Title: Predicting trajectories in simulated classical mechanics environment using recurrent neural networks.

Author: Nikita Koloskov (Minerva University, Class of 2023).

Problem & Background:
I am exploring the use of recurrent neural networks (RNNs) to learn underlying physics in classical mechanics scenarios, specifically predicting the trajectories of single ball objects in 2D physics simulations. I am using Phyre simulator to generate data and isolate the intuitive physics problem from contextual inference. My researc his motivated by potential uses in various fields, such as robotics, military, neuroscience, and general AI. My capstone discusses how RNNs can be leveraged to understand the underlying physics of classical mechanics scenarios by attempting to learn Newton's Laws of Motion and the Law of Conservation of Momentum. I am using the analysis of how well RNNs can predict trajectories as a proxy for speculating how accurately the models learn these laws of Physics.

Approaches:
I am using a bottom-up research approach. I started with using simple Feedforward Neural Networks to predict one time step at a time in a free-fall scenarios, using input that allows to solve the same problem analytically. I am then transitioning to predicting the whole free-fall and bounce in a single dimension scenarios from using only initial state as an input. Step by step I am increasing complexity of the problem: by including the second coordinate, then including two other balls to the input, and finally transitioning to scenarios with fall and collisions. I am comparing the performance of Recurrent Neural Networks models (among which are Vanilla RNN, GRU, and LSTM) to that of Reservoir Computing (RC) models (among which are Echo State Networks and its deep variations) on each of the tasks mentioned above.

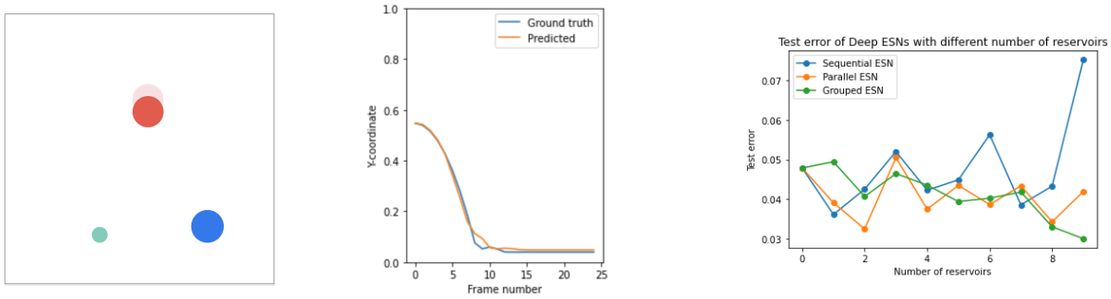


Figure 1. (left) Visual output from Phyre physics simulator; (middle) Predicted and ground truth red ball's Y-coordinate time-series in free-fall; (right) Test set performance of Deep ESN models as a function of number of reservoirs.

Outcomes:
Traditional RNN architectures demonstrated outperformed RC models on each of the explored tasks. All models demonstrated a solid ability to predict the trajectory of the ball in the scenarios where it free-falls and bounces from the ground, but struggled to provide reasonable predictions in scenarios where it collided with other balls. Arguably, RC models demonstrate a better grasp of the underlying physics while struggling with the precision of predicted values. It especially manifests in the shape of predicted trajectories for the free-fall scenarios: RC models correctly predicted the fall strictly along a straigt line, as well as the bounce once the ball hits the ground. In contrast, traditional RNNs do not grasp such aspects, but the free-fall trajectories are predicted with a much smaller error.

Table of contents

Section 1. Introduction

In this project, I explore the capabilities of recurrent neural networks to learn underlying physics in classical mechanics scenarios. I focus my research on the problem of predicting the trajectories of single ball objects in 2D physics simulations with no more than 3 ball objects on the scene, in order to more deeply explore this topic. To generate our data, I use the Phyre simulator, which provides an unlimited source of data and allows me to isolate the intuitive physics problem from contextual inference.

My research is motivated by a wide range of potential use cases in various disciplines, ranging from robotics and military to neuroscience and general AI. In robotics, for example, understanding the physical dynamics of objects and predicting their trajectories could enable us to analyze the impact of physical interventions. Similarly, in military applications, the ability to predict the trajectories of artillery/missile parts after impact could be a powerful tool for defense tactics. Moreover, modeling how our brain perceives object dynamics could be useful in neuroscience research, as well as in general AI tasks, such as modeling the perception of the surrounding.

In this paper, I will discuss the implications of my research and how neural networks could be leveraged to understand the underlying physics of classical mechanics scenarios. To predict trajectories correctly, one should correctly use the Newton's Laws of Motion, as well as the Law of Conservation of Momentum. Analysis of how well the recurrent neural networks can predict the trajectories of the objects can serve as a proxy for speculating how correctly the models learn the mentioned laws of Physics.

Section 2. Previous Work <THIS SECTION IS NEW>

In this research paper, I investigate the use of Recurrent Neural Network (RNN) and Echo State Network (ESN) architectures for predicting the trajectory of an object in a simulated classical mechanics scenario. I build on previous works that have explored the use of deep learning techniques in image understanding, trajectory, and destination prediction.

In Mottaghi et al.'s (2015) work on Newtonian image understanding, the authors proposed a novel framework for analyzing static images by modeling the dynamics of objects within them. They utilized a CNN to learn object features and then used a physics-based model to predict the future state of each object. The model was trained on a large dataset of labeled images, resulting in a powerful tool for predicting the trajectories of objects in images. The authors demonstrated the effectiveness of their approach by evaluating it on several datasets and showing that it outperformed previous state-of-the-art methods. The problem of their approach, however, is that it relies on only 12 modeled trajectories, and the proposed model is only able to classify and predict the trajectories according to this set. These trajectories also don’t consider interacting with the objects on the scene other than ground.

Expanding on this idea, Mottaghi et al. (2016) investigated the prediction of the effect of external forces on objects in images. They introduced a framework that combines deep learning with physics-based models to predict the effect of a force applied to an object in an image. The model first learns a representation of the object and the force, and then predicts the future state of the object after the force is applied. The authors showed that their approach achieved state-of-the-art results on several datasets, including real-world images. The suggested model solves a contextual information inference problem; the trajectory prediction, however, narrows down to a binary predicting if the object can move along a specified direction or no, and if yes, then how much.

In Song et al.'s (2020) work on destination prediction, the authors proposed using a Deep Echo State Network (DeepESN) to predict the destination of an object based on its trajectory. DeepESN is a variant of the popular Echo State Network (ESN) that is capable of learning complex temporal dynamics. The authors showed that their model outperformed several baselines on a dataset of taxi trajectories, demonstrating its potential for applications such as

traffic prediction. The moving space of the considered problem, however, is a city rather than a classical mechanics environment. So instead of being a problem of intuitive physics, it is a problem of modeling human decisions.

Finally, Wu et al. (2017) proposed a recurrent neural network (RNN) based approach for modeling trajectories. Their method uses a variant of RNN called Long Short-Term Memory (LSTM) to capture long-term dependencies in the trajectory data. The authors evaluated their model on several datasets, including human motion tracking and vehicle navigation, and showed that it outperformed several previous methods. Similarly to the previous paper, however, this paper is also a problem of modeling human behavior rather than of intuitive physics, as it is focused on predicting the trajectories of moving around the city.

Overall, these previous works highlight the potential of deep learning techniques for a variety of tasks related to image understanding, trajectory, and destination prediction. These methods are not only highly accurate, but also have the potential to be applied in real-world scenarios, such as traffic prediction and human motion tracking. In my research, I will be exploring the potential of recurrent neural networks to predict trajectories in the scenarios regulated by deterministic laws of physics.

Section 3. Data Preparation

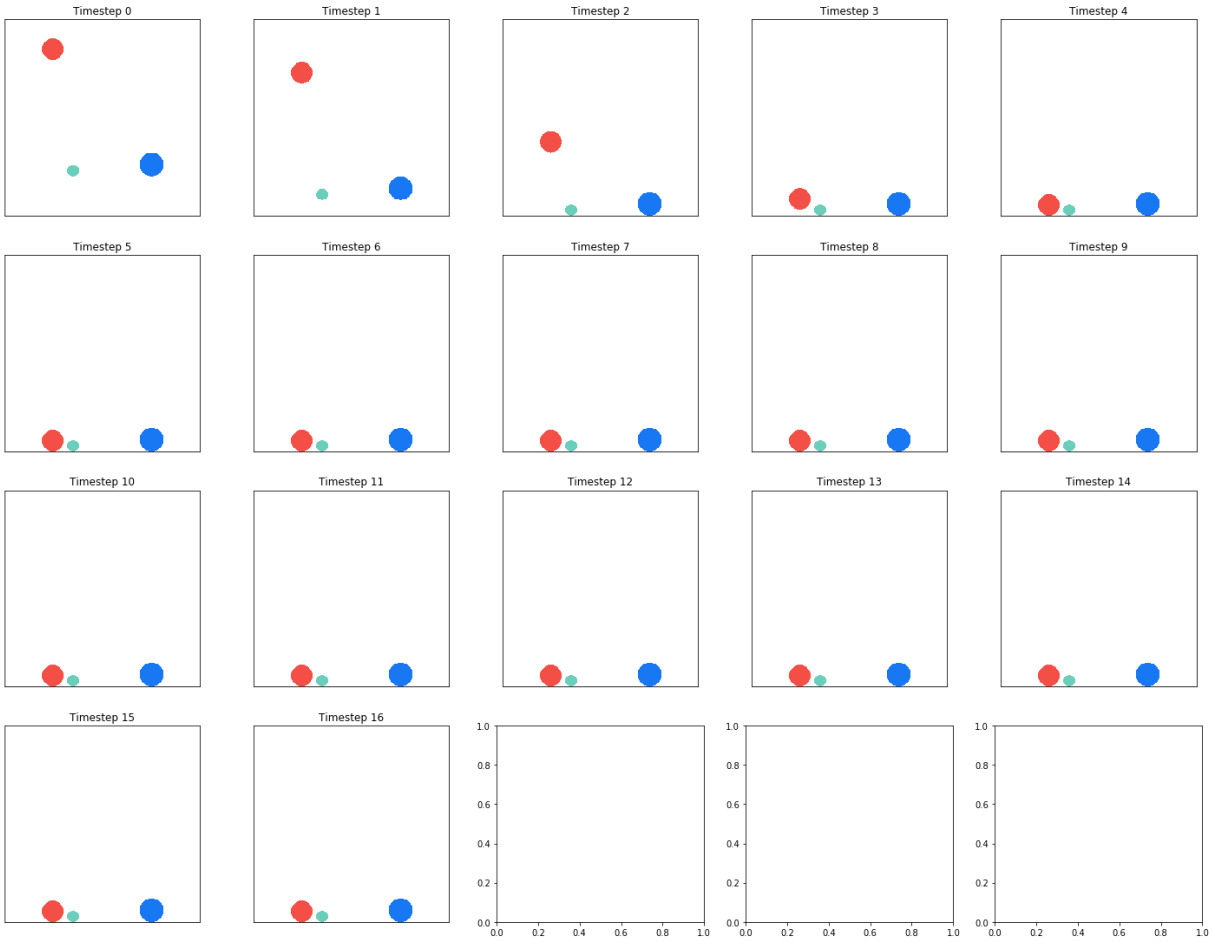
To generate data, I am using a simulator that comes with Phyre benchmark package (Bakhtin et. al., 2019). I am adapting the generated data to a format that is understood by PyTorch Deep Learning framework (Paszke et. al., 2019).

3.1. Overview of the simulator

Phyre benchmark (Bakhtin et. al., 2019) comes with a two-dimensional simulated environment with Newtonian physics and a set of tasks to evaluate Intuitive Physics algorithms. The initial state of each task contains some number of geometric bodies that can be differentiated by shape and color. A user can place one or two more objects on the scene (depending on the task tier), constrained to be the balls of the red color. The user can customize the location and the radius of the red ball that is placed on the scene. Once the user action is defined, Phyre can simulate the scenario according to the programmed physics. Once the simulation is complete, Phyre can provide two types of output - visual (a set of graphical data, each picture representing the state of the simulation at a particular step), and numerical (a tensor that contains data about each object on the scene at every simulation step). In numerical output, each object is characterized by 14 numbers, as described in the Jupyter Notebook supporting Bakhtin et. al. (2019) paper:

- x- and y-coordinates in pixels of the center of mass divided by scene width and height respectively;
- Orientation (angle) of the object in radians divided by 2π;
- Diameter in pixels of the object divided by scene width;
- One hot encoding of the object shape, in the order of “ball”, “bar”, “jar”, “standing sticks”;
- One hot encoding of the object color, in the order of “red”, “green”, “blue”, “purple”, “gray”, “black”.

Example of the visual output is given in Figure 1 (a) below. Example of the numerical output of the initial state of the scenario (i.e. containing just green and blue balls, without the red ball) is given in Figure 1 (b).



(a)

```
Initial featurized objects shape=(1, 2, 14) dtype=float32
[[[0.35  0.229 0.    0.059 1.    0.    0.    0.    1.    0.
   0.    0.    0.   ]
  [0.75  0.261 0.    0.121 1.    0.    0.    0.    0.    1.
   0.    0.    0.   ]]]
```

(b)

Figure 1. Graphical output of the sample simulation (a) and numerical output for the initial state (b) of the Phyre simulator. Retrieved from https://github.com/facebookresearch/phyre/blob/main/examples/01_phyre_intro.ipynb

Phyre provides a set of templates containing objects with different topologies. Each template contains a set of similar tasks with objects arranged differently in the initial state. While using the simulator, users have ability to choose a template, task, action (where to put the ball(s) and how big to make them), and the simulation stride (how big is the gap between two simulation steps).

3.2. Preparation of the datasets

For the purpose of my research project, I am only focusing on scenarios with three balls, where I am aiming to predict the trajectory of the red ball. Thus, the only information I will need to retrieve from the simulator is x- and y-coordinates of each ball and their diameters. While I will use visual output for the visualization purposes, my models only deal with numerical output, which allows me to separate the problem of learning Physics from the problem of inferring information from visual data.

To generate data that only contains three balls on each scene, I am constraining to the template with code name '00000', tasks from which can be seen in the demo playground that supports Bakhtin et. al. (2019) paper. For each task in this template, I am randomly sampling actions - triplets x- and y-coordinates and diameter of the red ball. Using the simulator's functionality, I am filtering the simulations that are invalid (when parts of the red ball fall outside of the

simulator space or overlap with the other balls). The remaining simulations are further processed for each individual type of tasks that I will be exploring in my research.

To make my data easily usable with PyTorch framework, I need to wrap it using PyTorch Dataset class. I created an abstract `ClassicalMechanicsDataset` class that extends the abstract PyTorch `Dataset`. This abstract class serves as an umbrella data class for all my research, and provides the following functionality:

- It defines an abstract method `generate_data()` that must be implemented by the child classes. In child classes, this method invokes Phyre simulator for simulated scenarios, each of which will be stored in a separate file.
- This class implements a static function `train_test_split(path, test_frac)` that takes the path to raw data files and test fraction as input, and outputs train and test dataset objects. This method can create dataset objects of child classes, so it doesn't need to be reimplemented during inheritance.
- It implements `__len__(self)` method that is required by abstract `Dataset` class by counting the number of stored data files corresponding to the given `ClassicalMechanicsDataset` instance.

For each research task that I have in the process, I am extending ClassicalMechanicsDataset and implementing `generate_data()` method, as well as `__getitem__(self, idx)` method required for implementation by an abstract Dataset class.

Section 4. Free-fall analysis

I start my research with exploring how machine learning models can predict the simplest type of trajectories, namely the trajectory of an object in the free-fall. Just free-fall, however, can be analytically calculated using the equations of kinematics, and, thus, does not require machine learning techniques to predict. To slightly increase the complexity of the task, instead of just a single free-falling object, I am attempting to train models that will predict the movement of a free-falling ball that hits the ground and bounces. This is a more challenging task because such trajectory cannot be explained with a single mathematical equation. Moreover, as we can see from Figure 2, the bounce is not fully elastic, which introduces a hidden physics parameter that machine learning models will need to infer - coefficient of restitution.

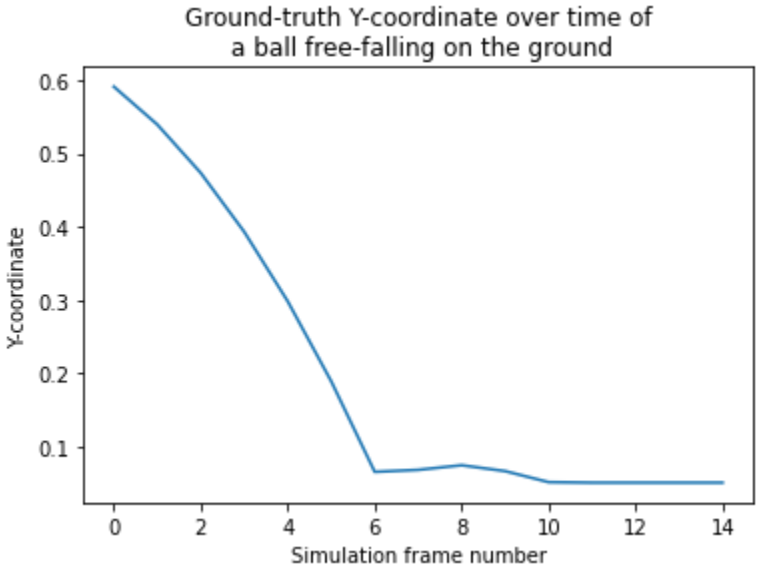


Figure 2. Simulated Y-coordinate of a ball free-falling on the ground. In case of the completely elastic collision, the ball would bounce to the initial height level.

4.1. Experiments with Feedforward Neural Networks

Starting from a naïve approach, I chose a feedforward neural network (FNN) as a first model to investigate patterns in the generated data. This was motivated by two main considerations: (1)

the ability of FNNs to identify complex patterns in data and (2) their relative ease of implementation. As such, an analysis of the results obtained with the help of feedforward networks could potentially guide further research directions, as well as serve as a reference point to compare the performance of more complex models.

4.1.1. Predicting one step at a time <THE ORGANIZATION OF THIS SECTION IS CHANGED>

While I increased the complexity of the problem by adding the bounce to the ball's trajectory, I decided to start with exploring how well linear networks can learn simple kinematic equations.

Movement of the object along a straight line can be described with the following equation:

$$\vec{r} = \vec{r}_0 + \vec{v}_0 \Delta t + \frac{1}{2}\vec{a}(\Delta t)^2 \qquad (1),$$

where $\vec{r}$ and $\vec{r}_0$ represent current and initial positions respectively, $\vec{v}_0$ is initial velocity, $\vec{a}$ is acceleration, and Δt is the time of movement. In case of the free-fall due to gravity, when the movement is strictly vertical, assuming the direction of vertical Y-axis upwards and the direction of initial velocity downwards (which is the case for the free-fall if the movement doesn't start from rest), this equation can be rewritten as

$$y = y_0 - v_0 \Delta t - \frac{1}{2} g(\Delta t)^2 \qquad (2),$$

Equation 2 gives us the relationship between initial coordinate y_0, initial vertical velocity v_0, acceleration due to gravity g, and time step Δt. We can simplify this equation in the following way:

- Given that our simulation is discrete, Δt is defined to be one simulation frame. This simplification removes time dependency from the equation 2.
- Let y_0 be the coordinate of the object on the current frame, and y be the coordinate of the object on the following frame. Then, v_0 is the velocity of the object on the current frame. In such framing, we treat each frame as initial for calculating the following frame.

- Vertical velocity is the first derivative of the y-coordinate. Given the discretized time, and assuming there was some movement preceding the current frame, it can be expressed using the equation of numerical differentiation:

$$v_0 \approx \frac{y_0 - y_{-1}}{\Delta t} = y_0 - y_{-1}.$$

This simplifies the dependency of the following coordinate on speed in equation 2 to the dependency on coordinates on two subsequent preceding frames.

- Acceleration is the first derivative of velocity. We can use the same trick as above to get:

$$g \approx \frac{v_0 - v_{-1}}{\Delta t} = v_0 - v_{-1}.$$

We already expressed v_0 in terms of two subsequent coordinates. In the same way,

$$v_{-1} \approx y_{-1} - y_{-2},$$

which brings us to

$$g = y_0 - 2y_{-1} + y_{-2}$$

This simplifies the dependency of the following coordinate on acceleration due gravity in equation 2 to the dependency on coordinates on three preceding frames.

So equation 2 simplifies to the equation of the form

$$y = f(y_0, y_{-1}, y_{-2}).$$

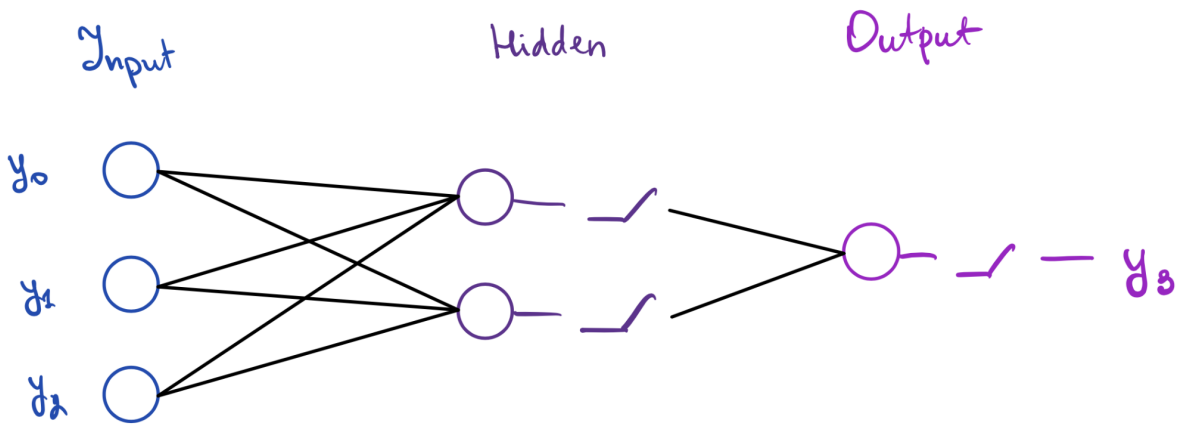
It means that knowing the coordinates on the three preceding frames is enough to predict the coordinate on the next frame. The purpose of the neural network will be to learn this function f. In fact, making the substitution of the equations above to 2 will give us the analytical form of f, so I expect the neural network to learn it without problems. As a result, I expect linear network to make close to perfect prediction of the next coordinate on a free-fall part of the ball movement, and have a noticeable prediction error for the bounce part.

The neural network design can be inspired from the discussion above. The size of the input layer is 3, which corresponds to three subsequent coordinates. The hidden layer has 2 neurons, which are supposed to represent speed and acceleration. Finally, the dimensionality of the output is 1, which corresponds to predicted following coordinate.

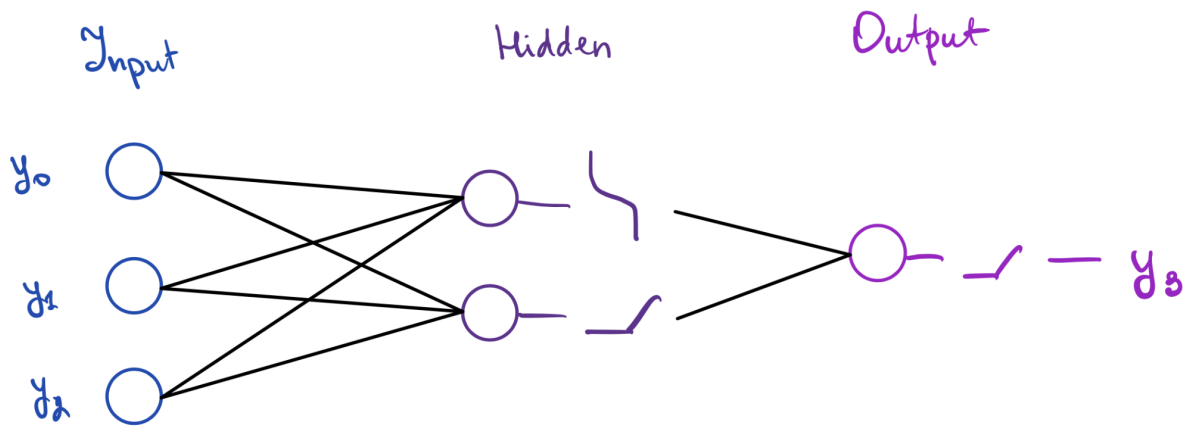
For this scenario, I generated a dataset of 5360 simulations, 1608 of which were used as a test set. I implemented 3 different variations of this simple network that I trained and tested using the simulated data.

The base model is a feedforward network with a single two-neuron hidden layer followed by the ReLU activation. The second model uses ReLU activation for one of the hidden neuron and cotangent activation for the second one to account for velocity being able to take different signs. Finally, the third one uses ReLU activatinos for both hidden neurons but also includes a direct connection from one of the input neurons to the output. Schematically, all three models are shown in Figure 3.

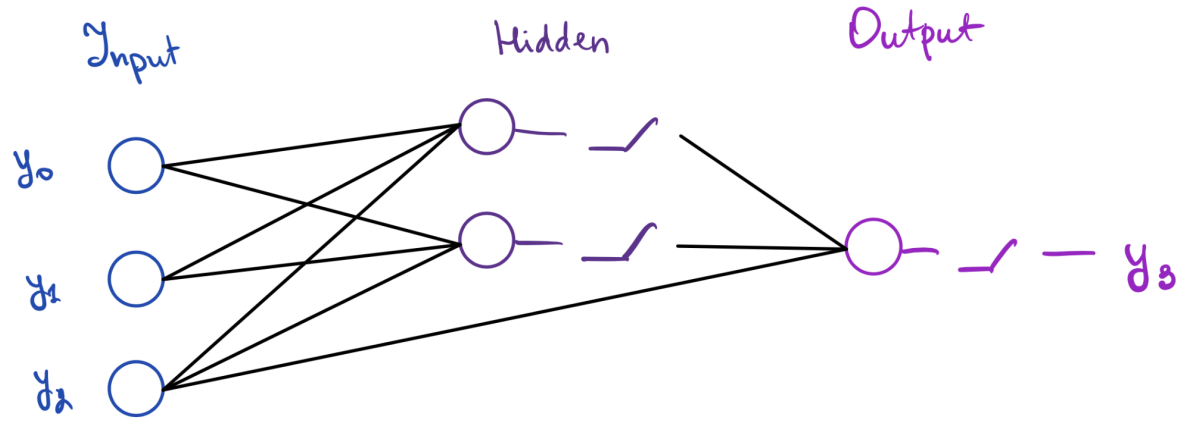
To quantify the model performance, I used Root Mean Squared Error (RMSE) loss.



(a)



(b)



(c)

Figure 3. Feedforward neural network designs for single step prediction. (a) Single hidden layer design with ReLU activations; (b) single hidden layer design with ReLU and Cotan activations; (c) single hidden layer with ReLU activations and direct connection from one of the input neurons to output.

All three models demonstrate similar level of performance, as shown in Table 1, with the second model slightly outperforming the other two. Given that all the numerical data that we get from simulator is normalized between 0 and 1, we can interpret this RMSE value as an average percent error (relative to the scene size) in predicting a single coordinate value. The network design with ReLU and cotangent activation functions seems like the best choice out of these three models.

Model	RMSE
Dense with ReLU activations	0.01827
Dense with ReLU and Cotan activations	0.01155
Dense with direct connection from input to output	0.01819

As expected, these models make nearly perfect predictions of coordinates during the free-fall stage of the ball's movement, while having a noticeable prediction error for the frames where the bounce is observed. It can visually be noticed in Figure 4 that demonstrates the predicted "trajectory map" of the first 10 simulations from the test set by the model with ReLU and cotangent activations. Here, by "trajectory map" I mean the target coordinates within the same simulation.

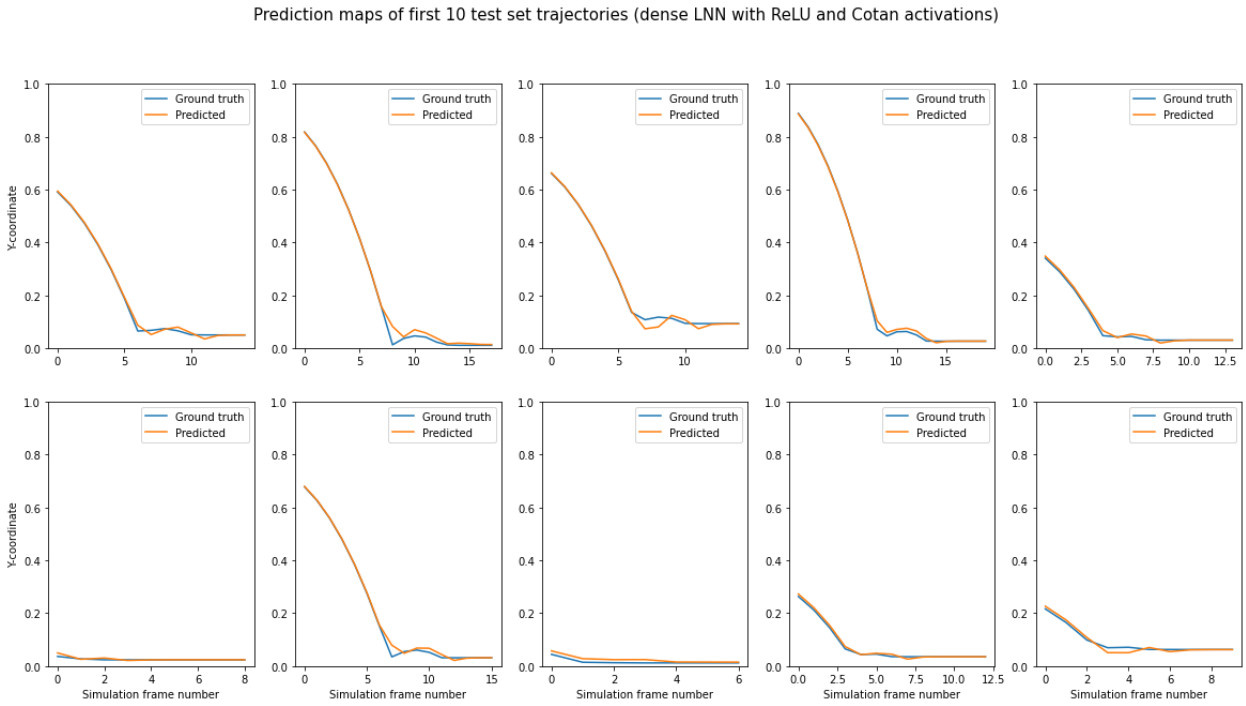


Figure 4. First 10 trajectory maps from the test set predicted by FNN with ReLU and Cotan activations.

4.1.2. Predicting the entire trajectory

While predicting the following coordinate from three preceding is a good first research step, it doesn't have any practical value. Thus, I decided to increase the complexity of the task by bringing it closer to the goal problem - predicting the trajectory of the object from the initial state of the scene. My next step, therefore, is to train neural networks that using only initial state will predict the entire trajectory of a free-falling ball that bounces from the ground.

I started with generating a simulated dataset, where each simulation consists of 25 frames. The y-coordinate of the ball on the first frame is separated as input to the network, while the sequence of the rest 24 coordinates is used as an output. I ended up with a dataset of 5360 simulations, 1072 out of which were used as a test set.

I wasn't sure if a single initial coordinate is enough information for predicting the entire trajectory. To resolve my concern, I decided to overfit my feedforward network to understand what patterns it learns from the train set, hoping that analysis of these patterns will indicate what extra data I need to include.

One of the reasons for overfitting is noise learning, which can be caused by having too large network (Ying, 2019). I relied on this overfitting method by creating a large 5-layer linear network that takes a one-dimensional input, has four hidden layers with 256, 128, 64, 32 neurons respectively, and outputs a vector of size 24.

After training such a model, I got a RMSE of 0.0414. The first 10 predicted trajectories with the ground truth are shown in Figure 5.

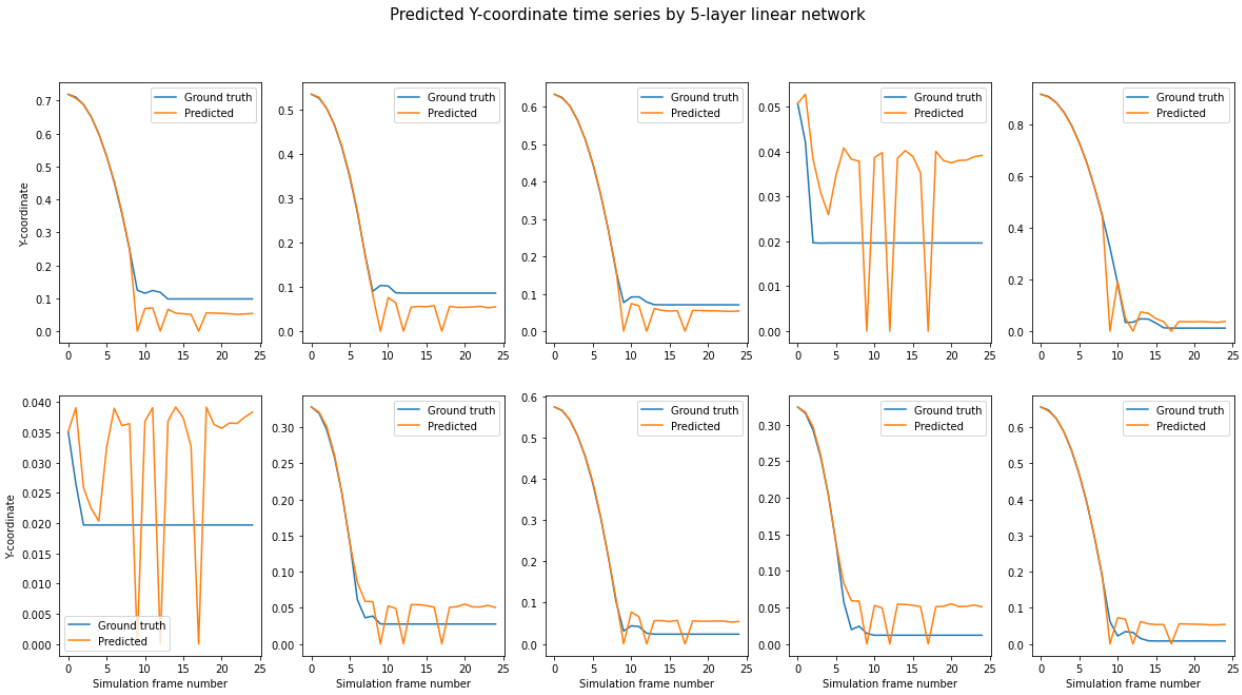


Figure 5. First 10 y-coordinate time series, ground truth and predicted by overfitted 5-layer linear network. Plots are zoomed to see the bounce level.

From Figure 5, which was intensionally zoomed to the scale of the whole trajectory, we can see that while the ground truth trajectory has a bounce on different levels, the model predicts it on about the same level - around 0.04-0.05 units. Seeing these results prompted me that the quantity that makes the ground truth bounce height differ is the diameter of the ball. In the absence of this information, the neural network just learns some descriptive statistics about this quantity from the training set, presumably mean value.

To test my assumption, I modified this model to take two values as input - initial coordinate and the ball's diameter. Testing this model on a corresponding data, I received an RMSE of 0.0239 on the test set, with the first 10 predicted trajectories shown in Figure 6.

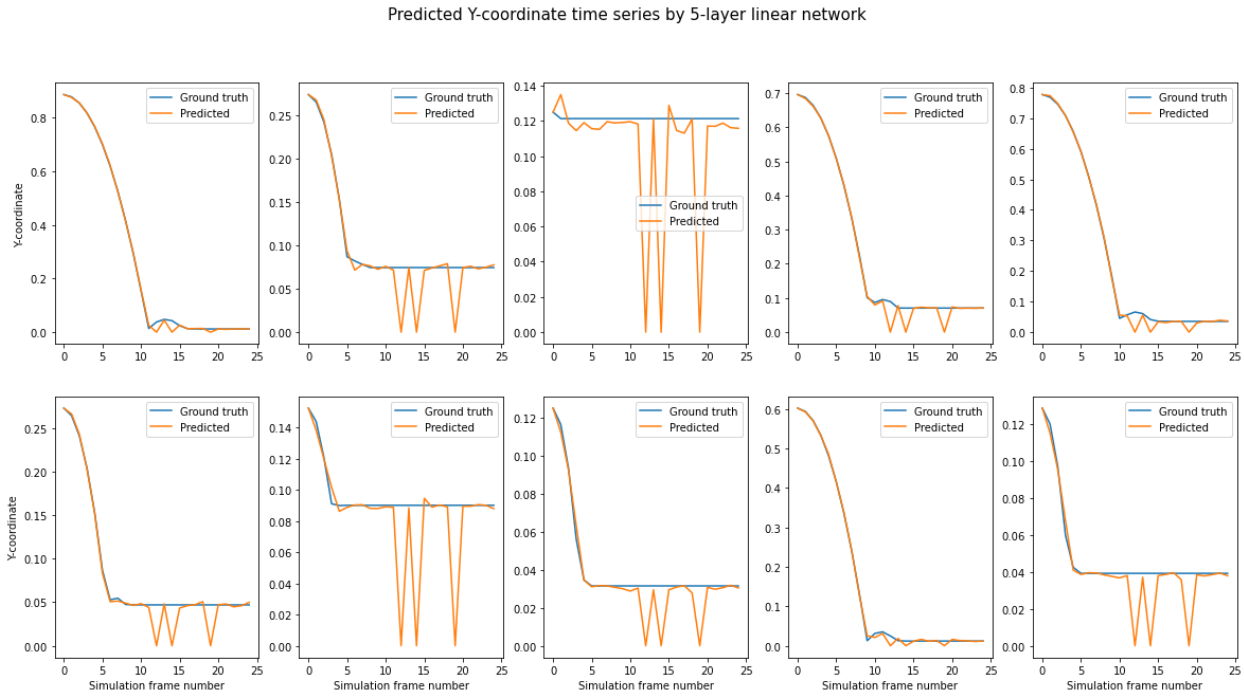


Figure 6. First 10 trajectories from the test set, ground truth and predicted by 5-layer linear model that takes initial coordinate and ball diameter as input. Plots scaled to see the bounce level.

From these results, we can see that the model now predicts the bounce level correctly. The only remaining unusual pattern that we see on the plot is noise, which is not surprising given that the model is overfitted due to large network size (Ying, 2019).

These findings conclude my exploration of feedforward neural networks. As next steps, I will explore how recurrent neural networks will perform on the same kind of tasks.

4.2. Experiments with Recurrent Neural Networks <THIS SECTION HAS SIGNIFICANT CHANGES COMPARED TO FULL DRAFT>

Recurrent neural networks (RNNs) are good for trajectory prediction problems because they are designed to learn from sequence data (Song et. al., 2020). RNNs are able to make predictions based on data from previous time steps, making them well-suited for trajectory prediction problems where the output of one step is used as input for the next step. RNNs are also able to

capture long-term dependencies, which is important for predicting trajectories that span longer periods of time.

4.2.1. Traditional Recurrent Neural Networks

When it comes to predicting trajectories, there are a few neural network models that are worth considering. Vanilla RNNs are a straightforward option that are good at learning patterns over time. However, they can struggle with capturing long-term dependencies due to the vanishing gradient problem. GRUs are a more lightweight alternative to LSTMs and can use gating mechanisms to avoid the vanishing gradient problem. LSTMs are great at capturing long-term dependencies thanks to their memory cells and gates that selectively remember or forget information. All three models have proven to be effective for trajectory prediction tasks in areas such as robotics, autonomous driving, and video analysis. Ultimately, the choice of model depends on the specific task and the properties of the input data.

4.2.1.1. Architectures and Results

To choose an optimal RNN architecture, I decided to run a Grid Search to find an optimal set of hyperparameters, i.e. optimal number of recurrent layers, hidden neurons within each layer, and dropout rate for the number of hidden layers greater than 1. I found that the optimal test performance is achieved with a single recurrent layer that consists of 32 hidden neurons.

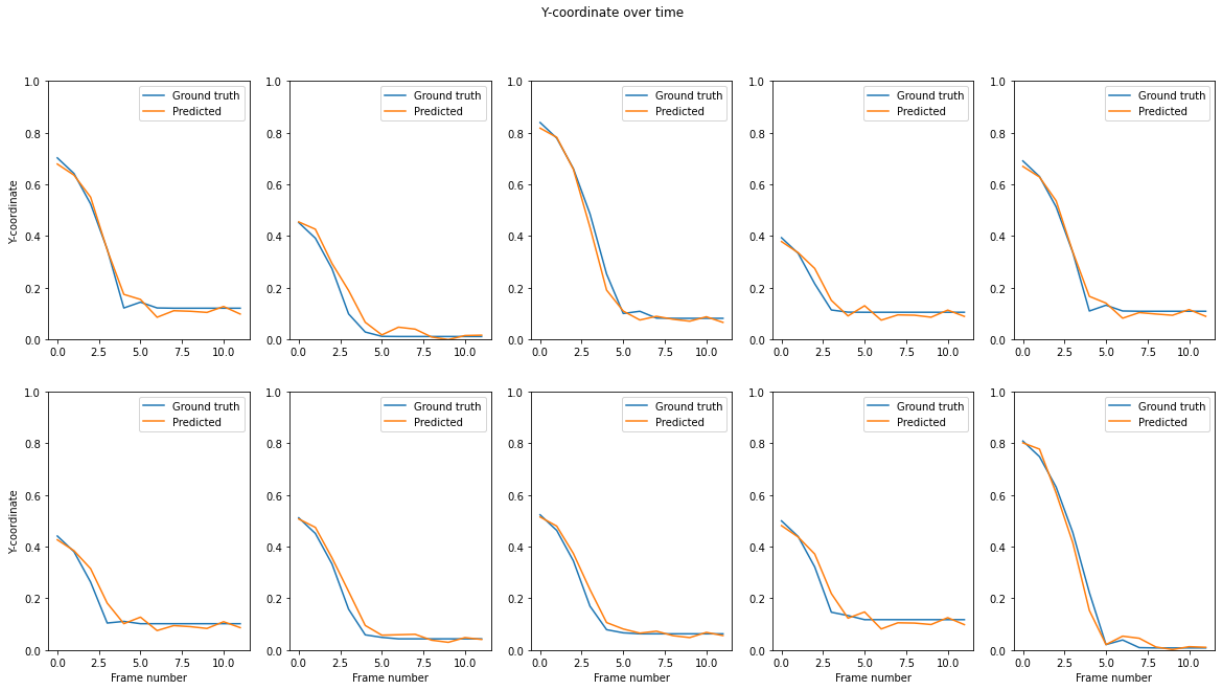
For the reason described in section 4.2.1.2. below, I had to make a slight modification of the model for testing as compared to the training. During the training stage, each of the models (Vanilla RNN, GRU, and LSTM) only consists of input layer followed by recurrent layer and the linear output layer. When the model is run on the test data, I added the post-processing function that bounds the outputs of the neural network between 0 and 1. The rationale behind this post-processing lies in all the data being bound between 0 and 1, so any output of the model that exceeds these bounds will not make sense. Mathematically, this post-processing function can be expressed as

$$f(x) = \max(0, \min(1, x))$$

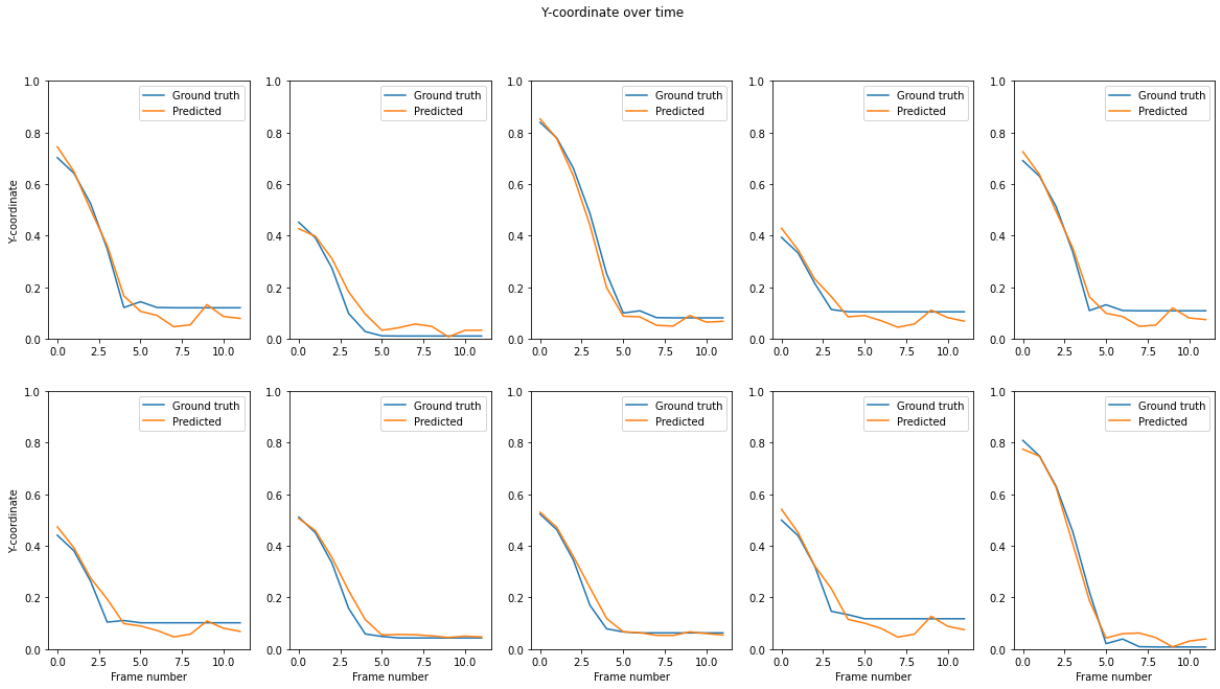
The results of running the chosen recurrent neural networks on one-dimensional free-fall data are given in Table 2. Surprisingly, the best performance out of the three models was demonstrated by the Vanilla RNN. The first 10 predicted Y-coordinate series by each of the models are shown in Figure 7.

Model Type	RMSE
Vanilla RNN	**0.0278**
GRU	**0.0304**
LSTM	**0.0314**

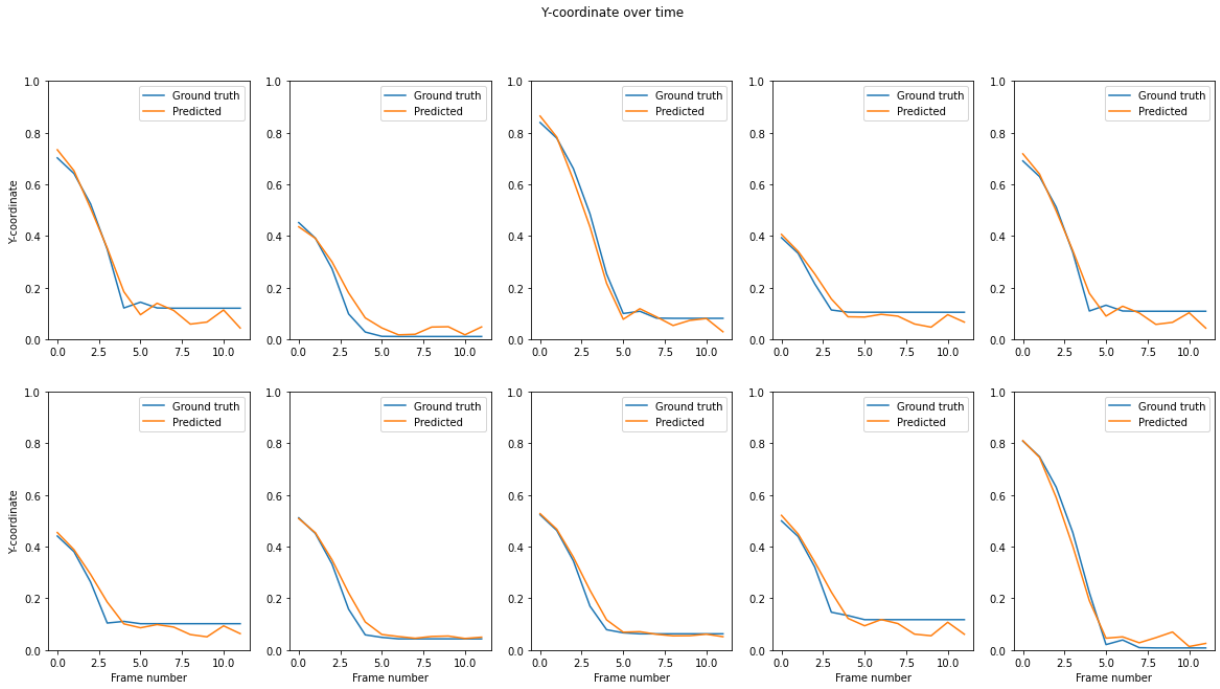
Table 2. Comparison of root mean square errors of each of the RNN types on one-dimensional free-fall prediction.



(a)



(b)



(c)

Figure 7. First 10 predicted trajectories from the test set. (a) Vanilla RNN; (b) GRU; (c) LSTM.

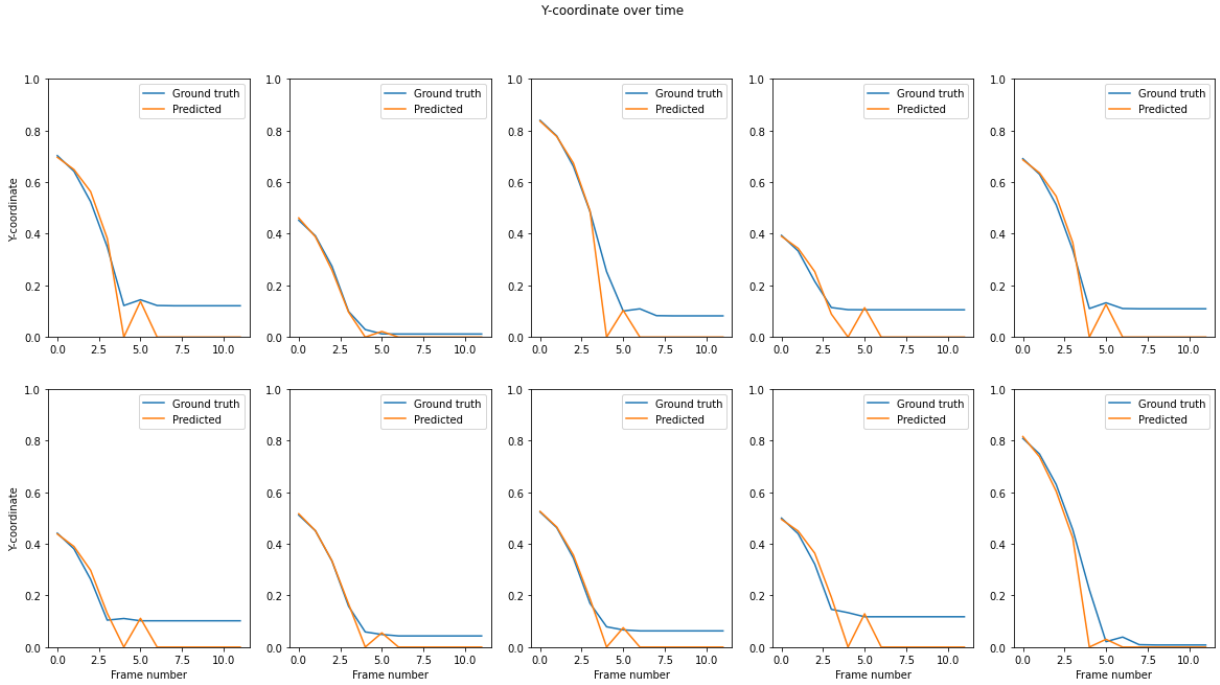
4.2.1.2. Non-convergent models with ReLU layer

One of the discoveries that I had while experimenting with recurrent neural networks is that adding a ReLU activation for the final layer can noticeably decrease the model's performance. My rationale behind adding this function lied in the fact that my data is bound between 0 and 1. Adding ReLU to the final layer would prevent the model from predicting the negative values, which in theory was supposed to improve the model performance. In reality, however, the performance of each type of the models decreased. Quantification of this decrease is given in Table 3, while the first 10 trajectories from the test set are shown in Figure 8. By looking at these trajectories, we can see that the models are still predicting reasonable trajectories; however, these predictions have a noise similar to that produced by overfitted feedforward neural network.

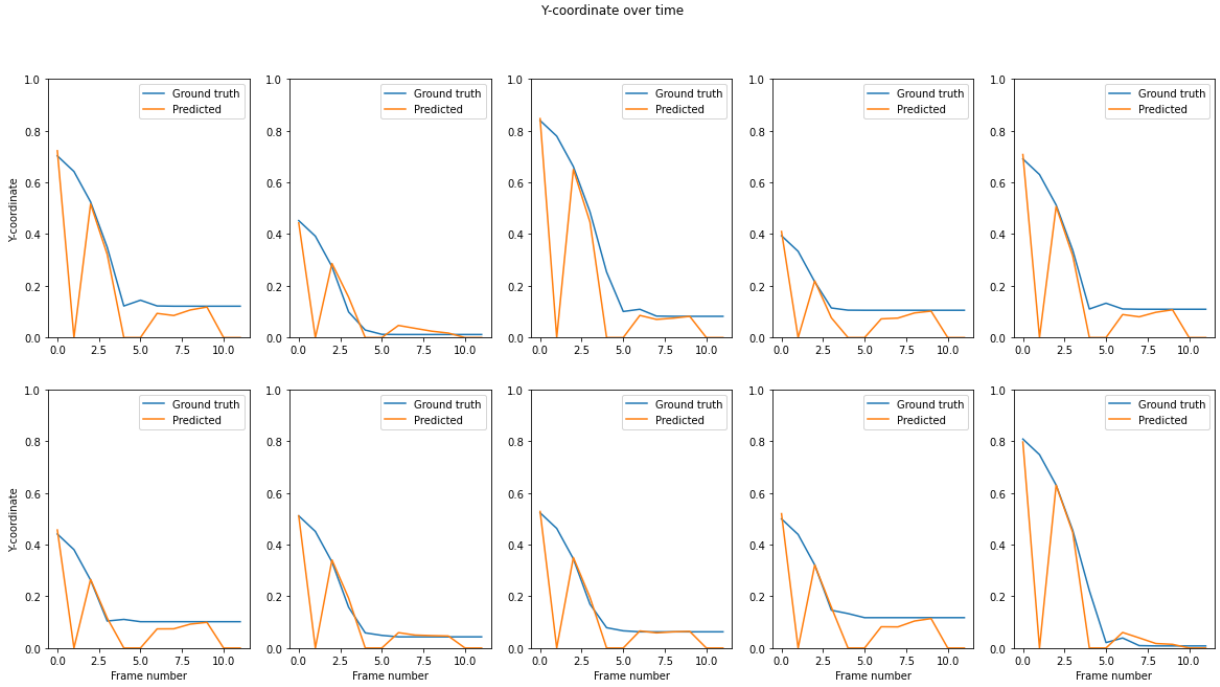
My hypothesis about the reason for this performance decreased is that adding the ReLU activation changes the shape of the error function. While without ReLU the error function might have been convex (or at least have a small amount of similar local minima), adding ReLU could have changed the shape of the error function to have more local minima. Landing in one of such local minima would result in the output that looks somewhat reasonable but is far from ideal.

Model Type	**RMSE**
Vanilla RNN with ReLU	0.1375
GRU with ReLU	0.1143
LSTM with ReLU	0.1869

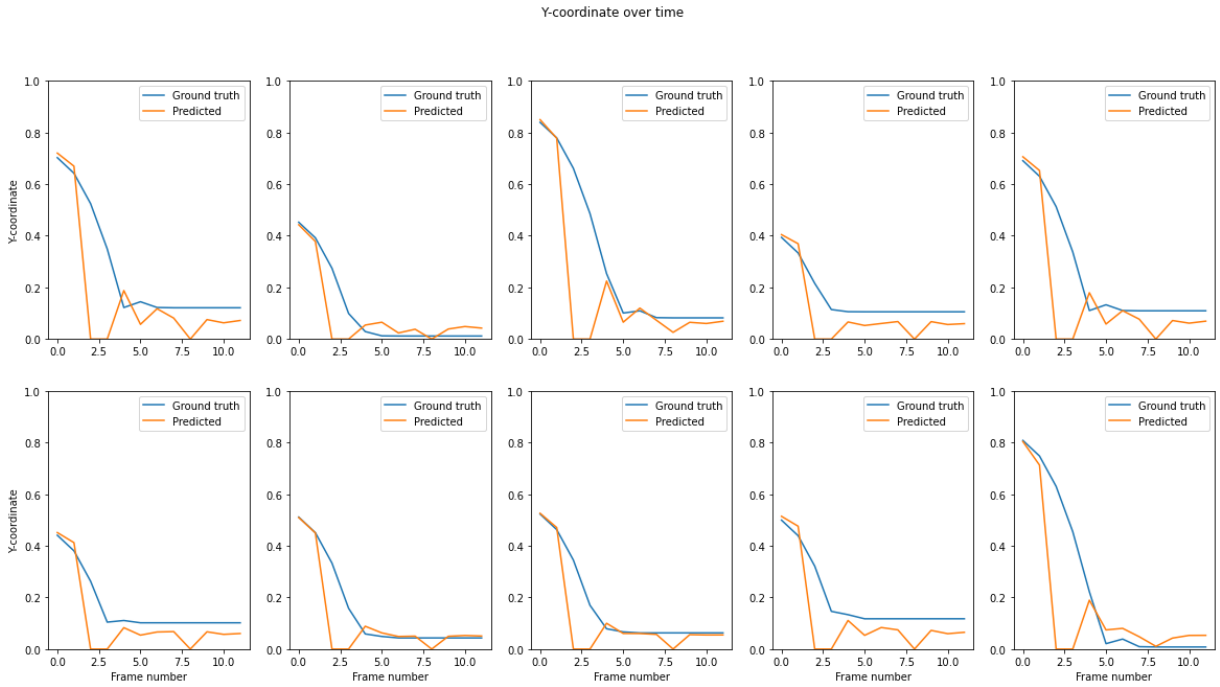
Table 3. Comparison of root mean square errors of each of the RNN models with ReLU layers on one-dimensional free-fall prediction.


Y-coordinate over time
Ground truth
Predicted
Y-coordinate
Frame number

(a)


Y-coordinate over time
Ground truth
Predicted
Y-coordinate
Frame number

(b)



(c)

Figure 8. First 10 predicted trajectories from the test set. (a) Vanilla RNN with ReLU; (b) GRU with ReLU; (c) LSTM with ReLU.

4.2.2. Reservoir Computing Models

While the traditional recurrent neural networks demonstrated a good performance, just exploring them is not a compelling research on RNNs' ability to predict trajectories. In general, if the training data is long, it becomes challenging to train RNNs due to vanishing and exploding gradients problem (Bengio et. al., 1994). Presumably, this is what happens when I add a final ReLU layer to RNN models.

The problem of training RNNs using long data is solved by the Reservoir Computing (RC) paradigm that is being actively developed over the past decade (Song et. al., 2020). Echo State Network (ESN), the most widely know RC model, "has been recognized as the most efficient network structure for training RNNs" (Malik et. al., 2016). It has also been proven to be effective for chaos prediction (Wolchover, 2018), which makes it a perfect choice for trajectory prediction problem. This is why I decided to explore various ESN architectures, including deep ESN models, for trajectory prediction tasks.

A classical Echo State Machine consists of the following components (Malik et. al., 2016):

- Input layer, neurons of which are randomly connected with the reservoir neurons.
- Reservoir - a recurrent neural network with randomly (usually not sparsely) interconnected neurons. The connectivity of neurons within the reservoir is usually quite small (around 10%). The weights of the reservoir, as well as the weights between the input and the reservoir layers, are fixed and not trainable.
- Output (readout) layer - a linear layer densely connected with the reservoir. The weights between the reservoir and the output layer are trainable and are trained in a single epoch. Typically, Ridge Regression is used as readout layer as it is optimal for highly correlated inputs, as it happens in the reservoir, neurons of which receive responses from each other (Lukosevicius, 2012).

A schematic representation of the Echo State Network with 4-dimensional input, reservoir of size 7, and one-dimensional output is shown in Figure 9 (Verzelli et. al., 2019).

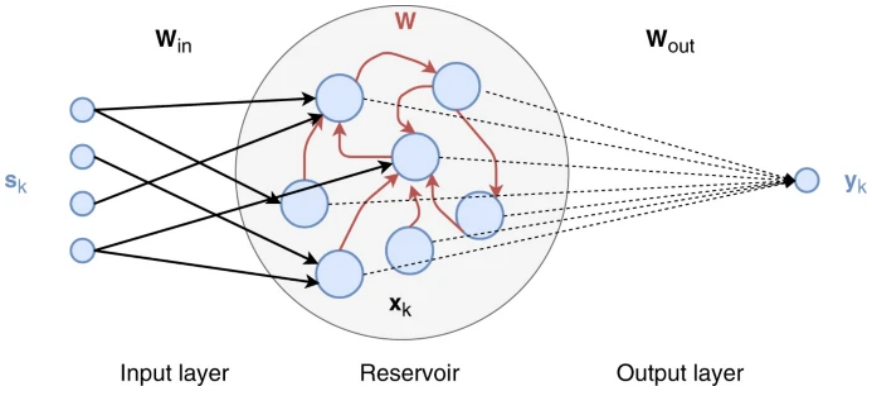


Figure 9. Echo State Network diagram with 4-dimensional input, reservoir of size 7, and one-dimensional output. Retrieved from "Echo State Networks with Self-Normalizing Activations on the Hyper-Sphere" (2019) by Verzelli, P., Alippi, C., and Livi, L.

4.2.2.1. Architectures and Results <THE ORGANIZATION OF THIS SECTION IS CHANGED>

I starting with testing out how different Echo State Network architectures will perform on predicting one-dimensional free-fall with a bounce. To implement the models, I used a Reservoirpy framework created by Trouvain et. al. (2020).

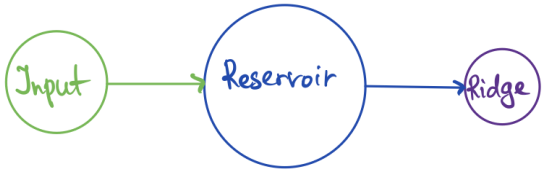
To test the ability of ESNs to predict trajectories, I implemented several different architectures. I started with a simple single reservoir Echo State Network, the number of hyperparameters of which allows to run Grid Search to find their optimal set. I found out that the optimal hyperparameters of ESN with Ridge Regression Readout for free-fall prediction are:

- Reservoir size of 70 neurons;
- Leaking rate of 0.7;
- Spectral radius of 0.95;
- Ridge coefficient of 0.01.

A number of papers, among which are papers by Malik et. al. (2016) and Song et. al. (2020), suggest that Deep Echo State Networks show better performance when compared to classical ESNs. While I don't have enough computational resources to run a Grid Search for hyperparameter optimization in Deep ESNs, I decided to take the optimal set that I found for the simple ESN and use it while creating Deep ESN models. The deep models that I tried for the free-fall prediction are:

- Sequential ESN, inspired by Multilayered ESN presented by Malik et. al. (2016). I modified the model presented in this paper by adding a readout layer between each pair of reservoirs. The rationale dehind it is that stacking several reservoirs without the readout layer is not different in principle from having a single reservoir with larger size and modified connectivity;
- Parallel ESN, suggested by Song et. al. (2020);
- Grouped ESN, suggested by Song et. al. (2020).

All four type of ESN (classical plus three deep architectures) are schematically shown in Figure 10.



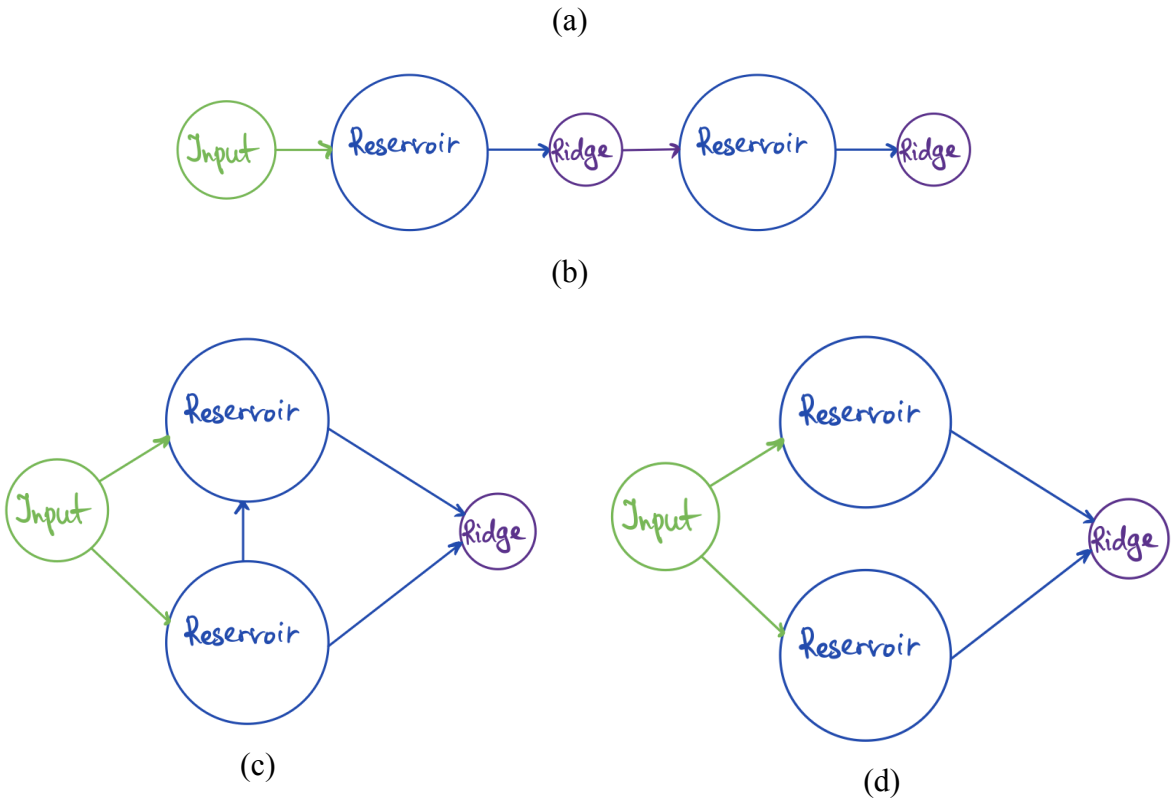


Figure 10. Implemented Echo State Network Architectures. (a) Classical ESN; (b) Sequential ESN; (c) Parallel ESN; (d) Grouped ESN.

It is easy to note that if any of the deep architectures will consist of a single reservoir, it will correspond to a simple ESN. To test the performance of these models, I trained a simple ESN as well as all the deep models with the number of reservoirs between 2 and 10. The performance of each of all of these models on the test set is shown in Figure 11, where the first point in every plot corresponds to the performance of a simple ESN.

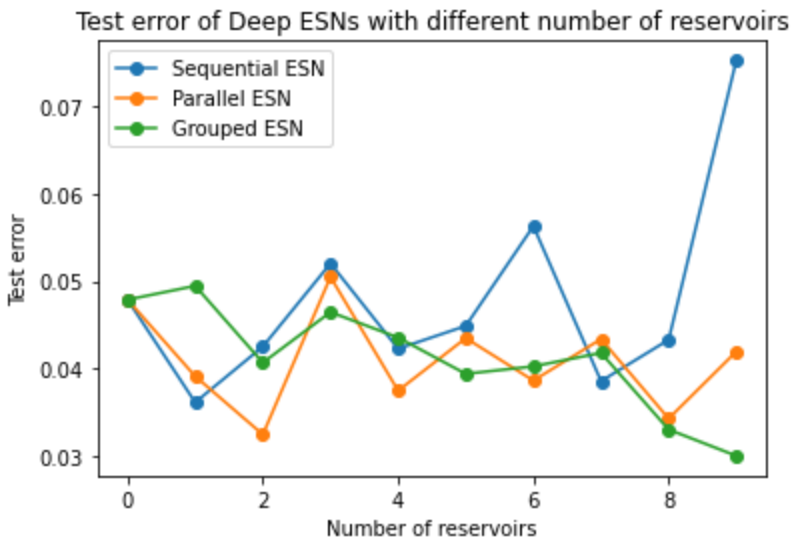
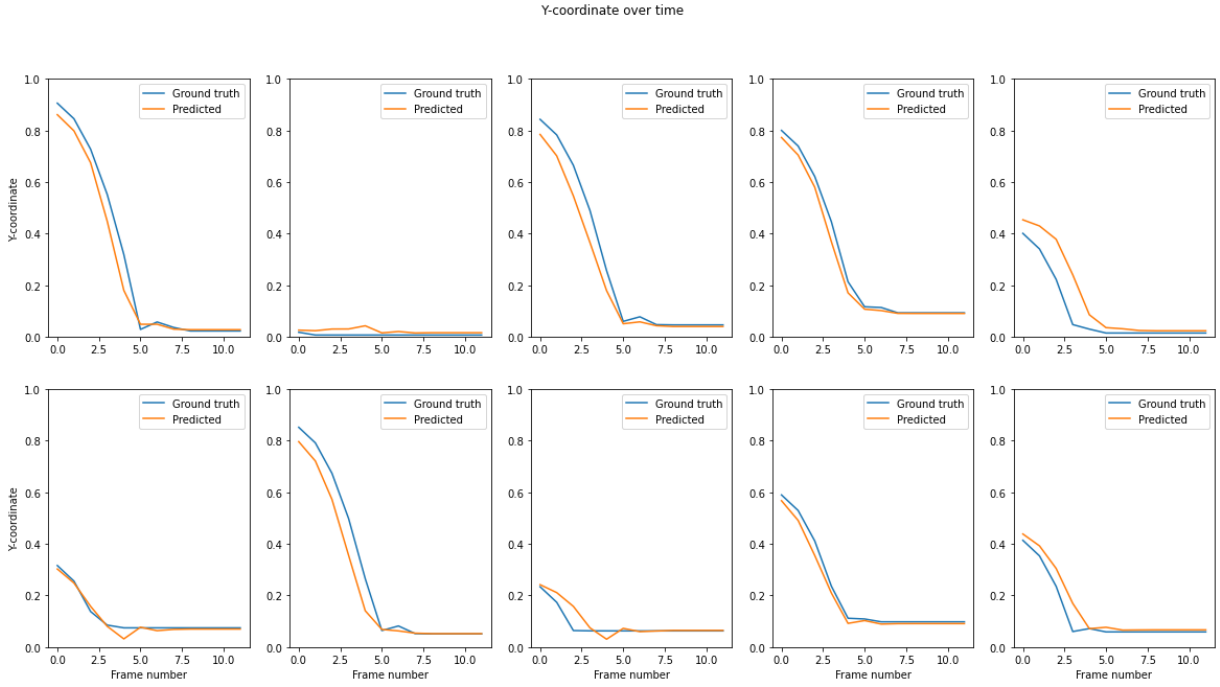
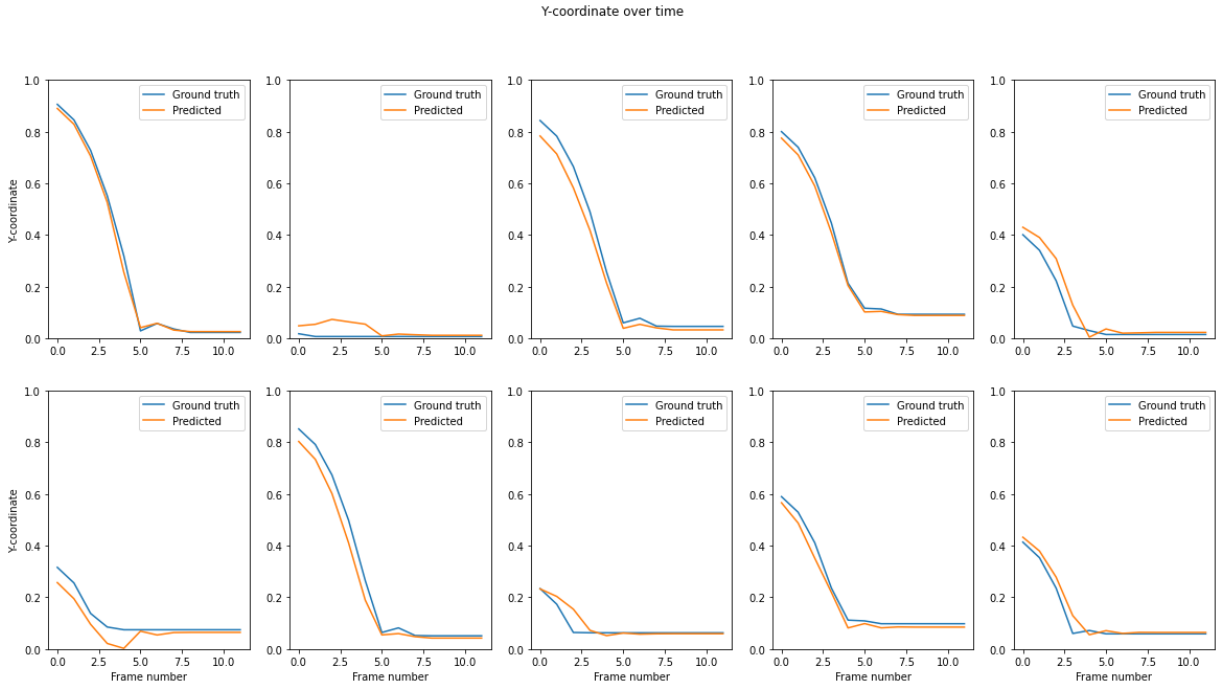


Figure 11. Performance of the Echo State Networks on the test set as a function of the number of reservoirs.

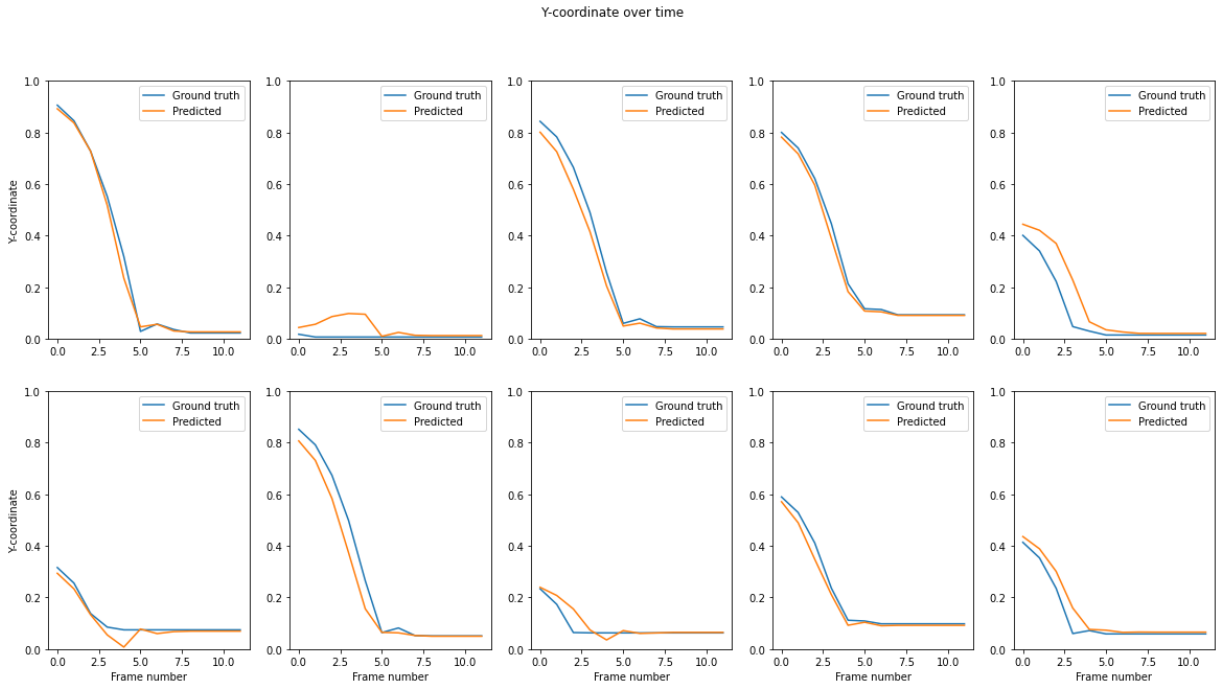
We can see that, perhaps with the exception of Grouped ESN, increasing the number of reservoirs doesn't improve the test set performance. Moreover, we can see that all of these models perform slightly worse when compared to the traditional recurrent neural networks. The first 10 test set trajectories for simple ESN and each of the deep ESN with two reservoirs are shown in Figure 12.


Y-coordinate over time
Ground truth
Predicted
Y-coordinate
Frame number

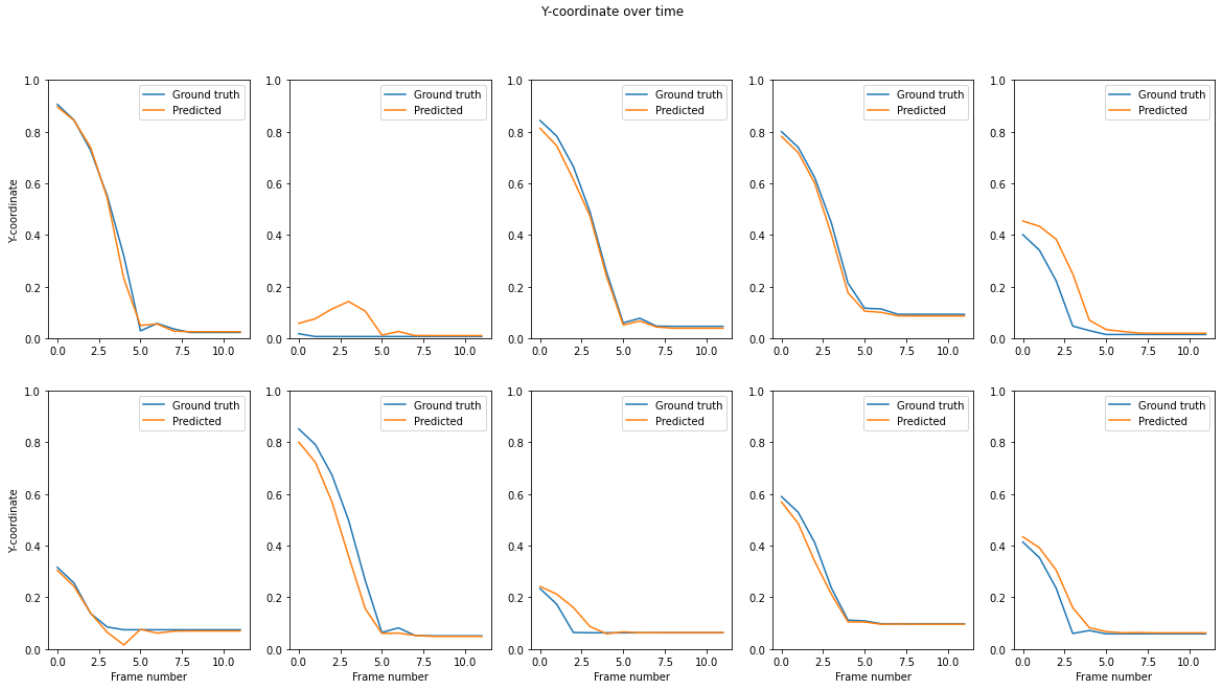
(a)


Y-coordinate over time
Ground truth
Predicted
Y-coordinate
Frame number

(b)



(c)



(d)

Figure 12. First 10 test set predicted trajectories by Echo State Networks. (a) Simple ESN; (b) Sequential ESN; (c) Parallel ESN; (d) Grouped ESN.

4.2.2.2. ReLU layer experiment <THIS SECTION IS NEW>

After noticing that adding ReLU layer prevents training of traditional RNNs to converge properly, I decided to check if adding this layer also affects the performance of Echo State Networks. To do it, I decided to run an experiment on two ESN models, the only difference between which is the presence or absence of the final ReLU layer. My null hypothesis is that the mean test errors will be the same for both models, with an alternative hypothesis that the mean error will be less for the model *without* the final ReLU layer. I decided to make a decision on whether to reject the null hypothesis based on the t-test with 0.05 significance level.

To perform the test, I generated two relatively small datasets (1110 free-fall simulations). Then, I made 80 random train-test splits with the test fraction of 0.2 (222 simulations), and for each of these splits I trained both models and saved the resulting test loss. A sample size of 80 for each model is chosen for Central Limit Theorem to hold properly. Once training and calculating test loss is complete for each split and each model, I run the difference of means test. I got a p-value of 0.22 on the one-tailed test, which is insignificant on the chosen significance level. This means that the test failed to reject the null hypothesis, or, in other words, presence of the final ReLU layer doesn't make the performance of the model worse. The histogram of the test losses is given in the Figure 13.

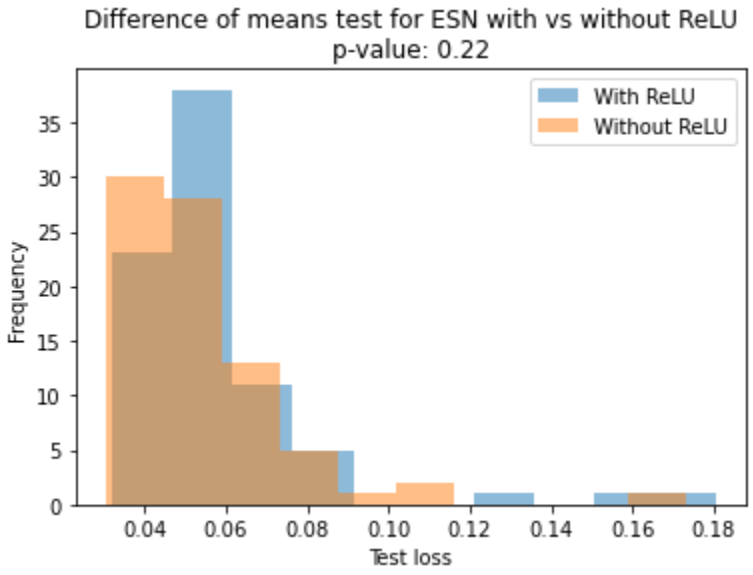


Figure 13. Histogram of test losses for ESNs with and without final ReLU layer.

4.2.3. Generalizing free-fall scenarios

Now that the one dimensional free-fall is explored, there are a few more intermediate steps that need to be done to come closer to the trajectory prediction tasks in scenarios with collisions.

4.2.3.1. Free-fall in two-dimensions

Up to this point, I was using a single Y-coordinate, knowing that X-coordinate stays constant during the free-fall. Now I want to test if my models will be able to infer that this coordinate is constant.

For this purpose, I generated a new dataset that includes X-coordinate into the output. I simulated 52,810 scenarios, 10562 out of which were used as a test set.

Running grid search for two-dimensional free-fall prediction resulted in the same set of hyperparameters for Echo State Network models. For traditional Recurrent Neural Networks, while a single recurrent layer was still found to be optimal, the number of hidden layers needs to be increased to 64.

An interesting finding is that for all types of models that I tried, the test error noticeably decreased, although the number of parameters that need to be predicted increased two times. While it is incorrect to compare the errors between one and two-dimensional free-fall prediction *to make any inference*, I was expecting the error to rise due to the increased output dimension.

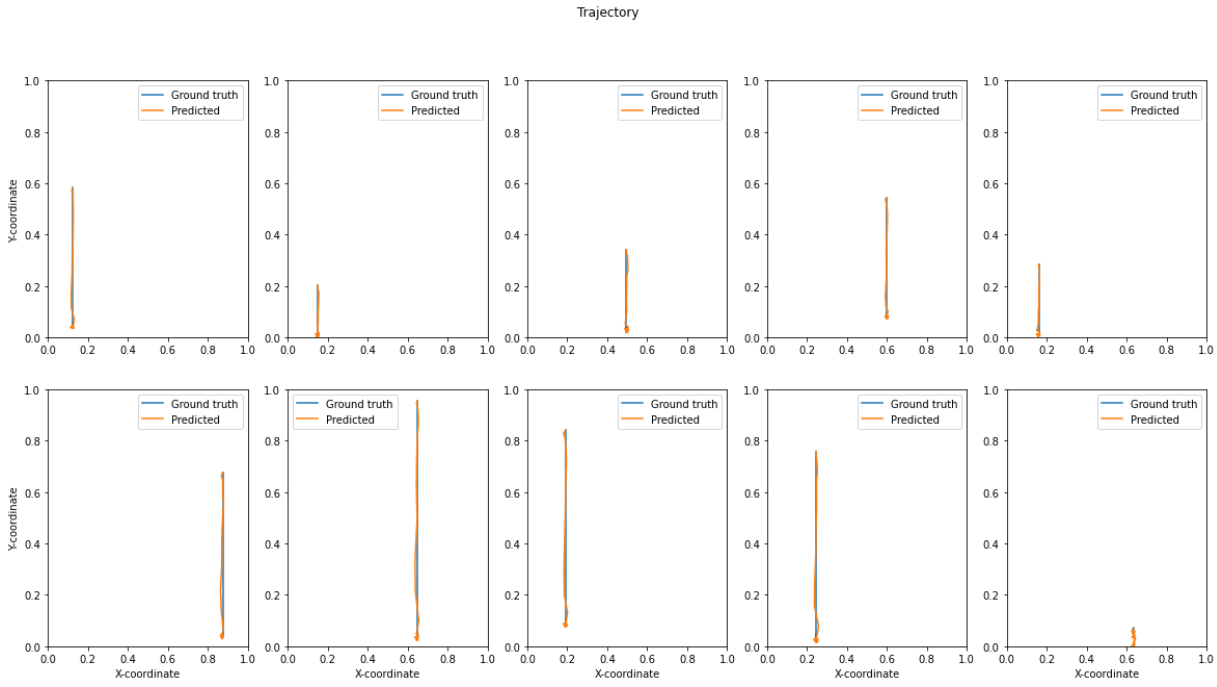
The test set error for each of the models is given in Table 4. The first 10 predicted y-coordinate sequences and full trajectories from the test set are shown in Figure 14.

Model	**RMSE**
Vanilla RNN	0.0182
GRU	0.0160
LSTM	0.0164
ESN	0.0320
Sequential ESN	0.0413

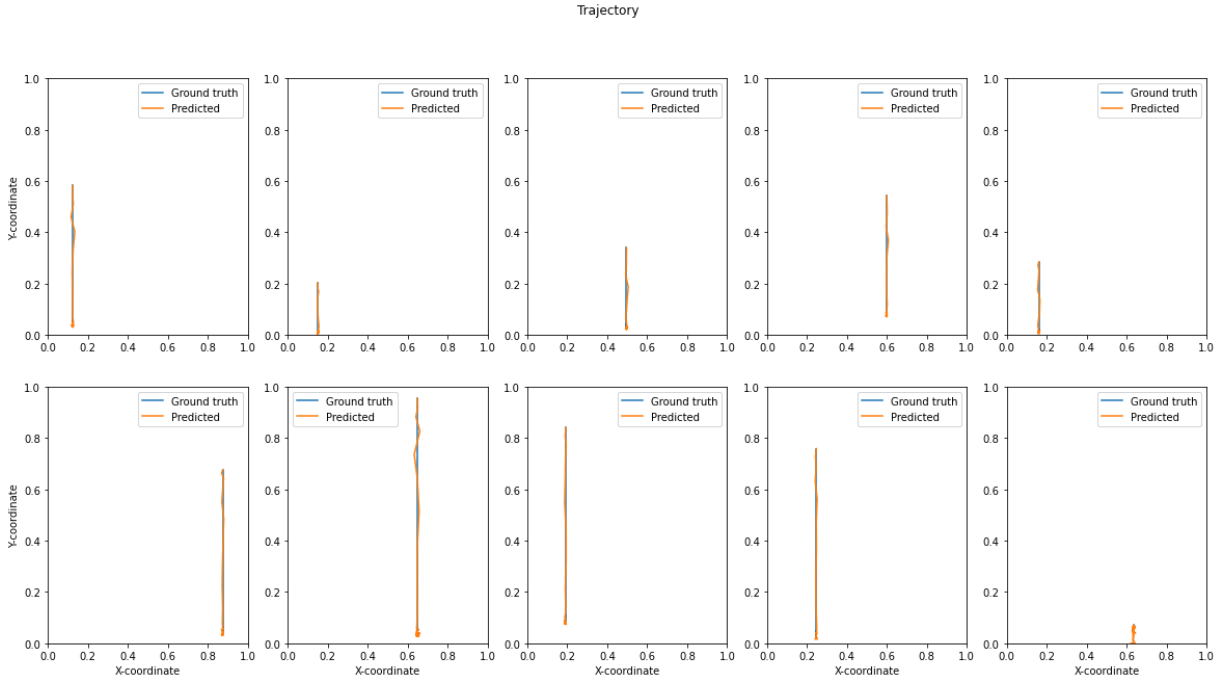
Parallel ESN	0.0314
Grouped ESN	0.0314

Table 4. Root mean square error of each model on the test set for two-dimensional free-fall prediction.

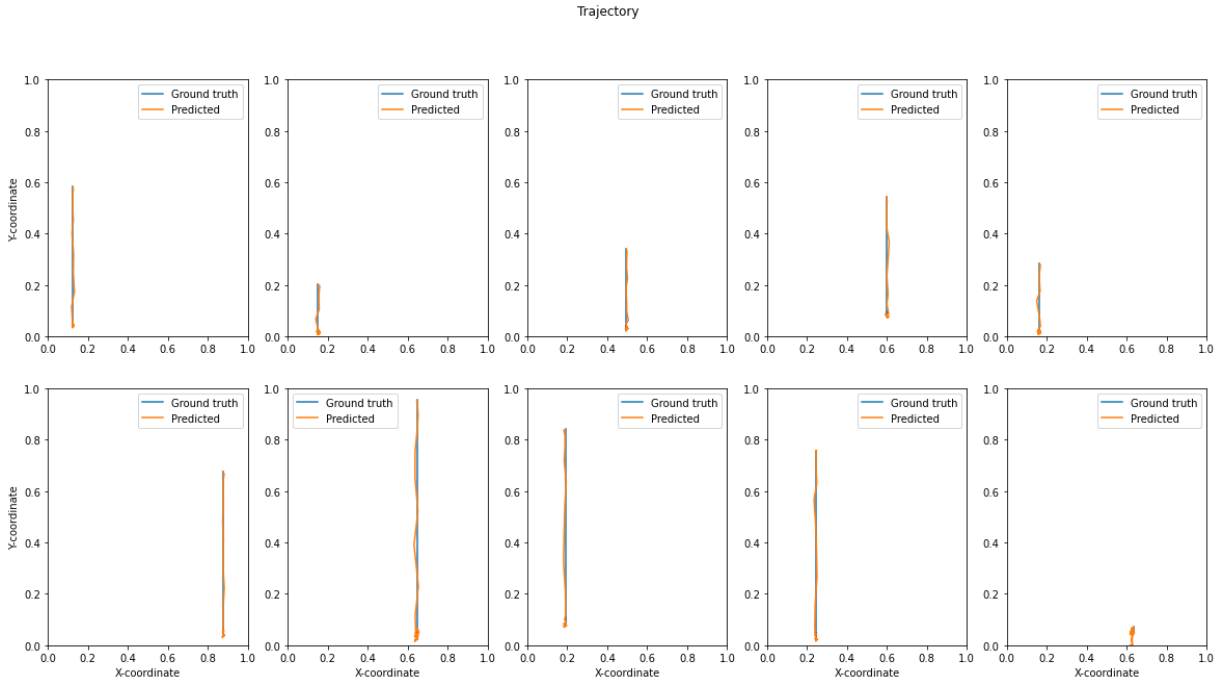
We can see that the Reservoir Computing models perform noticeably worse compared to the traditional RNNs. From Figure 14, however, we can notice a significant difference in the predicted trajectories: while traditional RNNs do a good job predicting the free-fall trajectories very closely, we can still see oscillations in predicted values for both coordinates. Meanwhile, ESNs do a worse job resembling the trajectories closely; however, they perfectly learn that free-fall is happening along the straight line.



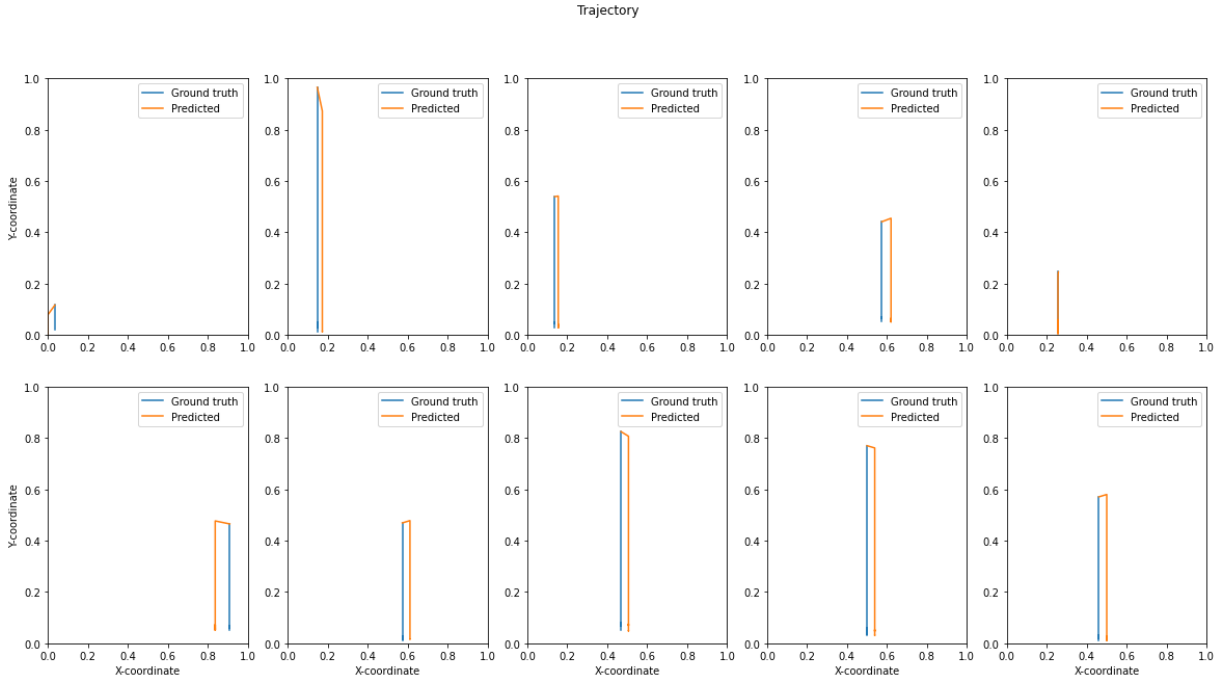
(a)


Trajectory
Ground truth
Predicted
Y-coordinate
X-coordinate

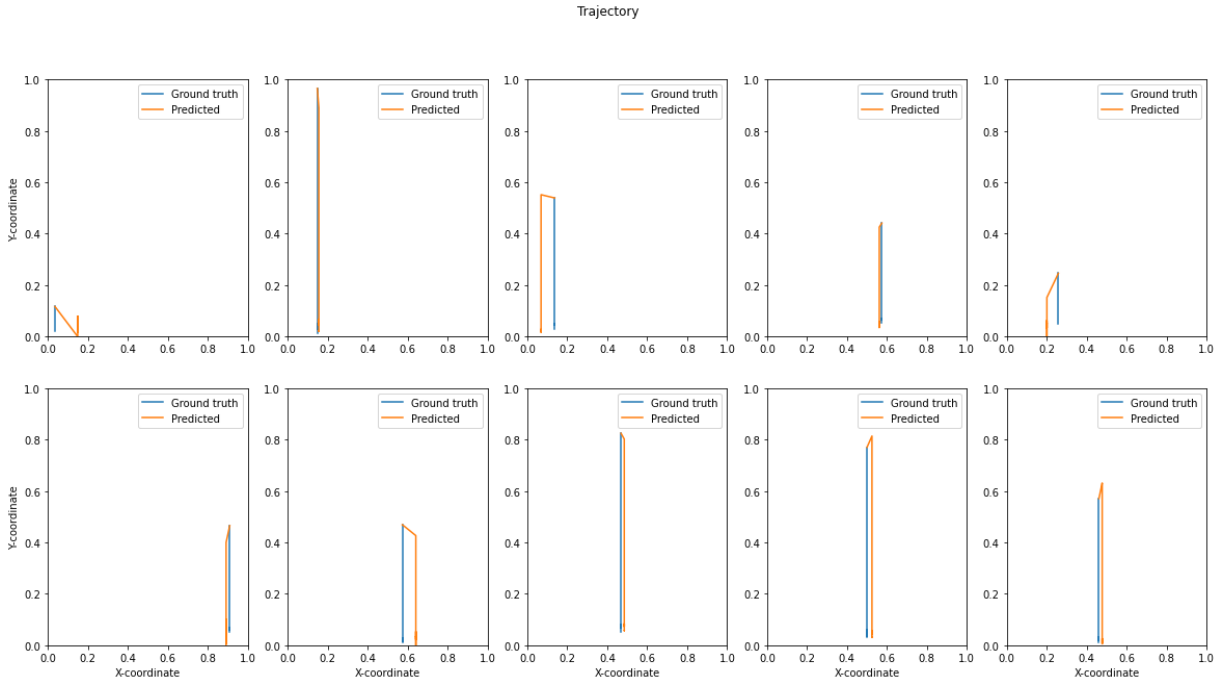
(b)


Trajectory
Ground truth
Predicted
Y-coordinate
X-coordinate

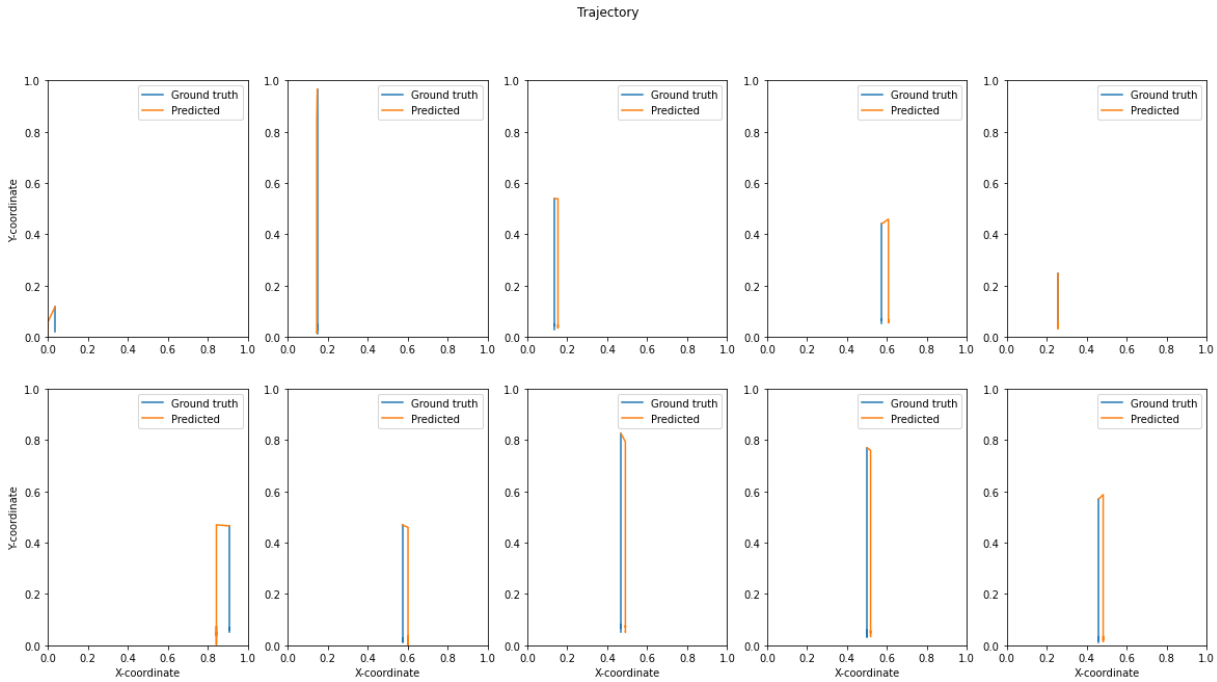
(c)


Trajectory
Ground truth
Predicted
Y-coordinate
X-coordinate

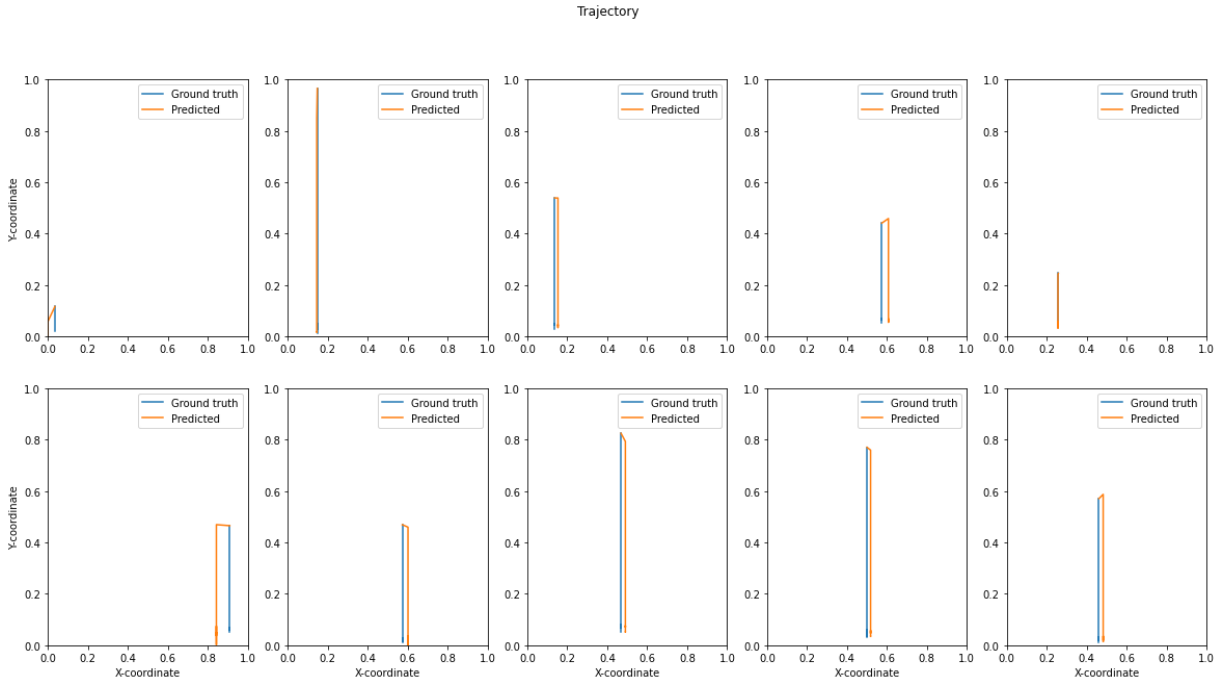
(d)


Trajectory
Ground truth
Predicted
Y-coordinate
X-coordinate

(e)



(f)



(g)

Figure 14. Predicted two-dimensional free-fall trajectories by each of the models. (a) Vanilla RNN; (b) GRU; (c) LSTM; (d) ESN; (e) Sequential ESN; (f) Parallel ESN; (g) Grouped ESN.

4.2.3.2. Scenarios with multiple balls

The last layer of complexity that I can add to predicting the free-fall trajectory of the ball is adding an information about other balls on the scene to the input.

Table 5 shows free-fall prediction RMSE value of each of the models on the test set. As we can see, adding extra information to the input did not affect the performance of the traditional models, while the performance of Reservoir Computing models actually improved. My hypothesis is that the extra input serves as extra variable for regularization in Ridge Regression, which positively affects the model performance.

Model	**RMSE**
Vanilla RNN	0.0194
GRU	0.0165
LSTM	0.0158
ESN	0.0251
Sequential ESN	0.0374
Parallel ESN	0.0257
Grouped ESN	0.0257

Table 5. RMSE of predictions on the test set for free-fall scenarios with multiple balls.

4.3. Intermediate Conclusion

In this section, I explored how different types of Recurrent Neural Networks can solve the task of predicting the trajectory of the ball that is free-falling and then bouncing from the ground. After comparing the performance of traditional RNNs versus the Reservoir Computing models, I can claim with certainty that traditional RNNs are doing better job minimizing the prediction error. By assessing the visualizations of models' predictions, it might look like RC models might better resemble the "Physics" of the free-fall. A good manifestation of it is demonstrated in two-dimensional free-fall prediction: while the error is minimized better by traditional RNNs, they fail to understand that the free-fall happens strictly along a straight line. RC models do learn

this patter; however, they fail to correctly infer the x-coordinate along which the free-fall is happening.

Section 5. Movement with collisions

Now that the research of the free-fall scenarios is complete, it is time to move to the actual subject of my research - scenarios where three balls fall and collide with each other. The complexity of these scenarios is much larger compared to the free-fall, so I will need to use much larger datasets. However, larger datasets also mean increase in the training time, which makes it difficult to perform the hyperparameter optimization. Thus, I will need to rethink the way I use my data and optimize my models. The new methods will be discussed in this section.

5.1. Optimizing data engineering <THIS SECTION HAD SOME CHANGES>

There are two aspects that need to be considered when changing the data engineering approach. The first one is limiting the amount of free-fall scenarios: while now I don't constrain my scenarios to be free-fall, simulating randomly sampled actions will still include a lot of free-fall scenarios. The fraction of these scenarios will need to be limited, which will be discussed in section 5.1.1. The second aspect regards the wise data usage for hyperparameter optimization and training the models, which will be discussed in section 5.1.2.

5.1.1. Limiting the fraction of free-fall scenarios <THIS SECTION IS NEW>

As discussed in section 3.2, to generate the datasets, I randomly sample the actions (position and size of the red ball) and use these actions on each of the tasks in Phyre template with code name '00000'. However, now that the scenarios are not constrained to free-fall, the fraction of free-fall simulations still remains quite high. This means that training the model with such data will bias it towards predicting the free-fall. Thus, the number of free-fall scenarios must be constrained in the generated dataset. Unfortunately, Phyre doesn't provide an API for implementing such constrains. Instead, here is the approach that I am taking.

Let's define t as the total number of valid simulations, f as the number of free-fall scenarios among t, ϵ as the desired fraction of free-fall scenarios, and r as the number of free-fall simulations that need to be removed from the pool. We can detect the free-fall simulations as those where the x-coordinate of the red ball stays the same.

Now, to limit the fraction of the free-fall simulations to ϵ, we need to remove such number r of them that the following equation holds:

$$\frac{f-r}{t-r} = \epsilon$$

From this equation we can express the number r of the simulations that need to be removed.

$$r = \frac{f-\epsilon t}{1+\epsilon}$$

Thus, the strategy is to simulate some number of scenarios, labeling the free-fall simulations in some special way. Once the whole dataset is generated, randomly sample r free-fall simulations, according to the chosen fraction ϵ, and remove them from the dataset.

5.1.2. Changing the data usage for training vs. hyperparameter optimization

Scenarios where the movement of the ball is not constrained in 1 dimensions and interacts with other balls are much more complicated. Training the machine learning model to predict the movement in such scenarios will require significantly more data. However, more data means longer training time, which in fact results in exponentially longer Grid Search computations.

To tackle this problem, I decided to generate two separate datasets for training and hyperparameter tuning, the only difference between which is their size. For hyperparameter tuning, I will create a relatively small dataset. Obviously, lack of data will result in inability to train the models properly, i.e. in their poor final performance. However, my assumption is that despite every model being undertrained, their relative performance when trained on smaller dataset should preserve compared to the larger dataset. Once I find the set of optimal hyperparameters using a smaller dataset, I can use it to train the model using the bigger dataset, which must include a larger variety of scenarios.

5.2. Predicting single ball trajectory

NOTE (this will be cut in the final submission):

I was not yet able to obtain the proper results for this section because trying out neural networks took more time than I expected. While training RC models takes quite a reasonable amount of time on any dataset size (as the training is happening in a single epoch), generating a huge dataset and training the traditional models is a very computationally expensive process.

For now, in this section are presented the results of training all the models on a smaller dataset and with hyperparameters optimized “on eye”. I expect that I will have something more reasonable to show by the defense meeting, or, worst case, it will be ready for the final submission.

To train the models for trajectory prediction in scenarios with collision, I generated a dataset of 91,256 simulations, 18,251 out of which were held out as a test set. I then trained the neural networks and obtained noticeably different results.

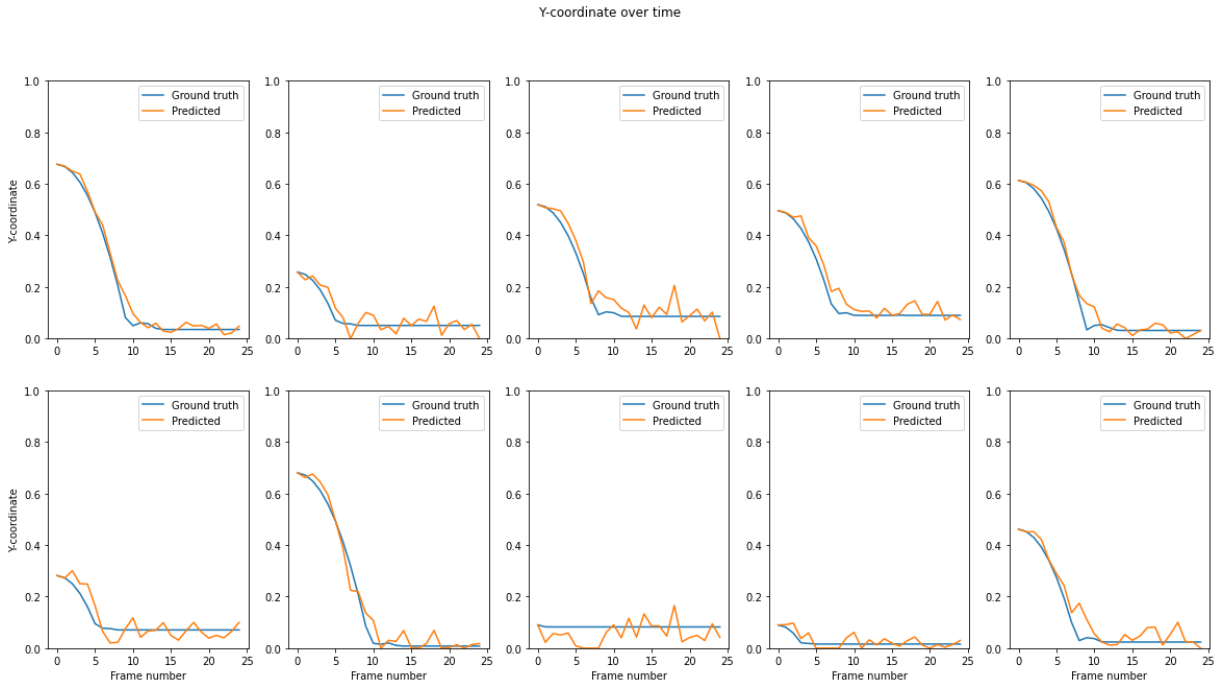
For traditional RNNs, I increased the number of neurons in the hidden layer to 96 (after trying out different layer size, but not through the proper Grid Search yet). The RMSE of traditional RNN models on the test set is given in Table 6. We can notice a three times decrease in performance as compared to predicting solely free-fall with a similar input.

Model	**RMSE**
Vanilla RNN	0.0537
GRU	0.0532
LSTM	0.0450

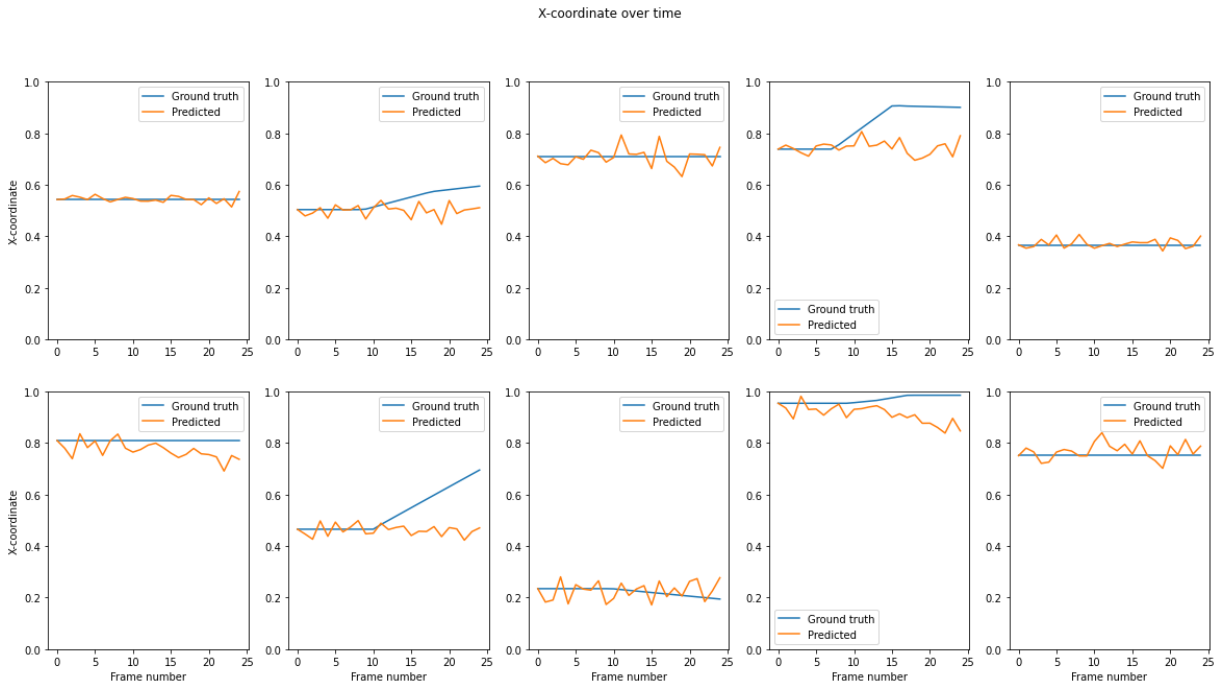
Table 6. RMSE of traditional models on predicting trajectories with collisions from the test set.

Figures 15-17 <PLEASE I NEED AN ADVICE ON HOW TO PRESENT THIS FIGURE BETTER> shows the predicted trajectories by each of the traditional models, together with predicted x-coordinate and y-coordinate time series. By looking separately at each coordinate prediction, we can notice that it is very noisy, which potentially might indicate lack of the

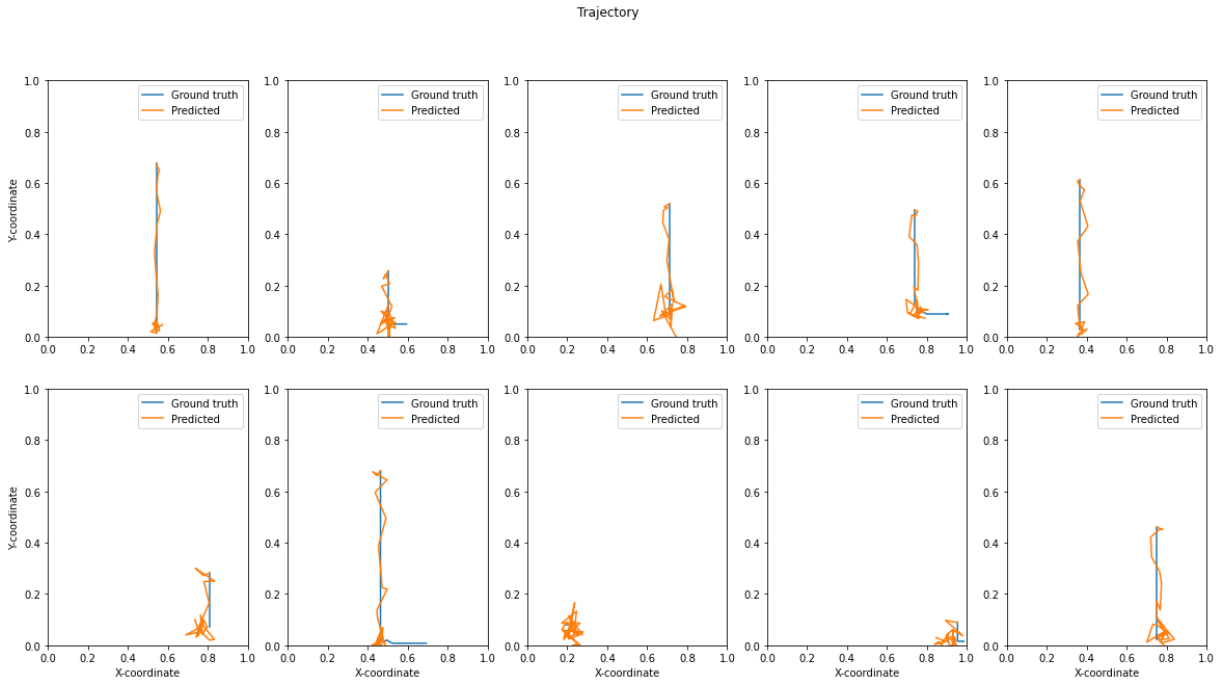
training data. Among the three models, LSTM does the least noisy predictions. These predictions are also not really accurate - while they are doing a good job predicting that the object falls down, they fail to correctly infer the collisions and movement after they happened.



(a)

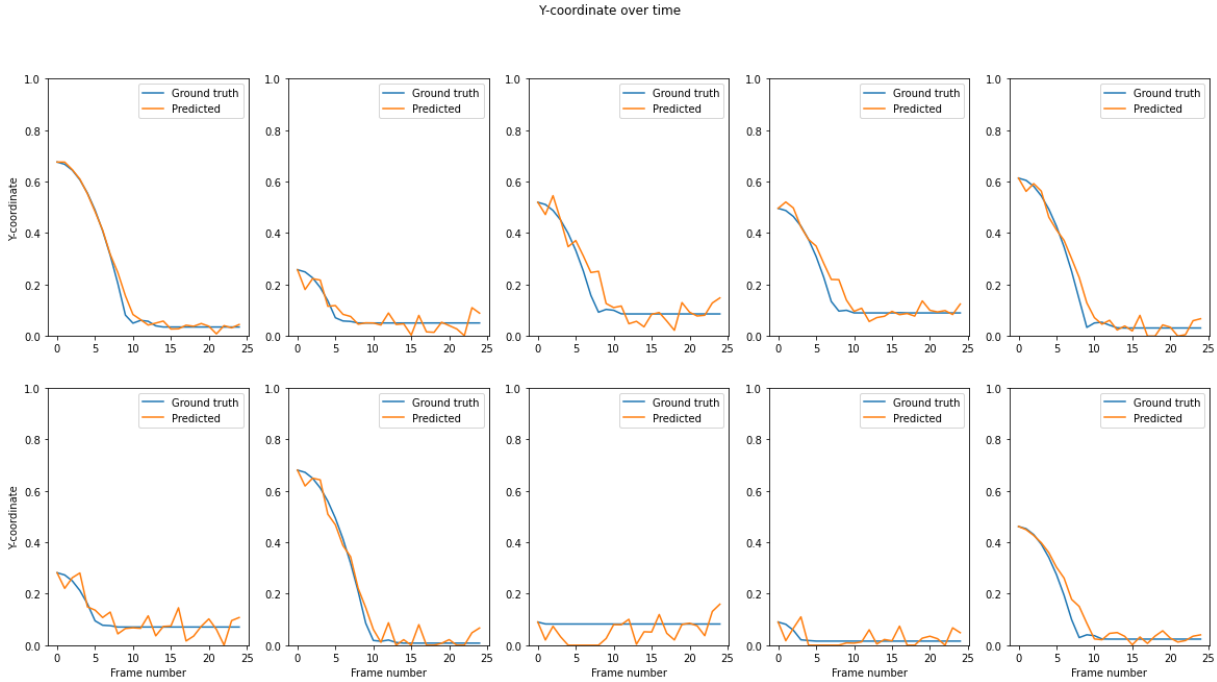


(b)

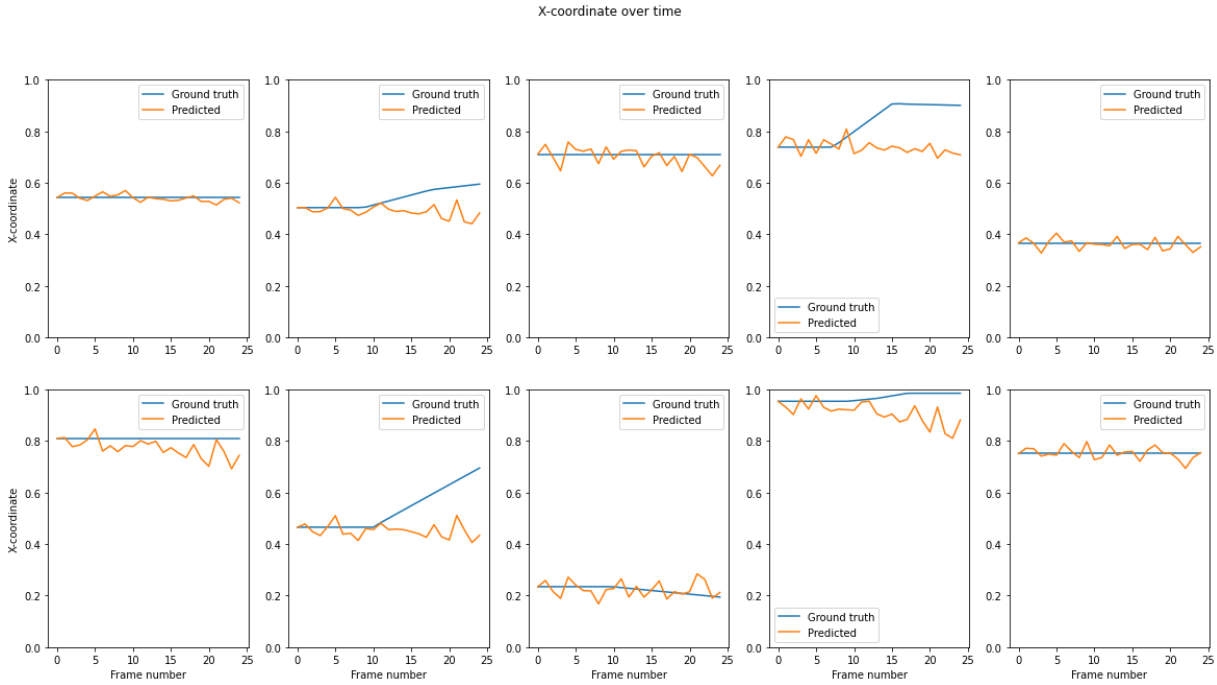


(c)

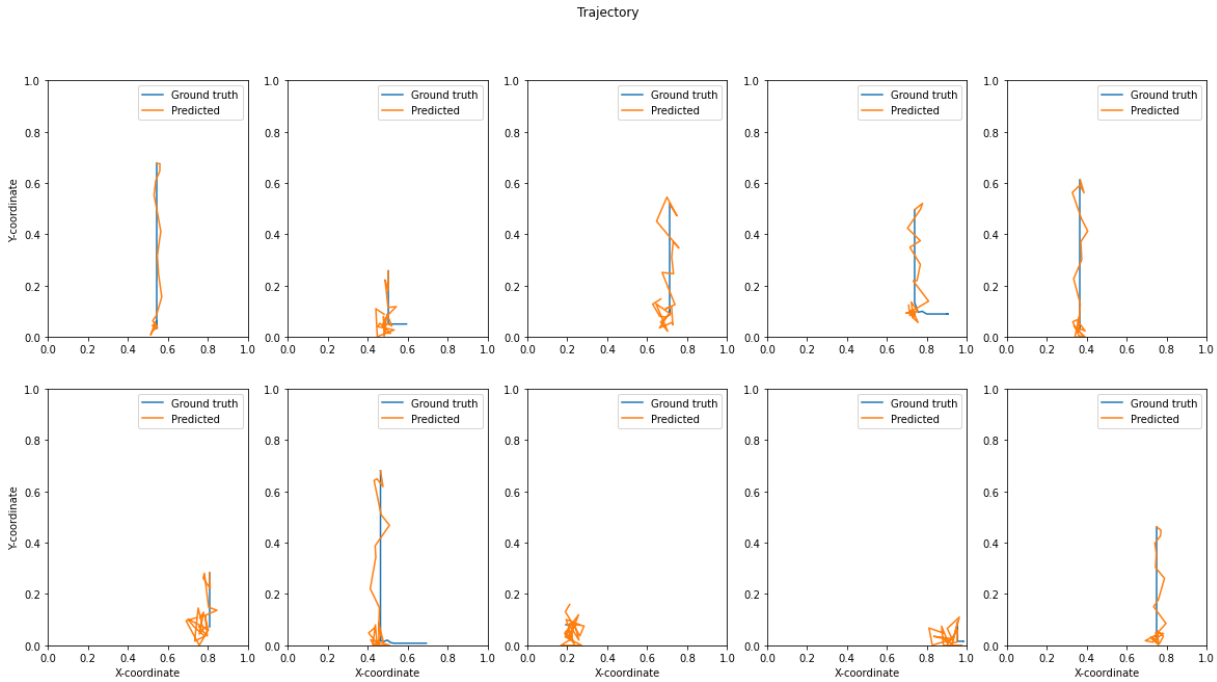
Figure 15. Frist 10 test-set predictions of Vanilla RNN on scenarios with collisions. (a) Y-coordinate timeseries; (b) X-coordinate timeseries; (c) Full trajectory.


Y-coordinate over time
Ground truth
Predicted
Y-coordinate
Frame number

(a)

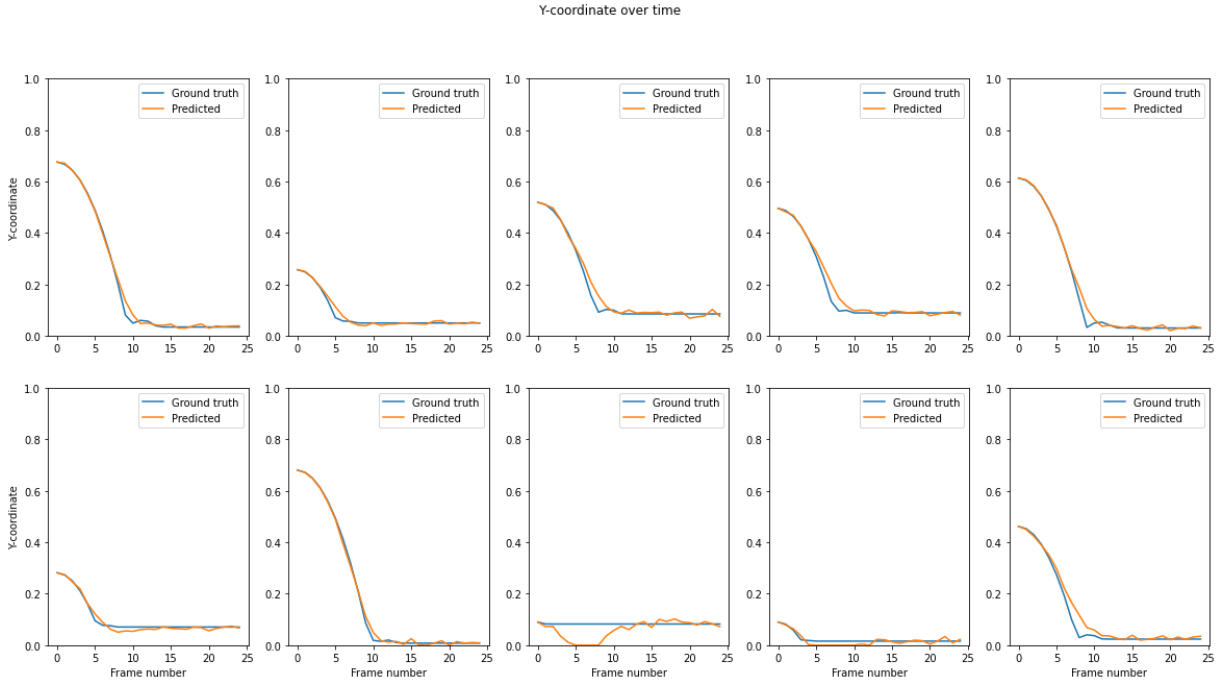

X-coordinate over time
Ground truth
Predicted
X-coordinate
Frame number

(b)

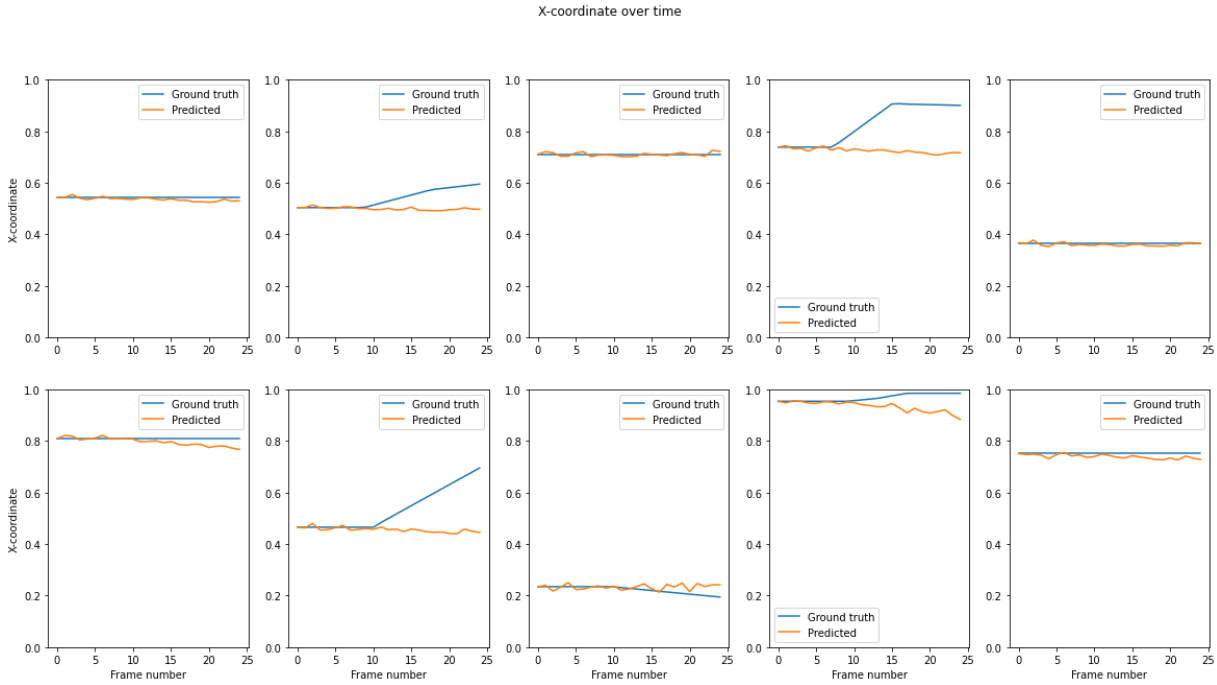


(c)

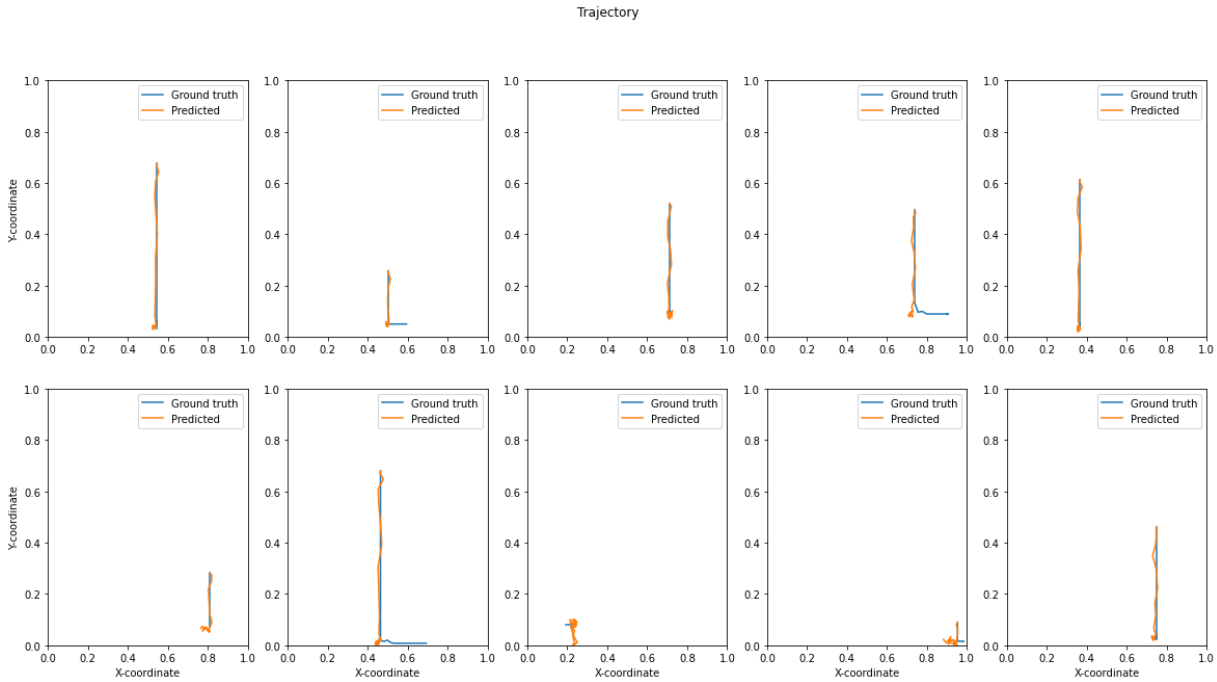
Figure 16. Frist 10 test-set predictions of GRU on scenarios with collisions. (a) Y-coordinate timeseries; (b) X-coordinate timeseries; (c) Full trajectory.


Y-coordinate over time
Ground truth
Predicted
Y-coordinate
Frame number

(a)


X-coordinate over time
Ground truth
Predicted
X-coordinate
Frame number

(b)



(c)

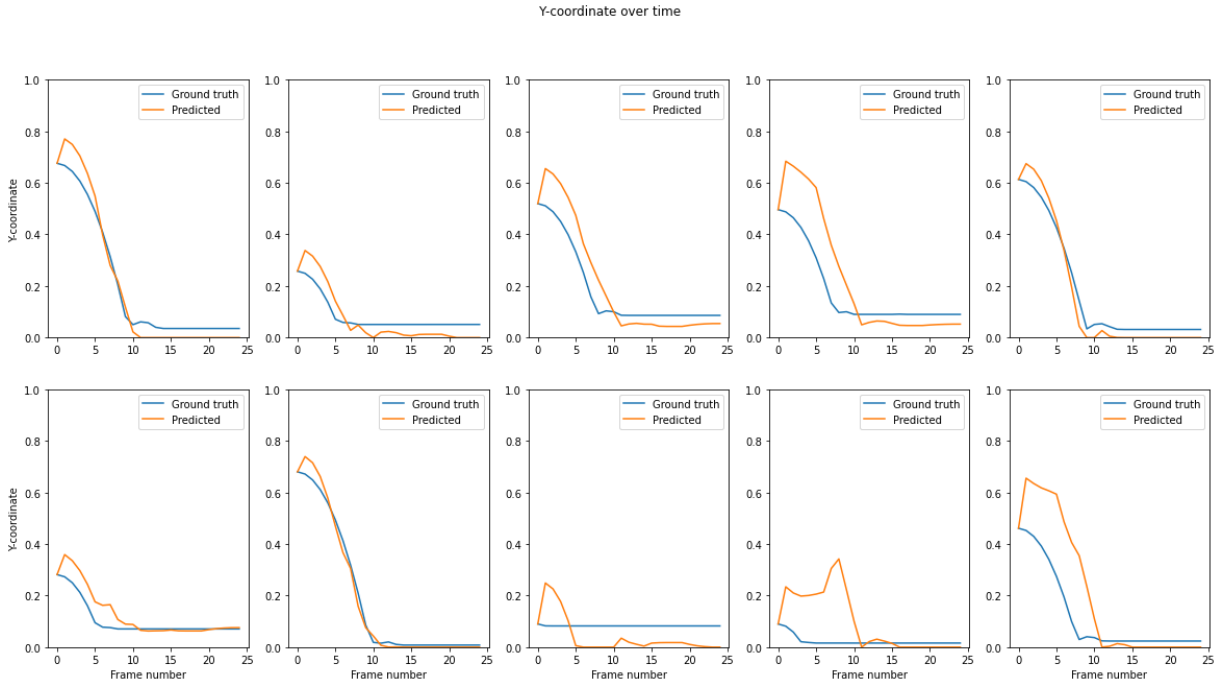
Figure 17. Frist 10 test-set predictions of LSTM on scenarios with collisions. (a) Y-coordinate timeseries; (b) X-coordinate timeseries; (c) Full trajectory.

As for the Reservoir Computing models, the situation is a little bit different. While the predictions done by these models are also far from being correct, the amount of noise that is present in these models is drastically less. A common pattern that I see from looking at ESN-predicted trajectories is that it predicts that the object first moves up before falling down, which I cannot explain. Moreover, in contrast with traditional RNNs that tend to predict that the ball will not move to the sides, ESN models try to guess which side it will move to, yet barely doing it correctly. Table 7 shows RMSE results of different ESN models on the test set. Figures 18-21 show the first 10 predicted trajectories, x-coordinate, and y-coordinate timeseries from the test set.

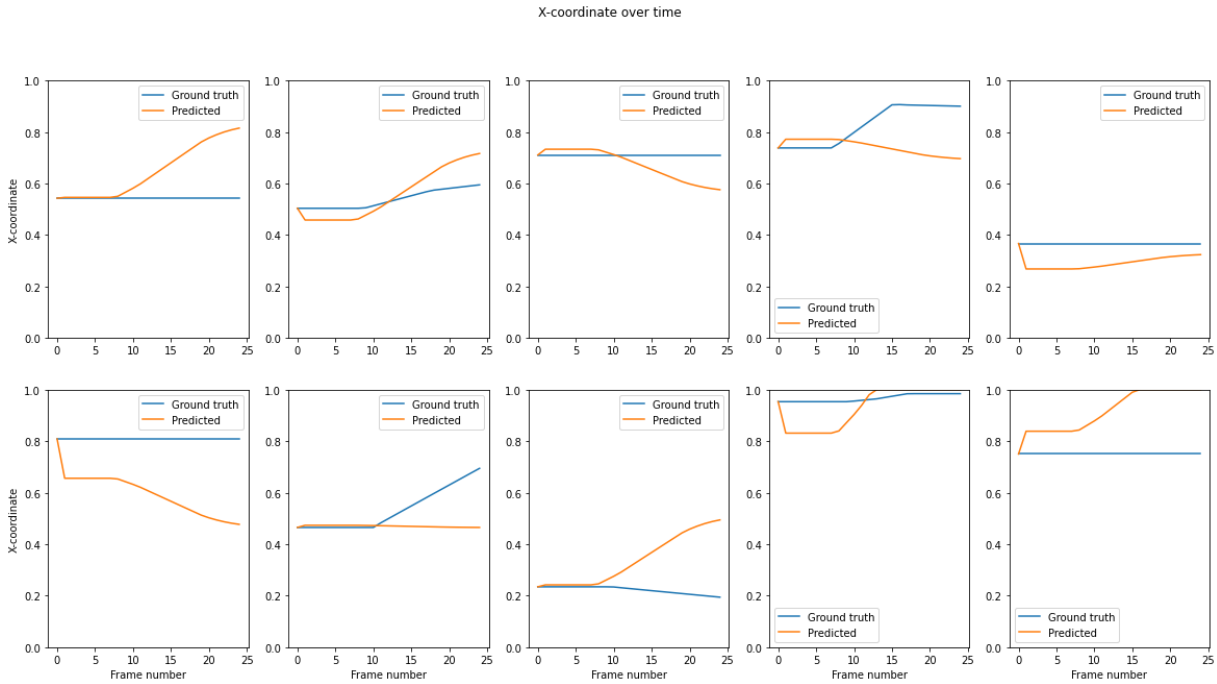
Model	**RMSE**
ESN	0.111

Sequential ESN	0.1520
Parallel ESN	0.1120
Grouped ESN	0.1316

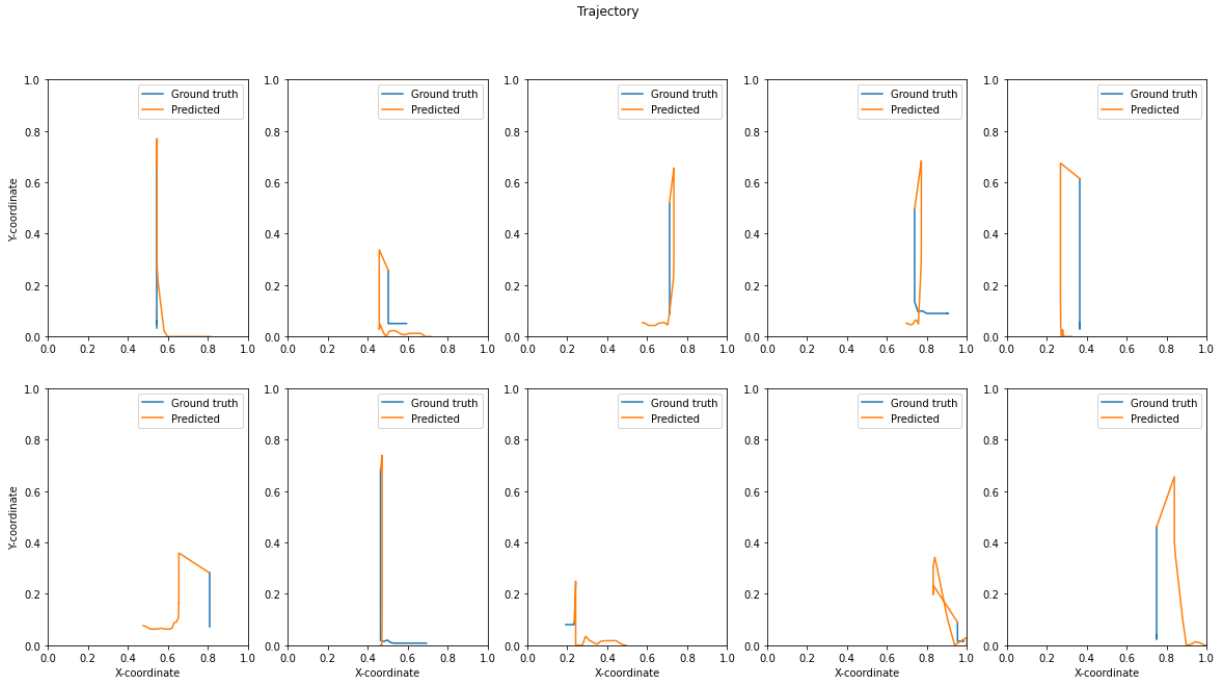
Table 6. RMSE of RC models on predicting trajectories with collisions from the test set.



(a)

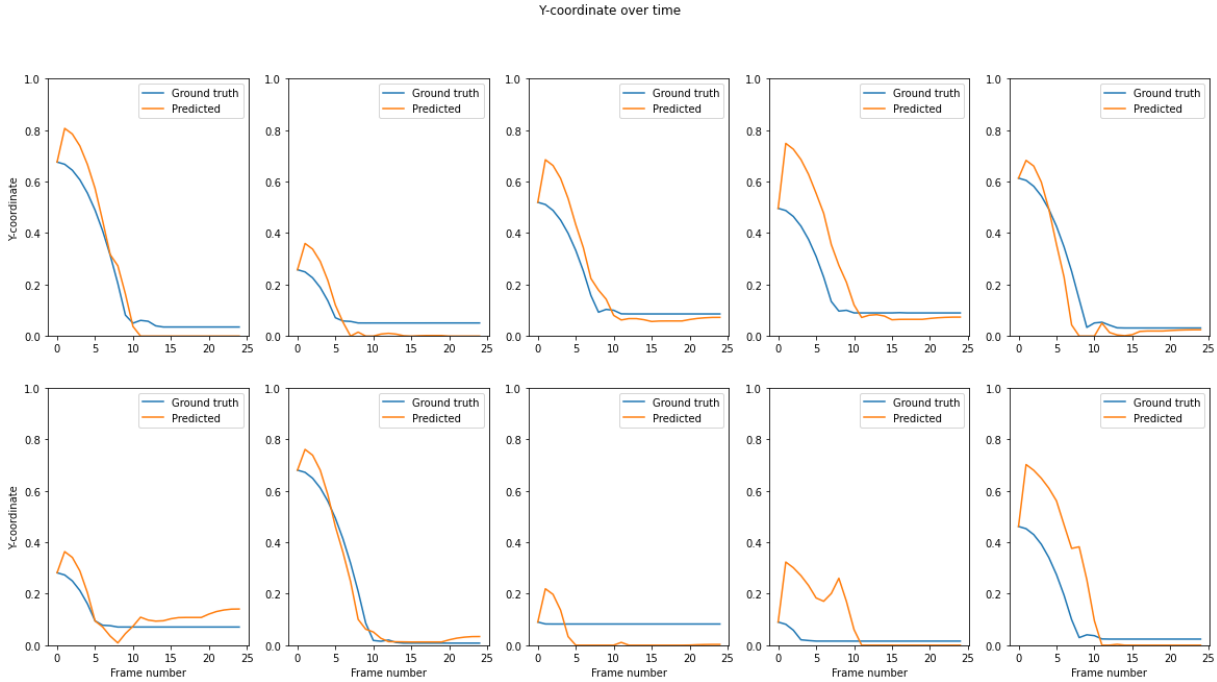


(b)

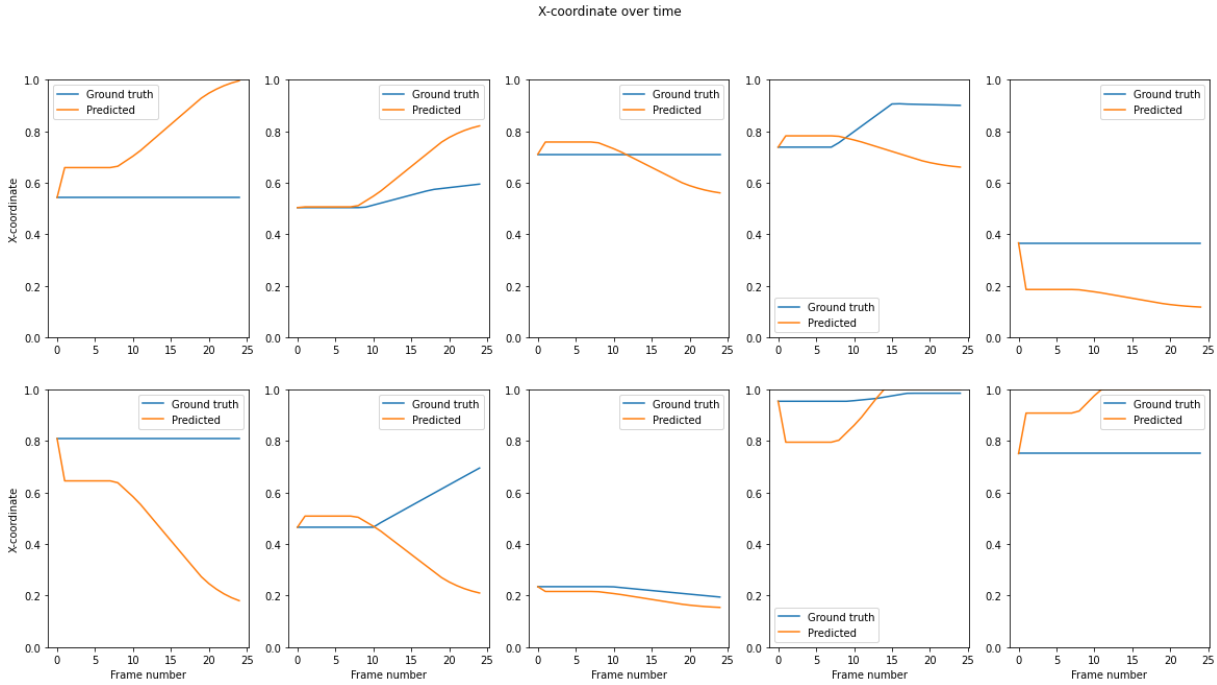


(c)

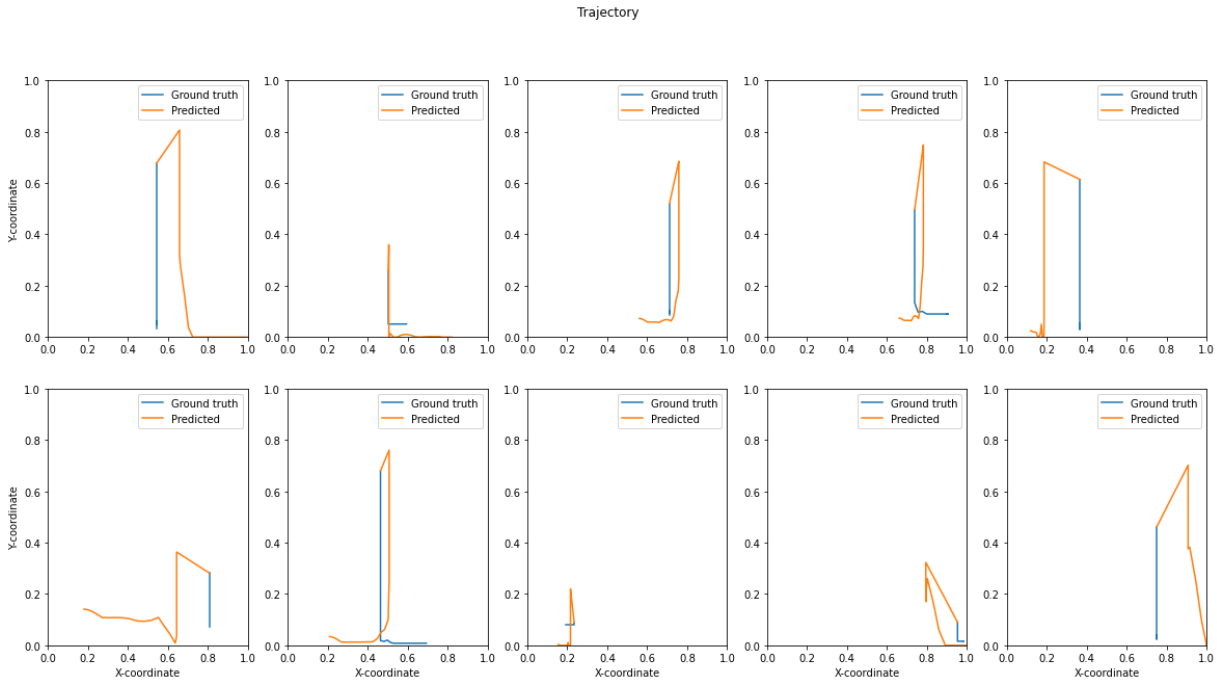
Figure 18. Frist 10 test-set predictions of ESN on scenarios with collisions. (a) Y-coordinate timeseries; (b) X-coordinate timeseries; (c) Full trajectory.


Y-coordinate over time
Ground truth
Predicted
Y-coordinate
Frame number

(a)

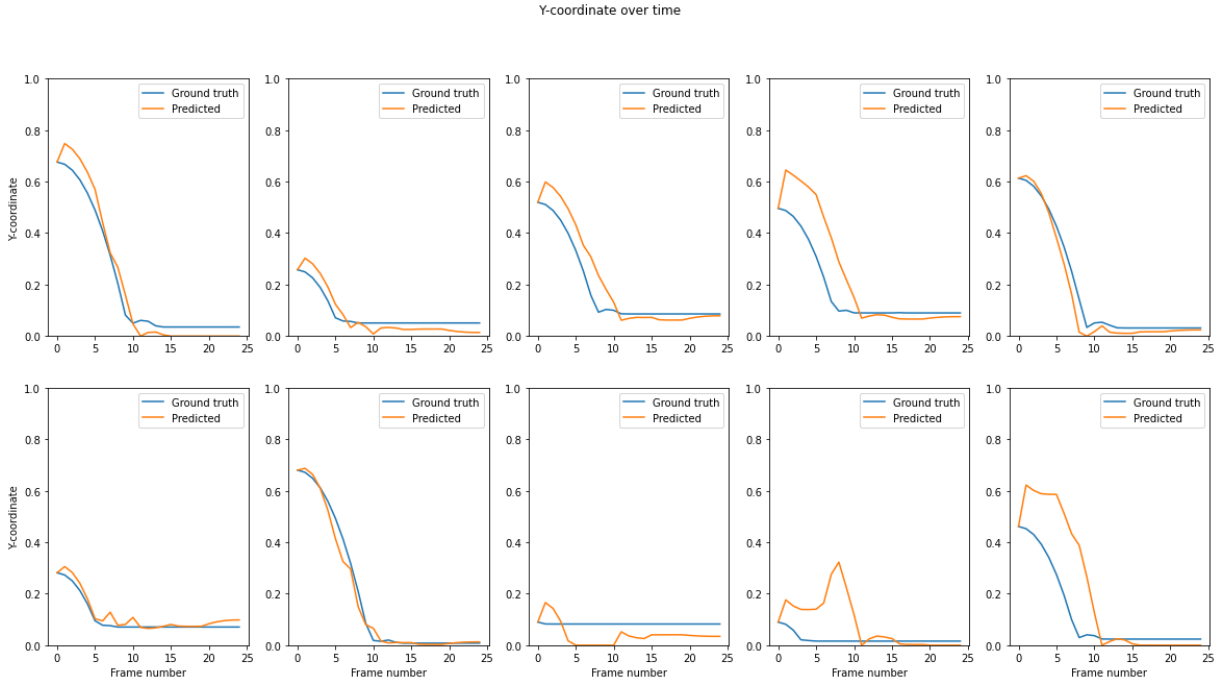

X-coordinate over time
Ground truth
Predicted
X-coordinate
Frame number

(b)

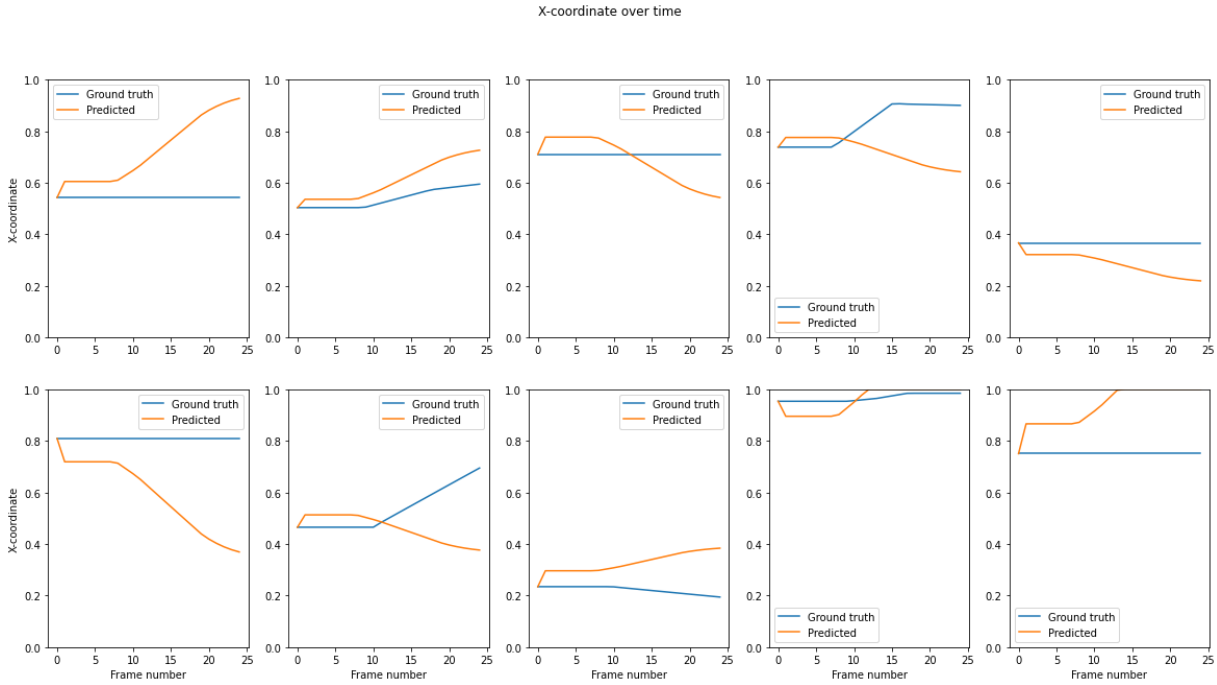


(c)

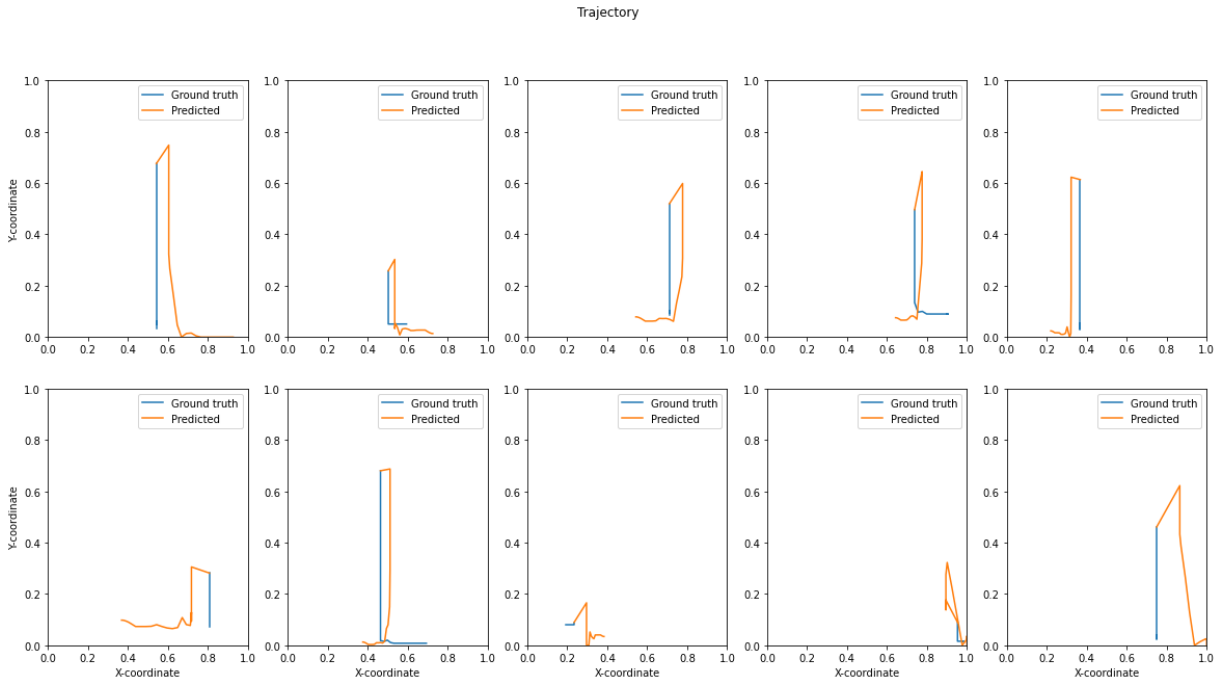
Figure 19. Frist 10 test-set predictions of Sequential ESN on scenarios with collisions. (a) Y-coordinate timeseries; (b) X-coordinate timeseries; (c) Full trajectory.


Y-coordinate over time
Ground truth
Predicted
Y-coordinate
Frame number

(a)

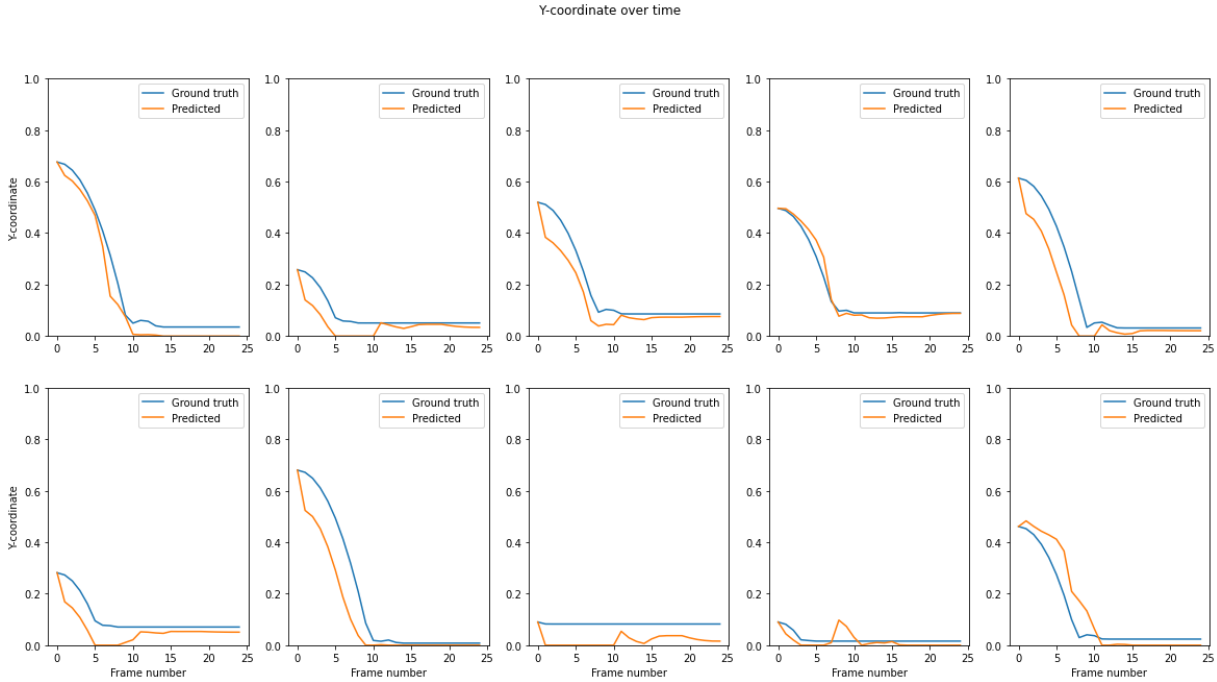

X-coordinate over time
Ground truth
Predicted
X-coordinate
Frame number

(b)

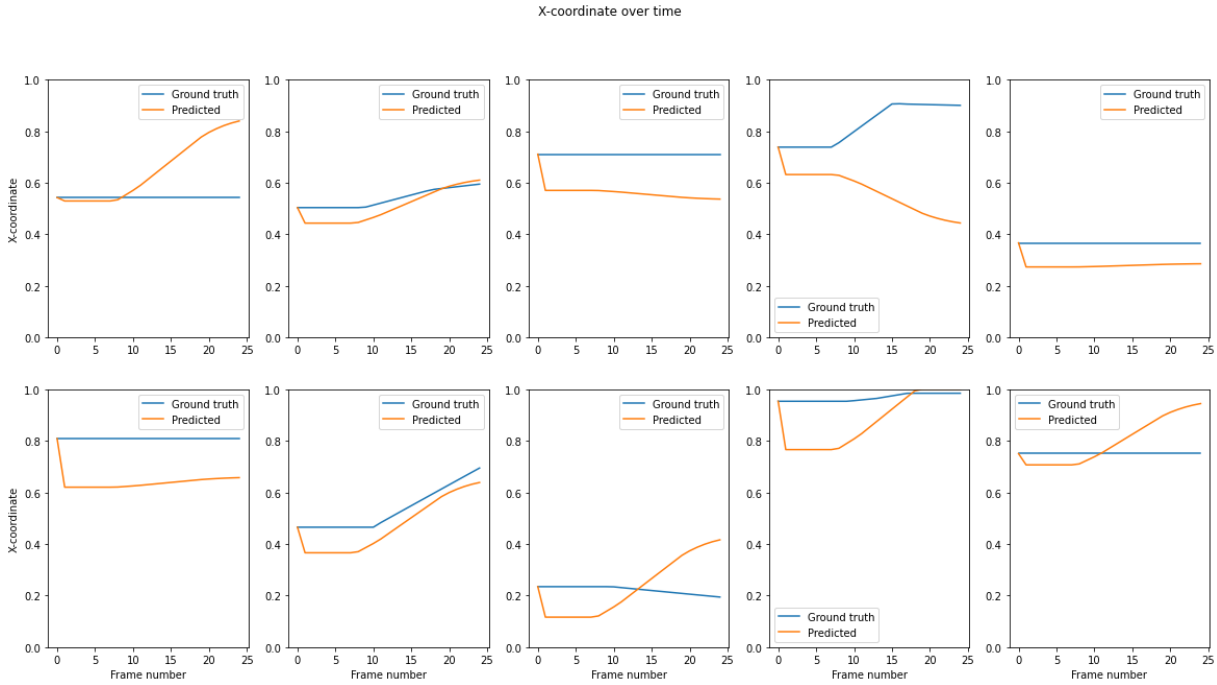


(c)

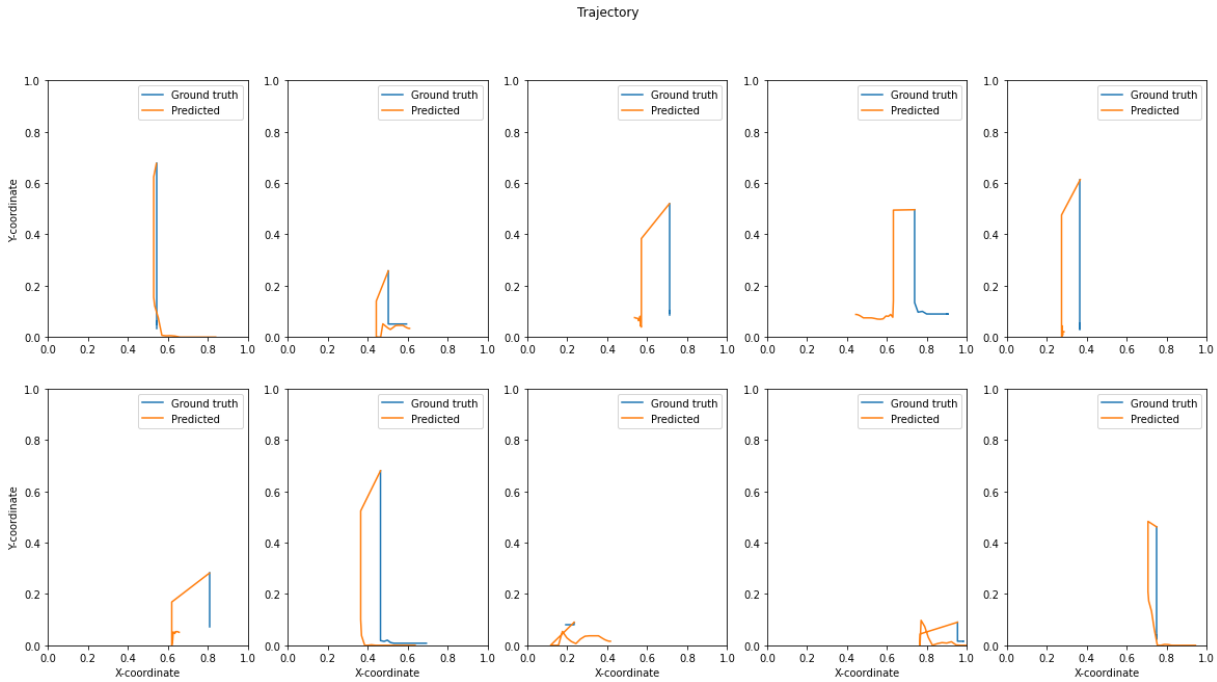
Figure 20. Frist 10 test-set predictions of Parallel ESN on scenarios with collisions. (a) Y-coordinate timeseries; (b) X-coordinate timeseries; (c) Full trajectory.


Y-coordinate over time
Ground truth
Predicted
Y-coordinate
Frame number

(a)


X-coordinate over time
Ground truth
Predicted
X-coordinate
Frame number

(b)



(c)

Figure 21. Frist 10 test-set predictions of Grouped ESN on scenarios with collisions. (a) Y-coordinate timeseries; (b) X-coordinate timeseries; (c) Full trajectory.

5.3. Predicting evolution of the entire scene

NOTE (this will be cut in the final submission):

The situation in this section is similar to the one above. From now this section is the copy of a corresponding section from the full draft, but hopefully I can show something else during the defense.

For a curiosity purpose, I tried to check how well ESN will deal with predicting not just a single ball trajectory but the entire scene evolution.

As expected, the model that is predicting trajectory of three balls instead of 1 has around three times larger error. The RMSE on the test set appeared to be 0.1598 for the scene evolution prediction. Moreover, as can be seen in Figure 22, the model predicts the trajectories of the balls in such a way that they overlap during their movement. Given a poor performance, I will not use this model further on.

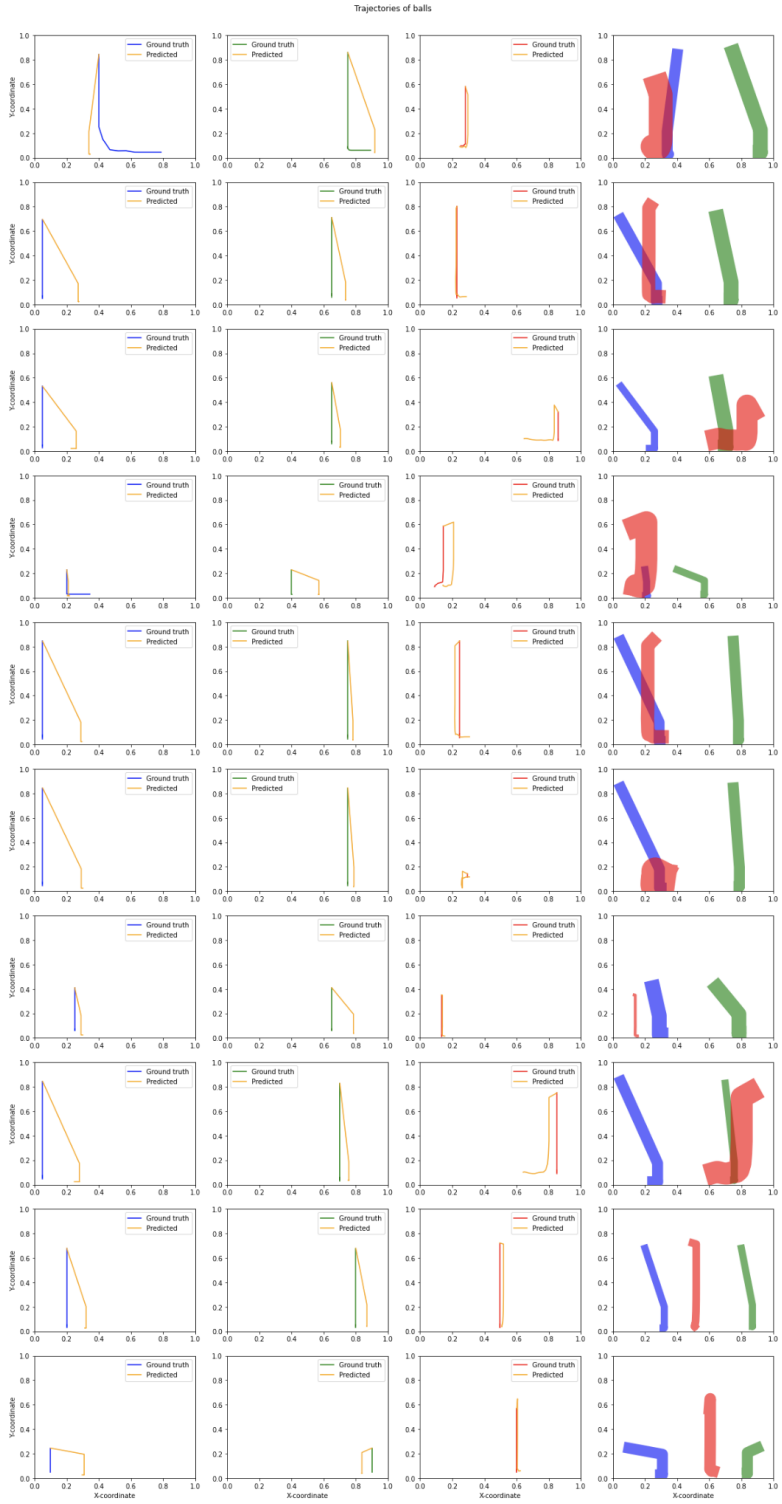


Figure 22. Predicted and ground truth trajectories of each of the three balls on the first 10 simulations from the test set (columns 1-3) and predicted trajectories of all three balls together (column 4).

Section 6. Conclusion and Future Work

I am postponing writing this section until I have all my results ready, because otherwise I might need to do a lot of rewriting. This section will include:

- Summary of my process
    - Prediction of one step at a time
    - Determining optimal input using FNNs
    - Training RNNs and ESNs to predict free-fall
    - Generalizing to 2D and multiple balls
    - Predicting the trajectory of a single ball in presence of collisions
    - Predicting the trajectory of all three balls in presence of collisions
- Discussion of the results, which models I think are better and why so.
- What I learned from doing this project:
    - Using PyTorch for data manipulations and training deep learning models.
    - Better understanding of how to choose the model and how to tune hyperparameters for it
    - Better understanding of what are ESNs, how they work, and how to implement them using Reservoirpy
    - How to use heuristics to understand the data
    - How to do research using the bottom-up approach
- Future work:
    - Adding more objects on the scene, both dynamic and static
    - Adding objects of the shapes that are different from the balls

- Simulate 3D data instead of 2D
- Combine with contextual inference models to use real-world data

HC/LO Notes

Tags	
CAP Issues	
Carterochka/capstone PRs	
Last update date	@December 9, 2022

HCs

▼ #rightproblem

While defining the problem that I want to solve with my capstone, I was following a problem analysis process. Stemming from my desire to do a research that combines Computer Science and Physics, I started with researching different aspects of how these fields overlap. By doing initial literature review, I identified a field of Intuitive Physics as a currently developed research direction. I couldn't find any research paper that implements a bottom-up approach - starting from simplistic scenarios, and slowly complicating tasks to reach the real world complexity. Rather than that, all the research papers that I encountered were trying to take the real data and fit the models that learn specific aspects applicable to such data. This guided an approach that I wanted to implement in my capstone - start from an oversimplified intuitive physics task, and slowly increase the complexity. The rest of the work lied in scoping - from finding efficient intervention to the system, to predicting trajectories, to predicting trajectories using Echo State Networks, to predicting trajectories using ESN in scenarios with no more than 3 balls.

The strength of the HC application lies in covering all the key aspects of #rightproblem: understanding the current state (i.e. current state of research), analyzing the obstacles (i.e. my lack of knowledge and how I can effectively gain it in the process), and navigating the scope. As a result, I had a clearly scoped capstone that can be explored in depth over the course of two semesters.

▼ #breakitdown

While I imagined a roadmap of my work with the incremental complexity appoach, I still needed to break down each step of that roadmap into specific tasks, successful completion of which ensures a steady progress of my research. I have broken down the entire process into the following tasks:

- Planning the infrastructure (which included setting up Jira, Github, Cloud folders, and development environment)
- Making a literature review
- Planning the code architecture
- Implementing the algorithm within the currently researched step (which included a dataset generation, creating and debugging machine learning models, and tuning the hyperparameters)
- Grooming new tasks and further research directions
- Creating and iterating on the capstone writeup.

Along the way I had a multilevel breakdown of my work, components of which added up together in the results that are provided in this deliverable. Clear multilevel breakdown of tasks that added up to achieve the big goal is what constitutes a strong application of this HC in my capstone process.

▼ #modeling

The application of this HC in my capstone can be broke down into two isolated cases. The first case lies in my interaction with the Phyre simulator. Here, I applied #modeling by tuning the simulator to produce the dataset for each of the scenarios, as well as implementing the threshold of free-fall scenarios when working with collisions. What I mean by tuning is limiting the set of simulated scenes to ones that don’t have any extra objects but three balls. Moreover, for the free-fall I had to implement a free-fall check. The second case lies in actually modeling the trajectories using neural networks. A strong application here lies in identifying an appropriate type of neural network for such modeling and optimizing hyperparameters and data to achieve better results.

▼ #algorithms

This HC was already applied in a lot of moments, yet this list is not finalized. The applications already done include implementing the dataset generation algorithm

that invokes Phyre simulator, training and tuning the neural networks, including leveraging the overfitting for the research purposes, and implementing hyperparameter optimization. The applications to be done include implementing the visual data encoding and putting my research together into the software package.

▼ #dataviz

There are two main aspects of this HC application: creating a well-designed visualizations that guide and support the interpretation of neural networks' results, and schematically demonstrating the architecture of those neural network in a way that is clear to comprehend. Both these application can be found in the research part of my capstone, yet I am still planning to experiment with the layouts of my data visualizations to make it easier to see the results.

▼ #optimization

I am not sure if I will keep this HC in the final version of my capstone, but so far there is no other meaningful HC that I can see as a substitution. A prominent yet extremely important application of this HC in my capstone that was done by me is hyperparameter tuning for ESNs using Grid-Search. Another application lies in optimizing the hyperparameter tuning and training process for the scenarios with collisions, where I generated two separate datasets to optimize the computation time.

▼ #designthinking

I am constantly reiterating the work that I am doing, which can be seen in two aspects. Aspect one is my research - I am constantly reviewing my work, being proactive to reimplement or make adjustments to my work if I find how to make it better. To ensure I am doing this effectively, I am using a pull review system on Github - all my work does not go to master or feature branch right away, but rather stays in a pull review for at least a day so that I can review it with a fresh mind. The second aspect is reiterating my writeup. If you compare this full draft to midterm deliverable, you will notice that the organization of my essay changed from just a chronological recap of my work to a coherent story telling. Here, as all my deliverables take place in github as well, comparing different versions of my work can be used to analyze the effectiveness of my design thinking approach. Keeping track of the drafring progress using external instruments is what constitutes a strong application of this HC.

▼ #strategize

My research follows the bottom-up approach, which requires an application of this HC to strategize the incremental complexity. In the strategy process, I need to make sure that each following step is (1) in the right direction, (2) provides a right enough increment of complexity to the task, and (3) ensures that the following task is not ambiguous but is defined clearly. The research part of my capstone shows a clear structure of such increments.

▼ #organization

The full draft organization was changed compared to the midterm deliverable to resemble the flow of narration that I want to see in my final capstone. Instead of just a chronological recap of my progress, the current organization reflects my research progress, explaining the reasoning behind researching specific scenarios and models and guiding towards the next steps. While the HC description defines a strong application of #organization through a sophisticated organizational structure, in my case I tried to keep it simple instead, so that the readers are not lost in the flow of the narration.

▼ #critique

One of the applications came when I was doing preliminary research to identify the topic and the approach that I want to take. By applying #critique, I was able to see that none of that papers I encountered use bottom-up approach, so I decided to try. The second application was in analyzing the results that I obtained during my research process. My failures were not reflected in the paper because they don’t show any meaningful results, but critical analysis of those papers is what allowed me to move on with my research.

▼ #gapanalysis

This application is yet to come when I conduct a proper research to conclude whether any previous work is using a bottom-up approach for trajectory prediction.

▼ #variables

Application of this HC comes during the choice of variables to be fed into the model for each scenario. As such, as was shown during my linear network experiments, insufficient data might lead to inability of the model to do anywhat accurate predictions. Thus, I needed to carefully choose a set of variables that will represent

all the information about the scene. A second part of this HC application will come from visual data encoding, when I will need to come up with a process to encode a visual information into a set of pre-defined variables.

▼ #plausibility

Plausibility checks were essential throughout my research process to understand whether my model can indeed produce the results that I want. For example, whenever my model was predicting negative values, even though I put a constraint on it, that means that the results are implausible for the model I intended to have and I have to change something. Same trick applies for overfitting - if the model started detecting patterns that should not appear, then I might reconsider the way I train my models.

Feels to me like this doesn't pass anywhere near being a strong application yet. Hopefully, there will be space to strengthen it in the remaining work, or I will switch it to another HC for the final version of the capstone paper.

▼ #heuristics

My proudest invention in this capstone (at least quick google search didn't find anything about it) is using overfitting to understand what data should be added to the model input. This feels similar to "Juxtaposing opposite problems o suggest reciprocal solutions" creative heuristic among the list that we studied in the first year (link for reference), but in the context of machine learning.

Hopefully the remaining work will give enough space to apply this HC more in depth to make a strong case for it in the final paper.

▼ #audience

This HC will be applied in the future, when I will finalize my deliverables. As my capstone output, I plan to have three deliverables: (1) capstone paper, which is a technical paper with a lot of details, (2) medium blogpost (or a series of posts) for more general audience (including those who just want to start engaging into intuitive physics), and (3) software package/library for trajectory prediction using visual inputs. Each of these deliverables are dedicated to different audiences - paper for researchers, medium - for general audience (I could say "my grandma", but poor her doesn't understand English), and library - for practical machine learning users.

LOs

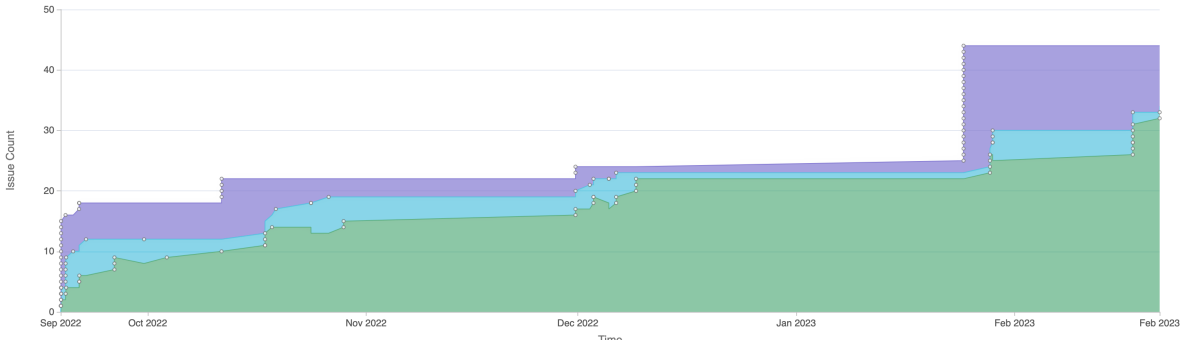
▼ #cs156-neuralnetworks

Neural networks approach is a central approach of my capstone. More specifically, I tested out a few types of neural networks, namely Feedforward Neural Networks, Recurrent Neural Networks (including GRU and LSTM), and Echo State Machines (including Deep architectures). For each neural network type and the choice of architecture design, I explained the rationale behind such a choice. For each of these models, I performed a hyperparameter tuning, as well as compared their performances. This process fits the rubric for 4 on this LO, which makes me sure in the strength of this LO application.

▼ #cs162-agile

I used (and keep using) an agile approach in my capstone process for several reasons. First, and most important, is that my research process might change directions according to the findings that I obtain. As such, waterfall planning would fail in the very beginning of my journey. Second, by breaking up my work in 2-week sprints and reviewing the progress at the end of each such period, I could analyze how well my work is progressing and what do I need to change in my approaches. Finally, by having grooming sessions each sprint, I could adjust and clarify the roadmap of the future work that I need to follow.

I used Jira as a tool to keep up with the agile process. First, it allowed me to make use of #strategize by allowing to categorize my tasks into epics, completing each of which symbolized a milestone. Second, it provides a range of tools to access my productivity. A cumulative flow diagram below visualizes the progress that I was making while working on my capstone throughout the semester.



▼ #cs110-ComputationalSolutions

By formulating an algorithmic solution to the intuitive physics problem, I systematically broke the problem down into a clear, ordered set of concrete steps, and then translate these steps into programming language statements that a computer can execute. I applied techniques such as machine learning modeling and algorithmic design to optimize my solution and make it more efficient. In order to do this, I compared and contrast different algorithmic choices in order to determine which one is the most effective for solving the problem. I also used data structures such as arrays, hash tables, and custom dataset structures to store and manipulate data, as well as computational techniques such as Grid-Search to optimize my solution. Finally, I analyzed and critiqued my solution to ensure its accuracy and efficiency.

▼ #cs156-modelmetrics

I used a range of common model performance metrics to assess the performance of my machine learning models and algorithms. I used the Root Mean Squared Error (RMSE) metric to evaluate how close the predicted values were to the actual values. This metric is particularly effective for this task, as it measures the average magnitude of the error and is suitable for predicting continuous values. I also used predicted data visualizations to assess how well the model was learning the patterns in the data. This allowed me to assess the accuracy of the model in a more qualitative way, by analyzing the relationship between the predicted and actual values and identifying any discrepancies. This gave me a more holistic understanding of the model's performance, by giving me insight into the accuracy of both the predictions and the underlying patterns.

▼ #cs110-PythonProgramming

I first wrote Python programs to implement the desired algorithms. I ensured that my code was well-structured, commented, and documented, so that it would be easy for others to understand and modify. Next, I used Python code to analyze the performance of my models. I used Python libraries, such as numpy and matplotlib, to plot my results and visualize the performance metrics. Finally, I used Python to compare the performance of different algorithms and data structures. I used the

libraries that help keeping track of performance metrics, such as the time taken to train the model or the accuracy of the model.