2AMM10 2021-2022 Assignment 2 & 3: Group 22

Ruichen Hu(1674544) Simin Sun(1692933) Siyue Chen(1657402)

June 20, 2022

Contents

1 Problem Formulation

The traditional analytical and problem-solving methods for physical and engineering problems usually involve sophisticated computational techniques, which can be time-consuming and resource-intensive. An alternative to these complicated approaches is to build machine learning models that simulate the physical phenomenon and generate the predicted computation results based on the historical data.

This report focuses on using deep learning models to simulate dynamics of the charged particles. The models work like black-boxes such that we do not have to consider how the results are computed physically.

1.1 Task (2)

This task aims at predicting the position of 5 charged particles in a two-dimensional Euclidean space. The particles are under mutual interaction with others and move along the time. This means the proposed method should simulate the kinematics equation $\mathbf{v}_i^t = \mathbf{v}_i^{t-1} + \mathbf{a}_i^t \cdot \Delta t$ and $\mathbf{x}_i^t = \mathbf{x}_i^{t-\Delta t} + \mathbf{v}_i^t \cdot \Delta t$, where $\mathbf{v} \in \mathbb{R}^2$ denote the particle velocities, $\mathbf{a} \in \mathbb{R}^2$ denote the particle accelerations, $\mathbf{x} \in \mathbb{R}^2$ denote the positions, $t \in \{0, 0.5, 1.0, 1.5\}$ denotes the time in seconds and $i \in \{1, 2, 3, 4, 5\}$ denotes the index of the particle.

More formally, given five particles with their constant charges $c_i \in \{-1, 1\}$, velocities v_i^0 and positions $\mathbf{x}_i^0$ at initial time point $t = 0$, the model needs to simulate the behavior of each particle in the subsequent 1.5 seconds by predicting their positions $\mathbf{x}_i^t$ for every 0.5 seconds.

The underlying intuition is to figure out whether the deep learning method is applicable for this specific problem. Specifically, whether the predicted positions are correlated with the true values and how reliable the predictions can be. As the movement of the particles is time-dependent, and each previous state of the particles determines a next position, we think this task can be solved by recurrent neural network(RNN). RNN has a recurrent module that learns the pattern of the last timestep and forward this pattern to the next round, which corresponds to our task that use the last states to predict the next positions. In terms of measuring the adaptability of the model to dynamics simulation, the distance between the predicted positions and the true positions would be a desirable metric. The lower the distance is, the better the model performs.

1.2 Task (3.1)

In task 3.1, three fixed charges in a triangle configuration are set in the plane. The simulations sampled with $\Delta t = 0.1$, that is 100 ± 10 steps are given as particle p_1's trajectory. We need to predict the values of the charges c_2, c_3, c_4 by observing the trajectory of p_1 which is shot into the plane. The value of p_1 is fixed to 1, and the values of the charges c_2, c_3, c_4 are sampled uniformly from the interval $[1, 0]$.

The input data of the model would be the trajectory of a particle, which is a series of 2-dimensional coordinates in the plane. At each time point, the location is determined by the past locations and the intensities of the charges.

We came up with two solutions: one is flatting the positions information and using corresponding charges as labels. In this case, we ignore the time dimension and feed all the data in Convolutional neural network(CNN). The other solution is treating it as a time series, thus we consider to solve the problem by recurrent neural network(RNN). RNN has a recurrent module that learns the pattern of the last timestep and forward this pattern to the next round, which corresponds to our task that use the last states to improve our prediction. In terms of measuring the adaptability of the model to dynamics simulation, the error between the predicted charges and the true charges would be a desirable metric. The lower the distance is, the better the model performs. This is a regression task, we therefore decided use mean-square error(MSE).

1.3 Task (3.2)

The given data in Task 3.2 and Task 3.1 are both trajectory data. The distinction is that in Task 3.2, we must forecast the trajectories in the continuous time frames while in Task 3.1, we must anticipate the charges of three particles.

Task 3.2 investigates if we can train a model to appropriately continue a simulation after it has learned its dynamics.We must forecast the positions of the particle P_1 during the next 40 ± 20 steps based on the given data positions information for 100 ± 10 steps. 150 simulations are provided for training, 100 simulations are given for validation, and 100 simulations are offered for testing.

This is a typical time series problem, and we can forecast the upcoming positions by using the information from the previous positions. The newly predicted position can then be included as known information to recursively predict the following 40 ± 20 steps. This serves as a further reminder to use recurrent neural networks (RNN). The evaluation is similar to Task 2, and a desirable metric would be the distance between the anticipated positions and the true positions. The model' performances are better when the distance are closer.

2 Model Formulation

2.1 Task (2)

As mentioned in the preceding part, RNN is a suitable solution for the given problem. Specifically, we decide to use long short-term memory(LSTM) neural network to capture the pattern in particles' movement and generate a prediction on positions. The motivation of using LSTM is due to its good combination of long- and short-term memory. The ordinary RNN usually only focus on the most recent states such that the past information is lost when there is a long time sequence. Besides, it suffers from vanishing or exploding gradient as the sequence length increases. In comparison, LSTM introduces gate mechanism that regulates the update of long-term memory and the storage of short-term memory such that the gradient-related issues are alleviated and the states produced early are retained selectively. In the context of task **(2)**, the impacts of the initial velocities and positions can be preserved and propagated through the whole sequence when LSTM is applied.

Task **(2)** comprises of a training phase and an evaluation phase. For both phases, we use the same architecture, however, with slight difference in how data is processed.

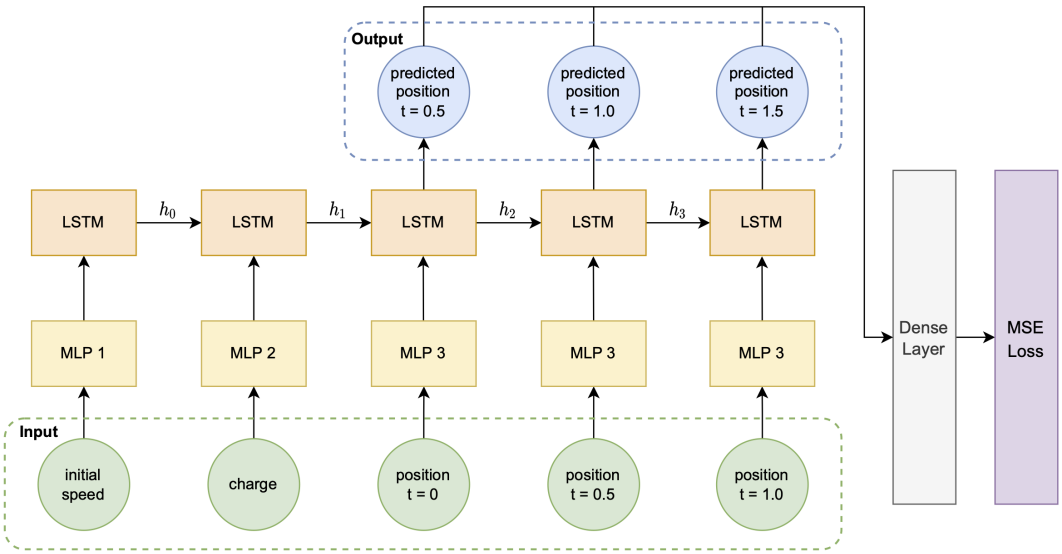


Figure 1: Architecture of the LSTM neural network (h_i denotes the hidden states between each timestep)

As Figure 1 shown above, the first two inputs are initial spped and charge, because we consider these two items representing the property of the particles, and all the subsequent behaviors of the particles are based on them. After forwarding them into the model, we use multilayer perceptrons (MLPs) to capture the their patterns , and then feed the embeddings into LSTM module to initialize the parameters. In this way, the resulted parameters as well as hidden states h_0 and h_1 convey the information of initial states to each subsequent input.

The motivation of using MLPs is that we believe the LSTM block can learn the patterns better with the embeddings, as the characteristics of the original input data may not be easy to capture. Similarly, we also preprocess the position values at $t = 0$, $t = 0.5$ and $t = 1.0$ with a MLP to get a representative embedding before feeding into LSTM block.

As for the LSTM block, it is shared for all inputs throughout the whole training and evaluation phase. There are 5 timesteps for each iteration, the last three of which aim at predicting the positions after 5 seconds of the corresponding input positions and giving the outputs. The dimension of the output is the same as that of input such that the model is equivariant to transformation. To further condense the features of the output, we design a dense layer block attached to the last layer of LSTM. The dense layers also transform the dimension of the output tensor to the same dimension as the true values.

As this is a regression problem that we need to predict the positions as close as to the true values, we calculate mean squared error (MSE) to figure out how the model performs. Assume the dimension of the final dense layer is k. Let $\mathbf{P} \in \mathbb{R}^{n \times m}$ denote n sequences of m predicted positions, and $\hat{\mathbf{P}} \in \mathbb{R}^{n \times m}$ denote n sequences of m true positions, then the MSE loss is as follows:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (P_{i\cdot} - \hat{P}_{i\cdot})^2 \tag{1}$$

From the definition is it obvious that the close the MSE value to zero is, the better the predictions are.

By far, we interpret how the model is formulated for training to simulate particle dynamics. With respect to evaluation, we use the same architecture but with different operations. The concrete description is given in Section 3.1.4

2.2 Task (3.1)

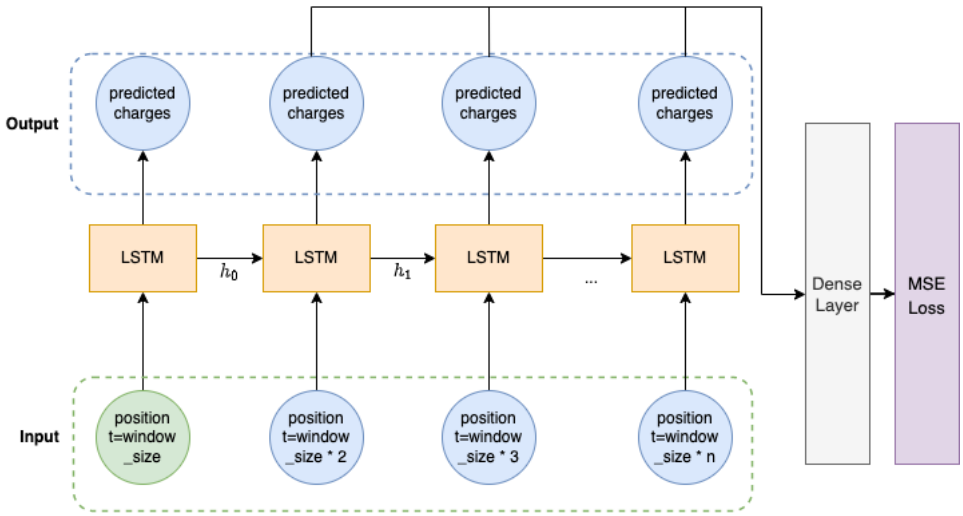


Figure 2: Architecture of the LSTM neural network

In Section 2.1's opening paragraph, it was explained why we employ RNN in general and LSTM in particular. The results of utilizing MLP in task 3.1 are not as good as anticipated, despite the advantages we highlighted in Section 2.1. The reason could be that the input data for task 2 is less extensive and complex. Figure 2 highlights those architectural alterations compared to task 2.

The LSTM block, on the other hand, is utilized for all inputs over the whole training and evaluation phase. Each iteration consists of n (window_size in our model implementation) timesteps and predicts the values of three negatively charged particles corresponding to the input positions. We create a dense layer block and attach it to the last layer of the LSTM to further compress the characteristics of the output charge values. The output tensor's dimension is also changed by the dense layers to match the dimension of the true values. MSE, which we apply and describe in task 2, serves as the loss function.

2.3 Task (3.2)

As Figure 3 shown, the architecture of task 3.2 is quite similar to the design we implemented in task 2. The motivation of using LSTM, MLP and MSE have been discussed in Section 2.1. The distinction is that we predict the location of time n+1 by looking at the trajectory during the first n timestamps.

The LSTM block is utilized for all inputs over the whole training and evaluation phase. For each iteration, there are n (or window_size in our implementation) timesteps with the goal of forecasting the position following n timestamps of the associated input locations and providing the outputs. In order for the model to be transformation-equivariant, the output and input dimensions must match. We create a dense layer block that is connected to the last layer of the LSTM to further compress the output's features. The output tensor's dimension is also changed by the dense layers to match the scale of the true values. Only the last vector of output is our final prediction for next position.

We feed the model the known trajectories in order to obtain a single prediction. We integrate this newly anticipated position as known information and give it to the model to make additional prediction in order to obtain continuous predictions during the testing parse. To plot the continuous paths of k timestamps and gather sufficient forecasts, we repeat this process k times.

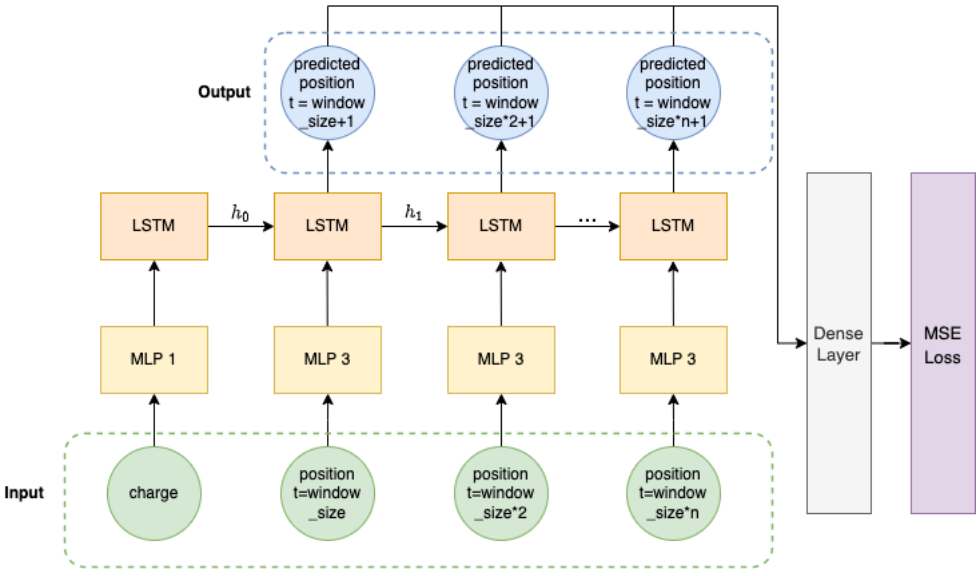


Figure 3: Architecture of the LSTM neural network

3 Implementation

3.1 Task (2)

3.1.1 Data Handling

In the original dataset, each velocity and position is represented by two coordinate values. Thus, for the five particles, they are represented by a matrix $M \in \mathbb{R}^{2\times5}$. We flatten this matrix to a vector $V \in \mathbb{R}^{1\times10}$ such that it can be processed by the MLP layers. The velocity set, along with charge set and position set, is packed for constructing a dataloader. For each dataset among training, valiadation and test set, we build a dataloader. Besides, as we use Mini-batch method to train the model, all dataset is divided into batches with the size 100 while generating the dataloader. The changes of dimension with different operations are revealed in Table 1.

		Original	Processed	Batched
Training data	velocity	(10000, 1, 2, 5)	(10000, 10)	(100, 10)×100
	charge	(10000, 5, 1)	(10000, 5)	(100, 5)×100
	position	(10000, 4, 2, 5)	(10000, 4, 10)	(100, 4, 10)×100
Validation data	velocity	(2000, 1, 2, 5)	(2000, 10)	(20, 10)×100
	charge	(2000, 5, 1)	(2000, 5)	(20, 5)×100
	position	(2000, 4, 2, 5)	(2000, 4, 10)	(20, 4, 10)×100
Test data	velocity	(2000, 1, 2, 5)	(2000, 10)	(20, 10)×100
	charge	(2000, 5, 1)	(2000, 5)	(20, 5)×100
	position	(2000, 4, 2, 5)	(2000, 4, 10)	(20, 4, 10)×100

Table 1: Data dimension of the original dataset, processed dataset and batched dataset

Furthermore, as it is required to explore the relationship between different training data size with the model performance, we generate six sets by taking the first 100, 1000, 2500, 5000, 7500, 10000 data samples from the original training set. For simplicity, we call them TRAIN-100, TRAIN-1000, TRAIN-2500, TRAIN-5000, TRAIN-7500, TRAIN-10000, respectively. Each one of these training sets is loaded into a dataloader with the batch size 100.

3.1.2 Model Implementation

Figure 4 shows the details of the implemented model. There is a MLP constructed for each category of input. Here, all the MLPs have the same structure, namely one linear layer coupled with a ReLU layer. We purposely construct a very shallow and simple structure for MLP, because the number of data samples are fairly small and a complicated MLP block can result in overfitting. Due to the same reason, the subsequent LSTM module has only one hidden layer with 16 neurons. This LSTM is shared for results of all three MLPs. Its output is further forwarded into a dense layer block with two linear layer and one ReLU layer. The last linear layer maps the last dimension of the output tensor to 10, which corresponds to the dimension of the true positions. After that, the MES loss is computed to compare the difference between predictions and true values.

In the model, we also designed a dropout layer attached to the linear layer in the dense layer block to reduce overfitting. However, due to performance issues we removed it(see Section 4.1.3).

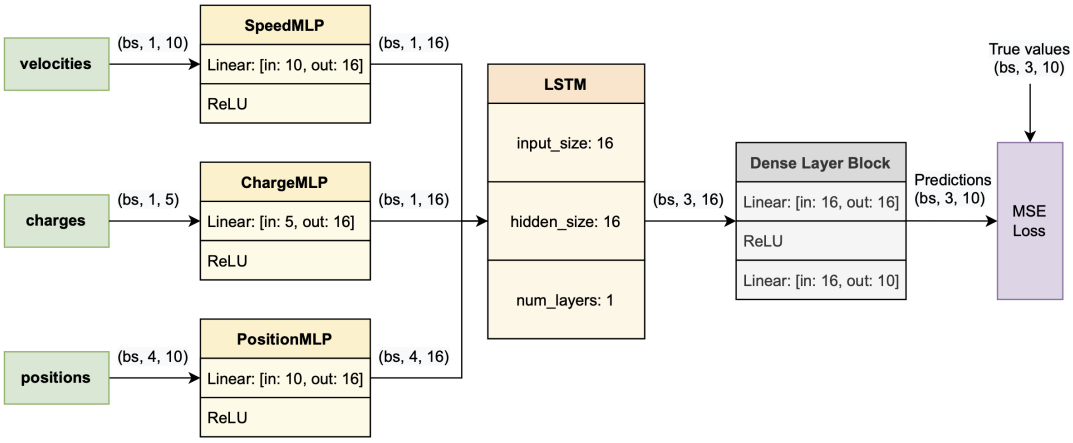


Figure 4: Implementation of the model(*bs* denotes batch size)

3.1.3 Training

As we have built 6 dataloaders with different training data size, we generate for each dataloader a model sequentially. For each dataloader, we adapt mini-batch strategy with the batch size 100 to train the model. During each training, we feed velocities, charges and positions at 4 time points of 5 particles batch by batch. The velocities and charges data is unsqueezed to add a dummy layer first such that the dimension match the input size of MLP and LSTM. The annotations above the arrows in Figure 4 denotes the dimension of the data in the format of (batch size, timestep, position). The embeddings from the MLPs are concatenated and fed into LSTM, in which the embeddings are processed in a sequential order. Followed by LSTM is the dense layer block that further learn the patterns and adjust the output dimensions. For each batch, we compute a MSE loss by comparing the output representations of the dense layer with the corresponding true values. The final loss is the average value of all batches.

In terms of optimization, we use the Adam optimizer, because in practical this optimizer contributes to faster convergence and requires less number of epochs than other optimizer like SGD.

The validation data is fed into the model after every epoch of training. The model is switched to *model.eval()* to avoid certain training-specific measures affects the validation. Again, we get the resulted prediction and compute the MSE loss with validation data. The validation loss is our criterion to fine-tune the model. For both training and validation phase, we compare the losses with the loss derived by a linear baseline model, whose predictions conform to the formula $\mathbf{x}_i^t = \mathbf{x}_i^0 + \mathbf{v}_i^t \cdot t$.

Since we find the data size is small and the patterns are easy for a LSTM network to learn, the hyperparameters including network depth and number of neurons are kept small to restrain the tendency toward overfitting. Other regularization methods are not adopted because the model performance deteriorates dramatically after taken these measures(see Section 4.1.3).

3.1.4 Evaluation

To evaluate the model, we use the test set with the model mode turning to *model.eval()*. Here, the input includes initial velocity, charge and the initial positions at $t = 0$. The positions at other time points serve not as input, instead, they are only used as true values when computing loss. That means, we needs to operate manually on LSTM module. We first feed initial speed and charges into MLP and LSTM to get a hidden state and cell state. Then these two tensors are together with the MLP embedding of the position at $t = 0$ forwarded into LSTM and dense layers to get the prediction at $t = 0.5$. The predictions are

Table 2: Task 3.1 Data Reconstruction: window_size = 40

		Original	Reconstructed
Training	charge	800*(3,)	(800, 50*, 3)
	position	800*(100 + 10, 2)	(800, 50, 80)**
Validation	charge	100*(3,)	(100, 50, 3)
	position	100*(100 + 10, 2)	(100, 50, 80)
Testing	charge	100*(3,)	(100, 50, 3)
	position	100*(100 + 10, 2)	(100, 50, 80)

* $50 = input\_data\_length - window\_size = 90 - 40$
** $80 = 2 \times window\_size = 2 \times 40$

treated as the input tensors for next timestep to generate the next predictions at $t = 1.0$. We repeat the process until get 3 predictions and then compute the MSE loss. Again we compare the losses of the trained model with the test loss of baseline model.

3.2 Task (3.1)

3.2.1 Data Handling

As the original training dataset is the positions of positively charged particle p_1 during a simulation up to $t = 10 \pm 1$, the dimensions of the input data is range from 90 to 110. We want the length of the input data are of same size, therefore, we slice the data and only keep the data from timestamp 1 to 90.

And we want to use a short trajectory to predict the value instead of a single time point. Therefore, we reformat the data. The original data is 800 simulations and the size of each sub-elements is $(100 + 10, 2)$, which is the positions information over 10 ± 1 seconds. The new dataset is of size 800*50*80, which are still 800 simulations. Each sub-element is of size 50*80, means 50 time slots (= 90− window_size), each of them contains 40 window_size $*(x, y) = 80$ positions. Window_size is a customize varible which means the length of trajectory we want to use, and 40 means we use 4 seconds trajectory to predict possible charges and we will get 50 predictions. Only the last one will be our final prediction. In Figure 5, when there are $90 * (x, y)$ positions available and the window size is 5, we demonstrate how the model functions with a sliding window. Our final prediction for three particles (negative values for our data) in this situation is [0.3, 0.5, 0.7]. When the window_size = 1, we get the result of normal implementation and the prediction is worsen than window_size = 40.

The data reconstructed processes are summarized in Table 2:

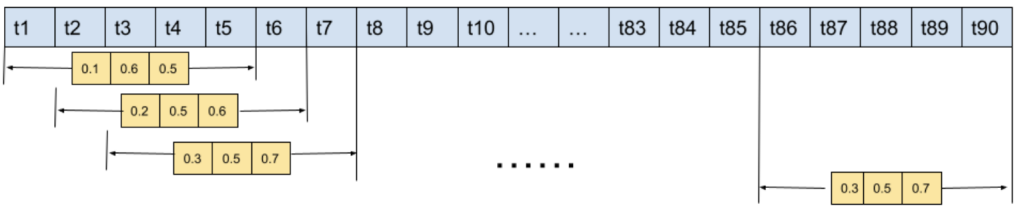


Figure 5: Showcase for sliding window and charge prediction: window_size = 5

3.2.2 Model Implementation

The implemented model's specifics are shown in Figure 6. The following LSTM module only has one hidden layer with 32 neurons because the quantity of data samples is rather modest and a complex structure could lead to overfitting. A dense layer block containing two linear layers, a dropout layer, and a Sigmoid layer is where its output is then further

forwarded. The last linear layer converts the final output tensor dimension from 32 to 3, which corresponds to the number of particles. The difference between predictions and true values is then compared using the MES loss computation.

We intentionally add the dropout layer since, in the absence of this layer, the expected values will be clustered in a limited range of around 0.5. According to our experiments, we believe the model could be easily stuck in the local minimizer and a dropout layer could be useful by adding some randomization and lessening overfitting.

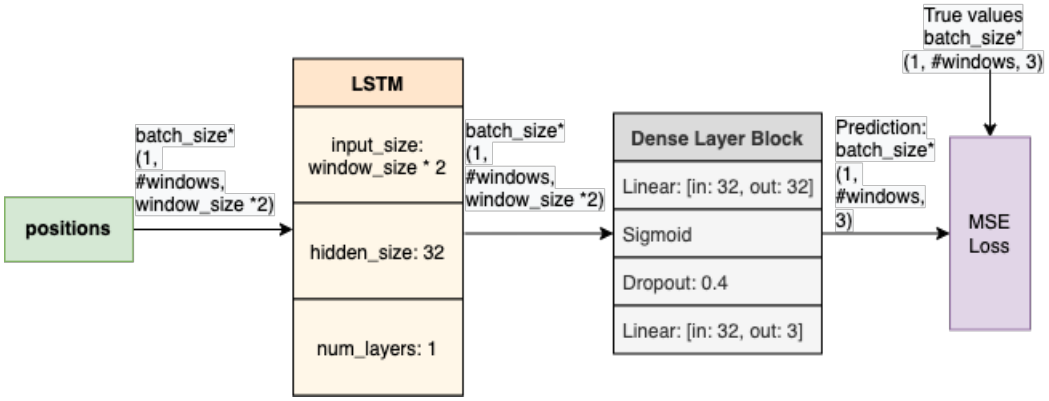


Figure 6: Task 3.1: Implementation of the model

3.2.3 Training

We use dataloader with batch size = 50 and feed the reconstructed data to train the model. In Figure 6, this input data are denoted as batch_size(1, windows, window_size*2). The input data will be fed into LSTM, in which the positions' information are processed in a sequential order. Followed by LSTM is the dense layer block that further learn the patterns and adjust the output dimensions. For each batch, we compute a MSE loss by comparing the output representations of the dense layer with the corresponding true values. The final loss is the average value of all batches.

Layers Hyperparameters: As shown in Figure 6. The LSTM module only has one hidden layer with 32 neurons because the quantity of data samples is rather modest and a complex structure could lead to overfitting. A dense layer block containing two linear layers, a dropout layer, and a Sigmoid layer is where its output is then further forwarded. The last linear layer converts the final output tensor dimension from 32 to 3, which corresponds to the number of particles. Since we find the data size is small and the patterns are easy for a LSTM network to learn, the hyperparameters including network depth and number of neurons are kept small to restrain the tendency toward overfitting.

Optimization: we use the Adam optimizer, because in practical this optimizer contributes to faster convergence and requires less number of epochs than other optimizer like SGD.

Activation function: Sigmoid was first chosen(as shown in Equation 2) as the charges should be value within [-1, 0]. Sigmoid takes a real value as input and outputs another value between 0 and 1. Then we tried ReLu, which gave us better performance in terms of loss. ReLu is hence our preferred option.

$$sigmoid : \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

$$Relu(z) = \begin{cases} 0 & if : z < 0 \\ z & otherwise \end{cases} \tag{3}$$

Loss function: As this is a regression problem, we need to predict the charge values as accurate as possible. Therefore we choose to use mean squared error (MSE) to figure out

how the model performs. Assume the dimension of the final dense layer is k. Let $\mathbf{P} \in \mathbb{R}^{n \times m}$ denote n sequences of m predicted values, and $\hat{\mathbf{P}} \in \mathbb{R}^{n \times m}$ denote n sequences of m true positions, then the MSE loss shown as Equation 1.

Validation function: The validation data is fed into the model after every epoch of training. The model is switched to *model.eval()* to avoid certain training-specific measures affects the validation. Again, we get the resulted prediction and compute the MSE loss with validation data. The validation loss is our criterion to fine-tune the model. For both training and validation phase, we compare the losses with the loss derived by a linear baseline model, whose predictions conform to the formula $\mathbf{x}_i^t = \mathbf{x}_i^0 + \mathbf{v}_i^t \cdot t$.

3.3 Task (3.2)

3.3.1 Data Handling

As the original training dataset is the positions of positively charged particle p1 during a simulation up to $t = 10 \pm 1$, the dimensions of the input data is range from 90 to 110. The continous trajectory is positions within 4 ± 2 seconds. Our training data should be positions information during t timestamps and the label should be the locations of t+1. In this case, we also need to reconstruct data. First, we concatenate the data *simulation_train_task32* and *simulation_continued_train*. The outcome should be data of 14 ± 3 seconds, therefore the minimum data size should be 110. As we want the length of the input data are of same size, only data of last 110 timestamps have been kept.

We want to use a short trajectory to predict the value instead of a single time point, we reformat the data based on the window size as depicted in Figure 7. The original data is 150 simulations and the size of each sub-elements is $(100 + 10, 2)$, which is the positions information over 10 ± 1 seconds. The new dataset is of size 150*80*60, which are still 150 simulations. Each sub-element is of size 80*60, means 80 time slots $(= 110 - window\_size)$, each of them contains 30 window_size $*(x, y) = 60$ positions. Window_size is a customize varible which means the length of trajectory we want to use, and 30 means we use 3 seconds trajectory to predict possible next location and we will get 80 predictions. Only the last one will be our final prediction, which denoted as (X_n, Y_n) in Figure 7. The charges of three particles have been reformatted according to the windows size. If the window size = 30, then we transform the original data of (150, 3,) to the size of (150, 80, 3) by duplicating the charges value for 80 times.

The data reconstructed processes are summarized in Table 3:

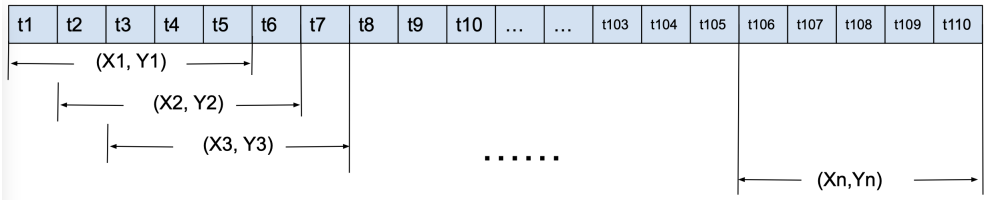


Figure 7: Showcase for sliding window and position prediction: window_size = 5

3.3.2 Model Implementation

Figure 8 shows the details of the implemented model. There is a MLP constructed for each category of input. Here, all the MLPs have the same structure, namely one linear layer coupled with a ReLU layer. We purposely construct a very shallow and simple structure for MLP, because the number of data samples are fairly small and a complicated MLP block can result in overfitting. Due to the same reason, the subsequent LSTM module has only one hidden layer with 32 neurons. This LSTM is shared for results of all three MLPs. Its output is further forwarded into a dense layer block with two linear layer and one ReLU layer. The last linear layer maps the last dimension of the output tensor to 2, which corresponds to

Table 3: Task 3.2 Data Reconstruction : window_size = 30

		Original	Reconstructed
Training	charge	$150 \times (3,)$	$(150,\ 80^{*},\ 3)$
	position	$150 \times (100 \pm 10, 2)$	$(150,\ 80,\ 60^{**})$
	position_continued	$150 \times (40 \pm 20, 2)$	
Validation	charge	$100 \times (3,)$	$(100, 80, 3)$
	position	$100 \times (100 \pm 10, 2)$	$(100, 80, 60)$
	position_continued	$100 \times (40 \pm 20, 2)$	
Testing	charge	$100 \times (3,)$	$(100, 80, 3)$
	position	$100 \times (100 \pm 10, 2)$	$(100,\ 60,^{***}60)$
	position_continued	$100 \times (40 \pm 20, 2)$	$100 \times (40 \pm 20, 2)$

* $80 = input\_data\_length - window\_size = 110 - 30$
** $60 = 2 \times window\_size = 2 \times 30$
*** for the test data, the input data length is 90, therefore:$60 = input\_data\_length - window\_size = 90 - 30$

the dimension of the true positions. After that, the MES loss is computed to compare the difference between predictions and true values.

In the model, we also designed a dropout layer attached to the linear layer in the dense layer block to reduce overfitting. However, due to performance was not as good as expected.

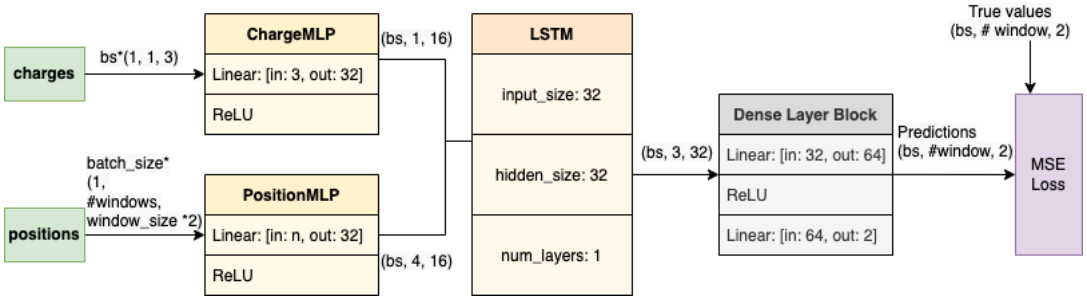


Figure 8: Task 3.2: Implementation of the model

3.3.3 Training

We use dataloader with batch size = 50 and feed the reconstructed data to train the model. In Figure 8, this input positions data are denoted as batch_size(1, windows, window_size*2). The embeddings from the MLPs are concatenated and fed into LSTM, in which the embeddings are processed in a sequential order. Followed by LSTM is the dense layer block that further learn the patterns and adjust the output dimensions. For each batch, we compute a MSE loss by comparing the output representations of the dense layer with the corresponding true values. The final loss is the average value of all batches. The general hyperparameters and settings can referred to Section 3.2.3.

4 Experiments and Results

4.1 Task (2)

4.1.1 Training and Validation

The loss curves for training and validation with different training data size is illustrated in Figure 9. For all models it take about 20 epochs to arrive at a loss plateau of about 0.01 both in training and validation. When the number of epochs grows to 50, the loss values reach the range from 0.005 to 0.002. This means the implemented model is effective

in solving the problem. Besides, the model trained with TRAIN-2500 witnesses the largest decreasing speed, while the losses of TRAIN-100 and TRAIN-5000 model decrease relatively slower than the others. This indicates the tuned hyperparameters may be more suitable for the model trained with TRAIN-2500. As for validation loss, it shows the same tendency and order of models, which means the model performs stably during validation phase.

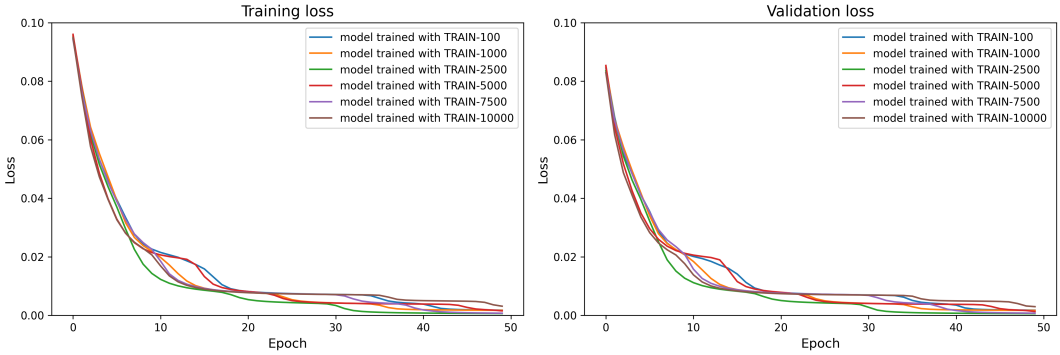


Figure 9: Loss curve for training and validation of different models

Training set	Training loss	Validation loss
TRAIN-100	0.0017606842957437038	0.0017555757388472556
TRAIN-1000	0.0017120633214712142	0.001677418254315853
TRAIN-2500	0.0006166297499090433	0.0005825793016701937
TRAIN-5000	0.001520061421394348	0.0013121907263994217
TRAIN-7500	0.0007310642432421445	0.0007426326107233763
TRAIN-10000	0.003118350304663181	0.0029650974348187447
Baseline	0.23693759987751642	0.23750895261764526

Table 4: Training and validation loss of the models trained with different training sets and baseline model at the 50th epoch

When comparing the training and validation loss at the 50th epoch for each training set in Table 4, we can find that in generally, all the models perform much better than using the linear baseline to make predictions. Besides, except Train-1000 and Baseline, the training losses are higher than the validation losses, although the difference is relatively slight. An explanation is that the some training samples contain noises such that some models in the training phase suffer from more losses. When it comes to validation, the validation set contains less data than most training dataset, thus the problem resulted from noises has less impacts on the results. As for the model trained with TRAIN-100, the number of training samples is much lower than the validation set. However, since we set the batch size to 100 for all training sets, the training with TRAIN-100 is actually updated with stochastic gradient descent, which is more likely to be affected by noises as every time there is only one data sample being processed.

During the training, we also figure out that the order of model performance is not reproducible. When the program is run for multiple times, the performance of the models varies. Our current best model, the one trained with TRAIN-2500, can become the worst one in seldom training cases. An interpretation to this phenomenon is that the performance of the models trained with different number of data samples are actually very similar. It is common that each time the training in deep learning network varies a bit, and this small variant leads to the change of the order in our case. From this perspective, it is hard to conclude which training size is the best for our model. In fact, no matter how the models change, the losses of the models are always much lower than baseline.

To further explore how the predictions match the true values of the training set, we visualize

the predicted and true positions of 3 random simulations of two models(Figure 10). Because we use the same model structure and hyperparameters for all models, we visualize only the results of the best model, TRAIN-2500, and the worst model, TRAIN-10000, to have a knowledge of the upper limit and lower limit of the model architecture. The full visualization for all models is in the Appendix A.

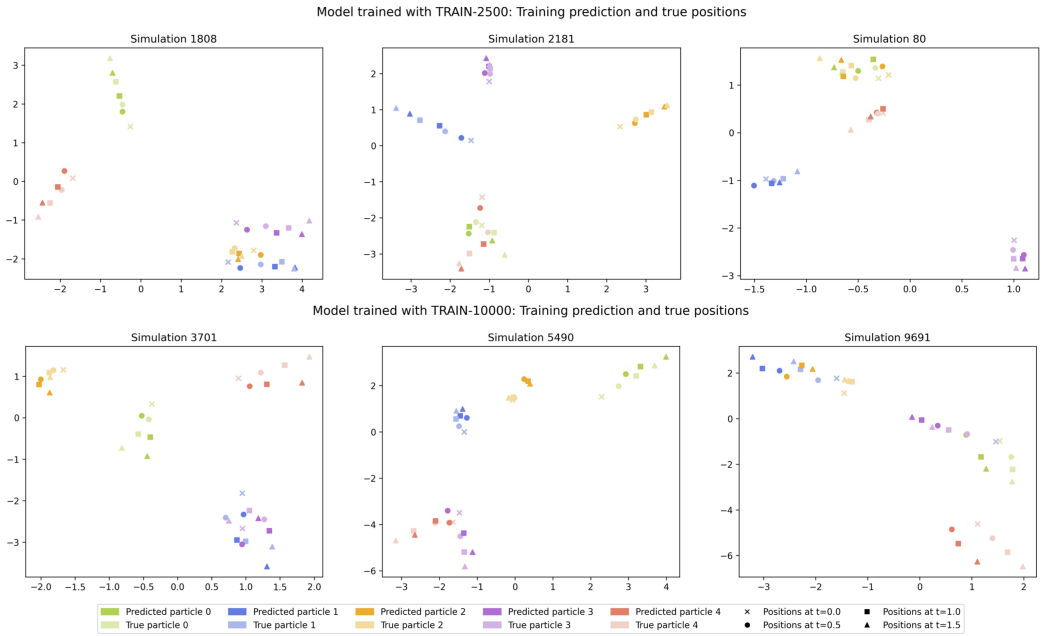


Figure 10: Visualization of 3 random simulations of the predicted positions generated by the best(TRAIN-2500) and the worst(TRAIN-10000) models during training

From the figure it is obvious that for TRAIN-2500, the most predicted positions more or less overlap with the true positions on each prediction horizon. This means, the best model has been indeed trained well to simulation the particle behavior in the training set. Compared to that, the predictions in TRAIN-10000, although not as close to the true positions as that of TRAIN-2500, are still in the neighborhood of those points. Besides, both sequences of plots suggest that the models do well when the particle trajectory is quasi-linear while when the trajectory is cluster-like or intersect with other trajectories, the model may be struggled in coming up with a correct prediction. This is especially the case for the model trained with TRAIN-10000.

4.1.2 Evaluation

First, we check how the loss values change at different prediction horizons with different training data sizes. The left plot in Figure 11 shows that for the models we trained in 4.1.1, the models trained with TRAIN-2500 is the best one when dealing with test data. This model is generalized well and does not show a tendency to be overfitting. Similar, the model with TRAIN-7500 also does well in accurate prediction. In comparison, the other four models are not as effective as a linear baseline model. Especially, the worst trained model, the one with TRAIN-10000, has more than three times of the loss values than that of TRAIN-2500. Therefore, it is clear that the not well-trained model is not reliable in evaluation.

If we focus on the prediction horizons, it is obvious that as the timestep increases, the losses of all models grows continuously. The reason is that the the prediction at each timestep is based on the prediction at the last timestep, then the errors are propagated through the sequence till the last timestep. As a result, the later the prediction, the more problematic it can be. Moreover, we can find the baseline model has the lowest loss at $t = 0.5$, while

its loss at $t = 1.0$ and $t = 1.5$ increases dramatically. This means, the pattern of the first prediction horizon is difficult for the model to learn such that they are not as good as a baseline, while in later time points the prediction accuracy is improved.

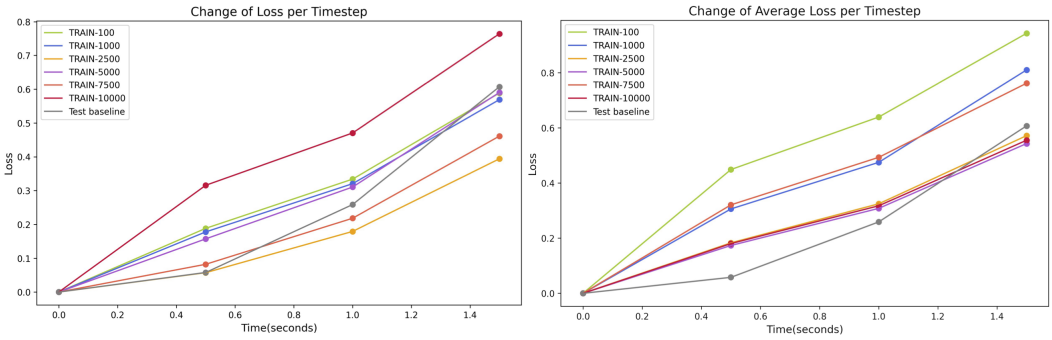


Figure 11: Test loss of models trained with different training data size at different prediction horizon after one evaluation(left) and 10 times evaluations(right, average loss)

As mentioned before that the order of the model performance is not reproducible, in order to have a more thorough investigation into the impacts of training sample size on model predictions, we run the program for 10 times and calculate the average test loss for each model. The right plot of the Figure 11 shows the results. It is not surprised that the model with TRAIN-100 has the highest test loss, because it is trained with the fewest training samples and prone to overfit. Then, it follow the models trained with TRAIN-1000 and TRAIN-7500. It is hard to interpret why the model of TRAIN-7500 shows dissatisfying performance, as theoretically the model trained with more data tends to be more robust to be overfitting. It may be that the hyperparameter setting is not suitable for this training set. As for the other models, they are within the same performance level, however, they are not as good as baseline model at $t = 0.5$ and $t = 1.0$. Therefore, we can conclude that our models are accurate when predicting the positions at $t = 0.5$ and $t = 1.0$, while the performance of some models is acceptable at $t = 1.5$. The training sample size is not necessarily correlated to the performance. This may be dependent on the specific model architecture and the hyperparameter settings.

We also visualize the predictions of the best and the worst model as what has been done in 4.1.1. In Figure 12, it is obvious that when the particles move in a linear trajectory, the models also give the predictions distributed in a linear shape. However, the models sometimes make mistake in the direction of the line, which results in larger distance between true and predicted positions as the particles go further from the initial position. Again, we can find that the worst model gets stuck in handling some particles with interacted trajectories, while this problem is much less severe in the best model. Despite this fact, the most predictions of the worst model are still near the true positions. If the required accuracy is not strictly high, this model may be useful in some scenarios. The full visualization for all models is in the Appendix B.

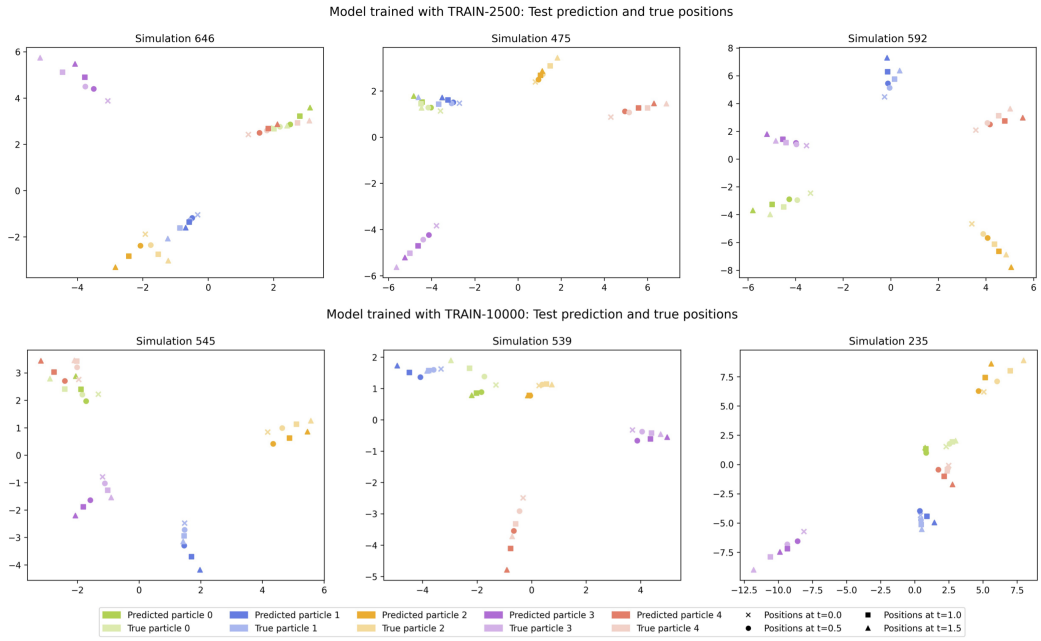


Figure 12: Visualization of 3 random simulations of the predicted positions generated by the best(TRAIN-2500) and the worst(TRAIN-10000) models during evaluation

4.1.3 An Additional Experiment: Regularization

The previous sections has revealed that our models suffer from overfitting to some extend. To deal with it, we tried some regularization measures including dropout and L2-regularization. Following Figure 13 shows the test losses after using L2-regularization with setting *weight_decay* to 0.001 or adding a dropout layer with setting the *rate* to 0.01 in dense layer block.

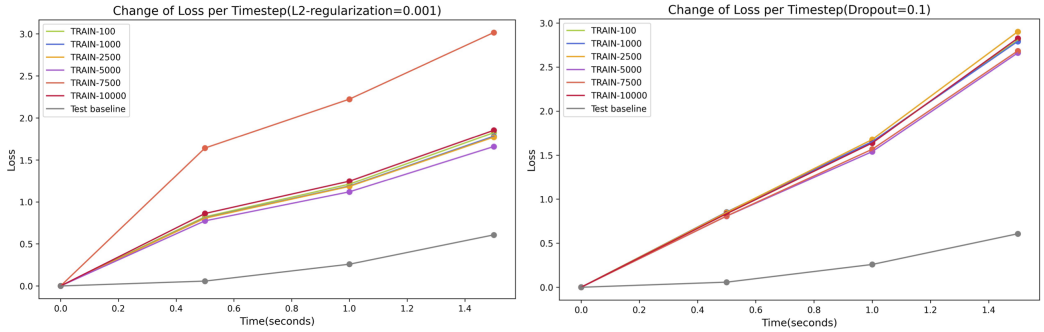


Figure 13: Test loss after using L2-regularization(left) or dropout(right)

The above two plots shows that even if small penalty resulted from L2-regularization or dropout can lead to significant increment in the test loss for all 6 models. The test loss is not reduced, but reversely, much higher than baseline. The underline cause is that to cope with the simple and small-scale training set, we design a shallow network with small number of neurons. The capacity of the model is already low and vulnerable to regularization. That is the reason why we do not use dropout or other regularization approaches in the final model.

4.2 Task (3.1)

4.2.1 Training and Validation

We rebuild the data in the experiments in order to format the input data. During the training phase, we alter the number of epochs, learning rate, dropout rate and number of layers among other things.

We used MSE as our loss matrices since this is a regression problem and the distances between the anticipated and real values can indicate the model's performance to some extent. During the training phase, we collect the training and validation loss for each epoch to examine the model's performance on the training task. As seen in Figure 14, the loss drops dramatically within 5 epochs and then slowly decreases during the training procedure. The initial loss of 0.227 was decreased to 0.018 on the last epoch, while the validation loss was lowered from 0.228 to 0.007. Validation loss is often smaller than training loss; nonetheless, the margin between them remains relatively constant, and training loss fluctuates. The reason for this phenomenon is during model training, we apply dropout, which penalizes model variance by randomly freezing neurons in a layer.

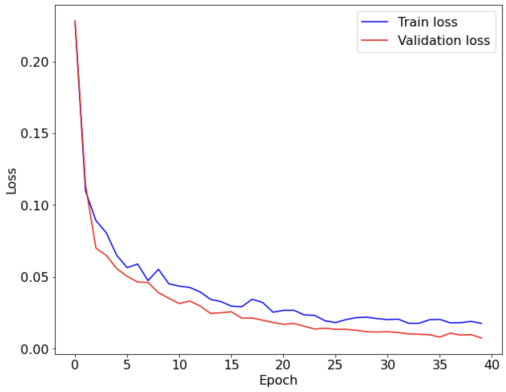


Figure 14: Loss over Epoch curve

4.2.2 Evaluation

On a test dataset, we evaluate our model. We begin by looking at the loss. As shown in table 5, the loss of the test set is lower than the loss of the train set, indicating that our model is not overfitting.

Train	0.016389641910791397
Validation	0.008314969949424267
Test	0.008172334171831608

Table 5: Task 3.1 loss summary

As the loss can't show us the virtual idea how our model perform, we also plot all the predicted charges' values for three particles over all simulations 15. By observing the general trends, we can know how accurate our predictions are.

We additionally present Figure 15 for all the anticipated charges' values of three particles over all simulations because of loss can't convey us a virtual picture of how our model performs. We can assess how accurate our predictions are by looking at the overall patterns.

We chose a few scenarios to plot in order to examine the good and bad simulations the model perform. We can deduce from Appendix C that when particle charges are approximately comparable, predictions seem to be more accurate. The predictions fail when there are extreme cases of three charged particles, as seen in Appendix D. As illustrated in simulations 3 and 16 and other comparison pairings, our model has a higher accuracy when the particle charges are large and a poorer accuracy when the charges are comparatively small.

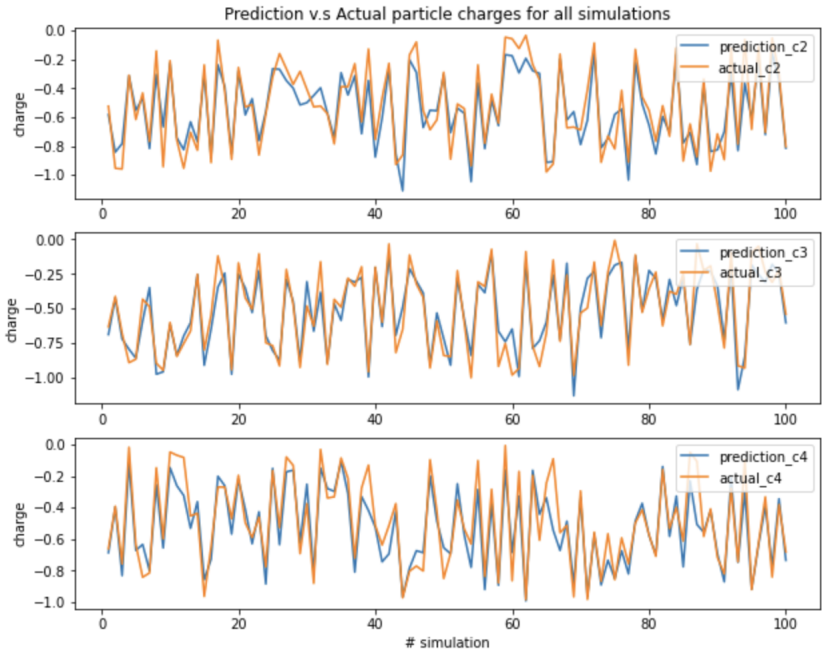


Figure 15: Prediction v.s Actual charges for c2, c3, c4

4.3 Task (3.2)

4.3.1 Training and Validation

Figure 16 illustrates the loss curves for training and validation. In both training and validation phrases, it takes around 30 epochs to reach a loss threshold of roughly 0.01. The loss values range from 0.005 to 0.002 as the number of epochs reaches 60. The model performs consistently during the validation phase as evidenced by the validation loss, which exhibits the same trend and order of models.

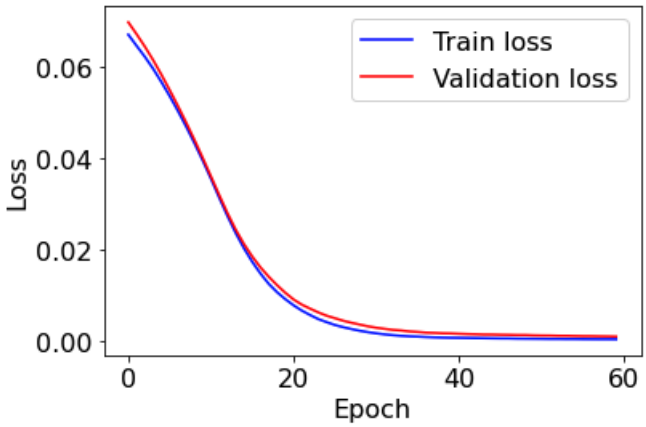


Figure 16: Task3.2 - Loss over Epoch curve

4.3.2 Baseline

Additionally, we put into practice a straightforward linear baseline where we extrapolate between the last two timesteps of the input positions. As demonstrated in Table 6, we compare the losses between our model' and the baseline'. We also trace the baseline predictor's trajectories which are represented by the purple lines in the Figure 17 and Figure 18. Our model outperforms the baseline in terms of quantitative and qualitative performance and

can make some accurate predictions.

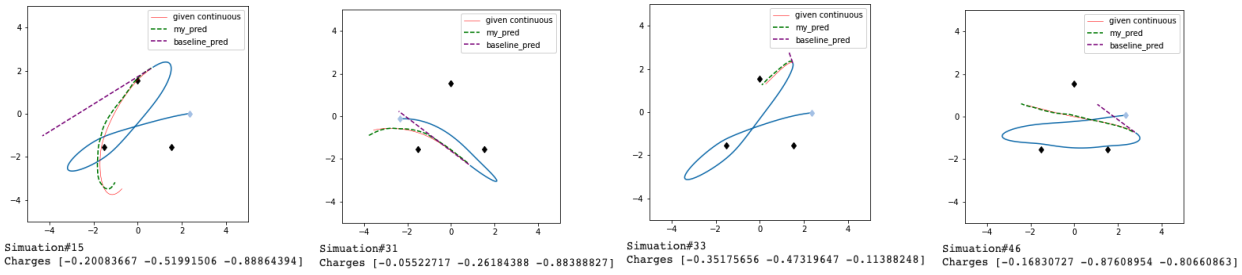
Train	0.0002747625360886256
Validation	0.0008573965542018414
Test	0.3285925749863506
Baseline	2.884847350343732

Table 6: Task 3.2 loss summary

4.3.3 Evaluation

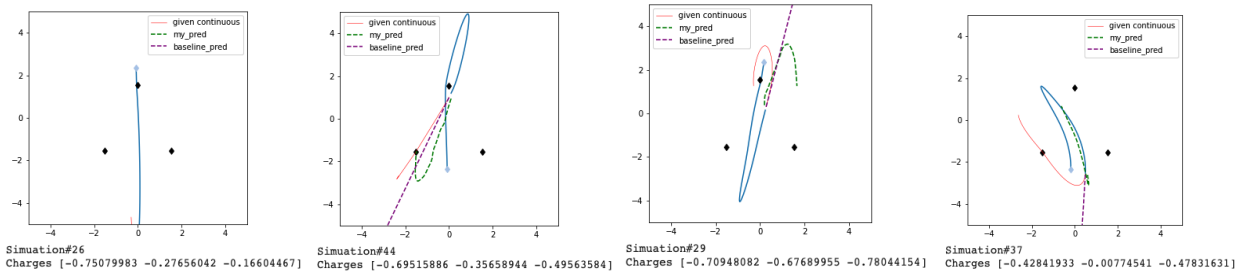
To gather the loss for evaluation, we execute our model on a test dataset and display it in Table 6. The test loss is significant when compared to the train and validation losses, indicating that the model underperformed in the test dataset and possibly overfitting.

We plot trajectories of all simulations to acquire a visual understanding of how the model performs. Figure 17 demonstrates that the predictions are rather accurate in those situations and the general trajectories are the same. The model was unable to predict the continuous trajectory under some circumstance , as shown in Figure 18.



(a) (b) (c) (d)

Figure 17: Task 3.2 - Correct cases



(a) (b) (c) (d)

Figure 18: Task 3.2 - Failed cases

5 Conclusion

5.1 Task (2)

Our implemented models have verified that the proposed method is applicable to the task of predicting the positions of 5 particles at a certain time point t given the charges, initial positions and velocities. Based on the architecture we design, the model trained with 2500 data samples distinguishes from other counterparts. This means that for this specific problem, the larger training set does not guarantee better performance. It may be that the data in

training set is similar, thus, training with more data samples does not contribute to higher prediction accuracy. In terms of the prediction horizon, the later the time point is, the more inaccurate the predictions will be. This is due to the RNN nature that the prediction errors are propagated and accumulated along the timeline.

One limitation of our models is that they are prone to be overfitting. This may be on the one hand that the hyperparameter or model structure is not optimal, on the other hand that the data is so easy to learn that the model can become overfitting easily. In the future, other architectures can be tested to verify this hypothesis. Besides, in our experiment we use the same architecture for all training sample size. More experiments can be designed such that there is a specific hyperparameter setting for each training set. In this way, we may mine some correlations between training set sizes and model performance.

5.2 Task (3.1)

To estimate the values of three negative charges giving the positions of positively charged particle p during a simulationm, we decided to use LSTM as our model. According to the test findings, our model can perform effectively in the majority of situations, especially when the three particles' values are generally similar. When three electrons' charges differ greatly from one another, the predictions are inaccurate. The fact that the data set is limited and the training data for certain situations is few could be one factor. If we have more training data which shows that the charges of the three electrons differ significantly from one another, we might expect improved performance.

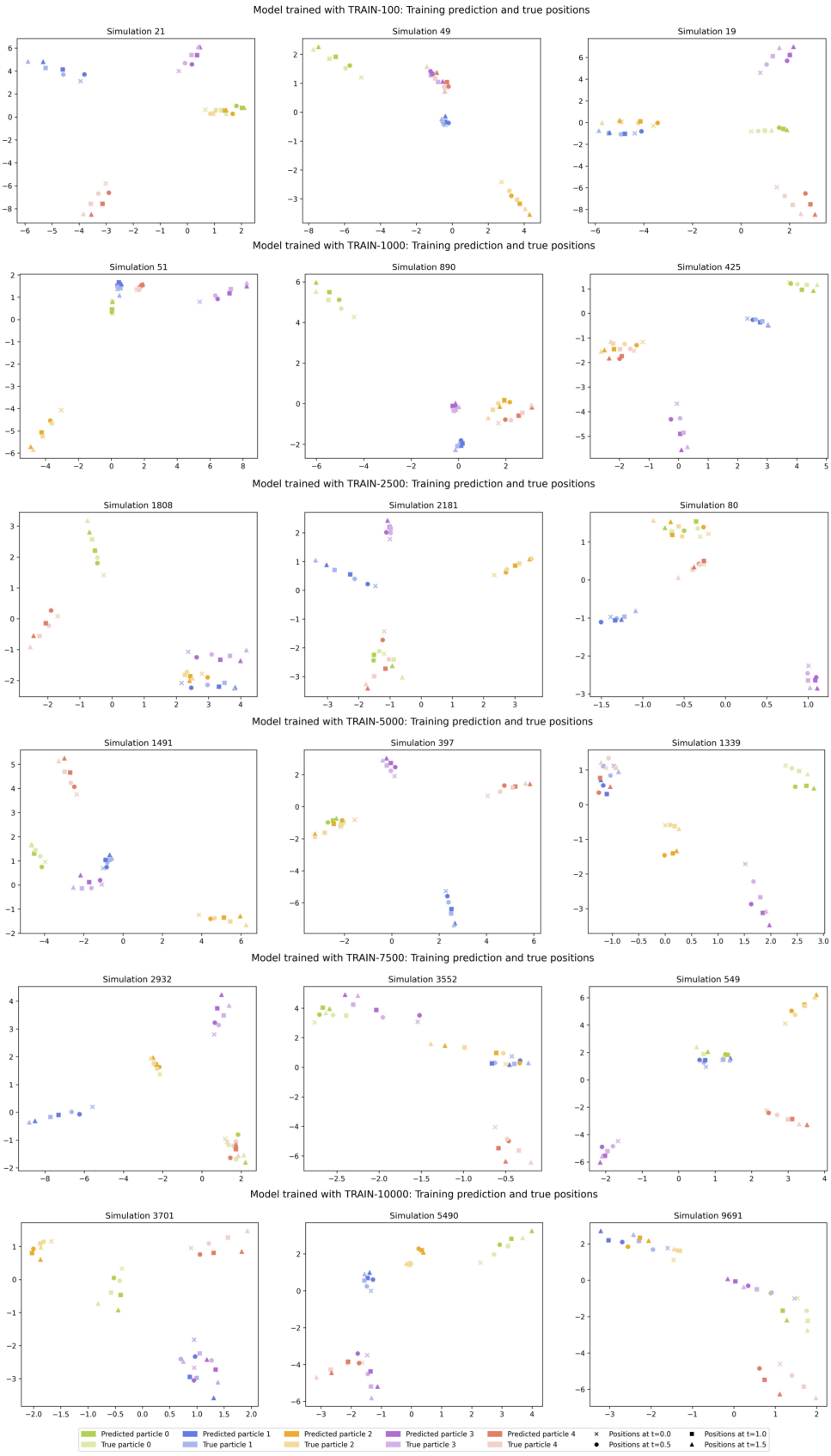
We might think about data augmentation, especially for the failed cases, in order to perform better. Regularization, on the other hand, might also be a solution.

5.3 Task (3.2)

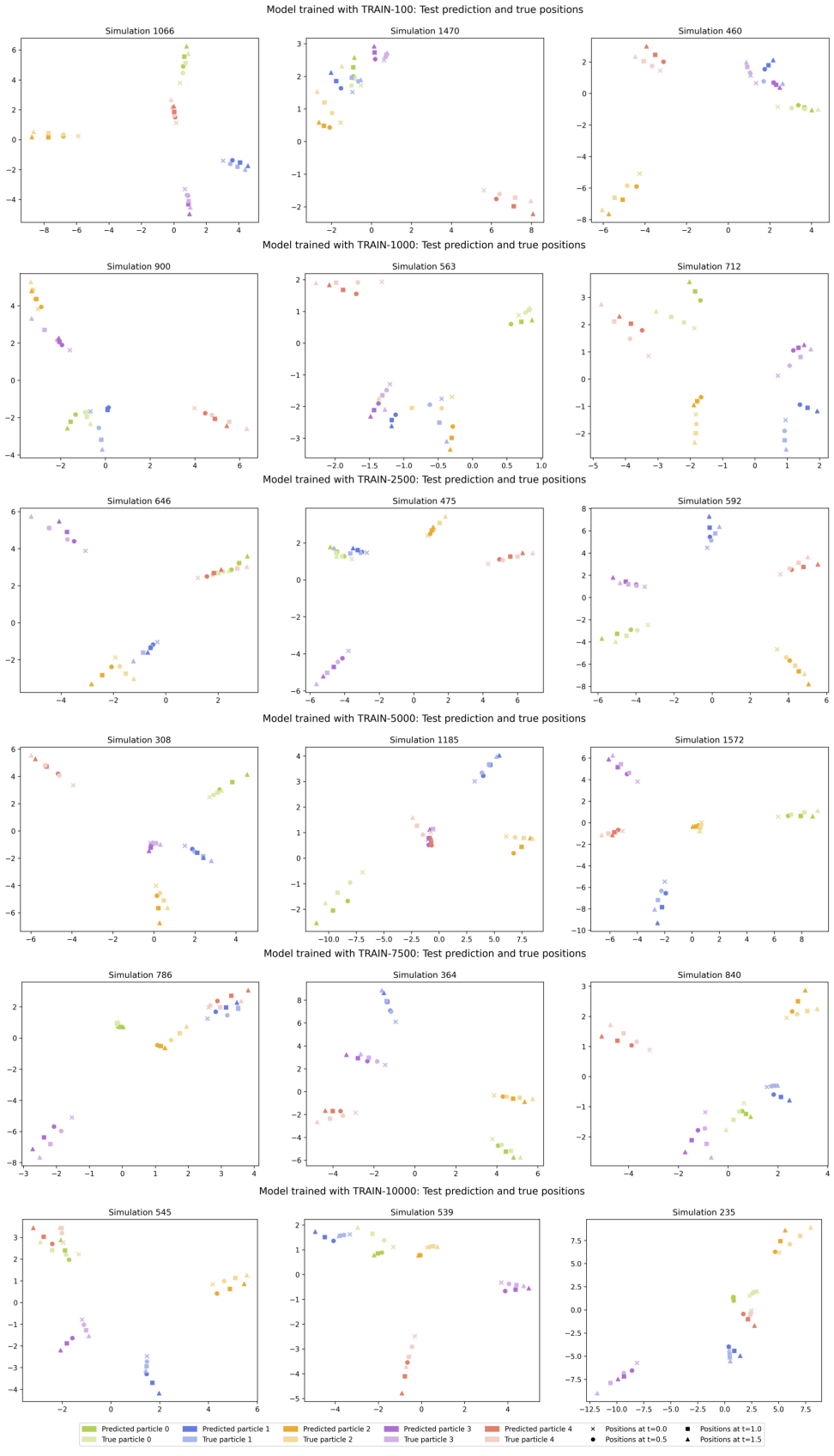
Our aim is to forecast the route of the positively charged particle p_1 for the extra 2 to 6 seconds while using its positions from the simulation up to $t = 10 \pm 1$. We choose MLP to preprocess the input and train the data using LSTM. According to the test results, our model can produce some reliable predictions and outperform the linear baseline in terms of both quantitative and qualitative performance.

Our models' propensity for overfitting is one of its drawbacks. This could be due to a number of factors, including an unfavorable hyperparameter or model structure and the ease with which the data can be learned. The parameters might be optimized in the future, and the architecture might even change.

A Task (2): Visualization of 3 random simulations for the predictions generated by 6 models with the corresponding true positions in training phase



B Task (2): Visualization of 3 random simulations for the predictions generated by 6 models with the corresponding true positions in test phase



C Task (3.1): Correct cases

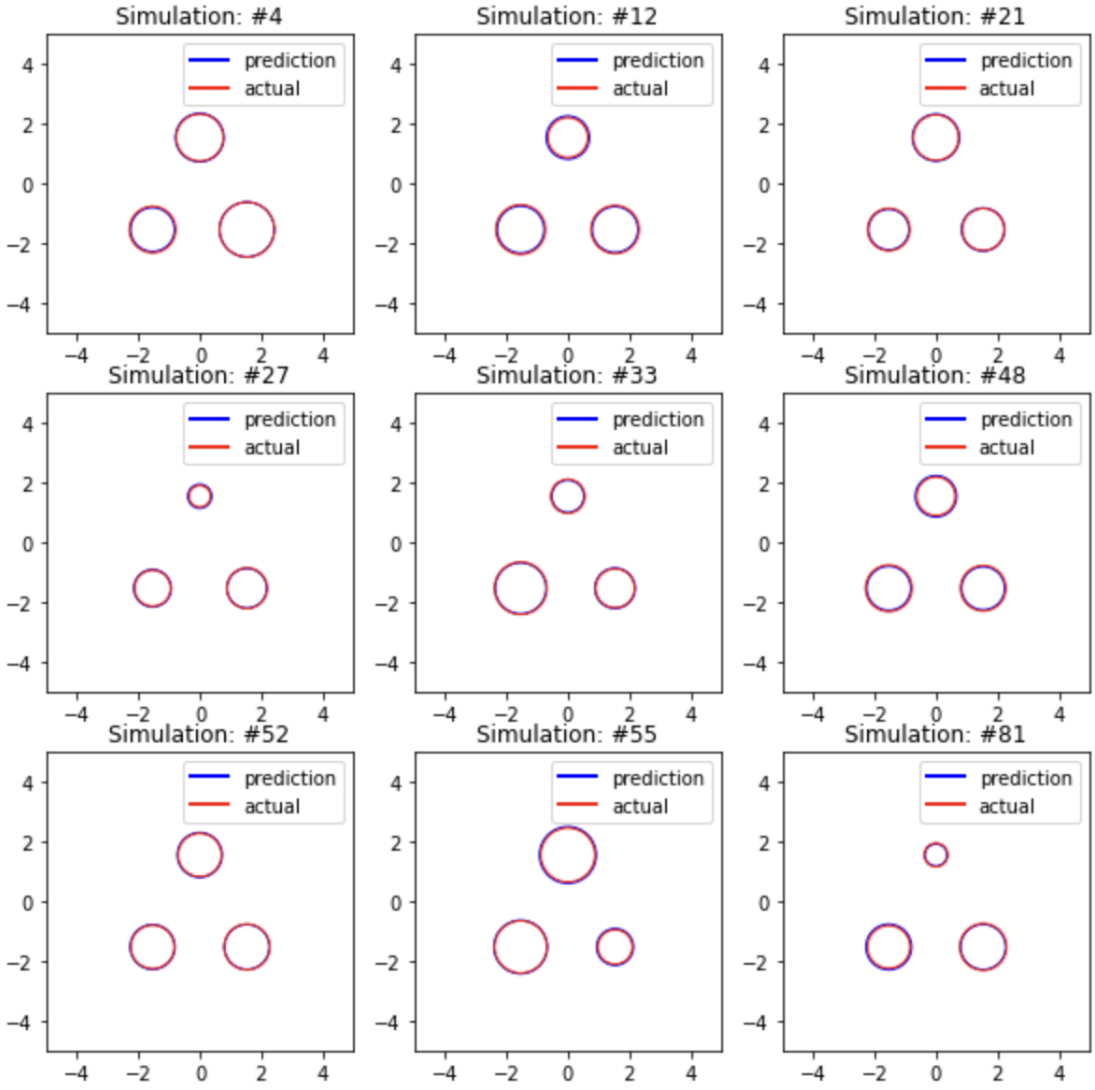


Figure 19: Task 3.1 - Correct cases

D Task (3.1): Failed cases

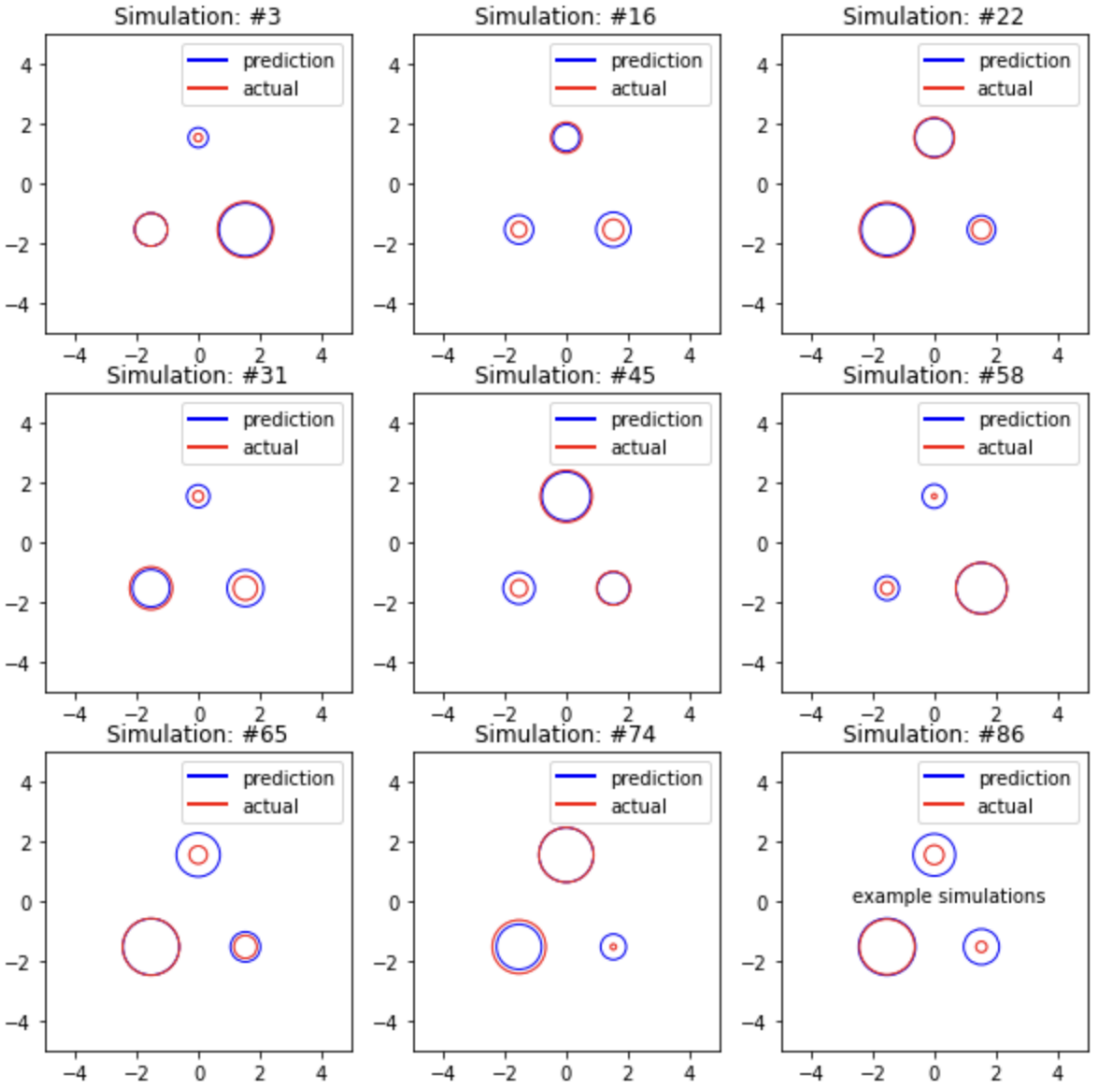


Figure 20: Task 3.1 - Failed cases