2AMM10 2021-2022 Assignment 2 & 3: Group 41

Nikhil Patni Silvia Sultana Aishvarya Viswanathan

June 20, 2022

1 Task 2

1.1 Problem formulation

For this assignment, we are tasked with building a deep learning model that can approximate the dynamics of charged particles evolving in a plane over time. We have a system of n particles p1, p2,. . ., p5 moving in the plane evolving over a period of time. For the sake of our experiments, the final positions considered of the particles are at discretized timestamps of t= 0, 0.5, 1 and 1.5. (i.e) with a standardized interval of t = 0.5. At any time t, the system is described by the position ${x_i}^t \epsilon R^2$ and velocity ${v^t}_i \epsilon R^2$ of each particle p_i . Each particle has an associated charge $c_i \epsilon\{1, 1\}$ that remains constant over time. All initial positions ${x_i}^0$ and velocities ${v^0}_i$ are given, and at any time t, the evolution of the system is governed solely by the forces of all pairs of particles acting upon each other. For any two particles, the force between them depends on their charges and relative locations. We are provided with three partitions of the data for this task: a training set of 10000 simulations, a validation set consisting of 2000 simulations, and a test set made up of 2000 simulations. A sample of the provided train set can be seen in Figure 1.

To sum up, this assignment is to evaluate if it is possible to predict the future state of a complex dynamical system with a deep learning model. We are trying to create a solution that auto-learns what this equation [${x^t}_i = {x_i}^{t-\Delta t} + {v^t}_i \cdot \Delta t$] does. To achieve this, we need to build a solution that would take in initial positions, charges of the 5 particles and the initial velocity and be able to predict the final position of the particle at the any of the required time point (0.5, 1, 1.5) with relation to the initial position at t= 0.

1.2 Model formulation

In order to learn relationship between different non-linear variables in a complex equation, we use Multi-layer perceptron to calculate final position of particles based on multiple features albeit without the softmax/sigmoid layer in order to use the MLP for regression purposes. SO our model, MLP for regression has no activation function in the output layer (i.e) it uses identity function as activation and produces an output of continuous values.

The model consists of 4 Fully connected layers with input being initial positions, velocities and charges and final output of final positions of all 5 particles. It uses Adam optimizer and Root Mean Square loss (squared L2 norm) function. The Mean square loss (MSE) is used here to quantify the loss and the Root Mean square error (RMSE) is derived from it to obtain RMSE between prediction and actual final positions. The loss function can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = (y_n - \hat{y_n})^2 \tag{1}$$

where n is the batch size. If reduction is not 'none' (default 'mean'), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

```
[tensor([[[-1.9361,  3.6532, -4.5496, -3.0958,  0.1073],
          [-0.2276, -2.7631,  8.2998, -3.0921, -2.8976]]]),
 tensor([[[-2.3387,  4.4387, -5.1946, -4.0422,  0.4396],
          [-0.2173, -2.9732,  9.4907, -3.3710, -3.1883]]]),
 tensor([[[-0.7878,  1.5892, -1.2906, -1.9173,  0.6539],
          [ 0.0267, -0.4211,  2.3834, -0.5785, -0.5676]]]),
 tensor([[-1.,  1.,  1.,  1., -1.]])]
```

Figure 1: Sample of train set (Batch size= 1)

y_n and $\hat{y_n}$ are tensors of predicted and actual final positions with a total of n elements each. Finally, the square of L is taken as RMSE.

1.3 Implementation and training

1.3.1 Data Handling and Preprocessing

We have training dataset of 10000 simulations, validation set of 2000 simulations and testing dataset of 2000 simulations. Each simulation contains position (x,y coordinate) of all 5 paricles at t=0,0.5,1,1.5 and also the initial velocity (fixed) and charge (fixed) of all 5 particles. In this problem, we only have position which is dynamic and is changing w.r.t. time in each simulation and the velocity and charges of all particles are fixed and given to us at t=0. The task is that we need to predict the position of all particles at time=t by using the initial position, velocity and the charges. So we split the problem into 4 timepoints in which the initial timepoint is (t=0) and final timepoint would be t=0.5,1 and t=1.5. We created three train dataloaders for t=0- t=0.5, t=0- t=1 and t=0- t=1.5. To create the dataset, we used initial position and final position of all particles by giving positions_train[:,0,:,:] and [:,1,:,:], initial velocity and charges of all particles. Similarly we created datasets for validation and testing. The validation and testing data are set aside and only the training dataset is used to train the model. Testing dataset is only used during the evaluation of our model so that it can replicate a real-world scenario and present unforseen distinct particles simulation to our model. In the end, we are using train and validation data loaders of batch_size=8 and test loader of batch_size=5.

1.3.2 Model

We are using a Multilayer perceptron (MLP) which is a fully connected class of feedforward artificial neural network (ANN) to fit our problem formulation. Our MLP architecture consists of four linear layers which increased neurons as shown in Fig 2. The initial layer takes an input of a vector of size 25 (initial x and y coordinates, initial velocity coordinates and charge for all 5 particles) and the output layer gives the final position (x and y coordinates) for all 5 particles. We used ReLU activation function in all linear layers over sigmoid to reduce likelihood of vanishing gradients and to introduce non-linearity. It is a very common issue if you're designing a deep neural networks.

The first linear layer takes an input of a vector of size 25 as a tensor which has initial x and y coordinates, initial velocity coordinates and charge (5 values) for all 5 particles. The second layer of size (64, 128) then takes the activation map of first layer and outputs the activation of size 128 which is then passed to third linear layer and finallly the last linear layer takes the input of size 256 and outputs the linear tensor of size 10. The output tensor is comprises of final x and y coordinates at time=t for all 5 particles. We have used increasing number of neurons in dense fully connected hidden layers so that it will try to learn the non-linear relationship between a complex equation variables. We are not using any activation function in output layer as we want the output as it is which model predicted. During training, we then calculate the Root Mean Squared Error (RMSE) loss by comparing the output tensor with the actual tensor (final position of all 5 particles at time t) and then backpropagate the

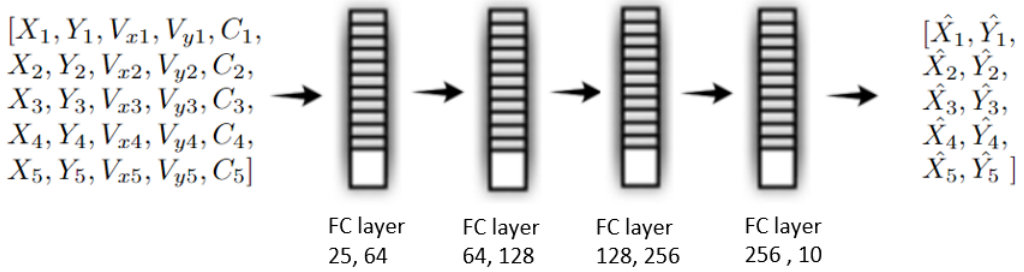

Figure 2: Model Architecture

loss through the network to update the weights and bias (model parameters). An illustration of the architecture of our model is provided in Fig 2.

1.3.3 Loss Function

This model uses Multilayer perceptron for regression and RMSE loss would be a good choice. In order to retrieve this loss, we use MSE loss function and then take the square root of the loss found. The function equation used to calculate said loss is defined here. (1)

1.3.4 Training the network

We initialize the network, loss function, and optimizer (Adam) with learning rate and weight decay. To train the network, we pass coordinates (X and Y) with both their initial velocities (V_x, V_x) along with charge (C) of all 5 particles as a single tensor (i.e) with a 25 values through 4 Fully connected layers. At the first layer, the tensor of 25 is converted into 64, then 128 followed by 256 and at the final Fully connected layer, we get the output as 10 values which correspond to the X and Y coordinates of the final position of each of the five particles at required time point. The fully connected layers are activated with 'ReLU'. No softmax/ sigmoid is used in the final layer as the task is regression and not classification. The architecture is demonstrated in Section 1.3.2. We are then computing loss between actual position recorded of the five particles at the required timestamp with the coordinate the network spits out. After which the losses are backpropagated to calculate the gradients of our model followed by updation of the weights aided by Adam optimizer.The average training loss is calculated by dividing the summation of the running losses by the size of the train_loader. We used the validation dataset to calculate validation loss of our model so that we can tune the hyperparameters and also to correct overfitting in network. After the experimentation explained in Section 1.4, we finalized Adam as optimizer with learning_rate= 1e-4 and We also used MSE loss as the loss function with batch_size of 8 for both train and validation loader. With these parameters, we got the Min_validation loss of **0.2748** at Epoch 97. We chose 100 as epoch size as we can see that losses started to converge and we are also not seeing any over-fitting happening in the model. This is currently the best result we have achieved.

1.4 Experiments and Discussion

1.4.1 Experimental Setup

We use different hyper-parameter settings for training the model based on the different data loaders we created for our model implementation, analyze the results at three different time steps, and finally choose the best hyper-parameter combination obtained for testing purposes on the test data loader on three different time steps. The test data loader is kept out of the model implementation and training phases, and it is only used to forecast final locations once the model has been tuned.As explained previously in previous section (add name) we use the Mean Squared Error(MSE) [1] suitable for our regression purpose.

We selected to evaluate two commonly used optimizers for deep learning tasks, Adam[2] and Stochastic Gradient Descent(SGD)[3]with momentum utilizing a single hyperparameter setting and then see their validation losses for our experimental configuration. We used the Adam optimizer with a learning rate of 1e-4 and a weight decay rate of 3e-5 to compare the optimizers. We set the learning rate for SGD with momentum to 0.001 and the momentum to 0.9. As illustrated in Figure, their validation losses are tracked down. Even though both optimizers exhibit almost identical convergence,(as explained in next section) the validation loss with Adam for all three time steps is significantly better for the same amount of epochs. As a result, we proceed with our optimizer selection of Adam for our future experiments only at time step t=0.5. We further experiment using different learning rates on this optimiser and then finally on different batch sizes. We use learning rates of 0.01,0.001 and 1e-4, as well as batch sizes of 8,16, and 32, to find the optimum combination for our model.

The last experimental configuration is based on jumping time steps to estimate final placements. For example, at t=1.5, we forecast the location of the charged particle, but the input data is from t=0. We test all possible combinations, including one in which we don't leap time steps, i.e., we forecast the position at t=1 based on input data at t=0.5. We present this experiment as an additional insightful experiment as mentioned in the assignment rubric. We further tested against different training data sizes.

1.4.2 Experimental Results and Analysis

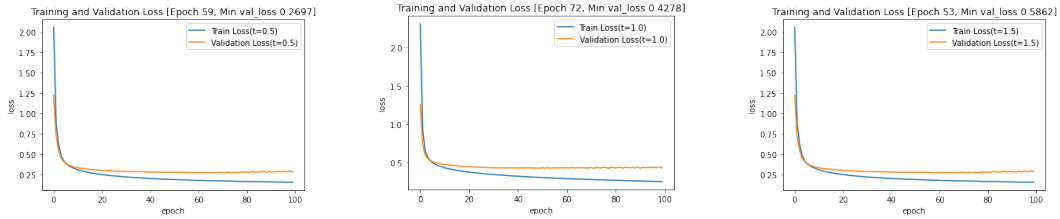

Figure 3: Training the model using Adam optimizer for three time steps

Figure 1 shows that the best loss obtained using Adam for times t=0.5,1 and 1.5 was 0.2697,0.4278 and 0.5862, respectively. It's simple to see how, when given a beginning location at t=0, the model has a harder difficulty predicting the end position when the final time is significantly later than our initial knowledge time. As shown in Figure 2, the best loss obtained using SGD for periods t=0.5,1 and 1.5 was 0.3421,0.5433 and 0.7046, respectively. This follows a similar trend, with the loss increasing as the time step grows, making precise forecasts impossible. So, we made the choice to use Adam optimizer for our model.

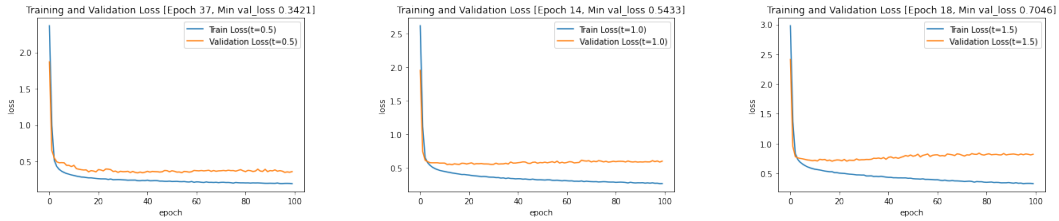

Figure 4: Training the model using SGD optimizer for three time steps

Once an optimal optimizer is chosen, we experimented with three different learning rates (0.01, 0.001 and 1e-4) for it's hyperparameter. We see that the losses for both training and validation is very eratic for larger learning rates and gets progressively better for smaller learning rates. This is due higher learning rates are undesirably divergent. [4] Lower learning rates take controlled steps but requires many steps to reach convergence and a mid value slightly leaning to the smaller end is optimal and the best value is chosen through testing for different cases. From our testing, we chose 1e-4 as our learning rate for our optimizer Adam.

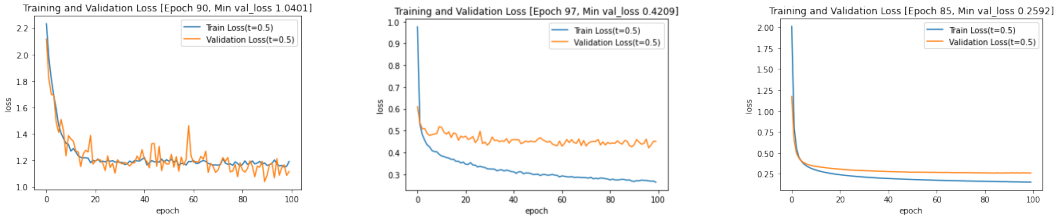

Figure 5: Training the model using Adam optimizer for three different learning rates

Now that the optimizer and learning rate has been decided upon, we experiment to decide on the batch size. We ran an analysis on 3 choices: 8, 16 and 32. Our best loss result is from batch size 8. This may be due to the fact that small batches offer a regularizing effect as they add noise to the learning process. [5] The high variance obtained from the estimate of gradient issue that might arise from using small batches can be stabilized using small learning rate. A batch size of 8 turns out to be suitable for our requirement.

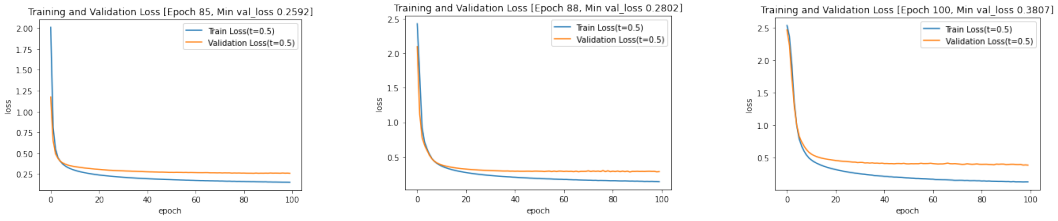

Figure 6: Training the model using Adam optimizer for three different batch sizes for lr= 1e-4

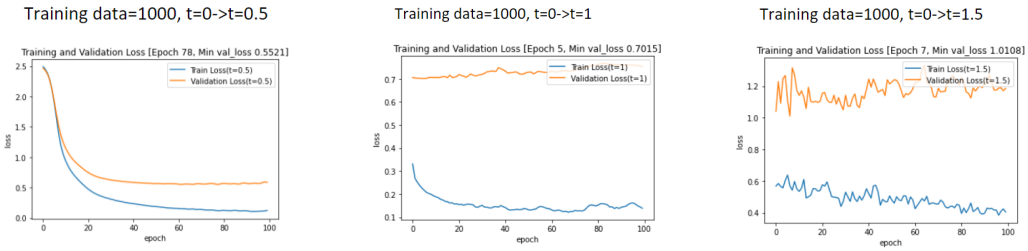

Figure 7: Performance of model at different time steps with initial time step t=0 being constant

While we experiment with the model's predictions when jumping timesteps, we see that when the time steps are too far apart, the model has difficulties predicting. Figure 7shows that the model works best in predicting the position at t=0.5 when the particle is at t=0. However, as the time steps are increased, the performance deteriorates. This is due to the fact that when we jump time steps, we are not actually capturing the dependencies between the intermediate places that the particle had gone through before reaching the final position since we leave the data of t=0.5, 1.0 out of it.

Furthermore, the complexity of our model is insufficient to extrapolate the data two steps beyond the original placements. Because the particle positions would be widely apart and extremely diverse across big time steps, it is difficult to approximate the locations using a model like MLP. In this case, Graph Neural Networks (GNN) would have been a superior estimator. Furthermore, if we had been permitted to capture temporal dependencies between networks at the same time, networks like LSTM and GRU would have fared extremely well.

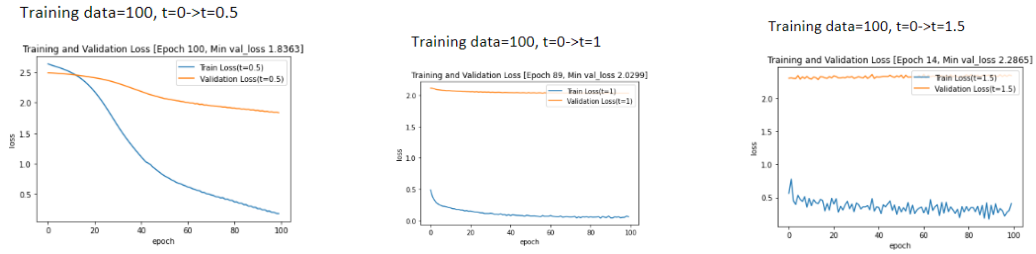

Figure 8: Performance of model with decreased train size at different time steps with initial time step t=0 being constant

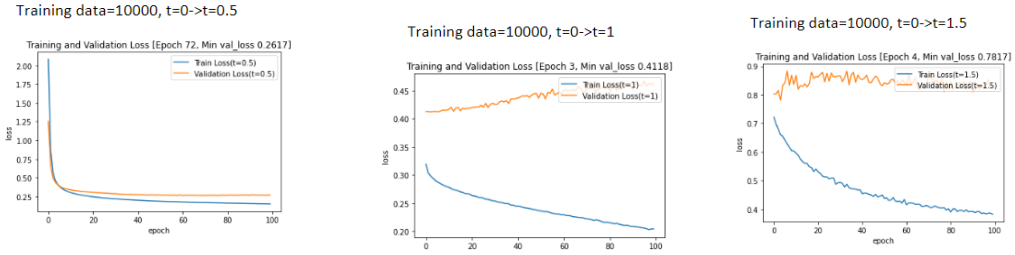

Figure 9: Performance of model with different train size at different time steps with initial time step t=0 being constant

We also varied the training data from 100 to 1000. We plot the performance of 100 training data in Figure 8 and 10,000 in Figure 9.

Based on our experiments, we established that for this final coordinate prediction task, the best synergy in hyperparameters for our specific model occurs under the setting of :

1. **Optimizer** : Adam optimizer
2. **Batch Size** : 8
3. **Learning Rate** : 1e-4
4. **Number of epochs** : 100

We can see that validation losses started to converge and do not see any over-fitting happening in the model. This is currently the best result we have achieved.

Another experiment we ran was to compare the results of our model with the actual results one obtains when they use the equations given [${x^t}_i = {x_i}^{t-\Delta t} + {v^t}_i \cdot \Delta t$] to see if our deep learning model was able to learn how to approximate the dynamics of charged particles evolving in a plane over time. The results obtained by both methods are tabulated here 1, 2 and 3.

1.4.3 Testing and Results

Now that we have used the above fine-tuned model with the hyperparameters decided on using validation data to test our network. We can now test our model by simulating a real

Losses	t=0-0.5	t=0-1.0	t=0-1.5
RMSE Loss on test data	1.8935	2.0854	2.3639
RMSE loss compared with baseline	1.9018	2.1259	2.4515
R2_score	0.3998	0.4250	0.4093
R2_score compared with baseline	0.4065	0.4340	0.4186

Table 1: Our Model compared with simple linear baseline on training_size=100

```
Model loaded from <== MLP-checkpoint_1.pt
Best val_loss: 0.25903461
Output pos:  tensor([-0.3869,  0.7751,  2.6730,  1.6499, -4.9308,  1.1021,  0.7335,  2.4499,
        -1.5099,  3.7415], device='cuda:0')
Actual pos:  tensor([-0.1832,  0.5443,  2.7473,  1.3889, -5.0052,  1.2809,  1.0530,  2.1327,
        -1.5505,  3.6140], device='cuda:0')
RMSE Loss after 200 simulations: 0.27393190387636424


Output pos:  tensor([-2.3708,  3.8172, -0.3151, -0.9971,  1.7124, -1.0464, -3.3814,  1.9224,
         2.2935, -2.2597], device='cuda:0')
Actual pos:  tensor([-2.1900,  3.6610, -0.6610, -0.6277,  1.9556, -1.0905, -2.9526,  2.0430,
         2.0230, -2.2324], device='cuda:0')
RMSE Loss after 400 simulations: 0.2772325385175645


RMSE Loss on test data: 0.27723254
R2_score: 0.9826
```

Figure 10: Final output coordinates from model compared with actual x-y coordinates of the particles after t = 0.5

world setting by using the test loader with batch_size of 5 and the coordinates, velocities and charges of all 5 particles will be passed according to batch size to our model similiar to training. The Model then returns final tensor of 10 values which are the final position of all 5 particles past a required time t. This value is then compared with the label to get the correctness/accuracy. We then calculate the average test accuracy over batch_size on how close the network can predict the position of the particles. Our MLP for regression model has performed quite well on the given dataset and achieved a final RMSE Loss on test data of **0.2950** and final R2_score of **0.9826**. A sample snippet of our output can be found in Figure 10 for t = 0.5. Here, the 'Output pos' tensor gives the values spit out by our model and the 'Actual pos' is the true position of the particles and can be compared.

1.5 Conclusion

We have discussed and implemented a MLP for regression model to predict the final positions of five particles based on initial positions, velocities and charges of each. We achieved a R2_score of 0.9826 with a RMSE Loss of 0.2772. As the requirement of the task is to predict the final position of the particles after particular interval and not the capturing time dependencies between different interval timestamps (i.e) investigating transitions from one timestamp to another like t= 0.5 to t=1.0 and then to t= 1.5. MLP for regression works fine. In case, capturing time dependencies is required, LSTM would have been a better choice. On a general scale, Graph neural nets (GNN) work well for problems like particle dynamics so that is also a solution option for this assignment. The GNN model we came up with for this assignment did not perform better than the MLP implemented. In fact, the MLP was seen to approximate the complex functions better. So that would be a direction to look into for improvement of the solution for the current problem statement.

Losses	t=0-0.5	t=0-1.0	t=0-1.5
RMSE Loss on test data	0.5682	0.7096	1.0620
RMSE loss compared with baseline	0.5560	0.7397	1.1304
R2_score	0.9386	0.9193	0.8566
R2_score compared with baseline	0.9453	0.9215	0.8603

Table 2: Model compared with simple linear baseline on training_size = 1000

2 Task 3.1

2.1 Problem formulation

For assignment 3.1, we are tasked with building a solution to study the case of a single particle interacting with three charges that are present at fixed locations in the plane (i.e)We need a model that can predict the charges of the three stationary particles based on the path of the fourth particle's motion trajectory. So, we have a single moving particle p_1 that is moving in the same plane as three particles p_2, p_3, p_4 that are present at fixed positions carrying charges of c_2, c_3, c_4. Our task is to predict c_2, c_3, c_4 based on the path of p_1. One can consider this a special case of a particle system with n = 4 as described in assignment 2, but with the position of particles p_2, p_3, p_4 fixed. One difference is that we fix the charge of particle p_1 to 1, and charges c_2, c_3, c_4 are now sampled uniformly from the interval [-1, 0], i.e. they are no longer sampled from set { -1, 1}.

For any two particles, the force between them depends on their charges and relative locations. The dataset is generated by shooting a particle into the plane such that it can interact with the three fixed charges. The three charges are placed in a triangle configuration around the center of the plane while particle p_1 is shot in from a random position on the side of the plane, with an initial velocity pointing towards the approximate center of the plane. The generated simulations are of varying length. All simulations start at t = 0 and end at t = 10 $\pm$ 1. We are provided with these simulations alongside the values of charges c_2, c_3, c_4 for training such that we can predict the charges from the simulation. We are provided with the simulations sampled with $\Delta t = 0.1$, (i.e) 100 $\pm$ 10 steps. We are provided with only positions ${x_1}^t \epsilon$ R of particle p_1. Finally, we have 800 simulations for training, 100 for validation and 100 for testing.

2.2 Model formulation

Here, since we need to build a model to associate the trajectory to the charges such that given a new trajectory, the model can predict the charges at the end of the trajectory simulation. In order to achieve this, we need the retain information regarding the trajectory route over the entire 110 step length. This calls for requirement for memory retention over a relatively long period. LSTM Network (Long short term memory),a variant of RNN, would be suitable for this task. As it's main strength is that it is capable of learning long-term dependencies which is exactly what we need. LSTM are RNN with the provision of a cell state capable of carrying over 'useful information' (learnt via training to identify). The LSTM has the ability to remove or add information to the cell state, carefully regulated by structures called gates. LSTM have three gates. The first is the 'forget gate' through which the existing hidden state and new information are passed through a sigmoid to determine importance and forget current cell state memory accordingly. The second is the 'input gate' which takes the tan h and sigmoid of the current hidden state and input value and pass them through a point wise function to determine the input to the cell state (long term memory). Now the third gate is to decide the new value of the hidden state by taking the point wise multipication output of the tan h of the new cell state and the sigmoid of the existing hidden state and new input. This way, with the help of these three gates, the memory of the cell state and hidden state are refreshed to retain both long and short memory.

Our model takes the trajectory (x and y coordinate) of p_1 and predicts the charges of the

Losses	t=0-0.5	t=0-1.0	t=0-1.5
RMSE Loss on test data	0.2797	0.4309	0.8198
RMSE loss compared with baseline	0.2713	0.4769	0.8909
R2_score	0.9821	0.9640	0.9085
R2_score compared with baseline	0.9842	0.9574	0.9040

Table 3: Our Model compared with simple linear baseline on training_size = 10000

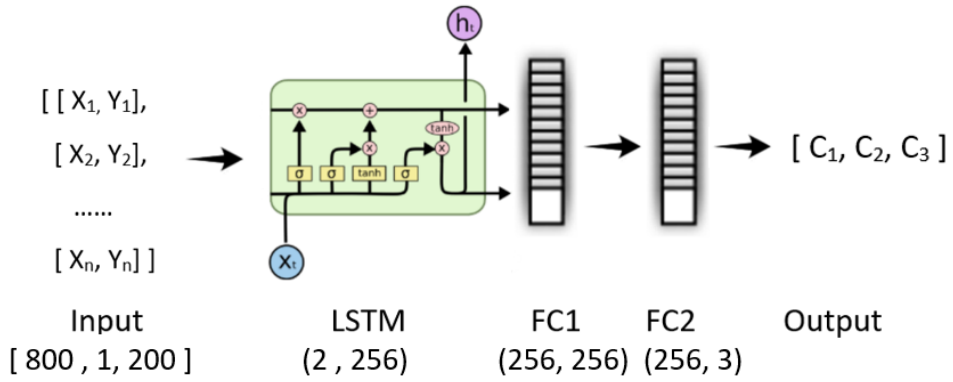

Figure 11: Model Architecture

three stationary particles $p_2, p_3 and p_4$. Our model architecture constitutes of one LSTM cell followed by 2 fully connected layer that outputs 3 values which are the charges themselves. It uses SGD optimizer and Mean Square loss (squared L2 norm) function. The loss function can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = (y_n - \hat{y_n})^2 \tag{2}$$

where n is the batch size. If reduction is not 'none' (default 'mean'), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

y_n and $\hat{y_n}$ are tensors of predicted and actual charges with a total of n elements each.

2.3 Implementation and training

2.3.1 Data Handling and Preprocessing

Our given dataset consists of x and y coordinates that constitute the trajectory of the particle p_1 and the final charges (at last point of simulation) of particles p_2, p_3, p_4. The simulations comprising the particle's trajectory are sampled at $\Delta t = 0.1$, (i.e) 100 ± 10 steps resulting in a maximum duration of simulations of 110. But since our simulation data contains samples of uneven length (some of size 103, 107, 110, etc) and inputs of varying lengths cannot be feed into sequence based Neural nets, we need to convert them to have same length. All the missing positions (x, y coordinates upto 110 simulation timepoint) of the simulations are padded with zeros using Numpy's *pad()* to make the input length all equal. There were no other preprocessing required as the data provided were in the required format suitable for our case.

2.3.2 Model

We are using a LSTM model which is an RNN with provision of cell state to fit our problem formulation. Our LSTM based model architecture consists of a LSTM cell and two fully connected layers. Firstly, we initialize the hidden state and cell state with random inputs. The initial layer which is the LSTM takes an input of size 2 (the x and y coordinates of the particles at each timepoint in the trajectory simulation) and spits out a rewritten hidden state with a dimension of 256 and internally the new cell state is also created based on hidden state and input values. The hidden state of dimension 256 created is then passed through a fully connected layer to get an output of dimension 256. Which is then passed to another fully connected layer to obtain a final output of size 3. (which are the three charges of

particles $p_2, p_3 and p_4$) The LSTM cell has intial activation functions constituting of sigmoid and tanh functions and the first fully connected layer is provided with a ReLU activation. The second fully connected layer does not have an activation as we need the output(charge) as it is. An illustration of the architecture of our model can be seen in Figure 11.

2.3.3 Loss Function

In our model, a LSTM based architecture is used to predict charges and for this regression-like problem formulation, MSE loss is a good choice. The MSE calculation can be found here. (2)

2.3.4 Training the Network

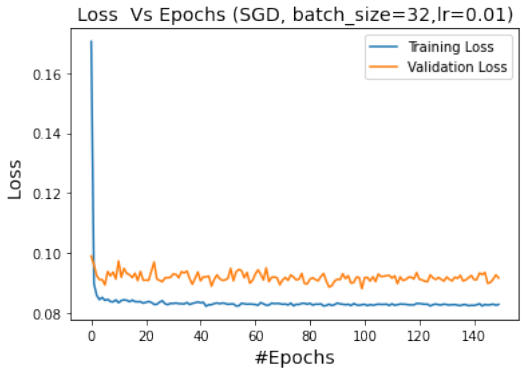

Figure 12: Training the model with SGD Optimiser

We initialize the network, loss function, and optimizer (SGD) with learning rate and momentum. To train the network, we pass batches of simulation where each simulation contains the x and y coordindates of each timepoint in simulation. At the first layer, which is the LSTM cell, the an input of size 2 (x and y) are passed and, the hidden state and cell state are generated and then hidden state is then passed to the fully connected layer which produces a fresh hidden state of dimension 256 with an activation function 'ReLU'. This hidden state is passed to the final fully connected layer which has no activation function and it produces an output consisting of three values which are the predicted charges of the stationary particles at the last timepoint of the stimulation. The architecture is demonstrated in Section 1.3.2. We then compute the loss between actual charge the three particles at the end timestamp with the charges the network spits out. After which the losses are backpropagated to calculate the gradients of our model followed by updation of the weights aided by SGD optimizer.

The average training loss is calculated by dividing the summation of the running losses by the size of the train_loader. Finally the train losses are stored for all epochs and plotted in Figure 12. We use the validation dataset to calculate validation loss of our model so that we can tune the hyperparameters and also to correct overfitting in network if any. After the experimentation explained in Section 2.4.2, we finalized SGD as optimizer with learning_rate= 0.01 and momentum = 0.9. We also used MSE loss as the loss function with batch_size of 32 for both train and validation loader. With these parameters, we got the Min_validation loss of **0.0898** at Epoch 146. We chose 150 as epoch size as we can see that losses started to converge and we are also not seeing any over-fitting happening in the model. This is currently the best result we have achieved.

2.4 Experiments and Discussion

2.4.1 Experimental Setup

For our problem formulation, we have options of different hyper-parameter settings for training the model that can fit. So, we tested them and analyze the results to finally

choose the best hyper-parameter combination obtained for testing purposes on the test data loader created from test dataset provided. The test data loader is kept out of the model implementation and training phases, and it is only used to forecast final locations once the model has been tuned. As explained previously in previous section 2.3.3 we use the Mean Squared Error(MSE) [1] suitable for our regression purpose.

We chose to analyze several of the most common regression optimizers in their default parameter settings, as provided by Pytorch, before deciding which one to employ and finetuning it accordingly.Except for Adam, who uses 0.001, we chose Adam[2], Adagrad [6], RMSProp [7], and SGD [3] with learning rate values of 0.01. For them, we also set the batch size to 32.We also kept the batch size consistent at 32 for them.As illustrated in Figure, their validation losses are tracked down. We then choose the best optimiser out of all of them, which is the SGD(explained in next section), and train the model against different batch sizes to see if there are any major variations in training performance.

Because our measure for assessment is the achieved loss, and we strive to decrease them as much as possible, it makes sense to experiment with multiple loss metrics on an optimizer independent from the final chosen optimizer. This extra work is performed on another optimizer to offer a distinct set of perspectives on the loss measures. As a result, we compute the average MSE, RMSE, and MAE loss on our data. The first two are L2 norm-based, whereas the last is L1 norm-based. These different experiments were evaluated based on metrics of evaluation such as best validation loss, worst validation loss, pattern of loss over epochs, etc.

2.4.2 Experimental Results and Analysis

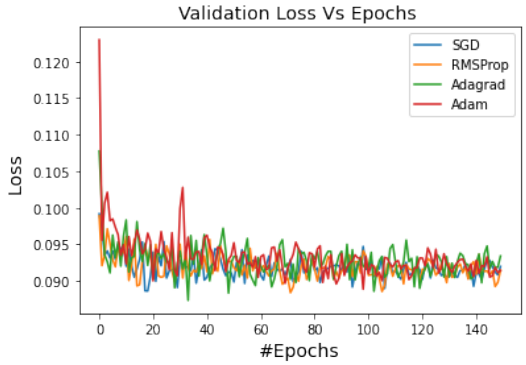

Figure 13: Training the model using different optimiser and batch sizes

It can be observed from Figure 13) that determining which optimiser performs the best is nearly difficult since their default parameter values are so close. This also reveals that, given our issue situation, using a different optimiser may not have a substantial influence. Adam's learning curve is exceedingly steep, and he converges in the same way as the others do. As a result, in order to make an impartial conclusion, we offer an instance of individual plots of Adagrad optimiser(Figure 14) where we can clearly see that there are high oscillations and the validation loss is unstable, and so we would want to exclude this optimiser from our testing. While Adagrad aims at gently scaling and annealing learning rates, optimisers with momentum like RMSProp speeds up learning along those gradient directions that are persistent during training [8].RMSProp also tries to dampen the oscillations which can be seen from the smooth and stable plot. RMS prop also takes away the need to adjust learning rate, and does it automatically. More so, RMSProp chooses a different learning rate for each parameter. Since RMSProp speeds up training, it shows better stability and is similar to the minimum validation losses we obtained for Adam and SGD. SGD with momentum also seems to find more flatter minima than Adam, while adaptive methods tend to converge quickly towards sharper minima.

We opted to tabulate SGD and Adam (Table 4 to demonstrate more clear interpretation and

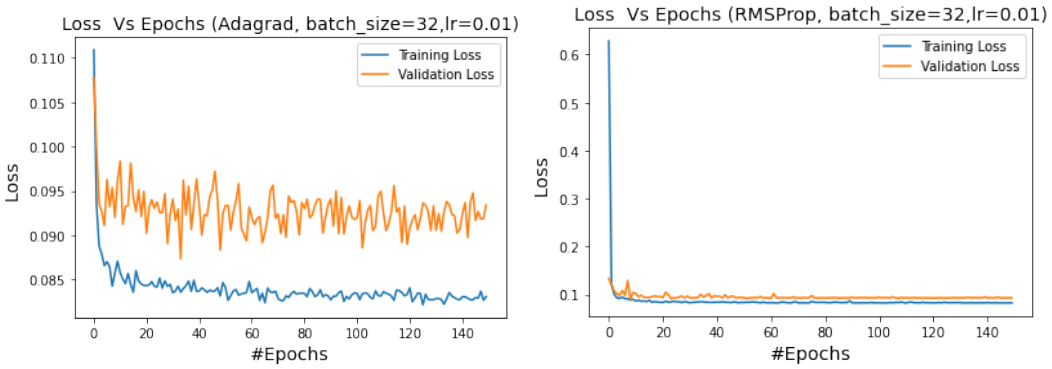

Figure 14: Training the model using different optimiser and batch sizes

so put down their best validation loss, worst validation loss, and average test loss acquired for prediction on the test data. The table shows that there is relatively little difference between them, however SGD may have fared somewhat better. As a result, we opt to use Stochastic Gradient Descent (SGD) for our final testing. Many deep learning works like [9] have been presented which show that in some experiments,Adam produced worse validation accuracy than SGD across all deep learning workloads considered.

The SGD method is then tested on different batch sizes (Figure 15) to see if there are any notable changes in performance. However, we can see that batch size has little effect, and we picked batch size 32, which had the lowest validation loss of all with a value of 0.0886. We also see that the loss converges quite quickly and does not improve much beyond 100 epochs, therefore we train it for low epochs only.

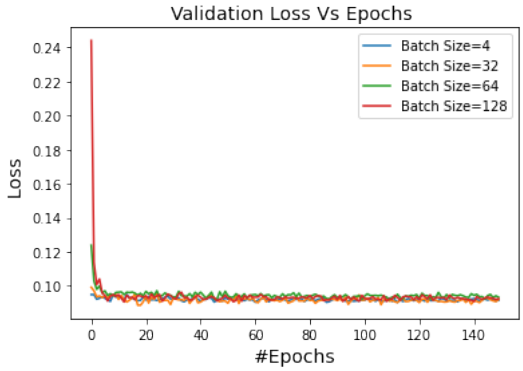

Figure 15: Training the model using different optimiser and batch sizes

Similarly, the batch size performances (Figure 15) are comparable, thus we chose a batch size of 32 since it has the lowest validation loss feasible (a value of 0.0886). Hence our final optimiser setting is :

1. **Optimizer** : Stochastic Gradient Descent(SGD) optimizer
2. **Batch Size** : 32
3. **Learning Rate** : 0.01
4. **Momentum** : 0.09
5. **Number of epochs** : 150

As previously stated, we intended to compare and analyze loss metrics on a different optimizer (Adam) than the chosen one (SGD) to provide an alternative and impartial metric evaluation(Table 5. For, our regression task, Mean Squared Error(MSE) performs the best

for predicting the charges while MAE performs the worst. MAE is less biased for higher values and thus for higher values it may not adequately reflect the performance when dealing with large error values. In case of non linear models,the MSE loss function is widely used as a non-convex loss function for estimating the parameters of nonlinear models such as in supervised neural networks[10]. One drawback of using MSE over MAE is that MSE is more sensitive to outliers than MAE[11]. Additionally, deep learning researchers suggest that for choosing a suitable metric for evaluating a regression model one should always look at the available data and target at hand [12, 13, 14]. Even though not practically tested, we assume our data does not have any outliers and hence we proceed with MSE loss.

2.5 Testing and Results

We test the model using the test dataset supplied to us, which has 100 simulations, using our final optimizer and hyper parameter settings. The batch size is set at 10 for ease of calculation and for no other scientific reason. We keep track of the average test prediction loss as measured by MSE Loss(0.252). The batch with the lowest loss and the batch with the highest loss are then extracted from the test data. We depict the simulations for the best and worst circumstances in order to understand why the charge forecast is excellent or terrible in each case.To demonstrate, we provide one example (Figure 16)from the batch of successful forecasts and one from the batch of bad predictions, noting the predicted charges and the associated real charges in each case.

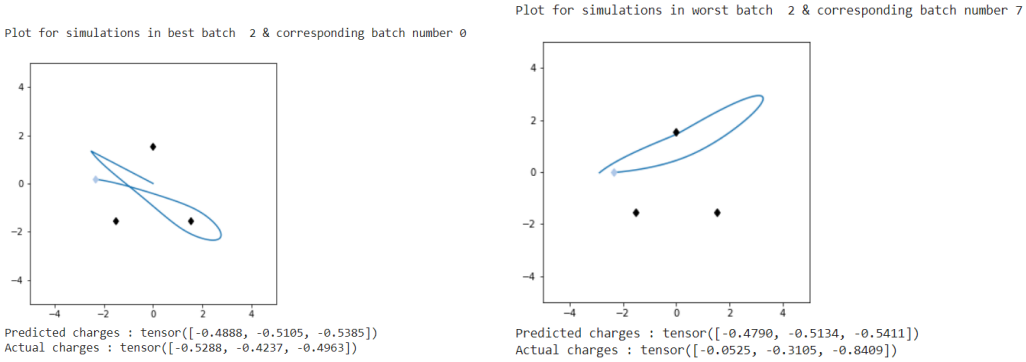

Figure 16: Plotting the simulations with the best and worst charge prediction(1)

On this topic, we explored the dynamics of a single particle p1 moving in a plane with three fixed charges c2, c3, and c4. We anticipated the other three charges based on the set trajectory of the charged particle. According to physics, the velocity of charged particles is influenced by both electric and magnetic forces. Though it is not clearly stated here how the charges interact with each other or how the three charges impact the trajectory of the particle, deep learning algorithms may actually approximate the link between charges and the trajectory of a particle present in that domain to some extent.

When we look at the plot for the best charge prediction, we can see that the discrepancy between the predicted and real charge is small, around 0.1. We can also observe that when the trajectory of the particle with the fixed charge happens close to the other charges, it produces a very excellent prediction because it can learn the link between its own velocity and the other charges. In the worst-case scenario, the input simulation of the particle

Metric	ADAM	SGD
Best Validation Loss	0.08861	**0.08826**
Worst Validation Loss	0.1229	**0.099**
Test Prediction Loss	**0.2506**	0.25108

Table 4: Adam(lr=0.001) Vs SGD (lr=0.01)

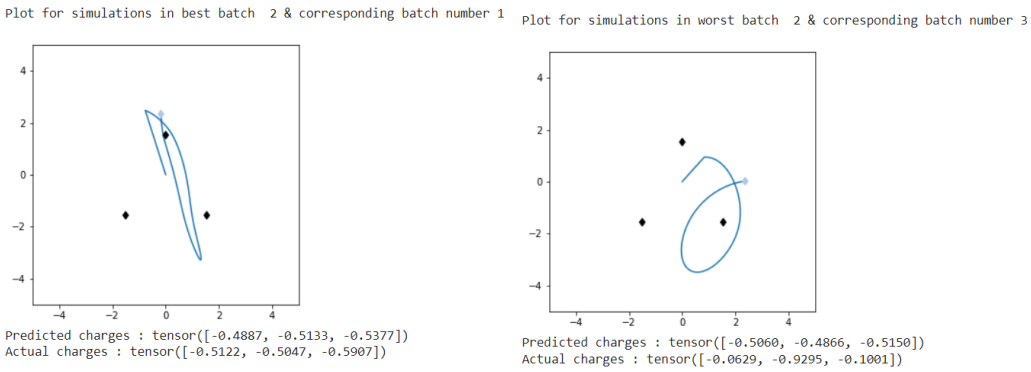

Figure 17: Plotting the simulations with the best and worst charge prediction(2)

is centered around a single charge, and so the model was unable to develop a very good relationship between the input and the target.

Looking at another set of examples in Figure 17, we can see that the motion should be someplace near the other charged particles for the best scenario to occur. Using the same instance, we can observe that the worst case forecast happened when the other charges were not very close to the trajectory. When the trajectory of the fixed particle was near to the other fixed particles, the model trained well throughout the different timesteps in our LSTM network. Hence using deep learning tasks, it is actually possible to learn the dynamics of different particles interacting with each other with more complex networks.

2.6 Conclusion

While deep learning approaches are being used in many multidisciplinary fields, their use in particle dynamics remains an unknown and exciting study topic. Our fundamental LSTM network captures good interaction between a particle's motion and other external charges. We believe that a more complicated LSTM network, such as stacked LSTMs or GRUs, might do even better. A possible future work is to extract feature embeddings from the initial input by using MLP layers and then feed the embeddings into networks like LSTM that can capture the time dependencies.

3 Task 3.2

3.1 Problem formulation

For assignment 3.2, we are tasked with building a solution to study the case of a single particle's trajectory and learn how to simulate a continuation of the particle's path. (i.e)We need a model that can predict the coordinates x and y of the path the particle p_1 would tend to take by looking at it's previous trajectory coordinates. Our task is to predict x and y coordinates of path of p_1 over an additional 4 ± 2 seconds, i.e. ending the entire simulation at $t = 14 \pm 3$. The dataset is generated by shooting a particle into the plane such that it can interact with the three fixed charges. The three charges are placed in a triangle configuration around the center of the plane while particle p_1 is shot in from a random position on the side of the plane, with an initial velocity pointing towards the approximate center of the

Losses	RMSE	MAE	MSE
Best Validation Loss	0.29	0.26	0.08
Worst Validation Loss	0.60	0.44	0.29
Test Prediction Loss	0.28	0.50	**0.24**

Table 5: Different Metrics on Adam Optimiser

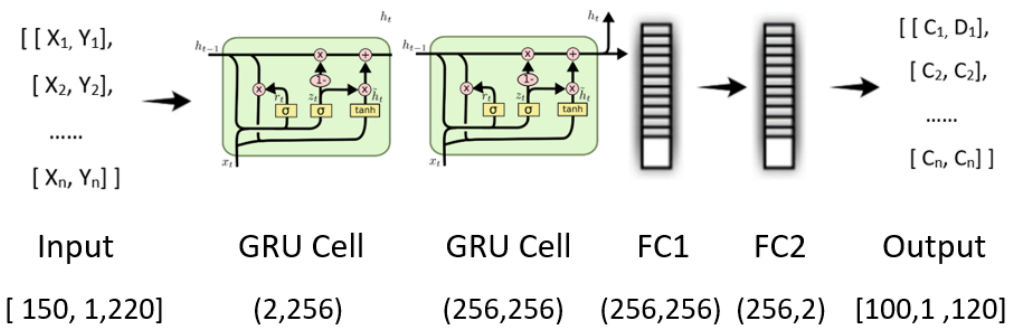

Figure 18: Model Architecture

plane. The generated simulations are of varying length. All simulations start at t = 0 and end at t = 10 ± 1. For this particular task, we also have a subset of simulations for an additional 4 ± 2 seconds, i.e. ending the entire simulation at t = 14 ± 3.The two pieces of the simulations are provided separately, such that one forms the initial path to be provided, and the other the path we wish to predict. We are provided with the simulations sampled with $\Delta t = 0.1$, (i.e) 100 ± 10 steps for the first part and an additional 40 ± 20 steps for the second part. We are provided with only positions ${x_1}^t \epsilon$ R of particle p_1. Finally, we have 150 simulations for training, 100 for validation and 100 for testing.

3.2 Model formulation

Here, since we need to build a model to that learns the coordinates x and y of the particle p at given timestamps such that it can predict the next 60 timestamps for the particle's trajectory based on path so far. In order to achieve this, we need the retain information regarding the trajectory route over the entire 110 (approximate) step length. This calls for requirement for memory retention over a relatively long period. LSTM and GRU, variants of RNN, are suitable for this task. Based on further model building and experimentation, our final model is based on GRU. Gated Recurrent Unit based Neural Net, is a varaint of LSTM where it combines both the cell state and hidden state found in LSTM into one state (the hidden state). GRU makes use of gates to regulate editing of information on the hidden state. Gates are neural nets which regulate the flow of information being passed from one time step to the next. This updation is done with different combinations of sigmoid and tanh functions. GRU has two gates: reset and updation gates. The reset gate uses point-wise tensor multiplication of combination of input and existing hidden state with a sigmoid output of the combination itself. This gates helps the GRU decide how much of the existing hidden state to forget. The updation gate also uses sigmoid along with tanh nets to decide what to retain from current input and current hidden state. GRUs are modified or lightweight version of lstm and it eliminates the need for the additional cell state in LSTM by combining the cell and hidden state to only hidden state that passes information along the network. The model architecture has been illustrated in Figure 18.

Our model architecture constitutes of two GRU cells followed by 2 fully connected layer that outputs 2 values (x and y coordinates of the next timestamp in trajectory). It uses Stochastic Gradient Descent (SGD) optimizer and Mean Square loss (squared L2 norm) function. The loss function can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = (y_n - \hat{y_n})^2 \tag{3}$$

where n is the batch size. If reduction is not 'none' (default 'mean'), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

y_n and $\hat{y_n}$ are tensors of predicted and actual coordinates with a total of n elements each.

3.3 Implementation and training

3.3.1 Data Handling and Preprocessing

Our given dataset consists of x and y coordinates that constitute the trajectory of the particle p_1. The simulations comprising the particle's trajectory are sampled at $\Delta t = 0.1$, (i.e) 100 $\pm$ 10 steps resulting in a maximum duration of simulations of 110. But since our simulation data contains samples of uneven length (some of size 103, 107, 110, etc) and inputs of varying lengths cannot be feed into sequence based Neural nets, we need to convert them to have same length. Also, as we are attempting to find the consecutive coordinates from the next timestamp after 110th step, it helps to have the last coordinate of the input data to be the actual location to have a smoother trajectory without breaks.So, the padding cannot be done with zeros like assignment 3.1. Instead we could pad the missing points with the last existing input that way it simulates the particle's last true coordinate with deviation. Hence, all the missing positions (x, y coordinates upto 110 simulation timepoint) of the simulations are padded with last coordinate available in that particular simulation's data using Numpy's *pad()*. There were no other preprocessing required as the data provided were in the required format suitable for our case.

3.3.2 Model

We are using a GRU model which is a LSTM with provision where it combines cell state and hidden state into just one hidden state to pass information through the network to fit our problem formulation. Our GRU based model architecture consists of two GRU cells and two fully connected layers. Firstly, we initialize the hidden state with random input. The initial layer which is GRU takes an input of size 2 (the x and y coordinates of the particle at each timepoint in the trajectory simulation) and spits out a rewritten hidden state with a dimension of 256. This hidden state is passed to the next GRU as input and it in turn spits out another output of dimension 256 as well. This output is then feed to a fully connected layer that takes in an input of 256 and spits out an output of 256 as well. Finally this output of dimension 256 is then passed to another fully connected layer to obtain a final output of size 2. The second GRU is run in a loop of length equal to the required number of timepoints and the output per iteration is recorded as it is the coordinates of the particle at that timepoint. The GRU has its own inner activation in the form of gates of combinations of sigmoid and tanh. The first fully connected layer is provided with a ReLU activation. The second fully connected layer does not need an activation as we need the output(coordinates) as it is. An illustration of the architecture of our model can be seen in Figure ¡3¿.

3.3.3 Loss Function

In our model, a GRU based architecture is used to predict the next coordinates (x, y) of the trajectory and for this regression-like problem formulation, MSE loss is a good choice. The MSE calculation can be found here. (2)

3.3.4 Training the Network

We initialize the network, loss function, and optimizer (SGD) with learning rate and momentum. To train the network, we pass batches of simulation where each simulation contains the x and y coordindates of each timepoint in simulation. At the first layer, which is a GRU cell, the an input of size 2 (x and y) are passed and the hidden state is generated and then hidden state is then passed to the next GRU cell which produces an output of dimension 256. This output is then passed to first fully connected layer which produces a fresh hidden state of dimension 256 with an activation function 'ReLU'. This hidden state is passed to the final fully connected layer which has no activation function and it produces an output consisting of two values which are the predicted coordinates of the particle at that

timepoint. The consecutive coordinates are pulled using a loop that passes the hidden state for as many iterations as further timepoints of the stimulation required. The architecture is demonstrated in Section 1.3.2. We then compute the loss between actual coordinate of the particle for each timestamp with the predicted coordinate for that timestamp that the network spits out. After which the losses are backpropagated to calculate the gradients of our model followed by updation of the weights aided by SGD optimizer.

The average training loss is calculated by dividing the summation of the running losses by the size of the train_loader. We use the validation dataset to calculate validation loss of our model so that we can tune the hyperparameters and also to correct overfitting in network if any. After the experimentation explained in Section 3.4.2, we finalized SGD as optimizer with learning_rate= 0.01 and momentum of 0.9. We also used MSE loss as the loss function with batch_size of 64 for train loader and validation loader. With these parameters, we got the Min_validation loss of **1.7389** at Epoch 346. We chose 350 as epoch size as we can see that losses started to converge and we are also not seeing any over-fitting happening in the model. This is currently the best result we have achieved.

3.4 Experiments and Discussion

3.4.1 Experimental Setup

For the given problem formulation, we require a model that has the capability to retain information over a longer period of time (i.e) long short term memory. Since there are two models that fit our criterias, LSTM and GRU, we built both and tested them out to select the one that best translates our requirement into a machine learning model. Since GRU is an improvement from LSTM to capture long-term dependencies, we select LSTM as our baseline model and compare it with GRU's prediction. We need to test different hyper-parameter settings for training the model created for our model implementation, analyze the results and finally choose the best hyper-parameter combination obtained for testing purposes on the test data loader. The test data loader is kept out of the model implementation and training phases, and it is only used to forecast final locations once the model has been tuned.As explained previously in equation (3) we use the Mean Squared Error(MSE) [1] suitable for our regression purpose.

For this problem, we avoid repetitious tasks such as determining which optimiser to employ and instead utilize SGD optimiser based on our conclusion from Task 3.1. However, we do experiment with different batch sizes to see if a little altered issue setting affects performance or not. To choose an appropriate batch size four model, we opted to analyze several batch sizes for our deep learning job, 32, 64, and 128 to see the best validation loss, worst validation loss, and pattern of behavior of the losses throughout the epochs for our experimental setup. As seen in the chart, a batch size of 64 appears to be most appropriate in our scenario.

In section 3.3.1, we explained why pad the empty places with the last coordinates of the simulation. We want to experiment what happens when we pad the empty places of the input simulation with zero and train it on the same network.

As our last experiment, we compare our model's performance with a simple linear baseline as mentioned in the assignment rubric.

Batch Size	Best Validation Loss	Worst Validation Loss	Average Testing Loss
Batch Size=32	1.51	**3.30**	1.30
Batch Size=64	**1.42**	3.42	1.29
Batch Size=128	1.88	3.32	**1.28**

Table 6: Comparison of different batch sizes in GRU

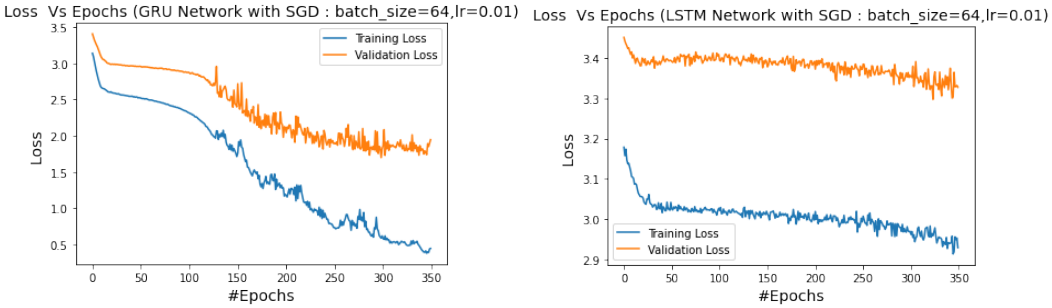

Figure 19: GRU vs LSTM

3.4.2 Experimental Results and Analysis

We plot the performance (Figure 19 of our LSTM network with our GRU network, and see a very bad performance from LSTM network. The model is unable to approximate the relation between it's current trajectory and the continued trajectory. The validation loss stagnates around 3.329 and does not improve further. According to the authors in [15], they observe that in general GRUs outperform LSTM networks on low complexity sequences while on high complexity sequences LSTMs perform better. Even though it is difficult to infer whether the complexity of our sequence is on the lower side or not, we may speculate that this is one of the reasons GRU outperforms LSTM greatly. We also know that due to the problem of the vanishing gradient, Gated Recurrent Unit (GRU), have been developed to enhance network memory to remember previous states and preserve long sequence dependencies. It has a gated structure to learn long-term relations of sequential data. For our input data, we have have long sequences of total length 220 and hence GRU performs well.The comparison by [16] demonstrates that when the quantity of training dataset is minimal, GRU beats LSTM in several other similar regression tasks like as time series forecasting, which matches our scenario and accords with their results.

In various other similar regression tasks like time series forecasting, [16] shows GRU outperforms LSTM when the amount of training dataset is small, which indeed matches our case and agrees with their conclusions.

We tabulate the performance of different batch sizes in Table 6 using GRU and observe that a batch size of 64 performs best. According to many papers that use sequence modelling using GRU like that of [17], most of them observed that batch sizes of 32, 64 and 128 show very close performance. For our problem setting too,they show very similar losses. Even though there is very little knowledge among even the deep learning practitioners to define the rule for choosing a batch size, the authors of [18] give a small but insightful explanation to it. According to them when the batch size is very small, the approximation will have very high variance, and the resulting stochastic gradient update will be very noisy and in contrast, when the batch size is very large, the batch gradient will almost exactly match the true gradient,and correspondingly two randomly sampled batches will have almost the same gradient. As a result, doubling the batch size will barely improve the update – we will use twice as much computation for little gain. Hence, we choose our batch size to be somewhere between and make it 64.

We experiment what happens if we pad the empty positions of the particles in the simulations with coordinates (0,0). Intuitively it is easy to understand that when the last position is (0,0), the model would find it difficulty to learn the continues simulations since it can no longer match with the last position the particle was previously in. Another intuition behind doing this experiment was the check if the loss decreases since in regression task all the zero values would multiply out thus decreasing the loss. As it can be seen from the Figure 20, the loss initially does start from a lower value than it did originally, however we also see that it does not learn well and the loss does not decrease much. Thus our experimental result for this matches with our intuition.

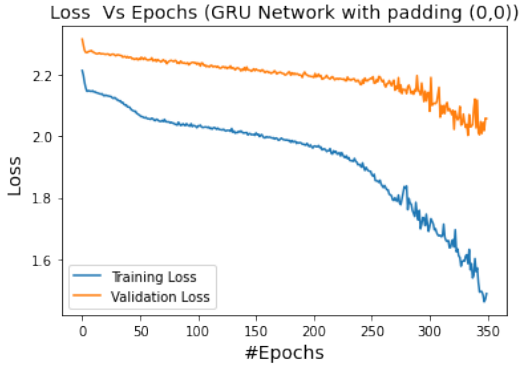

Figure 20: Performance of GRU using padding(0,0)

```
Average loss for linear baseline : 1.2266186475753784
Average loss for linear baseline : 1.098045825958252
Average loss for linear baseline : 0.7036644816398621
Average loss for linear baseline : 0.8553469777107239
Average loss for linear baseline : 1.410868525505066
Average loss for linear baseline : 1.4298721551895142
Average loss for linear baseline : 0.501125156879425
Average loss for linear baseline : 1.3904789686203003
Average loss for linear baseline : 0.6801984310150146
Average loss for linear baseline : 0.7997604608535767
Total average loss: 1.0095980167388916
```

Figure 21: Linear Baseline Model Prediction

As our last experiment, we create a simple linear baseline where we extrapolate in the direction between the last two timesteps of the input simulation.In other words, given the last two time steps t1 and t2 from the test simulation and it calculates the value of t1 + (t1 - t2) to give the prediction. We use a basic mean squared loss for evaluation and the obtained average loss for 10 batches of our testing data can be seen in Figure 21. The average loss for this linear baseline is 1.009 which seems to be better than our GRU model which has an average MSE of 1.819. This slight difference can be attributed to machine learning method's weakness in extrapolating data. Machine learning has typically been limited to interpolating data, or making predictions about a scenario that is "between" two other, known scenarios. Machine learning cannot extrapolate – that is, make predictions about scenarios outside of the known conditions – since it only learns to represent current data locally as correctly as possible. Thus extrapolation works well with linear and other types of regression.

3.5 Testing and Results

We train the model using SGD optimizer with hyperparameter settings of batch size 64, learning rate 0.01 and a momentum of 0.09 and plot down the trajectories of the continued simulations. Using these parameter settings, we test on batch sizes of 10 and we get an average MSE loss of 1.819. It shows that the model performs pretty bad when asked to predict the continued simulations.

We plot one instances of best continued simulation and one instance of worse continued simulation as shown in Figure 22. The red is the actual continued simulation and green is the trajectory predicted by our model. We can see that the in the best case, the model is able to somewhat correctly identify the direction of the continued trajectory even though it is far from being perfectly accurate. Among all the plots we obtained from the batch of test data that performed the best, we notice that the maximum it can do is correctly identify the direction of trajectory. In the case of worst batch of data, the model just learns the exact representation of the original trajectory. Though we were unable to infer in what cases or why our network is unable to capture the dependencies between the input and output simulations. A possible reason can be underfitting for this. Underfitting occurs when the network is unable to effectively represent the link between input and output variables,

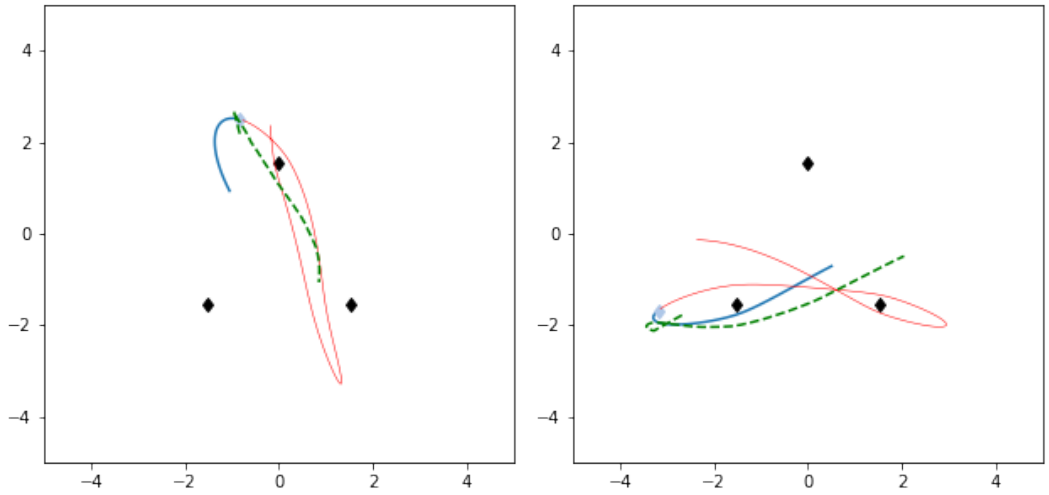

Figure 22: Best Continued Simulation vs Worst Continued Simulation

resulting in a high error rate on both test and train. A very bad test loss might be an indication towards it, however we cannot confirm it. Bad performance might be due to lack of enough training data for such a complex sequence generation regression problem. It also might be that our model is overly simplistic for this task and better feature extraction from sequences are needed.

3.6 Conclusion

In this job, we carried out a more sophisticated task from 3.1 in which we attempted to continue a particle's course for the following 60 time steps at most. The test loss was substantial, and the path predictions were not very accurate. We believe that model complexity is one of the primary causes of poor performance. We constructed a Gated Recurrent Unit (GRU), and while it outperformed the baseline LSTM model, we believe additional improvements are achievable. This challenge may also be framed as a Sequence to Sequence machine learning prediction, therefore a seq2seq model based on an encoder-decoder pair would have been an excellent choice.

We feel that extracting feature embedding in the early layers and then passing them to capture the time dependencies between them would have been a superior regression strategy, with similar reasoning as in 3.1.

References

[1] Mean MSE. Mean squared error: Definition and example, 2022.

[2] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[3] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, pages 462–466, 1952.

[4] Jeremy Jordan. Setting the learning rate of your neural network., Aug 2020.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[6] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. *Advances in neural information processing systems*, 30, 2017.

[7] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[8] Matthew D Zeiler, M Ranzato, Rajat Monga, Min Mao, Kun Yang, Quoc Viet Le, Patrick Nguyen, Alan Senior, Vincent Vanhoucke, Jeffrey Dean, et al. On rectified linear units for speech processing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3517–3521. IEEE, 2013.

[9] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[10] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[11] Davide Chicco, Matthijs J Warrens, and Giuseppe Jurman. The coefficient of determination r-squared is more informative than smape, mae, mape, mse and rmse in regression analysis evaluation. *PeerJ Computer Science*, 7:e623, 2021.

[12] Weijie Wang and Yanmin Lu. Analysis of the mean absolute error (mae) and the root mean square error (rmse) in assessing rounding model. In *IOP conference series: materials science and engineering*, volume 324, page 012049. IOP Publishing, 2018.

[13] Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.

[14] Tianfeng Chai and Roland R Draxler. Root mean square error (rmse) or mean absolute error (mae)?–arguments against avoiding rmse in the literature. *Geoscientific model development*, 7(3):1247–1250, 2014.

[15] Roberto Cahuantzi, Xinye Chen, and Stefan Guttel. A comparison of lstm and gru networks for learning symbolic sequences. *arXiv preprint arXiv:2107.02248*, 2021.

[16] Peter T Yamak, Li Yujian, and Pius K Gadosey. A comparison between arima, lstm, and gru for time series forecasting. In *Proceedings of the 2019 2nd International Conference on Algorithms, Computing and Artificial Intelligence*, pages 49–55, 2019.

[17] Ru Ni and Huan Cao. Sentiment analysis based on glove and lstm-gru. In *2020 39th Chinese Control Conference (CCC)*, pages 7492–7497. IEEE, 2020.

[18] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.