Simulating particle trajectory

Assignment 2 & 3 - Deep Learning (2AMM10)
Eindhoven University of Technology

Group-7

Ambarish Moharil (1704818)
Kunal Geed (1736051)
Mert Lostar (1668846)

1 Problem Formulation

Understanding the relationship between different objects and how they interact with each other is an extremely complicated task, especially in systems that involve multiple objects. This understanding plays an especially important role in the task of simulation, as you are supposed to understand and model the continuous interaction between different objects. Furthermore, these simulations play an important role in science and engineering as they are necessary to conduct experiments and gain insight about the outcomes of those experiments. However, these simulations can be extremely expensive to compute as they are generated sequentially in very small steps to ensure accuracy. To address these issues, we make use of deep learning. In this report, we focus on simulating the dynamics of multiple charged particles in a plane over time. Simulating the dynamics of this system can be complicated due to each particle being attracted or repulsed by other particles in the system based on its charge and distance to other particles.

First 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 depend on their charges and locations. For this task we aim to investigate if we are able to predict the future state of dynamic system containing n objects that interact with each other. 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 actual and predicted position of a particle. We would aim to minimize this distance for each particle in the system. Formally, we consider the following task.

1. Sequence prediction (Time-series forecasting): We aim to build a model that is able to capture the underlying dynamics of a system over time and use that information to accurately predict how this system evolves over time. 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 $t = 0$. Furthermore, as we are understanding the underlying dynamics of the system, we are not required to sequentially generate all the steps in the simulation before t and can directly predict the position of the particles which is less computational expensive.

In task 3, we consider a special case of task 2, where we consider a single particle that is moving and three other particles that are *fixed*. All other particles, c_1, c_2, and c_3, have a charge sampled uniformly from the interval $[-1, 1]$, as opposed to -1,1 in task 2.

As we focus on predicting the dynamics of a system with only a single particle, we divide task 3 into two further subtasks. In task 3.1, we aim to identify the charges c_2, c_3, c_4 that lead to the trajectory of particle p_1. In task 3.2, we aim to predict the continuation of the particle's trajectory based on its initial trajectory. This is opposed to task 2, where we only make use of the information at $t = 0$ to predict the position of the particles. Here we predict the trajectory of a single particle based on more than just the trajectory at $t = 0$. In task 3.1, we would measure the performance of our performance by considering the sum of the squared difference between the predicted and actual charges of the fixed particle. For task 3.2, we would measure the success of the model based on how close the predicted trajectory is to the actual trajectory.

1. Predicting Parameters: We aim to build a model that is able to predict the parameters that can best explain 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. Forecasting: We aim to predict the build a model capable of forecasting the trajectory of a particle in a dynamic system. 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.

2 Data

For task 2, the data is generated by randomly initializing the preliminary positions, velocity, and charges of five particles. The system is simulated for 1.5 seconds following a 3 second warm up. The simulation consists of discrete time (of 0.001 seconds) steps. We are then given a training set consisting of 10,000 simulations, validation and test set of 2,000 simulations. For task 3, the dataset are generated by shooting a particle in a plane with three fixed particles (in a triangle formation). Simulations of varying lengths (9 to 11 seconds) are generated. For task 3.1, we are provided the charges of the fixed particle so that the model can learn to predict these. For task 3.2, the simulation is continued for additional 2 to 6 seconds. The additional simulations are given separately, as this is the trajectory that we aim to predict for a given particle and charges.

3 Model Formulation

3.1 Task 2

Already moved this to the other template

3.2 Task 3.1

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 which 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 the performance 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

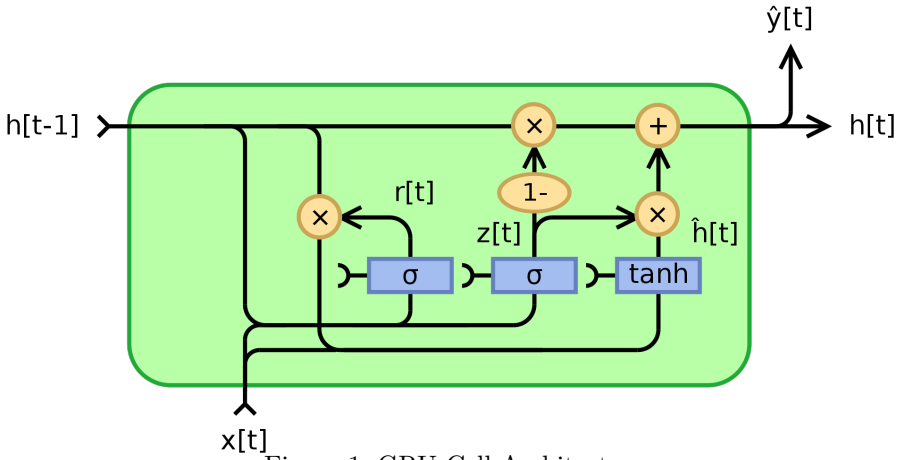

Figure 1: GRU Cell Architecture
[source]

3. Benchmark Multi-layer Perceptron (MLP)
4. Baseline Linear Predictor

3.2.1 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 $x =< x_1, x_2, x_3.....x_t >$.

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

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:

$$r_t = \sigma(W_r x_t + U_r h_{t-1}) \tag{2}$$

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:

$$z_t = \sigma(W_z x_t + U_z h_{t-1}) \tag{3}$$

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:

$$h_t = (1 - z_t)h_{t-1} + z_t \tilde{h_t} \tag{4}$$

where

$$\tilde{h_t} = \tanh(Wx_t + U[r * h_{t-1}]) \tag{5}$$

$*$ 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 2 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 3 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 $x =< x_1, x_2, ...x_t >$, we pass it through a linear layer in order to map it on the target domain. The linear layer is as follows:

$$y = xA^T + b \tag{6}$$

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 $batch\_size \times 3$.

3.2.2 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:

$$\text{out}(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{\text{in}-1}} \text{weight}(C_{out_j}, k) \star \text{input}(N_i, k) \tag{7}$$

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.

3.2.3 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 6). 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 6) 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*.

3.2.4 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:

$$x_i^t = x_i^t + (x_i^t - x_i^{t-\Delta t}) \tag{8}$$

We again hypothesize that

H_{0_2}:- The baseline linear predictor fails to capture the time dependencies and performs worse than that of the *GRURegressor* and *ConvGRURegressor*.

3.3 Task 3.2

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

3.3.1 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 7) 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 4). 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 4). 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:

$$f_{(1,t)} = h_t A^T_{(1,t)} + b_{(1,t)} \tag{9}$$

$$f_{(2,t)} = \max(0, f_{(1,t)}) \tag{10}$$

$$\hat{y}_t = f_{(2,t)} A^T_{(2,t)} + b_{(2,t)} \tag{11}$$

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.

3.3.2 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 :-

$$i_t = \sigma(h_{t-1} U_i + x_t W_i + b_i) \tag{12}$$

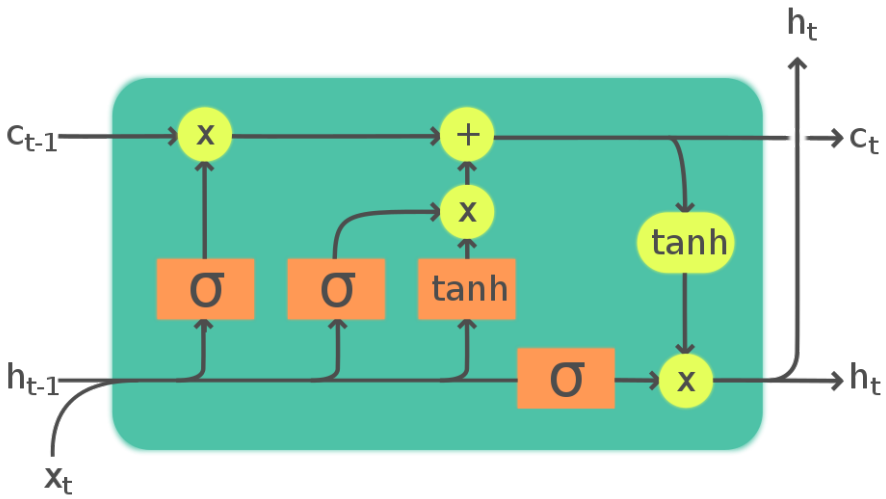

Figure 2: LSTM Cell Architecture
[source]

$$f_t = \sigma(h_{t-1}U_f + x_tW_f + b_f) \tag{13}$$

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:-

$$o_t = \sigma(h_{t-1}U_o + x_tW_o + b_o) \tag{14}$$

$$c_t = \tanh(x_tW_c + h_{t-1}U_c + b_c) \star i_t + f_t \star c_{t-1} \tag{15}$$

$$h_t = \tanh(W_h c_t) \star o_t \tag{16}$$

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 $\tanh(x_tW_c + h_{t-1}U_c + b_c)$ 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 16). 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 9, 10, 11. The dimensions of the vector remain the same as well.

4 Experiment Design & Model Implementation

4.0.1 Weight Initialization

4.0.2 Learning Schedule

Given the weight initialization and the optimizer above, we now describe the learning schedule of our models formulated for task 3.1 and task 3.2

5 Evaluation & Results

6 Conclusion

References

7 Appendix