2AMM10 2021-2022 Assignment 2 & 3: Group 7

Ambarish Moharil Mert Lostar Kunal Geed

June 19, 2022

1 Task 2

1.1 Problem formulation

We study a system of n particles $p_1, p_2, .., p_n$on a plane evolving over time. At a certain time t, a particle i in the system is described by its position $x_i^t \in \mathbb{R}^2$, velocity $v_i^t \in \mathbb{R}^2$ and its charge $c_i \in -1, 1$. This system is only affected by particles present on the plane, and the force between particles depends on their charges and locations. For this task, we aim to investigate if we are able to predict the future state of a dynamic system containing n objects that interact with each other at time t. If we are able to accurately predict future states, we can skip steps during a simulation and do not have to sequentially compute each step. To measure the success of our model, we would consider the Euclidean distance between the actual and predicted position of a particle. We would aim to minimize this distance for each particle in the system. The data for this task was generated by initializing five particles at random positions and velocities. The positions are then updated at discrete time steps, which form the dataset used for this task. Formally, we consider the following task.

1. **Graph Embedding:** Our final goal is to build a model to accurately predict the positions of the n different particles at time t in the system given their initial positions and velocity at time $t = 0$. When given enough data to train on, the model should be able to adapt to any number of particles n and any arbitrary time t. With this aim in mind, we model each simulation as a graph using the initial information provided at $t = 0$ where each node represents a particle and every node is connected to every other node since each particle in our simulation setting affects every other particle. We then create node embeddings and use these embeddings to predict the positions of each particle, since each node represents a particle. Formulating the problem as a graph embedding problem, in theory, allows us to use the same model without retraining for a different number of particles since it just means adding more nodes to the graph. Furthermore, incorporating the time t we are predicting for into the initial node features allows us to make predictions for any t with the same model.

1.2 Model formulation

As stated in the section above, we are interested in devising models which, ideally, given the initial (at $t = 0$) positions $(x_1^0, x_2^0, \ldots, x_n^0)$, initial velocities $(v_1^0, v_2^0, \ldots, v_n^0)$ and the charges $(c_1, c_2, \ldots, c_n)$ of n particles $(p_1, p_2, \ldots, p_n)$, are able to predict the positions of these n particles at any time t in the future. With this goal in mind, we implement a graph based approach which we call PhysicsSAGE. It models the problem as a graph where each node represents a particle, and since each particle affects every other particle in the simulation, the graph of a simulation is fully connected. Using SAGEConv, which is the GraphSAGE layer from Hamilton et al. [2017], we then learn an embedding for the nodes and pass these embeddings through a fully connected linear layer to predict the positions at time t for every node in the graph as the output. Even though we were not able to test this hypothesis since we don't have data available for simulations with different numbers of particles, the graph based approach in theory allows us to scale the learned model to any number of particles without any need for additional retraining. Moreover, since the time horizon we

are predicting for is part of the initial node features, we are able to give predictions for any arbitrary time t.

The PhysicsSAGE architecture can be summarized as follows: The initial node features along with an edge index are passed into SAGEConv, which embeds ($N \times n \times input\_dim$) shaped tensors into ($N \times n \times hidden\_dim$), where N is the number of simulations in a single batch (batch size), n is the number of particles in the simulations and $input\_dim$ is the length of the input feature vectors for individual nodes and $hidden\_dim$ is the length of the embedded feature vector of each node. The embedding is passed through a drop out layer with a drop out rate of 0.3 to avoid over-fitting before being passed through the two fully connected linear layers. The first linear layer maps the embeddings to shape ($N \times n \times \frac{hidden\_dim}{2}$) and a ReLU activation function is applied before passing the output to the second linear layer to capture the non-linearity. The second layer then predicts the output positions for every node by mapping the output of the first linear layer to shape ($N \times n \times 2$).

1.3 Implementation and training

We first concatenate the initial positions, initial velocities, and charges of the particles and the time horizon. We are predicting the positions for this particular simulation into feature vectors of length 6. The input feature tensor, which is our initial node features, then becomes ($N \times n \times 6$) where N is the number of simulations (or samples) and n is the number of particles in the simulation. Since we are modeling simulations as graphs, we also need an edge index for each simulation. We create edge indices representing fully connected, where each node has a connection with every other node, and graphs with n nodes.

In order to be able to train the model on simulation batches of size B, we essentially combine edge indices and node features of B simulations to create a huge graph with sub-graphs (graphs of individual simulations) that are disconnected from each other. We train 9 models based on different combinations of training set sizes ($10^2, 10^3, 10^4$) and time horizons ($0.5, 1, 1.5$) each of these models share the same $hidden\_dim$ hyper-parameter choice of 192. During training, we report training loss and validation loss after validating the model on the provided validation set of 2000 simulations. We only save the model if there's any improvement in the validation loss.

1.3.1 Loss function

To train the model over the simulations in the training set, we make use of the Mean-Squared Error (MSE) loss. The is the squared L2-norm, hence it measures the square of the difference between the predicted value and the actual value. In our case, we get the square of the difference between the predicted position of particle and the actual position. We select a batch size of 10 and pass them through the network in order of their time horizon. From the batch, we make a prediction for the arbitrary time point defined in the instance. Let X(0) be the input graph for that particular instance. We are asked to predict the positions of the particles at particular time point (t). Hence, we describe the MSE loss to be the $(X_n(t) - X_n^{'}(t))^2$, where $X_n(t)$ is the actual positions of the particle n and $X_n^{'}(t)$ is the predicted position of that particle. As we are predicting the position of multiple particles, we average the loss for each particle to get the MSE loss for that instance.

1.3.2 Optimization

To minimize the loss we make use of standard back-propagation to update the weights of the network. Specifically, we make use of mini-batch stochastic gradient descent a learning rate (l) of 0.01 and a momentum (η) of 0.9. The momentum helps accelerate the gradient vectors (weights) in the correct direction and hence achieve convergence faster. We use the default update rules as described on `PyTorch`, which were adapted from the works of Sutskever et al. [2013]. The updates rule for each epoch (E) is shown in equation 1 and 2.

$$v_{t+1} = \eta * v_t + g_{t+1}(\theta_t) \tag{1}$$

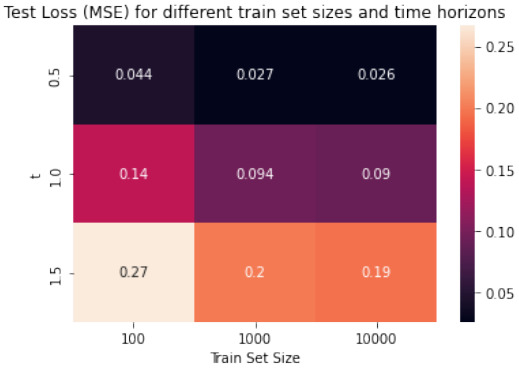


Figure 1: MSE Loss reported for the trained models on the test set

$$\theta_{t+1} = \theta_t - l * v_{t+1} \tag{2}$$

In equation 1 and 2 $g_{t+1}(\theta_t)$ is the gradient of the loss function at θ_t.

1.4 Experiments and results

We compare our models with a simple linear baseline which tries to predict the position of a particle at a target time horizon t based on the initial position (x_i^0) and initial velocity (v_i^0) with the following equation

$$x_i^t = x_i^0 + v_i^0 t. \tag{3}$$

In our first experiment setting, we evaluate 9 different models trained on varying values of train set sizes and time horizons on the provided test set of 2000 simulations. If a model is trained using a specific time horizon, t then we also evaluate it on the test set by trying to predict the position of the particles at that time horizon t. We compare these models based on MSE loss. You can see the performance of each of these 9 models in Figure 1. The results show that performance improves if the model is provided more training samples and decreases the further in time we try to predict from the initial state. Of course, we need a baseline to compare these loss values. Thus, we also make predictions on the test set for all three time horizons using the simple linear predictor and report the MSE loss. The results can be seen in Table 1. The results behave similarly to our own model, the performance deteriorates the further we try to predict from the initial positions. However, we see that all of our models, regardless of training set size and time horizon we predict for, outperform the baseline. Furthermore, the difference between the performance of the baseline and our model increases further when we increase the time horizon we are predicting for. Which makes sense, as the longer we simulate, the more the particles interact with each other.

t	0.5	1	1.5
Loss	0.05	0.25	0.6

Table 1: MSE loss reported on the test set by the Simple Linear Baseline for different time horizons

In our second experiment, we test our model's ability to predict for time horizons it did not see during training. We do this by training a model using a time horizon $t - 0.5$. However, then evaluating the model on the test set by trying to predict for time horizon t. Once again, we test different combinations of training set sizes and time horizons, six models in total, as we can't do this for $t = 0.5$. The results can be seen in Figure 2. The results behave similarly to our previous experiment. The performance improves if we have access to more

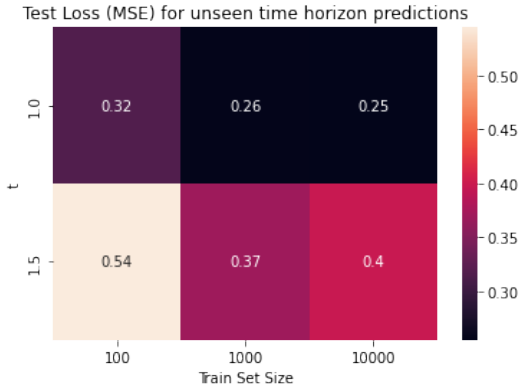


Figure 2: MSE Loss reported for models trained on t-0.5 and evaluated for t on the test set

simulations during training and deteriorates the further into the future we try to predict. The results prove that even if we don't have data available for a specific time horizon, if we have access to enough data for previous time horizons, we can make predictions that are more accurate than the simple baseline.

1.5 Conclusion

In this section for task two, we introduced a problem where we tried to predict the future state of a dynamic system containing n charged particles that interact with each other at time t. Our approach for the solution of the problem was modeling it as a graph embedding problem, with the goal of being able to adapt to simulations with different number of particles and the ability to make predictions for any time horizon t. In our experiments, we showed that our approach was better than the baseline linear predictor, which completely ignores the influence particles have on each other, for all the different combinations of the experiment setting. Furthermore, we were able to prove that our model was able to adapt to time horizons that were not observed during training and still perform better than the baseline given enough training data, even though there was a decrease in performance. We were unable to test our hypothesis of the model being able to adapt to simulations with different numbers of particles due to the lack of such simulation data. However, in theory, it would only mean adding extra nodes to the simulation graph. Thus, we are confident that the model could adapt to such situations.

2 Task 3.1

2.1 Problem formulation

In this task, we study the trajectory dynamics of a single particle moving on a plane. The plane also has three fixed particles with charges c_2, c_3, c_4 in a triangle configuration at the center of the plane. At a certain time (t) the system dynamics are described using the position of the moving particle, fixed particles and the charges of the particles. In this system, the movement of the free particle over time is only affected by its interaction with the fixed charged particles. Contrary to task 2, where we assumed that charges are from the set $-1, 1$, in this task the charges of the free particle is fixed to 1 and the charges of the fixed particles are from the *interval* $[-1, 1]$. In this task, we aim to predict the charges c_2, c_3, c_4 of the fixed particle by analyzing the trajectory of the free moving particle. This task is investigating if we can build a model that is able to extract information about the properties of the simulation, given that it has only been trained on existing simulations with known properties. To measure the success of our model, we would consider the difference between the charges predicted by our model, based on the particles' trajectory, and the actual charges. For this task, the data is generated by shooting the free particles into the plane with the fixed charges. The movement of the particle is then simulated for a varied

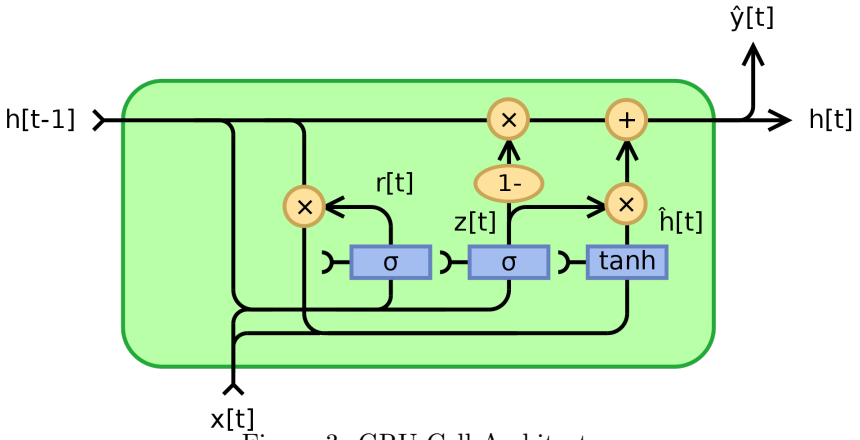


Figure 3: GRU Cell Architecture
source

length of time, and these simulations, along with information about the fixed charges, are provided as a dataset. Formally, we consider the following task:

1. Predicting Parameters: We aim to build a model that is able to predict the parameters of the system based on the trajectory followed by the particle. Our goal is to build a deep learning model that is able to accurately predict the charges of the fixed particle in the system that is able to explain the trajectory of the free particle.

2.2 Model formulation

As stated in the problem statement above, we study the dynamics of a particle system as a goal of this task. We have an unbounded moving particle ($p1$) which is fired in a system of three distinct particles ($c2, c3, c4$) with fixed charges ($c_i \in [-1, 0]$ and $i \in [2, 4]$) separated apart from each other by a fixed distance in a triangular plain. The goal of the task is to study the trajectory of the particle ($p1$) and predict the fixed charges c_i corresponding to the respective particles p_i with $i \in [2, 4]$. We have been provided with a training set of 800 simulations (every simulation consists of a different initial position at $t = 0$). The particle follows a sequence of positions from $t = 0$ to $t = 10 \pm 1$ with a step size of $\Delta t = 0.1$. Implying a sequence of positions of length 100 ± 10 for every simulation of the particle $p1$. This implies that the position of the particle $p1$ at time $t, t \in [0, 10]$ depends directly on the position of the particle $p1$ at time $t - 1, t - 2..t_0$. Hence, the past information is extremely necessary to capture the correlations between the positions of the particle $p1$ at any given time $t, (t \in [0, 10])$ in any given simulation of the system. Hence, we need to formulate a model that is robust in capturing this time dependency and historical information of a given sequence over some time period T. We model our architecture as a Recurrent Neural Network (RNN) that takes in a time sequence as an input, learns the mapping between the trajectory of the particle $p1$ and the three fixed charges $c2, c3, c4$ in a given simulation and then predicts these charges based on the learned embeddings of the trajectory of the particle $p1$. To compare the performances of the models and to get an idea about the relative performances, we model 3 distinct architectures to capture the stochastic process that governs this system of particles:

1. GRURegressor
2. ConvGRURegressor
3. Benchmark Multi-layer Perceptron (MLP)

2.2.1 GRURegressor

We model a Recurrent Neural Network consisting of a Gated Recurrent Unit (GRU) cell. The unrolled network can be seen as an architecture consisting of L layers sequentially

stacked one after the other on a horizontal axis. In general a RNN consists of a hidden vector h and an output y (this output is optional depending on whether we need an output at every time step t of the input sequence) which operates on a variable input sequence $x =< x_1, x_2, x_3.....x_t >$.

$$h_t = f_x(h_{t-1}, x_t) \tag{4}$$

where f_x is a non-linear activation function. A RNN can learn the distribution over the input sequence and can predict the next positional co-ordinate given the current position and the contextual information in the form of a hidden state from the previous time step. This d dimensional hidden vector can then be passed through another non-linear activation function (ReLU or Sigmoid) to learn the distribution over the target charges. We will now look into the information flow within the Gated Recurrent Unit (GRU) cell. We make use of different gates that regulate the information flow within the GRU cell an to compute the current hidden state vector h. r is the reset gate, z is the update gate within the GRU cell. Let x_t be a set of positional co-ordinates ($x_t \in \mathbb{R}^2$) at any time $t, t \in [0, 10]$, then the activation of the reset gate at time t can be given as:

$$r_t = \sigma(W_r x_t + U_r h_{t-1}) \tag{5}$$

where W_r and U_r are the shared parameters for the reset gate and σ is the sigmoid activation function. Similarly, the activation of the update gate at time t can be given as:

$$z_t = \sigma(W_z x_t + U_z h_{t-1}) \tag{6}$$

where W_r and U_r are the shared parameters for the update gate. The hidden state vector at time t in a GRU cell is computed as:

$$h_t = (1 - z_t)h_{t-1} + z_t \tilde{h_t} \tag{7}$$

where

$$\tilde{h_t} = \tanh(W x_t + U[r * h_{t-1}]) \tag{8}$$

$*$ is the Hadamard product. We compute the hidden state vector at every time step over the input sequence x. The idea is to learn a set of shared parameters over a time sequence T and capture the historical dependency. It is to be noted that the parameters that the model learns are U, U_r, U_z, W, W_r, W_z. These are fixed parameters and can be said to be shared by all the time steps in the given input sequence. This is a major advantage as it restricts the number of parameters to be learned over time and these parameters capture the contextual or historical dependency over the time T. The sigmoid function used in the formulation of equation 5 acts as an information barrier. It blocks all the inputs that map to 0 and allows only the flow of relevant information. This is helpful in handling the problem of vanishing gradient when back-propagating the loss updates while training the network. This also allows the hidden state h_t to drop any past information that is irrelevant in the future and keep its representation compact. Furthermore, the update gate z in equation 6 controls how much information in the past (from the previous hidden state) should be carried over to the current hidden state. As the reset and update gates are separate for every hidden state, each hidden unit will learn to capture the dependencies over different time scales. Hidden units that learn to capture the short-term dependencies will have reset gates that are active, and those units that capture the long-term dependencies will have the update gates quite active. Once we have the hidden vector h_t $[10 \times 256]$ (batch_size = 10 and hidden_dim = 256) in the input sequence $x =$, we pass it through a linear layer in order to map it on the target domain. The linear layer is as follows:

$$y = xA^T + b \tag{9}$$

The output of the first linear layer (10×256 dimension) is then passed through a Rectified Linear Unit (ReLU) function to capture the non-linearity in the mapping. Furthermore, the output of the ReLU layer (10×256 dimension) is then again passed through another linear layer to finally map the input sequence to the targets. This output of the last linear layer is the prediction of the model for the the respective charges c_i of the particles p_i where $i \in [2, 4]$ and has the dimension 10×3 or in general $batch\_size \times 3$.

2.2.2 ConvGRURegressor

Gated Recurent Unit's (GRU's) are theoretically robust in handling long term dependencies due the gating mechanism. The reset gate r and the update gate z control the information flow by dropping irrelevant past information with respect to the current input and also capturing long term relevant dependencies given the current input. When learning a distribution over an input sequence, it becomes interesting to investigate whether the model can capture these long term and short term dependencies from distilled or filtered input sequences. If we summarize the given input sequence $x =< x_1, x_2, ...x_t >$ using relevant mathematical operations, can the GRURegressor learn these dependencies remains an interesting question to investigate. As stated previously, the problem at hand is a many-to-one mapping problem where the input sequence x is condensed into a hidden vector h_t, which is used in predicting the target charges. As the input sequence grows larger and larger, it becomes inquisitive to understand whether the hidden vector h_t still compactly represents the short and long term dependencies at every time step t in the input sequence x or we are dumping too much historical information into a single hidden vector of d (256) dimensions that has lost it's relevance given a large input sequence. To summarize the given input sequence of positional co-ordinates x_i such that $x_i \in \mathbb{R}^2$ and $x =< x_1, x_2, ...x_t >$ where $i \in [0, t]$, we apply $1D$ convolution on the input sequence x. As stated previously, let the length of the sequence x be L and the *batch_size* be N. Let C be the channels or filters selected for the convolution task. In our architecture we set the number of input filters $C_$in to 64 and the number of output filters C_{out} to $C_{\text{in}} \times 2$ i.e 128 . Let k define the kernel size selected for the convolution operation. We set the kernel size k to 4 in our architecture. The output of the convolution can be given as follows:

$$\text{out}(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{\text{in}-1}} \text{weight}(C_{out_j}, k) \star \text{input}(N_i, k) \tag{10}$$

where $\star$ is the cross-correlation operation. It can also be regarded as the *sliding dot product* or *sliding inner product*. The *stride* controls the stride for the cross-co-relation, where the input to the function can be a single element or tuples. In our model architecture, we set the *stride* to 4 which indicates that we take four positional co-ordinates $(x_i, y_i), (x_{i+1}, y_{i+1})$ to calculate the cross-correlation and filter down the input sequence x. The output of the $1D$ convolution then follows as an input to the GRURegressor explained in section 3.1.1. Remaining model architecture is same as that of the GRURegressor stated above.

2.2.3 Multilayer Perceptron

In order to stress on the time dependant nature of the input sequence x and on the importance of capturing this property, we model a Multilayer Perceptron Feed Forward Network that maps the input sequence x to the corresponding charges for the respective simulation. We design a feed forward netwrok that has 2 hidden layers and 1 output layer and each layer consists of a single neuron. The neurons present in the hidden layer apply a ReLU (Rectified Linear Unit) activation on a linear mapping of the input sequence. The input sequence is fed as a $1D$ array consisting of $2 \times L$ elements, which is a flattened array of x and y positional co-ordinates of the particle $p1$. The first hidden layer applies a linear function (equation 9). The dimension of this input vector to the neuron of the first hidden layer is $[N \times 1 \times L]$ or $[N \times L]$, where N is the batch size and L is the sequence length. Let H be the hidden dimension of the linear layer, so the output of the linear layer has the dimension $[N \times H]$. The ReLU activation looks as follows:

$$ReLU(x) = \max(0, x)$$

The ReLU function captures the non-linearity of the process being modelled.The dimension of the output of the ReLU function is $[N \times H]$. This output vector is then fed to the neuron of the next hidden layer, which also applies a linear transformation followed by a ReLU activation. The output vector of the second hidden layer has the dimension $[N \times H/2]$. The output layer too consists of a single neuron which applies a linear transformation (equation

9) over an output dimension of 3. This output layer maps the hidden input vector to the target domain of the corresponding charges of the particles $c2, c3, c4$. The dimension of this output vector is $N \times 3$. It is to be noted that this model is a baseline model and our conjecture or hypothesis for the model is:

H_{0_1}:- The baseline MLP Feed Forward Network fails to capture the time dependencies and performs worse than that of the *GRURegressor* and *ConvGRURegressor*.

2.3 Implementation and training

2.3.1 Loss function

To train these models we make use of the MSE loss, similar to Assignment 2. We choose MSE loss as the target value we are trying to predict is continuous in the range $[-1, 1]$ and hence our model also outputs a continuous value. We train the model using predefined batch sizes. MSE loss works well for this problem as intuitively we are measuring the square of the difference between the predicted value and the actual value of the target. Hence, a lower difference indicates that our predicts our better and a bigger distance indicates they are worse. Formally, this means that for each charged particle n in the system, we predict the charge $c_n^{'}$ and have the actual charge c_n. Therefore, the MSE loss for this particle would be $c_n^{'} - c_n$. As there are multiple particles in a single data point, the loss takes the form shown in equation 11:

$$L(x, y) = \frac{\sum_{i=1}^{n}(x_i - y_i)^2}{n} \tag{11}$$

2.3.2 Optimization

To minimize the loss we make use of standard back-propagation use SGD with a learning rate (l) of 0.01 and momentum (η) of 0.01 with Nesterov momentum enabled. Compared to normal momentum, where we select a velocity and make a (big) step in that direction with momentum, in Nesterov momentum we first make a step in the direction of the initial velocity and then change the direction based on the new location. We use the default update rules as described on `PyTorch`, which were adapted from the works of Sutskever et al. [2013]. Hence, the update rule for Nesterov momentum is shown in equation 12 and 13.

$$v_{t+1} = \eta * v_t - g_t(\theta_t + \eta * v_t) \tag{12}$$

$$\theta_{t+1} = \theta_t + l * v_{t+1} \tag{13}$$

In equation 12 and 13, $g_t(\theta_t + \eta * v_t)$ is the gradient of the loss function at $\theta_t + \eta * v_t$.

2.3.3 Weight Initialization

In the models described for task 3.1, three different kinds of layers are utilized. A linear layer, a 1-D convolutional layer (Conv1D), and a gated recurrent unit (GRU) cell. For the linear layers, the each weight is initialised from a uniform distribution $U(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{in_feature}$r. Here $in\_feature$ stands for the size of the input sample. For the Conv1D layer, the weights are initialzed from a uniform distribution $U(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{in\_feature}$, where $k = \frac{groups}{C\_in * kernel_size}$. Finally, we have the GRU cell, for which the weights are initialized from a uniform distribution $U(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{hidden\_size}$. $hidden\_size$ is the number for features in the hidden state [Paszke et al., 2019]. These are all default approaches for initializing weights used by `PyTorch`.

2.3.4 Learning Schedule

Given the optimizer, weight initialization and loss function for the task 3.1, we now describe the learning schedule that we follow in order to train our formulated models for the given task. We set a constant learning rate of 10^-2 across all the epochs in our training. We follow

this same learning rate for all the models formulated for task 3.1. No learning rate annealing was performed while iterating over the epochs. The momentum (η) was set to a constant value of 0.9 for the all epochs while training the models for the task 3.1. Further, we set the batch size (N) to 10. The training set comprises of 800 simulations. Each simulation consists of the position of the particle $p1$ from $t = 0$ to $t = 10 \pm 1$ with a step size of $\Delta t = 0.1$. This means that the simulations in the training have varying sequence lengths L, such that L varies between 90 and 110. To tackle the varying sequence lengths L, of the particle $p1$, we perform zero padding over the input sequence. We identify the maximum sequence length L in the 800 respective simulations and then pad the length difference of each sequence with 0's. This makes the length of the input sequences uniform across the training set. We do the same for the simulations in the test set and the validation set. We iterate over the training set in mini-batches of 10. For every mini-batch we compute the output from each of the formulated models. In this case, the output is the predicted charges for particles $c2, c3, c4$ and has values $\in [-1, 0]$. Thus, for every mini-batch in an epoch, we calculate the mean squared error loss between the predicted and actual charges of the particles $c2, c3, c4$. It is to be noted that we only compute the output at the last time step of the input sequence L and not at every consecutive time step (for *GRURegressor* and *ConvGRURegressor*).The loss computed at every mini-batch k, ($k \in N$) is fed to the optimizer which then takes a step to update the weights and attempts to reach the minimum of our convex loss function. Furthermore, we sum the loss corresponding to every mini-batch in our training set and average it out over the total number of mini-batches in our training set to get an *overall average training loss*. A similar regime is performed over the validation set. The validation set consists of 100 simulations of the trajectory of particle $p1$ from $t = 0$ to $t = \pm 10$, $\Delta t = 0.1$. The batch size for the validation set is also 10. We again compute the loss for every mini-batch in the validation set but we do not feed this loss back to the network through the optimizer to make the relevant gradient updates. The validation loss of the network acts as an evaluation metric which helps us in measuring the learning ability of the network. Ideally, we stop the learning schedule when the validation loss of the network stops improving.

2.4 Experiments and results

In our experimentation, we train our model on the training set of 800 simulations and validate the training on a validation set that consists 100 simulations. The formulated problem is a regression problem as the labels or the target charges $c2, c3, c4$ possess continuous charges $\in [-1, 0]$. The evaluation metric used to analyze the performance of the model is the *mean squared error loss* during the training and validation. For every mini-batch k in the training set, we keep track of the mean squared error loss and average it over the N batches sampled in an epoch E. This is the average training loss which is computed over the train set. Similarly, we compute the average validation loss (referred as validation loss further) over the validation set while training the model. The validation loss acts as a metric that defines the length of our training or the number of epochs for which we should ideally train our model. Once, the validation loss stops improving for approximately 20 epochs, we stop our training regime and save the model with the best validation loss until the training is stopped. We train our models for 400 epochs using the standard stochastic gradient descent optimizer and evaluate the training and the validation loss. For the *GRURegressor* we get the results for validation and on our test data with the learning rate $l = 0.01$ and the nesterov momentum $\eta = 0.9$. Similarly, for the *ConvGRURegressor*, we obtain the results by training the model with the same values of l and η as stated above. To avoid overfitting of the model, we add regularization layers in the network architeture, where we set the dropout rate at 0.5. We also use batch normalization in the convolutional layers and the fully connected layers of the Multilayer Perceptron to stabilise our training regime. Finally, we get the results of our experimentation for the benchmark Multilayer Perceptron network using the $l = 0.01$ and η as 0.9.

From Table 2 and figures 4, 5 and 6, we find that the ConvGRURegressor model is the least overfitting model and generalizes well to the test data as compared to the other two models.

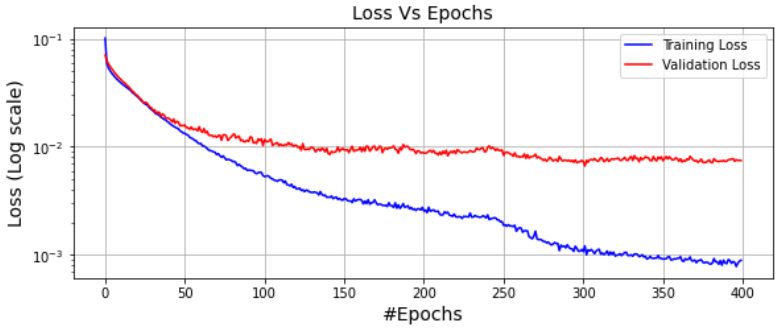


Figure 4: Training and Validation Loss ConvGRURegressor

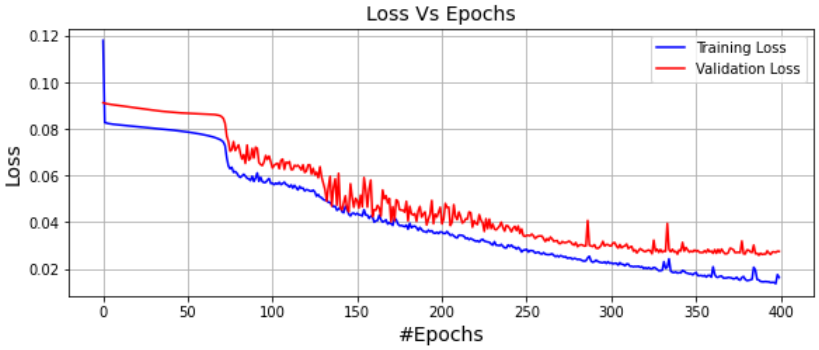


Figure 5: Training and Validation Loss GRURegressor

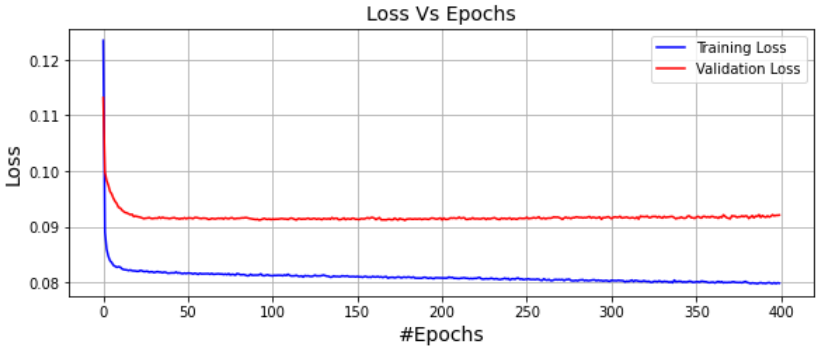
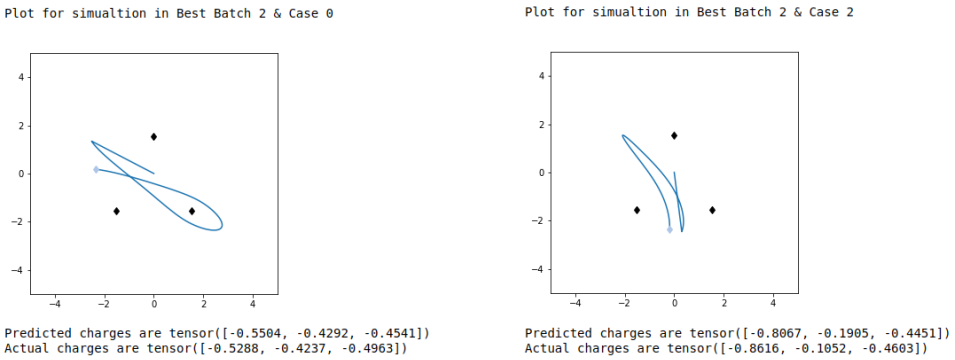


Figure 6: Training and Validation Loss MLP

Table 2: Average Validation Loss and Test Loss

Model	**ConvGruRegressor**	**GruRegressor**	**MLP**
Best Validation Loss	0.00663	0.025780	0.09115
Average Test Loss	0.008431	0.024177	0.08489

The best validation loss (for a batch $\tilde{k}$ in the validation set) for the ConvGRURegressor is **0.00663** and the best test loss is also provided by the ConvGRURegressor, which is **0.008431**. From the experimentation, we can confor to the hypothesis that $1D$ convolution summarizes the long sequence quite well and extrapolates the observed correlation between the sequential time positions of the moving particle $p1$. After conducting the experiments we now analyze the individual cases where we study the trajectory of the particle $p1$ and predict the respective charges $c2, c3, c4$ of stationary fixed particles $p2, p3, p4$. We will make this analysis only for the ConvGRURegressor as it was found to be the best performing model. As stated above and from table 2, we find the avaerage test loss to be **0.008431**.Furthermore, we report the individual batch losses for each batch present in the test set. We find the best



(a) Case 0 Batch 2

(b) Case 2 Batch 2

Figure 7: Simulations from Best Batch 2

batch in test set to be batch 2 and the relatively worst batch to be batch 3. The following table summarizes the best and the worst batch losses. From Table 3, we see that the batch

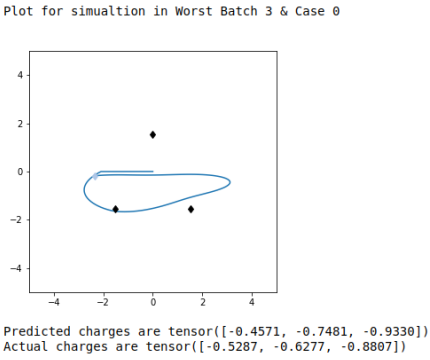
Table 3: Batches with The Best and the Worst Test Loss

Batches	**Best Batch : Batch No** 2	**Worst Batch : Batch No** 3
Batch Test Loss	0.00355	0.01442

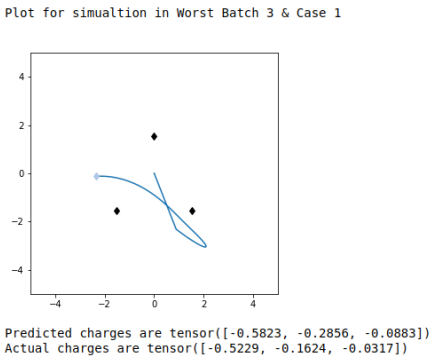
2 has the lowest test loss. Looking at two of the individual cases from this batch we find that the ConvGRURegressor actually does a good job in predicting the charges $c2, c3, c4$ of particles $p2, p3, p4$. The model captures the non-linearity of the input positional sequence of the particle $p1$ and learns a fairly good distrbution over the 3 dimensional output space of the charges figure 7. When we take a look at the worst predictions in the batch, it can be seen that the predictions are not off by much and the relative errors are quite low. We could still argue that this relative worst performance is due the model still over-fitting on the input data to a certain extent. For the worst batch 3 in our test set, it can be seen that for most of the cases, the model overestimates the charge $c2$ and under-estimates the charges $c3\&c4$. Our assumption while formulating the charges was to weigh all the three charges equally or give equal importance to the prediction of all three charges. The model seems to weigh the charge $c2$ slightly more than that of the other two charges, although this is the case only in batch 3 and this trend doesn't necessarily hold for the other 9 batches in the test set. One way to solve tackle this bias would be to penalize the prediction for the charge $c2$ and pull it down while pushing more importance for the charges $c3, c4$.

2.5 Conclusion

In this section for task 3.1, we introduce a problem where we study the trajectory of the particle $p1$ from time $t = 0 \text{to} t = 10$ with a step size of $\Delta t = 0.1$. The aim of the task was to predict the charges $c2, c3, c4$ corresponding respectively to the particles $p2, p3, p4$ based on the trajectory of particle $p1$, which was fired into this triangular plane of three fixed charges with an initial velocity $v1$ at time $t = 0$. In order to capture this time dependant nature of the positions of the particle $p1$ we provided three architectures, where two are the implementation of Recurrent Neural Networks (RNN's) and the third is a simple baseline Multilayer Perceptron. From the experimentations performed in this section, we show that the Gated Recurrent Neural based model's capture the trajectory and time-dependant nature of the positions of the particle $p1$ quite well and predict the charges of the fixed particles $p2, p3\&p4$ quite accurately. Inn the experimentation's stated in this section we also show that $1D$ convolution quite robustly summarizes a long input sequence and captures the correlation with minimum information loss, as it performs better than the Vanilla GRURegressor and the baseline Multilayer Perceptron. The Multilayer Perceptron doesn't account for the $1D$ highly structured grid structure of the data and as per our assumption performs worse than



(a) Case 0 Batch 2



(b) Case 2 Batch 2

Figure 8: Simulations from Worst Batch 2

both the RNN models. Finally, we conclude that it is possible to infer information about a system with one particle moving in a field with charges, using *relevant* deep learning methods.

3 Task 3.2

3.1 Problem formulation

For task 3.2, we follow up on the same setting described in section 2.1 for task 3.1. In this task we aim predict the trajectory of the free particle for a limited time based on its past trajectory. Specifically, we aim to build a model that is able to use the positions of the free particle during a simulation and continue its trajectory for an additional seconds. The model aims to learn the dynamics of the system based its initial simulation and continue it with high accuracy. Generally, this task has many applications as we can build a model that is able to observe the trajectory of a particle in a certain system and predict how it will evolve in the future. The data for this task is generated in a similar manner as task 3.1. For task 3.2, the simulations are continued for an additional (varying) time period. These two simulations (original and additional) are provided seperately to form the dataset for this task. Formally we consider the following task.

1. Forecasting: We aim to predict the build a model capable of understanding the underlying dynamics of the system from past states to predict future states accurately. Formally, we attempt to build a model that is able to use information about the particle trajectory till t=10 $\pm$ 1, to predict its future trajectory for another 4 $\pm$ 2 seconds.

3.2 Model formulation

For the task 3.2 we are presented with the trajectory of the particle $p1$, fired with an initial velocity $v1$ at time $t = 0$ which moves in the plane for $t = 10 \pm 1$ seconds. The simulations of this particle are sampled with a step size of $\Delta t = 0.1$ resulting into a sequence of length 100 ± 1. Our aim is to capture the trajectory of this particle from $t = 0$ to $t = 10$ and then predict its trajectory for another $t = 4 \pm 2$ seconds and ending the simulation at $t = 14 \pm 3$ seconds. As discussed previously, the position of the particle at time t depends on the position of the particle at time $t - 1, t - 2$.. and so on. Hence, there is significant correlation between the current position and the previous positions of the particle $p1$. The positions at different time-steps $t \pm \Delta t$ are sequential and depend directly on the historical information regarding the initial positions and velocity with which the particle was fired. If the position of the particle $p1$ at $t = 0$ changes in every simulation, then it is very likely that the particle follows a different trajectory in each and every simulation. Apparently, it is exactly this system dynamic that we need to capture, learn and the predict the trajectory of the particle $p1$ for the next 4 ± 2 seconds. In contrast to the many-to-one sequence mapping problem

formulated for task 3.1, where we were presented with the particle trajectory (of $p1$) and predicted the corresponding fixed charges $c2, c3, c4$ of the fixed particles $p2, p3, p4$, this a many-to-many sequence generation problem. To achieve the task of predicting the particle trajectory, we need to formulate a model that learns the distribution over the given sequence, captures the time dependencies and the correlations and generates an output for every time step of the continued time period ($t = 4 \pm 2$) resulting into a sequence of positions of particle $p1$ with length 40 ± 20. In order to achieve this task, we model our model architecture using two Recurrent Neural Networks (RNN's) acting as an encoder and a decoder. The intuition behind the architecture is that the encoder learns the distribution over the input sequence and encodes it into a higher d dimensional hidden vector. This d dimensional hidden vector is then fed as an input along with the last two co-ordinates (of the input sequence) to the decoder. The decoder then predicts the position at every time step of the target sequence using this d dimensional hidden vector and the co-ordinates from the last time step t_τ of the input sequence. We design 2 encoder decoder architectures to predict the future sequence of the particle $p1$ and the evaluate their relative performances. The model's formulated are as follows:-

1. ConvGRUEncoderDecoder
2. LSTMEncoderDecoder
3. Baseline Linear Predictor

3.2.1 ConvGRUEncoderDecoder

The intuition behind the model is same as that of the *ConvGRURegressor* described in section 3.1.2. The $1D$ convolution layer truncates the given input sequence using cross-correlations (equation 10) and then feeds it to a Gated Recurrent Unit (GRU) cell as described previously in section 3.1.2. In the encoder model, the dimension of the hidden vector h_t is $[N \times 512]$, where N is the batch_size and 512 is the dimension H of the hidden vector. We set the number of input channels C_{in} to 128 and the number of output channels C_{out} to $C_{\text{in}} * 2$. The kernel size k is set to 4 and the stride chosen for convolution is also set to 4. After performing convolution over the input sequence, we get a vector of dimension $[N \times C_{\text{out}} \times 13]$. The convolution truncates the input sequence of length 220 (flattened array of x and y co-ordinates) to a sequence of length 13 with 256 convolution filters.The GRU cell of the encoder takes in the convoluted input sequence and then produces a hidden vector of dimension $N \times 512$ at every time step of the input sequence. The output of the encoder is the hidden vector generated at time $t = \tau$, where τ is the last time step of the input sequence. As mentioned above, the dimension of the hidden vector h_τ is also $[N \times 512]$ (equation 7). The decoder unit is also a Recurrent Neural Network (RNN) that uses a Gated Recurrent Unit (GRU) cell but there is a structural difference in the architecture of the decoder and the encoder. This architectural difference is important as it directly affects the loss computation and the way we perform back-propagation in the network. The decoder takes in the hidden vector $h_{(t,t=\tau)}$ computed by the encoder at time $t = \tau$ which is the last time-step of the input sequence. Along with $h_{(t,t=\tau)}$, the decoder also takes the last positional co-ordinate of particle $p1$ i.e $x_{(t=\tau)}$. The decoder then produces an output at every time step after $t = \tau$ till $t = 14$. The output $\hat{y}_t$ at time t of the decoder is computed using the hidden vector h_t ($N \times 512$ dim). The decoder computes this hidden vector h_t at time t by taking the hidden vector of the previous time-step (h_{t-1}) and $x_{(t=\tau)}$ as an input (equation 7). To calculate the output $\hat{y}_t$ at time-step t, the decoder passes the hidden vector h_t through a linear layer that outputs a $N \times 512$ dim vector. This $[N \times 512]$ dimensional vector is then passed through a ReLU activation function which captures the non-linearity over the input space. The output of the ReLU activation is also a $N \times 512$ dim vector which is then passed through a final linear output layer that maps the hidden embedding to a $N \times 2$ dimensional vector. Hence, at every time step (after $t = \tau$), the model predicts a set of positional-co-ordinates, predicting the trajectory of the particle for time-steps $t = 4 \pm 2$. The output layer at every time-step is defined as follows:

$$f_{(1,t)} = h_t A^T_{(1,t)} + b_{(1,t)} \tag{14}$$

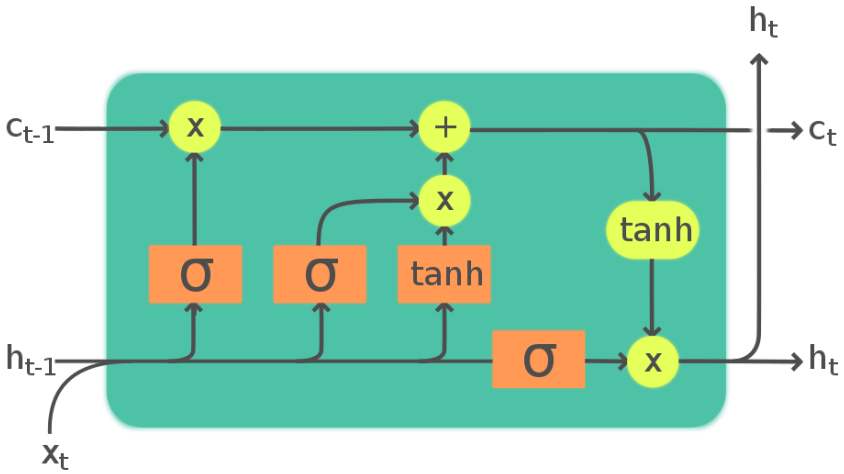


Figure 9: LSTM Cell Architecture
source

$$f_{(2,t)} = \max(0, f_{(1,t)}) \tag{15}$$

$$\hat{y}_t = f_{(2,t)} A^T_{(2,t)} + b_{(2,t)} \tag{16}$$

where $A^T_{(2,t)}$ is a parameter matrix that has the dimension $[512 \times 2]$ or in general $[H \times O]$ where H is the dimension of the hidden vector and O is the output dimension.

3.2.2 LSTMEncoderDecoder

We model two Recurrent Neural Networks (RNN's) as an encoder and a decoder where the output of one is fed as an input to the other. Both, the encoder and the decoder consist of a gated cell mechanism called the Long-Short-Term-Memory (LSTM) cell. If the input sequence is of length L_1 over some time period T, then the unrolled encoder architecture can be visualized as a sequential stack of L_1 cell units over a horizontal axis, where every cell takes the hidden vector from the previous cell state and the current input x_t (at time t) as an input. Similarly, the encoder can be seen as sequential stack (over a horizontal axis) of L_2 layers, where L_2 is the sequence of time steps for which we need to generate a sequence of the positional co-ordinates for particle $p1$. The difference between the encoder and the decoder architecture is that, the decoder produces an output at every time-step t over the sequence L_2. In theory, the gating mechanism makes the LSTM's robust over long sequences. To investigate this theoretical property of the LSTM's, we do not truncate the input sequences using a $1D$ convolution as done before with the *ConvGRUEncoderDecoder*. An LSTM cell makes use of an input gate and a forget gate to control the information flow within the cell. The input gate i_t and the forget gate i_t are described as follows :-

$$i_t = \sigma(h_{t-1} U_i + x_t W_i + b_i) \tag{17}$$

$$f_t = \sigma(h_{t-1} U_f + x_t W_f + b_f) \tag{18}$$

where, U_i, U_f, W_i, W_f are the shared parameters between different time steps that the model needs to learn and σ denotes the sigmoid activation function. Similar to the input and the forget gate, the LSTM's also have an output gate. Finally, the output of an LSTM cell are two vectors, the cell state c_{ct} and the hidden state h_t. They are described as follows:-

$$o_t = \sigma(h_{t-1} U_o + x_t W_o + b_o) \tag{19}$$

$$c_t = \tanh(x_t W_c + h_{t-1} U_c + b_c) \star i_t + f_t \star c_{t-1} \tag{20}$$

$$h_t = \tanh(W_h c_t) \star o_t \tag{21}$$

where $\star$ is the hadamard product and U_o, U_c, W_o, W_c, W_h are also the shared parameters between the timesteps that the model intends to learn during the training regime. The encoder takes the input sequence which is tensor of shape $[N \times 2]$ where N is the batch size and 2 indicates the corresponding x and y co-ordinates of the particle $p1$ at time t. The forget gate acts as an information barrier that removes the irrelevant information from the previous hidden state h_{t-1} and updates the previous cell state c_{t-1} given the current input x_t. The forget gate uses a sigmoid (σ) function which blocks all the input values that map to 0. This helps in handling the vanishing gradient while performing back-propagation through the network. Similar to the forget gate, the input gate tends to remove or stop the values that map to 0 and are convolved with $\tanh(x_t W_c + h_{t-1} U_c + b_c)$ and is then added to the cell state after convoluting it with the output of the forget gate. This defines the current information or the cell state vector which is then used to compute the current hidden state h_t of the LSTM cell (equation 21). Thus, we compute the hidden vector as stated above for every time step t where $t \in [0, 10]$. Similar to the *ConvGRUEncoderDecoder* architecture stated above, we pass the hidden vector $h_{(\tau)}$ where τ is the last time step of the input sequence L_1, along with positional co-ordinates x_τ as an input to the decoder. h_τ has dimensions $[N \times 512]$. The working of the decoder is similar to that of the decoder stated for *ConvGRUEncoderDecoder*, where we compute the output at every time step for the extended sequence over $t = 4 \pm 2$. The output at each time step is computed using equations 14, 15, 16. The dimensions of the vector remain the same as well.

3.2.3 Baseline Linear Predictor

Further we create a baseline linear predictor which predicts the positions 2 time steps in the future. It is a simple linear baseline model which extrapolates 2 time steps in the future and predicts the current position as:

$$x_i^t = x_i^t + (x_i^t - x_i^{t-\Delta t}) \tag{22}$$

3.3 Implementation and training

3.3.1 Loss function

To train model for this task, we once again use the MSE loss as defined in section **??**. We choose the MSE loss for this task as for each step of the simulation (past the initial 100 steps), the model predicts the position of the different particle. As we are predicting positions, which are continuous values, the MSE loss is suitable as it would measure the squared distance between the predicted and actual position for each time step. Formally, the MSE loss of a single step would be $(x_n - y_n)^2$, where $(x_n - y_n)$ is the euclidean distance between the predicted position (x_n) and the actual position (y_n). As we are predicting for multiple steps, we once again calculate the final loss using equation 11, but with n equal to the number of steps in the future we are predicting.

3.3.2 Optimization

The optimzer used for task 3.2 is the SGD with nestorov momentum. The approach is the same as the one described in section 2.3.2.

3.3.3 Weight Initialization

In addition to the different kinds of layers/cells in task 3.1, we make use of an additional type of layer when developing models for task 3.2, namely the LSTM cell. The weights of the LSTM cell are initialized from a uniform distribution $U(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{hidden\_size}$

[Paszke et al., 2019]. Similar to the GRU cell, *hidden_size* is the number of features in the hidden state [Paszke et al., 2019].

3.3.4 Learning Schedule

With the loss function, optimizer and the weight initialization defined above we now describe the learning schedule of both the encoder-decoder models formulated for task 3.2. As stated above, we have a sequence prediction problem at hand and we build two autoregressive model architectures using the Gated Recurrent Unit Cell and Long-short-term memory cell architectures. The training data consists of 150 simulations of the trajectory of particle $p1$ from $t = 0$ to $t = 10$ with $\Delta t = 0.1$, sampled from the 800 simulations that we had for task 3.1. Apart from this, the training data also consists additional positional co-ordinates as continued simulation trajectory for another $t = 4 \pm 2$ seconds after $t = 10$ seconds for each of the 150 simulations in the training set. Similar to task 3.1, we face the issue of varying simulation lengths L of the particle trajectory not only for the initial 100 ± 10 simulations but also for the next 40 ± 20 continued simulation sequence. Doing a zero padding on the differing sequence length as compared too the maximum sequence length in the training sense would create problems while training the model. In contrast to the situation in task 3.1, we feed the output of the encoder i.e the hidden vector h_τ at $t = \tau$, where τ is the last time stamp in the input sequence, to the decoder along with the last positional co-ordinates x_τ ($\in \mathbb{R}^2$. If the last positonal co-ordinate x_τ becomes 0 due to the zero-padding, we are essentially feeding a sequence of $\tilde{L}$ 0's to the decoder where $\tilde{L}$ is the continued simulation length. This will surely aggravate the vanishing gradient problem, making the model sparse and failing to capture the distribution over the input sequence. Hence, instead of zero padding the sequence, we copy the last positional co-ordinates of the input sequence length to fill the difference between the maximum sequence length in the training data. The intuition behind doing this is to force the model to capture the stationary nature of the particle after a certain time $t^{'}$ where $t^{'} < 10$. Once the varying sequence length's are handled, we sample the batches from the training set in batches of size of 10. In theory, the hidden state from the encoder and x_τ are passed to the decoder which predicts the output at every time step over the sequence $\tilde{L}$. The loss at every time step is calculated by computing the mean squared error between the predicted and the actual target position. A total loss over all the time steps in sequence $\tilde{L}$ is then backpropagated through the network and the optimizer makes the relevant gradient updates based on the loss input to minimze the loss.

$$Loss = \sum_{t} Loss_t \tag{23}$$

While implementing the learning schedule, the loss computation varies a bit. For every batch k ($k \in N$) in the training set, we get the decoder output at every time step $t^{'}$ as the forecasted position and we stack it in a tensor list. We do this over the entire forecasted sequence $\tilde{L}$ and get a output tensor of size $N \times L$. We pre-process our targets in the training set such that they have a dimension $150 \times \tilde{L}$, where 150 is the number of training simulations. The dataloader loads the targets for the mini-batch k in a manner such that the tensor size is $N \times \tilde{L}$. We then pass the predicted output tensor ($N \times \tilde{L}$) for a mini-batch k and the actual target tensor ($N \times \tilde{L}$) to the loss criterion and compute element-wise mean squared error. This element-wise mean squared error for the batch k is then fed to the optimizer which updates the weight matrix W with the objective of minimizing the mean squared error loss. We repeat this process for all the k mini-batches in our training set over E epochs and train the encoder-decoder models.

3.4 Experiments and results

The setup of our experiment consists of a training set that has 150 simulations of the trajectory followed by particle $p1$ from $t = 0$ to $t = 10$ with a step size of $\Delta t = 0.01$. These 150 simulations have been sampled randomly from the set of 800 simulations that was used to achieve the goals formulated in task 3.1. Our validation set consists of 100 simulations of

the moving particle $p1$ which was fired with an initial velocity $v1$ and the test set consists of 100 distinct (from the validation and the training set) simulations of the particle $p1$. With the aim of predicting the trajectory of the particle $p1$ for the next 4 ± 2 seconds, we set up an experiment where we train the *ConvGRUEncoderDecoder* model over 400 epochs and use a standard mini-batch stochastic gradient descent algorithm to optimize our mean squared error loss criterion by finding the global minimum and making the relevant weight updates. For our experiment, we set the learning rate l of the optimizer to 0.01 and the nesterov momentum to 0.9.

$$\text{validation loss}_E = \frac{\sum_k^N \text{validation loss}_k}{N} \tag{24}$$

Table 4: Average Validation Loss and Test Loss

Model	**ConvGruEncDecod**	**LSTMEncDecod**	**Linear Model**
Best Validation Loss	1.41696	3.27537	8.47635
Average Test Loss	1.382536	2.833698	8.26148

The evaluation metric used to monitor the performance of the model is the mean squared error loss over the predicted sequence $\hat{\tilde{L}}$ and the actual target sequence $\tilde{L}$ of the particle $p1$. Throughout our training regime we monitor the overall training and validation loss for every epoch and ideally stop training when the validation loss stops improving. The overall average training loss for an epoch is computed by taking the average over the training loss for every mini-batch $k \in N$ for an epoch. It can be given as follows:

$$\text{training loss}_E = \frac{\sum_k^N \text{training loss}_k}{N} \tag{25}$$

where training_loss_k is computed using equation 23. Similarly, we compute the validation loss for an epoch as :-

The overall training and validation loss of the model can be computed by averaging the respective loss for each epoch over the total number of epoch E over which the model was trained. While training the models we encounter overfitting to a significant degree and we reduce it by adding drop-out's (0.5) into the final output layer of each time step of our encoder-decoder architectures. Additionally, we also make use of batch normalization in the convolution layers of the *ConvGRUEncoderDecoder* architecture. We report the results obtained after training both the models for 400 epochs with a learning rate and nesterov momentum specified above. Furthermore, we also compare our results from both encoder-decoder architectures with a baseline linear predictor.

Furthermore, we also report the training and validation loss for the models.

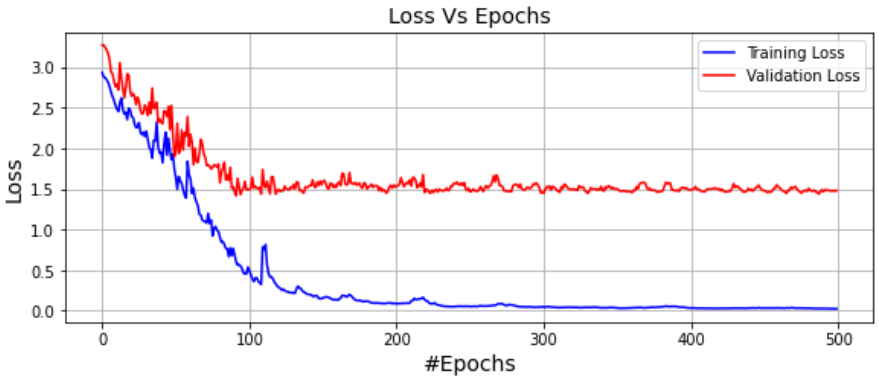


Figure 10: Training and Validation Loss ConvGRUEncoderDecoder

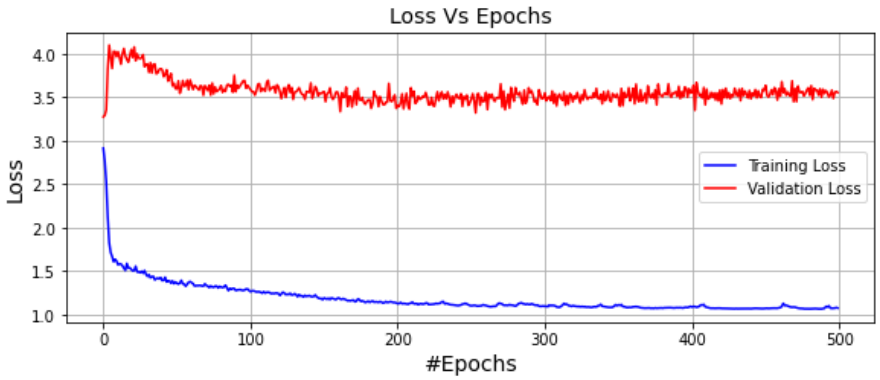


Figure 11: Training and Validation Loss LSTMEncoderDecoder

From figure 10 and 11, we can clearly see the evident overfitting of the models even after applyig regularization to some extent. Although the ConvGRUEncoderDecoder model outperforms the LSTMEncoderDecoder model by quite some extent. Another aim of the experimentaiton was to measure the extent of robustness of the LSTM gated architectures in capturing long sequences as they are theorized to do so. As evident from the results, summarizing the sequences using a $1D$ convolution layer outperforms the vanilla LSTMEncoderDecoder architecture by far and in-scope of this experimentation, problem formulation and dataset, LSTM's fail to robutly capture the ditributuion of the input sequences and forecast the next time-series or a sequence of positional co-ordinates at different time steps. As GRU's are robust in capturing short term sequences robustly, combining them with a $1D$ convolutional layer yields better results as compared to the two other model's formulated in this experimentation. We also analyze the performance of the model over the k mini-batches in our test dataset. We obtain the best overall test loss for the batch 7 and the worst over test loss over batch 2 in our test dataset (Table 5). After analyzing the individual test losses

Table 5: Batches with The Best and the Worst Test Loss

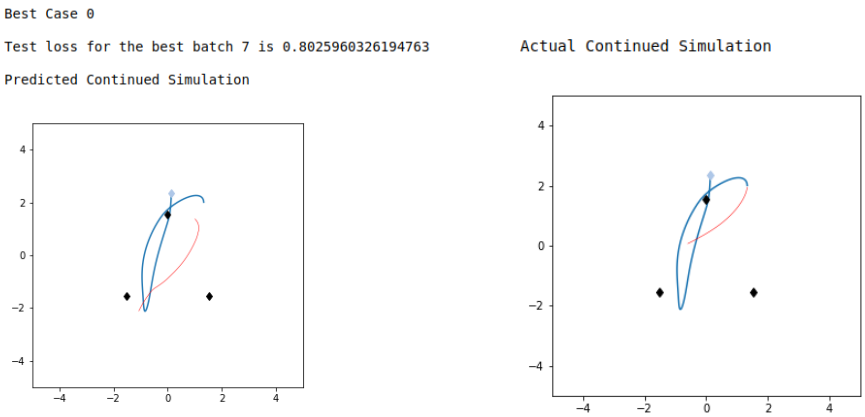
Batches	**Best Batch : Batch No 7**	**Worst Batch : Batch No 2**
Batch Test Loss	0.80259	2.0330

for every batch in the test set, it becomes interesting to have analyze the predictions and understand how well the model is forecasting the future positions of particle $p1$ for the next 4 ± 2 seconds. So, we perform individual analyses over the results predicted by the model for our best and worst test batch.

Figure 12 and 13 describe the best predicted trajectories of the particle $p1$. Although not entirely perfect, the ConvGRUEncoderDecoder model presents a fair prediction of the particle trajectory. The direction and the smooth curvatures in the trajectory seem to have captured to a fair extent by the model. Although the worst case performance of the model seems quite off. The difference between the loss for the best and the worst batch is relatively high. The reason behind the predictions of the worst cases being off seems to be from the over-fitting nature of the model as it is clearly struggling to generalize over the test data. The high error variance in the estimations is quite evident both from figure 10 and table 5. Adding more regularization to the model and limiting the overfitting over the training data should improve the performance of the model.

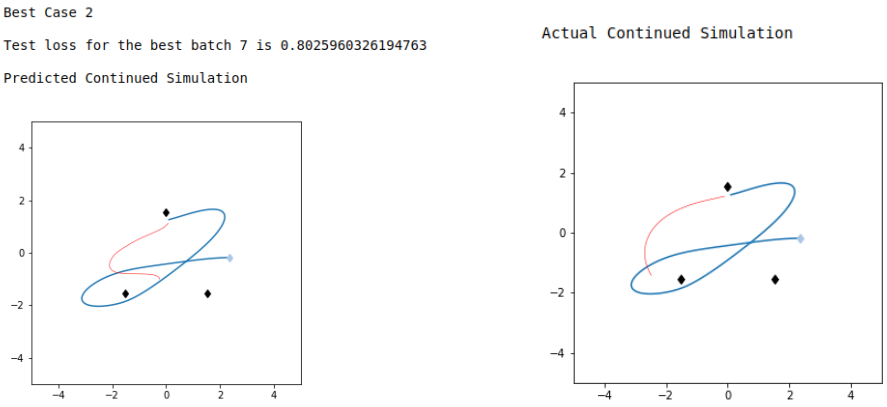
3.5 Conclusion

In this section we formulate the problem of sequence forecasting, where we predict the future positions of particle $p1$ for the next 4 ± 2 seconds given the input sequence for the initial 10 seconds. We design two autoregressive Recurrent Neural Network based architectures to tackle this many-to-many sequence mapping problem. The aim of the experimentation



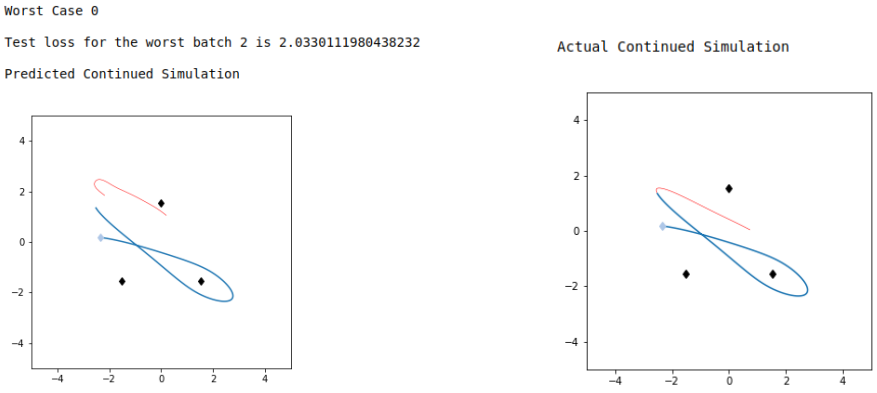
(a) Predicted Trajectory Case 0 Batch 7 (b) Actual Trajectory Case 0 Batch 7

Figure 12: Simulations from Best Batch 7



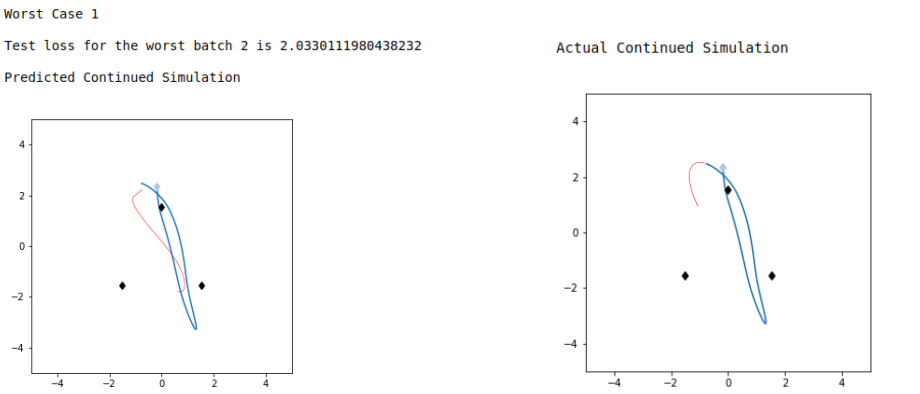
(a) Predicted Trajectory Case 2 Batch 7 (b) Actual Trajectory Case 2 Batch 7

Figure 13: Simulations from Best Batch 7



(a) Predicted Trajectory Case 1 Batch 2 (b) Actual Trajectory Case 1 Batch 2

Figure 14: Simulations from Worst Batch 2



(a) Predicted Trajectory Case 1 Batch 2

(b) Actual Trajectory Case 1 Batch 2

Figure 15: Simulations from Worst Batch 2

was to train the models over the input data and analyze their robustness in predicting the future positions of a moving particle $p1$ with an initial velocity $v1$, in a plane of fixed particles $p2, p3, p4$ with fixed charges $c2, c3, c4$. Our experimentation's show that a $1D$ convolution operation is robust in summarizing a given input sequence and bolsters the ability of a Gated Recurrent Unit cell in capturing the time-dependencies and correlations over a long input sequence. Furthermore, we also study the results obtained by modelling a LSTMEncoderDecoder architecture and find out that the LSTM architecture is not as robust as per the theoretical expectations in modelling long time-sequences in context of the problem and experiment formulated in this section. Finally, we also find empirical evidence regarding the performance of a baseline linear predictor and its struggles in capturing the complex highly structured time-dependent nature of the given trajectory of the moving particle $p1$.

References

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2017. URL `https://arxiv.org/abs/1706.02216`.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page III–1139–III–1147. JMLR.org, 2013.

4 Appendix

4.1 Model Architectures for Assignment 2

```
he model architecture:

PhysicsSAGE(
 (sage): SAGEConv(6, 192)
 (drop): Dropout(p=0.3, inplace=False)
 (fc1): Linear(in_features=192, out_features=96, bias=True)
 (fc2): Linear(in_features=96, out_features=2, bias=True)


No of trainable parameters 21,218 in the model
```

Figure 16: PhysicsSAGE Architecture

4.2 Model Architectures for Assignment 3 Task 3.1

```
The model architecture:

 GRURegressor(
  (rnn): GRUCell(2, 256)
  (fc1): Linear(in_features=256, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=3, bias=True)
)

 No of trainable parameters 266,243 in the model
```

Figure 17: GRURegressor Architecture
Time taken to train the GRURegressor model : 843.459 seconds ≈ 14 minutes over 400 epochs

```
The model architecture:

ConvGRURegressor(
 (conv1d_1): Conv1d(1, 64, kernel_size=(4,), stride=(4,), padding=valid)
 (conv1d_2): Conv1d(64, 128, kernel_size=(4,), stride=(4,), padding=valid
 (rnn): GRUCell(128, 256)
 (drop): Dropout(p=0.5, inplace=False)
 (fc1): Linear(in_features=256, out_features=256, bias=True)
 (fc2): Linear(in_features=256, out_features=3, bias=True)
)

No of trainable parameters 396,227 in the model
```

Figure 18: ConvGRURegressor Architecture
Time taken to train the ConvGRURegressor model : 723.121 ≈ 12 minutes over 400 epochs

```
The model architecture:

 BenchmarkFNN(
  (drop): Dropout(p=0.5, inplace=False)
  (fc1): Linear(in_features=220, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=3, bias=True)
 )
```

Figure 19: Benchmark MLP Architecture
Time taken to train the Benchmark MLP model :84.752 ≈ 1.4 minutes over 400 epochs

4.3 Model Architectures for Assignment 3 Task 3.2

```
The model architecture:

 ConvGRUForecaster(
  (dropout): Dropout(p=0.2, inplace=False)
  (batchnorm): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv1d_1): Conv1d(1, 128, kernel_size=(4,), stride=(4,), padding=valid)
  (conv1d_2): Conv1d(128, 256, kernel_size=(4,), stride=(4,), padding=valid)
  (encoder): GRUCell(256, 512)
  (decoder): GRUCell(2, 512)
  (drop): Dropout(p=0.5, inplace=False)
  (fc1): Linear(in_features=512, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=2, bias=True)
)

 No of trainable parameters 2,371,458 in the model
```

Figure 20: ConvGRUEncoderDecoder Architecture
Time taken to train the ConvGRUEncoderDecoder model : 437.752 seconds ≈ 7.3 minutes over 400 epochs

```
 The model architecture:

  seq2seq(
   (encoder): LSTMEncoder(
     (rnn): LSTMCell(128, 256)
     (drop): Dropout(p=0.5, inplace=False)
     (fc1): Linear(in_features=256, out_features=256, bias=True)
     (fc2): Linear(in_features=256, out_features=256, bias=True)
     (conv1d_1): Conv1d(1, 64, kernel_size=(4,), stride=(4,), padding=valid)
     (conv1d_2): Conv1d(64, 128, kernel_size=(4,), stride=(4,), padding=valid)
   )
   (decoder): LSTMDecoder(
     (rnn): LSTMCell(2, 256)
     (drop): Dropout(p=0.5, inplace=False)
     (fc1): Linear(in_features=256, out_features=256, bias=True)
     (fc2): Linear(in_features=256, out_features=2, bias=True)
   )
  )

   No of trainable parameters 892,610 in the model
```

Figure 21: LSTMEncoderDecoder Architecture
Time taken to train the LSTMEncoderDecoder model : 440.986 seconds ≈ 7.3 minutes over 400 epochs