2AMM10 2021-2022 Assignment 2 & 3: Group 28

Stan Meijerink, 1222737 Sam van der Velden, 1017871
Danylo Vasylyshyn, 1815709

June 20, 2022

1 Task 2

1.1 Problem formulation

The goal is to predict the x and y coordinates of the locations of each of the 5 particles after a certain time (0.5, 1, 1.5 seconds) has passed. These coordinates will be dependent on the starting coordinates, the starting velocities, the charge of each of the 5 particles, and their previous locations.

This problem can be described as a sequential prediction. The model will need to predict a sequence of numbers (coordinates) based on an earlier sequence of numbers (initial positions, velocities, charges). Then with the new coordinates it will need to predict the next coordinates down the line.

For the performance of the model the fraction of the euclidean distance between the predicted end point and the actual end point relative to the sum of the euclidean distance between the actual end point and the starting point together with the euclidean distance between the predicted end point and the actual end point will be determined by using the following formula.

$$performance(x_p, y_p, x_a, y_a, x_i, y_i) = \frac{\sqrt{(x_a - x_p)^2 + (y_a - y_p)^2}}{\sqrt{(x_a - x_p)^2 + (y_a - y_p)^2} + \sqrt{(x_a - x_i)^2 + (y_a - y_i)^2}}$$

In the above seen formula x_p and y_p represent the predicted end point coordinates for each particle from the model, x_a and y_a represent the actual end point coordinates for each particle of the target, and x_i and y_i represent the initial starting point for each particle.

1.2 Model formulation

For this model an RNN will be used in the form of an LSTM to deal with the vanishing/-exploding gradient problem. This RNN will have a single input array and a single output array.

The input of the model consists of a 2-dimensional array where the outer most dimension consists of the number of simulations. The dimension within contains, for each simulation, 25 data points. These are the x-coordinates of the positions (1-5), then the y-coordinates of the positions(6-10), then the x-velocity (11,15), then the y-velocity (16,20), and then the charge of each of the particles (21,25). The ordering is done in such a way that for each data group of 5 the first point represents that data for the first particle and so on-wards.

The model outputs a 2-dimensional array where the outer layer again represents the number of simulations. Then the inner layer has for each simulation 20 data points. Where the first 5 are the x-coordinates, then the following 5 are the y-coordinates, then we get 5 data-points for the x-velocity and 5 for the y-velocity. The reason that the velocity is also in the output is while for every step the LSTM-cell makes it is given the output of the previous cell. To get a better representation of the data it was assumed that it would make sense to also predict the velocities so that they could be used as new input data again.

The loss function is a determined with the following formula as given below. In the formula x_p is the predicted x-coordinate from the location of the particle, y_p is the predicted y-coordinate from the location of the particle, x_a is the actual x-coordinate from the location of the particle, and y_a is the actual y-coordinate from the location of the particle.

$$Euclid\_distance(x_p, y_p, x_a, y_a) = \sqrt{(x_a - x_p)^2 + (y_a - y_p)^2}$$

$$loss = Euclid\_distance$$

The euclidean distance formula calculates the distance between the predicted point to the actual point. The loss is then calculated from this euclidean distance with the option of adding extra details like taking the square root of the euclidean distance if this improves the model. The experimentation will go deeper into this.

1.3 Implementation and training

The model (see figure 1) which is implemented consists of an LSTM-cell which is given the input and an initial h_t and c_t which are both initialized with 0's. Both h_t and c_t are 2-dimensional array's where the first dimension represents the simulation and the second dimension contains the features of the model, which consist of 256 features per simulation.

Before the data goes through the LSTM-cell it will go through a linear layer to encode the data and generate 256 features from the input. Then after the data has gone trough the LSTM-cell both h_t and c_t are updated. Then with the h_t an output will be calculated using a linear layer. This layer brings down the 256 features per simulation to 20 numbers of which the first 10 represent the x and y coordinates and the latter 10 the x and y velocity.

The data will then, when it has not reached the final time target, be appended by the charges and inputted into the next encoder layer after which it will be inputted into the next LSTM-cell together with the previous h_t and c_t output. While there are 3 time targets there are 3 LSTM-cells in the model.

After an time target is reached the output data will be appended to an output list which will in the end return the predictions for all time targets. The loss will be calculated for each output and will be back propagated through time.

For the training a teacher force of 0.5 will be used between the decoder of an LSTM-cell and the encoder of the next LSTM-cell. This teacher force will replace the predicted output with the actual correct data points half of the time. This choice was made to let the data not only train on the correct data points but also on some possibly incorrect predictions to get to a more accurate final time prediction.

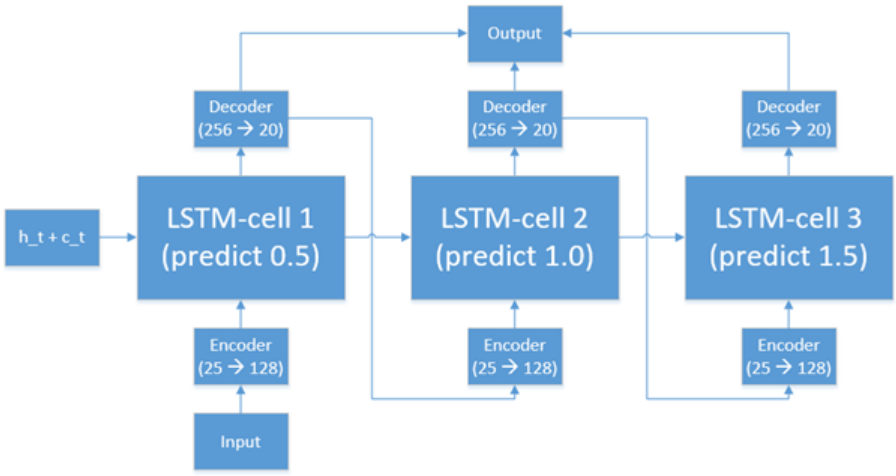

Figure 1: The architecture of a the default model

It was found during the model creation that using different LSTM-cells gave an higher model performance then just one LSTM cell. This change was most apparent for the time targets further away. The results from both types of model can be seen below in table 1.

	1 LSTM-cell (3 times)	3 LSTM-cells
Training loss	0.3972	0.3342
Validation loss	0.4957	0.4544
Testing loss 0.5	0.2917	0.2915
Performance 0.5	0.6348	0.6320
Testing loss 1.0	0.4737	0.4314
Performance 1.0	0.6543	0.6733
Testing loss 1.5	0.7346	0.6526
Performance 1.5	0.6379	0.6615

Table 1: Performance of different model styles

1.4 Experiments and results

During the experimentation the following experiments have been performed and will be described in detail below.

- Change loss formula
- Adding/changing linear layers
- Change batch size
- Change output hidden dimension size of the LSTM-cells
- Change the teacher force
- Inspect at amount of epochs
- Compare the model to a simple linear model

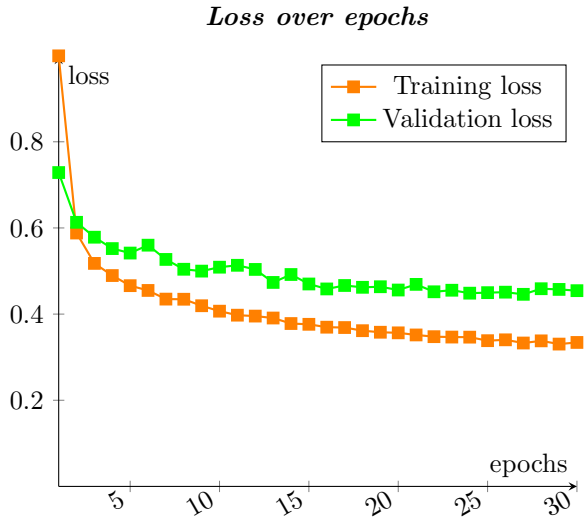

Figure 2: Experiment on amount of epochs compared to the training and validation loss

Figure 2 describes the epochs when training the model. As can be seen after about 25 epochs the drop in loss starts to get quite small. An experiment with over 100 epochs showed that after a while the model would start to over-fit and the validation loss would get worse again. This combined with the fact that the epochs run fast with only about 5 seconds per epoch, 30 epochs will be sufficient to train the data.

	64	128	256 (default model)	512	1024	2048
Training loss	0.3807	0.3612	0.3342	0.2959	0.2637	0.2579
Validation loss	0.4758	0.4663	0.4544	0.4698	0.4719	0.4541
Testing loss 0.5	0.2941	0.2956	0.2915	0.2968	0.2939	0.3030
Performance 0.5	0.6315	0.6303	0.6320	0.6257	0.6249	0.6173
Testing loss 1.0	0.4568	0.4479	0.4314	0.4637	0.4696	0.4612
Performance 1.0	0.6620	0.6659	0.6733	0.6583	0.6555	0.6581
Testing loss 1.5	0.6923	0.6713	0.6526	0.6818	0.6684	0.6513
Performance 1.5	0.6493	0.6557	0.6615	0.6534	0.6568	0.6607

Table 2: Experiment in changing the output hidden dimension size of the LSTM-cells

Table 2 describes the effect of changing the amount of output hidden dimensions of the LSTM-cells in the model. As can be seen the model performs best at 256 hidden dimensions. 2048 hidden dimensions also performs quite well, however takes too long to run in comparison to 256 hidden dimensions.

	Default model	Switched model
Training loss	0.3342	0.3044
Validation loss	0.4544	0.3469
Testing loss 0.5	0.2915	0.3613
Performance 0.5	0.3680	0.2997
Testing loss 1.0	0.4314	0.3227
Performance 1.0	0.3267	0.4332
Testing loss 1.5	0.6526	0.3342
Performance 1.5	0.3385	0.6475

Table 3: Experiment in switching loss and performance measures

Table 3 describes the effect of switching the performance and loss functions. To be able to execute this the value of the performance as described in the problem description was inverted and thus calculated as $1 -$ [normal performance]. Thus in this case a performance (default model) and a loss (switched model) of close to 0 are preferred and 1 is the maximum. As can be seen switching the loss and performance made no difference within the 0.01 run to run variation of the model.

	default model	$\sqrt{loss}$	$loss^2$
Training loss	0.3342	0.5544	0.1839
Validation loss	0.4544	0.6194	0.3875
Testing loss 0.5	0.2915	0.5022	0.1776
Performance 0.5	0.6320	0.6357	0.6034
Testing loss 1.0	0.4314	0.6142	0.3378
Performance 1.0	0.6733	0.6759	0.6523
Testing loss 1.5	0.6526	0.7540	0.6961
Performance 1.5	0.6615	0.6674	0.6482

Table 4: Experiment in changing the loss function

Table 4 describes the effect of changing the loss function. In the default model, the euclidean distance between the predicted location and the target location is used as loss. In first case, the square root of this distance was taken, in the second it was squared instead. as can be seen the squared loss degrades performance, while the square loss performed within run to run variance.

	default model	single Linear 256	extra linear layer (dim 128)	both previous
Training loss	0.3342	0.3531	0.3919	0.4000
Validation loss	0.4544	0.4852	0.4965	0.5087
Testing loss 0.5	0.2915	0.3229	0.2910	0.3017
Performance 0.5	0.6320	0.6117	0.6389	0.6300
Testing loss 1.0	0.4314	0.4629	0.4883	0.5052
Performance 1.0	0.6733	0.6599	0.6475	0.6413
Testing loss 1.5	0.6526	0.6936	0.7269	0.7408
Performance 1.5	0.6615	0.6499	0.6386	0.6353

Table 5: Experiment in changing the linear layers

Table 5 describes the effect of changing the linear layers in the model. As can be seen, changing the dimension of the single linear encoding layer to 256 degrades the performance at time 0.5 , while it improves the performance at later times. this is amplified by the addition of a layer to both the encoding and decoding parts

	1	5	10 (default model)	25	50	100
Training loss	0.3563	0.2069	0.3342	0.1723	0.1750	0.1769
Validation loss	0.4917	0.4111	0.4544	0.3526	0.3285	0.3429
Testing loss 0.5	0.2122	0.1976	0.2915	0.1499	0.1382	0.1437
Performance 0.5	0.5690	0.5895	0.6320	0.6205	0.6398	0.6382
Testing loss 1.0	0.4478	0.3503	0.4314	0.3162	0.2968	0.3032
Performance 1.0	0.6197	0.6491	0.6733	0.6570	0.6639	0.6590
Testing loss 1.5	0.8919	0.7192	0.6526	0.6610	0.6071	0.6397
Performance 1.5	0.6207	0.6434	0.6615	0.6516	0.6602	0.6537

Table 6: Experiment in changing batch size

Table 6 describes the effect of changing the batch size. As can be seen changing the batch size to 50 seemed to have a improvement in loss compared to the default model and an performance which is within the default variation (of 0.01), which means a batch size of 50 performs better. It also has the beneficial side effect that it runs a lot faster then the default model.

	0.25	0.50 (default model)	0.75
Training loss	0.3545	0.3342	0.2956
Validation loss	0.4449	0.4544	0.4804
Testing loss 0.5	0.3007	0.2915	0.2899
Performance 0.5	0.6266	0.6320	0.6319
Testing loss 1.0	0.4316	0.4314	0.4769
Performance 1.0	0.6742	0.6733	0.6514
Testing loss 1.5	0.6221	0.6526	0.7003
Performance 1.5	0.6719	0.6615	0.6470

Table 7: Experiment in changing the teacher force

Table 7 describes the effect of changing the teacher force. As can be seen a higher teacher force gives a more accurate prediction for the close time targets and a worse prediction for the time targets further away. A lower teacher force does the exact opposite. To balance between the two options the 0.5 teacher force would be best in predicting all 3 time targets.

Table 8 shows the effect of using our model compared to just using a simple linear model. As can be seen using our model has a huge improvement to just using a simple linear model.

	Default model	Simple linear model
Training loss	0.3342	20.6845
Validation loss	0.4544	19.9605
Testing loss 0.5	0.2915	16.4738
Performance 0.5	0.3680	0.1491
Testing loss 1.0	0.4314	20.3080
Performance 1.0	0.3267	0.2177
Testing loss 1.5	0.6526	25.0161
Performance 1.5	0.3385	0.2642

Table 8: Experiment in switching loss and performance measures

1.5 Conclusion

In conclusion, the final model consists of the change of a batch size to 50. The performance of the model is adequate at about 0.65. as mentioned before this 0.65 is the ratio of the euclidean distance between the prediction and target over the euclidean distance between the start start and the target. naturally any data regarding direction and exact proximity is not included here. To provide a better visualization of the predictions and their comparison to their target the image below is provided with a small sample of simulations. In these it can be see that the model is moderately adequate for exact positional predictions, it is quite informative when one merely aims to infer the direction of the movement over time. Additionally, the performance is poor at low velocities while at high velocities the model is performing quite accurately.

As far as potential improvements go, some suggestions can be made:

- The model currently includes the initial starting velocities as data-points, however since no intermediary velocities are included, back propagation over the complete output is impossible here, if this data is included, more accurate predictions could be made.
- As the Attraction between particles is partially dependant on the distance between them, the euclidean distance between the particles could be provided as an additional input for the model, potentially improving performance.
- at the cost of computational intensity, a hybrid approach could be used where the simulation is partly simulation where the forces between the particles are simulated at all time-points and are consequently provided as additional inputs when training the model.

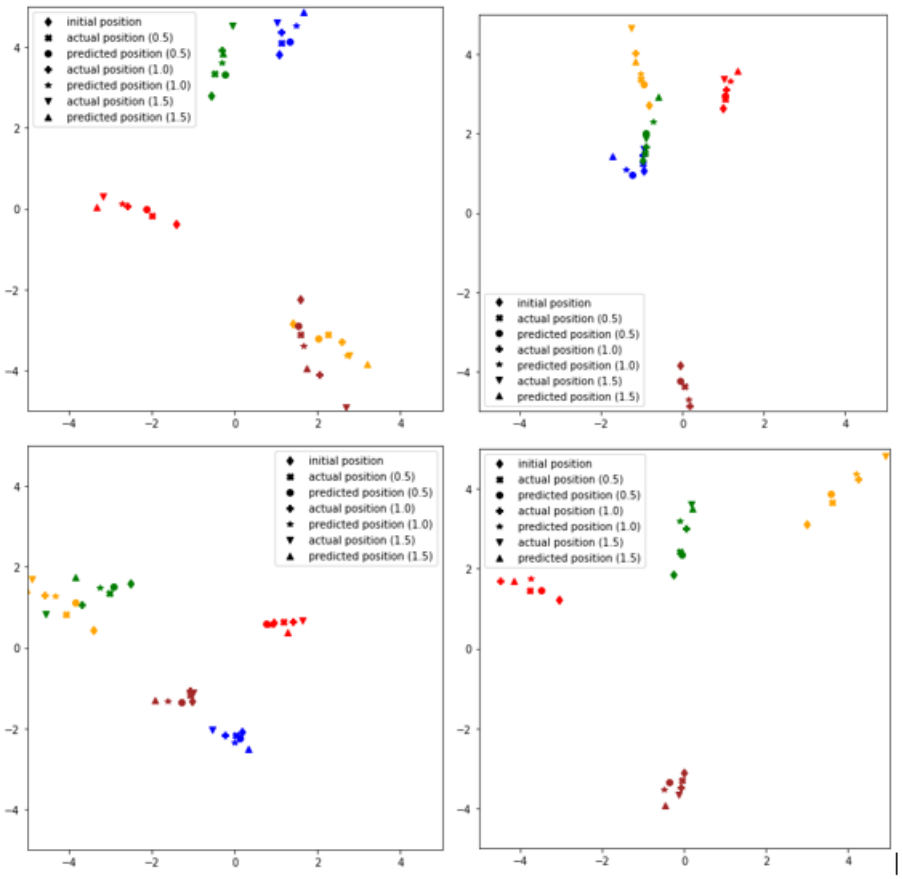

Figure 3: representations of predicted coordinates in 2d space

2 Task 3.1

2.1 Problem formulation

The goal of the problem is to predict the charges c_1, c_2 and c_3 of the particles with a fixed position given the trajectory of other particle moving in the 2D space which depend on the above-mentioned charges. The trajectory data is provided in form of a sequence of positions of each of the particles in the space with a fixed time step between those positions. Formally the data-set contains the initial trajectory data, with each data point being a trajectory $D = \{\mathbf{x}_1, \mathbf{x}_2 ... \mathbf{x}_n\}$, where $\mathbf{x}_t \in \mathbf{R}^2$ is a coordinates of particle in the 2D space at time t. The labels for each data-point are given in form $T = \{c_1, c_2, c_3\}$, where c_1, c_2 and c_3 are the corresponding charges.

The problem described is a supervised learning problem. Based on the given sequence, and the corresponding label we have to perform a sequence classification. Classification, is because even though the output data contains a couple data points, they are not sequential and independent of each-other so all can be viewed all can be represented as a single prediction.

To evaluate the model, we will simply use the L1 loss, which is simply the sum of euclidean distances between the predicted data and the actual data:

$$L = \sum_{i=0}^{N} |y - \hat{y}|$$

where y is the ground truth $\hat{y}$ is the predicted values and N size of the prediction (in our case 3 charges).

The above loss function, is also easily interpret-able, and could be used as measure of the model performance.

2.2 Model formulation

As mentioned above - the given task is a sequence classification problem. The recurrent architecture will be used for predictions. The need for recurrent architecture comes from the fact that the input data is sequential. One other reason for using a recurrent architecture is the varying input size which RNN's, in contrary to most other Neural-Network architectures, handle well.

In order to deal with the problem of the exploding and vanishing gradients during training, when doing back-propagation through time in a long sequences, the LSTM will be used instead of the simple RNN.

Although it's required to predict 3 values, they are not correlated, so it's undesirable for the network to try to find correlations between the charges. However, it's important that the network predicts them in the same order each time. It was therefore decided to output all the 3 charges from a single LSTM cell.

2.3 Implementation and training

Implementation of the model consists of the two parts: Encoder and Decoder.

Encoder is an LSTM network. It takes as an input, the simulated sequence (trajectory data), and produces a hidden state, which should hold a useful summary of that trajectory, that is then passed to the Decoder.

Decoder is a single step LSTM. It may contain multiple stacked cells, but only one step on the time axis (does a single prediction). It takes as an input the hidden state produced by the encoder for h_{t-1} and a dummy variable for the x_t (h_{t-1} and x_t are the corresponding inputs of LSTM cell). Output of the decoder is a vector with 3 values each corresponding to the required charge.

Hyper-parameters of the model are: depth (amount of stacked LSTM cells in a single iteration) and size of the hidden state. Those impact the complexity of the model, and therefore the complexity of patterns in the data that it can potentially learn.

2.4 Experiments and results

Initially, the model produced 3 successive outputs for each of the charges. Such architecture had 2 flows:

- Model attempts to find correlations between successive prediction of charges, which logically, are independent;
- Although not substantially in this case, the longer back-propagation through time causes the exploding and vanishing gradient.

So it was then decided to predict all 3 charges in one go, which yielded better results.

Because the model is relatively simple (only predicts 3 numerical values, based on a relatively short input sequence), we gravitated towards smaller values for both: size of the hidden dimensions and depth of the Neural-Network. Below you could see the results for different hyper-parameters. Intersection of the table shows the validation loss after 20 epochs given different hyper-parameters.

Hidden State Size \Depth	1	2
40	**0.113**	0.121
100	0.117	0.127
200	0.120	0.137

Table 9: Hyper-parameter tuning

Now that we have evaluated out model let us view the cases when it has the worst performance to figure out it's flaws (figure 4). Apparently from the figure, the worst performance is observed when the trajectory of the particle does not come close to the fixed particles, whose charges we want to predict.

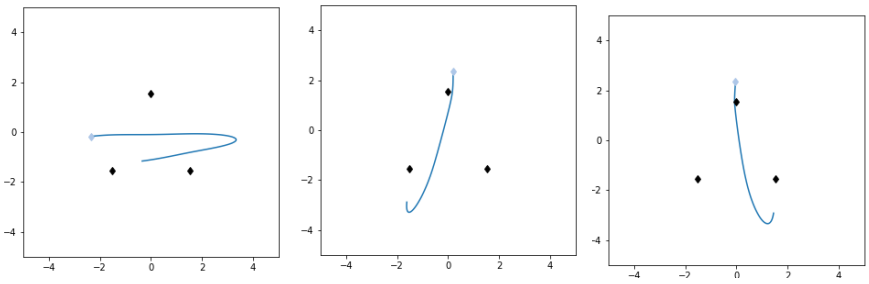

Figure 4: Cases with the worst performance

2.5 Conclusion

Although recurrent architectures could be used for sequence classification, depending on the specific task, with reasonable accuracy, for our specific task the below improvements/variants could also be considered.

In the end, one of the possible improvements is to try some alternative non-recurrent architecture. This idea came from the fact that, although the input data is sequential, we should not necessarily take advantage of it's auto-correlation. For instance the Decoder part only takes the hidden state of the Encoder as a meaningful input, the x_t input is just dummy, so if the input is just the summary of the trajectory, one of the options would be to try to solve the problem with simple multi-layer perceptron, however one should still deal with varying input size somehow.

The other possible improvement is the implementation of an attention mechanism. It could be particularly useful for the given task, since for particular charge of the output, we are more interested in a certain part of the trajectory (most likely the one close to the fixed particle). This brings the need to again decompose output into three separate outputs for each charge. However, we still don't want to introduce possible correlation between charges. So instead of feeding the output of the previous cell to the next cell, it is possible to only use attention as an input, which is a transformer network.

3 Task 3.2

3.1 Problem formulation

The goal of the task is, given the 100 ± 10 steps of initial simulation, predict the continuation of the 40 ± 20 steps. The input data is the same as in assignment 3.1, however the ground truth for output data contains the continuation of the particles trajectory: $D_c = \{\mathbf{x}_{n+1}, \mathbf{x}_{n+2} ... \mathbf{x}_{n+m}\}$, where n is the length of the input sequence in steps, m is the length of the output sequence in steps and $\mathbf{x}_t \in \mathbf{R}^2$ is the position of the particle in 2D space at time t.

The given problem is, again, a supervised learning problem. However, in contrary to the previous subtask (3.1), in subtask 3.2 the goal is to predict the sequence based on the other sequence, so it is a Sequence to Sequence problem.

In this case it is desirable to use the previous values for predicting the successive ones while it is clear and logical that the successive values are dependent on the previous ones (the next position of the particle is partially derived from the previous position).

The simple way to measure the loss of this model would be to look at the average distance between the predicted position of the particle and it's true position at that time. We use mean of the Pairwise-Distance function for it (because L1 loss takes the distance element-wise rather than pairwise).

$$L = \frac{1}{N} \sum_{t=0}^{N} |\mathbf{y}_t - \hat{\mathbf{y}}_t| \tag{1}$$

where N is the length of output sequence, y_t is the ground truth at time t (position of the particle), and $\hat{y}_t$ is the predicted position of the particle at time t.

This loss function is easily interpret-able and at the same time intuitively representative of the goal of finding the curves that are close together.

There is however one improved version of such loss. Rather than just looking at the distance between the predicted trajectories, we could look at the difference between their velocities. That way, we would incorporate some more logic of the simulation into the network, and help the network, by decreasing the loss function, if the two trajectories are relatively distant, but they're velocities match, which means, their shape is similar. We can approximate the velocity at time t by $v_t = \mathbf{y}_t - \mathbf{y}_{t-1}$, and therefore have the following formula for loss:

$$L = \frac{\sum_{t=0}^{N} |\mathbf{y}_t - \hat{\mathbf{y}}_t| + \sum_{i=0}^{N} |(\mathbf{y}_t - \mathbf{y}_{t-1}) - (\hat{\mathbf{y}}_t - \hat{\mathbf{y}}_{t-1})|}{2N} \tag{2}$$

This loss is less interpret-able and is not suitable for using as a performance metric of the Neural Network. However, supposedly, is more effective for training.

3.2 Model formulation

The model for this task is a Sequence to Sequence model, again consisting of Encoder and Decoder part. There's however an addition to the model (or rather an improvement), namely the attention mechanism (from the lecture notes).

Reason for the recurrent architecture is similar to the one in the above task - first of all the data is sequential, and it is desirable to learn how the successive values depend on the previous ones, we have varying input and output size. It is also possible that we would want our model to refer to some particular data in the input sequence, for making the prediction (for example looking at the trajectory of a particle close to a fixed point, to figure out how strong they attract).

For lowering the effect of the vanishing and the exploding gradients - the decision was to use the LSTM cells. Those problems are likely to happen in the given setup, since we have a long input and output sequence and calculate loss on each of the data points in the output sequence.

The model is also auto-regressive, meaning, it could take multiple previous outputs as an input. Such approach could help the model do better predictions, since we do not only provide the location of the previous particle, but also the data that implicitly contains the velocity and trajectory of a particle over some little previous period of time.

It is also an option to provide model the predictions of the charges done by the model from the assignment 3.1. This is also a possibility to reveal to the model some details of the simulation, rather than trying to force it to learn them.

Hyper-parameters of the model therefore include:

- depth (stacked LSTM)
- teacher force
- inclusion of the predicted charges
- the number of previous positions to feed to the next LSTM cell
- size of the hidden state
- size of the parameter space for the attention part

3.3 Implementation and training

The Encoder part of the model simply takes the input sequence and runs it through the sequence of LSTM cells (possibly stacked). So a single forward pass returns the hidden state after the last cell, the cell state and the stored outputs of the encoder (which will be used for attention).

In the decoder, each cell takes as an input the hidden state produced by the previous cell (or the encoder in case with the first cell) for input h_t, as well as the concatenation of: a sequence of N last positions (N is a hyper-parameter) in the simulation, context vector c_i (derivation in Figure 5) and optionally the charges predicted on the input sequence by the model from task 3.1.

Loss is calculated by formula (2), discussed in the problem formulation section.

There is a subtlety to applying the teacher force when we input the sequence of the previous values longer than one back into the network. Because both alternating between the sequence being fully taken from ground truth or previous predictions, and alternating between individual values in the sequence being taken from the ground truth or the predicted values, does seem to introduce very messy data for the input of the Neural Network, it's better to completely switch on or off the teacher force for models that takes sequences of previous values larger than 1.

The decoder too uses a GRU with hidden state s_k where s_0 is initialized as

$$s_0 = \tanh(W_s h_1). \tag{5.47}$$

The difference is that at every time step i, the decoding GRU is presented with a (different) context vector c_k. This context vector is computed as follows. First we compute the logits for the weights of the attention vector

$$e_{ij} = v_a^\top \tanh(W_a s_{i-1} + U_a h_j) \tag{5.48}$$

from which we compute the attention vector as

$$\alpha_k = \text{softmax}(e_k). \tag{5.49}$$

This attention vector is then used to compute the context vector as

$$c_i = \sum_j \alpha_{ij} h_j. \tag{5.50}$$

We can then feed the context vector, together with the hidden state and the previous output to the decoding GRU to get the next hidden state, and thu the next context vector

$$s_i = g(y_{i-1}, s_{i-1}, c_i). \tag{5.51}$$

To get the output, we can now feed the previous output, the hidden state, and the context vector to an MLP with softmax output:[17]

$$P(y_i \mid x_1, \ldots, x_k, y_1, \ldots, y_{i-1}) = \text{softmax}(W_o s_i + C_o c_i + U_o E y_{i-1}) \tag{5.52}$$

where E is an embedding matrix. This architecture is shown in Figure 5.25 for the different task of translating sentences from English to Dutch.

Figure 5: representations of predicted coordinates in 2d space

3.4 Experiments and results

There have been a lot of experiments performed with this model because of the big list of proposed improvements and hence a lot of hyper-parameters that the model has. Below we will view how changing different hyper-parameters changes the training process. The training process will be represented by two plotted lines: the orange line showing the loss on training data (formula (2)) and the blue one showing the performance of the model on the validation set (formula (1) - average distance between the predicted and the ground truth trajectories). Each successive model contains all the features of previous one, with the noted addition.

This is a simple default model (without attention, non auto-regressive, without feeding in the charges, with teacher force disabled) with depth of 3:

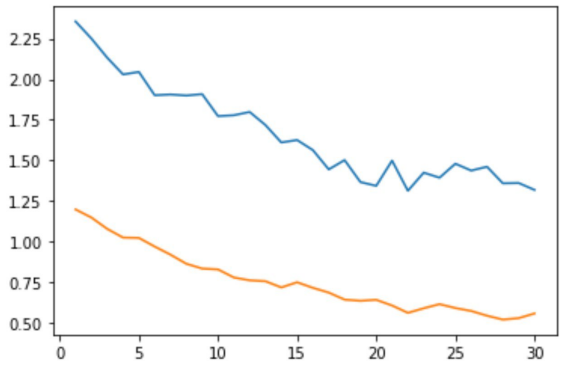

Figure 6: Model with stacked depth of 3

For this one we included the predicted charges as the model's input:

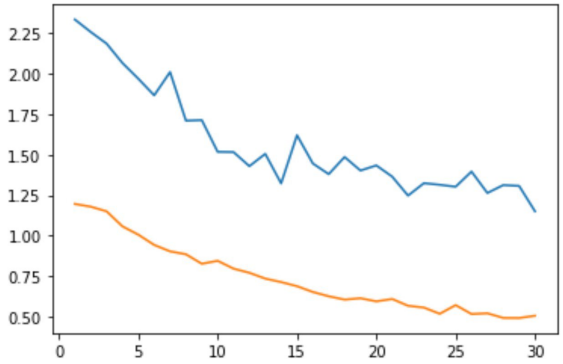

Figure 7: model with inputting charges

The experiments have shown that training with the teacher force on does not bring good results, but training without the teacher force does bring a little improvement:

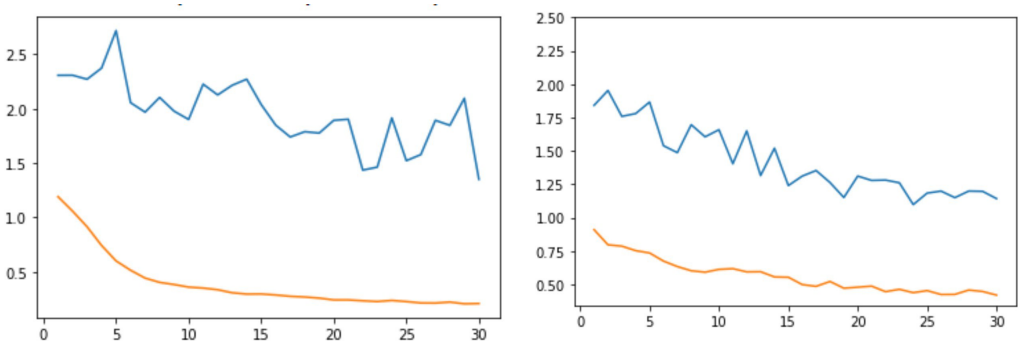

Figure 8: training with and then without the teacher force

Making the model auto-regressive (looking at 5 previous values to make the prediction), smooths out the loss, but does not bring substantial improvement. However, it could be trained for longer, because the loss function still seems to decrease even at the last epochs:

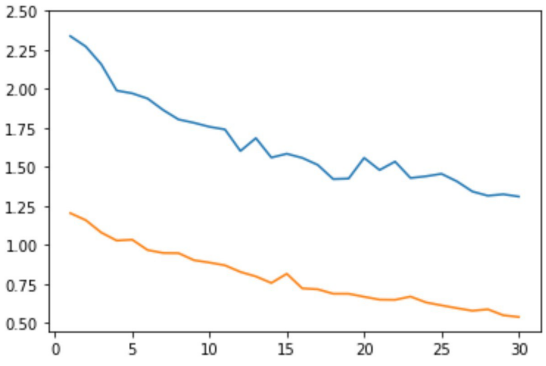

Figure 9: auto-regressive model

Finally, this model implements the attention mechanism:

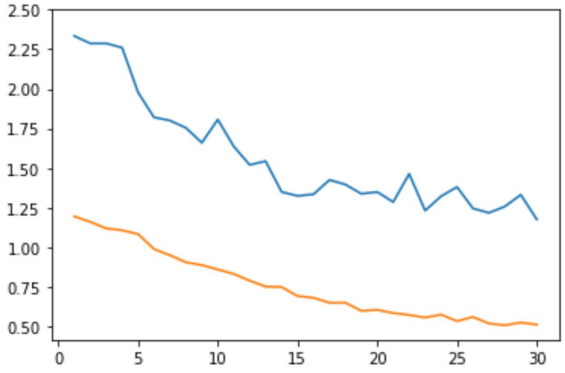

Figure 10: model with attention

Below you can see the example of the model continuing the simulation.

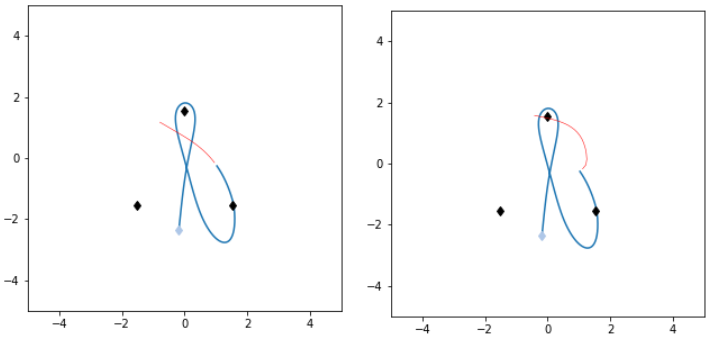

Figure 11: (Visualizing results) ground truth and the predicted simulation

Finally, while the extrapolation gives the average distance of 3.3 units on the test set, our model (the latest one with attention) has the average distance of 1.17 (figure 12).

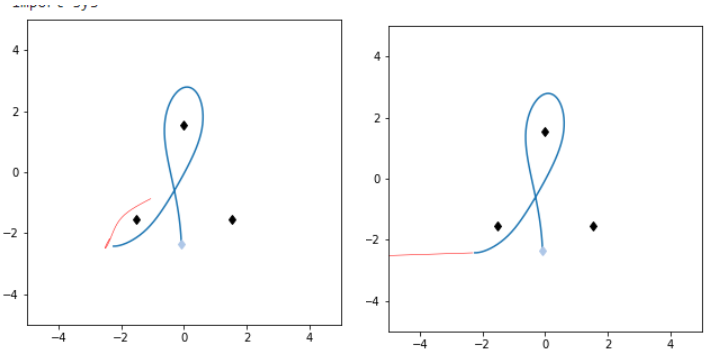

Figure 12: Prediction vs extrapolation

3.5 Conclusion

Although using machine learning to more efficiently perform the simulation is an attractive idea, the recurrent architecture presented above, even with all the implemented advancements does not do a very good job. It is probably due to the fact, that although the Neural Network could try to generalize the task, the actual simulation process, that although could be computationally expensive, provides an almost optimal solution with negligible numerical drift caused by the used integration scheme.