Report for exercise Final Project from group G

Tasks addressed:	5
Authors:	Angelos Kafounis Caner Karaoglu Joaquin Gomez Sanchez
Last compiled:	2022–08–31
Source code:	`https://github.com/joaquimgomez/ml-crowds-group-g`

The work on tasks was divided in the following way:

Angelos Kafounis	Task 1	34%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	34%
Caner Karaoglu	Task 1	33%
	Task 2	33%
	Task 3	34%
	Task 4	33%
	Task 5	34%
Joaquin Gomez Sanchez	Task 1	33%
	Task 2	34%
	Task 3	33%
	Task 4	34%
	Task 5	32%

Report on task 1, Description of the example, discussion of challenges, and your choice of method

Throughout this report sheet we are going to explain our findings after analyzing and trying to reconstruct equivalent results from the research paper "Prediction of Pedestrian Speed with Artificial Neural Networks" by Tordeux et al. [1].

In their research, the authors try to obtain the speed of a set of pedestrians walking in specific scenarios. They implemented and compared the results given by two different approaches. Both models have common inputs, i.e., the positions or velocities of surrounding neighbors and obstacles; and a common obvious output, i.e., the velocity, but it could be possible also to output the position or the acceleration rate.

The first model is the **Weidmann model for the fundamental diagram** [2]. This model is able to compute the velocity or the acceleration rate of the pedestrians using as input the mean spacing $\bar{s}_K$ with K (10 in the paper) closest neighbours ($\bar{s}_K = \sum_{i=1}^{K} \sqrt{(x-x_i)^2 + (y-y_i)^2}$), a time gap T, the pedestrian size ℓ and a desired speed ν_0. This model is a explicit linear function, i.e., it directly describes the reality through a mathematical function, which requires specific parameters and has the following form:

$$\nu = FD(\bar{s}_K, \nu_0, T, \ell) = \nu_0 \left(1 - e^{\frac{\ell - \bar{s}_K}{\nu_0 T}}\right) \tag{1}$$

On the other hand, the second model is a non-explicit nonlinear function, more specifically it is an **Artificial Neural Network** (Equation 2). This model allows us to have any of the outputs mentioned above and also any input, since it is flexible in construction and training. In the original paper, the authors used $2K+1$ inputs, i.e., the mean spacing and K relative positions (coordinates x and y).

$$\nu = NN(H, \bar{s}_K, (x_i - x, y_i - y, 1 \leq i \leq K)) \tag{2}$$

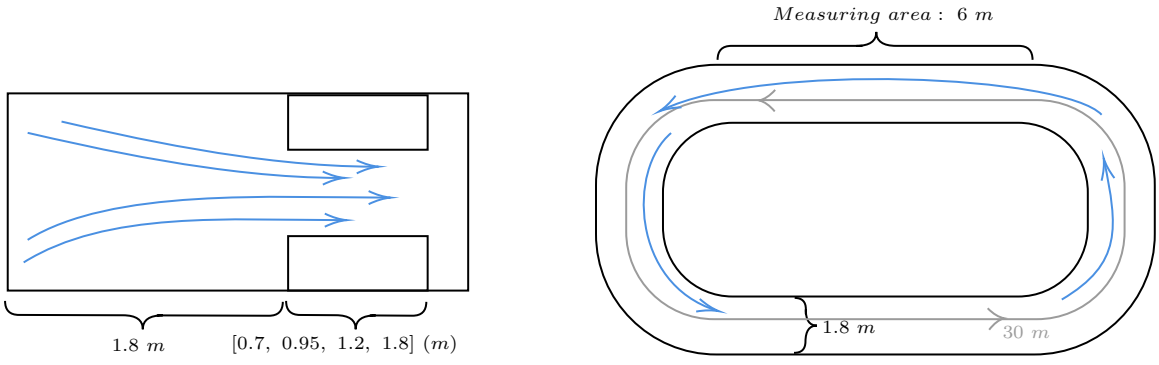
By taking a look at the original paper we notice that it analyzes the differences between a classical model, i.e., a model that we could say that it does not learn because it uses the **fundamental diagram**, a description of the relation between speed and surrounding distance spacing to the neighbours [3]; and a model that learns a function (i.e., there is no concrete mathematical given formulation) that potentially better fits the reality described in the data. We have decided to work with the last one, because it fits better with the objectives of this practical course.

The main point is also the "special" and specific characteristics of the used data which represents the challenge itself. The authors have used two different datasets [4, 5], both obtained in laboratory conditions consisting of:

- Dataset **R**: A ring/corridor experiment on a closed geometry of length 30 m and width 1.8 m for different density levels (from 0.25 to 2 ped/m^2) and different number of participants (from 15 to 230).
- Dataset **B**: A bottleneck geometry with 1.8 m with in front of the bottleneck and different bottleneck widths (from 0.70, 0.95, 1.20 to 1.80 m). Each experiment has 150 participants.

The measures were taken every 10 s to deal with pseudo-independent measurements and each sample contains around 2000 observations.

In order to illustrate the geometries of both scenarios and their implications, we have created the graphics that can be seen in Figure 1.



(a) Bottleneck geometry (dataset B)

(b) Ring geometry (dataset R)

Figure 1: Geometries in the datasets

This data represents a challenge because of the geometries. For the bottleneck case, we may expect that all the pedestrian walk in a normal human speed, changing the notion of "normal" speed for every age group as we saw in the previous exercises, or even faster. The pedestrians should reduce their velocities when reaching and crossing the bottleneck, because in a bottleneck we may expect a reduction in the mean space and an augmentation in the density. For the ring geometry, at least from our knowledge, we should not expect big changes on the speed around the ring, because the geometry does not change. Moreover, in the last scenario the geometry is not a straight line so it should be difficult for pedestrians to change their velocities if they have to border the ring at the same time, especially when there are people around. In this case, we expect that the behavior in a ring only depends on the number of pedestrians and the density.

The authors also illustrated this in Figure 2. We can see how the scenarios have a direct impact on how pedestrians behave in terms of velocity depending on the mean spacing with the 10 closest neighbors. As the authors stated in the original paper, it was observed that the speed for a given mean spacing tends to be higher in the bottleneck than in the ring, when the system is congested [1].

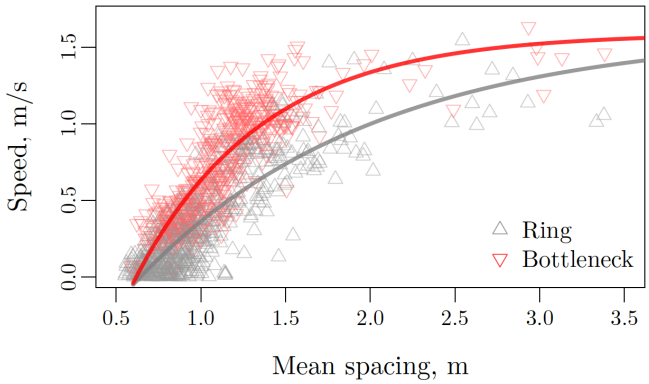


Figure 2: Pedestrian speeds as a function of the mean spacing with the 10 closest neighbours for both scenarios (from [1])

As previously mentioned, we are going to focus this project on the Neural Network model because it fits better with the objectives of this practical course, and it is possibly the model that offers more possibilities in terms of adaptation. Regarding the datasets, there are 4 available bottleneck datasets, from which we are going to use the first three, i.e., `uo-180-070`, `uo-180-095` and `uo-180-120`; and eight corridor datasets, from which we are going to use `ug-180-030`, `ug-180-060` and `ug-180-085`.

We are going to train and test the neural network introduced before for both sets/experiments. As in the paper, we are going to analyze how mixing of both datasets impacts the performance of the model.

Report on task 2, Implementation of the neural network

As mentioned in Task 1, we are going to implement and train a Neural Network, and here we are going to explain our implementation.

As a counterpoint, and from our point of view, the paper does not go deeply in the explanation of the neural network, therefore, we have made some decisions given the lack of information.

First, the authors used the R's package `neuralnet` [6], but we are going to use the modern Python framework `PyTorch`[1]. Concretely, we are going to use `PyTorch Lightning`[2], a library that allow us to prototype faster, while avoiding errors and delegating minor aspects of PyTorch. We have made this decision based on the fact that, while R is commonly used in the academia, it is very little lax to carry out specific and/or modern things. Also, we have that recently Pytorch has become also common in the academia, while sharing the space with TensorFlow in the industry. These aspects justify their use.

From the paper we know that the authors have used a training/evaluation schema consisting of cross-validation and bootstrapping over 50 sub-samples, using half of the data while training and the other half while testing. The optimization has been done using the basic back-propagation algorithm, the only one offered by the `neuralnet` package, together with the well-known Mean Square Error (MSE)[3] to estimate the error.

Regarding the neural network architecture, after trying the configurations (1), (2), (3), (4,2), (5,2), (5,3), (6,3) and (10, 4) ($(\cdot,\cdot)$ indicating the number of hidden layers and the number of units per layer, respectively), they found that the best one consists of a unique hidden layer with 3 nodes. They justified the decision base on the fact that while the MSE for the training continues decreasing for bigger network the test MSE does not, then it overfits because the learning gap increases. Thus, they trained a $(2 * K + 1)$-3-1 neural network, where the inputs are: $2 * K$ relative positions (i.e., $(x_i - x, y_i - y)$ for $1 \leq i \leq K$; two for two coordinates) and one for the mean spacing.

	Original	**Our implementation**
Language and Framework/Library	R + `neuralnet`	Python + Pytorch Lightning
Architecture	(2*K + 1)-3-1	(2*K + 1)-3-1
Activation function	?	ReLU
Optimization method	Back-propagation	Adam
Learning rate	?	1e-3
Error estimation	MSE	MSE
Training/Evaluation schema	Cross-validation + bootstrapping	Cross-validation + bootstrapping

Table 1: Comparison between the original implementation and our one

In Table 1 the difference between their implementation and our one can be seen. Apart from the already described aspects, we have also taken decisions about other relevant configurations. Regarding the activation function, even if it is one of the most important decisions, because it defines the non-linearity of neural networks, the authors have not specified the used one. Due to that, we have decided to use ReLU, a well-known and popular activation function due to its general purpose good behavior [7]. We also tried the Sigmoid activation, but it delivers worst results.

For the optimization method, another of the cornerstones of neural networks, from the description in the R manual [6] we only know that the package uses the basic and original back-propagation algorithm. This method is not currently being used because it is too slow, even if we are working with small neural networks, as in our case. Due to that, we decided to use the Adam optimization algorithm [8], which is considered nowadays the standard. Adam has different parameters to adjust the optimization, the learning rate, the exponential decay for the first moment estimates, the exponential decay for the second moment estimates, and a term to improve numerical stability, but we have used only the learning rate, since we do not expect to perform complicated optimizations. The paper does not mention a specific value, thus we use 1e-3 because it is the standard and the value that typically works fine for (relatively) big and small neural networks.

For the error estimation and training schema, we have decided to keep the original configuration, since we want to approximate (or maybe improve) their results. They did not mention the number of folds used in Cross Validation, thus we have decided to use 5 folds. For the bootstrapping, due to some problems we have faced, first with Google Colab and also with our computers, we decided to use 5 sub-samples with 9000 instances.

[1] Available at `https://pytorch.org`.

[2] Available at `https://www.pytorchlightning.ai`.

[3] The MSE is defined as $MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2$.

Regarding the batch size, we have used 32, because the paper does not mention anything about this either and this value is common for small neural networks. Lastly, in order to avoid overfitting, we have used Early Stopping with patience of 10 and a maximal 50 number of epochs. Almost all the training processes we have carried out have finished before the 50 epochs, which means that the Early stopping is useful in order to not reach the maximum number of epochs and overfit the model.

As a side note, we can mention that we have discarded possible (and also typical) improvements like dropout because, as we are going to explain, the model performs well with the introduced configuration.

In order to train the models and perform the experiments we have implemented the following methods:

- `simple_train`: This method performs a training of the model previously introduced and also computes the results for the test set. It outputs a plot with the training curves and the losses. This method is independent from the next ones.
- `train_and_evaluate`: It implements the K-Fold Cross Validation, outputting at the end the losses of the last step of training and validation, and the test set, for every fold.

```
def train_and_evaluate(model_args, kfolds, batch_size, max_epochs, train_validation_split, test_split):
  kf = KFold(n_splits=kfolds, random_state=1234, shuffle=True)

  # Train with K-Fold CV and save losses
  cv_training_losses = []
  cv_validation_losses = []
  cv_test_losses = []
  for fold, (train_idx, valid_idx) in enumerate(kf.split(train_validation_split)):
    print("CV ITERATION {}".format(fold))
    train_loader = DataLoader(CrowdDataset(train_validation_split.iloc[train_idx].reset_index(drop=True)), batch_size = batch_size, shuffle = True, num_workers = 2)
    val_loader = DataLoader(CrowdDataset(train_validation_split.iloc[valid_idx].reset_index(drop=True)), batch_size = batch_size, shuffle = True, num_workers = 2)
    test_loader = DataLoader(CrowdDataset(test_split.reset_index(drop=True)), num_workers = 2)

    # Train model
    model = TordeuxNet(model_args)
    trainer = pl.Trainer(accelerator="gpu", devices=1, max_epochs = max_epochs, callbacks=[EarlyStopping(monitor="val_loss", mode="min", patience=10), TQDMProgressBar(refresh_rate=15)])
    trainer.fit(model, train_loader, val_loader)

    # Save training and validation losses
    cv_training_losses.append(float(model.training_metrics[-1]))
    cv_validation_losses.append(float(model.validation_metrics[-1]))

    # Compute and save test losses
    test_mse = trainer.predict(model, test_loader)
    cv_test_losses.append(float(torch.stack(test_mse).mean()))

  return cv_training_losses, cv_validation_losses, cv_test_losses
```

Figure 3: Implementation of the `train_and_evaluate` method

- `bootstrap_cv`: Using the previous method, it performs bootstrapping combined with K-Fold Cross Validation for the given number of bootstrapping sub-samples, number of instances and folds.

```
def bootstrap_cv(dataset, bootstrapping_iterations, bootstrapping_num_samples, model_args, folds, batch_size, max_epochs, diff_test_dataset = None):
  bootstrap_training_losses = []
  bootstrap_validation_losses = []
  bootstrap_test_losses = []
  for i in range(0, bootstrapping_iterations):
    print("BOOTSTRAP ITERATION {}".format(i))
    # Select random for bootstrapping iteration at random
    bootstrap_data = dataset.sample(n = bootstrapping_num_samples)

    # Half of the data for training, half for testing
    if diff_test_dataset is not None:
      train = dataset.sample(n=int(bootstrapping_num_samples / 2))
      test = diff_test_dataset.sample(n=int(bootstrapping_num_samples / 2))
    else:
      train, test = train_test_split(bootstrap_data, test_size=0.5)

    # Cross-validation
    train_losses, validation_losses, test_losses = train_and_evaluate(model_args, folds, batch_size, max_epochs, train, test)

    bootstrap_training_losses.append(np.mean(train_losses))
    bootstrap_validation_losses.append(np.mean(validation_losses))
    bootstrap_test_losses.append(np.mean(test_losses))

  # Print results
  print("Train mean: {} -- Train STD; {}".format(np.mean(np.array(bootstrap_training_losses)), np.std(np.array(bootstrap_training_losses))))
  print("Validation mean: {} -- Validation STD: {}".format(np.mean(np.array(bootstrap_validation_losses)), np.std(np.array(bootstrap_validation_losses))))
  print("Test mean: {} -- Test STD: {}".format(np.mean(np.array(bootstrap_test_losses)), np.std(np.array(bootstrap_test_losses))))
```

Figure 4: Implementation of the `bootstrap_cv` method

As it can be seen in the implementation in Figure 4 we have consider an extra dataset input called `diff_test_dataset` that will allow us to perform experiments for the Task 4 that requires different datasets for training and testing.

Regarding the implementation of the model, we have implemented the class `TordeuxNet` following the explained before configuration. It extends the PyTorch Lightning class `LightningModule`, which allows us to only have to implement the model description, the forward step, and the training/validation/prediction steps. Together with the model, we have implemented the class `CrowdDataset`, which extends `Dataset` from PyTorch, in order to make available the data to be used by the data loaders.

```
class TordeuxNet(pl.LightningModule):
    def __init__(self, args):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1 + 2 * args["k"], 3),
            nn.ReLU(),
            nn.Linear(3,3),
            nn.ReLU(),
            nn.Linear(3, 1),
        )
        self.lr = args['lr']
        self.loss = F.mse_loss

        self.training_metrics = []
        self.validation_metrics = []

    def forward(self, x):
        speed = self.net(x)
        return speed

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=self.lr)
        return optimizer

    def training_step(self, train_batch, batch_idx):
        x, y = train_batch
        x_hat = self.net(x.float())
        loss = self.loss(x_hat, y.float().unsqueeze(1))

        self.log('train_loss', loss)

        return {"loss": loss}
```

Figure 5: Implementation of the neural network using PyTorch Lightning

Figure 6 shows our implementation of the described neural network using PyTorch Lightning. We can appreciate how easy is to describe the neural network in the initialization method. For the forward step, we just have to take care of the input and the output of the network, while the backward step is totally delegated to PyTorch Lightning. We also have that for every training step, i.e. for every minibatch, we compute the network results and the loss. This is exactly the same for the not shown methods `validation_step` and `test_step`. This loss from the validation step is by the PyTorch `Trainer` module to guide the training and the possible early stopping.

```python
class CrowdDataset(Dataset):
  def __init__(self, dataframe):
    self.x_train=torch.tensor(dataframe.loc[:, dataframe.columns != 'SPEED'].values)

    self.y_train=torch.tensor(dataframe['SPEED'].values)

  def __len__(self):
    return len(self.y_train)

  def __getitem__(self,idx):
    return self.x_train[idx], self.y_train[idx]
```

Figure 6: Implementation of `CrowdDataset`

The implementation for the training can be found in the file `training.py`, and the one for the model and dataset in `NeuralNetwork.py`.

Another important aspect that we are going to explain in this task is the data preprocessing. Every given data sample consists of the pedestrian ID, the frame (with a frame rate of 1/16) and the coordinates X, Y and Z. In order to compute the speed (the ground truth of the model), the K closest neighbours, the mean spacing and the K relative positions, we have implemented the next methods that can be found in the file `data_processing.py`:

- `compute_speed`: The original dataset does not explicitly have the speed for every sample, then, this method computes the velocity for every pedestrian and frame. To do so, it substracts for every instance its position and the position of the next frame, and then multiplies by 16, because we have that the frame rate is 1/16. We know that, eventually, every pedestrian has one frame which is the last one, i.e., we cannot compute the speed. This last frame is deleted in every pedestrian for the final dataset.
- `obtain_neighbors`: In order to obtain the mean spacing and the relative positions for each pedestrian and frame, this method obtains the neighbors. We obtain for every pedestrian and frame an array with the IDs, x-y coordinates and distances to every pedestrian in the same frame. This array is sorted by the distance.

```python
def obtain_neighbors(dataframe):
  # Compute neighbors of each pedestrian by frame, because there are frames with more or less pedestrians
  neighbors_positions_per_id_frame = {}
  for _, frame in dataframe.groupby(by='FRAME'):
    pedestrian_ids_frame = set(frame['PedestrianID'])
    for id in pedestrian_ids_frame:
      neighbors_positions = []
      for neighbor_id in pedestrian_ids_frame - set([id]):
        xy_1 = np.squeeze(frame[frame['PedestrianID'] == id][['X', 'Y']].to_numpy())
        xy_2 = np.squeeze(frame[frame['PedestrianID'] == neighbor_id][['X', 'Y']].to_numpy())

        neighbors_positions.append([neighbor_id, xy_2[0], xy_2[1], np.linalg.norm(xy_1 - xy_2)])

      # Sort neighbors information by the distance
      neighbors_positions.sort(key=lambda e: e[3])
      neighbors_positions_per_id_frame[(id, frame.iloc[0]['FRAME'])] = neighbors_positions

  # Add column with neighbors to each pedestrian and frame
  dataframe['NEIGHBORS'] = np.nan
  dataframe['NEIGHBORS'] = dataframe['NEIGHBORS'].astype('object')
  for index, row in dataframe.iterrows():
    dataframe.at[index, 'NEIGHBORS'] = neighbors_positions_per_id_frame[(dataframe.iloc[index]['PedestrianID'], dataframe.iloc[index]['FRAME'])]

  return dataframe
```

Figure 7: Implementation of the `obtain_neighbors` methods

From all the data preprocessing process this step is the most complex since it is $O(N^3)$ in terms of computational complexity because at most we will need N^3 iterations being N the number of pedestrians if every pedestrian appears in every frame. We decided to saved every normalized and preprocessed dataset for $K = 10$ because it is not feasible to perform this computation every time we need the data.

- `compute_mean_spacing`: Given a number K of pedestrian, this method computes for every pedestrian and frame the mean spacing between it and its K closest neighbors. Before this, it deletes the frames with less than K pedestrians.
- `obtain_relative_positions`: This method simply add to every pedestrian and frame an array with the relative positions, i.e., $(x_i - x, y_i - y)$, to the K closest neighbors.
- `preprocess_dataset`: It concatenates all the previous methods in order to obtain a final data frame with all the required information, i.e., the speed or ground truth, the mean spacing and the K relative positions.
- `prepare_data_for_training_testing`: This method splits the relative positions in $2 * K$ columns, outputting at the end a dataframe with $2 * K1 + 1$ columns: $2 * K$ for the positions, 1 for the mean spacing and 1 for the speed, in order to compute losses. This final dataset will be adequately managed by the aforementioned `CrowdDataset` class.

As mentioned in the paper, we normalize the data before being consumed by the training methods. Firstly, we tried the standard deviation normalization, but it did not work as expected because we obtained losses bigger than the ones in the paper. Finally, we decided to normalize using min-max normalization, achieving then comparable losses. We noticed that while normalizing the data we obtain losses < 1, if we do not normalize then the losses are always > 100, fluctuating between 200 and 600. This gives to us intuition about the relevance of normalizing the data when training a neural network.

In order to decide on the final configuration, we tested some batch sizes and learning rates, but not in a standardized way, because we just wanted to double-check our decisions. We found that the model performs similarly for learning rates 1e-2 and 1e-4, and for batch sizes 64 and 128. Thus, we decided to keep the aforementioned values (1e-3 and 32).

We have trained our final model with the configuration explained before using a set of data combining the datasets `ug-180-060` (ring) and `uo-180-07` (bottleneck) (see Task 4 for the explanations about this decision). The combination consists of 5000 samples from the ring scenario and 5000 samples from the bottleneck scenario, in order to have a balanced dataset. We have trained it using a typical 80-10-10 data splitting schema, 32 as batch size, 50 as the maximum number of epochs, and early stopping with patience 10.

As we can see in the training curves from Figure 8 the model learned in a good way, with a small/negligible learning gap. The model reached its best loss at epoch 3 or 4, and then due to the patience of the early stopping, it finished after 10 epochs. The obtained test mean square error over the 10% of the dataset is 0.0370. Comparing this value with the last epoch learning MSE of 0.0379 and the last epoch validation MSE of 0.0384 we can conclude with high confidence that our model is not overfitted. Due to the small achieved value, we can also state that we probably reached the maximum power of this model for this specific problem, or maybe that we cannot improve it more because there is an intrinsic random component (noise) in the data, as could be expected.

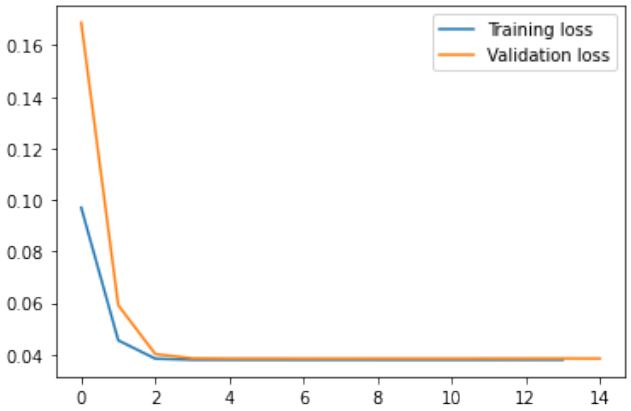


Figure 8: Training and validation curves

Report on task 3, Tests on a simple example

Since it makes no sense to prepare a test set, e.g. using Vadere, because it will be completely different from the conditions available in the laboratory, we have discarded this idea. Then, instead of just using a set for testing the model we decided to do the following.

There is a current in Machine Learning, for example in NLP for Language Models, that advocates training machine learning models in the most generalized way, and then testing/benchmarking these models for concrete or specific cases, this in order to test the generalization of them far from simple predictions with a test set.

Following this idea, and as we stated previously in the Task 2 part of this report, we trained our model using the average datasets in terms of densities from the provided ones, i.e., the datasets `ug-180-060` (ring) and `uo-180-070`. Combining as explained both datasets in order to train the model we expect two things:

- The model learns about general characteristics of human behavior for average densities (in our provided sets)
- The model learns specific consequences of different space geometries in the speeds, connecting this also with the mean spacing (e.g., reduced in a bottleneck scenario) and relative distances of K pedestrians.

As we stated also in Task 2, this model seems to predict correctly over the same mixed dataset. Now, let us see what happens for other densities.

Concretely, we have decided to use two datasets from the ring ones (`ug-180-030` and `ug-180-085`, one with more density and one with less density than the used training corridor set) and two from the bottleneck ones (`uo-180-095` and `uo-180-120`, both with more density than the training bottleneck set). From every dataset, we sampled randomly 5000 instances. Table 2 summarizes our findings the findings

	Ring/Corridor datasets		**Bottleneck datasets**	
	`ug-180-030`	`ug-180-085`	`uo-180-095`	`uo-180-120`
MSE	0.0227	0.0251	0.0243	0.0299
StDev	0.0334	0.0420	0.0299	0.0258

Table 2: MSEs and standard deviations for the prediction over different datasets

From the table, we can extract some conclusions about the generalization capacity of the model for the given training data. We can see that the MSE is between 0.02 and 0.3 for all the datasets, this is a comparable value with the test set used at the end of the training, which had an MSE of 0.0370. We could even state that the used test set is harder than these ones discussed here.

Nevertheless, we have to consider the standard deviation. For all the cases, it indicates clearly that, while the mean is good compared to the original test, it is not absolutely true, because there is large variability. This can be indicating to us that there are situations learned better than others.

Inspired by Exercise 4 we have performed PCA over the data used in these tasks. For that purpose, we have used the PCA based on SVD that we implemented for Exercise 4.

We wanted to perform this analysis because, due to the characteristics of the dataset, it could be great to have an intuition of the relevance of the different dataset components.

We found (see Figure 9) that around 50% of the variance is explained with the first two components (one of them the mean spacing) and we need a total number of 16 components in order the explain at least $\geq 90\%$ of the variability. Since we have 21 total components (we dropped the velocity for the PCA), and because of the model simplicity, we can state that we do not need to perform PCA to obtain a less complex dataset and then be able to perform training in an easier way.

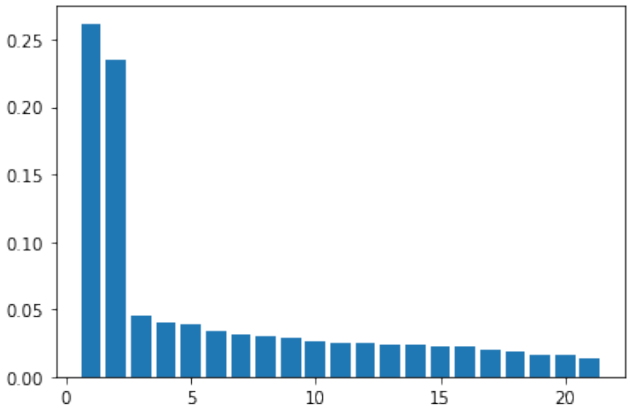
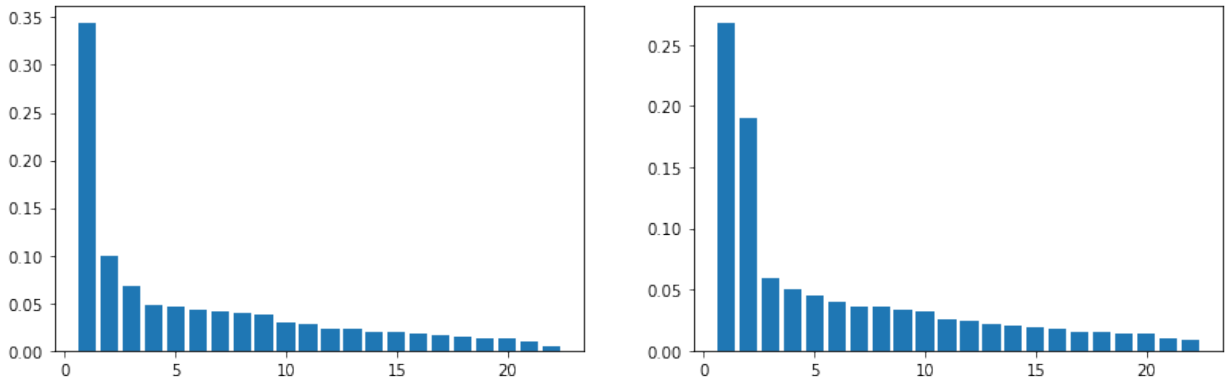


Figure 9: Energy per dataset component

Interesting conclusions can also be extracted if we look at the energy per component of both R and B datasets after mixing them. First, we have that we need 15 components in order to explain more than 90% percent of the variance, which means that the mixed set is harder than the separated sets. The other point is that for the dataset B, around 35% of the variance is explained by the mean spacing component. This connects with the introduced in the first task, that in the ring scenario the mean spacing has a big impact, at least compared with other aspects.



(a) B (b) R

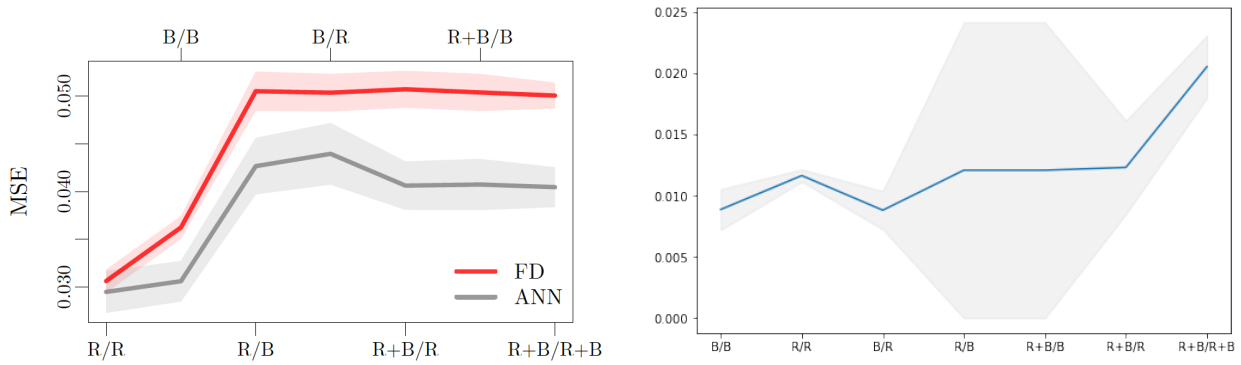
Figure 10: Energy per (separated) datasets component

Report on task 4, Comparison to their results

As mentioned in Task 2, in this one we are going to explain why we have trained our final model in the exposed way, and also we compare our results with the ones from the paper.

We have performed the same analysis as in the paper in order to decide the best combination of datasets for training and testing. I.e., we have performed the following analysis ($\cdot/\cdot$ indicates the training and the testing datasets): R/R, B/B, R/B, B/R, R+B/R, R+B/B, and R+B/R+B. These results helped us to choose the training data for our final model.

As happened for other aspects, the paper does not mention which dataset the authors used for this analysis. Therefore, we have decided to used the datasets `uo-180-070` (bottleneck) and `ug-180-060` (corridor/ring). This decision is based on the fact that both datasets are expected to have closed densities, so we expect they will work fine if we combine them in some way. In Figure 11 a comparison between our results (mean and standard deviation of 5 bootstrapping sub-samplings of 9000 elements) and the results from the paper (mean and standard deviation of 50 bootstrapping sub-samplings over an unknown number of elements) can be seen.



(a) Results in [1]

(b) Our results

Figure 11: Results for different training and testing sets

The first thing we may notice is that our model outperforms both models from the paper in terms of MSE for all the combinations, even if we consider the worst standard deviation, i.e. for the combinations R/B and R+B/B. Table 3 summarizes the improvements over the results for both paper methods. All the improvements are ¿ 50. Maybe it can be argued that our results have been obtained with a smaller number of bootstrapping sub-samples, which is actually true, but we have decided to follow with these results due to the lack of computation resources and time limitations.

	Improvement in terms of MSE						
	B/B	**R/R**	**B/R**	**R/B**	**R+B/B**	**R+B/R**	**R+B/R+B**
m**With respect to FD**	∼70%	∼65%	∼90%	∼80%	∼80%	∼80%	∼60%
With respect to ANN	∼85%	∼65%	∼85%	∼75%	∼75%	∼75%	∼50%

Table 3: Summarization of the improvements in terms of MSE

We have that in our case the models that better perform are B/B and B/R (the one with higher improvement), while for their model is R/R and B/B. In any case, the difference in our case between all the combinations except for R+B/R+B is really small, on average. In their cases, the combination that performed worst was B/R, but not far from R+B/R+B (our worst one). For use, the combination R+B/R+B is clearly the worst of all (also the one with less improvement concerning their neural network), but as it has been mentioned, it is, at the same time, better than their one.

As we have that R+B/R+B outperforms all the combinations for the author's model, although our best combinations are B/B and R/R, we decided to do the experiments/tests of the Task 3 training a model with the combination R+B. This is because, as explained, nowadays it is taking relevance the models benchmarking, i.e., we want to see how well the model generalizes for different concrete scenarios having that the model was trained in a general way. And, from our understanding, this combination is the one that better generalizes the problem.

As an extra note, we can easily notice that our model outperforms their FD model. This is not a surprise since their ANN model already outperformed the ANN. The discussion now could be how an artificial neural network makes possible the improvement of the results, given that it is a "free" model, i.e., it has no previous impositions describing the world. This shows that sometimes a "free" model is better than one already defined model, like FD.

With the results here exposed, we can conclude, as it has stated in the paper, that the data-driven approach using neural networks is suitable for the prediction of pedestrian performances in both scenarios, at least when using as input the mean spacing and the relative position of the $K = 10$ closest neighbors. This is good new since it could be easy for some scenarios like the complex geometries of the concerning scenarios to simply train a network on data instead of trying to define a mathematical expression that describes or approximates the reality.

Another interesting point is the implications and the reasons behind the use of a small neural network as the one here analyzed. This is going to be discussed in the following task.

Report on task 5, Discussion of the approach and architecture

With the aim of finding the best neural network architecture, the authors tested different architectures by increasing and decreasing the number of hidden layers and units per layer. Finally, they came to the conclusion that the model that delivers the best results for the tested data is a simple one layer with three neurons neural network (see their results in Figure 12).

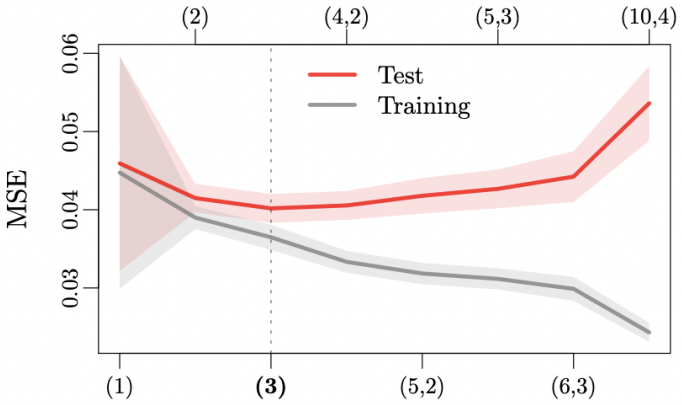


Figure 12: Losses with respect to different architectures of Neural Network

They noticed that by increasing the number of units/hidden layers of the model, thus making it more complex, it delivered highly overfitted models. The reason behind this phenomenon is that the dataset is quite simple. It consists only of $2 * K + 1$ closely-related dimensions.

A Neural Network is, theoretically, a universal approximator, i.e., we can use it to approximate any type of function because it is like a generalized regressor. Broadly speaking, and based on this fact, we can demonstrate that a network only weights different input parameters to produce a result, i.e., it performs a regression. Using this point of view, it makes sense to have a small neural network that weights the different inputs in some way and does not produce or derive extra information through transformations. This is what we expect that is happening with the proposed neural network, and it obviously justifies the use of small networks for low-dimensional datasets.

The crux of the question is that this utterly differs from the current standpoint in Machine Learning, i.e., overparametrization + minibatching. The main difference is that we are clearly performing a regression of different parameters in order to approximate the speed of the pedestrians, and this is, from far, an easy task that is expected to not require complex operations. For other machine learning tasks, especially in the context of classifying things, we typically do not have obvious inputs to simply weight, hence we cannot define, let us call it, "classical regressions". This has resulted in us having a widespread usage of overparametrization, because we need highly complex models with thousands of parameters to represent weights and transformations of the inputs, and these parameters need a heavy training. This necessity of complexity, from our understanding, is based on the limitation in our representation of the information and the limitations of the neural networks, because in the end they are only regressors, and we are using them to approximate functions to perform regression of problems that not necessarily can be solved through a regression.

References

[1] Antoine Tordeux et al. *Prediction of Pedestrian Speed with Artificial Neural Networks*. 2018. URL: https://arxiv.org/abs/1801.09782.

[2] Ulrich Weidmann. "Transporttechnik der fußgänger: transporttechnische eigenschaften des fußgängerverkehrs, literaturauswertung". In: *IVT Schriftenreihe* 90 (1993).

[3] Armin Seyfried et al. "The fundamental diagram of pedestrian movement revisited". In: *Journal of Statistical Mechanics: Theory and Experiment* 2005.10 (2005), P10002.

[4] Antoine Tordeux et al. *Data from: Prediction of Pedestrian Speed with Artificial Neural Networks*. Nov. 2017. DOI: 10.5281/zenodo.1054017.

[5] Christian Keip and Kevin Ries. "Dokumentation Von Versuchen Zur Personenstromdynamik-Projekt "Hermes"". In: *Institute for Advanced Simulation (IAS), Forschungszentrum Jülich* (2009).

[6] S Fritsch, F Guenther, and M Suling. *Neuralnet: Training of neural networks [Computer software manual]*. 2012.

[7] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011, pp. 315–323.

[8] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).