is not good practice as machine learning scientists, nevertheless we once again consider this only as a validity task.
Starting with results coming from the **corridor** scenario, all the models we constructed utilize **Adam** as optimizer, with different architectures and training:

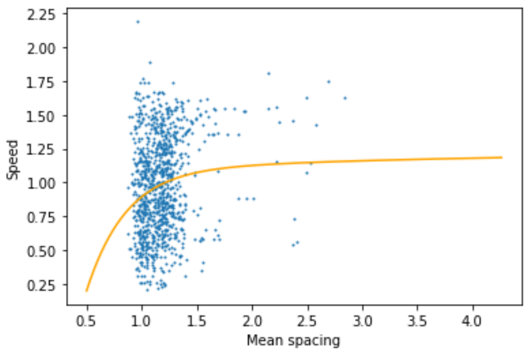
- **FD model:** architecture as described in **task 2**.
- **NN model:** after different trials changing hidden layers shape and activation functions, the best performing resulted to be a model with hidden architecture **(3,3,3)** and hidden activation function **tanh**.

Results can be appreciated in **Figure 12**, where one can appreciate how the NN is capable of resembling the observations (red dots on right vs blue dots on left) and the FD model also keeps a similar behaviour in both cases.

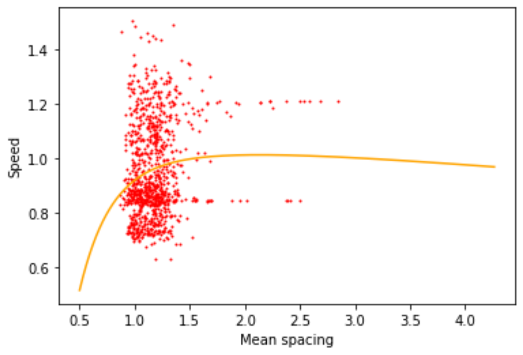
Regarding the **MSE** values:

- FD model on observations (training MSE): **0.1264**
- NN model predictions (training MSE): **0.0973**
- FD model on NN predictions (training MSE): **0.0296**

The much smaller MSE value of FD model on predictions with respect to the FD model on observations is possibly explainable by the NN's predictions, which result to be in a relatively smaller *speed* range when compared to the observations.



[a]



[b]

Figure 12: **Corridor:** FD model trained on observations (a) and FD model trained on NN predictions (b)

Proceeding with the results we can analyze the **bottleneck** scenario, where again all the models we constructed utilize **Adam** as optimizer, with different architectures and training:

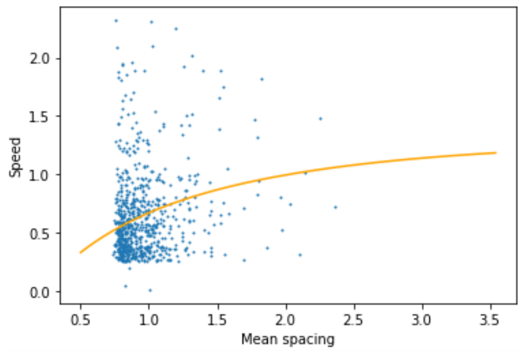
- **FD model:** architecture as described in **task 2**.
- **NN model:** the best performing resulted to be a smaller model with hidden architecture **(3,3)** and hidden activation function **tanh**.

Results can be appreciated in **Figure 13**, where once again it is visible how the NN is capable of resembling the observations (red dots on right vs blue dots on left) while the FD model seems to behave a little differently. This is possibly caused by the different dot density of observations and predictions, with the predictions being denser between (0.5, 0.75) along the speed axis, making the model behave accordingly.

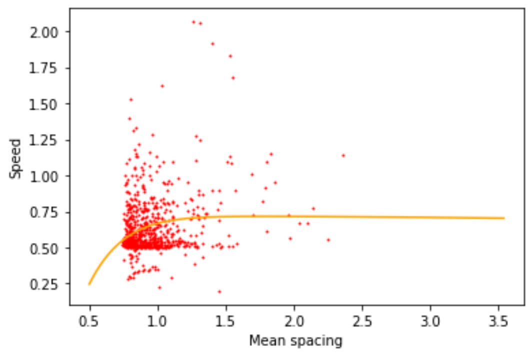
Regarding the **MSE** values:

- FD model on observations (training MSE): **0.1353**
- NN model predictions (training MSE): **0.0935**
- FD model on NN predictions (training MSE): **0.0420**

Before closing up the task we went for a more semantically interpretable result. In particular we created two FD models regarding the previously discussed *corridor* scenario. The first model is fitted on the observation data, giving physical meaning to the dynamical system with the *desired speed*, *pedestrian size* and *time gap* parameters. The second model is fitted instead on the prediction data (created thanks to the fitted NN), giving analogous meaning to the generated samples.



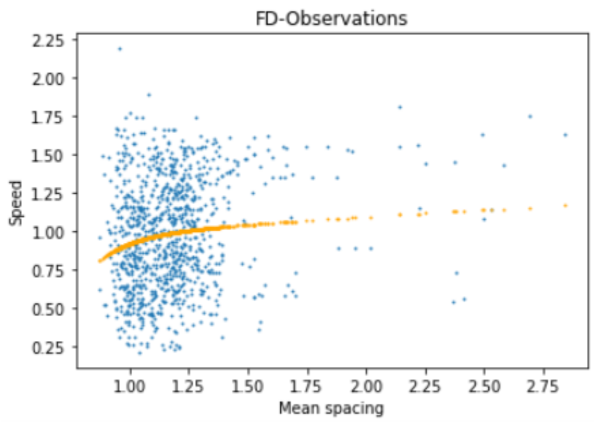
[a]



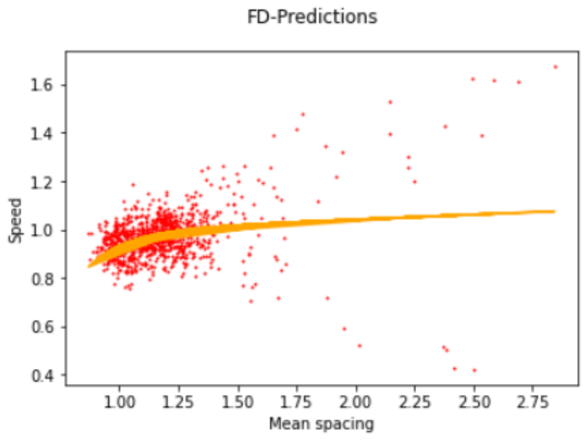
[b]

Figure 13: **Bottleneck:** FD model trained on observations (a) and FD model trained on NN predictions (b)

The result is successful if the parameters of the fitting on observations is comparable to the fitting on predictions. In **Figure 14** one can appreciate the FD fitted on the observations (which are in *blue*) vs the FD fitted on the predictions (which are in *red*).



[a]



[b]

Figure 14: Observations and FD fitted (a) Predictions and FD fitted (b)

What is immediately noticeable is that the predictions have a different *shape* with respect to the observations, leading to thinking the network has failed to approximate the data. The result is in fact fundamentally different from the already shown **Figure 12**. But since the MSE result was good we continued the experiment nevertheless, trying to understand which are the parametric values of the two generated FD models.

Since the parameters should be non other than the values coming out of the three nodes in the FD network output layer (by construction), we decided to save the batch average of these output values in a convenience structure (simple dictionary) while training. This means that for each batch of training we save the average value of each of the output nodes in a dictionary, creating therefore three indexed lists with the same length of the number of training batches.

The line of thought is that we expect the parameters to more or less stabilize toward a value the more the training goes on. This effect fortunately happened and is visible in **Figure 15**, showing the stabilization as the training advances.

Therefore we took empirically the last thousand iterations as the stabilized values for the parameters and we defined as final parameters of the FD model the average of these iterations. The results are shown in **Table 2**. It is at this point clear that the network has indeed generated predictions which give the same physical meaning as the observations, as the two FD models have extremely close parametric values.

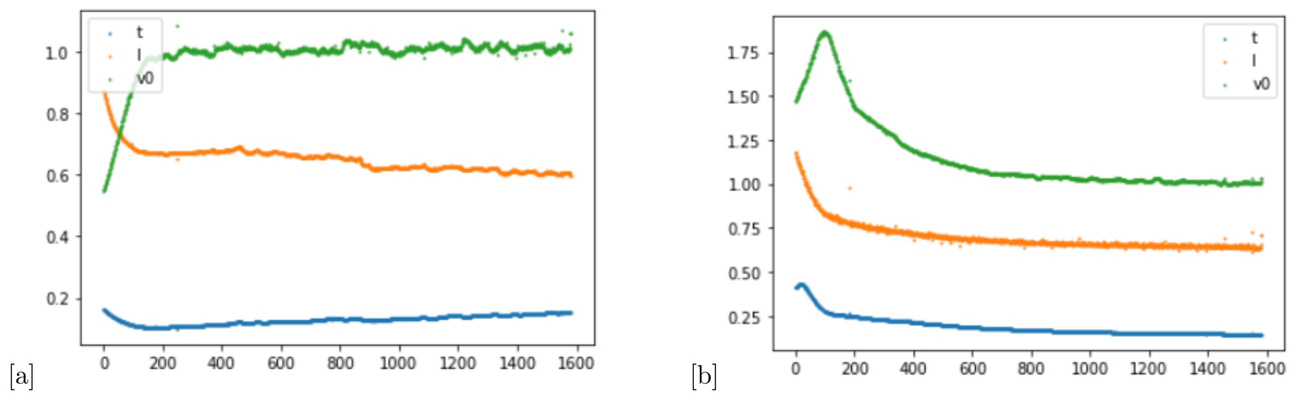


Figure 15: Parameters values batch by batch, showing stabilization

Parameter	FD on obs	FD on pred
desired speed	1.013	1.015
pedestrian size	0.616	0.649
time gap	0.142	0.15

Table 2: Parameter comparison between FD fitted on observations vs on predictions

Report on task 4, Comparison to the paper's results

Objective of this task is to try and reproduce the paper's results, which are fundamentally achieved sequentially:

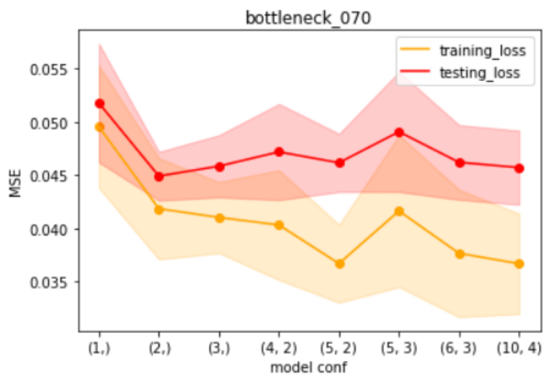
- Firstly, the researchers take into consideration a scenario (which one? Do not ask, there are things which humans are not worthy of knowing yet) and train models with incrementally big architectures on it. They utilize *sigmoid* as activation function and show how the incredibly small architecture with `(3,)` is the best performing (**Figure 2**).
- Secondly, after fixing the best architecture, they take a scenario for *corridor* and a scenario for *bottleneck* into consideration, training and testing multiple models on different combinations of training/testing scenario. **Figure 3** shows how the NN gets increasingly better performances with respect to the FD model, especially when being trained on mixed scenarios, showing how the model is capable of distinguishing better the situations.
- Last result to reproduce is more on the semantics of the NN results. The researchers fit an FD on the NN predictions to show that the *meaning* of the NN fitting goes to recreate data with approximately the same modelled *desired speed*, *pedestrian size* and *time gap*. Paper result comparison are shown in **Figure 16**

Experiment	R	B
ℓ (m)	0.64	0.61
T (s)	0.86	0.48
V_0 (m/s)	1.60	1.58

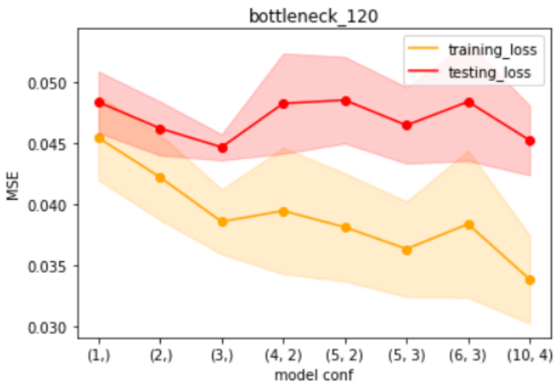
Experiment	R	B
ℓ (m)	0.63	0.66
T (s)	0.68	0.50
V_0 (m/s)	1.44	1.51

Figure 16: FD parameters of model fitted on observations (*left*) vs fitted on predictions (*right*).

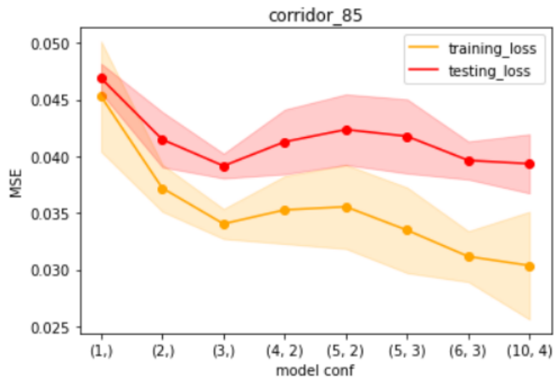
Results on different scenarios We trained the network on various scenarios: bottlenecks of 70 and 120 centimeters and corridors with 85 and 140 pedestrians. We used a bootstrap with 5 subsamples of 5000 elements performing a 5-fold cross validation (time constraints and available computation power reduced the number of bootstraps we could unfortunately try). The learning curves are reported in Figure 17. As we can see, the results in terms of MSE are comparable to the ones in the paper, sometimes even better. Furthermore, the learning curves tend to be pretty different depending on the scenario, this probably means that the plot of the paper refers to a specific scenario, and so obtaining the exact same curve is not that important. The network with one hidden layer with three units tends to provide good results, but it is curious how in the case of the bottleneck of 70 centimeters, the best model has one hidden layer, but two units. Moreover, in the case of the corridor of 140 centimeters, the network denoted by (10, 4) has comparable performance to (3,), but the latter is simpler and thus preferable. Nevertheless, from (5, 2) on the testing loss is starting to decrease significantly, suggesting that bigger models might actually surpass (3,) on that scenario.



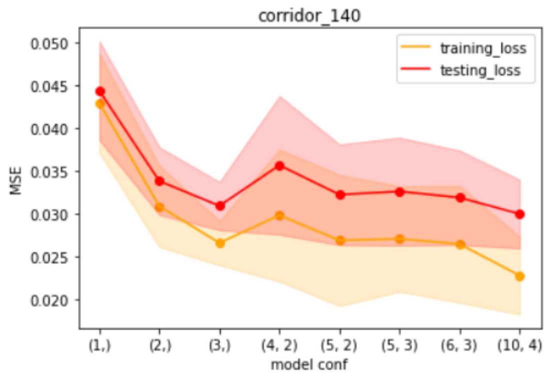
[a]



[b]



[c]



[d]

Figure 17: Learning curves of different networks achitectures on different scenarios.

Results on different combos of Corridor and Bottleneck To obtain this results we fixed the architecture to the one utilized in the paper: `(3,)`. We therefore went through the results we got in the previous paragraph, and eventually selected the two scenarios *Bottleneck_120* and *Corridor_85* to create the needed combinations, as they performed best with that architecture. Each result present in **Figure 18** was created by training the FD model using bootstrapping and cross validation (5 subsamples of 5000 samples each, 5 fold cv, early stopping) and training the NN to create the predictions, training in the same way. Testing of both models was done on the selected testing set (e.g. for B/C we trained models on the bottleneck scenario and tested them on the corridor) and the resulting MSEs created what can be seen in the figure.

It is clear that the results are differing from **Figure 3**. In our results the NN model performs better than the FD model in all cases but the most peculiar ones, having the testing data fundamentally different from the training one. Although the NN model performs better in almost all cases, it does not *greatly* outperform the FD model and also the MSEs look much higher for both of the models, when compared to the paper's results. To be honest this was not the result we expected and even trying different combinations of scenarios (e.g. we tried to produce the figure with all the combinations between *Bottleneck_120*, *Bottleneck_180*, *Corridor_85* and *Corridor_140*) did not give results that were fundamentally different from the ones proposed. After various testings, time constraints stopped us from further investigating the motivations for these differences. What we suspect is again a difference in method between us and the researchers. As already seen in the last paragraph, as it highlighted some differences in results, the same could possibly have happened here.

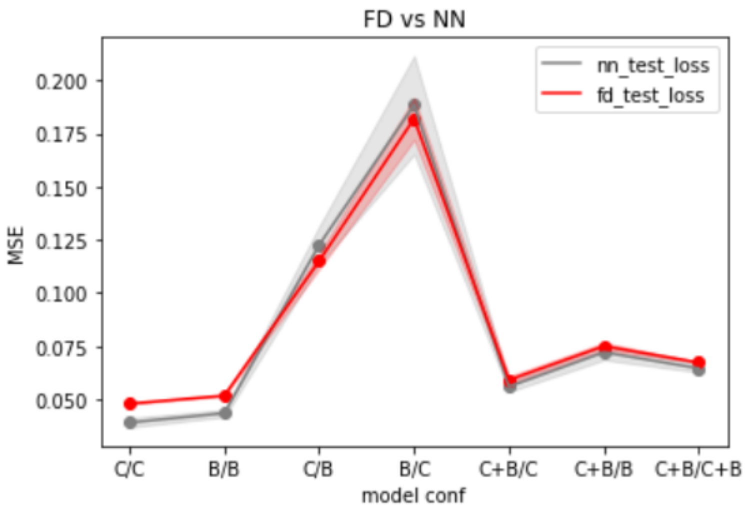


Figure 18: Comparison of test losses on models constructed with different train/test combos.

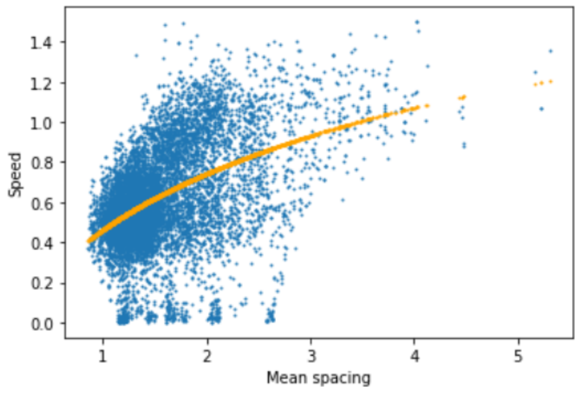
NN meaning through FD parameters We continued our comparison towards a more interpretable result, trying to replicate the third result of the paper (**Figure 16**). To do so we first set a couple of scenarios, in this case *Corridor_85* and *Bottleneck_070*. The main objective here is to construct two different FD Weidmann models per each scenario through the training already proposed in **Task 2** and compare the results between the two (again per each scenario), in a similar fashion to what we already achieved at the end of **Task 3**. The main difference is that this time we are working with not self-constructed data.

During the implementation we faced an obstacle we had already faced for the same goal in **Task 3**, regarding the gathering of output values from the three output nodes. Since we train the net by an MSE loss constructed between the output ground truth speed and the Weidmann formula created by the three output nodes, we do not actually get in output the nodes values, but only the predicted speed. So, to store the nodes outputs we need to update the aforementioned dictionary (explained in **Task 3**) inside the model `call` method, leading to the need of an eagerly run. Abilitating the **eagerly run** considerably slows down the execution, a problem which was not that relevant in the Vadere datasets since they are relatively little (less than 2000 samples), but absolutely time consuming in the bigger datasets used by the paper. All of this to say that we had eventually to take a random subsample of the datasets (we decided to make the subsample with dimension 10000) to fit the delivery time constraints.

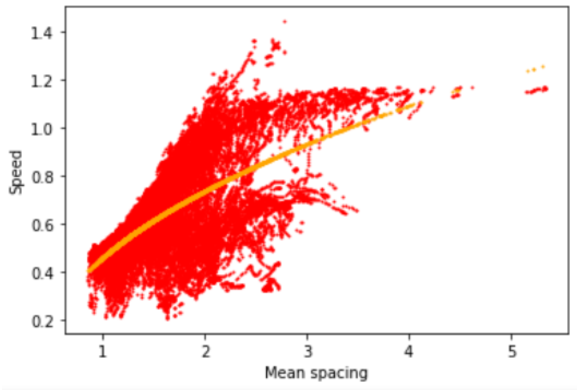
Eventually we built the two models for each one of the considered scenarios, as can be seen in **Figure 19**, where first row compares corridor scenario (FD model on observables (left) and predictions (right)) and second row goes instead for the bottleneck scenario.

As we mentioned already at the end of **Task 3**, from these training we got the parameters value batch by batch, as average of the output nodes per single batch. Plotting these values should show some reaching of stability, and that actually happens as shown in **Figure 20**.

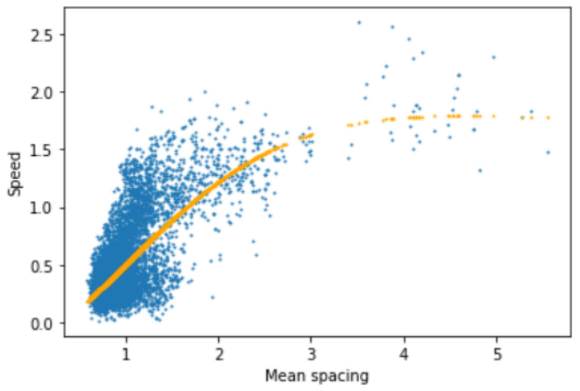
Again from these plots we empirically chose to cut the end of these zigzags, getting the final ensemble of parameter values to formulate and identify the definitive parameter values as averages of these ensembles. The comparison between the FD models on observables and predicted are shown in **Table 3**, highlighting how the physical meaning of the generated data is catching the real meaning of the observables even though the predictions differ in shape with said observables.



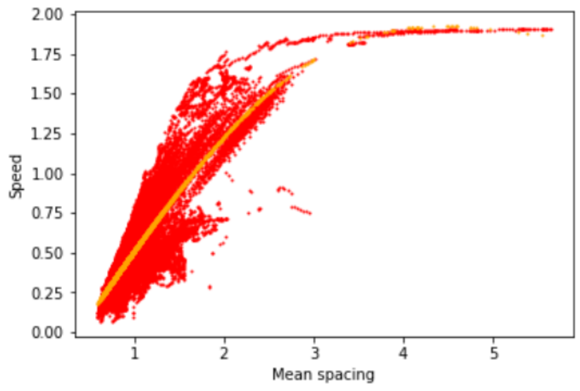
[a]



[b]



[c]



[d]

Figure 19: Data observable (*left*) and predictions(*right*) with respective FD model. Both Bottleneck_070 (*bottom*) and Corridor_85 (*top*).

model	***desired speed***	***pedestrian size***	***time gap***
FD Corridor Observations	0.957	0.274	1.259
FD Corridor Predictions	0.867	0.287	1.146
FD Bottleneck Observables	2.176	0.399	1.004
FD Bottleneck Predictions	2.354	0.404	1.019

Table 3: Parameter comparison between FD fitted on observations vs on predictions

Dropout Since in the paper the best model was very simple - one hidden layer with three units - and bigger models were leading to overfitting, we though that it might be a good idea to add regularization to tackle this phenomenon without reducing too much the complexity of the network. We tested both geometries, respectively the bottleneck of 70cm and the corridor of 85cm. In both cases we applied dropout after each fully-connected ("Dense" in Tensorflow) layer, with value 0.3 and 0.1. Unfortunately the results are not very good and Figure 21 shows that probably this technique was too harsh and led to the opposite effect, i.e. a sort of underfitting.

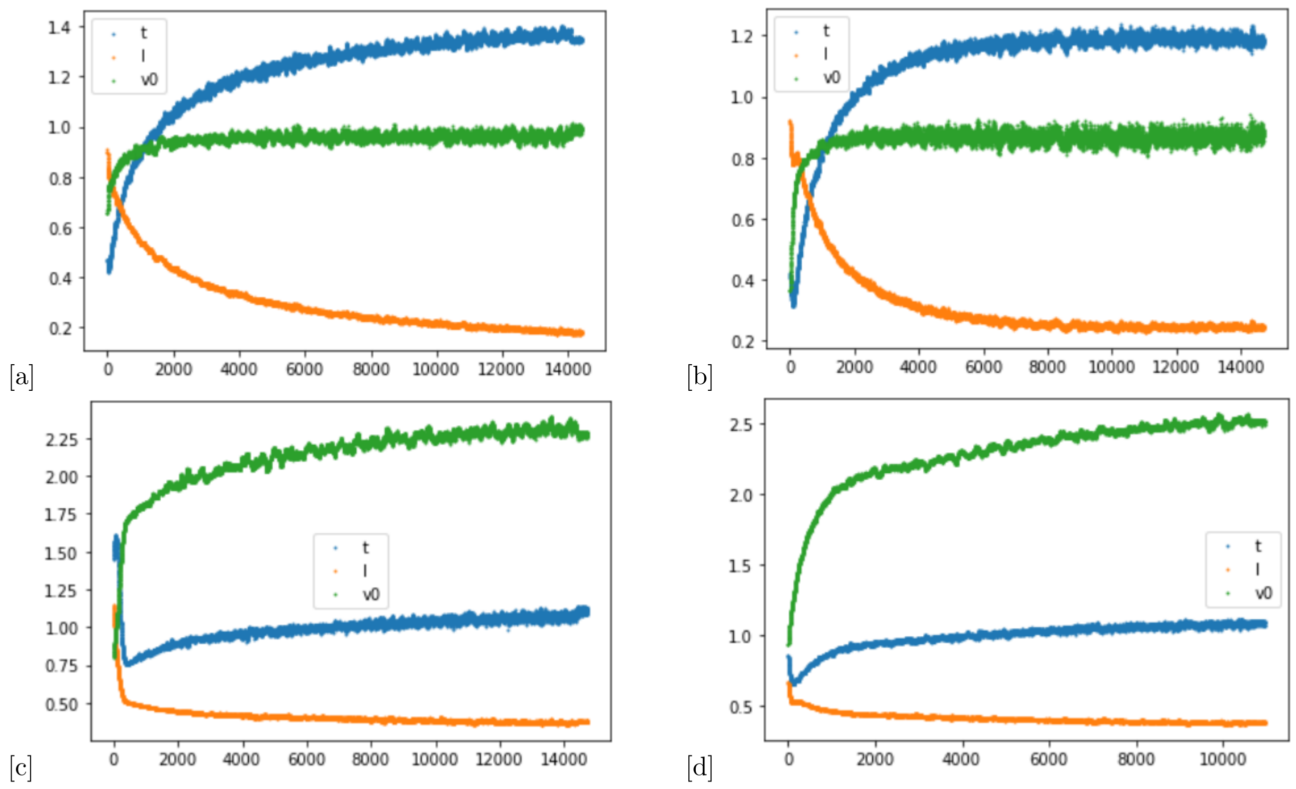


Figure 20: Parameters stabilization: FD model on corridor observable (*top left*), corridor prediction (*top right*), bottleneck observables (*bottom left*) and bottleneck predictions (*bottom right*).

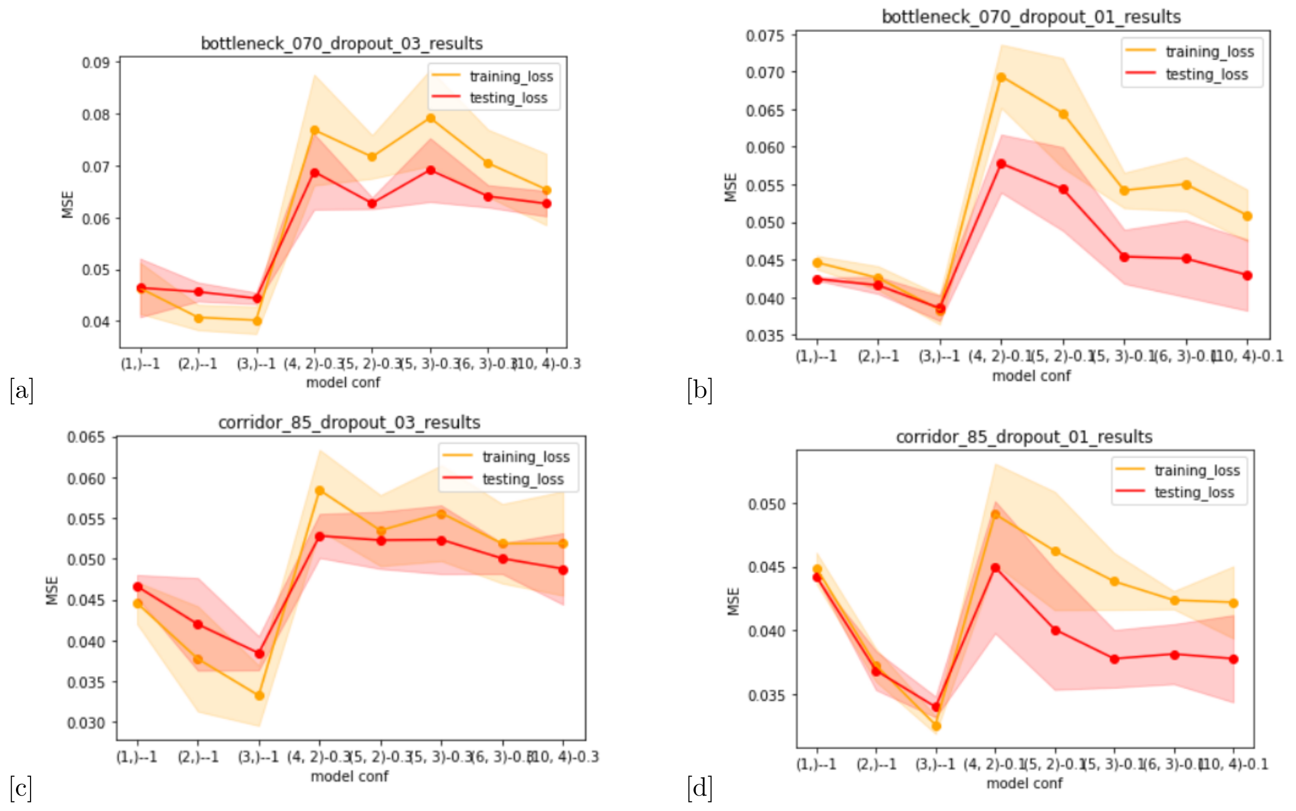


Figure 21: Results with dropout. The first row report the results on the bottleneck scenario with 70 pedestrians, while the second one is for the corridor scenario with 85 pedestrians. The plots in the first column have a dropout of 0.3, while in the second column the droout value is 0.1.

Principal components analysis We performed some experiments increasing or decreasing the number of attributes in the data used to train the neural network, trying to understand if the approach of the paper was improvable from this point of view. A natural development of this was to try to apply the PCA to the data and train the network in the PCA space, using a certain number of principal components. We tested the configurations and obtained the data reported in Table 4

Energy retained	Number of principal components	Loss
100%	21	0.03748 (std dev: 0.0013)
91.04%	17	0.03916 (std dev: 0.0019)
75.12%	12	0.03939 (std dev: 0.0006)
54.59%	7	0.04121 (std dev: 0.0013)
23.03%	2	0.05911 (std dev: 0.0005)

Table 4: Configuration and results obtained using PCA on the data to train the network.

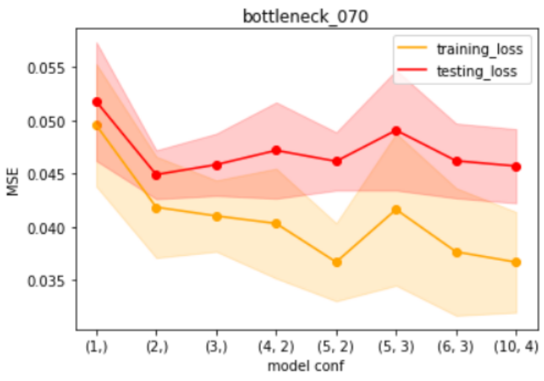
As we can see, 12 principal components provide anyway a good approximation with almost half of the inputs size. With 7, results start getting worse, bust still not that terrible, while with 2 - as can be expected - the loss increases and it is significantly worse than the one of equivalent models trained with the original data. Overall we think this is a nice way to reduce the input size, improving the speed of training, without manually selecting actual attributes to remove, leading probably to much worse performance.

Reduced input size The input of the neural network consists of 21 attributes: the mean spacing of the pedestrian w.r.t. its 10 nearest neighbors and the relative positions of those neighbors expressed as couples of coordinates $x - x_i$ and $y - y_i$. The mean spacing, therefore, can be derived from those neighbors positions. For this reason, we wanted to check if this attribute is really important and we tried to train and test a NN with the same data used so far, but without the mean spacing. The results are depicted in Figure 22, where the scenarios bottleneck with 70 people and corridor with 85 were used to test different architectures of neural networks (reported in the x-axis of each plot in Fig. 22). As it is possible to see, compared to the case with the original data (1st columns of Fig. 22), removing the mean spacing actually degrades slightly the performance (2nd column of Fig. 22). We can conclude that the attribute in object could be removed, with a small penalty in MSE. The obtained speedup, though, is negligible, therefore it might be wiser to instead keep it in the data. It would be interesting to experiments with a smaller number of nearest neighbors, but we were not able to do that due to time constraints.

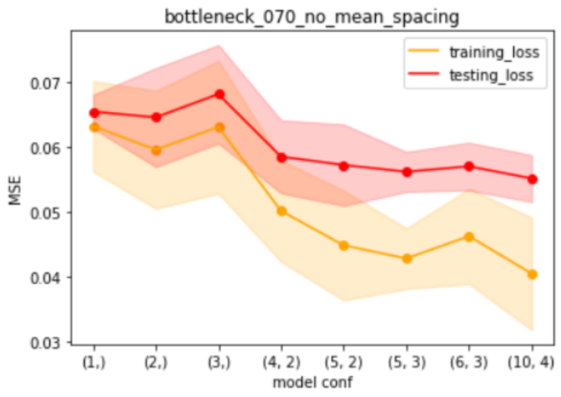
Change activation function On a side note, we also attempted changing the hidden layer activation function from *sigmoid* to *tanh*. We did not attempt *relu* since it is an activation function introduced for reasons connected to gradient propagation problem in much deeper nets, which is not our case at all. The results came out quite similar to the use of *sigmoid* and for sure not noteworthy even when changing the scenario at hand.

Report on task 5, Discussion on the approach and architecture

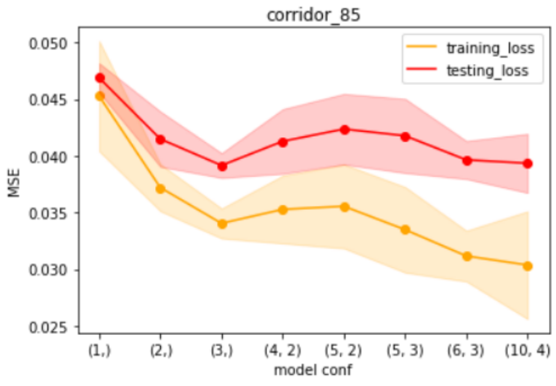
The classic approach to predict pedestrians' behavior is the fundamental diagram, which infers for example the speed as a function of few parameters with a clear physical meaning (e.g. mean spacing, time gap, etc.). This approach can be unsatisfactory in case of complex geometries, which of course influence the pedestrians' dynamics. For this reason, Tordeux et al. [4] propose to use a different approach: use a neural network, which instead has many parameters, but none of them has a clear meaning. The authors show how this model actually performs better, especially in cases of mixed scenarios (Figure 3) and that it is able to tell apart different geometries, in fact, when fitting the FD model on the NN's predictions, two distinct curves appear, one for the bottleneck and one for the corridor (Figure 4), highlighting that the network's prediction for one geometry and the other lie on two different distributions.
However, we noticed that different scenarios were producing substantially different plots and results, but in the paper only one specific scenario looks to be used, without actually specifying which one it is or how they got that specific ensemble of data. They also say that the pedestrians' speeds are given, but they actually need to be derived from the positions and the frame rate, since they are not present in the dataset. Even though computing them is pretty straight forward, there might be some details in the authors' process that a reader might miss. More in general, a lot of information is missing in the paper, making it difficult for the readers to exactly reproduce those results. For example, they state that the training is performed through



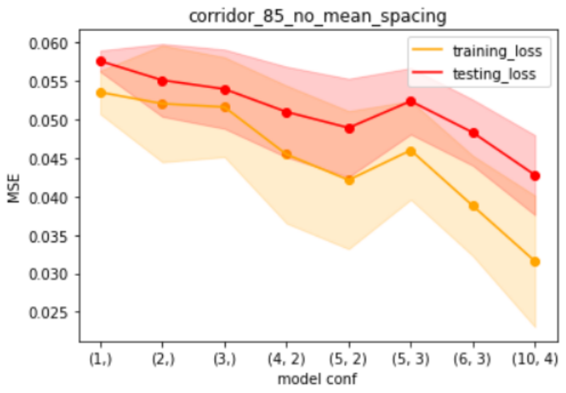
[a]



[b]



[c]



[d]

Figure 22: Comparison of the same networks trained with (1st columns) and without (2nd columns) mean spacing.

bootstrapping, but the size of the different subsamples and the number of folds in the cross-validation are not specified. The same happens for almost all the training hyperparameters such as the batch size, the number of epochs, the learning rate and so on. The best model is the neural network with only 3 hidden units, which is peculiar, being unusually shallow and simple. At this point they might have tried to apply some regularization techniques on larger networks, but this was not tested. Furthermore, as we have shown previously, some larger networks actually perform similarly to the one with 3 hidden neurons, and the error is in a descending trend as the complexity of the models increases (after a local minimum at (3,)). This further encourages to try larger networks with regularization as they might actually outperform simpler models. Our results with dropout were not better than the ones without it, but a more extensive test could be carried on, also with different regularization techniques other than dropout.

As a final point, we would like to mentions few additions and extensions that could be made to the project. First of all, our results using PCA seem promising, it could be exploited and other dimensionality reduction techniques could be explored. We also noticed that removing mean spacing degrades the performance a bit, but what about changing the number of nearest neighbors to be considered? Furthermore, our dropout was probably too harsh, but it could be tested on more complex geometries such as T-junctions, supermarkets simulations etc., which might require more complex networks where regularization is needed. Also, since we have information about the time (i.e. the time steps), it would be interesting to exploit it taking into consideration the speed of the pedestrian in the previous time step, using models such as RNNs or HMMs, encoding information about the past to predict the speed at each time step, while so far the time information was simply lost, sending to the network some "screenshots" of the simulation in a random order. One last, more ambitious attempt could be to use imitation learning / reinforcement learning to learn an agent who behaves like a pedestrian and learns to interact with others (avoiding them and dynamically changing its speed). The observations could be considered as expert's trajectories and, after having such agent, many of them could be placed in a scenario with a desired geometry to simulate the behavior of a crowd, making it possible for us to simply "read" their speeds.

References

[1] Forschungszentrum julich: Dataset of experiments with pedestrians. `https://ped.fz-juelich.de/database`.

[2] Dirk Helbing, Lubos Buzna, Anders Johansson, and Torsten Werner. Self-organized pedestrian crowd dynamics: Experiments, simulations, and design solutions. *Transportation Science*, 39:1–24, 02 2005.

[3] Andreas Schadschneider, Wolfram Klingsch, Hubert Klüpfel, Tobias Kretz, Christian Rogsch, and Armin Seyfried. *Evacuation Dynamics: Empirical Results, Modeling and Applications*, pages 3142–3176. Springer New York, New York, NY, 2009.

[4] Antoine Tordeux, Mohcine Chraibi, Armin Seyfried, and Andreas Schadschneider. Prediction of pedestrian speed with artificial neural networks. 01 2018.

[5] Antoine Tordeux, Mohcine Chraibi, Armin Seyfried, and Andreas Schadschneider. Prediction of pedestrian speed with artificial neural networks. 01 2018.