Learning Differential Equations for Stock Market Predictions

Aditya Singh	Nishtha	Vaidic Jain	Vartika Gupta
170056	180489	180845	180849
adisingh@	nishthaa@	vaidic@	vartikag@

CS771A - Course Project
Faculty: Dr. Piyush Rai
Indian Institute of Technology Kanpur

Abstract

This paper reports our attempt to model time series data to make stock price predictions. It features three different machine learning models namely Long Short Term Memory (LSTM), Neural Ordinary Differential Equations (Neural ODE), and Neural Stochastic Differential Equations (Neural SDE) for capturing the trends. Detailed procedures for building each model is reported along with comparison of their quality of forecast. Data for training and testing is extracted from the NSEpy library [1].

Disclaimer: The work carried out in the project has not been re-used from any of our existing/past projects at IIT Kanpur or elsewhere. All the referred papers have been listed in the References section.

1 Problem Description and Motivation

Accurately predicting stocks has been a very sought out problem due to its financial significance. However, stocks prices are difficult to forecast because of their chaotic behaviour to perturbations such as day to day political and economic event, company's performance, global news, to name a few. Various statistical models have been formulated to predict these low signal to noise ratio time series data. Machine learning brought a breath of fresh air with its powerful regression techniques. Of these, neural networks algorithms like DNN, CNN, RNN were the ones to gain the highest success. Complex models based of these neural networks are actively in use in top financial firms.

Li et al. (2020) in their paper [2] suggested an algorithm to train Neural Stochastic Differential Equations (Neural SDE) making it an immediate candidate to try and train stock market prices since stochastic equations were the key statistical tool currently being used for hedging stocks and securities. The algorithm in [2] was inspired by a paper in 2018 on Neural Ordinary Differential Equations (Neural ODE) [3].

Based on these advancements, we were motivated to try and model closing price of stocks using Neural ODE and Neural SDE, and see how well they compare with an already matured stock price prediction algorithm (RNN based LSTM).

2 Literature Review

2.1 LSTM - Long Short-Term Memory

Recurrent Neural Network: The traditional *Recurrent Neural Networks* provides the flexibility of carrying the information from previous cells to be used in future cells and help take in account the significance of sequence of inputs in a much more efficient manner. For each time step t, the activation $a^{(t)}$ and output $y^{(t)}$ are calculated as:

$$a^{(t)} = h_1(W_{aa}a^{(t-1)} + W_{ax}x^{(t-1)} + b_a) \qquad y^{(t)} = h_2(W_{ya}a^{(t)} + b_y)$$

Here, $W_{aa}, W_{ya}, W_{ax}, b_a, b_y$ are the parameters and h_1, h_2 are the activation functions. The most common networks used to train recurrent neural networks include Back-propagation Through Time (BPTT) and

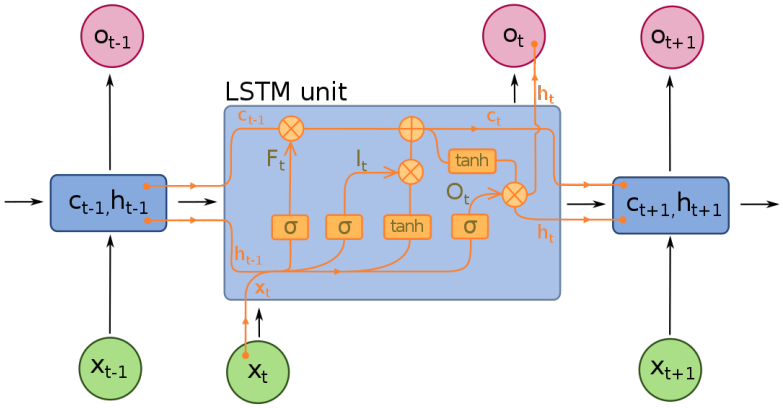
Real-Time Recurrent Learning (RTRL) out of which BPTT is most popular [4].

Vanishing Gradient Problem: The issue with these however is the so called Short-Term memory problem as they are not efficient in carrying information for long sequences. They suffer from the *Vanishing Gradient Problem* i.e. the gradient becomes negligible as it back propagates through time and therefore stops learning and updating the parameters of the layers (typically earlier layers). Consequently, the network appears to forget or have short-term memory. The solution to this problem came with the advent of LSTM and GRU cell based RNN's.

LSTM cell: The *Long Short-term Memory* model uses various gates and the cell state to manage the flow of information across each cell. It passes down the important information to the future cells and discards (or forgets) the irrelevant information. It carries this process via the cell state which acts as the memory of the network. The process of deciding which information to keep/discard is undertaken using various gates.

Figure 1: A single LSTM unit cell. Image Courtesy - *François Deloche, Wikimedia Commons.*
The mathematical equations behind these gates are [5]:

$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$
$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$
$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$
$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$
$c_t = f_t \circ c_t + i_t \circ \tilde{c}_t$
$h_t = o_t \circ \sigma_h(c_t)$
$\{\circ :$ Element-wise Product$\}$



Here, $x_t \in \mathbb{R}^d$ is the input vector to the LSTM unit; $f_t \in \mathbb{R}^h, i_t \in \mathbb{R}^h,$ and $o_t \in \mathbb{R}^h$ are the forget gate's activation vector, input gate's activation vector, and output gate's activation vector respectively. $h_t \in \mathbb{R}^h$ is the hidden state vector which is also the output of the LSTM cell; $\tilde{c}_t \in \mathbb{R}^h$ and $c_t \in \mathbb{R}^h$ are the cell input activation vector and the cell state vector respectively. Here, d is the number of input features and h is the number of hidden units

Gates in LSTM cell: First the forget gate decides which information should be kept or thrown away based on previous hidden state and current input. Next, the previous hidden state and the input vector passes through the input gate to generate the input gate activation vector and the $\tilde{c}_t$ vector. Then these activation vectors are used to get the cell state vector c_t. In the end, the hidden state vector is calculated with the combination of cell state vector and output activation vector. The hidden state vector and the cell state are then passed forward to the subsequent cells.

Why LSTM for Stock Prediction: LSTM has been popularly used for time series modelling. It serves the advantage of taking into consideration the sequence of inputs as a regular RNN. On top of that, it solves the Short Term Memory Problem and can pass down information from earlier time steps quite efficiently.

2.2 Neural ODE - Ordinary Differential Equations

Neural ODE architecture was proposed by Chen et al. [3]. Neural ODEs are neural network models which generalize standard layer to layer propagation to continuous depth models. We can construct and efficiently train models via ODEs as they are used to describe the time derivatives of dynamics. Neural ODEs present a new architecture with much potential for reducing parameter and memory costs, improving the processing of irregular time series data. A general ODE is as follows:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta)$$

where $t \in \{0, ..., T\}, \theta$ signifies other external parameters and $h_t \in \mathbb{R}^d$ is the hidden units dynamics of the model.
Auto Depth Selection: Unlike RNN which solves the time series model at depths, Neural ODE model

adaptively compute the output by sampling more at locations of high derivative values, thus giving a smoother curve. Fig. 2 from the original paper of Neural ODE [3] provides an illustration of this feature.

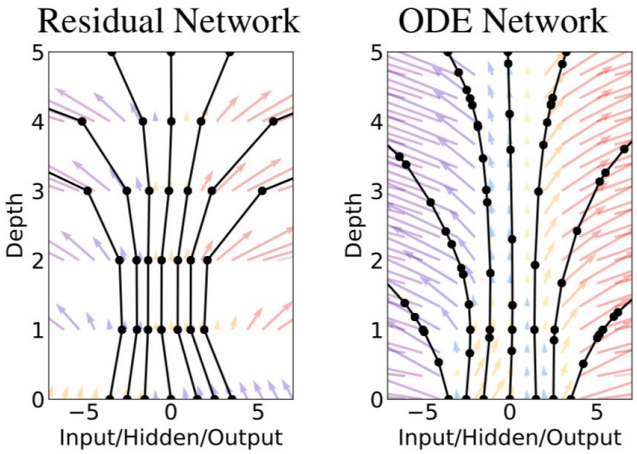


Figure 2: The adaptive computation of output in the ODE Network. Image Courtesy - *Chen et. al [3]*

Back-propagation using Adjoint Method: A big problem in using ODEs for machine learning was that the amount of steps used in evaluation an ODE was huge, and storing all the information for back-propagation was very memory expensive. A solution to this problem was presented in [3], where back-propagation was replaced with solving a different adjoint ODE. It reduced the computation of the loss gradient while scaling linearly with problem size, offering low memory cost, and explicit control over numerical error. The loss function an ODE tries to optimize is [3] -

$$\mathcal{L}\left(z\left(t_1\right)\right) = \mathcal{L}\left(z\left(t_0\right) + \int_{t_0}^{t_1} f\left(z\left(t\right), t, \theta\right) dt\right) = \mathcal{L}(\text{ODESolver}(z\left(t_0\right), f, t_0, t_1, \theta))$$

where z is the state variable (say, a stock's closing price) and θ signifies the parameter fed to the model (say, its last K days closing prices).

Why ODE for Stock Prediction: Even though using ODEs are not a good way to model stock prices, we tried using them since our end goal was to use Neural SDE for prediction. Neural ODEs are based upon the same principles on which Neural SDEs work, therefore, trying Neural ODEs before SDEs was the nature course of action. Also, the adaptability of Neural ODE's depth over RNN hinted towards a better performance of ODE's over LSTM.

2.3 Neural SDE - Stochastic Differntial Equations

Neural ODE makes an assumption that all the processes have deterministic dynamics, and model them with differential equations. This assumption, though not true in general, is specially false for stock prices which are notorious for their noisy trends. Stochastic differential equation are the variant of the ordinary differential equations, which incorporate random white noise in the form of derivative of Brownian motion or Wiener process. An SDE looks like-

$$dX_t = \mu(X_t, t)dt + \sigma(X_t, t)dW$$

where $X_t \in \mathbb{R}^d$ is a stochastic process, $\mu : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ is the *drift* function, $\sigma : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ is the *diffusion* function, and $dW \in \mathbb{R}^m$ is an m-dimensional Wiener process. For stocks prices, m = 1, which is equivalent to a random walk problem.

After the paper on Neural ODE was published, various researchers reported its stochastic variants with the ability to model noisy data. Issues with these models were-

- **Back-propagation Problem:** The backward step for updating parameters had high memory cost because it needed to store a lot of intermediate quantities to back-propagate due to the small time

step. Li et at. in their paper [2] proposed a method motivated by their previous paper on Neural ODE of solving the SDE backwards in time using Stratonovich integral and Itô calculus. This step essentially reduced the most memory expensive step to constant space cost.

- **Noise sampling in the forward pass:** The Noise sampled during the forward pass needed to be stored for using it in SDE solver for back-propagation. Li et al.[2] solved this issue by generating the noise for back-propagation from the random seed used in the forward pass, via a recursive Brownian tree.

Following the settlement of these two main issues, we were free to explore their usefulness for our objective.

Why SDE for Stock Prediction: Financial organization hold very high regard for SDEs because of its vast success in the financial world. With the success of Neural ODE, and similar methods of training incorporated into Neural SDE, it is a fairly obvious method to try and model stock prices. Since we could not find any published literature of implementation of Neural SDE based stock prices prediction, we were eager to learn and make use of this new powerful tool for our objective.

3 Dataset

The data set used in the project is the stock price information of **Reliance Industries Ltd.** from January 2015 to August 2020. The idea was to inspect the robustness of the model by predicting the Coronavirus market crash via data learned from the 2017-18 stock market crash. The ability to be able to predict the quick recovery of RIL after Coronavirus crash was another parameter to evaluate the model upon.

The dataset was extracted from the NSEpy library[1] for Python. The feature columns in the dataset were *Date, Symbol, Series, Prev. Close, Open, High, Low, Last, Close, VWAP, Volume, Turnover, Trades, Deliverable Volume* and *Percent Deliverable*. We extracted five features, namely, *Date, Open, High, Low, Close*. The final feature we wanted to predict was the closing price of Reliance Industries.

Short term prices (say, an hour's) are extremely noisy and sensitive, which makes it difficult to model. Therefore, it is common to use at least a day's statistics for training. With approximately 252 working days in a year, for 6 consecutive years, our data set comprised of 1401 data samples. It was divided into the training (1121) and test data(280) in a **80%-20% ratio**.

Figure 3: Historic data of Closing Stock Price of Reliance vs. Time (in Years)

For prediction of time series data, a window of past data is commonly used as an input since recent events hold a higher significance in time related predictions compared to relatively older events. Intuitively, a company's stock price tomorrow will directly be impacted by what it was today, while it's value (say) 5 months back won't affect it much. We attempted to predict the closing price ($\mathbf{p}$) of each day based on closing price of past $\mathbf{K}$ days. Mathematically-

$$\mathbf{Y}^n = \mathbf{p}^n$$

$$\mathbf{X}^n = \left[\mathbf{p}^{n-K}, \mathbf{p}^{n-K+1}, \ldots, \mathbf{p}^{n-2}, \mathbf{p}^{n-1}\right]^T$$

4 Tools and Frameworks used

The entire project was compiled on Jupyter based online IDE platform- Google Colab. LSTM was implemented using Keras's inbuilt LSTM class. Neural ODE and SDE models were built on PyTorch library. To implement advanced ODE solvers for Neural ODE, TorchDyn Library [6] was imported. *Standard Scaler* function was extracted from the Scikit-Learn Library [7] to scale the dataset. Lightening Module interface was used for trainer-model interaction.

5 Model Architecture

5.1 LSTM Model

We tried out combinations of multiple LSTM, Dense and Regularising layers with different combination of number of parameters. The model architecture that gave the best results is described below:

- **Input:** The input to the model was a sequence of closing price of past 64 days. Each of these inputs belong to the 80% training examples which are further normalized using *Standard Scaler* as $x_{\text{scaled}} = \frac{x-\mu}{\sigma}$. The final shape of the *Training Input Set* was (1057, 64, 1) and of the *Testing Input Set* was (216, 64, 1).
- **LSTM Layer:** Keras based LSTM layer in the sequential model took the sequence of inputs with 64 units and returned an output of size of 64 units.
- **Dropout Layer:** It followed the LSTM layer and was introduced because high depth models are very susceptible to overfitting. Dropout regularized the model and ensuring better performance over test data. The tuned dropout rate was 25%.
- **Fully Connected Dense Layers:** The Dropout followed 4 Fully Connected layers with dimensions as given in the figure 4. All these layers used ReLU activation function which is defined as
  $f(x) = \begin{cases} x, & x > 0 \\ 0, & \text{otherwise} \end{cases}$
- **Output:** The model output is the closing stock price given each input sequence. The final shape of *Training Output set* used was (1057,) and *Test Output set* used was (216,).

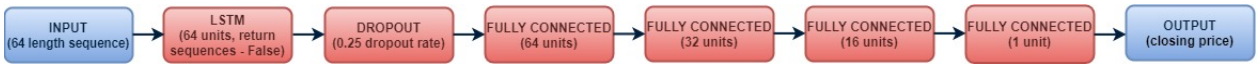


Figure 4: LSTM Model Architecture

Early Stopping: We used Early Stopping as another measure to avoid overfitting and prevent excessive training. The strategy was to stop training once the validation error stopped decreasing.
Hyper-parameters: The tuned hyperparameters associated with the best model are given in Table 1:

Hyperparameter	Value	Hyperparameter	Value
Epochs	150 (maximum)	Batch Size	32
Loss Function	MSE	Optimizer	Adam
Initial learning rate	0.001	$\beta_{1,\text{adam}}$	0.9
$\beta_{2,\text{adam}}$	0.999	Sequence Length (K)	64

Table 1: Hyper-Parameters of the LSTM Model

5.2 ODE Model

We used TorchDyn[6] module, part of the broader DiffEqML software ecosystem. It offers an intuitive access-point to model design for continuous-depth learning and implementing advanced ODE solvers like dopri, Milstein, Runge-Kutta, which works better in terms of speed and accuracy compared to lower order solvers. The model architecture is explained as follows:

- **Input Layer:** The input was a sequence of 22 days past prices, scaled and changed into Tensor datatype to be used in PyTorch Modules. The final shapes of the *Training Input* and *Test Input* were (1089, 32) and (248,32) respectively. The following layers were a part of the Neural ODE Solver.
- **Dense Layers:** The input was fed into a sequential network of 4 fully connected layers. Each layer uses ELU activation function, which is defined as
$$f(x) = \begin{cases} x, & x > 0 \\ e^x - 1, & x \leq 0 \end{cases}$$
These dense layers were then solved using a dopri8 ode solver.
- **Fully Connected Layer:** The output of ode solver was fed into another fully connected layer which gave a scalar as an output.
- **Output:** Given each input sequence, the final *Training Output data set* has the dimension of (1089,) and the *Test Output data set* has the dimensions (248,)

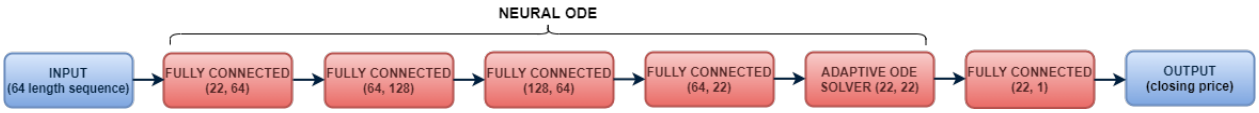


Figure 5: Neural ODE Model Architecture

Lightning Module: A standard PyTorch lightning library was used for training.

Hyperparameters: The tuned hyper parameters associated with the model are given in Table 2

Hyperparameter	Value	Hyperparameter	Value
Epochs	600	Sequence Length (K)	22
Loss Function	MSE	Optimizer	Adam
Learning rate	0.001	Weight Decay (L2)	0.1
atol (Absolute Tolerance)	1.4×10^{-5}	ODE Solver	dopri8
Sensitivity	adjoint	Train-Test split	80%-20%

Table 2: Hyper-Parameters of the Neural ODE Model

5.3 SDE Model

The authors of [3] implemented a library TorchSDE [8] for solving Neural SDEs. However, after confronting issues while implementing their package, we took inspiration from their code and decided to implement a basic Neural SDE architecture on PyTorch. It didn't have the advanced adjoint sensitivity for back-propagation, nor the Brownian tree implementation for the Noise Layer. Also, [3] had a latent variable implementation since they were trying to model dynamics with missing data. We were (then) not acquainted with LVMs and missing data for stock prices was uncommon, therefore, we didn't assume a prior distribution for the drift and diffusion of the SDE.

- **Input Layer:** Similar to the ODE, the input to the model consisted of the closing price of past 22 days, which was scaled and converted into a tensor suitable for PyTorch class operations. The final shapes of the *Training Input* and *Test Input* were (1089, 22) and (248,22) respectively. The following layers are a part of the Neural SDE Solver.
- **Upscaling:** The input was upscaled from a size of 22 to size of 50 using a fully connected layer for being fed into a Sequential network of Drift and Diffusion. The fully connected layer was ReLU activated.

- **SDE code formulation:** Since we implemented the architecture of SDE from scratch, we used the Euler solver instead of sophisticated adaptive solver for integrating the differential equation. The SDE was further simplified using reference from [9] as-

$$dY_t = \mu(Y_t, X_t, t)dt + \sigma(Y_t, X_t, t)dW \sim \mu(Y_t, X_t, t)dt + \sigma(Y_t, X_t, t)\sqrt{t}\,\mathcal{N}(0, 1) \tag{1}$$

  where $\mathcal{N}(0, 1)$ is a random sequence of size 50 extracted from a 1D gaussian with zero mean and standard deviation of one. The *Drift* and *Diffusion* of the SDE were networks described below.

- **Drift Layer:** Drift layer's task was to capture the trend (increasing, decreasing, or stable) based on past prices, and was a ReLU activated fully connected with input of size 50 and output of size 50.

- **Diffusion Layer:** The diffusion layer learnt the dataset noise over its trend, and had a sequence of two fully connected ReLU activated layers with hidden layer size of 100, and output size of 1 (scaler).

- **Output:** The forward pass was executed using the Eq. 1, with μ and σ calculated from the drift and diffusion layers to get $Y_{t+1} = Y_t + dY_t$.

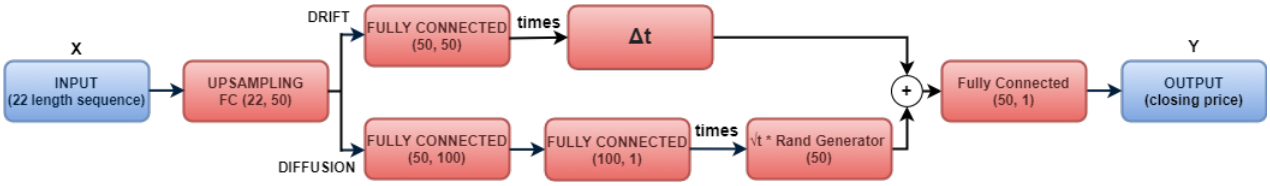


Figure 6: Neural SDE Model Architecture

Hyperparameters: The tuned hyper parameters associated with the model are given in Table 3

Hyperparameter	Value	Hyperparameter	Value
Epochs	200	Sequence Length (K)	22
Loss Function	MSE	Optimizer	SGD
Learning rate	1e-4	Weight Decay (L2)	5e-4
Learning rate Decay	0.1	ODE Solver	Euler
Momentum	0.9	Train-Test split	80%-20%

Table 3: Hyper-Parameters of the Neural SDE Model

6 Experimental Observations

We ran many experiments on all three of our models with ranging parameters and hyperparameters to get the best results and Fig. 7 shows the snapshot of our predictions.

The three models comfortably fitted the train dataset, while SDE gave superior results over the other two on test data. The Mean Squared Error on the test data is shown in Table 4.

Model	Test MSE
LSTM	0.11783
Neural ODE	0.07701
Neural SDE	0.0587 $\pm$0.0001

Table 4: Models Comparison

Other major observations on the model performance are-

- The Neural ODE model gave better prediction than LSTM (by $\sim$35%) for the given dataset. Neural SDE implementation further improved the accuracy of prediction ($\sim$50% of LSTM MSE) (see Fig. 7).
- Neural ODE model took lot more iterations than Neural SDE to converge (600 vs 200).

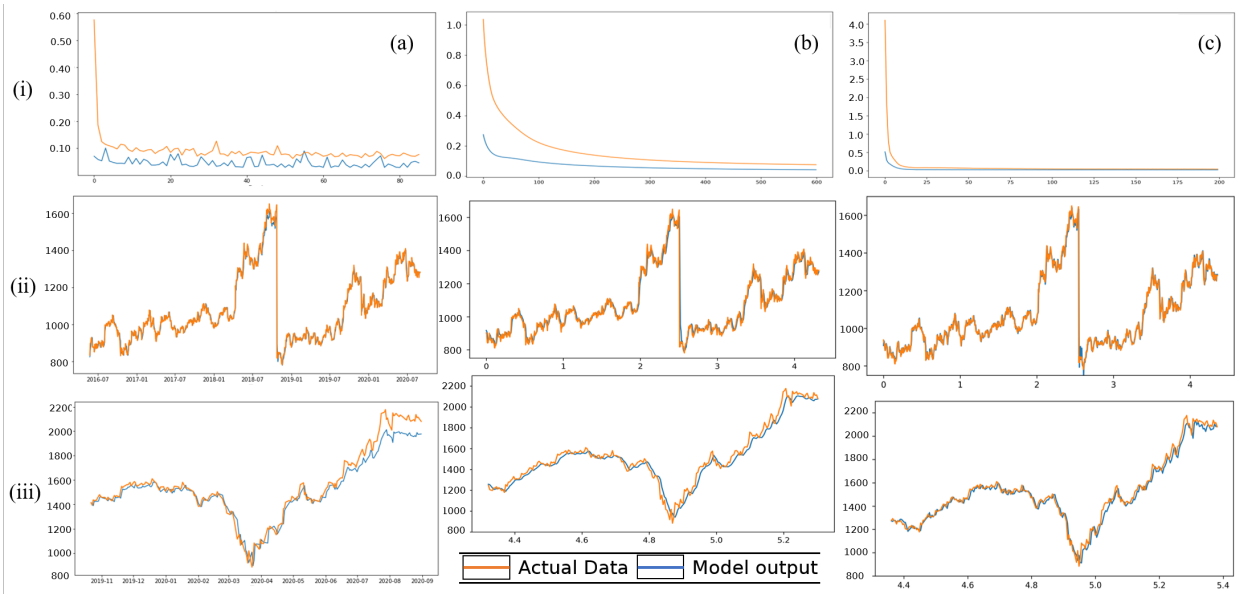


Figure 7: (i) MSE Loss history, (ii) Train Set Prediction, and (iii) Test Set Prediction for (a) LSTM, (b) Neural ODE, and (c) Neural SDE models. The vertical axis for (ii-iii) are in INR. Horizontal axis of (i) is number of epochs, while (ii-iii) are Date or time per year. **The blue and orange legends for (i) are Training loss and Test loss respectively**

- LSTM test price prediction diverged from its true value while trying to predict the quick recovery of RIL after Coronavirus crash.
- All the three models were successful in predicting the Coronavirus crash after learning from the 2017-18 market crash. Neural ODE and Neural SDE were also able to predict the stock's quick recovery.
- Neural ODE has a tendency to over-fit because of its assumption that the dynamics is deterministic. This is reflected in its higher MSE than Neural SDE's, which models noise intrinsically.
- Presence of uncertainty element in the SDE architecture led to different models with each training. Therefore, MSE reported for Neural SDE is over 5 different random seeds while training.

7 Future Work

The original papers of the differential models [2, 3] were based upon Latent models while we didn't model the input or the features (like drift and diffusion) as a latent variable. Probabilistic modeling of these variables is a future task that may further improve the accuracy.
The differential equation of Neural SDE had an Euler update rule which is known to accumulate error as it integrates. Therefore, the training of the model can be improved by implementing adaptive SDE solvers like Milstein, Stochastic Runge-Kutta Method, etc.
The backward propagation step for SDE is slow and memory intensive. Therefore, the existing code be further refined by integrating the TorchSDE package [8] which uses adjoint sensitivity and Brownian tree to back-propagate.

References

[1] NSEpy library: https://nsepy.xyz/
[2] Li et al.; Scalable Gradients for Stochastic Differential Equations, 2020, arXiv.
[3] Chen et al.; Neural Ordinary Differential Equations, 2018, arXiv.
[4] Staudemeyer et al; Understanding LSTM - A tutorial into Long Short-Term Memory Recurrent Neural Networks, 2019, arXiv.
[5] Hochreiter et al.; Long Short-Term Memory, 1997, DOI.
[6] Poli et al.; TorchDyn: A Neural Differential Equations Library, 2020, arXiv
[7] Pedregosa et al.; Scikit-learn: Machine Learning in Python, JMLR, **12**, pp. 2825-2830, 2011, arXiv
[8] Li et al.; TorchSDE: A Stochastic Differential Equations Library, 2020, URL
[9] Moehlis, Jeffrey M., Notes on Wiener Process, Special Topics in Biological Dynamics, Fall 2001, URL