Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Civil and Environmental Engineering

Saket Jain

# *Application of Machine Learning methods to flow problems in unsaturated soil*

Master's thesis in MSc Geotechnics and Geohazards
Supervisor: Prof. Rao Martand Singh, NTNU, Dr. Ivan Depina (Research Scientist, SINTEF), Mr. Emir Ahmet Oguz (PhD Candidate, NTNU)
June 2020

NTNU
Norwegian University of Science and Technology

KLIMA DIGITAL°

# 1. Preface

This thesis is about the application of different machine learning techniques to the process of infiltration in the field of Geotechnical Engineering. It is a part of the project Klima Digital, which is a spin-off project of Klima2050 in collaboration with SINTEF. This report fulfils the requirements of TBA4900: Geotechnical Engineering, Master's Thesis (30 Credits), as part of International program in MSc Geotechnics and Geohazards at NTNU, Trondheim, during spring semester of 2020.

Trondheim, 11th June 2020

(Saket Jain)

# 2. Acknowledgement

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend my sincere thanks to all of them.

I am highly indebted to *Prof. Rao Martand Singh (Supervisor - NTNU), Dr. Ivan Depina (Co-Supervisor – SINTEF), and Mr. Emir Ahmet Oguz (Co- Supervisor - NTNU)* for their guidance and constant supervision as well.

As for providing necessary information regarding the project & also for their support in completing the project.

I would like to express my gratitude towards members of Norwegian University of Science and Technology (NTNU), Trondheim, Norway and SINTEF

for their kind co-operation and encouragement which helped me in completion of this project.

My thanks and appreciations also go to my friends for their help in developing the project and people who have willingly helped me out with their abilities.

*Thank you!*

# Contents

1. Preface ..... 2
2. Acknowledgement ..... 3
3. Abstract ..... 5
Chapter 1 ..... 6
Introduction ..... 6
1.1 Background ..... 6
1.2 Objectives ..... 6
1.3 Limitations ..... 7
1.4 Approach ..... 7
1.5 Structure of the Report ..... 7
Chapter 2 ..... 14
Machine Learning Techniques to simulate infiltration ..... 14
2.1. Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) ..... 17
2.2. Physics-Informed Neural Network (PINN) ..... 27
Chapter 3 ..... 31
Theorical Background of Infiltration and Data Generation ..... 31
3.1 Richard's Equation ..... 31
3.2 Soil Water characteristic curve (SWCC) ..... 32
3.3 Soil Matric Potential or Pressure head $\psi$ ..... 33
3.4 Modelling SWCC ..... 33
3.5 Soil type description ..... 34
3.6 Data Generation ..... 37
Chapter 4 ..... 42
Modelling with Python Code ..... 42
4.1 Long Short-Term Memory (LSTM) or Time series prediction ..... 43
4.2 Physics-Informed Neural Network (PINN) ..... 45
Chapter 5 ..... 47
Results & Discussions ..... 47
5.1 LSTM ..... 47
5.2 PINN ..... 55
5.3 Discussion ..... 58
Chapter 6 ..... 60
Conclusions ..... 60
References ..... 62
Appendix 1 ..... 63

# 3. Abstract

Machine Learning (ML) is showing promising results in various fields of science and engineering. In this thesis, idea to apply machine learning to the infiltration process in the soil is explored. In order to do this, two main Machine Learning techniques are identified, Long Short-Term Memory (LSTM) and Physics Informed-Neural Networks (PINN). Both of these techniques use very different concepts to achieve the same goal. LSTM is used for sequential or timeseries data, therefore values of water content $(\theta)$, and pressure head $(\psi)$ were calculated and arranged in space and time. PINN uses the underlying Richard's equation to mimic infiltration. Both techniques have their own drawbacks but in this study PINN proved to be better than LSTM. All the modelling was done using Python 3.6 in Sypder, Anaconda.

# Chapter 1
# Introduction

## 1.1 Background

The knowledge of hydrophysical properties of soil is extremely valuable in several disciplines of science all the way ranging from agriculture to ecology *[1]*. Hydrophysical characteristics of soil i.e., water retention curve and hydraulic conductivity in saturated and unsaturated zones have been historically measured experimentally or estimated using mathematical or statistical models. However, due to the recent developments in the field of Artificial Intelligence (AI) and Machine Learning (ML), we have come closer to solve such intricate problems in the field of geotechnical engineering, using AI or ML. Moreover, due to our ever-increasing computing power (which follows Moor's law) and the rise of importance and the amount of data, these methods have gained significant importance in the recent times. This provides us with an opportunity to develop methods based on this data science of Machine learning, to compete or complement our knowledge/models of these physical processes.

In Machine Learning, Artificial Neural Networks (ANNs) are used to identify patterns and trends in data which can be missed otherwise. Historically, this is implemented to solve several problems in the field of geotechnical engineering. Most of these applications were on liquification analysis, pile foundation, slope stability, particularly where finding analytical solutions were difficult *[2][3]*. Other applications included settlement of foundations, soil property estimation, site characterization, parameter estimation, prediction of the movement of slopes. Another technique called Convolutional Neural Network (CNN) which specialize in image recognition, has been used for grain size distribution using images, landslide susceptibility mapping etc. Similarly, there are other techniques in Machine Learning, which have been used in past to solve several other problems in geotechnical engineering. Table 1 gives a list of research done with ML and AI techniques to solve geotechnical problems. In this thesis, the infiltration process in unsaturated soil has been studied by using Machine Learning.

## 1.2 Objectives

The main objective of this thesis was to develop a machine learning model which can replace the physical models to replicate the infiltration process in an unsaturated soil. Moreover, one of the major objectives of this research is also to explore the problems which can be addressed in geotechnical Engineering using Machine Learning. The objectives of this thesis are as follows:

- Identification of Different Machine Learning techniques which can be used to mimic infiltration process into the soil mass.
- Modelling our data in a way which is suitable to the ML technique to process.
- Identifying the potential and limitations of these techniques by studying the results.
- Discussing other problems in geotechnical engineering, which can be addressed using these and other methods in ML.

## 1.3 Limitations

The scope of this study is limited to theoretically generated data. Therefore, performance of the models will be needed to be tested on experimental data, which is outside the scope of this thesis. Sometimes ML models are very specific to datasets. Therefore, they might need to be optimized in order to use them for another dataset. Moreover, the models suggested can be studied more given noise in the data, but ultimately it mainly boils down to the lack of time. Lastly, COVID-19 has definitely affected the work pace of this thesis.

## 1.4 Approach

Two Machine Learning techniques namely Long Short-Term Memory (LSTM) and Physics informed Neural Networks (PINNs) were identified to simulate infiltration. After a detailed understanding of these techniques data was generated using a Python code named as *RichardsEquationdatagenerator.py*. Then, the data was modelled to feed both the algorithms. Afterwards, results were studied separately of the individual techniques. Finally, they were compared to discuss which technique should be preferred.

## 1.5 Structure of the Report

The structure of the report is as follows:

- Chapter 1 outlines the objectives of the study.
- Chapter 2 gives a detailed understanding of the Machine Learning Techniques used.
- Chapter 3 introduces to the background of Infiltration Process and Data Generation.
- Chapter 4 introduces and explains the Python code and how does it address Infiltration through LSTMs and PINNs.
- Chapter 5 presents and discusses the results produced by both techniques
- Chapter 6 states the conclusions of the thesis.

| No | Researchers | Data collection methods | Techniques | Results |
|---|---|---|---|---|
| 1 | Pile driving records Reanalysed using neural networks | | | |
| | Goh 1996 | Actual pile driving records | Back Propagation Neural Networks | They indicated that the neural network predictions are more reliable than the conventional pile driving formulae |
| 2 | Application of an Artificial Neural Network for Analysis of Subsurface Contamination at the Schuyler Falls Landfill, NY | | | |
| | Rizzo and Dougherty 1996 | Historical Data | Artificial Neural Networks | Applied and tested a new pattern method on a variety of site characterization problems, called it “SCANN” (Site characterization using Artificial Neural Networks), Unlike the kriging methods, SCANN is data-driven and requires no estimation of a covariance function. It uses a feed-forward counter propagation training approach to determine a "best estimate" or map of a discrete spatially distributed field. |
| 3 | Prediction of Pile Bearing Capacity Using Artificial Neural Networks | | | |
| | Lee and Lee 1996 | In situ pile load tests obtained from a literatures | Error Back Propagation Neural Networks | The results showed that the neural networks predicted values corresponding the measured values much better than those obtained from Meyerhof’s equation. |
| 4 | General regression neural networks for driven piles in cohesionless soils | | | |
| | Abu-Kiefa 1998 | Historical Data | General Regression Network | Concluded that the GRNNM is applicable for all different conditions of driven piles in cohesionless soils. |
| 5 | Prediction of Pile Capacity Using Neural Networks | | | |
| | Teh et al. 1997 | Historical Data | Back Propagation Neural Networks | The study showed that the neural network model predicted the total capacity reasonably well. The neural-network-predicted soil resistance along the pile was also in general |

| | | | | |
|---|---|---|---|---|
| | | | | agreement with the CAPWAP solution. |
| 6 | Subsurface Characterization Using Artificial Neural Network And GIS | | | |
| | Gangopadhyay et al., 1999 | Historical Data | Multilayer perceptron using the backpropagation algorithm | The integrated approach of ANN and GIS, is shown to be a powerful tool for characterizing complex aquifer geometry, and for calculating aquifer parameters for ground-water flow modeling. |
| 7 | Artificial intelligence techniques for the design and analysis of deep foundations | | | |
| | Nawari et al., 1999 | Historical Data | NN, and Generalized Regression Neural Network | Based on the results from this investigation, it appeared that the proposed neural network models furnish a pragmatic and a reliable alternative for the current analysis and design techniques of axial pile capacity and laterally loaded piles. |
| 8 | Bayesian Neural Network Analysis of Undrained Side Resistance of Drilled Shafts | | | |
| | Goh et al., 2005 | Historical Data | Bayesian neural network algorithm | The developed neural network model provided good estimates of the undrained side resistance adhesion factor. Furthermore, one distinct benefit of this neural network model is the computation of the error bars on the predictions of the adhesion factor. These error bars will aid in giving confidence to the predicted values and the interpretation of the results. |
| 9 | Undrained Lateral Load Capacity of Piles in Clay Using Artificial Neural Network | | | |
| | Das and Basudhar, 2006 | Historical Data | Back Propagation Neural Networks | The developed ANN model is more efficient compared to empirical models of Hansen and Broms. |
| 10 | Prediction of Friction Capacity of Driven Piles in Clay Using the Support Vector Machine | | | |
| | Saumi, 2008 | Data Base | SVM | With the database collected by Goh (1995) the study shows that SVM has the potential to be a useful and |

| | | | | |
|---|---|---|---|---|
| | | | | practical tool for prediction of friction capacity of driven piles in clay. |
| 11 | Modelling Pile Capacity Using Gaussian Process Regression | | | |
| | Pal and Deswal 2010 | Actual piledriving records in cohesion-less soil | Gaussian Process (GP) Regression and SVM | The GP regression approach works well in predicting the load-bearing capacity of piles as compared to the SVM approach. Another conclusion from this study is that the Pearson VII function kernel performs well in comparison to the radial basis function kernel with both GP- and SVM-based approaches to model the pile capacity. The results of this study also suggest that GP regression works well as compared to the empirical relations in predicting the ultimate pile capacity. |
| 12 | Prediction of Pile Settlement Using Artificial Neural Networks Based on Cone Penetration Test Data | | | |
| | Nejad and Jaksa 2010 | Database | Back Propagation Neural Networks | The results indicate that back-propagation neural networks have the ability to predict the settlement of pile with an acceptable degree of accuracy (r=0.956, RMSE=1.06 mm) for predicted settlements ranging from 0.0 to 137.88 mm. |
| 13 | Intelligent Computing for Modeling Axial Capacity of Pile Foundations | | | |
| | Shahin 2010 | Historical Data | Artificial Neural Networks (ANN) | The results indicate that the ANN models were capable of accurately predicting the ultimate capacity of pile foundations and compare well with what one would expect based on available geotechnical knowledge and experimental results. |
| 14 | Neural Network Model for Predicting the Resistance of Driven Piles | | | |

| | | | | |
|---|---|---|---|---|
| | Park and Cho 2010 | data from dynamic piles load test | Artificial Neural Network (ANN) | The results showed that the ANN model served as a reliable and simple predictive tool to predict the resistance of the driven pile with correlation coefficient values close to 0.9. |
| 15 | Neural Network Application in Prediction of Axial Bearing Capacity of Driven Piles | | | |
| | Harnedi and Kassim 2013 | Pile Driving Analyzer (PDA) | Artificial Neural Network (ANN) | The results showed that the neural network models give a good prediction of axial bearing capacity of piles if both stress wave data and properties of both driven pile and driving system are considered in the input data. |
| 16 | Application of Artificial Neural Network for Predicting Shaft and Tip Resistances of Concrete Piles | | | |
| | Momeni et al., 2015 | Pile Driving Analyzer (PDA) | Artificial Neural Network (ANN) | Founded that a network with five hidden nodes in one hidden layer yields the best performance. Additionally, through a sensitivity analysis, it was founded/ that the pile length and cross sectional area are the most influential parameters in predicting the bearing capacity of piles |
| 17 | Analysis of Ultimate Bearing Capacity of Single Pile Using the Artificial Neural | | | |
| | Wardani et al., 2013 | Full-Scale Pile Load Test and SPT | Artificial Neural Network (ANN) | The results showed that neural networks can be used for prediction of ultimate bearing capacity of single pile foundation and the model have the highest performance among the other methods, even though the difference is not too big. |
| 18 | ANN Prediction of Some Geotechnical Properties of Soilfrom their Index Parameters | | | |
| | Tizpa et. al 2014 | Database | Arificial Neural Network (ANN) | Comparison between the results of the developed models and experimental data indicated that predictions are within a confidence interval of 95 %. According to the performed sensitivity analysis, |

| | | | | |
|---|---|---|---|---|
| | | | | Atterbeg limits and the soil fine content (silt+clay) are the most important variables in predicting the maximum dry density and optimum moisture content. |
| 19 | Load–settlement modeling of axially loaded steel driven piles using CPT-based recurrent NNs | | | |
| | Shahin 2014a | Pile Load Tests, and (CPT) Data | Recurrent neural network (RNN) | Founded that the developed RNN model has the ability to reliably predict the load–settlement response of axially loaded steel driven piles, and thus, can be used by geotechnical engineers for routine design practice. |
| 20 | Evolutionary-Based Approaches for Settlement Prediction of Shallow Foundations on Cohesionless Soils | | | |
| | Shahnazari et. al 2014 | Historical Data | Polynomial regression, genetic programming (GP), & gene expression programming (GEP) | In this study, the feasibility of the EPR, GP and GEP approaches in finding solutions for highly nonlinear problems such as settlement of shallow foundations on granular soils is also clearly illustrated |
| 21 | State-of-the-Art Review of Some Artificial Intelligence Applications in Pile Foundations | | | |
| | Shahin 2014b | Historical Data | Artificial intelligence | AI techniques perform better than, or at least as good as, the most traditional methods. |
| 22 | Artificial Neural Network Model for Prediction of Bearing Capacity of Driven Pile | | | |
| | Maizir et. al 2015 | Pile Driving Analyzer (PDA) test data | Artificial Neural Network | The results show that the ANN model serves as a reliable prediction tool to predict the resistance of the driven pile with coefficient of correlation (R) values close to 0.9 and mean squared err (MSE) less than 1%. |
| 23 | Toward improved prediction of the bedrock depth underneath hillslopes: Bayesian inference of the bottom-up control hypothesis using high-resolution topographic data | | | |

| | | | | |
|---|---|---|---|---|
| | Gomes et al. 2016 | High-resolution topographic data | Numerical modeling, and Bayesian analysis | The results demonstrate that the proposed DTB model with lumped parameters mimics reasonably well the observed regolith depth data with root mean square error (RMSE). |
| 24 | Determination bearing capacity of driven piles in sandy soils using Artificial Neural Networks | | | |
| | Mazaher and Berneti 2016 | Database | MLP Neural Network | The NN has very high efficiency in predicting load carrying capacity of metal piles, and it is concluded that soil internal friction angle, soil elastic modulus, pile diameter and pile length respectively have maximum effect on load carrying capacity of piles. |

*Table 1 - Summary of some applications of AI and ML techniques in geotechnical engineering [3].*

# Chapter 2
# Machine Learning Techniques to simulate infiltration

In this thesis, an attempt was made to predict the pore pressure head, and the water content in unsaturated soil by two Machine Learning techniques. First technique is called Long Short-Term Memory (LSTM). It is an extension of Recurrent Neural Network and has been explained in detail in the sections below. This technique required to pose this infiltration problem as a time-series prediction or sequential data problem.

LSTM is a very powerful and proven technique whose applications can be seen for various timeseries data emanating from sensors, stock markets and government agencies. In addition to these, this technique is also pretty good at text generation, sequencing genomes, handwriting recognition, Natural Language Processing (NLP), and even at music generation *[4]*. Before proceeding on to the original data set, this technique was tested on opening price of google stocks on NASDAQ for the last 3.5 years. Then a prediction was made of the opening stock price of the same for the 20 days. Figure 1 below shows the values of opening stock price for the last 3.5 years and Figure 2 shows real vs the predicted price for the next 20 days. This can be refined and tuned to produce much better results than this. Furthermore, same technique was also tested on another two datasets. Figure 3 shows the result of the 1st dataset which is generated using a sine curve with some noise. In this case, model is trained from 0 to 200 timesteps and predicts from 201 to 400 timesteps. Result of second dataset is shown in Figure 4, where a damping equation is used to generate data without noise. Whereas, model is trained for 0 to 100 timesteps and predicts from 101 to 200 timesteps.

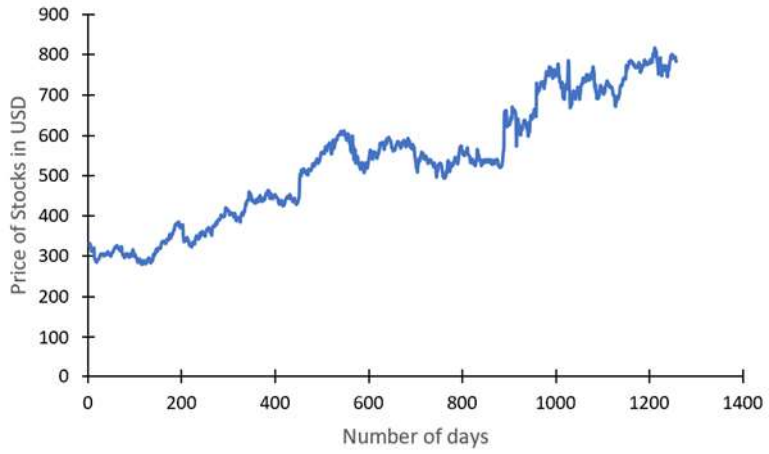



*Figure 1 - Opening Stock prices of google at NASDAQ for the last 3.5 years.*

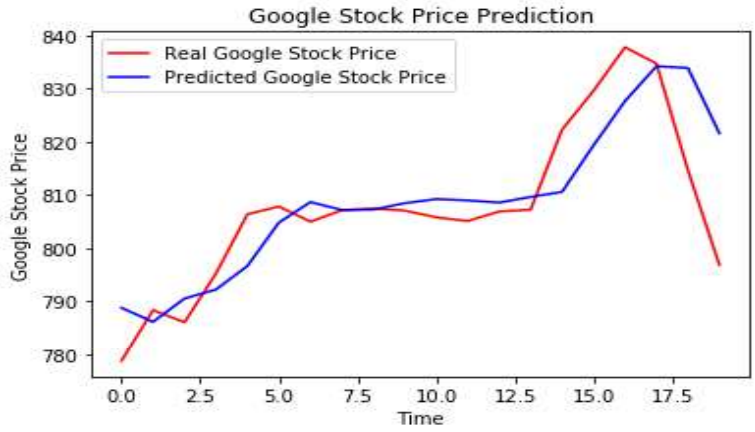



*Figure 2 - Real vs predicted opening stock prices of google at NASDAQ for the next 20 days.*

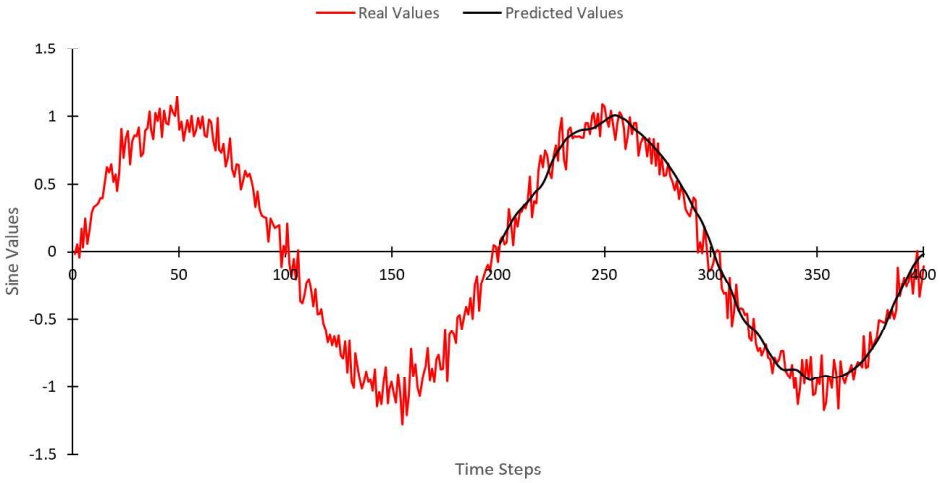



*Figure 3 – Real vs predicted values of a sine curve with noise (0 – 200 training set, 201 – 400 testing/validation set)*

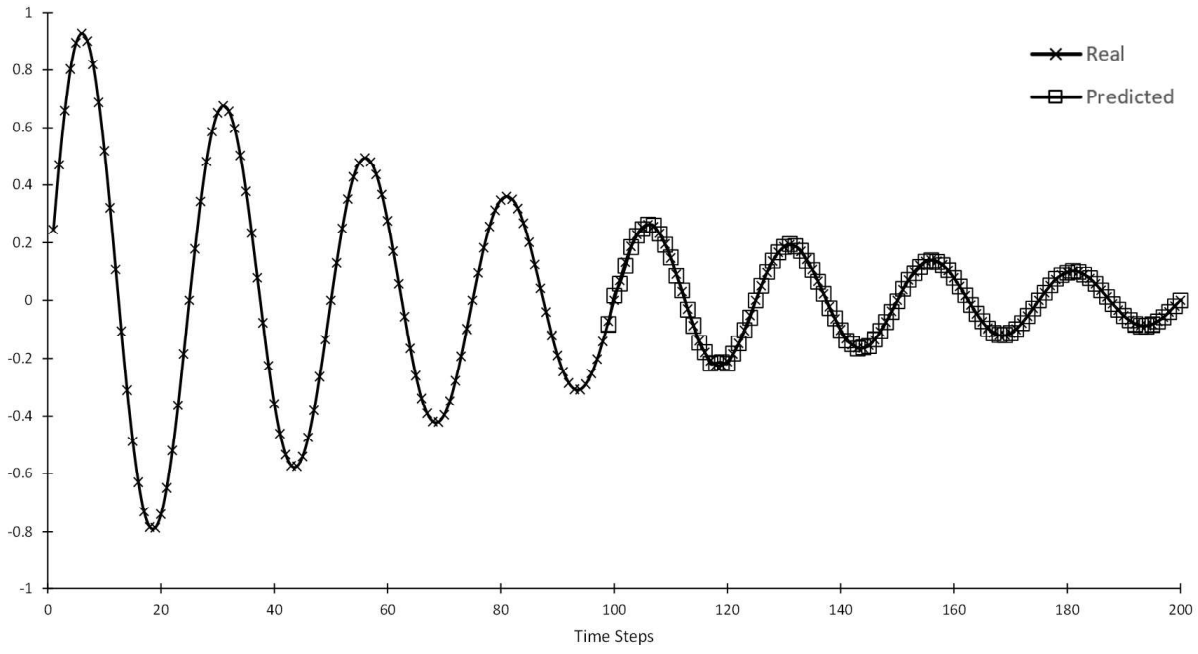



*Figure 4 – Real vs predicted values on a damping curve (0 – 100 training set, 101 – 200 testing/validation set).*

After LSTMs, another ML technique, Physics Informed Neural Networks (PINNs), was tried to mimic infiltration. This technique helps us to move forward from an approach, in which huge amount of data is fed into deep learning algorithms, to extract knowledge and hidden patterns in the data. It is done in a manner, which is agnostic to the underlying scientific principles driving these variables, therefore techniques like LSTMs are also called Black Box. These black box models have been very successful and show very promising results in commercial problems, computer vison, speech recognition etc [5],[6]. However, these techniques don't really work on a lot of scientific problems, often because of the lack of scientific data required for these models. Moreover, since these methods are black box methods, interpretability is very limited. This is very important especially in any scientific application, because that will be the basis for the further scientific research.

We can better understand with the dichotomy *(* Figure 5*)* between Theory – based data science models (PINNs) verses Data Science models *[7]*. X- axis represents the amount of data being used, and Y-axis represents the amount of theory utilized. In the green region, there are purely theory-based models, based on equations, scientific theories, numerical models etc. Despite their huge progress, they contain certain significant knowledge gaps, to describe certain processes that are either too complex to understand or too difficult to observe directly. In the blue, we have data science models, that have ample amount of data, but agnostic to the underlying scientific theories. Both green and blue zone make an ineffective use of knowledge of scientific theory and data. Therefore, there is a need for developing data science methods which can use both scientific knowledge and data on an equal footing. This is the paradigm of Theory-guided data science, that tries to take unique ability of data science methods to automatically extract knowledge and pattern from data but without ignoring the treasure accumulated in scientific theories.

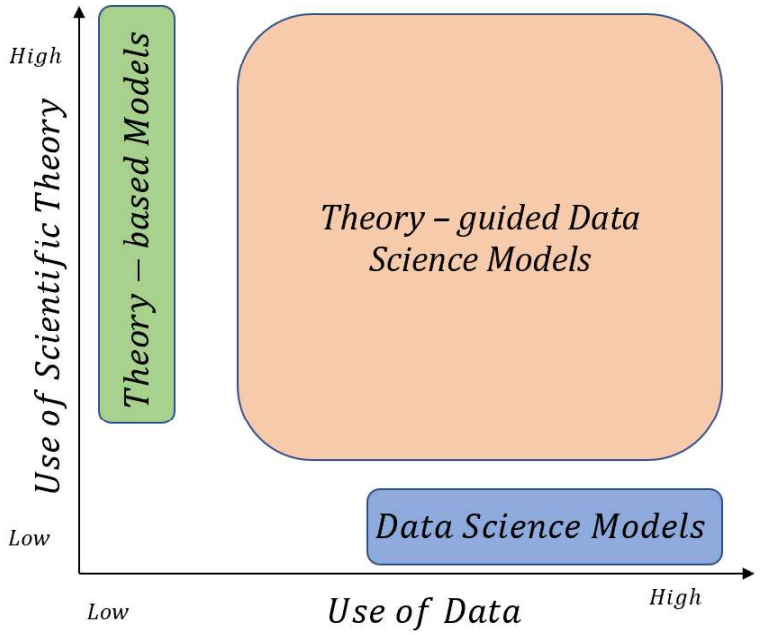



*Figure 5 - Dichotomy between scientific models vs data-science models.*

## 2.1. Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM)

Long Short-Term Memory Networks or in short LSTM networks are an extension of Recurrent Neural Networks (RNN). In order to understand LSTM, we first need to know Neural Networks.

Neural Networks are set of algorithms which are designed to closely mimic the working of a human brain to find and identify patterns in different forms of data (Figure 6 & Figure 7). This network comprises of several units of *Neurons/Perceptrons,* which are connected by *synapses* or *weights*. A biological neuron gives a response to a stimulus. This response is passed over to the next neuron in the network via synapses, and this continues. An artificial Neuron does the same by taking the input number as a stimulus. In response, it will perform a calculation on this number via some activation function like sigmoid. Then this result will be multiplied by a synaptic weight and passed on as an input (stimulus) to the next neuron in the network. It usually takes a network of multi-layer Neurons to successfully complete the training process and it is achieved by adjusting the synaptic weights in the network until a particular input leads to a target output.

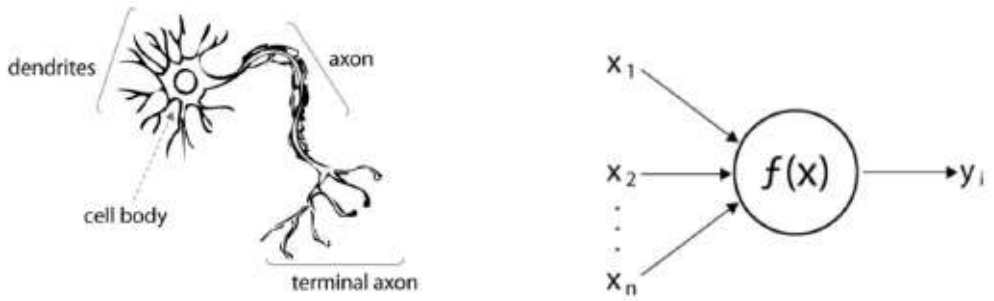



*Figure 6 - Shows the biological Neuron (left) and mathematical Neuron (right)*

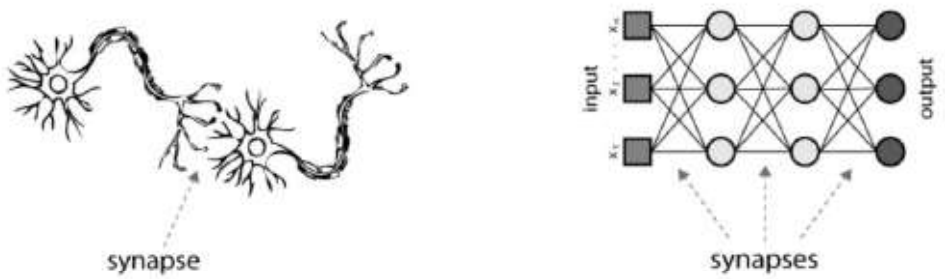



*Figure 7 - Shows the mathematical equivalent of biological synapse*

Recurrent Neural Networks or RNN's are the best suited algorithm for sequential data and have enormous applications like of which in Apple's Siri and Google's voice search/recognition, handwriting recognition, music generation etc. It is quite suitable for machine learning problems which involve sequential data, due to its ability to remember its input. Being recurrent in nature, it performs the same operation for every input, while the output of the current input depends on the previous computation. The produced output is then copied and sent back to the recurrent neural network as an input. To make a decision, it considers the current input and the output that it has learned from the previous input.

RNN's can be understood easily by the following example of a perfect roommate (because he cooks everyday), which is inspired from a book Deep Learning: Grokking *[8]*. Let's assume this perfect roommate is actually very organized and very methodical, and therefore he cooks in rotating sequence i.e., 1st day apple pie, 2nd day Burger, 3rd day chicken and then repeat. Therefore, it can be predicted what he is going to cook today based on what he cooked yesterday. Hence, his cooking schedule somewhat looks like Figure 8 starting with an apple pie on Monday. In Figure 9 we can see the output from last time, is being fed as an input for this time. Hence, this network is recurring in nature and therefore, called Recurrent Neural Network.

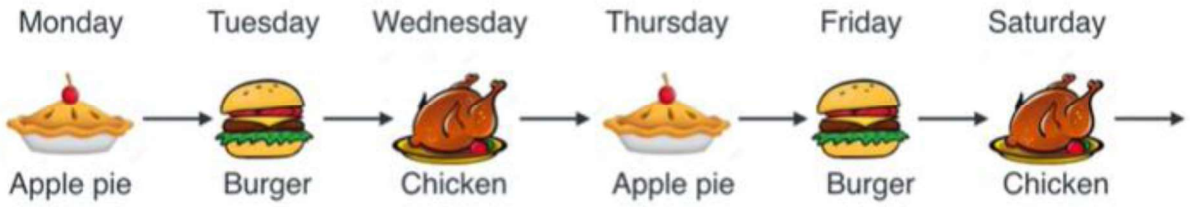



*Figure 8 - Shows Cooking schedule of the perfect roommate [8]*

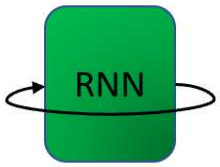



*Figure 9 - A typical RNN unit and its input*

However, RNN's usually have two inputs: one is a present input and the other is the output of the last computation looped in as input. This also can be understood by a very similar example again inspired from the textbook Grokking Machine Learning *[8]*. Again, we have the example of this perfect roommate. He is still very methodical and organized, but now his rule for cooking is a combination of two rules. He still cooks in the same sequence of Apple pie, Burger and Chicken, but now his decision to cook also depends on the weather. If it's sunny, he will go outside and enjoy the day and therefore, he will not be cooking and will just give the same thing as yesterday i.e., leftovers. If it's rainy he will stay at home and will cook the next dish on the list. In Figure 10, we can see on Monday he made an apple pie. On Tuesday we checked the weather and its sunny, so we get the apple pie from Monday. And Wednesday turns out to be rainy, so we get the next thing on the list i.e., Burger. On Thursday its rainy again so Chicken and on Friday its sunny so we get the chicken from Thursday, and so on and so forth. Therefore, an RNN like this looks like the one in Figure 11.

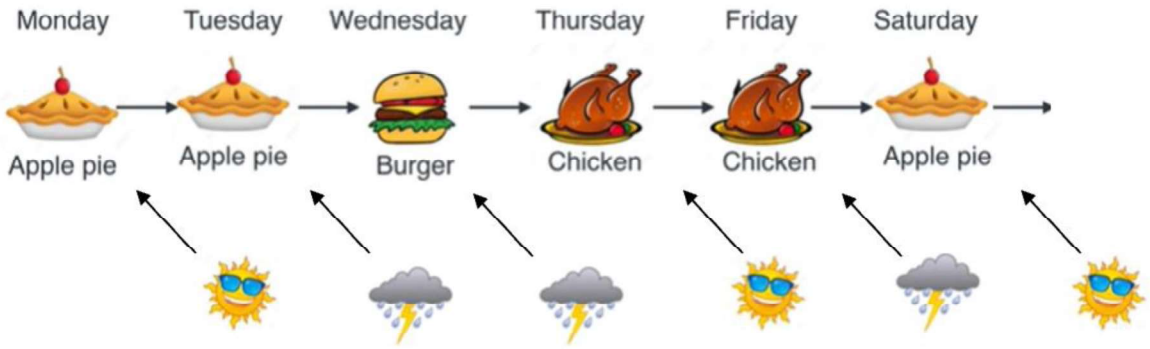



*Figure 10 - Cooking Schedule with weather [8]*

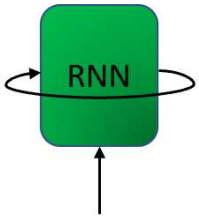



*Figure 11 – A typical RNN unit with two inputs*

In short, RNN has a short memory. While making a decision, it considers the current input and also what it has learned from the inputs it received previously. Therefore, RNN's are good at predicting sequential data. However, there are still two major issues that RNN's have had to deal with, exploding gradients and vanishing gradients.

Exploding gradients occurs when algorithm without much reason assigns an unreasonably high importance to the weights. Fortunately, this problem can easily be solved by truncating or squashing the gradients. On the other hand, vanishing gradient occurs when the value of gradient is very small, i.e., the learning rate of the model is practically zero. It was a major problem during 1990s and much difficult to solve than the exploding gradients. Fortunately, it was solved through the concept of LSTM by Sepp Hochreiter and Juergen Schmidhuber *[4]*.

## A mathematical perspective

In order to proceed with LSTM, we should take a look at RNN and vanishing gradient problem from a mathematical perspective. Then, we can have a clearer picture how LSTMs are effective in solving the underlining problem with RNN. Let's start off with a basic formula of RNN and then visualize it. It works on the following recursive formula.

$$S_t = F_w(S_{t-1}, X_t) \quad (1)$$

Where, $X_t$ is the input at time step t, $S_t$ is the state at time step t and $F_w$ is the recursive function.

Let's look at the simplest representation of RNN and call it a simple RNN *(* Figure 12*)*. In our example, the recursive function is a $tanh$ function. In equation (2) we multiply the input state $X_t$, with weights of X which is $W_X$. While, the previous state $S_{t-1}$ is multiplied with $W_s$, which is a weight of State or S. The sum of the two values is passed through the activation function $tanh$, which gives us the current or new state $S_t$. In order to get an output vector, we multiply the new state with $W_y$ as *in* Figure 12.

$$S_t = tanh(W_s S_{t-1} + W_X X_t) \quad (2)$$

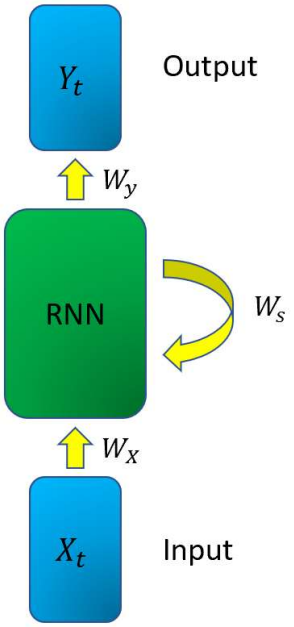



*Figure 12 - A simple RNN*

In unrolled RNN *(* Figure 13*)*, we have a previous state $S_0$, and the input at time step 1 is $X_1$. The RNN calculates the new state $S_1$, based on this recursive formula, and gives us the output $Y_1$, by multiplying it with the weight, $W_y$. In the next time step, this new state $S_1$, and $X_2$ serves as the input and give the next state $S_2$, and then the output $Y_2$. This same thing goes on for many times steps, but here it's important to note that, same weights are used throughout the network i.e., $W_X$, $W_s$, and $W_y$. In multilayer RNN, the output generated as $Y_1$, and $Y_2$ serves as input as shown in Figure 14.

As we know RNN learns through backpropagation through time*. We calculate the loss using the output and go back to update the weights, by multiplying gradients. As can be seen in Figure 15, Let's Assume each state has a gradient of 0.01 and we have 100 states, therefore we have to go back to each state and update the weights. To update the 1st state, the gradient will be $(0.01)^{100} \approx 0$. Therefore, the update in weights will be negligible, and thus the neural network wouldn't learn at all. And therefore, this problem is called vanishing gradient problem, which is addressed by LSTM.

**Backpropagation through time is a training algorithm used to update weights in recurrent neural networks like LSTMs. In order to do this, model completes the forward propagation to get the output, checks if the output is correct or not, to get the error, and then model goes back to find the partial derivatives of the error with respect to the weights, which enables it to subtract this value from the weights. Those derivates are then used by gradient decent algorithm to adjust the weights up or down, to minimize the error. This done over several iterations minimize a given function.*

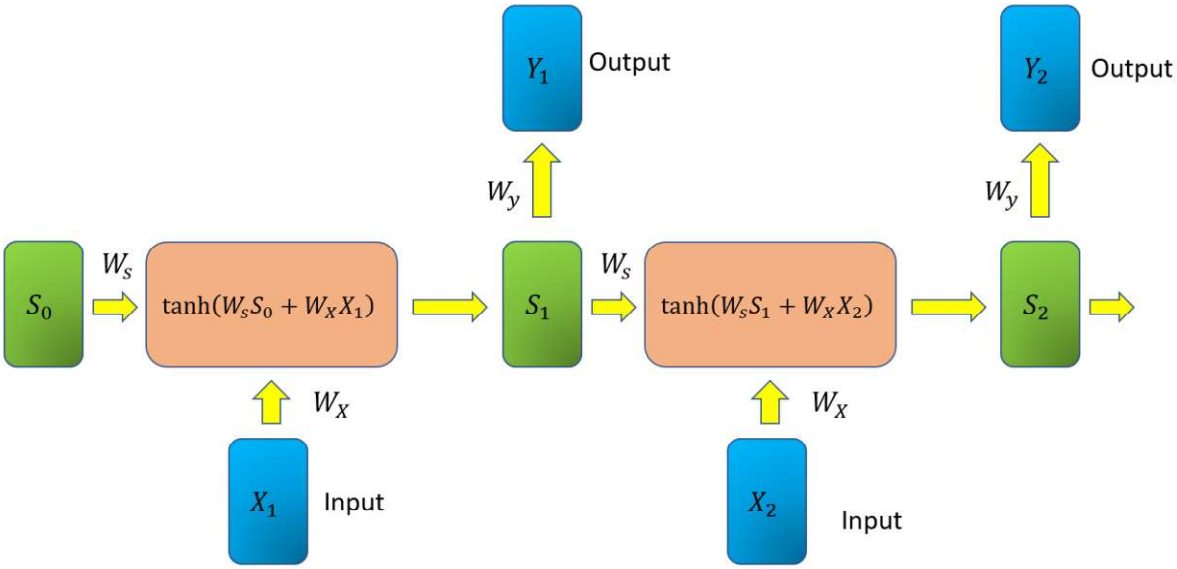



*Figure 13 - A Unrolled RNN*

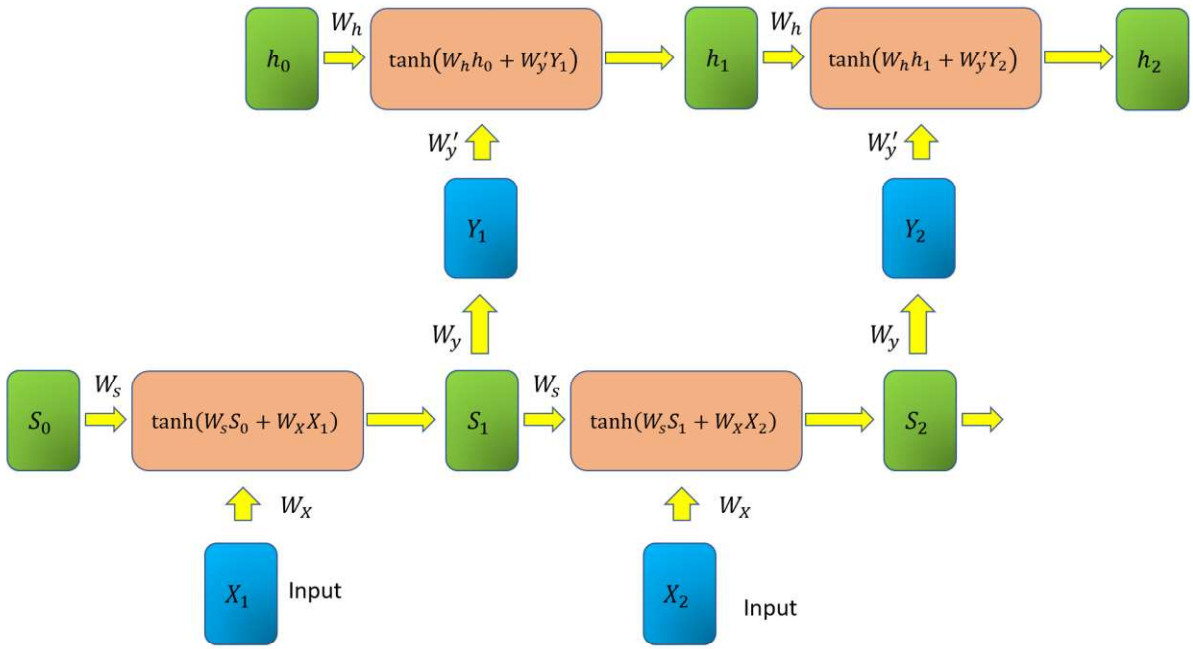



*Figure 14 - Multilayer RNN*

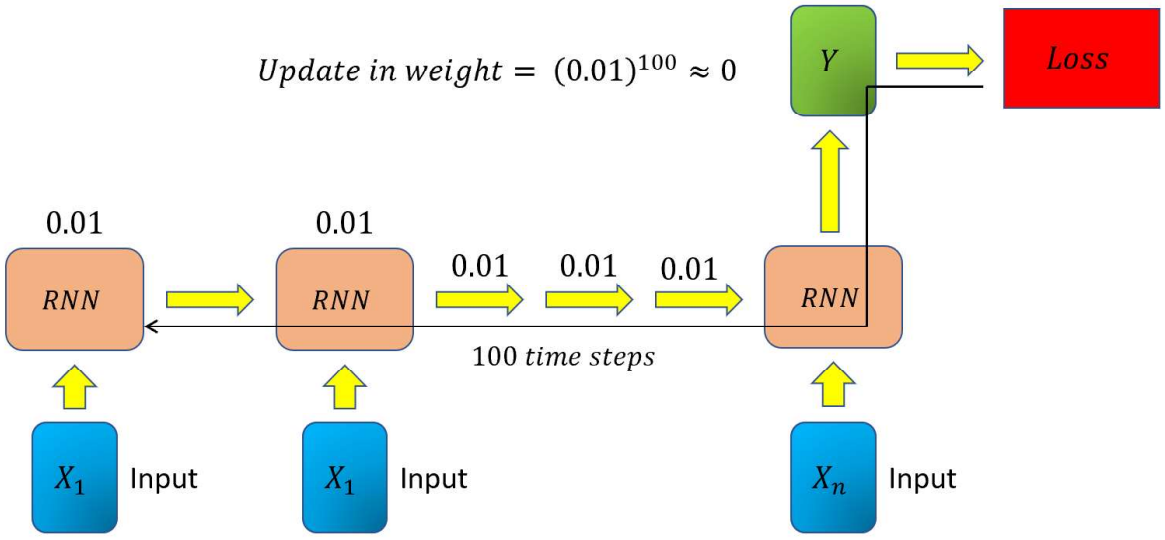



*Figure 15 - A visual representation of vanishing gradient problem in RNN*

As stated earlier, LSTM networks are an extension of RNN's, which basically extend the memory. LSTM's enable RNN's to remember inputs over a long period of time. They contain information in a memory, which is quite similar to the memory of a computer from which LSTM's can read, write or delete information.

This memory can be visualized as a gated cell, as the cell decides whether or not to store or delete information (i.e., if it opens the gate or not), based on the importance it assigns to the information. Importance is assigned through weights, which are learned by the algorithm. That means, the model learns by itself which information is important, and which isn't.

In an LSTM, you have three gates: input, forget or output gate. These gates determine whether or not to let new input in (input gate), delete the information because it is not important (forget gate), or let it impact the output at the current timestep (output gate). Figure 16 is an illustration of an RNN with its three gates.

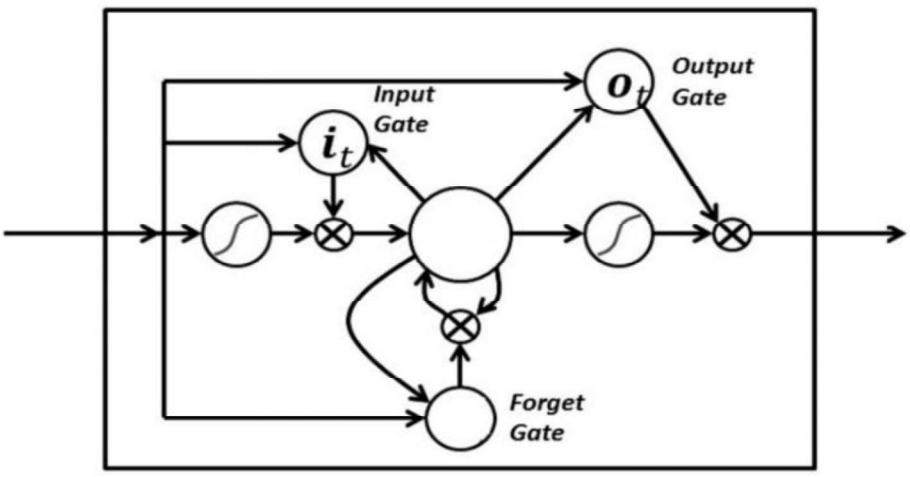



*Figure 16 - Schematic Diagram for a LSTM Unit cell*

The gates in an LSTM network are analog in the form of sigmoid, therefore they range from zero to one, instead just zero as one if it was digital. This enables them to arrange the information in the order of importance and enables it to perform much efficient *backpropagation through time*.

In the following example, we can see how LSTM solves the problem of vanishing gradient. As stated before, LSTM comprises of three gates and one cell state, and these are additional interaction to an RNN. Mathematical formulation of all the gates have been given below. In all the gates, previous state $S_{t-1}$ and $X_t$ are takes as input and are multiplied with respective weights i.e., $W_f, W_i, or\ W_o$ and then passed through a sigmoid activation function. One of the important things to note here is each gate has a different set of weights. Moreover, there are two different weights in one gate itself, one is to multiply with previous cell state and another for the input $X_t$. But both are represented as one weight to reduce the level of complexity, in visualization. $\widetilde{C_t}$ is an intermediate cell state which can also be calculated just like these gates but with its own set of weights and then by passing through $tanh$ activation function. And after that cell state is calculated by multiplying input gate with intermediate cell state and adding it to the product of previous cell state and forget gate. And then we pass the cell state through the $tanh$ activation and multiply it with the output gate.

$$\begin{aligned}
f_t &= \sigma\left(W_f S_{t-1} + W_f X_t\right) && Forget\ gate \\
i_t &= \sigma(W_i S_{t-1} + W_i X_t) && Input\ gate \\
o_t &= \sigma(W_o S_{t-1} + W_o X_t) && Output\ gate \\
\widetilde{C_t} &= tanh(W_c S_{t-1} + W_c X_t) && intermediate\ cell\ state \\
c_t &= \left(i_t \times \widetilde{C_t}\right) + (f_t \times c_{t-1}) && Cell\ State \\
S_t &= o_t \times tanh(c_t) && New\ State
\end{aligned} \qquad (3)$$

In the Figure 17*,* it can be understood in a much better way. Here, we have our old state $S_0$, the input $X_1$, and our previous cell state which is $C_0$**.** First, calculate the input gate by passing the previous state and input through sigmoid activation. Then, calculate our intermediate cell state by passing input and previous state through $tanh$ activation. After that multiply the input gate to intermediate cell state and then similarly, calculate the forget gate and multiply it with the previous cell state $C_0$. Then, add both of these products to obtain a new cell state $C_1$**.** This gives the output gate and then it is multiplied with the new cell state $C_1$ passed through $tanh$ activation. It gives us the new state $S_1$**.** Finally, this new cell state $C_1$ and the new state $S_1$ are passed over to the next time step so it can be used for further calculation. By following these steps LSTM solves the problem of vanishing gradient and works better than RNN, in terms of accuracy.

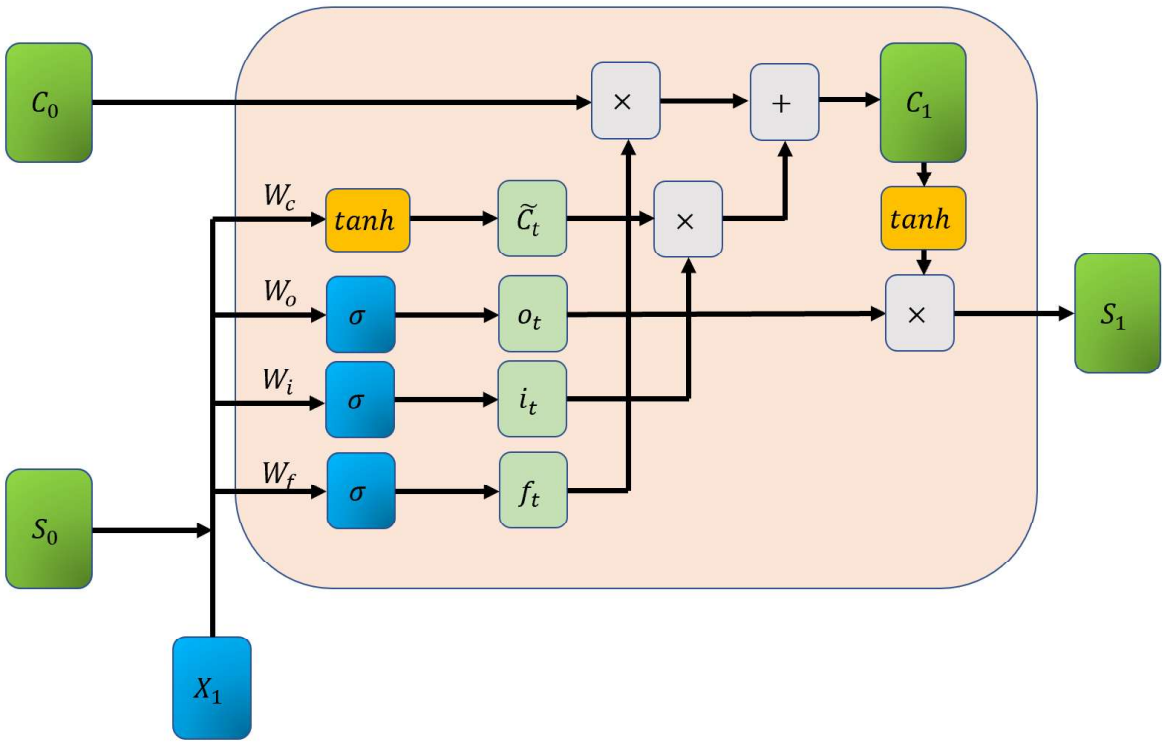



*Figure 17 - A visual representation of the working of LSTM.*

## Backpropagation through time (BPTT) in RNNs

After the output is generated in an RNN, we compute the prediction error and use the backpropagation through time algorithm to compute the gradient, which is change in prediction error with respect to the change in weights of the network (4). Gradients in all the time steps are added to find the final gradient and this gradient is used to update the model parameters. This learning process continues and is called gradient decent algorithm.

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_T}{\partial W} \tag{4}$$

$$W \rightarrow W - \alpha \frac{\partial E}{\partial W}$$

Where, $E$ is the total error,

$Where, E\ is\ the\ total\ error, E_T$ is the error in a single time step, $W$ is the weight and $\alpha$ is the coefficient to determine the change in weight.

Now, let's say we have a learning task that includes T time steps, then the gradient of the error on the k[th] time step is given by:

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial h_k} \frac{\partial h_k}{\partial s_k} \cdots \cdots \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial W}$$

$$= \frac{\partial E_k}{\partial h_k}\frac{\partial h_k}{\partial s_k}\left(\prod_{t=2}^{k}\frac{\partial s_t}{\partial s_{t-1}}\right)\frac{\partial s_1}{\partial W} \quad (5)$$

Now, $s_t = tanh(W_s s_{t-1} + W_X X_t)$,

So,

$$\frac{\partial s_t}{\partial s_{t-1}} = tanh'(W_s s_{t-1} + W_X X_t).\frac{\partial}{\partial s_{t-1}}(W_s s_{t-1} + W_X X_t)$$

$$= tanh'(W_s s_{t-1} + W_X X_t).W_s \quad (6)$$

Plug 6 into 5,

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial h_k}\frac{\partial h_k}{\partial s_k}\left(\prod_{t=2}^{k} tanh'(W_s s_{t-1} + W_X X_t).W_s\right)\frac{\partial s_1}{\partial W}$$

The last expression tends to vanish when k is large, this is due to the derivative of the tanh activation function which is smaller than 1.

So, we have,

$$\prod_{t=2}^{k} tanh'(W_s s_{t-1} + W_X X_t).W_s \to 0$$

So, for some time step k:

$$\frac{\partial E_k}{\partial W} \to 0$$

Therefore, the whole error gradient will vanish.

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T}\frac{\partial E_T}{\partial W} \to 0$$

The network's weights update will be:

$$W \to W - \alpha\frac{\partial E}{\partial W} \approx W$$

In addition, no significant learning will be done in reasonable time.

## Backpropagation through time (BPTT) in LSTMs

As in RNNs, the error term gradient is given by the following sum of T gradients (4). For the complete error gradient to vanish, all these T sub gradients need to vanish. If we think of it as a series of functions, then by definition, this series converges to zero if the sequence of its partial sums tends to zero. So, if we want the gradient not to vanish, our network needs to increase the likelihood that at least some of these gradients will not vanish.

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_T}{\partial W}$$

In LSTMs too, the gradient of the error for some time step k has a very similar form to the one in RNN:

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial h_k}\frac{\partial h_k}{\partial c_k} \dots \cdot \frac{\partial c_2}{\partial c_1}\frac{\partial c_1}{\partial W} = \frac{\partial E_k}{\partial h_k}\frac{\partial h_k}{\partial c_k}\left(\prod_{t=2}^{k}\frac{\partial c_t}{\partial c_{t-1}}\right)\frac{\partial c_1}{\partial W} \tag{7}$$

As we have seen $\prod_{t=2}^{k}\frac{\partial c_t}{\partial c_{t-1}}$, causes the gradients to vanish.

In LSTM, cell state is represented as,

$$c_t = (i_t \times C_t') + (f_t \times c_{t-1})$$

And therefore,

$$\begin{aligned}\frac{\partial c_t}{\partial c_{t-1}} &= \frac{\partial}{\partial c_{t-1}}\left[\left(i_t \times \widetilde{C_t}\right) + (f_t \times c_{t-1})\right] \\ &= \frac{\partial}{\partial c_{t-1}}\left(i_t \times \widetilde{C_t}\right) + \frac{\partial}{\partial c_{t-1}}(f_t \times c_{t-1}) \\ &= \frac{\partial i_t}{\partial c_{t-1}}.\widetilde{C_t} + \frac{\partial \widetilde{C_t}}{\partial c_{t-1}}.i_t + \frac{\partial f_t}{\partial c_{t-1}}.c_{t-1} + \frac{\partial c_{t-1}}{\partial c_{t-1}}.f_t \end{aligned} \tag{8}$$

We can denote the four elements comprising the derivative of the cell state by:

$$\begin{aligned}A_t = \frac{\partial i_t}{\partial c_{t-1}}.\widetilde{C_t} &= \frac{\partial}{\partial c_{t-1}}[\sigma(W_i[S_{t-1} + X_t])].\widetilde{C_t} \\ &= \sigma'(W_i[S_{t-1} + X_t]).W_i.\frac{\partial s_t}{\partial c_{t-1}}.\widetilde{C_t} \\ &= \sigma'(W_i[S_{t-1} + X_t]).W_i.o_{t-1}.tanh'(c_{t-1}).\widetilde{C_t}\end{aligned}$$

$$\begin{aligned}B_t = \frac{\partial \widetilde{C_t}}{\partial c_{t-1}}.i_t &= \frac{\partial}{\partial c_{t-1}}[\sigma(W_c[S_{t-1} + X_t])].i_t \\ &= \sigma'(W_c[S_{t-1} + X_t]).W_c.\frac{\partial s_t}{\partial c_{t-1}}.i_t \\ &= \sigma'(W_c[S_{t-1} + X_t]).W_c.o_{t-1}.tanh'(c_{t-1}).i_t\end{aligned}$$

$$\begin{aligned}C_t = \frac{\partial f_t}{\partial c_{t-1}}.c_{t-1} &= \frac{\partial}{\partial c_{t-1}}\left[\sigma\left(W_f[S_{t-1} + X_t]\right)\right].c_{t-1} \\ &= \sigma'\left(W_f[S_{t-1} + X_t]\right).W_f.\frac{\partial s_t}{\partial c_{t-1}}.c_{t-1}\end{aligned}$$

$$= \sigma'\big(W_f[S_{t-1} + X_t]\big).W_f.o_{t-1}.tanh'(c_{t-1}).c_{t-1}$$

$$D_t = \frac{\partial c_{t-1}}{\partial c_{t-1}}.f_t = f_t$$

We write the additive gradient (8) as:

$$\frac{\partial c_t}{\partial c_{t-1}} = A_t + B_t + C_t + D_t$$

Plug the value of $\frac{\partial c_t}{\partial c_{t-1}}$ into the original equation

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial h_k}\frac{\partial h_k}{\partial c_k}\left(\prod_{t=2}^{k}[A_t + B_t + C_t + D_t]\right)\frac{\partial c_1}{\partial W} \tag{9}$$

The presence of forget gate's activation allows the LSTM to decide, at each time step, that certain information should not be forgotten and to update the model's parameters accordingly. This allows the network to better control the gradients values.

Let's go over how this property helps us. Say that for some time step k<T, and we have a situation as follows,

$$\sum_{t=1}^{k}\frac{\partial E_T}{\partial W} \approx 0$$

Then, for the gradient not to vanish, model finds a suitable parameter update of the forget gate at time step k+1 such that,

$$\sum_{t=1}^{k+1}\frac{\partial E_T}{\partial W} \not\approx 0$$

It is the presence of the forget gate's vector of activations in the gradient term along with additive structure which allows the LSTM to find such a parameter update at any time step, such that the overall gradients don't vanish.

## 2.2. Physics-Informed Neural Network (PINN)

Physics Informed neural networks are quite unique and different than other Neural Networks. This technique provides a solution to the differential equations using Neural Networks. Due to a large amount of differential equations in engineering and science, this tool becomes very useful, in order to automatize these fields. One of the reasons of this being so unique is that, there is no training, testing or validation set.

In this technique, we are essentially posing every ODE/PDE and converting into an optimization problem and trying to automatize the whole process by using Neural Networks instead of Finite difference methods. So here, Neural Network can solve as well as learn from the solution and hence, it is a step forward towards full automation for solving differential

equations using Neural Networks. We can understand this properly by a simple example. So, let's say we have a function $u$ differentiable in $x$ and has a simple differential equation (10).

$$\frac{\partial^2 u}{\partial x^2} + a\frac{\partial u}{\partial x} = b \tag{10}$$

with boundary conditions as: $u(0) = u_o, u(1) = u_1, where\ x \in (0,1)$

To solve the above equation using Neural Networks, we deploy a single hidden layer Neural Network, which takes x as an input and gives u as output *(*Figure 18*)*.

$$u = NN(x)$$

As Universal approximation theorem suggests, we can always approximate the solution of u arbitrarily closely by a neural network. Hence, Neural Networks are quite excellent function approximators.

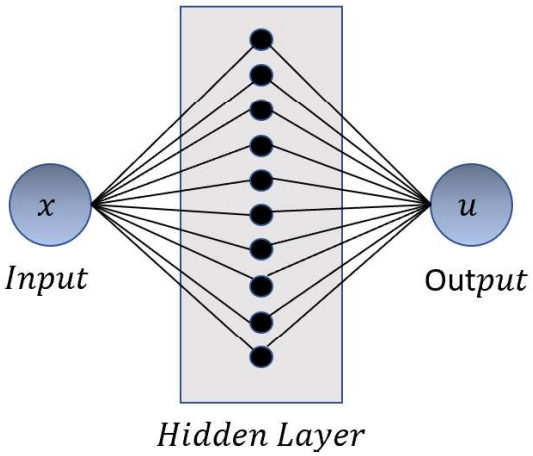



*Figure 18 – A typical neural network with single hidden layer consisting of 10 neurons with one input and one output.*

Now, to understand how it helps us, let's assume, a very simple neural network. As can be seen in Figure 19, It just have one input $x$, one hidden neuron $a_1$, activated by a sigmoid function (σ) and the output is a linear layer $u$.

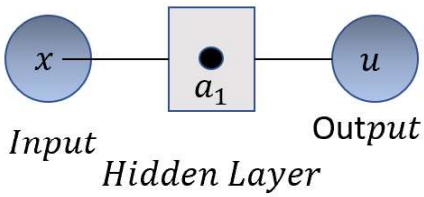



*Figure 19 – A Neural network with one hidden layer made of 1 neuron.*

So, we can write the following

$a_1 = \sigma(w_1 x)$

$u = w_2\sigma(w_1 x)$

$$\frac{du}{dx} = w_2\sigma'(w_1 x)w_1 \tag{11}$$

Where, $w_2$ is the weight in the neural network.

Similarly, we can calculate $\frac{d^2u}{dx^2}, \ldots etc.$

That means all derivates of u with respect to input x can be found. But it can be said that its only possible because, here we have just one single neuron in one single layer. But if we have multiple neurons or multiple hidden layers with multiple neurons, we can use autograd or automatic differentiation. The idea is similar to Backpropagation, we can always find out the difference of output using the difference of input, same as in backpropagation, and we use difference of loss function to the difference in weights. This automatic differentiation is present in TensorFlow package. Now using this, we can find out all the differential terms in the equation. Now, we can pose the whole problem as optimization problem, as shown in equation (12).

$$f = \frac{\partial^2 u}{\partial x^2} + a\frac{\partial u}{\partial x} - b = 0 \tag{12}$$

Now, since we can't make it exactly zero, as neural network can't give the exact solution but approximate it. Therefore, we can write it as follows

$$f = minimize\left(\left[\frac{\partial^2 u}{\partial x^2} + a\frac{\partial u}{\partial x} - b\right]^2\right) \tag{13}$$

Now, this is the cost function and we can minimize it using gradient descent. But we also need to accommodate the boundary conditions. We can do it buy adding that also to the cost function (14).

$$f = minimize\left(\left[\frac{\partial^2 u}{\partial x^2} + a\frac{\partial u}{\partial x} - b\right]^2 + [u_0' - u_0]^2 + [u_1' - u_1]^2\right) \tag{14}$$

We can see, this looks like an extremely clever way of posing the problem. The whole differential equation and all the boundary conditions together are now just an optimization problem.

So, while solving it, algorithm tries various values of x, between 0 to 1. Calculate the differential terms and tries to minimize the above-mentioned loss function. So, we can see, in reality there is no training or testing set as in all the conventional Machine Learning or Neural Network problems.

In Figure 20, $x$ and $t$ serves as inputs to the neural network, which figures out $u$. Now, Automatic differentiation is used to calculate all the differential terms in the differential equation. This can be channelled to the loss function and can be minimized using backpropagation.

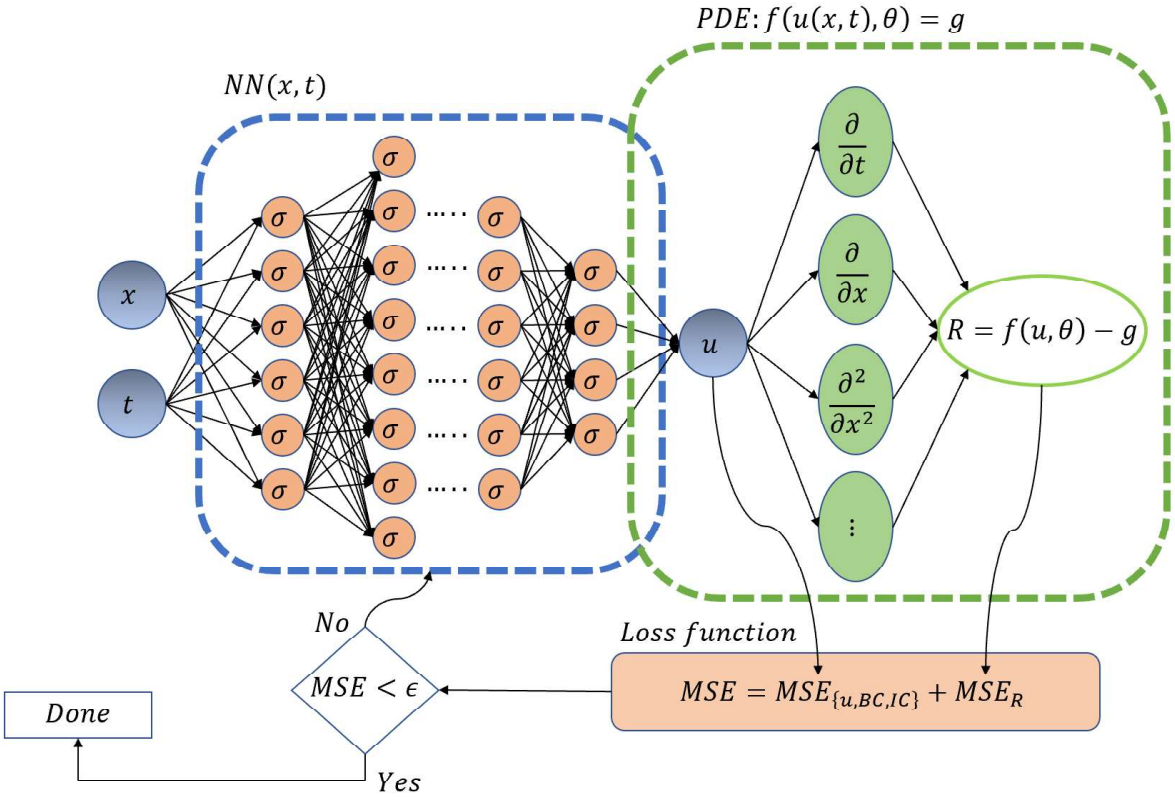



*Figure 20 - Schematic diagram to explain Physics informed Neural Network (PINN)*

# Chapter 3
# Theorical Background of Infiltration and Data Generation

Infiltration process in an unsaturated soil is essentially a two-phase flow of two immiscible fluids – air and water. The process of infiltration of surface water through the upper layers of soil, enriches the soil moisture, and subsurface flow through soils, that are partially filled with air. The understanding of this infiltration process is important for geotechnical engineers because due to infiltration, unsaturated soil is transformed to saturated soil which is unstable due to reduced effective stress and the suction forces in soil. Mathematically, the flow of water in a variably saturated or unsaturated soil is described by Richard's Equation.

## 3.1 Richard's Equation

Richard's equation can be obtained by combining continuity equation with Darcy's Law. Continuity equation in an unsaturated porous media having flow in one direction can be written as given below.

$$\frac{\partial \theta}{\partial t} + \frac{\partial q}{\partial z} = 0$$

Where, $\theta$ is water content, $q$ is the rate of flow, $t$ is the time, and $z$ is the depth.

Darcy's law states that

$$q = -Ki$$

Where, $K$ is hydraulic conductivity, $i$ is the hydraulic gradient $\frac{\partial H}{\partial z}$, and $H$ is the hydraulic head.

The above Darcy's law is for one dimensional saturated flow. For unsaturated flow Hydraulic head can be split in Suction Head $(\psi)$ and gravity head $(z)$. Therefore, we get

$$q = -K\frac{\partial}{\partial z}(\psi + z)$$

In addition, for unsaturated flow hydraulic conductivity $(K)$ is a function of both $\psi$ and $\theta$. Therefore, $\theta$ and $\psi$ are intrinsically related as follows

$$\frac{\partial \psi}{\partial z} = \frac{\partial \psi}{\partial \theta}\frac{\partial \theta}{\partial z}$$

Where, $\frac{\partial \theta}{\partial z}$ is the gradient of water content in vertical direction and $\frac{\partial \psi}{\partial \theta}$ is the specific water capacity or water storage constant.

Hence,

$$q = -K\left(\frac{\partial \psi}{\partial z} + \frac{\partial z}{\partial z}\right)$$

$$= -K\left(\frac{\partial \psi}{\partial \theta}\frac{\partial \theta}{\partial z} + 1\right) = -K\left(\frac{\partial \psi}{\partial \theta}\frac{\partial \theta}{\partial z}\right) - K$$

Defining, $D = K\frac{\partial \psi}{\partial \theta}$

And $D\ is\ soil - water\ diffusivity$

Therefore, we get

$$q = -\left[D\frac{\partial \theta}{\partial z} + K\right]$$

From continuity equation,

$$\frac{\partial \theta}{\partial t} = -\frac{\partial q}{\partial z}$$

Therefore,

$$\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial z}\left[D\frac{\partial \theta}{\partial z} + K\right] \quad (15)$$

This is the Richards equation which is used to describe one dimensional flow in an unsaturated media. It can also be expressed in terms of pressure head (16) *[10]*.

$$\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial z}\left[K(\psi)\frac{\partial \psi}{\partial z} + K(\psi)\right] \quad (16)$$

## 3.2 Soil Water characteristic curve (SWCC)

A soil-water characteristic curve (SWCC) describes the amount of water retained in a soil under the equilibrium at a given matric potential. This water retained can be expressed in terms of mass or volume of water content, $(\theta_m)$ or $(\theta_v)$. A SWCC plays a very important role in understanding the hydraulic properties, which are related to size and connectedness of pore spaces. Hence, SWCC is strongly affected by soil structure and texture, and other constituents like organic matter etc. Modelling water distribution and flow in unsaturated soils requires an understanding of SWCC, therefore it holds great importance in water management, and solute and contaminant transport in the environment. Generally, SWCC is highly non-linear and is quite difficult to obtain accurately. Because matric potential extends over several orders of magnitude for the range of water contents commonly encountered in practical applications. It is often plotted on a logarithmic scale. Figure 21 shows a general SWCC for sand, silt loam

and clay, and it shows very clearly that there is a drop in matric potential with the increasing particle size of the soil grains, i.e., decreasing capillary and adhesive forces.

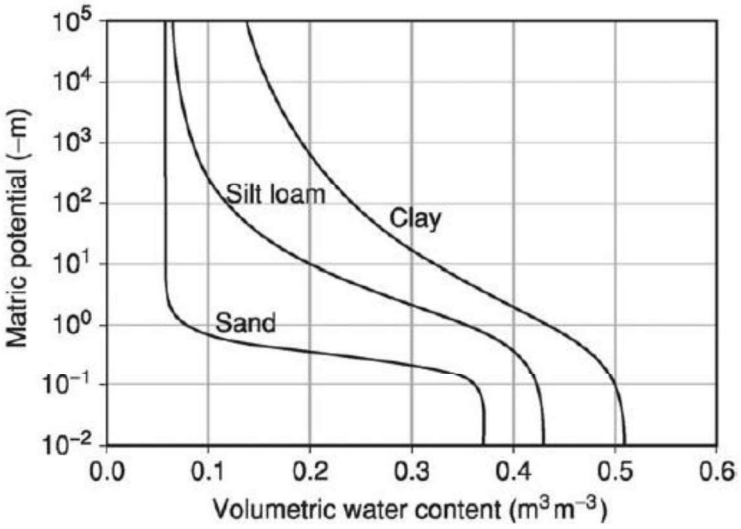



*Figure 21 - Typical soil-water characteristic curve for soils of different texture* [9].

## 3.3 Soil Matric Potential or Pressure head $(\psi)$

Matric potential is related to capillary and adsorptive forces acting between the three phases i.e., solid, liquid and gas *[10]*. Capillary forces are generated due to the surface tension of water making an angle of contact or the contact angle with the solid particles. It means that in the non-wetting air phase, curved liquid-vapor interfaces (menisci) are formed within the porous soil system. However, in addition to capillary forces soil also exhibit some adsorption forces. In this process of adsorption soil particle is enveloped by a thin layer of water. In clayey soil it is an important process, as clay has a smaller particle size, hence more surface area. In sandy soil, adsorption is quite insignificant due to less surface area, and hence capillary effect dominates. In general, however, matric potential is a combined effect of capillarity and surface adsorption, and hence two cannot be considered separately.

## 3.4 Modelling SWCC

Measuring soil water characteristics is a very laborious and time-consuming task. $\theta - \psi$ pairs measured, are usually very fragmented and constitutes very few measurements over the wetness range of interest. Therefore, for modelling and analysis purposes, and for characterization and comparison between different soils and scenarios, it is quite common to represent SWCC in a mathematical continuous form. Several approaches, ranging from empirical parametric expressions to physically based models, with parameters derived from measurable medium properties can be employed to represent a continuous SWCC.

One of the most effective and widely used parametric model for relating water content to matric potential is called van Genuchten model *[11]* and is denoted as VG (17).

$$\theta = \frac{\theta - \theta_r}{\theta_s - \theta_r} = \left[\frac{1}{1 + (\alpha\psi)^n}\right]^m \quad (17)$$

Where $\theta_r$ and $\theta_s$ are the residual and saturated water content, respectively. $\psi$ is matric potential or pressure head, and $\alpha, n$ and $m$ are parameters directly dependent on the shape of $\theta(\psi)$ curve. A common simplification is to assume that $m = 1 - 1/n$. Thus, the parameters required for estimation of the model are $\theta_r, \theta_s, \alpha$ and $n$. $\theta_s$ is sometimes known and easy to measure leaving only the three unknown parameters $\theta_r, \alpha$ and $n$ to be estimated from the experimental data in many cases.

Following formulations from van Genuchten *[10], [12]* were used to calculate water content $(\theta)$, hydraulic conductivity $(K)$, water storage coefficient $(C)$, and effective water content $(S_e)$.

$$S_e = \left[\frac{1}{1 + (\alpha\psi)^n}\right]^m \quad (18)$$

$$K = K_s s_e^{0.5}\left(1 - \left(1 - s_e^{1/m}\right)^m\right)^2 \quad (19)$$

$$\theta = \theta_r + \frac{\theta_s - \theta_r}{[1 + (\alpha\psi)^n]^m} \quad (21)$$

Where, $K_s$ is the saturated hydraulic conductivity and $S_s$ is the specific storage coefficient.

## 3.5 Soil type description

Data presented in the Table 2 has been used in the Python code *vanGenuchten.py* to produce the values of Water Content $(\theta)$, Hydraulic Conductivity $(K)$, and Water Storage Coefficient $(C)$. Two standard soils have been used to do this analysis.

| | $\theta_r$ [$m^3.m^{-3}$] | $\theta_s$ [$m^3.m^{-3}$] | $\alpha$ [$m^{-1}$] | $n$ [-] | $K_s$ [m/day] | $S_s$ [-] |
|---|---|---|---|---|---|---|
| **Hygiene Sandstone** | 0.153 | 0.25 | 0.79 | 10.4 | 1.08 | 1E-06 |
| **SiltLoamGE3** | 0.131 | 0.396 | 0.423 | 2.06 | 0.0496 | 1E-06 |

*Table 2 - Shows the description of the soil type used in this study.*

- **Hygiene Sandstone** is a member of Pierre formation *[13]*. It is thick bedded and frequently cross-bedded. Much of it is dark greenish grey and gritty. The remainder is light grey. The whole is calcareous where fresh. It loses it's lime in weathering, takes a paler-greenish tint, and becomes friable. It frequently contains carbonaceous matter resembling small sticks of wood turned to coal. It also contains fossils of invertebrates, but its fauna is not yet known to be distinctive of this horizon. Figure 22 shows the properties variation in hygiene sandstone with the change in pressure head $\psi$.

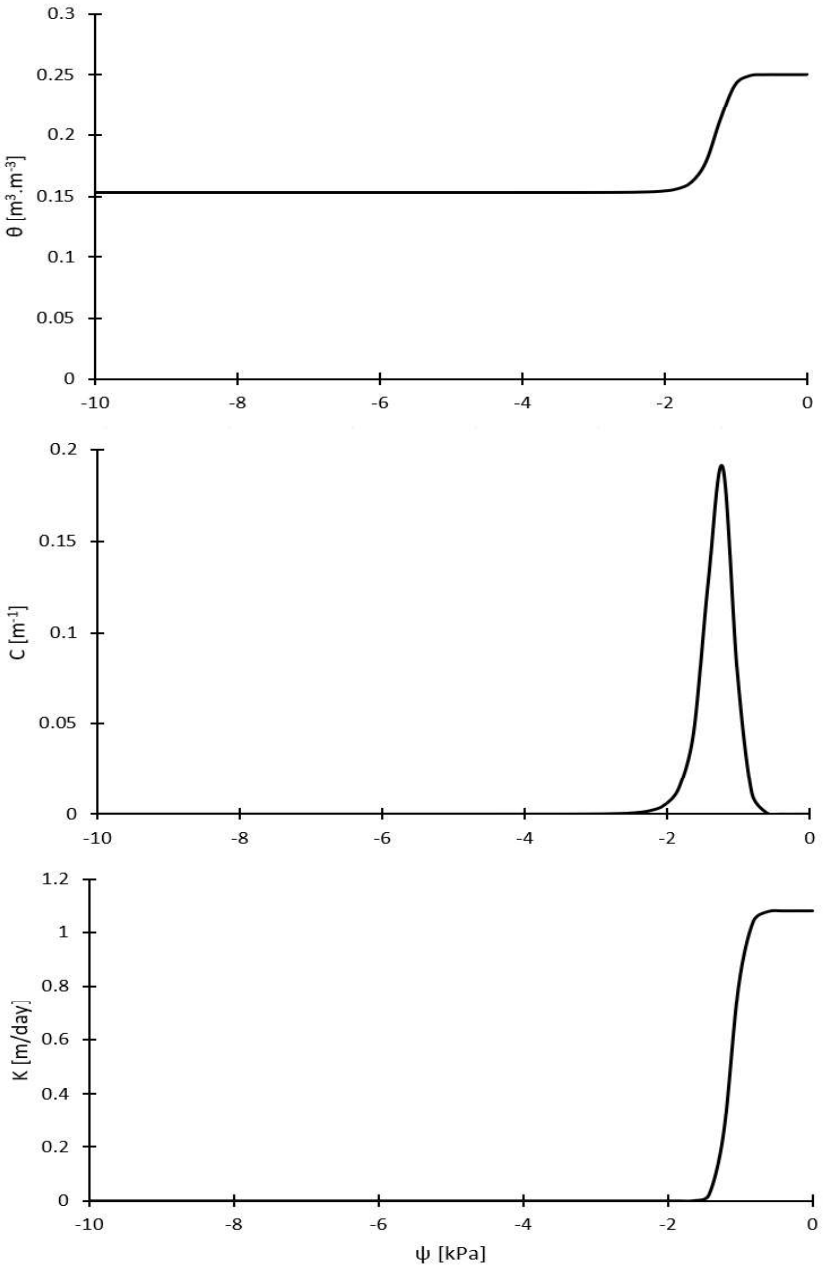



*Figure 22 - properties variation in hygiene sandstone with the change in pressure head ψ.*

- **SiltLoamGE3** belongs from Touchet series. It consists of deep, moderately well drained soils formed in recent alluvium on flood planes at elevations from 150 to 300 meters. It is typically found near Walla Walla River in Walla Walla County, Washington USA. It contains 10 to 18 percent of clay particles and have moderate permeability. Properties for this soil type is presented in the Figure 23 below.

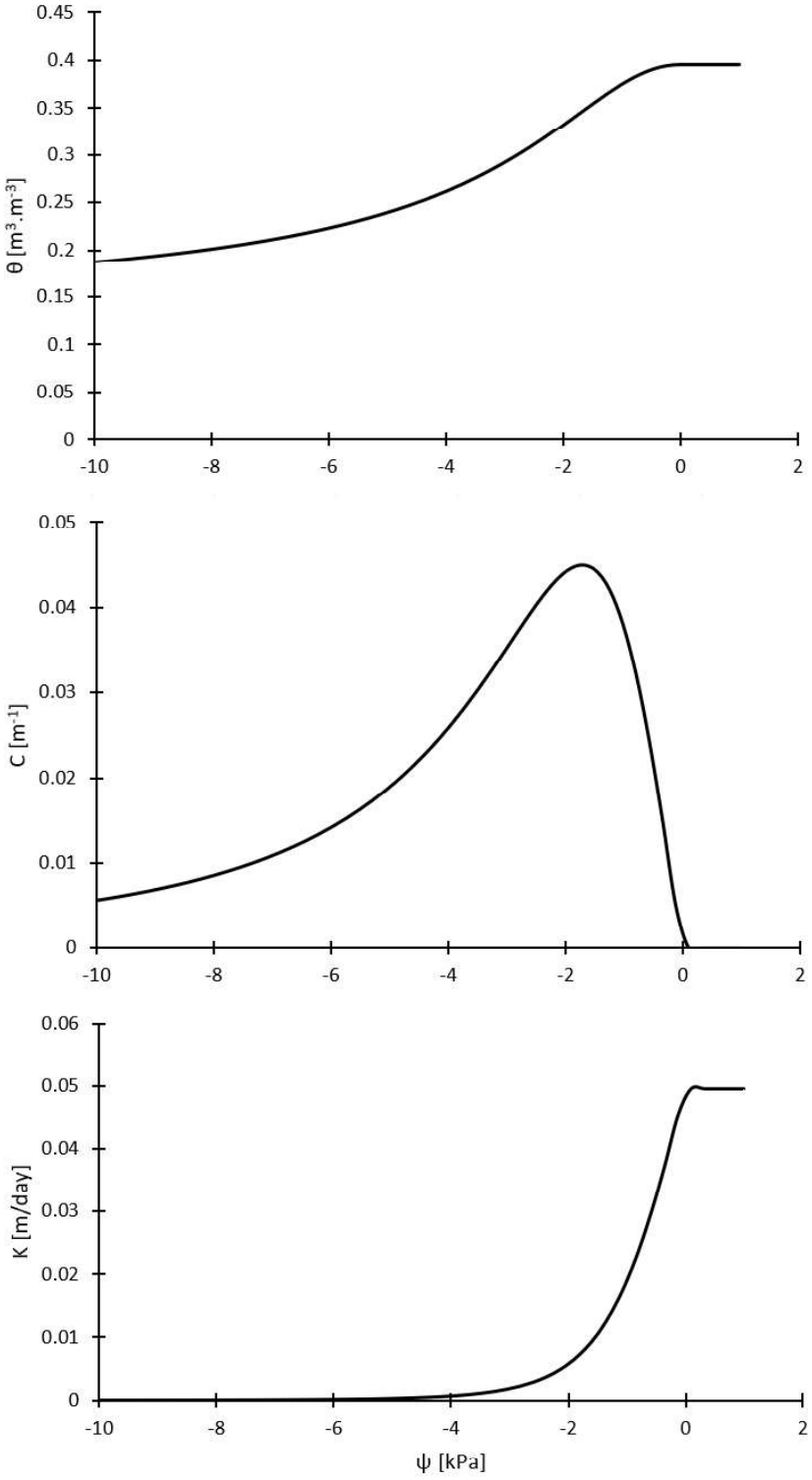



*Figure 23 - properties variation in SiltLoamGE3 with the change in pressure head ψ.*

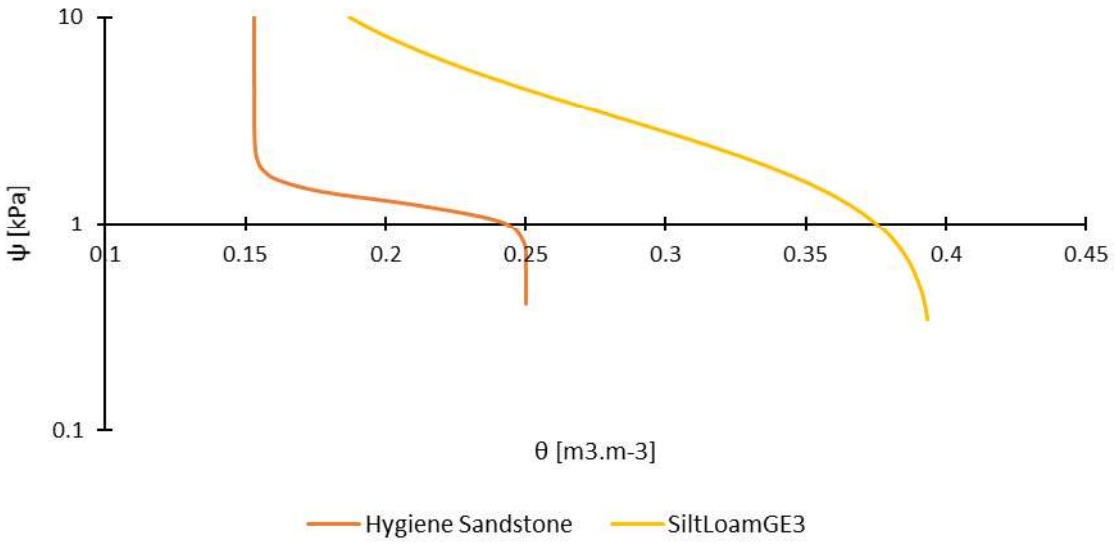



*Figure 24 - SWCC for Hygiene Sandstone and SiltLoamGE3*

## 3.6 Data Generation

As mentioned in chapter 2, Long Short-Term Memory (LSTM) networks, is a machine learning technique which is used to address time series problem or problems including sequential data. Therefore, to use this technique in this thesis, infiltration problem was modelled as a problem with sequential data and using a Python code *RichardsEquationGenerator* values of water content $(\theta)$, and pressure head $(\psi)$ were calculated at every 5 cm depth and 150 times a day for 10 days i.e., almost in every 10 minutes and was fed to the training algorithm. However, just to keep the figures below comprehensive, it was reduced to 10 times a day for 10 days. Moreover, it can be seen in the Figure 25, in the code snippet below, in line 148 infiltration flux can be changed. With line 149, 150 and 151 boundary conditions can be altered. Lines 154 and 155 are used to define the grid in space, while, line 160 defines the grid in time. For analysis purposes, two sets of data are created for each type of soil, one is with closed drainage and another with open drainage condition.

```
146 # This block of code sets up and runs the model
147 # Boundary conditions
148 qTop=-0.01 #infiltration flux
149 qBot=[]
150 psiTop=[]
151 psiBot=0 #Bottom pressure head, 0= closed drainage and empty '[]' means free drainage
152
153 # Grid in space
154 dz=0.05 #Datappoints at every 5 cm
155 ProfileDepth=5 # till a depth of 5 meters
156 z=np.arange(dz/2.0,ProfileDepth,dz)
157 n=z.size
158
159 # Grid in time
160 t = np.linspace(0,10,1500) #time period of 10 days of 1500 time steps i.e; approx 150 datapoints a day
161
162 # Initial conditions
163 psi0=-z #Hydrostatic initial conditions
164
165 # Solve
166 psi=odeint(RichardsModel,psi0,t,args=(dz,n,p,vg,qTop,qBot,psiTop,psiBot),mxstep=5000000);
167
168 print ("Model run successfully")
```

*Figure 25 - Shows the setup of the model in the Python code.*

Figure 26 shows the process of infiltration in **HygieneSandstone** with an influx of 0.01 m/day with closed drainage. In *Figure 26(b),* it can be observed, in the beginning the pressure distribution was hydrostatic, but as infiltration takes place it becomes constant to the depth, till the water reaches *i.e.,* around 3.5 meters. In *Figure 26(a),* discharge began to rise at around 60$^{th}$ observation, as soil approaches to its saturation value.

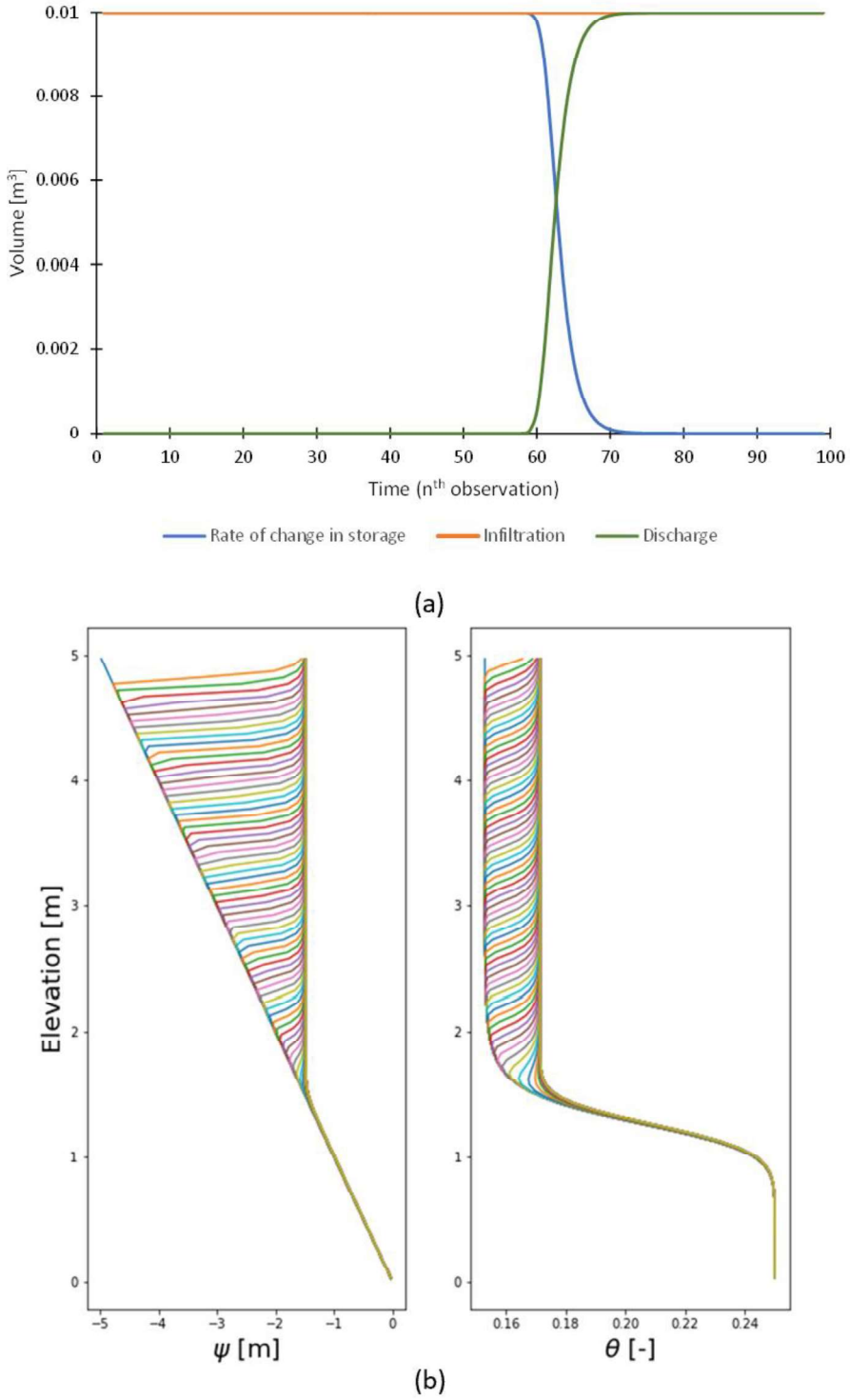



*Figure 26 - Shows the infiltration process of Hygiene Sandstone without drainage.*

Similar to Figure 26, Figure 27 also shows the process of infiltration in **Hygiene Sandstone** with an influx of 0.01 m/day but with open drainage. Therefore, this time In *Figure 27(b),* it can be observed, in the beginning the pressure distribution was hydrostatic, but as infiltration takes place it becomes constant to around -1.5 meters throughout the depth of the soil i.e., 5 meters. In *Figure 27(a),* we can observe that at the end, discharge becomes equal to the influx. It is because of the open drainage condition.

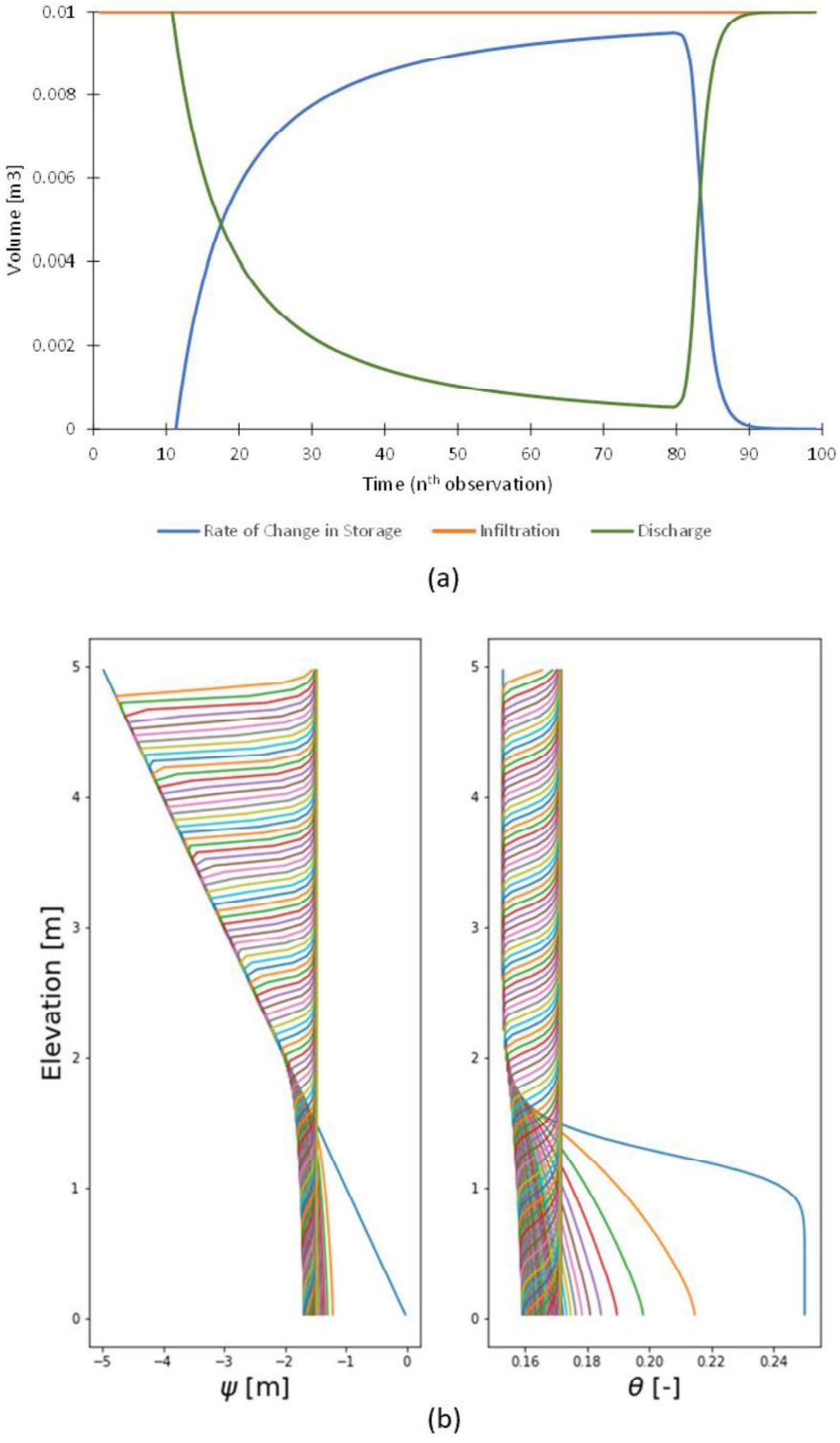



*Figure 27 - Shows the infiltration process of Hygiene Sandstone with open drainage.*

Figure 28 shows the process of infiltration in **SiltLoamGE3** with an influx of 0.03 m/day with closed drainage. In SiltLoamGE3 it was required to increase the influx as water penetration was not very significant with an influx of 0.01 m/day. Initial pressure distribution was hydrostatic in nature but, it can be observed in *Figure 26 (b),* that final pressure head is not constant as in previous case with Hygiene Sandstone. Moreover, in *Figure 28 (b)* it can be observed that, till around 85$^{th}$ time step, Rate of change of storage was equivalent to influx, and discharge was equal to zero. That means, there is accumulation of water in the soil with quite high build-up of pore water pressure. This can be a due to smaller particle size than that of the previous cases.

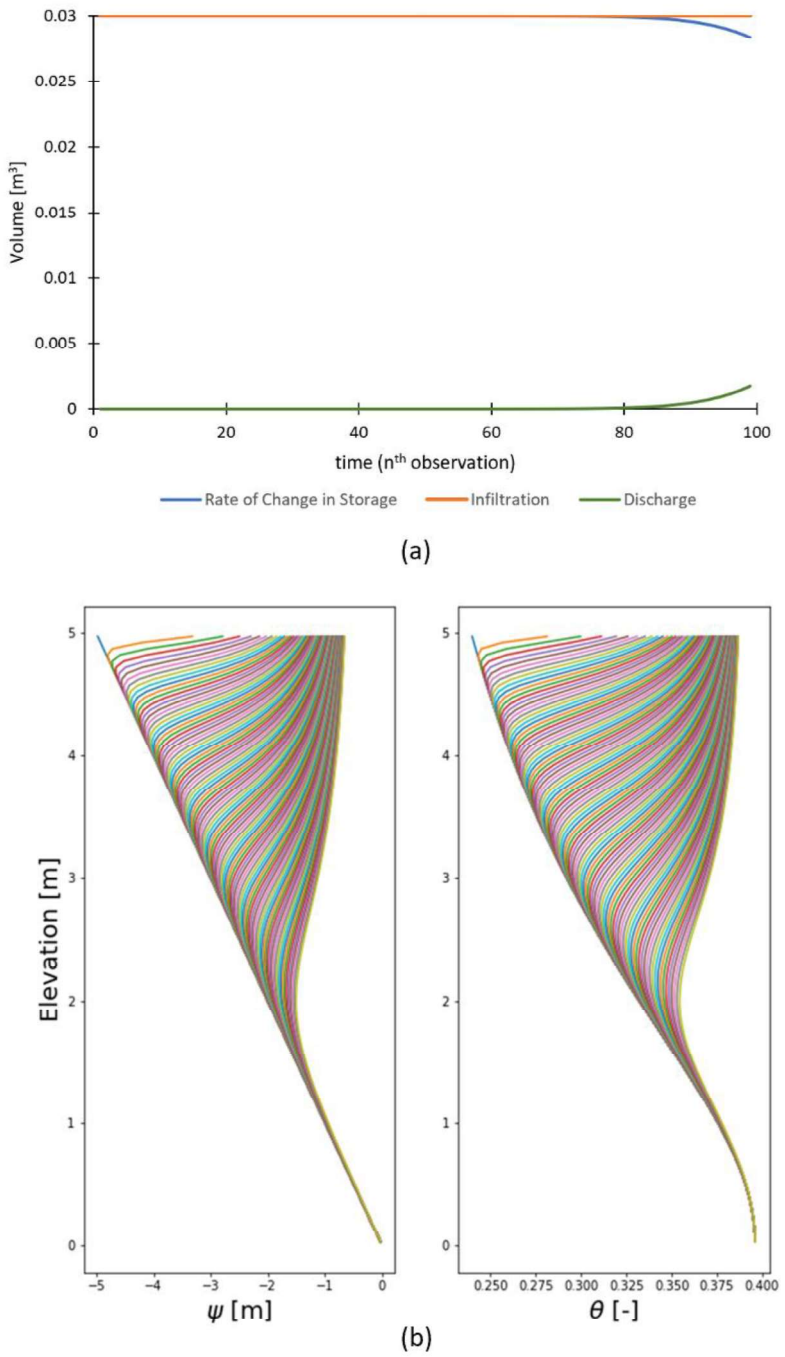



*Figure 28 - Shows the infiltration process of SiltLoamGE3 without drainage.*

Figure 29 shows the process of infiltration for **SiltLoamGE3** with an influx of 0.03 m/day with open drainage. As in closed drainage, In *Figure 28(b),* it can be observed, in the beginning the pressure distribution was hydrostatic, but as infiltration happens the final pressure head is not constant, it changes. In the top part, final pressure increases, while in the bottom part it decreases. The reason can be that as particle size decreases, adsorption forces start to dominate the matric potential or pressure head instead of capillary forces, therefore it becomes more unpredictable.

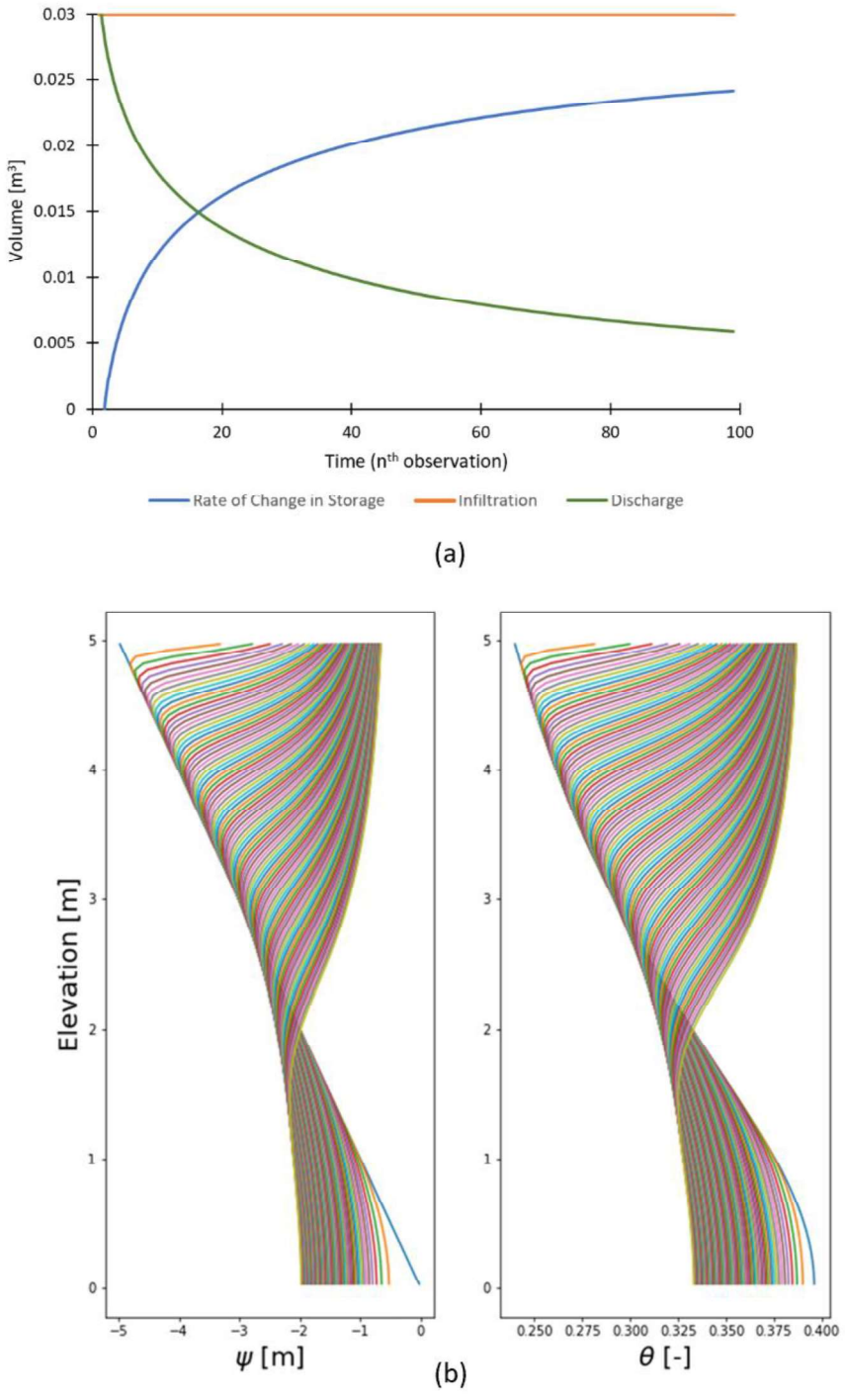



*Figure 29 - Shows the infiltration process of SiltLoamGE3 with open drainage.*

# Chapter 4
# Modelling with Python Code

This chapter in the thesis is dedicated to explaining the Python code used to apply Long Short-Term Memory (LSTM) Networks and Physics-Informed Neural Networks (PINNs) to the dataset to mimic the infiltration process. All the work has been done in Python 3.6, using Spyder from Anaconda. Anaconda is a free and open-source distribution, of the Python and R programming language for scientific computing. Spyder is the scientific Python Development Environment and it is a free Integrated Development Environment (IDE), that is included in Anaconda.

In Data Science, while doing Machine Learning, a lot of libraries and packages are commonly used. Those used in the thesis are as follows.

- **TensorFlow:** It is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library and is also used for machine learning applications such as neural networks.
- **Keras:** It is an open-source neural network library written in Python. It is capable of running on top of various libraries like TensorFlow, Microsoft Cognitive, Toolkit, R, Theano or PlaidML. It is designed to enable fast experimentation with deep neural networks, and it focuses on being user-friendly, modular and extensible.
- **NumPy:** It is a fundamental package for scientific computing with Python. This library adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- **Pandas:** It is a software library written for Python. It is used for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.
- **SciPy:** It is a free and open-source Python library used for scientific and technical computing. It contains modules for optimization, linear algebra, integration, special functions, signal and image processing. It builds on NumPy array object and is part of NumPy stack which includes tools like Matplotlib, pandas and SymPy and an expanding set of scientific computing libraries. The whole NumPy stack has similar users to MATLAB, GNU OCTAVE, and Scilab.
- **Matplotlib:** Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy.

## 4.1 Long Short-Term Memory (LSTM) or Time series prediction

Python code for LSTM was majorly divided in three parts as follows:

- Part 1: Data Pre-processing
- Part 2: Building the LSTM model
- Part 3: Making prediction and plotting

In Part 1: Data Pre-processing *(*Figure 30*)*, NumPy, pandas and Matplotlib libraries were imported. Using pandas, training set was imported and stored in a variable dataset_train. Here training set includes the value of *water content* $(\theta)$, at depths of 0.5 m, 1 m, 1.5 m, 2 m, 2.5 m, 3 m, 3.5 m, 4 m, 4.5 m, 5 m over a period of 10 days. After that using feature scaling all the data is scaled between 0 to 1 for more accurate predictions. Then the data is arranged in timesteps. To understand this, Let's assume there is following series called y.

$$y = \{x_1, x_2, x_3 \dots \dots . . x_m\}$$

The data in this series is arranged in n timesteps and the whole dataset has m observation, where $(n < m)$. So, training set that will be fed to LSTM unit will be $\{x_1, x_2, x_3 \dots \dots . . x_n\}$, and it will try to predict $x_{n+1}$, then the next training set will be $\{x_2, x_3, x_4 \dots \dots . . x_{n+1}\}$ and it will predict $x_{n+2}$.

```
# Part 1 - Data Preprocessing

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the training set
dataset = pd.read_csv('HS_CD_theta_cyclic.csv')

#Splitting the dataset for this run
training_set = dataset.iloc[:, :].values

# Feature Scaling
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (0, 1))
training_set_scaled = sc.fit_transform(training_set)

# Creating a data structure for training LSTM
X = []
y = []

mem=100
train_len=3000

for i in range(mem, 4499):
    X.append(training_set_scaled[i-mem:i, :])
    y.append(training_set_scaled[i, :])

X=np.array(X); y=np.array(y)
X_train, y_train = np.array(X[0:train_len,:,:]), np.array(y[0:train_len,:])
X_test, y_test = np.array(X[train_len:,:,:]), np.array(y[train_len:,:])
```

*Figure 30 - Part 1: Data pre-processing*

In Part 2: Building the LSTM model *(Figure 31)*, some modules of Keras are imported. After that, model is initialized, input layer has been defined in line 53. Then, several hidden layers are defined a hidden layer is defined at line 78. Number of Neurons are introduced in every layer, number of hidden layers and number of neurons in each layer can be changed to obtain good results. Moreover, in line 81, the model is compiled using adam optimizer and a loss function. Whereas, Adam optimizer is an optimizer that implements adam algorithm. It is stochastic gradient descent method that is based on adaptive estimation of first and second order moments. It is computationally efficient, occupies little memory, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters [15]. In Figure 31, mean_square_error is used as a loss function, but other loss functions can also be used for example mean_absolute_error. At the end in line 84, number of epochs and batch_size is defined, that can also alter to improve the model performance. Moreover, batch size is a number of samples processed before the model is updated. While the number of epochs is the number of complete passes through the training dataset, the batch size should be more than or equal to one and less than or equal to the number of samples in the dataset.

```
41 # Part 2 - Building the LSTM
42
43 # Importing the Keras libraries and packages
44 from keras.models import Sequential
45 from keras.layers import Dense
46 from keras.layers import LSTM
47 from keras.layers import Dropout
48
49 # Initialising the RNN
50 regressor = Sequential()
51
52 # Adding the first LSTM layer and some Dropout regularisation
53 regressor.add(LSTM(units = 50, return_sequences = True))#, input_shape =(1496,3,9)))
54 regressor.add(Dropout(0.2))
55
56 # Adding a second LSTM layer and some Dropout regularisation
57 regressor.add(LSTM(units = 100, return_sequences = True))
58 regressor.add(Dropout(0.2))
59
60 # Adding a third LSTM layer and some Dropout regularisation
61 regressor.add(LSTM(units = 100, return_sequences = True))
62 regressor.add(Dropout(0.2))
63
64 # Adding a seventh LSTM layer and some Dropout regularisation
65 regressor.add(LSTM(units = 100 , return_sequences = True))
66 regressor.add(Dropout(0.2))
67
68 # Adding a eight LSTM layer and some Dropout regularisation
69 regressor.add(LSTM(units = 50 , return_sequences = True))
70 regressor.add(Dropout(0.2))
71
72 # Adding a nineth LSTM layer and some Dropout regularisation
73 regressor.add(LSTM(units = 50))
74 regressor.add(Dropout(0.2))
75
76 # Adding the output layer
77 # The number of units in the output layer corresponds to the number of inputs at different depths
78 regressor.add(Dense(units = 9))
79
80 # Compiling the RNN
81 regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')
82
83 # Fitting the RNN to the Training set
84 regressor.fit(X_train, y_train, epochs = 100, batch_size = 32)
```

*Figure 31 - Part 2: Building the LSTM model*

In the last part or Part 3: Making prediction and plotting, predict function is used to predict the values using the model *(* Figure 32*),* and Matplotlib is used to plot the values real vs predicted values.

```
# Part 3: Predicting and Visualizing Plot
predicted_values_test = regressor.predict(X_test)
plt.plot(predicted_values_test,c='C0', label = 'Predicted Values')
plt.plot(y_test,c='C1',linestyle='--', label = 'Real Values')
plt.title(' Test dataset ')
plt.show()

predicted_values_train = regressor.predict(X_train)
plt.plot(predicted_values_train,c='C0', label = 'Predicted Values')
plt.plot(y_train,c='C1',linestyle='--', label = 'Real Values')
plt.title(' Training dataset ')
plt.show()

predicted_values_test = regressor.predict(X)
plt.plot(predicted_values_test,c='C0', label = 'Predicted Values')
plt.plot(y,c='C1',linestyle='--', label = 'Real Values')
plt.title(' Total Dataset ')
plt.show()
```

*Figure 32 - Part 3: Making prediction and plotting*

Same Python code is used to make prediction for $Pressure\ Head\ (\psi)$ values, with data arranged in same manner as $water\ content\ (\theta)$.

## 4.2 Physics-Informed Neural Network (PINN)

Physics-Informed Neural Networks have been applied on Richard's equation to solve two kinds of problems:

- Interpolation Problem
- Inference Problem

Originally, it was planned to solve a third type of problem including these two called Inverse problem. But due to the lack of time it wasn't completed. In order to solve these problems, Richard's equation was converted into a loss function, which can be used by PINN. To do this, we can use equation (16).

$$\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial z}\left[K(\psi)\frac{\partial \psi}{\partial z} + K(\psi)\right]$$

This equation can be reformulated as follows:

$$\frac{\partial \theta}{\partial \psi}\frac{\partial \psi}{\partial t} = \frac{\partial K(\psi)}{\partial z}\frac{\partial \psi}{\partial z} + K(\psi)\frac{\partial^2 \psi^2}{\partial z} + \frac{\partial K(\psi)}{\partial z}$$

where, $C(\psi) = \frac{\partial \theta}{\partial \psi}$, is a water storage function:

$$C(\psi)\frac{\partial \psi}{\partial t} = \frac{\partial K(\psi)}{\partial z}\frac{\partial \psi}{\partial z} + K(\psi)\frac{\partial^2 \psi^2}{\partial z} + \frac{\partial K(\psi)}{\partial z}$$

The derivative of $K(\psi)$ with respect to $\psi$ is evaluated as follows:

$$C(\psi)\frac{\partial \psi}{\partial t} = \frac{\partial K(\psi)}{\partial \psi}\frac{\partial \psi}{\partial z}\frac{\partial \psi}{\partial z} + K(\psi)\frac{\partial^2 \psi^2}{\partial z} + \frac{\partial K(\psi)}{\partial \psi}\frac{\partial \psi}{\partial z}$$

Then, the Loss function for the training of the Neural Network is then defined as:

$$f = C(\psi)\frac{\partial \psi}{\partial t} - \frac{\partial K(\psi)}{\partial \psi}\frac{\partial \psi}{\partial z}\frac{\partial \psi}{\partial z} - K(\psi)\frac{\partial^2 \psi^2}{\partial z} - \frac{\partial K(\psi)}{\partial \psi}\frac{\partial \psi}{\partial z} = 0 \qquad (22)$$

In both type of problems, most of the libraries used were same as were in LSTM except TensorFlow and scipy.io, and the Keras wasn't imported in this code. In both of the problems, a class called PhysicsInformedNN was formed. In that class, lower and upper bound values, values of hydraulic conductivity$(K)$, water storage constant $(C)$, analytically calculated value of $\frac{\partial K(\psi)}{\partial \psi}$, a list called layers, and the grid in space and time as values of $(x, t)$ was passed as an argument. The list layers included the number of neurons in each layer. The process in that class is explained step wise as follows:

1. A Neural Network was set up which takes input as $x\ and\ t$ and tires to give an output.
2. This output is then used to find the differential terms in the loss function.
3. Then the interpolated values of $(K), (C), and\ \frac{\partial K(\psi)}{\partial \psi}$ are put together with the differential terms in the loss function.
4. After this process is repeated to minimize the loss function.

In interpolation problem the values of $(x, t)$, provided to the program were randomly from all over the domain, and using interpolation function to interpolate the values of $(K), (C), and\ \frac{\partial K(\psi)}{\partial \psi}$ program gave a coloured contour map for the whole domain. In inference problem, boundary values of $(x, t)$ were provided to the program and it gave a coloured domain for all the whole domain.

# Chapter 5
# Results & Discussions

In this chapter, results from LSTM and PINN are presented and discussed. Python codes implementing LSTM and PINN were run several times with different configurations to optimize the model.

## 5.1 LSTM

For LSTM four datasets were chosen to implement the algorithm and was studied under different configurations.

- $Water\ Content\ (\theta)$ dataset for Hygiene Sandstone with Closed Drainage
- $Water\ Content\ (\theta)$ dataset for Hygiene Sandstone with Open Drainage
- $Water\ Content\ (\theta)$ dataset for SiltLoamGE3 with Closed Drainage
- $Water\ Content\ (\theta)$ dataset for SiltLoamGE3 with Open Drainage

Four more datasets were produced with pressure head values $(\psi)$ in Hygiene Sandstone and SiltLoamGE3 each with open and closed drainage conditions. These were produced to verify the results obtained from water content $(\theta)$ datasets. LSTM was applied on all four $Water\ Content\ (\theta)$ datasets and the performance of the model was studied by changing number of layers in the model, number of neurons in each layer, number of epochs and the size of training set for the model. Table 3 below shows the specifics of the standard initial model. This model was kept as a reference to compare with the other configurations of the model.

| | |
|---|---|
| Number of layers | 4 |
| Number of Neurons in each layer | 50 |
| Number of Epochs | 50 |
| Length of training set | 700 |

*Table 3 - Specification of Reference model for each dataset*

In first variation, number of neurons were fixed at 50, Number of epochs were fixed at 50, length of training set was 700, and three scenarios were tested with number of layers as 3,4 and

6 respectively. Since, feature scaling was applied to the dataset, all the values were squashed between zero to one. Therefore, all the predicted values are also between zero and one. In Figure 33, each red line in the graphs shows the water content build up at certain depth, and green lines are the predicted values on the same depths. In *Figure 33, (a), (b)* and *(c)* are results of water content in Hygiene Sandstone with closed drainage, and *(d), (e)* and *(f)* are the results of water content in SiltLoamGE3 with closed drainage. In all the graphs in *Figure 33*, it can be observed very clearly that the model is not able to predict for the last four lines i.e., after time step 700.

In Figure 34, graphs *(a), (b)* and *(c)* shows the water content in Hygiene Sandstone with open drainage, while *(d), (e)* and *(f)* shows the same in SiltLoamGE3 with open drainage. In *Figure 34*, too it can be observed pretty clearly that the model fails to predict the values of water content after time step 700. Apart from that no major trend can be observed in the results. Sometimes predicted values exceeds the range of 0 to 1, but that is because the limit is not applied to the predictions, it exceeds because it tries to follow the trend.

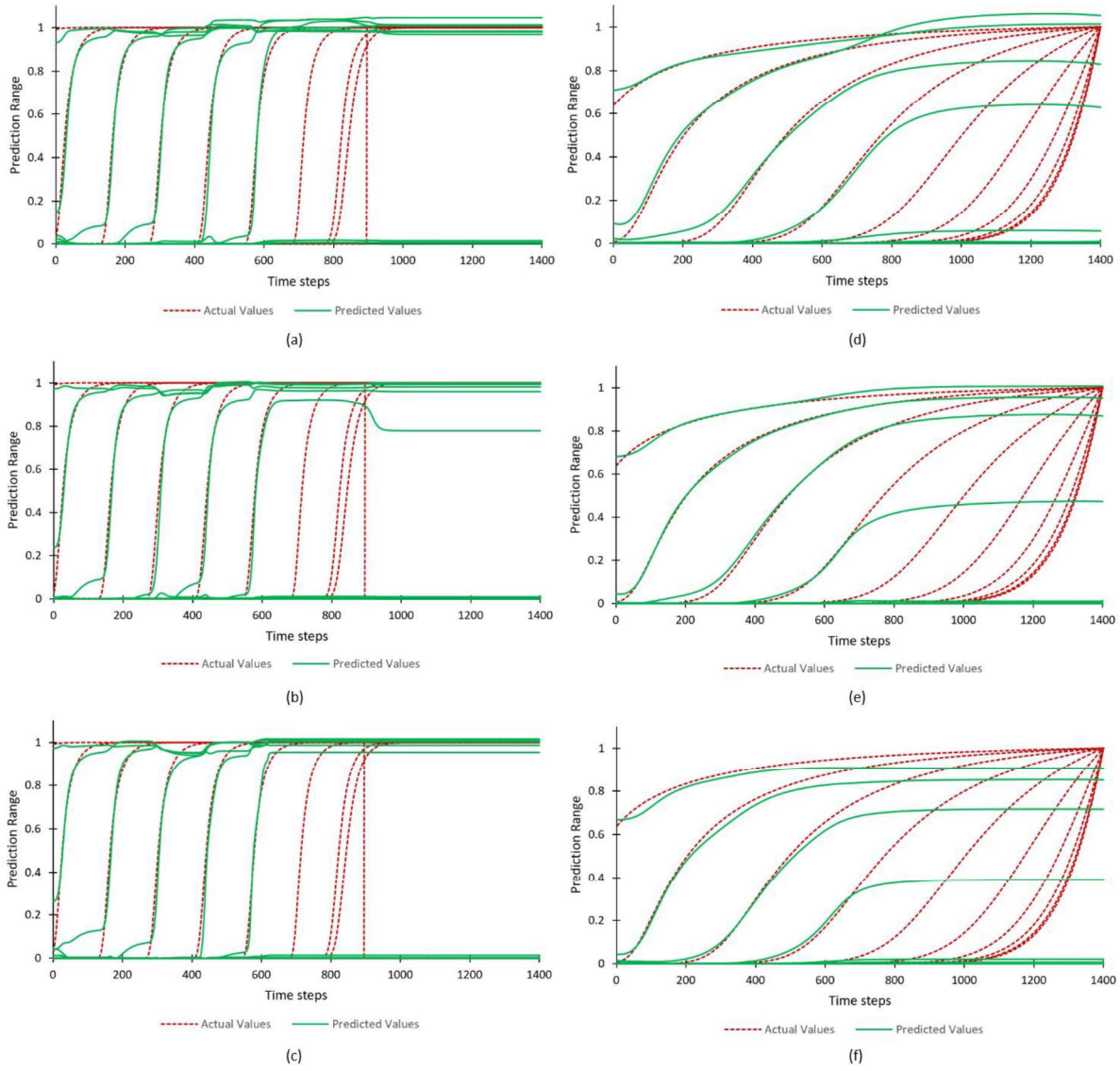



*Figure 33 - Shows the result of LSTM for Hygiene Sandstone with closed drainage (a) – 3 layers, (b) – 4 layers and (c) – 6 layers. (d) – 3 layers, (e) – 4 layers, and (f) – 6 layers shows the results for SiltLoamGE3 with closed drainage.*

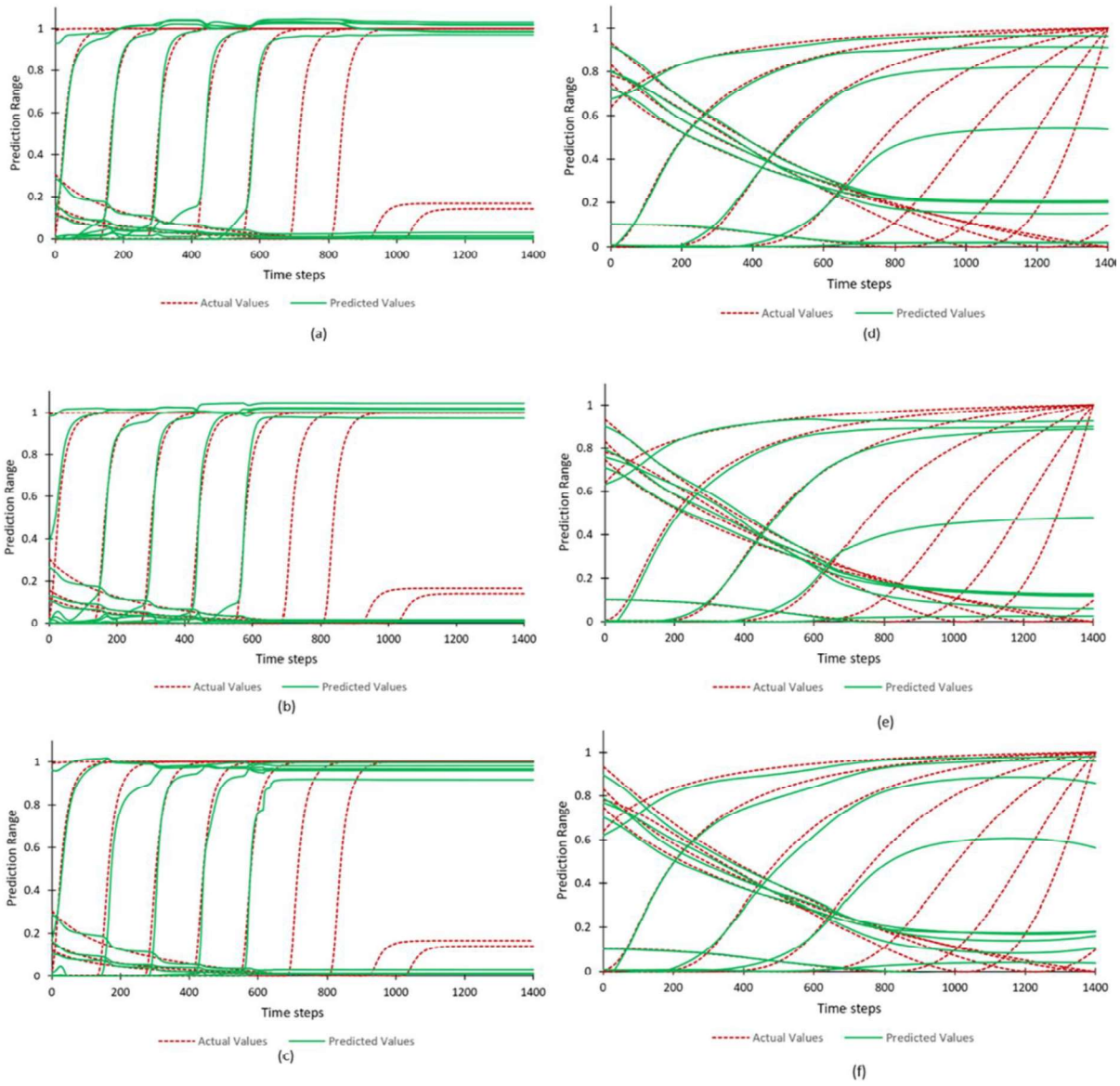



*Figure 34 - Shows the result of LSTM for Hygiene Sandstone with open drainage (a) – 3 layers, (b) – 4 layers and (c) – 6 layers. (d) – 3 layers, (e) – 4 layers, and (f) – 6 layers shows the results for SiltLoamGE3 with open drainage.*

In second variation, number of neurons in each layer was varied, while keeping number of layers, number of epochs and length of training set as fixed. In this case, three scenarios were tested with 30, 40 and 50 neurons in each layer and the results were presented in Figure 35 and Figure 36. *Figure 35, (a), (b)* and *(c)* are results of water content in Hygiene Sandstone with closed drainage, and *(d), (e)* and *(f)* are the results of water content in SiltLoamGE3 with closed drainage. Similarly, *Figure 36, (a), (b)* and *(c)* are results of water content in Hygiene Sandstone with open drainage, and *(d), (e)* and *(f)* are the results of water content in SiltLoamGE3 with open drainage. Again, just like *Figure 33* and *34*, In *Figure 35* and *36* same patterns are observed, that the model is not able to predict for the last four lines i.e., after time step 700. This suggests that neither widening nor deepening the network is effective, in order to improve the model performance.

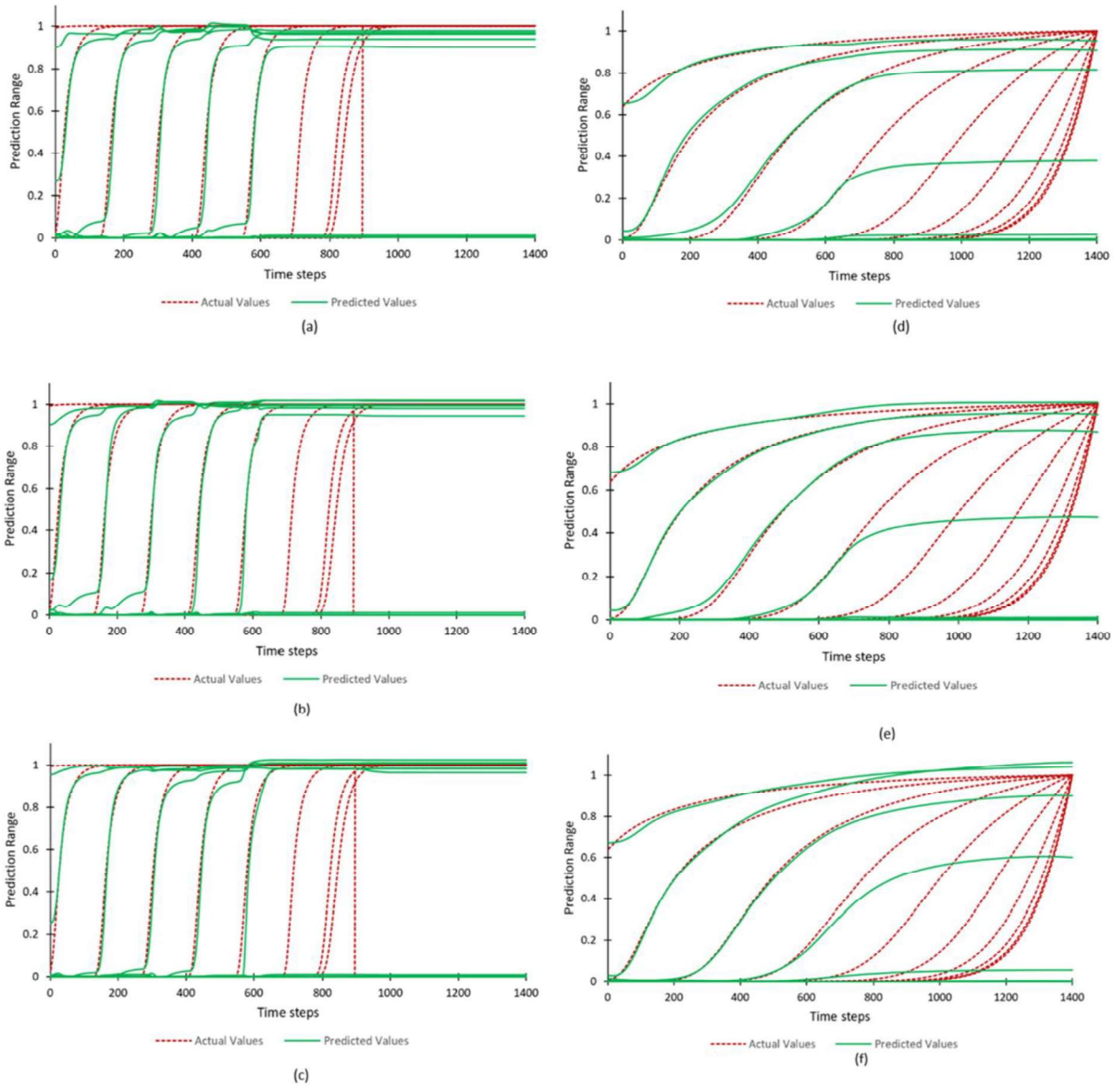
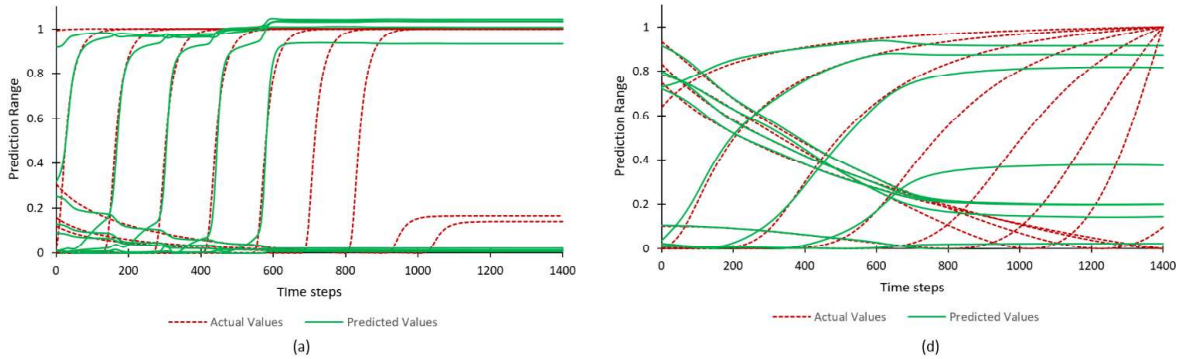



*Figure 35 - Shows the result of LSTM for Hygiene Sandstone with closed drainage (a) – 30 neurons, (b) – 40 neurons and (c) – 50 neurons. (d) – 30 neurons, (e) – 40 neurons, and (f) – 50 neurons show the results for SiltLoamGE3 with closed drainage.*



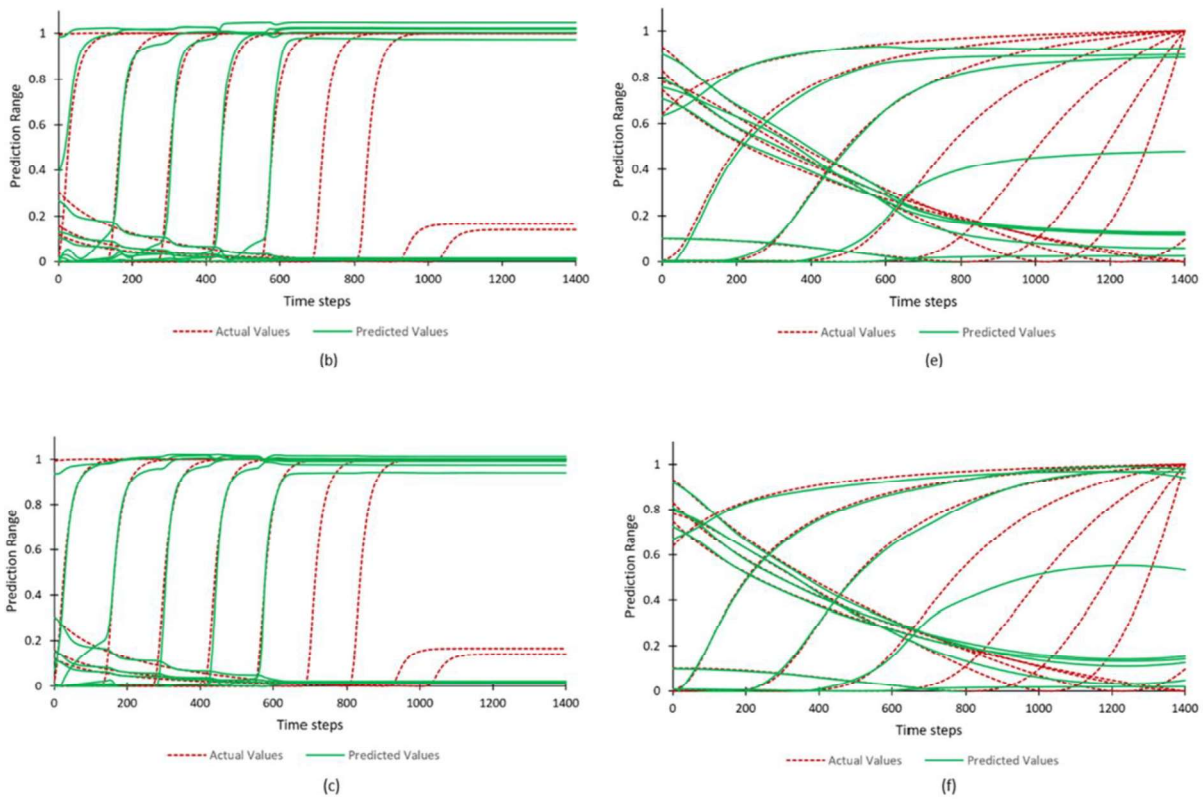



*Figure 36 - Shows the result of LSTM for Hygiene Sandstone with open drainage (a) – 30 neurons, (b) – 40 neurons and (c) – 50 neurons. (d) – 30 neurons, (e) – 40 neurons, and (f) – 50 neurons show the results for SiltLoamGE3 with open drainage.*

For third variation, number of epochs was changed from 50 to 100, while keeping number of layers, number of neurons in each layer and length of the training set was kept constant. This type of variation is supposed to reveal if the original model is overfitting the dataset or underfitting it. But as seen in Figure 37 *and* Figure 38, increasing the number of epochs, too doesn't bring any significant change in the results.

The last type of variation that is studied in this thesis is changing the length of training set. While, in this type number of layers, number of neurons in the layers and number of epochs are kept constant. Figure 39*, (a), (b)* and *(c)* are results of water content in Hygiene Sandstone with closed drainage, and *(d), (e)* and *(f)* are the results of water content in SiltLoamGE3 with closed drainage. While, *Figure 39 (a)* shows the result with 700 datapoints as training set, *(b)* shows 1000 and *(c)* shows 1500 datapoints as training set in Hygiene Sandstone with closed drainage. Similarly, *Figure 39 (d)* shows the result with 700 datapoints as training set, *(e)* shows 1000 and *(f)* shows 1500 datapoints as training set in SiltLoamGE3 with closed drainage. Moreover, Figure 40, shows the similar observations for open drainage condition in Hygiene Sandstone and SiltLoamGE3.

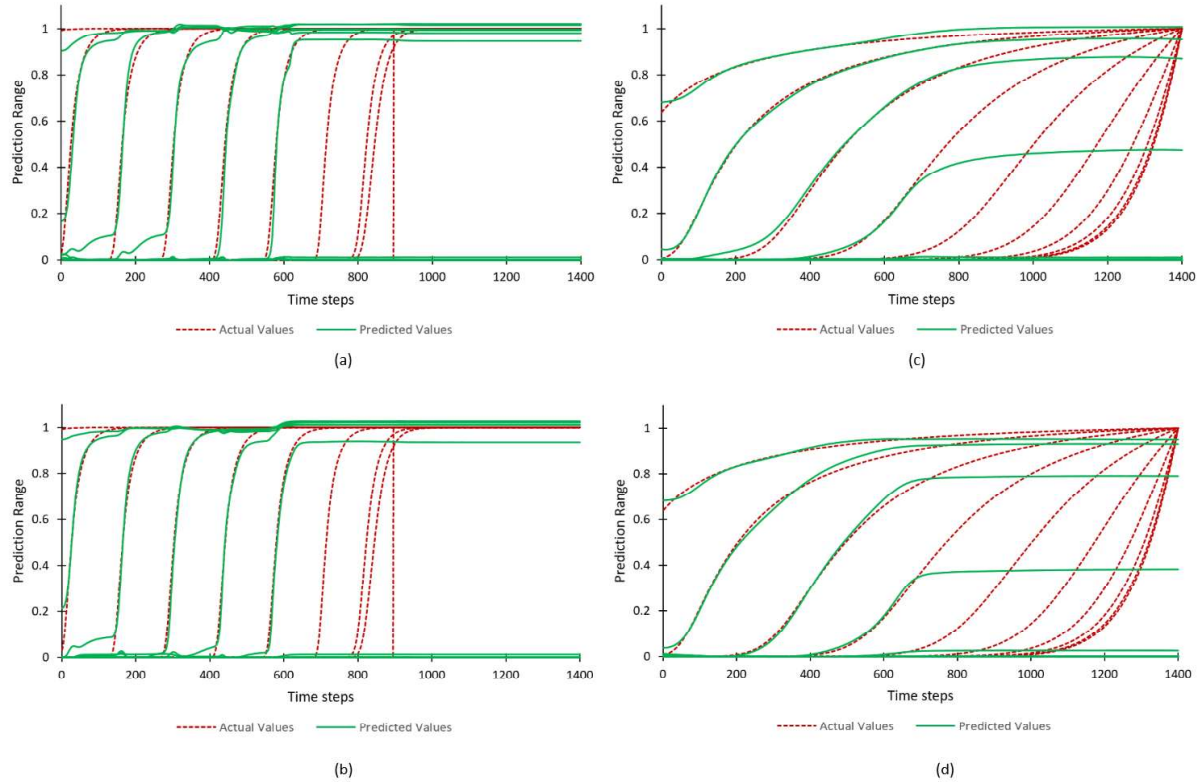



*Figure 37 - Shows the result of LSTM for Hygiene Sandstone with closed drainage (a) – 50 epochs and (b) – 100 epochs. (c) – 50 epochs and (d) – 100 epochs show the results for SiltLoamGE3 with closed drainage.*

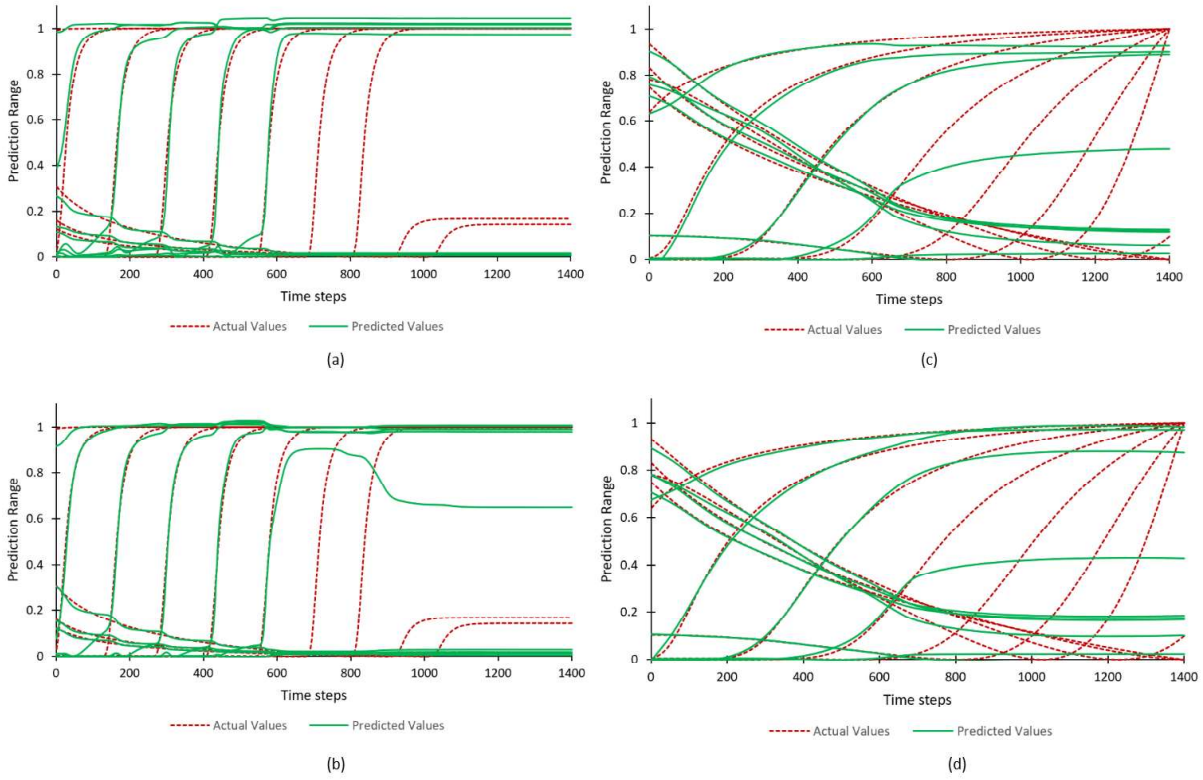



*Figure 38 - Shows the result of LSTM for Hygiene Sandstone with open drainage (a) – 50 epochs and (b) – 100 epochs. (c) – 50 epochs and (d) – 100 epochs show the results for SiltLoamGE3 with open drainage.*

In Figure 39 *and* Figure 40, it can very well be noticed, that with the increase in the length of training set, performance of the model increases quite a lot. This can be explained by taking a careful look on the dataset. At every depth water content is taking quite a steep and sudden jump at a certain point in time. Therefore, the model gives good prediction till the point in time, it was trained for. Because, for all the other depths which didn't made the jump yet, were more or less constant. Hence constant prediction for those depths. Figure 41 shows the result for pressure head ($\psi$) datasets, which are produced with 1500 datapoints as training set, 4 hidden layers and 50 neurons in each layer. Therefore, verifying the results produced in *Figure 39 and 40* are valid for pressure head too.

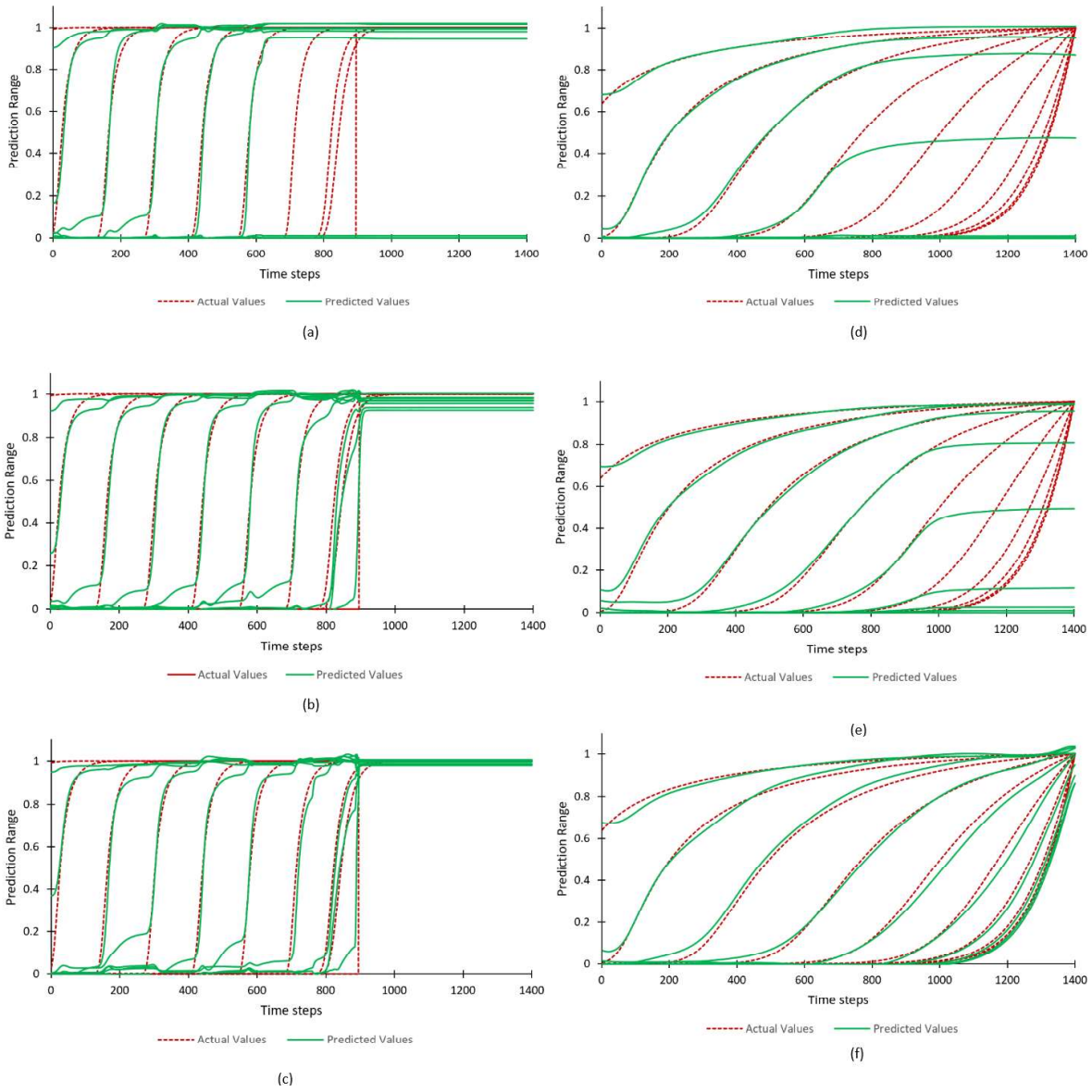



*Figure 39 - Shows the result of LSTM for Hygiene Sandstone with closed drainage (a) – 700 length of training set, (b) – 1000 length of training set and (c) – 1500 length of training set. (d) – 700 length of training set, (e) – 1000 length of training set, and (f)*

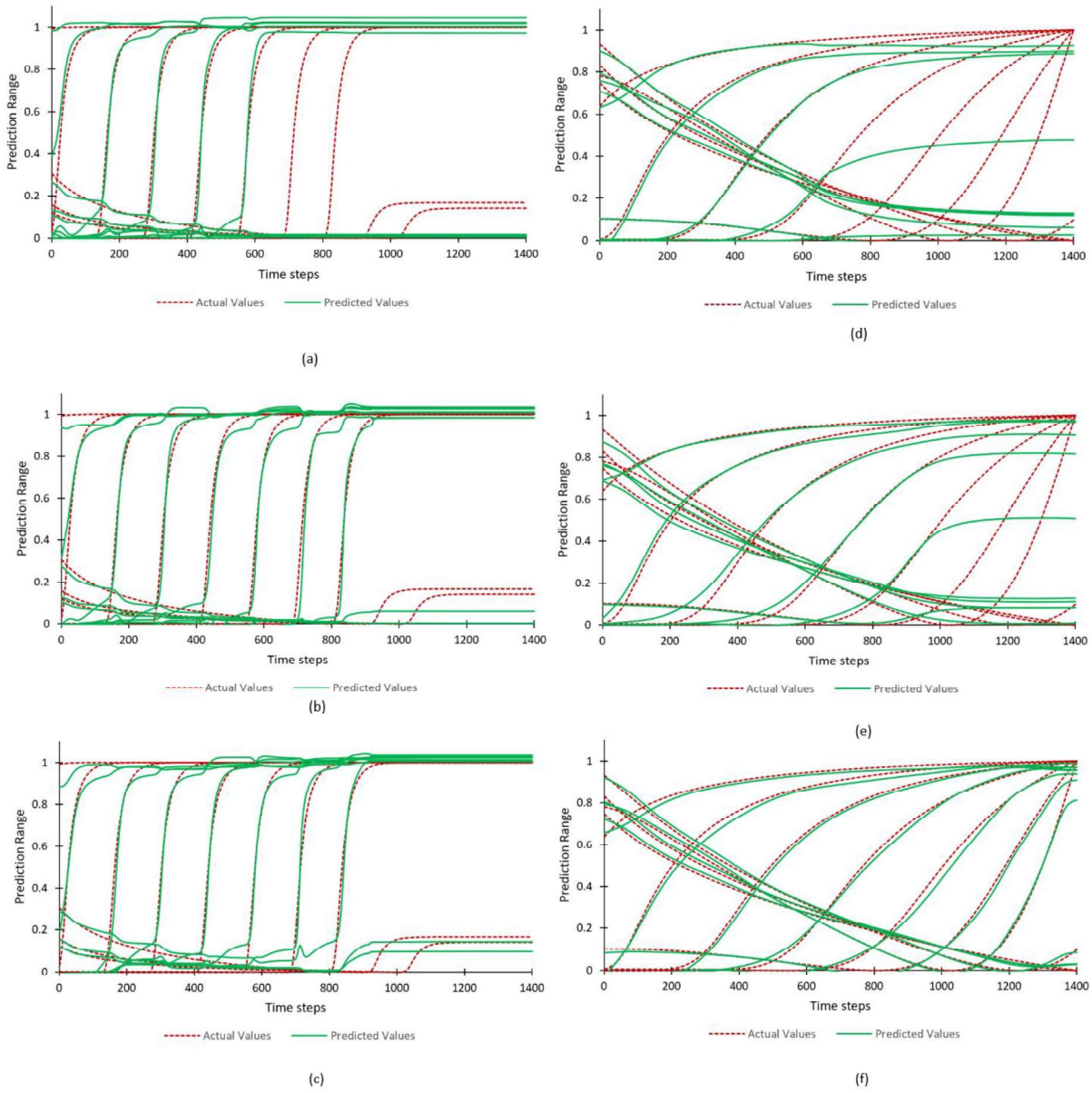
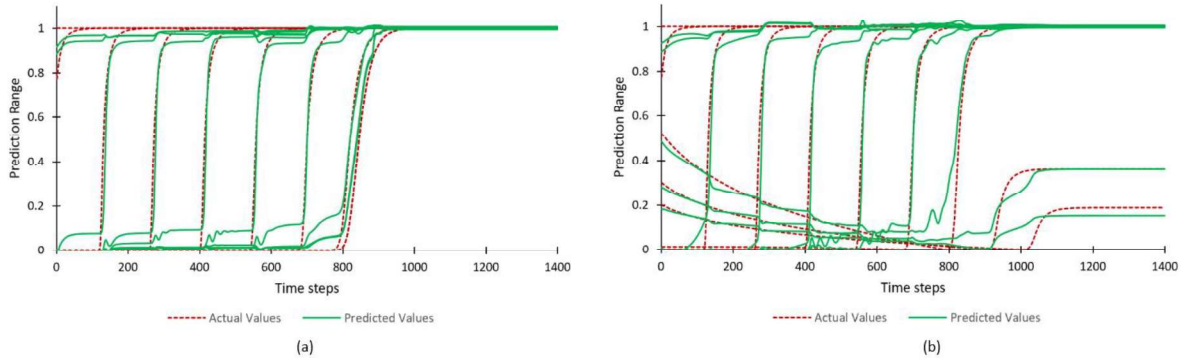



*Figure 40 - Shows the result of LSTM for Hygiene Sandstone with open drainage (a) – 700 length of training set, (b) – 1000 length of training set and (c) – 1500 length of training set. (d) – 700 length of training set, (e) – 1000 length of training set, and (f) – 1500 length of training set show the results for SiltLoamGE3 with open drainage.*



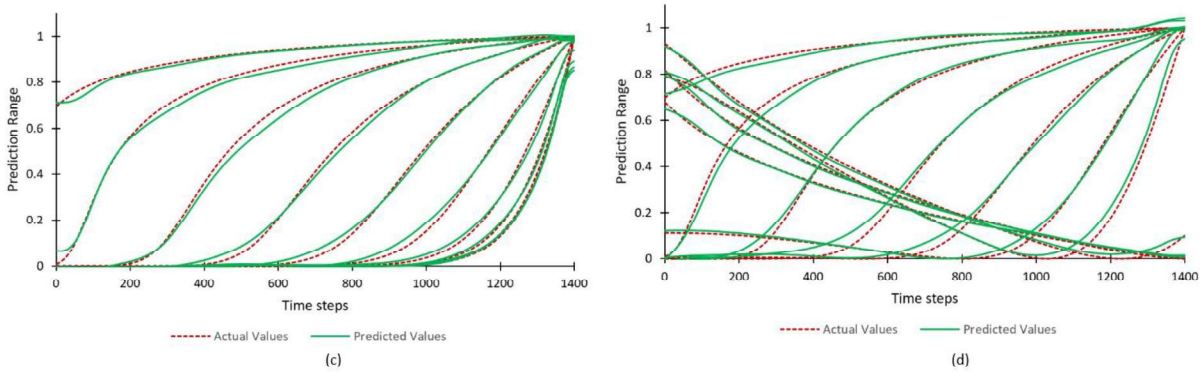



*Figure 41 - Shows the results for pressure head with length of training dataset = 1500, number of layers = 4, number of neurons in each layer 50 and number of epochs = 50. (a) Hygiene Sandstone Closed drainage (b) Hygiene Sandstone open drainage (c) SiltLoamGE3 Closed drainage and (d) SiltLoamGE3 open drainage.*

# 5.2 PINN

This section describes the results of the application of Physics – Informed Neural Networks on Richard's equation. This was done in two ways. In First application, collocation points were spread in the whole domain and this was called an Interpolation problem. Because at these collocation points values of Hydraulic conductivity, water content and water storage constant were provided. Using these, neural network was supposed to find the solution of Richards equation in the whole domain. This required to interpolate these properties in the domain. In the second application, these collocation points were provided on the boundary of the domain. Therefore, neural network was supposed to find the solution of Richard's Equation in the whole domain, but this time it was called as an Inference problem.

Figure 42 summarizes the result for Richards equation for interpolation problem. Specifically, given a set of 500 collocation points i.e., $N_u$, and are randomly distributed all around the domain. Solution of Richard's equation was found by training a 6 layered deep neural network with 20 neurons in each layer. This configuration resulted in lowest loss value i.e., $2.266936 \times 10^{-4}$. Other configurations of the model were tried with different number of collocation points, number of layers and number of neurons in each layer, loss values of these are presented in Table 4, Table 5 *and* Table 6 below. Moreover, it's important to note that none other significant trends can be observed in the *Table 4, 5 and 6*, except finding a best configuration with almost hit and trial like technique.

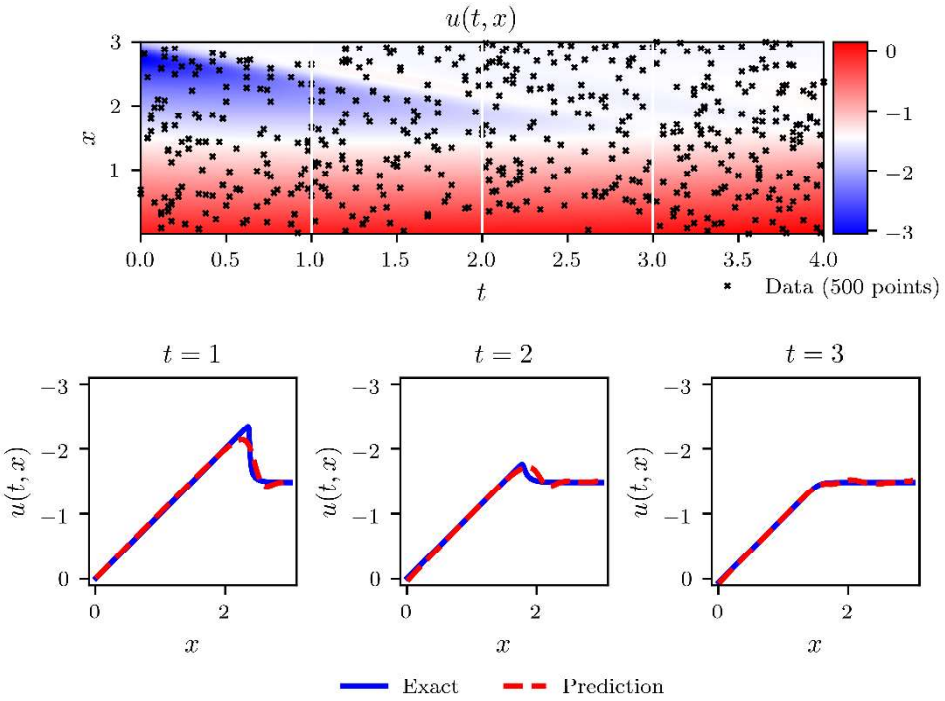



*Figure 42 - Top: Predicted Solution for u (x, t) along with the training data Nu = 500. Bottom: Comparison of the predicted and exact solution corresponding to the three temporal snapshots depicted by the white vertical lines in the top panel.*

| Layers | Neurons | | | | |
|---|---|---|---|---|---|
| | **10** | **20** | **30** | **40** | **50** |
| **2** | 4.90E-04 | 8.79E-04 | 1.67E-03 | 6.16E-04 | 1.10E-03 |
| **4** | 6.20E-04 | 7.87E-04 | 8.51E-04 | 1.75E-03 | 5.54E-04 |
| **6** | 9.63E-04 | 3.72E-04 | 5.01E-04 | 6.70E-04 | 1.26E-03 |
| **8** | 3.32E-03 | 2.14E-03 | 1.77E-03 | 3.72E-03 | 7.63E-04 |
| **10** | 7.86E-03 | 2.88E-03 | 1.84E-02 | 3.20E-03 | 1.06E-03 |

*Table 4 - Collocation points, $N_u$ = 200*

| Layers | Neurons | | | | |
|---|---|---|---|---|---|
| | **10** | **20** | **30** | **40** | **50** |
| **2** | 1.46E-03 | 1.03E-03 | 1.80E-03 | 1.80E-03 | 1.35E-03 |
| **4** | 2.00E-03 | 2.21E-03 | 9.70E-04 | 3.20E-03 | 1.90E-03 |
| **6** | 2.11E-03 | 2.27E-04 | 1.02E-03 | 2.91E-03 | 1.89E-03 |
| **8** | 2.80E-03 | 1.84E-03 | 2.01E-03 | 1.99E-03 | 1.87E-03 |
| **10** | 2.55E-03 | 5.78E-03 | 8.82E-04 | 3.48E-03 | 3.33E-03 |

*Table 5 - Collocation Points, $N_u$ = 500*

| Layers | Neurons | | | | |
|---|---|---|---|---|---|
| | **10** | **20** | **30** | **40** | **50** |
| **2** | 2.01E-03 | 2.59E-03 | 1.02E-03 | 2.20E-03 | 1.39E-03 |
| **4** | 1.74E-03 | 1.16E-03 | 6.50E-04 | 2.78E-03 | 1.31E-03 |
| **6** | 7.54E-04 | 1.21E-03 | 1.40E-03 | 1.29E-03 | 2.95E-03 |
| **8** | 1.09E-03 | 2.01E-03 | 3.94E-03 | 5.11E-03 | 7.35E-04 |
| **10** | 1.61E-03 | 6.59E-03 | 4.56E-03 | 4.91E-03 | 1.06E-02 |

*Table 6 - Collocation Points, $N_u$ = 700*

Result for the inference problem is summarized in Figure 43. It is generated with $N_u = 100$ and $N_f = 4000$, with a two layers deep neural network with 20 neurons in each layer. This set of 100 datapoints is randomly distributed initial and boundary data. The top panel of Figure 54 shows the predicted spatio-temporal solution of Richard's equation, along with the location of initial and boundary data. With this configuration an error of $5.081357 \times 10^{-2}$ is reported.

To further analyse the performance of this method, some parametric study was done to quantify its predictive accuracy for different number of training and collocation points, for different neural network architectures. Table 7 reports the resulting error for different number of initial and boundary training data $N_u$ and different collocation points $N_f$. While keeping the two layers deep neural network with 20 neurons in each layer constant. Though lowest error was encountered with $N_u = 100$ and $N_f = 4000$, but some results with lower loss were also found with $N_f = 10000$. Furthermore, Table 8 shows the resulting error for different number of hidden layers, and different number of neurons per layer, while the total number of training and collocation points is kept fixed to Nu = 100 and Nf = 4000. It is to be expected that as the number of layers and neurons is increased (hence the capacity of the neural network to approximate more complex functions), the predictive accuracy of the network should increase but unfortunately a pattern like this isn't visible in this case.

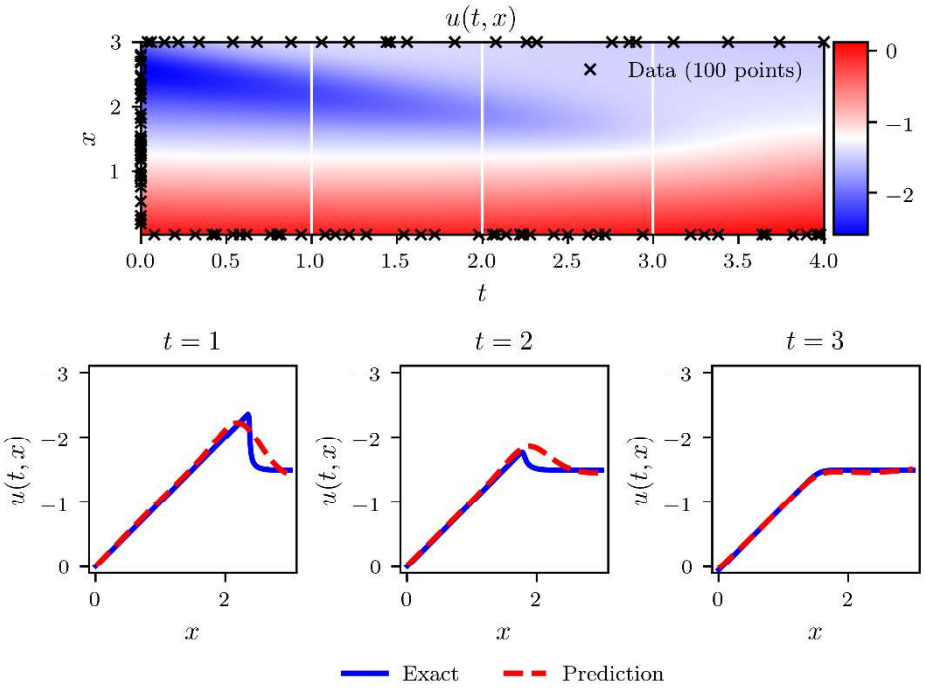



*Figure 43 - Richard's Equation: Top: Predicted solution u (x, t) along with the initial and boundary training data. Bottom: Comparison of the predicted and exact solution corresponding to the three temporal snapshots depicted by the white vertical lines in the top panel. Error for this case was $5.081357 \times 10^{2}$.*

| $N_u$ | $N_f$ | | | | |
|---|---|---|---|---|---|
| | **2000** | **4000** | **6000** | **8000** | **10000** |
| **20** | 1.28E-01 | 1.62E-01 | 2.11E-01 | 1.52E-01 | 1.24E-01 |
| **40** | 1.17E-01 | 9.70E-02 | 9.32E-02 | 9.62E-02 | 1.30E-01 |
| **60** | 1.67E-01 | 1.24E-01 | 2.25E-01 | 9.82E-02 | 6.48E-02 |
| **80** | 1.37E-01 | 1.19E-01 | 1.90E-01 | 1.35E-01 | 1.86E-01 |
| **100** | 1.36E-01 | 5.66E-02 | 1.06E-01 | 7.03E-02 | 8.46E-02 |
| **200** | 1.24E-01 | 7.29E-02 | 9.48E-02 | 1.11E-01 | 6.81E-02 |

*Table 7 - Richards Equation: Error between the predicted and the exact solution u (x, t) for different number of initial and boundary training data $N_u$, and different number of collocation points $N_f$. Here the network architecture is fixed to 2 layers with 20 neurons per hidden layer.*

| **Layers** | **Neurons** | | | | |
|---|---|---|---|---|---|
| | **10** | **20** | **30** | **40** | **50** |
| **2** | 1.43E-01 | 5.08E-02 | 1.29E-01 | 9.45E-02 | 1.90E-01 |
| **4** | 1.01E-01 | 1.04E-01 | 1.30E-01 | 9.86E-02 | 1.65E-01 |
| **6** | 2.36E-01 | 1.10E-01 | 1.37E-01 | 1.07E-01 | 1.49E-01 |
| **8** | 2.52E-01 | 1.04E-01 | 2.65E-01 | 1.25E-01 | 7.27E-02 |

*Table 8 - Richards Equation: Error between predicted and the exact solution u (x, t) for different number of hidden layers and different number of neurons per layer. Here the total number of training and collocation points is fixed to $N_u = 100$ and $N_f = 4000$, respectively.*

## 5.3 Discussion

Throughout this chapter results from LSTM and PINN are presented. In case of LSTM, using a Python code *RichardsEquationGenerator.py*, four separate datasets of water content $(\theta)$, in Hygiene Sandstone and SiltLoamGE3, with open and closed drainage conditions each were generated. In each dataset, value of water content was calculated using *RichardsEquationGenerator.py*, in 10 points in space along a depth of 0 to 5 meters with an equal interval of 0.5 meters. Both materials were subjected to an influx of 0.01 m/day of water, and water content was calculated in approximately every 10 minutes for 10 days, at every datapoint. Therefore, for each point in space there were 1500 sequential values of water content.

These sequential datasets were fed to LSTM, a part of it was used as training set and rest of it was used for testing the prediction. After varying number of layers, number of neurons in each layer and epochs, only parameter to which the model seems to improve was change in the length of training set Figure 39 and Figure 40. This behaviour of LSTM can be attributed to the fact, that Richard's equation is highly non-linear, and the data needed to train the LSTM was not quite sufficient. Since the model was trained in time and it was not interacting with different depths, it is safe to assume that model didn't understand when to make the transition from 0 to 1. Hence, if the model was trained till 700 timesteps, during prediction it successfully predicted the transition for the depths it was already trained for. Since, there was no learning between different depths, it didn't knew when the transition happens for rest of the depths. Hence, the model predicted constant or close to zero values for rest of the depths.

If the dataset consisted of several cycles of wetting and drying, instead of just wetting, LSTM would have performed better. Model would have learnt more about the wetting and drying

characteristics of the material with certain amount of flow. Alternatively, using spatio – temporal LSTM or ST-LSTM, this dataset can be trained in space and time *[14]*, hence better prediction.

In case of Physics-Informed Neural Network or PINN, Richard's equation was modelled as an optimization problem and was solved using neural networks. This was done by setting a neural network *u (x, t)*. It takes *x* and *t* as inputs and give out a value *u.* Now, this *u* is used to find differential terms in the Richard's equation, by differentiating *w.r.t x* and *t,* using automatic differentiation. Then, rest of the values of Hydraulic conductivity *K*, and water storage constant *C*, were provided, and then the loss function was calculated Equation (22). This process was repeated, in order to minimize the loss function.

Results produced with PINN for Interpolation problem were quite good, as the error was quite low i.e., $2.266936 \times 10^{-4}$. However, in inference problem error was quite high i.e., in the magnitude of $10^{-2}$. Although there is good reason to believe that on further probing in terms of different combinations of $N_u$ and $N_f$ values with deeper neural network architecture, one may arrive at a lower error in inference problem i.e., in magnitude of $10^{-3}$ or $10^{-4}$. Moreover, key strength of physics informed neural networks is believed to be quite accurate and data efficient as the underlying physical law is encoded in the neural networks *[15]*. Hence, this technique is different from usual neural network technique and makes use of known physical knowledge along with high computational power of neural networks.

Furthermore, LSTM is quite good and effective but for sequential and time series data. For solving ordinary differential equations or partial differential equations, which is often the case in science and engineering problem physics informed neural networks can perform better.

# Chapter 6
# Conclusions

This thesis explored the idea of applying machine learning to infiltration process in a soil. Machine learning techniques used in the process were vastly different in terms of working and nature from each other. First technique used in this thesis is called Long Short-Term Memory (LSTM). This technique specializes in sequential or time series data. Therefore, this technique is particularly good in predicting stock prices, weather patterns etc i.e., with sequential data. Hence, values of water content $(\theta)$, and pressure head $(\psi)$ in both the materials were arranged in sequential manner with 1500 datapoints at 10 different depths. Since, there was no learning between points at different depths, LSTM model treated sequential data in all the depths as totally different series. Therefore, in the result obtained model gave good prediction for those depths which transitioned from unsaturated to saturated phase within the training set.

The second technique used is called Physics-informed neural networks (PINN). Whereas, LSTM was a very traditional Machine learning technique in which there are well defined training and testing sets, in PINN, there were no strictly defined training or testing sets. In this technique, the underlying physical law, in our case Richard's equation itself was encoded in the neural network. Therefore, collocation points inside the domain are used to train the algorithm, then solution of the differential equation was predicted for the whole domain. And collocation points can be any number of points chosen from the domain that can be used to train the algorithm. This technique gave quite good result, in case of interpolation problem it gave an error of $2.266936 \times 10^{-4}$, while in case of inference problem it gave quite high error in magnitude of $10^{-2}$. This high error in inference problem can most likely be lowered by increasing the number of datapoints i.e., $N_u$ and $N_f$, and deepening and widening the neural network. If not then, this is a convergence and generalization problem of neural network i.e., the minimizer or set of values that minimizes the loss function found by neural network doesn't match the exact solution of the equation *[16]*. Further work can be done in this direction, to investigate.

Moreover, for this thesis PINN proved to be a much better method to mimic infiltration, as it can encode Richard's equation in neural network. Due to lack of time PINN wasn't modelled to produce a SWCC graph, but an inverse problem using PINN can definitely be modelled to produce one. In this type of problem model will be predicting the values of water content and pressure head and hence producing SWCC.

PINN is more suitable for this problem than LSTM because ultimately it can be developed to produce SWCC (though might need to figure out how to specify the material in the program),

but for LSTM we need to have a lot of sequential data to model and feed to the algorithm. And it would be very specific for the case. Moreover, PINN can be used much more widely in geotechnical engineering or engineering applications in general, due to the abundance of ordinary and partial differential equations.

There are still many questions that need to be addressed regarding PINN's convergence to the solution and generalization of the data. However, there is good reason to believe that PINN is a big step forward in the direction of automation to solve ODE and PDE, using theory driven data science. As this kind of approach allows to use the knowledge of scientific laws combined with the computational power of neural networks.

# References


[1] D. Porebska, C. Sławiński, K. Lamorski, and R. T. Walczak, “Relationship between van Genuchten’s parameters of the retention curve equation and physical properties of soil solid phase,” *Int. Agrophysics*, vol. 20, no. 2, pp. 153–159, 2006.

[2] N. Yousefpour and S. Fallah, “Application of Machine Learning in Geotechnics,” no. August, pp. 1–4, 2018.

[3] N. S. Juwaied, “Applications of artificial intelligence in geotechnical engineering,” *ARPN J. Eng. Appl. Sci.*, vol. 13, no. 8, pp. 2764–2785, 2018.

[4] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997, doi: 10.1162/neco.1997.9.8.1735.

[5] T. J. Sejnowski, P. S. Churchland, and J. A. Movshon, “Putting big data to good use in neuroscience,” *Nature Neuroscience*. 2014, doi: 10.1038/nn.3839.

[6] Nature, “Big data - Science in the petabyte era,” *Nature*, 2008.

[7] A. Karpatne *et al.*, “Theory-guided data science: A new paradigm for scientific discovery from data,” *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 10, pp. 2318–2331, 2017, doi: 10.1109/TKDE.2017.2720168.

[8] A. W. Trask, *Deep Learning: grokking*. 2016.

[9] M. Tuller and D. Or, “Water Retention and Characteristic Curve,” *Encycl. Soils Environ.*, vol. 4, no. January 2004, pp. 278–289, 2004, doi: 10.1016/B0-12-348530-4/00376-3.

[10] M. Tuller and D. Or, “Hydraulic conductivity of variably saturated porous media: Film and corner flow in angular pore space,” *Water Resour. Res.*, vol. 37, no. 5, pp. 1257–1276, 2001, doi: 10.1029/2000WR900328.

[11] M. T. van Genuchten, “A Closed-form Equation for Predicting the Hydraulic Conductivity of Unsaturated Soils,” *Soil Sci. Soc. Am. J.*, 1980, doi: 10.2136/sssaj1980.03615995004400050002x.

[12] L. A. Richards, “Capillary conduction of liquids through porous mediums,” *J. Appl. Phys.*, 1931, doi: 10.1063/1.1745010.

[13] N. M. Fenneman, *Geology of the Boulder District, Colorado*, vol. Chapter 3. 1905.

[14] Q. Tang, M. Yang, and Y. Yang, “ST-LSTM: A Deep Learning Approach Combined Spatio-Temporal Features for Short-Term Forecast in Rail Transit,” *J. Adv. Transp.*, vol. 2019, 2019, doi: 10.1155/2019/8392592.

[15] M. Raissi, P. Perdikaris, and G. E. Karniadakis, “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations,” *J. Comput. Phys.*, vol. 378, pp. 686–707, 2019, doi: 10.1016/j.jcp.2018.10.045.

[16] Y. Shin, J. Darbon, and G. E. Karniadakis, “On the Convergence and generalization of Physics Informed Neural Networks,” vol. 02912, pp. 1–29, 2020, [Online]. Available: http://arxiv.org/abs/2004.01806.

# Appendix 1

1. "vanGenuchten.py" – Sourced from "https://github.com/amireson/RichardsEquation"

```
# These are the van Genuchten (1980) equations
# The input is matric potential, psi and the hydraulic parameters.
# psi must be sent in as a numpy array.
# The pars variable is like a MATLAB structure.
import numpy as np

def thetaFun(psi,pars): #water content
    if psi>=0.:
        Se = 1.
    else:
        Se=(1+abs(psi*pars['alpha'])**pars['n'])**(-pars['m'])
    return pars['thetaR']+(pars['thetaS']-pars['thetaR'])*Se

thetaFun=np.vectorize(thetaFun)

def CFun(psi,pars): #water storage function
    if psi>=0.:
        Se=1.
    else:
        Se=(1+abs(psi*pars['alpha'])**pars['n'])**(-pars['m'])
    dSedh=pars['alpha']*pars['m']/(1-pars['m'])*Se**(1/pars['m'])*(1-Se**(1/pars['m']))**pars['m']
    return Se*pars['Ss']+(pars['thetaS']-pars['thetaR'])*dSedh

CFun = np.vectorize(CFun)

def KFun(psi,pars): #hydraulic conductivity
    if psi>=0.:
        Se=1.
    else:
        Se=(1+abs(psi*pars['alpha'])**pars['n'])**(-pars['m'])
    return pars['Ks']*Se**pars['neta']*(1-(1-Se**(1/pars['m']))**pars['m'])**2

KFun = np.vectorize(KFun)

def setpars():
  pars={}
  pars['thetaR']=float(raw_input("thetaR = "))
  pars['thetaS']=float(raw_input("thetaS = "))
  pars['alpha']=float(raw_input("alpha = "))
  pars['n']=float(raw_input("n = "))
  pars['m']=1-1/pars['n']
  pars['Ks']=float(raw_input("Ks = "))
  pars['neta']=float(raw_input("neta = "))
  pars['Ss']=float(raw_input("Ss = "))
  return pars
```

```

def PlotProps(pars):
  import numpy as np
  import pylab as pl
  import vanGenuchten as vg
  psi=np.linspace(-10,2,200)
  pl.figure
  pl.subplot(3,1,1)
  pl.plot(psi,vg.thetaFun(psi,pars))
  pl.ylabel(r'$\theta(\psi) [-]$')
  pl.subplot(3,1,2)
  pl.plot(psi,vg.CFun(psi,pars))
  pl.ylabel(r'$C(\psi) [1/m]$')
  pl.subplot(3,1,3)
  pl.plot(psi,vg.KFun(psi,pars))
  pl.xlabel(r'$\psi [m]$')
  pl.ylabel(r'$K(\psi) [m/d]$')
  #pl.show()
```

```
71
72  def HygieneSandstone():
73  pars={}
74  pars['thetaR']=0.153
75  pars['thetaS']=0.25
76  pars['alpha']=0.79
77  pars['n']=10.4
78  pars['m']=1-1/pars['n']
79  pars['Ks']=1.08
80  pars['neta']=0.5
81  pars['Ss']=0.000001
82  return pars
83
84 def SiltLoamGE3():
85  pars={}
86  pars['thetaR']=0.131
87  pars['thetaS']=0.396
88  pars['alpha']=0.423
89  pars['n']=2.06
90  pars['m']=1-1/pars['n']
91  pars['Ks']=0.0496
92  pars['neta']=0.5
93  pars['Ss']=0.000001
94  return pars
```

2. *RichardsEquationgenerator.py* – Sourced from "https://github.com/amireson/RichardsEquation"

```
 8 # Import all of the basic libraries (you will always need these)
 9 from matplotlib import pyplot as pl
10 import numpy as np
11
12 # Import a library that contains soil moisture properties and functions
13 import vanGenuchten as vg
14
15 # Import ODE solvers
16 from scipy.interpolate import interp1d
17 from scipy.integrate import odeint
18
19 # Select which soil properties to use
20 p=vg.HygieneSandstone()
21
22 # Richards equation solver
23 # This is a function that calculated the right hand side of Richards' equation. You
24 # will not need to modify this function, unless you are doing something advanced.
25 # This block of code must be executed so that the function can be later called.
26
27 def RichardsModel(psi,t,dz,n,p,vg,qTop,qBot,psiTop,psiBot):
28
29     # Basic properties:
30     C=vg.CFun(psi,p)
31
32     # initialize vectors:
33     q=np.zeros(n+1)
34
35     # Upper boundary
36     if qTop == []:
37         KTop=vg.KFun(np.zeros(1)+psiTop,p)
38         q[n]=-KTop*((psiTop-psi[n-1])/dz*2+1)
39     else:
40         q[n]=qTop
41
42     # Lower boundary
43     if qBot == []:
44         if psiBot == []:
45             # Free drainage
46             KBot=vg.KFun(np.zeros(1)+psi[0],p)
47             q[0]=-KBot
```

```
        else:
            # Type 1 boundary
            KBot=vg.KFun(np.zeros(1)+psiBot,p)
            q[0]=-KBot*((psi[0]-psiBot)/dz*2+1.0)
    else:
        # Type 2 boundary
        q[0]=qBot

    # Internal nodes
    i=np.arange(0,n-1)
    Knodes=vg.KFun(psi,p)
    Kmid=(Knodes[i+1]+Knodes[i])/2.0

    j=np.arange(1,n)
    q[j]=-Kmid*((psi[i+1]-psi[i])/dz+1.0)



    # Continuity
    i=np.arange(0,n)
    dpsidt=(-(q[i+1]-q[i])/dz)/C

    return dpsidt
```

```
# Richards equation solver
# This is a function that calculated the right hand side of Richards' equation. You
# will not need to modify this function, unless you are doing something advanced.
# This block of code must be executed so that the function can be later called.

def RichardsModelTransient(psi,t,dz,n,p,vg,qTfun,qBot,psiTop,psiBot):

    # Basic properties:
    C=vg.CFun(psi,p)

    # initialize vectors:
    q=np.zeros(n+1)

    if t>100:
        q[n]=qTfun(100)
    else:
        q[n]=qTfun(t)

    # Lower boundary
    if qBot == []:
        if psiBot == []:
            # Free drainage
            KBot=vg.KFun(np.zeros(1)+psi[0],p)
            q[0]=-KBot
        else:
            # Type 1 boundary
            KBot=vg.KFun(np.zeros(1)+psiBot,p)
            q[0]=-KBot*((psi[0]-psiBot)/dz*2+1.0)
    else:
        # Type 2 boundary
        q[0]=qBot

    # Internal nodes
    i=np.arange(0,n-1)
    Knodes=vg.KFun(psi,p)
    Kmid=(Knodes[i+1]+Knodes[i])/2.0

    j=np.arange(1,n)
    q[j]=-Kmid*((psi[i+1]-psi[i])/dz+1.0)

    # Continuity
    i=np.arange(0,n)
    dpsidt=(-(q[i+1]-q[i])/dz)/C

    return dpsidt
```

```
psi = np.linspace(-10,1)
theta = vg.thetaFun(psi,p)
C=vg.CFun(psi,p)
K=vg.KFun(psi,p)

pl.rcParams['figure.figsize'] = (5.0, 10.0)
pl.subplot(311)
pl.plot(psi,theta)
pl.ylabel(r'$\theta$', fontsize=20)
pl.subplot(312)
pl.plot(psi,C)
pl.ylabel(r'$C$',fontsize=20)
pl.subplot(313)
pl.plot(psi,K)
pl.ylabel(r'$K$', fontsize=20)
pl.xlabel(r'$\psi$', fontsize=20)

# This block of code sets up and runs the model
# Boundary conditions
qTop=-0.001 #infiltration flux
qBot=[]
psiTop=[]
psiBot=[] #Bottom pressure head, 0 = closed drainage and empty '[]' means free drainage

# Grid in space
dz=0.05 #Datappoints at every 5 cm
ProfileDepth=5 # till a depth of 5 meters
z=np.arange(dz/2.0,ProfileDepth,dz)
n=z.size

# Grid in time
t = np.linspace(0,10,1500) #time period of 10 days of 1500 time steps i.e; approx 150 datapoints a day

# Initial conditions
psi0=-z #Hydrostatic initial conditions

# Solve
psi=odeint(RichardsModel,psi0,t,args=(dz,n,p,vg,qTop,qBot,psiTop,psiBot),mxstep=5000000);

print ("Model run successfully")
```

```
# Post process model output to get useful information

# Get water content
theta=vg.thetaFun(psi,p)

# Get total profile storage
S=theta.sum(axis=1)*dz

# Get change in storage [dVol]
dS=np.zeros(S.size)
dS[1:]=np.diff(S)/(t[1]-t[0])

# Get infiltration flux
if qTop == []:
    KTop=vg.KFun(np.zeros(1)+psiTop,p)
    qI=-KTop*((psiTop-psi[:,n-1])/dz*2+1)
else:
    qI=np.zeros(t.size)+qTop

# Get discharge flux
if qBot == []:
    if psiBot == []:
        # Free drainage
        KBot=vg.KFun(psi[:,0],p)
        qD=-KBot
    else:
        # Type 1 boundary
        KBot=vg.KFun(np.zeros(1)+psiBot,p)
        qD=-KBot*((psi[:,0]-psiBot)/dz*2+1.0)
else:
    qD=np.zeros(t.size)+qBot

    # Plot vertical profiles
pl.rcParams['figure.figsize'] = (10.0, 10.0)
for i in range(0,t.size-1):
    pl.subplot(121)
    pl.plot(psi[i,:],z)
    pl.subplot(122)
    pl.plot(theta[i,:],z)
```

```
# Post process model output to get useful information

# Get water content
theta=vg.thetaFun(psi,p)

# Get total profile storage
S=theta.sum(axis=1)*dz

# Get change in storage [dVol]
dS=np.zeros(S.size)
dS[1:]=np.diff(S)/(t[1]-t[0])

# Get infiltration flux
if qTop == []:
    KTop=vg.KFun(np.zeros(1)+psiTop,p)
    qI=-KTop*((psiTop-psi[:,n-1])/dz*2+1)
else:
    qI=np.zeros(t.size)+qTop

# Get discharge flux
if qBot == []:
    if psiBot == []:
        # Free drainage
        KBot=vg.KFun(psi[:,0],p)
        qD=-KBot
    else:
        # Type 1 boundary
        KBot=vg.KFun(np.zeros(1)+psiBot,p)
        qD=-KBot*((psi[:,0]-psiBot)/dz*2+1.0)
else:
    qD=np.zeros(t.size)+qBot

    # Plot vertical profiles
pl.rcParams['figure.figsize'] = (10.0, 10.0)
for i in range(0,t.size-1):
    pl.subplot(121)
    pl.plot(psi[i,:],z)
    pl.subplot(122)
    pl.plot(theta[i,:],z)

pl.subplot(121)
pl.ylabel('Elevation [m]',fontsize=20)
pl.xlabel(r'$\psi$ [m]',fontsize=20)
pl.subplot(122)
pl.xlabel(r'$\theta$ [-]',fontsize=20)

# Plot timeseries
dt = t[2]-t[1]
pl.plot(t,dS,label='Rate of change in storage')
#pl.hold(True)
pl.plot(t,-qI,label='Infiltration')
pl.plot(t,-qD,label='Discharge')
pl.legend(loc="Upper Left")
pl.ylim((0,0.02))

t=np.arange(0,101,1)
qT=np.zeros(len(t))-0.01
print(qT)
```

3. Code for LSTM – *Figure 30, 31 & 32.*

4. PINN – code for Interpolation problem *(Integrated – as values of K, C and θ were calculated in the code)*

```
import sys
sys.path.insert(0, '../../Utilities/')

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
from scipy.interpolate import griddata
from plotting import newfig, savefig
from mpl_toolkits.axes_grid1 import make_axes_locatable
import matplotlib.gridspec as gridspec
from scipy.interpolate import interp1d
import time

np.random.seed(1234)
tf.set_random_seed(1234)

def thetaFun(psi,pars): #water content
    if psi>=0.:
        Se = 1.
    else:
        Se=(1+abs(psi*pars['alpha'])**pars['n'])**(-pars['m'])
    return pars['thetaR']+(pars['thetaS']-pars['thetaR'])*Se

thetaFun=np.vectorize(thetaFun)

def CFun(psi,pars): #water storage function
    if psi>=0.:
        Se=1.
    else:
        Se=(1+abs(psi*pars['alpha'])**pars['n'])**(-pars['m'])
    dSedh=pars['alpha']*pars['m']/(1-pars['m'])*Se**(1/pars['m'])*(1-Se**(1/pars['m']))**pars['m']
    return Se*pars['Ss']+(pars['thetaS']-pars['thetaR'])*dSedh

CFun = np.vectorize(CFun)

def KFun(psi,pars): #hydraulic conductivity
    if psi>=0.:
        Se=1.
    else:
        Se=(1+abs(psi*pars['alpha'])**pars['n'])**(-pars['m'])
    return pars['Ks']*Se**pars['neta']*(1-(1-Se**(1/pars['m']))**pars['m'])**2

KFun = np.vectorize(KFun)

from sympy import symbols, diff
psi, alpha, n, m, neta, Ks = symbols ('psi alpha n m neta Ks', real = True)
f=Ks*((1+abs(psi*alpha)**n)**(-m))**neta*(1-(1-((1+abs(psi*alpha)**n)**(m))**(1/m))**m)**2
dKdp=diff(f,psi)
```

```
def dKdpFun(psi,pars):
    alpha = float()
    n = float()
    m = float()
    neta = float()
    Ks = float()
    alpha = pars['alpha']
    n = pars['n']
    m = pars['m']
    neta = pars['neta']
    Ks = pars['Ks']
    dKdp = float()
    return dKdp

dKdpFun = np.vectorize(dKdpFun)
```

```
def setpars():
  pars={}
  pars['thetaR']=float(raw_input("thetaR = "))
  pars['thetaS']=float(raw_input("thetaS = "))
  pars['alpha']=float(raw_input("alpha = "))
  pars['n']=float(raw_input("n = "))
  pars['m']=1-1/pars['n']
  pars['Ks']=float(raw_input("Ks = "))
  pars['neta']=float(raw_input("neta = "))
  pars['Ss']=float(raw_input("Ss = "))
  return pars

def PlotProps(pars):
  psi=np.linspace(-10,2,200)
  plt.figure
  plt.subplot(3,1,1)
  plt.plot(psi,thetaFun(psi,pars))
  plt.ylabel(r'$\theta(\psi) [-]$')
  plt.subplot(3,1,2)
  plt.plot(psi,CFun(psi,pars))
  plt.ylabel(r'$C(\psi) [1/m]$')
  plt.subplot(3,1,3)
  plt.plot(psi,KFun(psi,pars))
  plt.xlabel(r'$\psi [m]$')
  plt.ylabel(r'$K(\psi) [m/d]$')
  #pl.show()

def HygieneSandstone():
  pars={}
  pars['thetaR']=0.153
  pars['thetaS']=0.25
  pars['alpha']=0.79
  pars['n']=10.4
  pars['m']=1-1/pars['n']
  pars['Ks']=1.08
  pars['neta']=0.5
  pars['Ss']=0.000001
  return pars

p=HygieneSandstone()

class PhysicsInformedNN:
    # Initialize the class
    def __init__(self, X, u, layers, lb, ub, fK, fdKdp, fC):

        self.lb = lb
        self.ub = ub

        self.x = X[:,0:1]
        self.t = X[:,1:2]
        self.u = u

        self.layers = layers

        # Initialize NNs
        self.weights, self.biases = self.initialize_NN(layers)

        # tf placeholders and graph
        self.sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
                                                     log_device_placement=True))
```

```
        # Initialize parameters
        self.fK = fK
        self.fdKdp = fdKdp
        self.fC = fC

        self.x_tf = tf.placeholder(tf.float32, shape=[None, self.x.shape[1]])
        self.t_tf = tf.placeholder(tf.float32, shape=[None, self.t.shape[1]])
        self.u_tf = tf.placeholder(tf.float32, shape=[None, self.u.shape[1]])

        self.u_pred = self.net_u(self.x_tf, self.t_tf)
        self.f_pred = self.net_f(self.x_tf, self.t_tf)

        self.loss = tf.reduce_mean(0.9*tf.square(self.u_tf - self.u_pred)) + \
                    0.1*tf.reduce_mean(tf.square(self.f_pred))

        self.optimizer = tf.contrib.opt.ScipyOptimizerInterface(self.loss,
                                                                 method = 'L-BFGS-B',
                                                                 options = {'maxiter': 50000,
                                                                            'maxfun': 50000,
                                                                            'maxcor': 50,
                                                                            'maxls': 50,
                                                                            'ftol' : 1.0 * np.finfo(float).eps})

        self.optimizer_Adam = tf.train.AdamOptimizer()
        self.train_op_Adam = self.optimizer_Adam.minimize(self.loss)

        init = tf.global_variables_initializer()
        self.sess.run(init)

    def initialize_NN(self, layers):
        weights = []
        biases = []
        num_layers = len(layers)
        for l in range(0,num_layers-1):
            W = self.xavier_init(size=[layers[l], layers[l+1]])
            b = tf.Variable(tf.zeros([1,layers[l+1]], dtype=tf.float32), dtype=tf.float32)
            weights.append(W)
            biases.append(b)
        return weights, biases

    def xavier_init(self, size):
        in_dim = size[0]
        out_dim = size[1]
        xavier_stddev = np.sqrt(2/(in_dim + out_dim))
        return tf.Variable(tf.truncated_normal([in_dim, out_dim], stddev=xavier_stddev), dtype=tf.float32)

    def neural_net(self, X, weights, biases):
        num_layers = len(weights) + 1

        H = 2.0*(X - self.lb)/(self.ub - self.lb) - 1.0
        for l in range(0,num_layers-2):
            W = weights[l]
            b = biases[l]
            H = tf.tanh(tf.add(tf.matmul(H, W), b))
        W = weights[-1]
        b = biases[-1]
        Y = tf.add(tf.matmul(H, W), b)
        return Y
```

```
    def net_u(self, x, t):
        u = self.neural_net(tf.concat([x,t],1), self.weights, self.biases)
        return u

    def net_f(self, x, t):
        u = self.net_u(x,t)
        #fK = self.fK

        def fn(m):
            fK = self.fK
            return fK(m).astype(np.float32)
        #fdKdp = self.fdKdp
        K_u= tf.py_func(fn, [u], tf.float32)

        def fn1(m):
            fdKdp = self.fdKdp
            return fdKdp(m).astype(np.float32)

        def fn2(m):
            fC = self.fC
            return fC(m).astype(np.float32)

```

```
        K_u= tf.py_func(fn, [u], tf.float32)
        dKdp = tf.py_func(fn1, [u], tf.float32)
        C = tf.py_func(fn2, [u], tf.float32)

        u_t = tf.gradients(u, t)[0]
        u_x = tf.gradients(u, x)[0]
        u_xx = tf.gradients(u_x, x)[0]

        f = C*u_t- dKdp*u_x*u_x -K_u*u_xx - dKdp*u_x

        return f

    def callback(self, loss):
        print('Loss: %e' % (loss))


    def train(self, nIter):
        tf_dict = {self.x_tf: self.x, self.t_tf: self.t, self.u_tf: self.u}

        start_time = time.time()
        for it in range(nIter):
            self.sess.run(self.train_op_Adam, tf_dict)

            # Print
            if it % 10 == 0:
                elapsed = time.time() - start_time
                loss_value = self.sess.run(self.loss, tf_dict)
                f_value = self.sess.run(tf.reduce_mean(tf.square(self.f_pred)))
                print('It: %d, Loss: %.3e, Time: %.2f, f: %.3e' %
                      (it, loss_value, elapsed, f_value))
                start_time = time.time()

        self.optimizer.minimize(self.sess,
                                feed_dict = tf_dict,
                                fetches = [self.loss],
                                loss_callback = self.callback)


    def predict(self, X_star):

        tf_dict = {self.x_tf: X_star[:,0:1], self.t_tf: X_star[:,1:2]}

        u_star = self.sess.run(self.u_pred, tf_dict)
        f_star = self.sess.run(self.f_pred, tf_dict)

        return u_star, f_star
```

```
if __name__ == "__main__":

    N_u = 500
    layers = [2, 20, 20, 20, 20, 20, 20, 1]

    data = scipy.io.loadmat('../Data/data4.mat')

    t = data['t'].flatten()[:,None]
    x = data['x'].flatten()[:,None]
    Exact = np.real(data['utot']).T

    psi = np.reshape(np.linspace(-10,0,10000), 10000)
    K = np.reshape(KFun(psi,p), 10000)
    C = np.reshape(CFun(psi,p), 10000)
    dKdp = np.reshape(dKdpFun(psi,p), 10000)

    # Establish interpolation functions
    fK=interp1d(psi,K,bounds_error=False,fill_value=(K[0],K[-1]))
    fdKdp=interp1d(psi,dKdp,bounds_error=False,fill_value=(dKdp[0],dKdp[-1]))
    fC=interp1d(psi,C,bounds_error=False,fill_value=(C[0],C[-1]))
```

```
    X, T = np.meshgrid(x,t)

    X_star = np.hstack((X.flatten()[:,None], T.flatten()[:,None]))
    u_star = Exact.flatten()[:,None]

    # Doman bounds
    lb = X_star.min(0)
    ub = X_star.max(0)

    ###############################################################################
    ######################## Noiseles Data ########################################
    ###############################################################################
    noise = 0.0

    idx = np.random.choice(X_star.shape[0], N_u, replace=False)

    X_u_train = X_star[idx,:]
    u_train = u_star[idx,:]

    model = PhysicsInformedNN(X_u_train, u_train, layers, lb, ub, fK, fdKdp, fC)
    model.train(0)

    u_pred, f_pred = model.predict(X_star)

    error_u = np.linalg.norm(u_star-u_pred,2)/np.linalg.norm(u_star,2)

    U_pred = griddata(X_star, u_pred.flatten(), (X, T), method='cubic')

    ###############################################################################
    ############################### Plotting ######################################
    ###############################################################################

    fig, ax = newfig(1.0, 1.4)
    ax.axis('off')

    ######## Row 0: u(t,x) ##################
    gs0 = gridspec.GridSpec(1, 2)
    gs0.update(top=1-0.06, bottom=1-1.0/3.0+0.06, left=0.15, right=0.85, wspace=0)
    ax = plt.subplot(gs0[:, :])

    h = ax.imshow(U_pred.T, interpolation='nearest', cmap='bwr',
                  extent=[t.min(), t.max(), x.min(), x.max()],
                  origin='lower', aspect='auto')
    divider = make_axes_locatable(ax)
    cax = divider.append_axes("right", size="5%", pad=0.05)
    fig.colorbar(h, cax=cax)
```

```
    ax.plot(X_u_train[:,1], X_u_train[:,0], 'kx', label = 'Data (%d points)' % (u_train.shape[0]), markersize = 2, clip_on = False)

    line = np.linspace(x.min(), x.max(), 2)[:,None]
    ax.plot(t[50]*np.ones((2,1)), line, 'w-', linewidth = 1)
    ax.plot(t[100]*np.ones((2,1)), line, 'w-', linewidth = 1)
    ax.plot(t[150]*np.ones((2,1)), line, 'w-', linewidth = 1)

    ax.set_xlabel('$t$')
    ax.set_ylabel('$x$')
    ax.legend(loc='upper center', bbox_to_anchor=(1.0, -0.125), ncol=5, frameon=False)
    ax.set_title('$u(t,x)$', fontsize = 10)


    ######## Row 2: u(t,x) slices ##################
    gs2 = gridspec.GridSpec(1, 3)
    gs2.update(top=1-1.0/3.0-0.1, bottom=1.0-2.0/3.0, left=0.1, right=0.9, wspace=0.5)

    ax = plt.subplot(gs2[0, 0])
    ax.plot(x,Exact[50,:], 'b-', linewidth = 2, label = 'Exact')
    ax.plot(x,U_pred[50,:], 'r--', linewidth = 2, label = 'Prediction')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$u(t,x)$')
    ax.set_title('$t = 1$', fontsize = 10)
    ax.axis('square')
    ax.set_xlim([-0.1,3.1])
    ax.set_ylim([0.1,-3.1])
```

```
358
359     ax = plt.subplot(gs2[0, 1])
360     ax.plot(x,Exact[100,:], 'b-', linewidth = 2, label = 'Exact')
361     ax.plot(x,U_pred[100,:], 'r--', linewidth = 2, label = 'Prediction')
362     ax.set_xlabel('$x$')
363     ax.set_ylabel('$u(t,x)$')
364     ax.axis('square')
365     ax.set_xlim([-0.1,3.1])
366     ax.set_ylim([0.1,-3.1])
367     ax.set_title('$t = 2$', fontsize = 10)
368     ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.35), ncol=5, frameon=False)
369
370     ax = plt.subplot(gs2[0, 2])
371     ax.plot(x,Exact[150,:], 'b-', linewidth = 2, label = 'Exact')
372     ax.plot(x,U_pred[150,:], 'r--', linewidth = 2, label = 'Prediction')
373     ax.set_xlabel('$x$')
374     ax.set_ylabel('$u(t,x)$')
375     ax.axis('square')
376     ax.set_xlim([-0.1,3.1])
377     ax.set_ylim([0.1,-3.1])
378     ax.set_title('$t = 3$', fontsize = 10)
379
380     fig.savefig(r'C:\Users\saket\.spyder-py3\Trial runs\data\Interpolation_Result.png',dpi=1200)
```

5. PINN – code for Inference

```
12 import tensorflow as tf
13 #import tensorflow_probability as tfp
14 import numpy as np
15 import matplotlib.pyplot as plt
16 import scipy.io
17 from scipy.interpolate import griddata
18 from pyDOE import lhs
19 from plotting import newfig, savefig
20 from mpl_toolkits.mplot3d import Axes3D
21 import time
22 import matplotlib.gridspec as gridspec
23 from mpl_toolkits.axes_grid1 import make_axes_locatable
24 from scipy.interpolate import interp1d
25
26 np.random.seed(1234)
27 tf.set_random_seed(1234)
28
29 class PhysicsInformedNN:
30     # Initialize the class
31     def __init__(self, X_u, u, X_f, layers, lb, ub, nu, fK, fdKdp, fC):
32
33         self.lb = lb
34         self.ub = ub
35
36         self.x_u = X_u[:,0:1]
37         self.t_u = X_u[:,1:2]
38         self.x_f = X_f[:,0:1]
39         self.t_f = X_f[:,1:2]
40         self.u = u
41
42         self.layers = layers
43         self.nu = nu
44
45         self.fK = fK
46         self.fdKdp = fdKdp
47         self.fC = fC
48
49         # Initialize NNs
50         self.weights, self.biases = self.initialize_NN(layers)
```

```
51
52          # tf placeholders and graph
53          self.sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
54
55                                                       log_device_placement=True))
56
57          self.x_u_tf = tf.placeholder(tf.float32, shape=[None, self.x_u.shape[1]])
58          self.t_u_tf = tf.placeholder(tf.float32, shape=[None, self.t_u.shape[1]])
59
60          self.u_tf = tf.placeholder(tf.float32, shape=[None, self.u.shape[1]])
61          self.x_f_tf = tf.placeholder(tf.float32, shape=[None, self.x_f.shape[1]])
62          self.t_f_tf = tf.placeholder(tf.float32, shape=[None, self.t_f.shape[1]])
63
64          self.u_pred = self.net_u(self.x_u_tf, self.t_u_tf)
65          self.f_pred = self.net_f(self.x_f_tf, self.t_f_tf)
66
67          self.loss = tf.reduce_mean(0.5*tf.square(self.u_tf - self.u_pred)) + \
68          0.5*tf.reduce_mean(tf.square(self.f_pred))
69
70          self.optimizer = tf.contrib.opt.ScipyOptimizerInterface(self.loss,
71                                                                  method = 'L-BFGS-B',
72                                                                  options = {'maxiter': 50000,
73                                                                             'maxfun': 50000,
74                                                                             'maxcor': 50,
75                                                                             'maxls': 50,
76                                                                             'ftol' : 1.0 * np.finfo(float).eps})
77
```

```
78          init = tf.global_variables_initializer()
79          self.sess.run(init)
80
81      def initialize_NN(self, layers):
82          weights = []
83          biases = []
84          num_layers = len(layers)
85
86          for l in range(0,num_layers-1):
87              W = self.xavier_init(size=[layers[l], layers[l+1]])
88              b = tf.Variable(tf.zeros([1,layers[l+1]], dtype=tf.float32), dtype=tf.float32)
89              weights.append(W)
90              biases.append(b)
91          return weights, biases
92
93      def xavier_init(self, size):
94          in_dim = size[0]
95          out_dim = size[1]
96          xavier_stddev = np.sqrt(2/(in_dim + out_dim))
97          return tf.Variable(tf.truncated_normal([in_dim, out_dim], stddev=xavier_stddev), dtype=tf.float32)
98
99      def neural_net(self, X, weights, biases):
100         num_layers = len(weights) + 1
101         H = 2.0*(X - self.lb)/(self.ub - self.lb) - 1.0
102
103         for l in range(0,num_layers-2):
104             W = weights[l]
105             b = biases[l]
106             H = tf.tanh(tf.add(tf.matmul(H, W), b))
107         W = weights[-1]
108         b = biases[-1]
109         Y = tf.add(tf.matmul(H, W), b)
110         return Y
111
112     def net_u(self, x, t):
113         u = self.neural_net(tf.concat([x,t],1), self.weights, self.biases)
114         return u
115
116     def net_f(self, x,t):
117         u = self.net_u(x,t)
118         #fK = self.fK
119
120         def fn(m):
121             fK = self.fK
122             return fK(m).astype(np.float32)
123         #fdKdp = self.fdKdp
124
125         K_u= tf.py_func(fn, [u], tf.float32)
126
127         def fn1(m):
128             fdKdp = self.fdKdp
129             return fdKdp(m).astype(np.float32)
130
131         def fn2(m):
132             fC = self.fC
133             return fC(m).astype(np.float32)
134
```

```
        K_u= tf.py_func(fn, [u], tf.float32)
        dKdp = tf.py_func(fn1, [u], tf.float32)
        C = tf.py_func(fn2, [u], tf.float32)

        # Evaluate K
        #K_u=tf.numpy_function(myfunc, u, [tf.float32], name='myfunc')
        #dKdp_u=fdKdp(u_np)

        # COnvert back to tensors
        #K_T=tf.convert_to_tensor(K_u)
        #dKdp_T=tf.convert_to_tensor(dKdp_u)
        #K_u = tf.gradients(K, u)[0]
        u_t = tf.gradients(u, t)[0]
        u_x = tf.gradients(u, x)[0]
        u_xx = tf.gradients(u_x, x)[0]
        #f = u_t + u*u_x - self.nu*u_xx
        f = C*u_t- dKdp*u_x*u_x -K_u*u_xx - dKdp*u_x
        #f = u_t - K_u*u_xx
        return f

    def callback(self, loss):
        print('Loss:', loss)

    def train(self):
        tf_dict = {self.x_u_tf: self.x_u, self.t_u_tf: self.t_u, self.u_tf: self.u,
                   self.x_f_tf: self.x_f, self.t_f_tf: self.t_f}

        self.optimizer.minimize(self.sess,
                                feed_dict = tf_dict,
                                fetches = [self.loss],
                                loss_callback = self.callback)

    def predict(self, X_star):
        u_star = self.sess.run(self.u_pred, {self.x_u_tf: X_star[:,0:1], self.t_u_tf: X_star[:,1:2]})
        f_star = self.sess.run(self.f_pred, {self.x_f_tf: X_star[:,0:1], self.t_f_tf: X_star[:,1:2]})
        return u_star, f_star
```

```
if __name__ == "__main__":

    #nu = 0.01/np.pi
    noise = 0.0
    nu=0.0002628460175498463

    N_u = 100
    N_f = 4000

    layers = [2, 20, 20, 1]

    data = scipy.io.loadmat('../Data/data4.mat')
    t = data['t'].flatten()[:,None]
    x = data['x'].flatten()[:,None]
    Exact = np.real(data['utot']).T

    # Extract data for the permeability function and its derivative
    psi = np.reshape(data['psi'], (len(data['psi'])))
    K = np.reshape(data['K'], (len(data['K'])))
    C = np.reshape(data['C'], (len(data['C'])))
    dKdp = np.reshape(data['dKdp'], (len(data['dKdp'])))

    # Establish interpolation functions
    fK=interp1d(psi,K,bounds_error=False,fill_value=(K[0],K[-1]))
    fdKdp=interp1d(psi,dKdp,bounds_error=False,fill_value=(dKdp[0],dKdp[-1]))
    fC=interp1d(psi,C,bounds_error=False,fill_value=(C[0],C[-1]))

    X, T = np.meshgrid(x,t)
    X_star = np.hstack((X.flatten()[:,None], T.flatten()[:,None]))
    u_star = Exact.flatten()[:,None]

    # Doman bounds
    lb = X_star.min(0)
    ub = X_star.max(0)
```

```
    xx1 = np.hstack((X[0:1,:].T, T[0:1,:].T))
    uu1 = Exact[0:1,:].T
    xx2 = np.hstack((X[:,0:1], T[:,0:1]))
    uu2 = Exact[:,0:1]
    xx3 = np.hstack((X[:,-1:], T[:,-1:]))
    uu3 = Exact[:,-1:]

    X_u_train = np.vstack([xx1, xx2, xx3])
    X_f_train = lb + (ub-lb)*lhs(2, N_f)
    X_f_train = np.vstack((X_f_train, X_u_train))
    u_train = np.vstack([uu1, uu2, uu3])

    idx = np.random.choice(X_u_train.shape[0], N_u, replace=False)
    X_u_train = X_u_train[idx, :]
    u_train = u_train[idx,:]

    model = PhysicsInformedNN(X_u_train, u_train, X_f_train, layers, lb, ub, nu, fK, fdKdp, fC)

    start_time = time.time()
    model.train()
    elapsed = time.time() - start_time
    print('Training time: %.4f' % (elapsed))

    u_pred, f_pred = model.predict(X_star)

    error_u = np.linalg.norm(u_star-u_pred,2)/np.linalg.norm(u_star,2)
    print('Error u: %e' % (error_u))


    U_pred = griddata(X_star, u_pred.flatten(), (X, T), method='cubic')
    Error = np.abs(Exact - U_pred)
```

```
    ############################# Plotting ###############################

    ######################################################################

    fig, ax = newfig(1.0, 1.1)
    ax.axis('off')

    ######## Row 0: u(t,x) ##################

    gs0 = gridspec.GridSpec(1, 2)
    gs0.update(top=1-0.06, bottom=1-1/3, left=0.15, right=0.85, wspace=0)
    ax = plt.subplot(gs0[:, :])

    h = ax.imshow(U_pred.T, interpolation='nearest', cmap='bwr',
                  extent=[t.min(), t.max(), x.min(), x.max()],
                  origin='lower', aspect='auto')
    divider = make_axes_locatable(ax)
    cax = divider.append_axes("right", size="5%", pad=0.05)
    fig.colorbar(h, cax=cax)

    ax.plot(X_u_train[:,1], X_u_train[:,0], 'kx', label = 'Data (%d points)' % (u_train.shape[0]), markersize = 4, clip_on = False)
```

```
    line = np.linspace(x.min(), x.max(), 2)[:,None]
    ax.plot(t[50]*np.ones((2,1)), line, 'w-', linewidth = 1)
    ax.plot(t[100]*np.ones((2,1)), line, 'w-', linewidth = 1)
    ax.plot(t[150]*np.ones((2,1)), line, 'w-', linewidth = 1)
    ax.set_xlabel('$t$')
    ax.set_ylabel('$x$')
    ax.legend(frameon=False, loc = 'best')
    ax.set_title('$u(t,x)$', fontsize = 10)

    ######## Row 1: u(t,x) slices ##################

    gs1 = gridspec.GridSpec(1, 3)
    gs1.update(top=1-1/3, bottom=0, left=0.1, right=0.9, wspace=0.5)

    ax = plt.subplot(gs1[0, 0])
    ax.plot(x,Exact[50,:], 'b-', linewidth = 2, label = 'Exact')
    ax.plot(x,U_pred[50,:], 'r--', linewidth = 2, label = 'Prediction')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$u(t,x)$')
    ax.set_title('$t = 1$', fontsize = 10)
    ax.axis('square')
    ax.set_xlim([-0.1,3.1])
    ax.set_ylim([0.1,-3.1])
```

```
    ax = plt.subplot(gs1[0, 1])
    ax.plot(x,Exact[100,:], 'b-', linewidth = 2, label = 'Exact')
    ax.plot(x,U_pred[100,:], 'r--', linewidth = 2, label = 'Prediction')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$u(t,x)$')
    ax.axis('square')
    ax.set_xlim([-0.1,3.1])
    ax.set_ylim([0.1,-3.1])
    ax.set_title('$t = 2$', fontsize = 10)
    ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.35), ncol=5, frameon=False)

    ax = plt.subplot(gs1[0, 2])
    ax.plot(x,Exact[150,:], 'b-', linewidth = 2, label = 'Exact')
    ax.plot(x,U_pred[150,:], 'r--', linewidth = 2, label = 'Prediction')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$u(t,x)$')
    ax.axis('square')
    ax.set_xlim([-0.1,3.1])
    ax.set_ylim([0.1,-3.1])
    ax.set_title('$t = 3$', fontsize = 10)

    fig.savefig(r'C:\Users\saket\.spyder-py3\Trial runs\data\Inference.png',dpi=600)
```

NTNU
Norwegian University of
Science and Technology

KLIMA
DIGITAL°