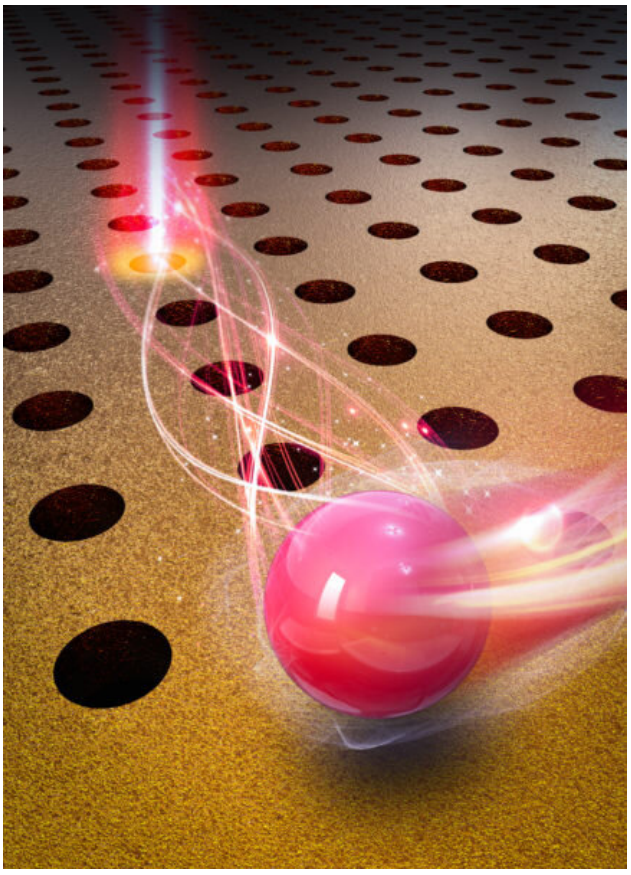
CHALMERS
UNIVERSITY OF TECHNOLOGY



Deep Learning for Optical Tweezers

DeepCalib Implementation for Brownian Motion with Delayed Feedback

Master's thesis in Complex Adaptive Systems

YANUAR RIZKI PAHLEVI

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022
www.chalmers.se

Master's thesis 2022

Deep Learning for Optical Tweezers

DeepCalib Implementation for Brownian Motion with Delayed Feedback

YANUAR RIZKI PAHLEVI

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Complex Adaptive Systems Master Program
Soft Matter Lab
Chalmers University of Technology
Gothenburg, Sweden 2022

Deep Learning for Optical Tweezers
DeepCalib Implementation for Brownian Motion with Delayed Feedback
YANUAR RIZKI PAHLEVI

Supervisor: Aykut Argun, Physics Department
Examiner: Giovanni Volpe, Physics Department

Master's Thesis 2022
Department of Physics
Complex Adaptive Systems Master Program
Soft Matter Lab
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration of nano-tweezers. Figure Reproduce from: https://phys.org/news/2020-09-micron-scale-optical-tweezers.html

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2022

Deep Learning for Optical Tweezers
DeepCalib Implementation for Brownian Motion with Delayed Feedback
YANUAR RIZKI PAHLEVI
Department of Physics
Chalmers University of Technology

Abstract

Brownian motion with delayed feedback, theoretically studied to take control of Brownian particle movement's direction. One can use optical tweezers to implement delayed feedback. Calibrating optical tweezers with delay implemented is not an easy job. In this study, Deep learning technique using Long Short Term Memory (LSTM) layer as main composition of the model to calibrate the trap stiffness and to measure the delayed feedback employed, using the trapped particle trajectory as an input. We demonstrate that this approach is outperforming approximation method in order to calibrate stiffness and to measure the delay in harmonic trap case.

Keywords: Deep Learning, Optical Tweezers, Delayed Feedback

Acknowledgements

I would like to thank Swedish Institute, without them, I am not gonna able to pursue master's degree. To Giovanni Volpe and Aykut Argun for the discussions, supervision and constructive feedback for this project. I would also like to give my warmest gratitude to my wife and sons, who patiently waiting in Indonesia and for my whole family for your support. All of your prayers is what make me able to sustained in this journey. Lastly, to all my friends in Gothenburg. Indonesian Student Union for every event that cheer me up. My food hunter group for every discussion almost every weekend. And to all students in Complex Adaptive Systems program, finally, we manage to endure the weirdest two years, and thank you for every discussion especially about our courses.

Yanuar Rizki Pahlevi, Gothenburg, June 2022

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

ACF	Auto-correlation Function
GRU	Gated Recurrent Unit
LSTM	Long Short-Term Memory
MSD	Mean Squared Displacement
MSE	Mean Squared Error
RNN	Recurrent Neural Network
TSD	Total Squared Displacement

Nomenclature

Below is the nomenclature of indices, parameters, and variables that have been used throughout this thesis.

Indices

i	Indices for iteration
t	Index for time step

Parameters

γ	Friction coefficient
δ	Delay
η	Viscosity
τ	Time scale
b	Bias
f_w	Activation Function
$h(t)$	Hidden state of RNN at time t
k_B	Boltzmann Constant
T	Temperature
W	Weight

Variables

F	Force
k	Stiffness
ω	Torque
$\mathbf{r}$	2D Position
$x(t)$	Position at time t

$\xi(t)$	Gaussian random noise at time t
$v(t)$	Velocity at time t

Contents

List of Figures

List of Algorithms

1

Introduction

The basic concept of optical tweezers was introduced by Arthur Ashkin in 1986, where he demonstrated that a focused laser beam can be used to stably trap micrometer-sized particle, which later this trapped particle can be used as a cantilever to probe microscopic force fields. As a device, optical tweezers can be used in many fields of physics and biology.

Despite how useful optical tweezers can be, most of the strategies used to observe it's potential are not giving satisfying results. Most strategies using top-down approach where everything start with human intuition on data analysis and then the idea is tested in experiment or simulation.

In this thesis work, we are trying to use the opposite approach. But only just a small part of the big idea of it. While common strategy using top-down thinking, here we are trying to understand the phenomena using bottom-up thinking where we start by analyzing the trapped particle.

The trapped particle in optical tweezers is not staying in the same position. It is moving due to the Brownian fluctuations. But the movement is always around the equilibrium point which depends on the trap type used. The movement also depends on how strong is the optical force used to trap the particle.

In optical tweezers, calibrating how strong the force is one of an important task. Succeed in this task will allow one to generate optical potentials optimized for specific applications. In order to calibrate the force or the stiffness of the optical tweezers, there are several methods available such as equipartition theorem, potential analysis, mean squared displacement (MSD) analysis, etc [1]. Despite the advantages of each method, we will face a problem if the force fields are non-conservative. To tackle this problem, a study to use deep learning to calibrate the stiffness of optical tweezers has been done and the results are more accurate than aforementioned methods[2].

Another application using Brownian motion as fundamental theory is navigating active particle towards desired direction. Monitoring particle's orientation of motion in a real time condition is experimentally impossible. Theoretical study to propose an approach to steer particle's movement to desired direction using optical tweezers with delayed position feedback gives promising result [3]. With delayed feedback,

the next position of the particles determine by the position of earlier time x(t - δ). Optical tweezers with delayed feedback system is the basic ingredient of this thesis project. But, if the mentioned study analyzing from theoretical side whether applying feedback gives control to steer the trapped particle orientation or not, here we analyze the trajectory of trapped particle in optical tweezers with delayed position feedback for two objectives:

- To calibrate the optical tweezers
- To predict the implemented delay

This thesis report organized as follows. In section 2, we provide some theories which fundamental for this thesis project. Then in section 3, we explain the methods that used in order to achieve our objectives. Thereafter, we provide the results of those methods in section 4 and discuss it in section 5. Finally, we conclude this project in section 6.

2

Theory

In this chapter we will give a brief explanation of some basic theory which are the foundation of this thesis. We will start with the definition and simple example of delayed-feedback system, then we will continue with basic understanding of optical tweezers, Brownian motion and some brief explanation about machine and deep learning.

2.1 Delayed-Feedback System

The most simple example of delayed-feedback system is our air conditioner system at home or at our office. First, air conditioner have temperature target which set by the user. If its environment temperature is not the same with the target, then the machine will give response depends on the differences. If the environment's temperature is colder than the target, then the air conditioner will temporarily turn off, the opposite, if the environment's temperature is higher than the target, the air conditioner will turn on.

When the environment's temperature reach the air conditioner temperature target, the air conditioner will automatically turn off until the environment's temperature change and the same loop will be repeated as long as the air conditioner is in stand by mode condition.

But, if we take a look more details, the air conditioner do not directly turn off when the temperature is reached. It is still working to decrease environment's temperature until some time, and then it will turn off. This 'some time' is what we called delay, and this system is one simple example of delayed-feedback system.

In control system field, delayed-feedback is very useful. One unstable dynamical system can be stabilized using delayed-feedback system by applying feedback perturbation proportional to the deviation of the current state of the system from its state in the past with some time difference [4].

2.2 Optical Tweezers

Since the Brownian motion discussed in this thesis is happened in an optical trap, it will be a benefit to put some basic information about optical tweezers. Optical tweezers is a focused laser beam that can trap microscopic particles. We can build optical tweezers by focusing a laser through a microscope. Optical tweezers set up is consist of three parts: trapping, imaging and position detection [1]. Laser beam used in optical tweezers is continuous wave with wavelength in the visible or near infrared region spectrum. The power of laser beam is between 10 mW and 1W [1].

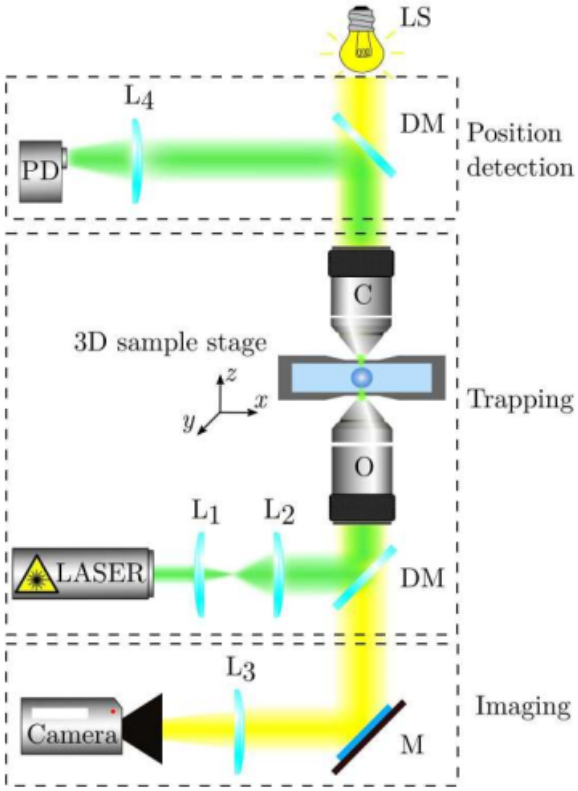


Figure 2.1: Basic Optical Tweezers Setup. Figure reproduced from Ref. [1]

A basic optical tweezers setup are consist of three parts; trapping, imaging and position detection. An example of basic optical tweezers setup can be seen in Fig. 2.1[1]. In the trapping section, we can see that the laser beam is expanded by lenses (L1, L2), and then directed by a dichroic mirror towards the objective. This incoming laser beam then focused by microscope objective (O), this focal point is the tweezers that can trap particle. As for imaging part, the concept is similar with trapping part, but now we use light source (LS) so the camera can get clear view for recording with enough lighting. To be able to use the trapped particle as a microscopic force transducer, it is necessary to track its position. We can see the particle's movement using recorded image or video, or we can use position detector (PD) which yield the time series of particle's position. Position detector detect

particle's position trough condenser (C) which collects the forward scattered light. The changes in scattered light pattern is due to particle's movement[1].

2.3 Brownian Motion of Trapped Particle

in 1827, while looking through his microscope, Robert Brown observed something interesting in his cavities. Particle trapped inside the cavities, pollen grains in water were moving. During that time, he cannot explain the phenomena behind this movement. This random movement later named after his name; Brownian motion. 80 years later, Albert Einstein published a paper which explain theoretically that what Robert Brown observed was pollen particles moved by water molecules surround it. This published article contained predictions that could be tested experimentally. On 1908, it was Jean Babstite Perrin who perform the experiment and prove Einstein's prediction. This experiment is important since then it is a proof of atom's existence. The same year, Paul Langevin presented a more straight-forward alternative way to analyze Brownian motion[5]

$$\frac{d}{dt}v(t) = -\eta v(t) + \xi(t) \tag{2.1}$$

Where $v(t)$ is particle's velocity at time t, η is medium's viscosity and $\xi(t)$ is Gaussian random noise at time t with mean zero and variance one.

In an optical trap, the trapped particle is in dynamic equilibrium, where there are two forces works on the trapped particle. The first one, is the thermal noise which pushing the particle out of the optical trap, while the optical trap trying to keep the particle in the equilibrium point of the optical trap. These two forces make the trapped particle not stay in one position at all time. It always move according to Langevin equation.

The movement of Brownian particle in an optical trap will also depend on the type of the optical trap. Here we want to analyze both conservative and non-conservative trap. Thus, we use two types of trap in the simulation for this thesis project; Harmonic trap and Rotational Force trap. The latter is an example of non-conservative force field where there is no straight-forward standard methods to calibrate the force fields.

2.3.1 Harmonic Trap

As mentioned before, a trapped particle in an optical tweezers moves due to Brownian fluctuations where these fluctuations depend on how strong the trap used also the type of the trap. To start with, we will study the trap which there is a straight forward standard method to calibrate the force fields, the Harmonic trap. Without delayed feedback, Brownian particle trapped in a harmonic trap with over damped limit described by [6]:

$$\frac{d}{dt}x(t) = -\frac{k}{\gamma}x(t) + \sqrt{\frac{2k_BT}{\gamma}}\xi(t) \tag{2.2}$$

where x(t) is particle's position with respect to the equilibrium (m), k is the stiffness (N/m), $\gamma$is the friction coefficient (kg/s), T is temperature (K) and ξ(t) is Gaussian random noise with zero mean and variance equal to one. With delayed feedback, the displacement of trapped particles will depend on earlier position. Thus, the Langevin equation modified into:

$$\frac{d}{dt}x(t) = -\frac{k}{\gamma}x(t-\delta) + \sqrt{\frac{2k_BT}{\gamma}}\xi(t) \tag{2.3}$$

Where δ is the delay (s) implemented to the particle. To see the difference between trapped particle trajectory with and without delay, we can see an example of each case on Fig. 2.2. We can see the trajectory looks smoother when we implement delayed feedback. Delayed feedback also loosen the stiffness of the trap to the particle, as we can see that the maximum distance of particle from equilibrium point with delayed feedback is higher than without delayed feedback. But, both cases still shows that the particle's movement converges around the equilibrium point, where in Fig. 2.2 the equilibrium point is at $x = 0$ µm.

2.3.2 Rotational Force Fields

Beside Harmonic trap, we also consider a non-conservative force fields in this thesis project, Rotational force fields which described by the following equation[2]

$$\mathbf{F(r)} = -k\mathbf{r} - \omega(\mathbf{r} \times \hat{\mathbf{z}}) \tag{2.4}$$

Where $\mathbf{r}$ is two-dimensional particle's position subject to a restoring force with stiffness k and torque ω which equal to: $\Omega\gamma$ where Ω is torque's frequency. Thus, the Langevin equation of particle trapped in rotational force fields described as follows:

$$\frac{d}{dt}x(t) = -\frac{\mathbf{F(r)}}{\gamma} + \sqrt{\frac{2k_BT}{\gamma}}\xi(t) \tag{2.5}$$

In rotational force fields, the particle's movement depends on stiffness k and torque ω. With delayed feedback, the Langevin equation modified into:

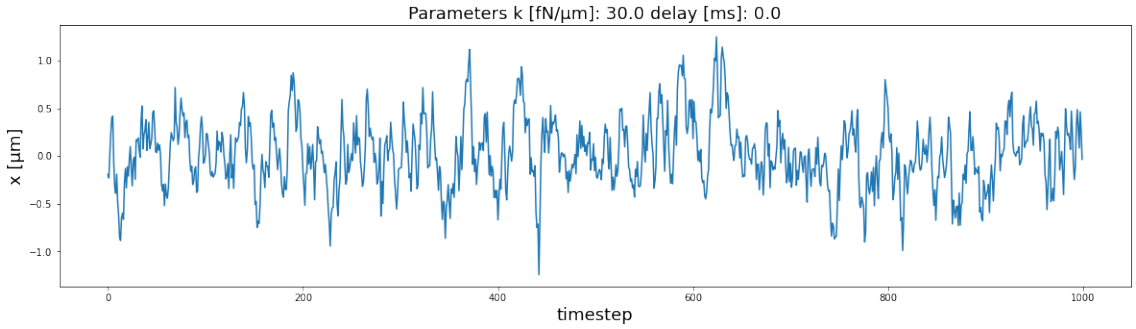
$$\frac{d}{dt}x(t) = -\frac{\mathbf{F(r(t\text{-}\delta))}}{\gamma} + \sqrt{\frac{2k_BT}{\gamma}}\xi(t) \tag{2.6}$$

With

$$\mathbf{F(r(t\text{-}\delta))} = -k\mathbf{r(t\text{-}\delta)} - \omega(\mathbf{r(t\text{-}\delta)} \times \hat{\mathbf{z}}) \tag{2.7}$$

Different from harmonic trap case, in rotational force fields we calibrate k and ω, also measure δ. A method to calibrate the torque ω of Brownian particles is available[7]. But it is to calibrate when there is no delayed feedback implemented.

(a)



(b)

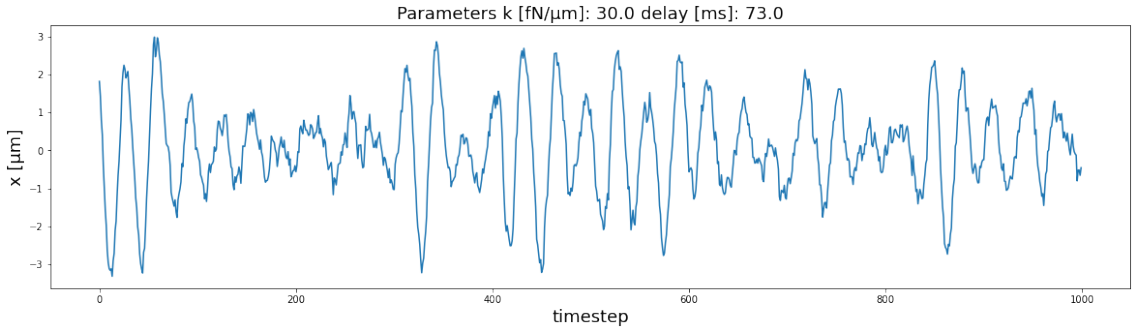


Figure 2.2: Trajectory example of trapped particle with $k = 30$ fN/µm with (a) $\delta = 0$ ms, (b) $\delta = 73$ ms

2.4 Deep Learning

2.4.1 Machine Learning and Deep Learning

If we want to teach a person about the differences between cakes, it will be easier for that person if we give them an example. The example of the cake can be a picture of each cake, or we just present all cakes that we want this person to learn. Because by seeing directly how the cakes looks like, what is the shape or colors of the cake, is easier to remember than if we only tell this person the details ingredients of each cake. By seeing the example of the cakes, people can learn about different type of cakes by recognizing different patterns between cakes.

The basic concept of machine learning is similar to that as well. Instead of codifying the details of knowledge into computers, machine learning technique seeks to automatically recognizing meaningful patterns from examples and observations[8][9]. Even though machine learning technique is a powerful technique, not all problems need to be solved using machine learning technique. Some suggestions on when we should use machine learning to solve our problem are[10]:

- Traditional engineering flow is not applicable or is undesirable due to a model or algorithm deficit.
- Sufficient training data exist or can be generated.
- The task does not require the application of logic, common sense or explicit reasoning.
- The task does not require detailed explanation on how the decision was made

Machine learning employ artificial neural networks in it's learning process. Artificial neural networks is a collection of nodes which consists of three type of layers; 1) Input layer, 2) Hidden layer and 3) Output layer. In basic artificial neural networks, each node on each layer connected to each node on an adjacent layer as shown in Fig. 2.3

The connection between nodes on adjacent layers, has different adjustable weights that will determine the output. The aim of machine learning is minimizing the difference between desired output value with current output value. We minimize the difference by adjusting the value of the weights using backward propagation.

Each node in hidden layer has value. This value determine by the sum of weighted input and bias of each node. This summation will be used as input to some activation function. the calculation on each node is described by the following equation:

$$Node'sValue = f(\sum_{i=0}^{n} x_i W_i + b) \tag{2.8}$$

where f is some activation function, x_i is the value of i-th input, and b is the bias of the hidden layer node. Activation function can be a nonlinear activation function

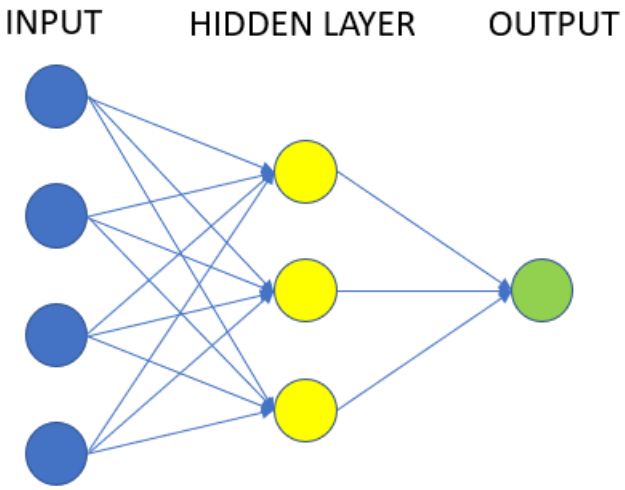


Figure 2.3: Illustration of Fully Connected Neural Network with 4 input and one hidden layer with 3 neurons

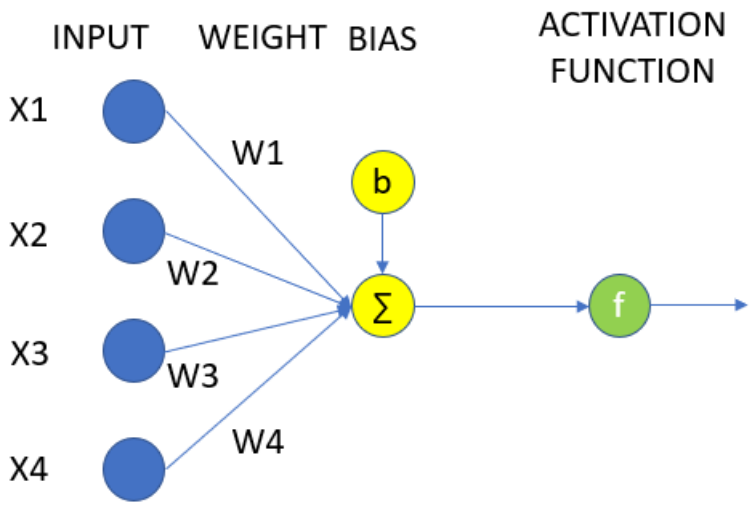


Figure 2.4: Illustration of activation function on each node in artificial neural network with 4 inputs and 1 hidden layer

such as sigmoid function or hyperbolic tan function, and also can be a linear function where f(x) = x. This calculation process illustrated in Fig. 2.4.

The terms Deep Learning, which consider as novel methodology to solve problems [12], correspond to Deep Neural Network. Where Deep Neural Network means where we used a model which consist of more than one hidden layer. And the activation function of each hidden layer can be different [8]. Deep Learning also describes as a family of learning algorithms rather than a single method that can be used to learn complex prediction using multi-layer and many hidden units[13]. To conclude, the only difference between machine learning and deep machine learning is the number of layers employed in the model. And most of the other things, such as type of learning, are the same. Since in this thesis project we used more than one hidden layer in our network model, thus, here we employ deep learning technique.

Primarily, there are three different type of deep learning [10][14], 1) Supervised Learning, 2) Unsupervised Learning, 3) Reinforcement Learning. Each type used to solve different kind of problem. Nowadays, we use supervised learning in most cases, such as image classification and natural language processing problem, including this thesis project is also one example of problem that can be solved using supervised learning. Then, we only focus on supervised learning.

2.4.2 Supervised Learning

Supervised learning is a machine learning technique where we train the network using training data set consists of pairs of input and desired output [15]. The goal here is, as mentioned before, we want to minimize the differences between our network output and desired output. The simplest example is image classification problem, where we have sufficiently large data to train the network, and we also already labeled each data to its classification (e.g. cats or dogs). Then we feed the input to our neural network model and then adjust the weight and nodes value using back-propagation technique based on our desired value. With this technique, the network will recognize the pattern which differentiate between dog and cat and create decision based on learned pattern.

There are so many artificial neural network architecture to be used in supervised learning. We choose a specific architecture depends on the nature of the problem that we are trying to solve. For example, if we want to solve an object detection problem, then we can use convolutional neural network (CNN) as our base architecture[16].

Another example is predicting people's home country based on their name. This kind of problem classified as sequence problem, where basically we can guess someone's origin based on the order of alphabets or character used in their name. Sequence problem require a network which able to keep information from past input when it learns the new input. For this kind of problem, we can use Recurrent Neural Network[17], because theoretically Recurrent Neural Network able to memorize the pattern from past input.

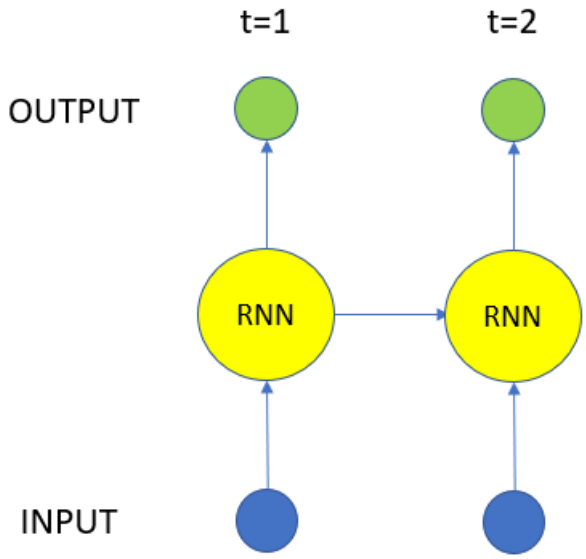


Figure 2.5: Simple RNN process, one input size and 2 different input

The input for our neural network in this thesis project will be the trajectory of trapped particles, which we want the network to recognize any pattern for different stiffness value and different delay from it. Thus, the problem of this thesis project can be classified as sequence problem. If in predicting home country based on people's name we want the network to recognize the pattern in the order of alphabet letters used in the input, here our initial guess is we want the network to recognize and memorize the pattern between each data points such as, the distance or the displacement of the particle. In order to do that, we can use Recurrent Neural Network (RNN).

2.4.3 Recurrent Neural Network (RNN)

Recurrent Neural Network (RNN) is a type of neural network which uses sequential data as an input. RNN commonly used for language translation, speech recognition or anything which use sequence data as input. The key feature of RNN is the "memory". RNN able to memorize previous input to analyze current input and influence current output. Thus, the output of RNN also depends on previous input. To be able to memorize past input, RNN has a feature called hidden state.

The process in simple RNN can be seen in Fig. 2.5. At time $t = 1$, we can see the first input fed to the RNN and then it gives an output. At time $t = 1$, RNN also update it's hidden state according to:

$$h_t = f_w(h_{t-1}, x_t) \tag{2.9}$$

Then, at time t = 2, we can see that the RNN receive the second input and the hidden state from previous time, represent by the arrow between the RNN, to calculate the output at time t = 2. This process continue for all input, thus, the hidden state will be updated for each time t. This is the mechanism that make RNN able to memorize previous input.

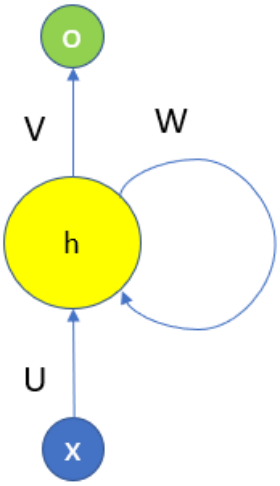


Figure 2.6: RNN Activation Function

in more details, the simplified mechanism inside RNN can be seen in Fig. 2.6. Where U is the weight connection between input nodes and hidden layer nodes, W is the weight connection between hidden layer and it's hidden state, V is the weight connection between hidden layer and output layer. The forward propagation in RNN governed by following equations:

$$a(t) = b + Wh(t-1) + Ux(t) \tag{2.10}$$

$$h(t) = f_w(a(t)) \tag{2.11}$$

$$o(t) = c + Vh(t) \tag{2.12}$$

$$y(t) = f_w(o(t)) \tag{2.13}$$

Where a is hidden layer value, b is bias of hidden layer, $h(t)$ is hidden state at time t, o is output layer value, c is output layer bias, f_w is activation function and y is the output.

Even though hidden state in RNN is able to memorize previous input, but if we take a look at equation (2.11) we can see that the hidden state is updated in each iteration, which cause another problem, that is short-term memory problem. If RNN trained with larger and larger data set, RNN will face vanishing gradient problem. To overcome this problem, there are two versions of RNN created; 1) Gated Recurrent Unit (GRU)[18] and 2) Long Short Term Memory (LSTM)[19].

LSTM has more features than GRU. If GRU has only two gates called Reset Gate and Update Gate, LSTM also has two more gates called Forget Gate and Output Gate. Gate is an additional artificial neural network which also has trainable weights and biases. The Update gate will decide whether the output value should be updated or not. The Reset gate will decide whether the previous output value is important or not. Forget gate will decide how much information from previous state should be kept. The Output gate will decide which value will be an output to hidden state, and will determine the new value of hidden state.

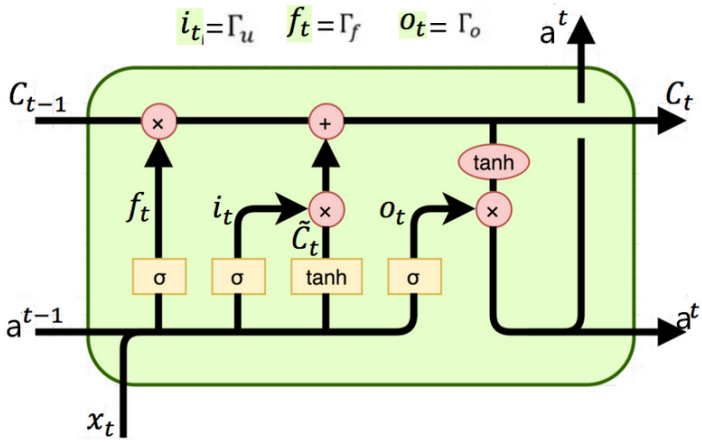


Figure 2.7: LSTM Basic Architecture, figure reproduced from Ref. [20]. In the source article they use hyperbolic tan (tanh) and sigmoid (σ) as activation function

The forward propagation in LSTM governed by following equations[20]:

$$\tilde{C}_t = f_w(W_c[a(t-1) + x(t)] + b_c) \tag{2.14}$$

$$\Gamma_u = f_w(W_u[a(t-1) + x(t)] + b_u) \tag{2.15}$$

$$\Gamma_f = f_w(W_f[a(t-1) + x(t)] + b_f) \tag{2.16}$$

$$\Gamma_o = f_w(W_o[a(t-1) + x(t)] + b_o) \tag{2.17}$$

$$c(t) = \Gamma_u \tilde{C}_t + \Gamma_f c(t-1) \tag{2.18}$$

$$a(t) = \Gamma_o f_w(c(t)) \tag{2.19}$$

Where f_w is activation function, $\tilde{C}_t$ is hidden state as in RNN, Γ_u is the update gate, Γ_f is the forget gate, Γ_o is the output gate, and $a(t)$ is the output.

Due to the architecture of GRU has less feature than LSTM, in terms of performance speed, GRU is 29.29% faster than LSTM. GRU performance surpass LSTM in the scenario of long text and small dataset, but inferior to LSTM in other scenarios[21]. Since our problem will use long trajectory and sufficiently large dataset, we employ LSTM network as base of our model.

2.4.4 Hyperparameters Optimization

Despite many advantages of machine learning application, the designing and training neural network is still challenging and unpredictable procedures, one of the reason is tuning the hyperparameters part. Hyperparameters refer to artificial neural network parameters that cannot be updated during the training of machine learning[22]. The importance of hyperparameters listed as follows[23]:

- Learning rates
- Momentum for RMSprop

- Mini-batch size
- Number of hidden layers
- Learning rates decay
- Regularization, which is used to reduce variance and avoid overfitting
- Activation functions
- Optimizer (e.g. Adam, etc.)

Common and easiest method but take longer time to optimize these hyperparameters is to tune it manually. So if we do not satisfy with our network performance, we change the hyperparameters and then re-train the network to see whether its performance improve or not. But, we can still reduce needed time by analyzing which hyperparameters really important for our problem.

3

Methods

In general, we want to calibrate the stiffness of the trap and to measure the delay using the trapped particle's trajectory. For harmonic trap with delayed feedback, we use two methods. First, we use approximation method based. Second method, we implement deep learning to calibrate the stiffness and predict the delay. After getting good result in harmonic trap, we then implement deep learning method to calibrate stiffness and predict the delayed feedback in rotational force fields.

3.1 Experiment Data

From experiment team, we receive four different files, each contain data points with different delayed feedback; 0, 10, 30 and 40 ms. We use the data with 0 ms delayed feedback to determine stiffness of the trap and friction coefficient used during the experiment. This information is really important because we will train the network using parameters that cover experiment parameters to ensure the network will be able to predict stiffness and delay from experiment data.

The experiment records Brownian motion of an optically trapped polystyrene sphere with diameter 200 nm, trap size 10µm, adjusted trapping power 10mW, and the temperature during experiment is 300 K. Experiment data with 0ms delay contain 10^5 measurement points, which shown in Fig. 3.1. To estimate the trap stiffness, we can use variance method on experiment data without delayed feedback [24]:

$$k = \frac{k_B T}{< x^2 >} \tag{3.1}$$

with k_B is Boltzmann constant with value 1.38064852 × 10-23 m^2kg/s^2K. Before we can use this value to estimate friction coefficient, we need to calculate the auto-correlation function of the trajectory.

The auto-correlation method to estimate stiffness described by the following equation[1]:

$$< x(t + \Delta t)x(t) >= \frac{k_B T}{k} e^{-\Delta t/\tau} \tag{3.2}$$

where τ is the timescale on which the trap and optical forces drive the particle

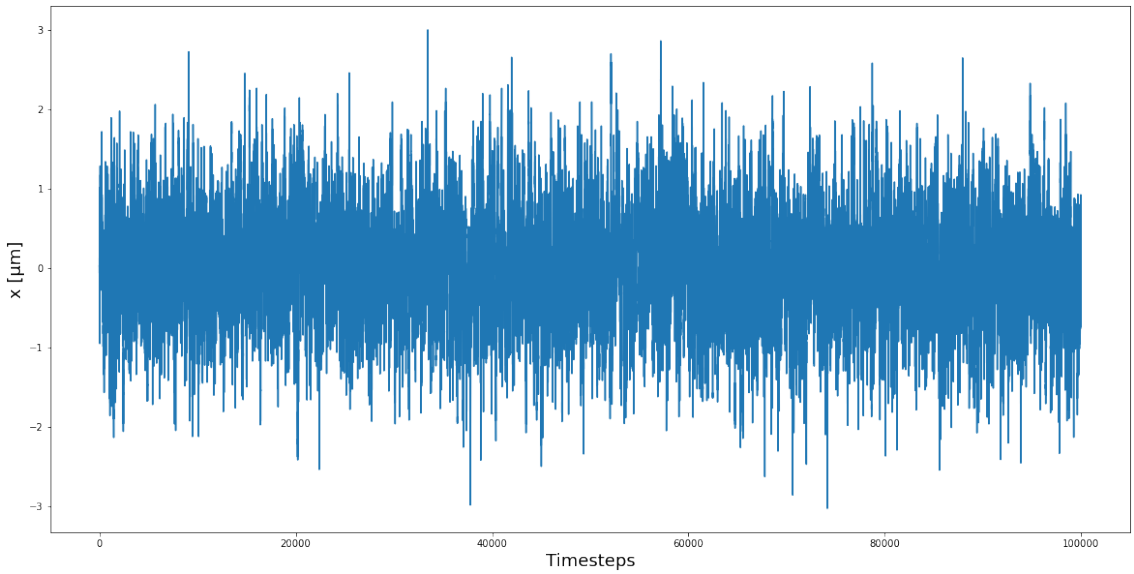


Figure 3.1: Experiment trajectory with delay: 0 ms. It contains 10^5 data points. Time between two measurement points is 10ms

towards the potential energy minimum[25].

$$\tau = \frac{\gamma}{k} \tag{3.3}$$

Thus, by calculating auto-correlation then create the best exponential function fit, one can estimate the value of τ and γ. To calculate auto-correlation of the trajectory, we use the following equation[26]:

$$ACF = \frac{< x(t)x(t+\Delta t) > - < x >^2}{< x^2 > - < x >^2} \tag{3.4}$$

3.2 Delay Prediction Using Approximation Method

For Harmonic trap with delayed feedback, we try to predict the delay using approximation method. In fact, there is no straight-forward formula to calculate delay from Brownian trajectory. But, we will estimate the delay with some manipulation steps to the Langevin equation. Suppose we have the Langevin equation of trapped particle in over damped Harmonic trap:

$$\frac{dx}{dt} = -\frac{k}{\gamma}x(t-\delta) + \sqrt{\frac{2k_BT}{\gamma}}\xi(t) \tag{3.5}$$

First, we multiply both sides with $x(t-T)$

$$\frac{dx}{dt}x(t-T) = -\frac{k}{\gamma}x(t-\delta)x(t-T) + \sqrt{\frac{2k_BT}{\gamma}}\xi(t)x(t-T) \tag{3.6}$$

To get rid of the random noise, so we can get a direct formula to estimate the delay, we take the average of both sides so the second term of right-hand side in equation (3.6) vanish, because $<\xi> = 0$

$$< \frac{dx}{dt}x(t-T) >= -\frac{k}{\gamma} < x(t-\delta)x(t-T) > +\sqrt{\frac{2k_BT}{\gamma}} < \xi(t)x(t-T) > \quad (3.7)$$

$$< \frac{dx}{dt}x(t-T) >= -\frac{k}{\gamma} < x(t-\delta)x(t-T) > \quad (3.8)$$

Delayed feedback in optical tweezers mean, the laser beam comes from the position of particle at $t - \delta$. We assume that the displacement due to this beam is the highest displacement because the incoming laser beam is the only thrust that applied to the particle in optical tweezers. Thus, if we vary T and numerically calculate equation (3.8), we can approximate that $\delta = T$ when $< \frac{dx}{dt}x(t-\tau) >$ is maximum.

3.3 Optical Tweezers Calibration and Delay Prediction Using Deep Learning

This second method is the main work that we want to implement to solve this problem, which is deep learning technique by using LSTM network as base of our model. Training an LSTM architecture is basically following the same procedure as training for other architecture. We start with processing the data, build the network and choose all parameters needed, train the network then evaluate the network performance. Here we employ DeepCalib software package[29].

3.3.1 Data Processing

For all deep learning or machine learning techniques, processing the data is one of the fundamental things. Not only one need sufficiently large dataset, but also need to ensure that the data has high quality so the network able to recognize and learn the pattern that we want the model to learn. Here comes the very first challenge of this project: data set, since we do not have that many trajectories of trapped Brownian particle.

In order to overcome this problem, we build a trajectory generator function in Python where we implement the delayed feedback and yield one dimension trajectory. We generate a simulated trajectory using parameters which also cover parameters that used by the experiment team, so we do not need to re-train the network. We generate simulated trajectories for training, and for performance test we generate new trajectories in order to avoid using same trajectory twice; during training and during performance test.

We used different ways to generate stiffness and delay. We generate stiffness in range value from 0.1 fN/µm to 350 fN/µm uniformly distributed in logarithmic

scale, while we generate delay uniformly distributed in linear scale in range 0 to 100 ms. We generate friction coefficient uniformly distributed in logarithmic scale with reference value : $6\pi\eta r$[1], where r is the particle's radius and η is the viscosity of the medium. Other Physical parameters are fix value, temperature T is 300 K and time step is 10 ms.

Another challenge for generating the trajectory is, there is possibility that the trajectory does not converge. To make sure the trajectory do not become too large to handle by our programming language, we set maximum and minimum position using trap size as reference: 10 µm. So if the particle escapes as the feedback shots, in trajectory generator function we set rules, that is, it will stay at maximum or minimum position.

The summary on how trajectory generator works gives in Algorithm (1). This function take number of trajectory that we want to create in one call, and will gives array of δ, k and position of each particle. User can change how this function calculate new position depends on the trap's type being used. Some example of trajectory generated by this function shown in Fig. 3.2.

Algorithm 1 Simulate Trajectory Function

```
Initialize physical parameters                          ▷ k, γ, δ
Initialize delay array
initialize position array
for i in range equilibration steps do                   ▷ each iteration represent 1 ms
    calculate new particle's position
    if particle's position > max position then
        new particle's position = max position
    else if particle's position < min position then
        new particle's position = min position
    end if
    update delay array
end for
for j in range data points×oversampling do              ▷ each iteration represent 1 ms
    calculate new particle's position
    if particle's position > max position then
        new particle's position = max position
    else if particle's position < min position then
        new particle's position = min position
    end if
    update delay array
    if j % oversampling = 0  then
        update position array
    end if
end for
Return δ, k and position array
```

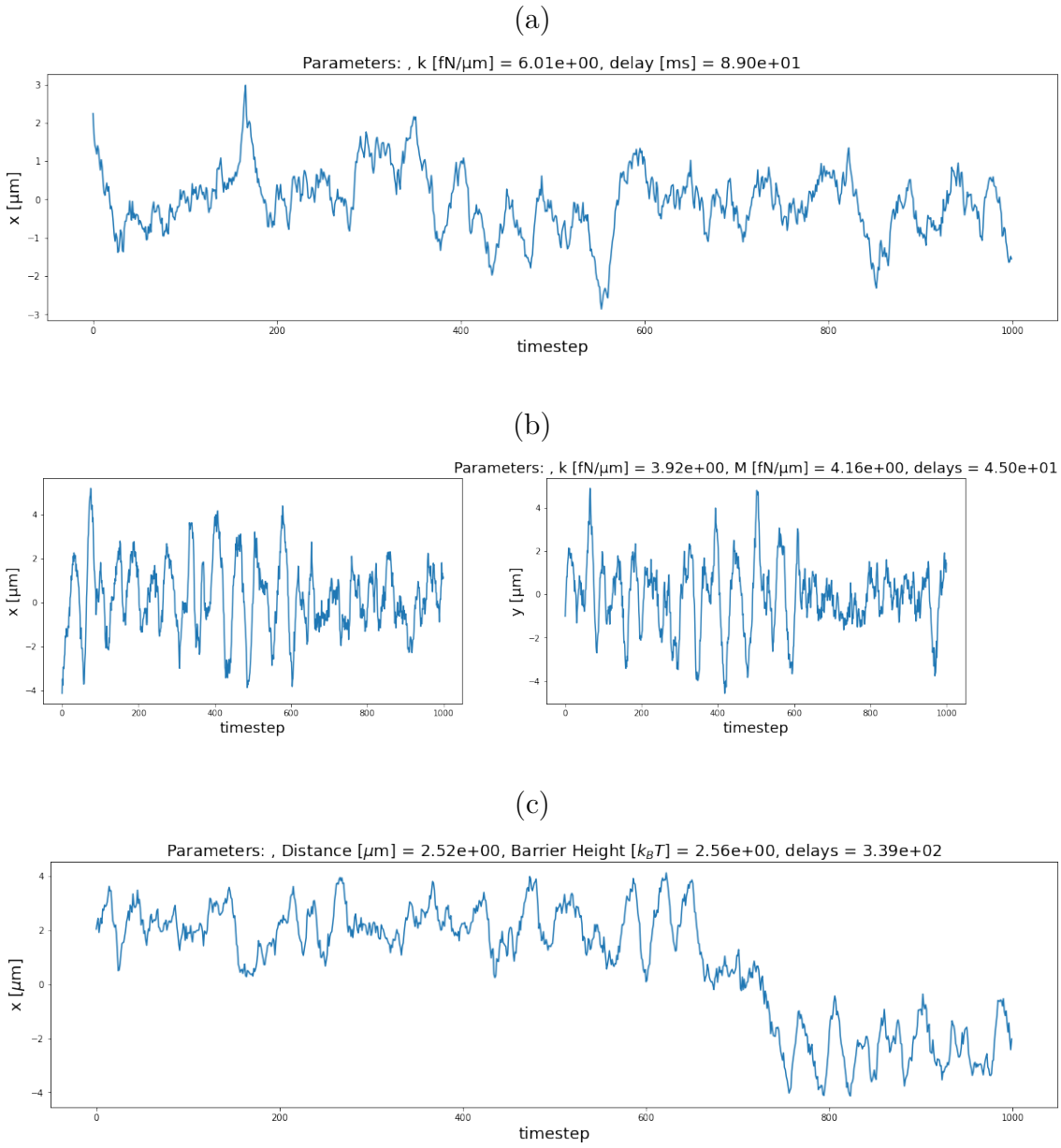


Figure 3.2: Generated trajectory using Simulate Trajectory function, with different force fields (a) Harmonic trap, (b) Rotational force fields, (c) Double well potential

3.3.2 Build the Network

To build the network, we employ DeepCalib software package and use default setting from it, especially `create_network.py`. Initially we used 3 LSTM layer as hidden layers, each consist of 1000, 250 and 50 neurons respectively. In output layer, Dense layer employed with number of neurons depends on type of trap currently analyzed. But, at least number of neurons in output layer is 2, because we want to predict stiffness and delay. As for the input, we use one node for each trajectory position. In our default setting, we use 1000 trajectory points.

The nature of our problem is we have so many data variance that we want to predict. one stiffness value have 100 different type of trajectory since we set our delay in range [0, 100] ms. Using the method to generate stiffness and delay as discussed in Section 3.2.1, the output will be in different range of number. One neuron which will predict stiffness value will be in range [-3,3] while the other which will predict the delay will be in range [0,1].

From the nature of the problems, we then can determine some hyperparameters. First, learning rate. In fact, there is no reference on how we pick the best learning rate for our model. The basic information is smaller learning rate will be slowing the network's learning process to reach saturation. Since the trajectory have so many variances, we use 1e-3 as our learning rate. This is because 1e-3 is not too small to ensure our training process do not take too long time to reach saturation, yet not too large learning rate to ensure that our model able to recognize enough details. Second, activation function at output layer. Seeing we will use different range of value in the output neurons, the best fit for this problem is linear function, where:

$$f_w(x) = x \tag{3.9}$$

We will then tune other hyperparameters after we get initial results to optimize our model performance if necessary.

3.3.3 Train The Network

After we build the network, now we set how are we going to train the network. Some suggest to use Stochastic Gradient Descent where we decrease learning rate during training process[27], but increasing batch size with constant learning rate during the training run actually gives better performance[28]. Thus, we use constant learning rate and will increase the batch size. This strategy will start with small batch size, then it will be increased after 1000 iteration for each batch size. So the network will learn that there is a wide variation in the data set first, then it will learn the details within increasing batch size. `simulate_trajectory` used to generate training data set for each training iteration. We set the default for batch size in this order: 32, 128, 512, 1024. with at least 1000 iteration for each batch size. If the training performance from this default is not saturated, we can continue the training process with adding either more iteration for latest batch size, or larger batch size.

Ideally, we want the prediction of our network is the same as our target, as illustrated in Fig. 3.3. Seeing the nature of how we want the prediction goes, it

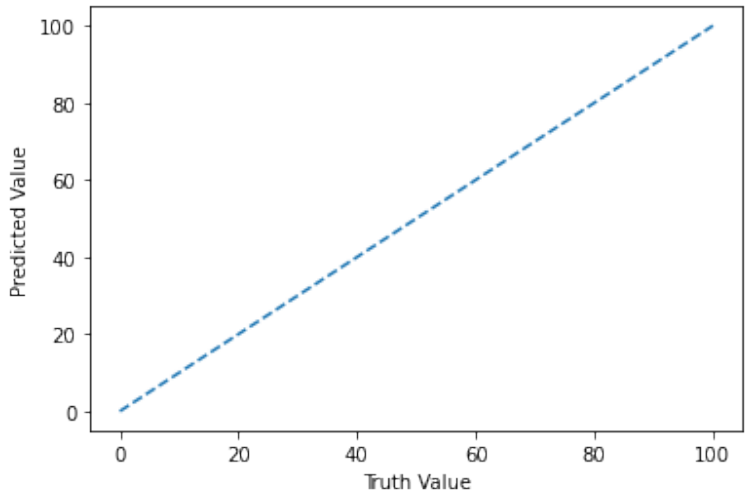


Figure 3.3: We want the network prediction fit to a linear line

is actually similar with linear regression problem. Thus, the best loss function for the model is Mean Squared Error (MSE), where we try to fit our prediction to the linear line as illustrated in Fig. 3.3. with MSE formula

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(Y_i - \hat{Y}_i)^2 \tag{3.10}$$

Where Y_i is the target value, $\hat{Y}_i$ is the network's prediction and n is number of data.

To do the training, we employ DeepCalib especially `train_network.py` function which accept network model, batch sizes in array form, number of iteration in array form and `simulate_trajectory` function as input. Batch size will determine how many trajectories generated in one training iteration. All training function made using Keras library. The summary on how the training will work is shown in Algorithm 2.

Algorithm 2 Training Function

```
Initialize batch size                          ▷ default:(32, 128, 512, 1024, 2048)
Initialize number of iteration for each batch size   ▷ default:1000 for each iteration
for i in number of batch size do
    for j in number of iteration for current batch size do
        generate trajectories as many as current batch size
        feed trajectories into the network
        calculate MSE and adjust trainable parameters
        if j % number of iteration×verbose = 0 then
            print MSE
        end if
    end for
end for
```

3.3.4 Performance Test

After training, it is time to test the network's performance using new generated trajectories. The algorithm in this part is pretty much the same with training part, but here the network do not adjust the trainable parameters. In order to do performance test, we also employ DeepCalib especially `plot_test_performance.py`. This function plot the prediction results directly after feed the newly generated trajectories.

Algorithm 3 Performance Test

1: Initialize number of trajectories
2: generate trajectories using `simulate_trajectory` function
3: initialize empty array to save prediction result
4: **for** i in number of trajectories **do**
5: feed i-th trajectory to the network
6: save the prediction result
7: **end for**
8: plot prediction result

3.3.5 Optimization

Performance Optimization will be performed if the results are not good enough. The optimization mostly will relate to hyperparameters. In this project, some hyperparameters considered: Activation Function, Regularization, Architecture such as number of neurons in each layer and number of hidden layers. Comparison with theory also need to be done to ensure we do not perform any unnecessary optimization.

4

Results

The results of all methods discussed in Section 3 present here. We start with the results from analyzing experiment data. Then, the results separated based on trap type. As mentioned before, there are two type of optical trap analyze in this thesis project; Harmonic and Rotational force fields.

4.1 Experiment Parameters

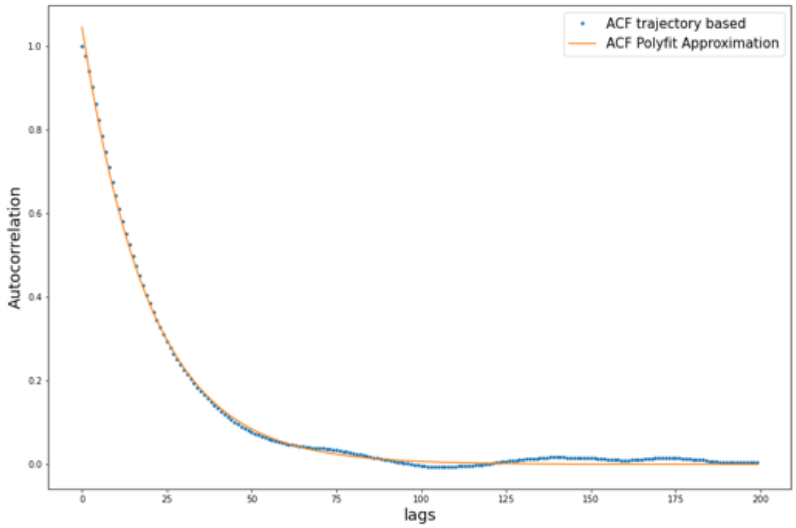


Figure 4.1: Calculated ACF and the best fit of exponential function

By calculating equation (3.1), the trap stiffness obtained is 8.49 N/m. Using this result we then can continue to calculate auto-correlation function using equation (3.4), the result of this calculation can be seen in Fig. 4.1. The exponential function that's the best fit for the calculated ACF is

$$y = 1.0441e^{-0.0503066} \tag{4.1}$$

Thus, we can estimate γ, where

$$0.0503066 = \frac{\Delta t}{\tau} \tag{4.2}$$

With Δt = 10 ms and k = 8.49 N/μm then

$$\gamma = (8.49)\frac{0.01}{0.0503066} \tag{4.3}$$

And we obtain that γ= 1.698 $\times$ 10^{-9} kg/s. As for others data from experiment, we assume that experiment team used exactly the same parameters, the only difference is the delayed feedback. Now we have to ensure that our trajectory generator do cover these parameters.

4.2 Harmonic Trap with Delayed Feedback

For harmonic trap, we use both methods; approximation method and deep learning technique.

4.2.1 Approximation Method Results

Approximation method as described in Section 3.2 gives result as shown in Fig. 4.2. Some predictions are on correct linear line, but some others are predicted wrong.

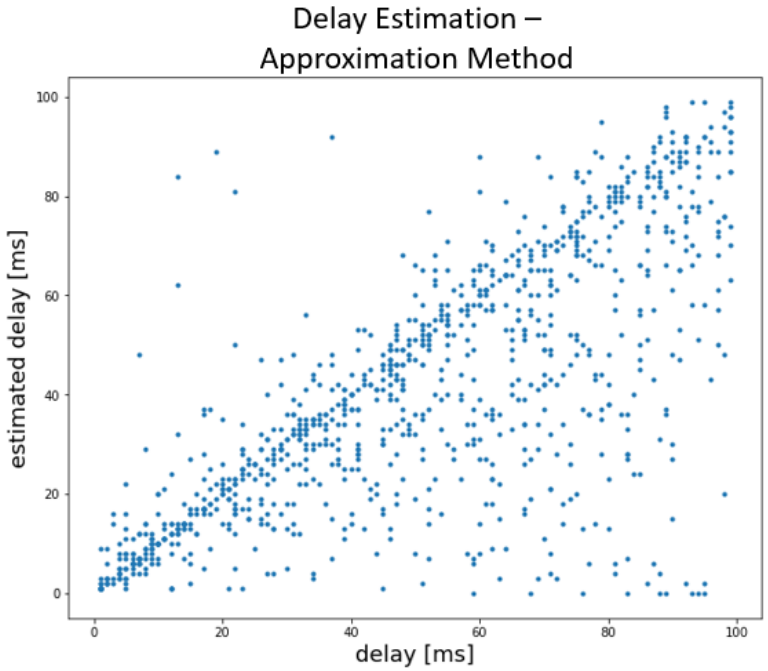


Figure 4.2: Approximation Method estimating delay result

After we got estimated delay using approximation method, we can estimate stiffness using variance method described by equation (3.1). This method also gives good for some predictions, but gives not good results for some others. The result of this stiffness prediction can be seen in Fig. 4.3.

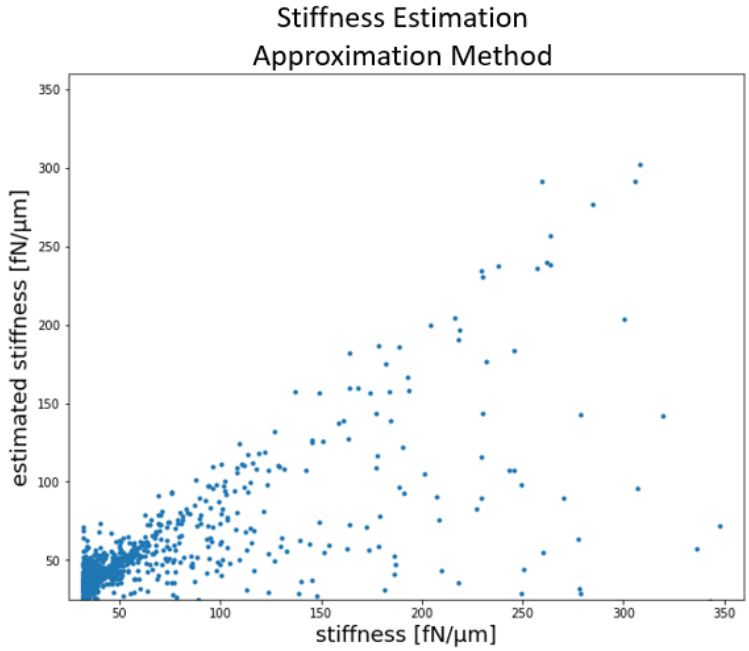


Figure 4.3: estimating stiffness result using variance method

4.2.2 Deep Learning Results

Here we used `simulate_trajectory` where we generate stiffness uniformly distributed in logarithmic scale, in range [0., 350] fN/μm for each iteration. As for delay time, we generate uniformly distributed in linear scale in range [0, 100]. The training end with final MSE 0.2784.

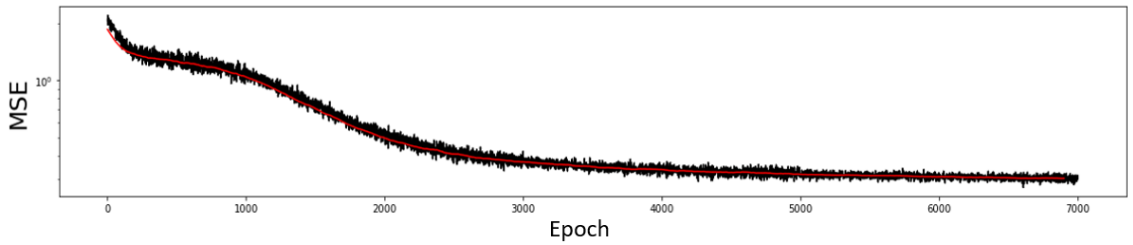


Figure 4.4: Training performance

The network's performance test gives great result for predicting stiffness. But, for large stiffness the prediction has larger deviation with the truth value. As for delay prediction, there are certain range that the network can not predict properly and it just predicts to half of the maximum delay. These results can be seen in Fig. 4.5.

This performance still can be optimized by setting some parameters and change on prediction value, so the network can learn easier. But, comparing these results with Approximation method, is a promising proof that Deep Learning can be used to predict the delay and calibrate the stiffness simultaneously.

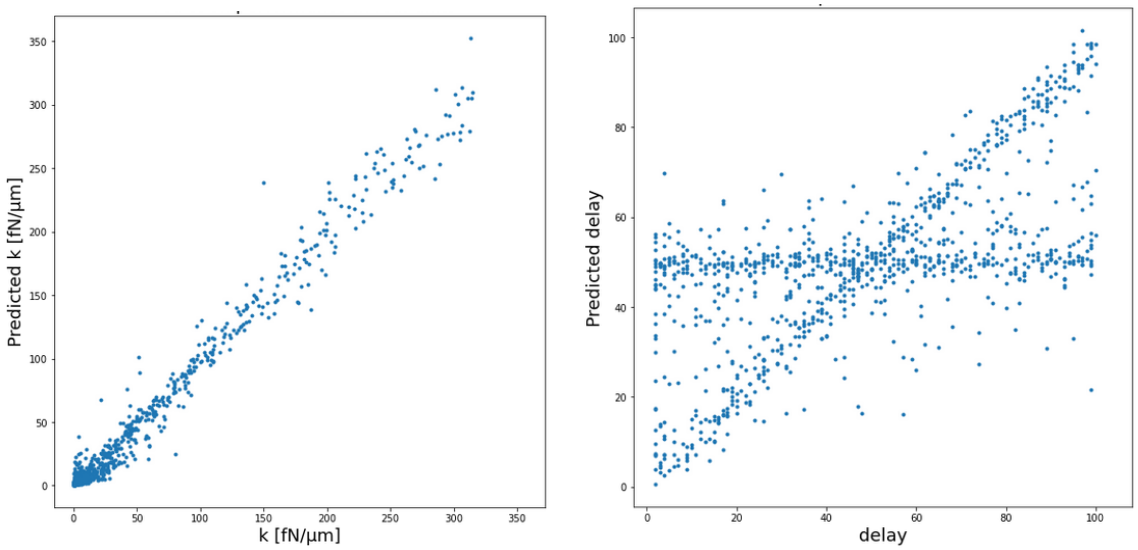


Figure 4.5: Prediction results using Deep Learning. Left-panel: Stiffness prediction, Right-panel: Delay prediction

4.2.3 Analyze the Deep Learning Result

As shown in the right panel of Fig. 4.5, the delay prediction result is still not good. It is like there is a region where the network can not predict the delay properly. Thus, to investigate this region, using the same trained network, we feed the network with trajectories with certain range of stiffness. First, we generate trajectories with stiffness range [0,31] fN/µm and then feed generated trajectories to the trained network to see how well the prediction is. Then, we repeat the same step but different stiffness range. Here we investigate two other range of stiffness, which is [32, 350] fN/µm and k larger than 50 fN/µm. Lastly, we compare the prediction of trained network in these three different range of stiffness by making a plot of each predicted stiffness and predicted delay to see how our network predictions relate to different range of stiffness.

Apparently, the network cannot predict both stiffness and delay properly when the stiffness value is below 31 fN/μm. The prediction performance becomes better and better with larger stiffness value. The comparison can be seen in Fig. 4.6. From the same figure, we also can see that the trained network still able to predict the stiffness properly even though we only train the network with stiffness range [0, 350] fN/µm, this unique performance can be seen in the top row right-column panel. Surprisingly, the delay prediction within this range also showing great results. Unlike the previous stiffness range [31, 350] where the delay predictions gives wider deviation from the truth value.

Now this become a problem since from Section 4.1 we know that the trap stiffness used during experiment is 8.5 fN/µm, which unfortunately is in range where the trained network cannot predict both stiffness and delay properly as shown in figure 4.6 left-column panel. Indeed, the network needs to be optimized to increase its

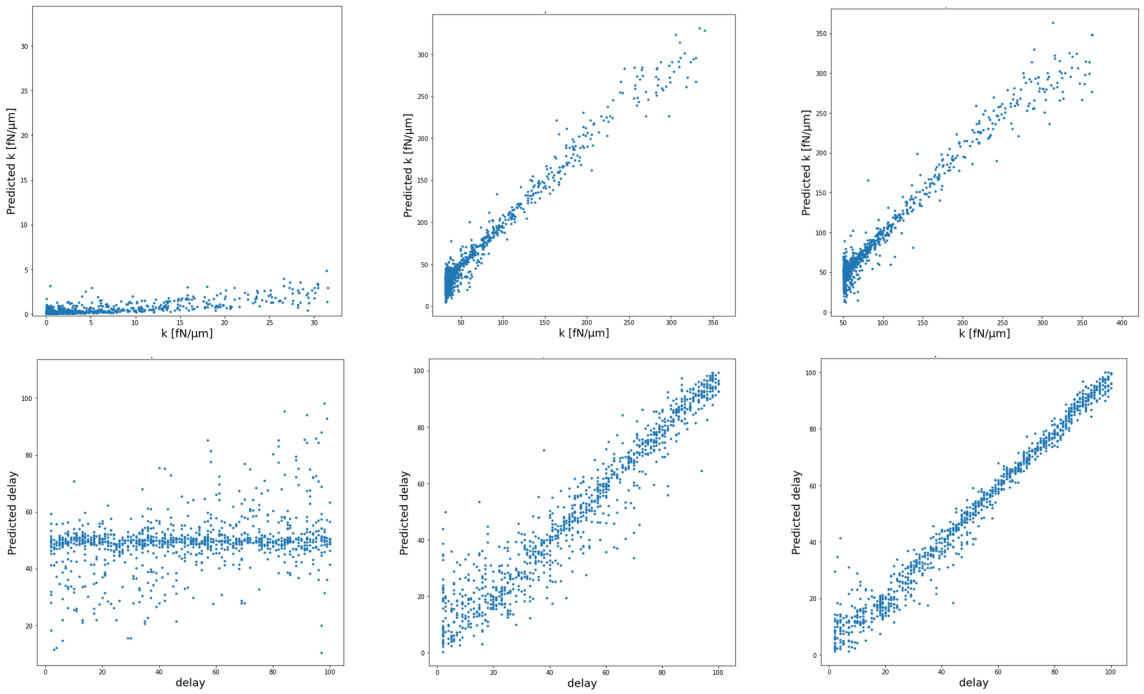


Figure 4.6: Predictions using trained network with different range of stiffness. left-column: k [0, 31] fN/µm, middle-column: k [31, 350] fN/µm, right-column: k [50, 350] fN/µm. top row is stiffness prediction, bottom row is delay prediction

performance.

Even though the trained network performance still needs to be optimized, this performance already slightly outperform Approximation method and Variance method in order to predict delay and trap stiffness respectively within certain range of stiffness. One can compare Fig. 4.2 and Fig. 4.3 with Fig. 4.6.

4.3 Optimized Performance

Since the preliminary results are still not good enough, which is normal in machine learning technique, now we go through optimization in order to increase the network performance. Here the optimization classified into two categories. First, optimization on trajectory generator. Second, optimization on network's side.

On trajectory generator, the change is on how the stiffness generated. Previously, it is generated uniformly distributed in logarithmic scale. We can see how the generated value using this generator distributed in Fig. 4.5 left panel, where small stiffness value are the most populated compare to higher stiffness value. Now stiffness value generated uniformly distributed in linear scale in range[0,100] fN/µm. Smaller range of stiffness used because this project does not really need stiffness value in order of magnitude of 2, this change still cover parameter used in experiment. This optimization will also reduce data variance.

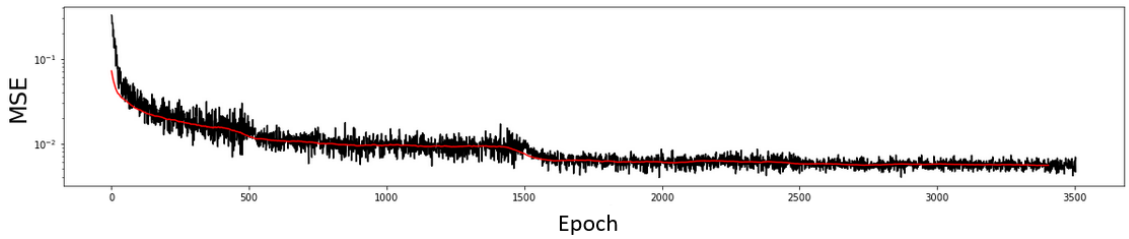


Figure 4.7: Training MSE performance. MSE saturated at 0.05. Black line is MSE of each epoch, red line is MSE average measured with 100 epoch interval

Second optimization is on network's side. Previously, neurons in the output layer has different range of value. Neuron for predicting stiffness have value in range [-3, 3], and neuron for predicting delay have value in range [0, 1]. These two different neurons will affect the state of previous layers trough back propagation process during training. Here, we want to try using same range for output layer, which [0, 1].

After modifying codes in accordance with above optimizations, we build a new network with exactly the same architecture and train this new network using exactly the same batch size and number of iteration as before, then test its performance. The training progress shown promising performance as we can see in Fig. 4.7 the MSE of optimized network saturated at low value compare with previous training performance.

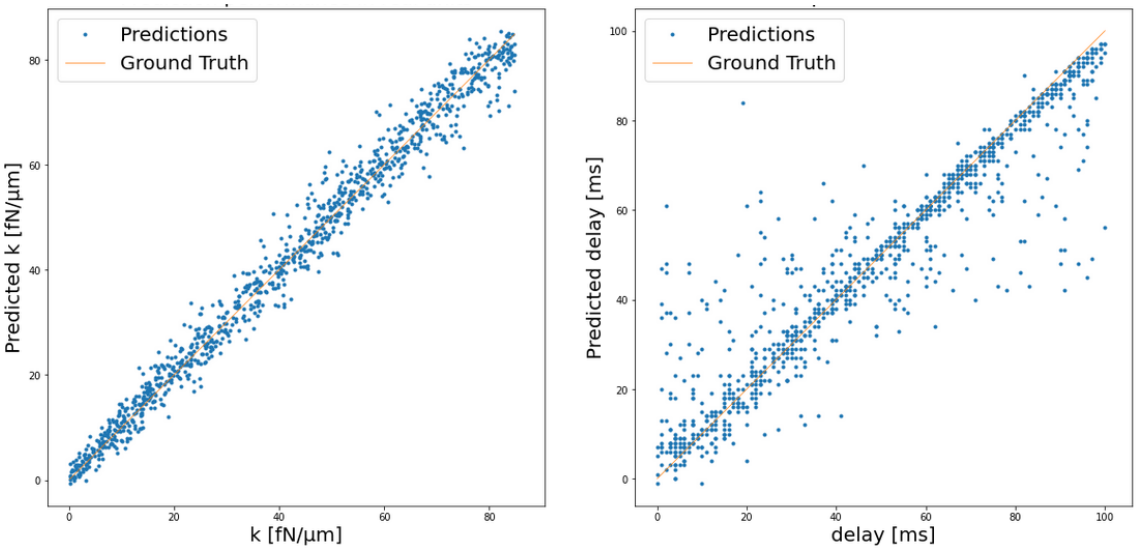


Figure 4.8: Prediction results using Deep Learning after simple optimization. Left-panel: Stiffness prediction, Right-panel: Delay prediction

The performance test results from these simple optimizations surprisingly greater than previous results. as shown in figure 4.8, trained network able to predict with smaller deviation from the truth targets value compare to previous results. This optimization already fix one issue: predicting stiffness value in range [0, 31]. Compare

with figure 4.5 top-left panel, the optimized trained network already outperform previous result and variance method in predicting stiffness value. Also of course outperform Approximation methods in predicting the delay.

4.3.1 Experiment Data as Test Data Set

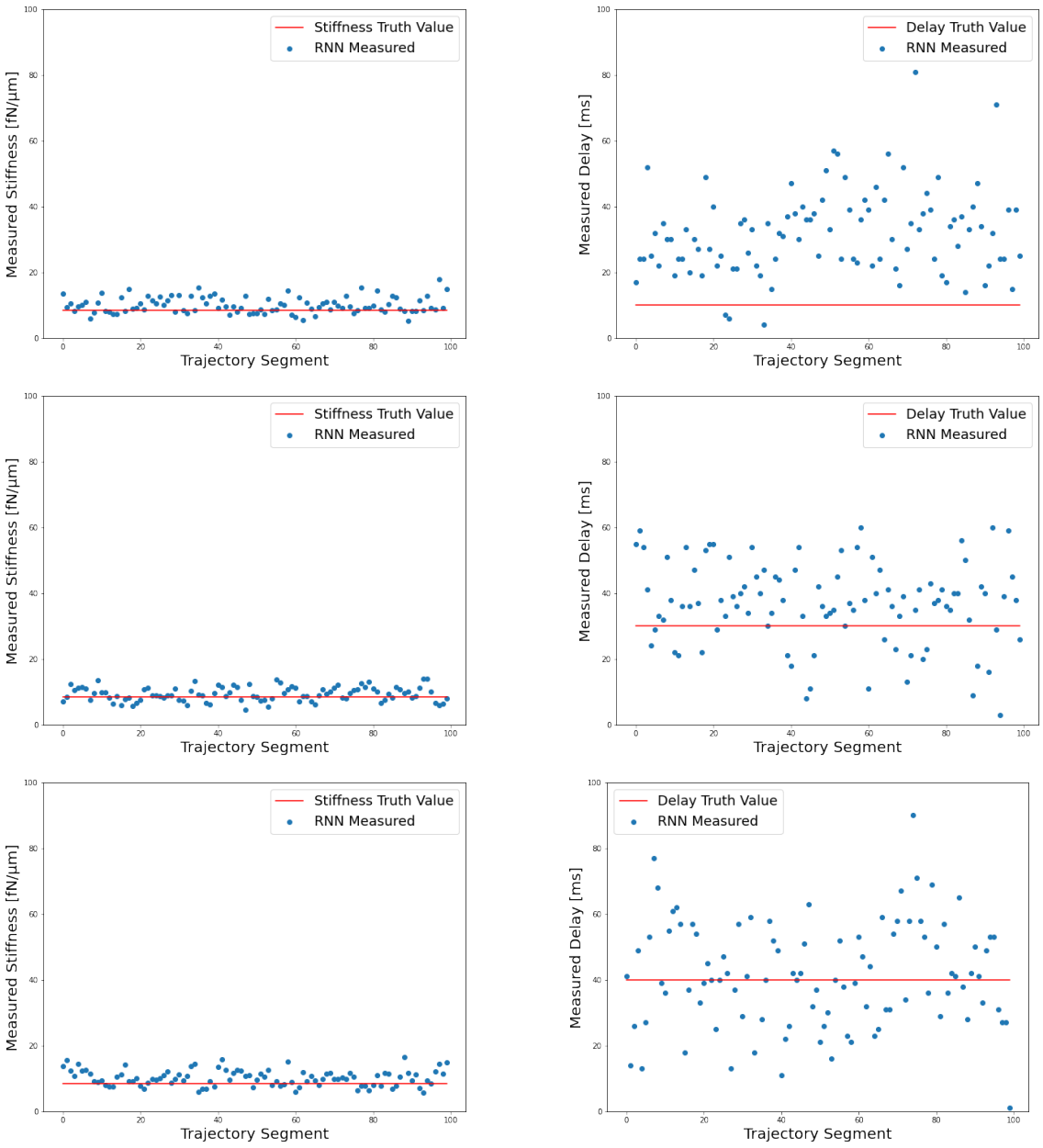


Figure 4.9: Predicting stiffness and delay from experiment data. The first row is predictions for data with 10 ms delay. The second row is predictions for data with 30 ms delay. The third row is predictions for data with 40 ms delay

With great result from optimized trained network performance test, now we can try to predict stiffness value and implemented delayed feedback from experiment data. To remind about received experiment data, we have three different trajectories;

trajectory with 10, 30 and 40 ms delay. With 10^5 measurement points for trajectory with 10 ms delay, 5×10^4 measurement points for trajectory with 30 and 40 ms delay.

In order to predict stiffness value and delay from experiment data, we process the data, so we have 1000 measurement points as single trajectory, in order to make it receivable by the optimized trained network which receives an input with 1000 data points. Then, we just feed the data directly into the network, re-scale the predictions into physical value, then create the plot to see the prediction results.

In general, optimized trained network able to predict the stiffness properly as shown in Fig. 4.9. The optimized trained network able to predict stiffness properly in all delayed feedback value. But unfortunately, the optimized trained network can not predict the delay properly. But, even though the delay predictions look like random, the mean value of these predictions is actually close to the truth value for experiment trajectory with 30 and 40 ms delay, which the estimation is 29.3 ms and 40.92 ms respectively.

4.4 Rotational Force Fields

Since in general we already have great result for harmonic trap, now we present the result of predicting stiffness and implemented delay in non-conservative force fields. If in harmonic trap, the force that constraining the particle is represented by k, in rotational force fields, the trapped particle is subject to restoring force which not only depends on stiffness k, but also torque ω, as described in Section 2.3.2. Thus, in this case, we build the exact same model, but with different number of neurons in the output layer, since now we have to measure three variables; stiffness k, torque ω, and delay δ.

Stiffness k, and ω is uniformly distributed in logarithmic scale with range [0,200] fN/μm and [-40, 40] fN/μm respectively. As for delay, we generate it uniformly distributed in linear scale in range [0, 100] ms. The final MSE of training process is 0.1480.

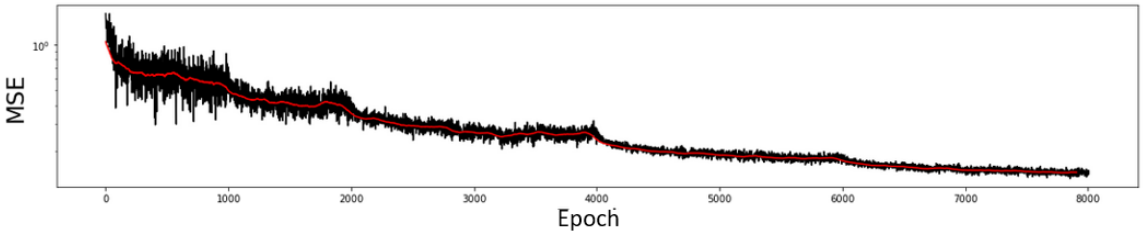


Figure 4.10: Training MSE performance for rotational force fields

The performance test's results are not bad compare to the Harmonic trap case. The trained network able to predict stiffness k, torque ω, and delay δ with a wide range of error, but the major trend direction are correct. The results shown in Fig. 4.11.

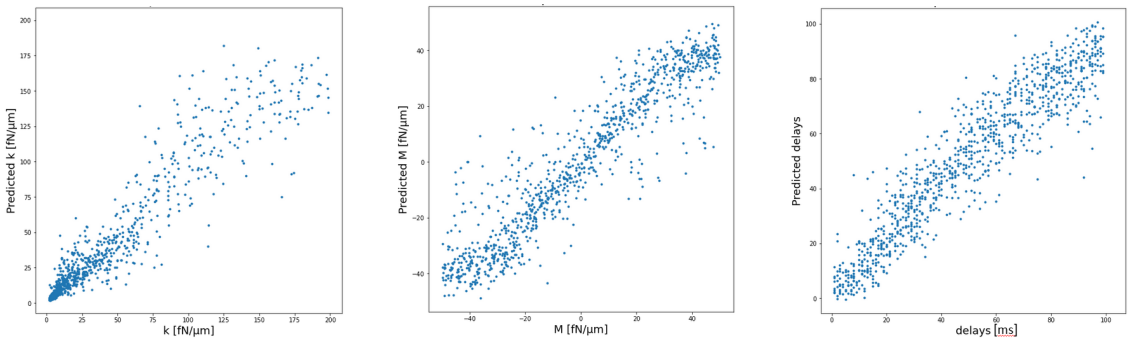


Figure 4.11: Prediction results for Rotational Force Fields. left-panel: stiffness k predictions. mid-panel: ωpredictions. right-panel: δpredictions. M in the middle-panel represent ω

4.4.1 Optimized Performance

From the left-panel of Fig. 4.11 we can see that the network predicts the stiffness k with wide range of predictions for large k. We also can see that small k generated more than large k. Here we try the same approach as on the harmonic trap case, we optimize the trajectory generator function, so it will generate the stiffness k and the torque ω uniformly distributed in linear scale. We also reduce the stiffness range to [0, 100] fN/μm, but still generate torque in range [-50, 50] fN/μ and delay in range [0,100] ms.

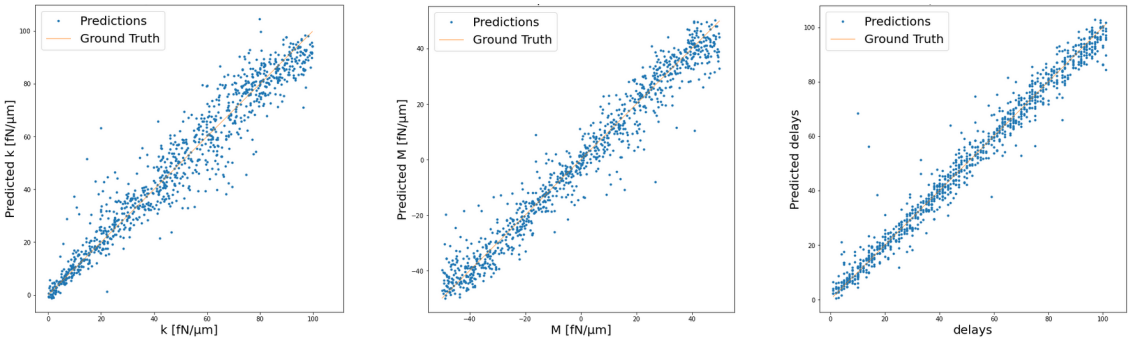


Figure 4.12: Predictions results for Rotational Force Fields with optimized trajectory generator. left-panel: stiffness k predictions. mid-panel: ωpredictions. right-panel: δpredictions. M in the middle-panel represent ω

The results of this optimization, even though the stiffness k predictions still have wide range of deviation for large k, but ω and delay predictions gives better result, as shown in Fig. 4.12.

5

Discussion

5.1 Why The Network Cannot Predict Delay Properly?

From Section 4.3.1, we know that even though the optimized trained network able to predict experiment trap stiffness from experiment trajectories, it still cannot predict delay properly. Optimization by tuning hyperparameters were also performed, but the optimized trained network still can not predict delay on experiment data properly. Hyperparameters that tuned are:

- Network Architecture. the latest architecture is using 3 LSTM layer as hidden layer, with number of neurons 4096, 1024, 256 respectively. final MSE still the same as previous architecture.
- Implement regularization terms. We try to implement regularization terms in order to make the weight values on each node small. But it gives not a better result.
- Change activation function on LSTM layer into ReLu. It also gives not a better result.

To investigate why the network cannot predict delay properly, we start with the basic of deep learning technique. As in Section 2.4, deep learning technique basically trying to recognize pattern from data and making decision based on that recognized pattern for another data set. Then, in a sense, deep learning technique will be hard to gives proper decision or prediction if the pattern is unrecognizable or if the data has similar pattern. The big question is, is our problem really has unrecognizable pattern? Then, we will continue with analyzing prediction results from Section 4, and we end with analyzing the similarities between trajectories.

5.2 Different Approach with Deep Learning

If previous approach we train a network which has two output; stiffness and delay, in this section we try different approach. Here we build new network, with the same architecture which gives the best result on previous approach, but now we only use

one neuron in output layer to predict only the delay.

We set the parameters as in the experiment, stiffness is 8.5 fN/μm and friction coefficient is 1.698 $\times$ 10^{-9} kg/s. And we generate delays uniformly distributed in linear scale. The purpose is, we want to know whether the network able to recognize any pattern or not. Then, we train the network with more than 3 $\times 10^6$ generated trajectories. The training performance saturated at MSE around 0.02.

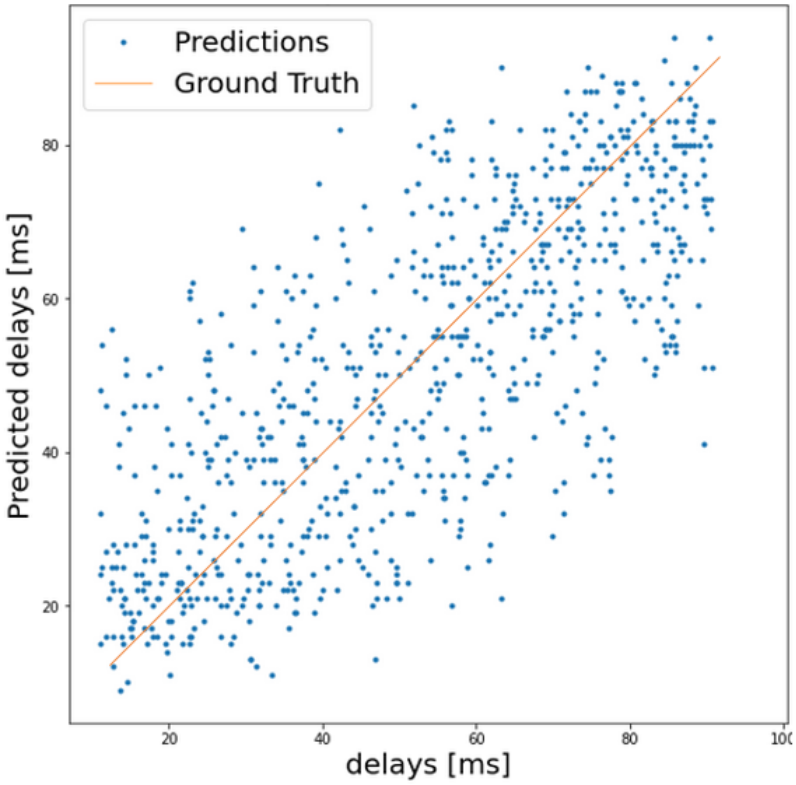


Figure 5.1: Predictions result with new approach: only using one output neuron to predict delay, and set the trajectory generator with the experiment parameters.

Performance test with another 1000 trajectories shows that the network still predict poorly. But, it is actually recognizing a pattern, seeing the major trend of the predictions is following the ground truth values, even though the results has wide range of error, for example, trajectories with 20 ms delayed feedback predicted in range of [10, 50] ms. Which also means, trajectories with delay in range [10,50] ms has similar pattern that recognized by the network. Thus, if we look at Fig. 5.1 we can conclude that for trajectory with specific delay, has a range of another delay values which gives similar pattern.

5.3 Prediction Result's Analysis

The particle's movement in over damped harmonic trap is depends on the stiffness of the trap. From equation (2.2), with same value of γ, the displacement will be smaller with large stiffness compare with small stiffness.

Now, the system that used in this project is described by equation (2.3), where the delayed feedback position also affect how the particle's movement. An example of this can be seen in Fig. 3.2, where the trapped particle's trajectory with delayed feedback implemented move in wider range of position even though the stiffness used in the simulation is the same. And since our model able to predict the stiffness properly even though the delay is different, now we compare how our model performance to predict the delay with constant stiffness value: 25.5 fN/μm. From this comparison, there are two different regions within our network predictions. The first region is where the delay predicted properly, the other region is where the delay predicted poorly. To gain better understanding, we feed the model with 100 trajectories with different stiffness 85, 50 and 8.5 fN/μm each with different delay in range[0,100]. The comparison between delay predictions of these four different stiffness value can be seen in Fig. 5.2.

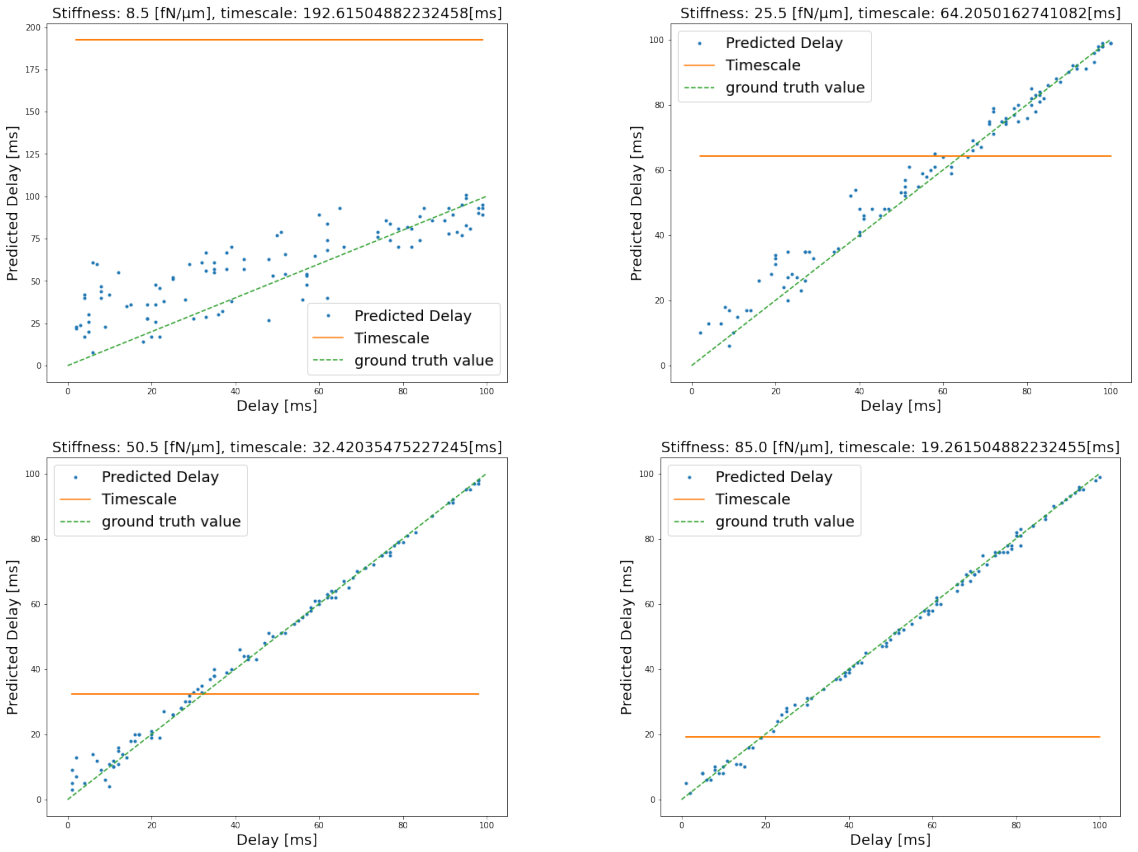


Figure 5.2: Comparing delay prediction with the optical tweezers' timescale τ. Each panel showing 100 predictions.

As we can see on Fig. 5.2 top-left one where the stiffness is 8.5 fN/μm and timescale 192 ms, which the same as the stiffness of the experiment data, all delay predictions have wide distance with the truth value. In the next panel, which show delay predictions for trajectories with stiffness 25.5 fN/μm, we see there is a region where the delay predictions become more precise. This precise region grows for the next two panels, where the stiffness value is 50.5 and 85.0 fN/μm respectively.

By adding the timescale for each trap used in Fig. 5.2, we can see there is a relation between the timescale and delay prediction. To remind us, as mentioned in Section 3.1, the timescale τ is the time on which the trap and optical forces drive the particle towards the potential energy minimum.

5.4 Trajectory Similarities Analysis

Deep neural network model that we employed here is trying to recognize a pattern in a sequence of data. Here, we want to investigate similarities between trajectories. As we have seen before in Fig. 2.2, delayed feedback made particle moves in wider range. But we can not compare particle's position directly, since for some stiffness, the range of positions is not too different to see. The only pattern that can be investigated is the displacement of the trapped particle between measurement points. The displacement is a value which affected by the stiffness and the delay directly. In order to study the pattern from trajectories, We compare trajectory in different regions marked by red and yellow box in Fig. 5.3. Red box represent properly predicted range of delay, and yellow box represent poorly predicted range of delay.

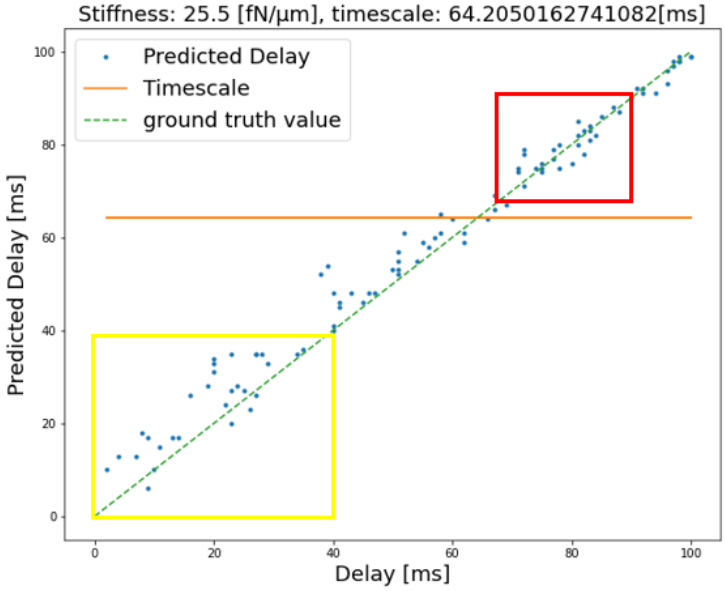


Figure 5.3: Two boxes (yellow and red) represent the range of parameters investigated to see how the delay affect the trajectory.

To see how the displacement behave along the time, it will be troublesome if we compare the displacement directly since it will be similar to trajectory plot. We also cannot use Mean Square Displacement (MSD), since our trapped particle's movement is constrained around the trap's energy minimum. Thus, we compare the total squared displacement (TSD) of trajectories. TSD described by the following equation:

$$TSD(t) = \sum_{i=0}^{t} (x(i+1) - x(i))^2 \tag{5.1}$$

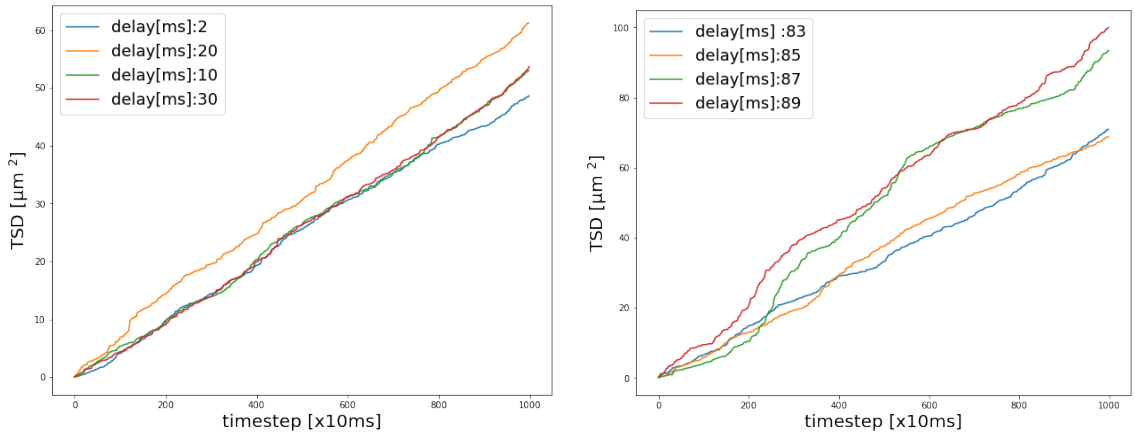


Figure 5.4: Total Squared Displacement (TSD) along the time. This analysis perform on trajectories with stiffness 25.5 fN/μm. Both panel shows TSD for 4 different trajectories in: left-panel) Yellow box region, right-panel) Red box region.

where TSD(t) is the Total Squared Displacement at time step t, and $x(i)$ is the particle's position at time i. We squared the displacement, so it will give positive value, and TSD will always increase with the time. With this method, trajectories with same stiffness and friction coefficient value, but different delay value yield similar slope of total squared displacement, it means the delay from these trajectories is harder to measure using machine learning technique due to similar pattern. In order to be able to recognize the similarities, we plot the TSD of 4 trajectories within each region in Fig. 5.3.

From the yellow box region, we pick 4 random trajectories, but we need to ensure that the delay value has large difference between each trajectory. Because we want to see in a wide range, whether the TSD of trajectories in this region give similar slope or not. As for trajectories from red box region, we do the opposite. Because we want to see whether similar delay values gives significant difference displacement or not.

3 of 4 Trajectories from the yellow box region gives similar slope as shown in the left-panel of Fig. 5.4. We can see that the TSD is similar for trajectories with delay 2, 10, 30 regardless of their significant difference in delay's value. Even though trajectories with delay 20 ms has higher TSD, but the slope is actually similar with the other three. On the contrary, trajectories from red box gives TSD that easy to recognize. It means that in this region, the trapped particle's displacement is easier to learn by the network.

After performing TSD analysis on simulated trajectories, we also need to check how similar is our experiment data. Using the same procedure, we found that the TSD of experiment data gives similar slope for all delay value. Considering experiment parameters that we obtain in Section 4.1, the timescale of optical tweezers used during experiment is 192 ms, which larger than delay implemented (10, 30, 40

ms). Based on TSD analysis as shown in Fig. 5.5, we can conclude that the experiment data has similar pattern which make it harder to measure the implemented delayed feedback using deep learning technique. Thus, delay prediction results of experiment data using deep learning technique as shown in Fig. 4.9 and in Fig. 5.1 is understandable.

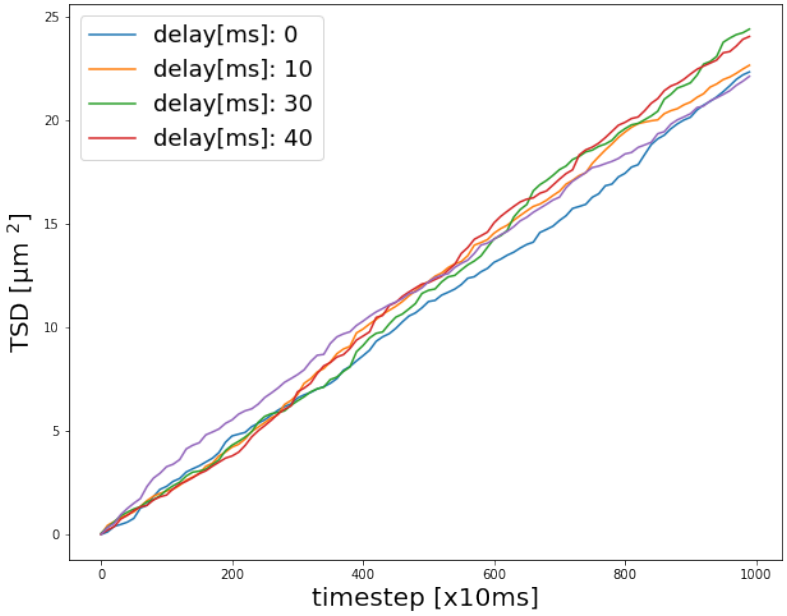


Figure 5.5: TSD of experiment data

5.5 2^{nd} Experiment

After getting the results using the first experiment, we then receive the second experiment result with larger delay in hope we can show that the network able to predict the delay properly when the delay is larger than the tweezers' timescale. In the second experiment, we have different set up of the optical tweezers, with delay 0 ms, 150 ms, 300 ms, 450 ms, 600 ms and 900 ms. In order to make sure that the network able to predict, we perform analysis as in Section 3.1 to get correct parameters to re-train the network if necessary.

By applying equation (3.1) on the second experiment's trajectory with 0 ms delay, we get the stiffness value, k, is 1.14 fN/μm, and by calculating autocorrelation function of the mentioned trajectory we get that the friction coefficient, τ, is 8.82 $\times 10^{-10}$ kg/s. From these values we can directly calculate the timescale of the optical tweezers used for the second experiment using:

$$\tau = \frac{\gamma}{k} \tag{5.2}$$

which gives τ : 773 ms.

Based on previous analysis in Section 5.3, using the second experiment's parameters, the network will be able to predict the experiment data with 900 ms delay.

But, it turns out it is not the case for our experiment data. The trained network still predicts the stiffness properly for all the experiment data. As for delay prediction, the network predictions have wide range of value for each data, even though if we look at the average the prediction is good for experiment data with 450 and 600 ms delay which the average of the delay predictions is 440 ms and 599.96 ms respectively. The details of the prediction on stiffness and delay of the experiment data with delay 300 ms, 450 ms and 600 ms are shown in Fig. 5.6.

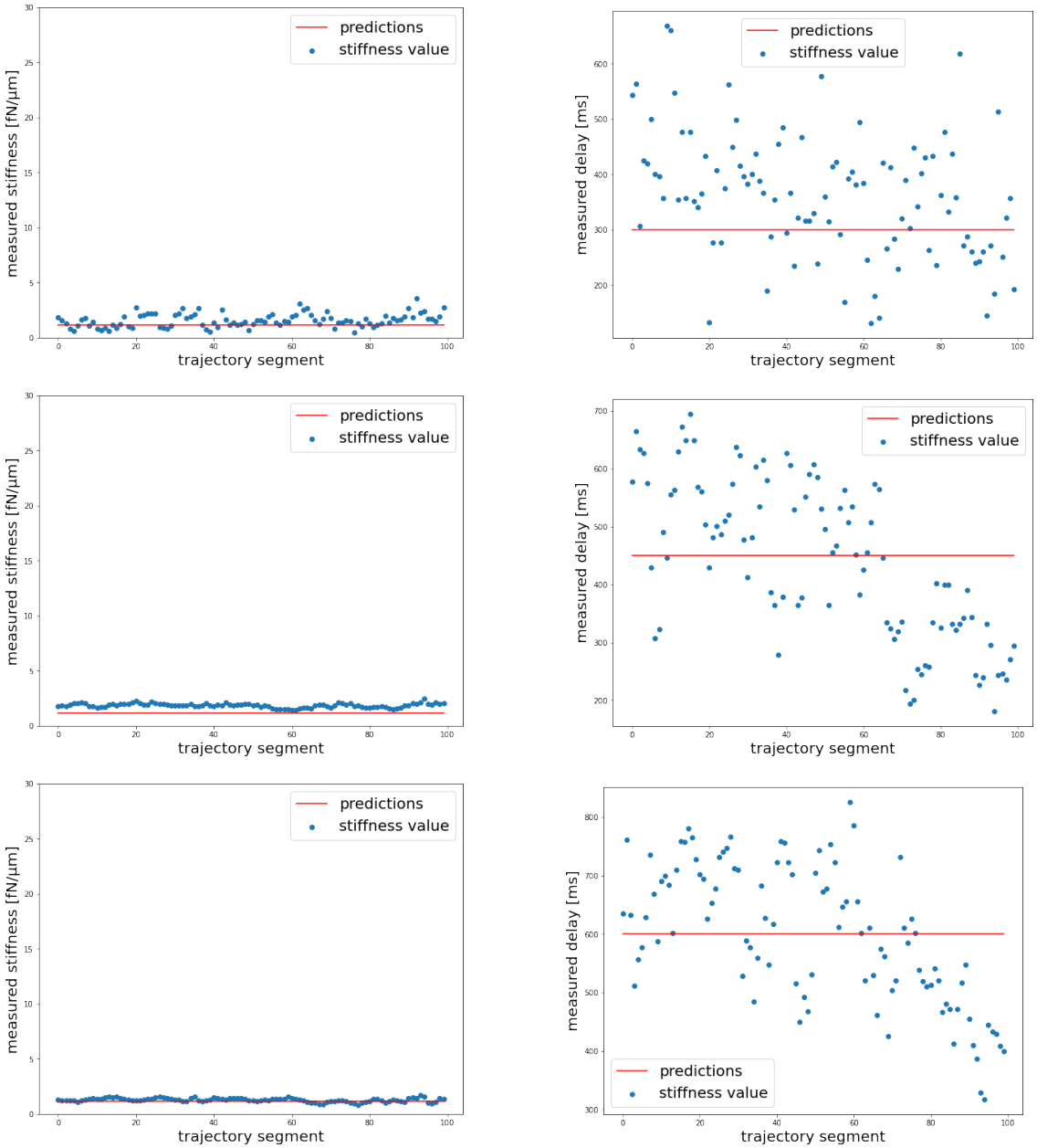


Figure 5.6: Predictions result using deep learning technique for the second experiment data with delay: Top row: 300 ms, middle row: 450 ms, and bottom row: 600 ms.

As for the experiment data with delay 900 ms, the delay prediction result is

not as good as we expected. As shown in Fig. 5.7, the network able to predict the stiffness properly, but for delay prediction the result is not good. The average value of predicted delay is 399.79 ms, which is not close to the truth value 900 ms.

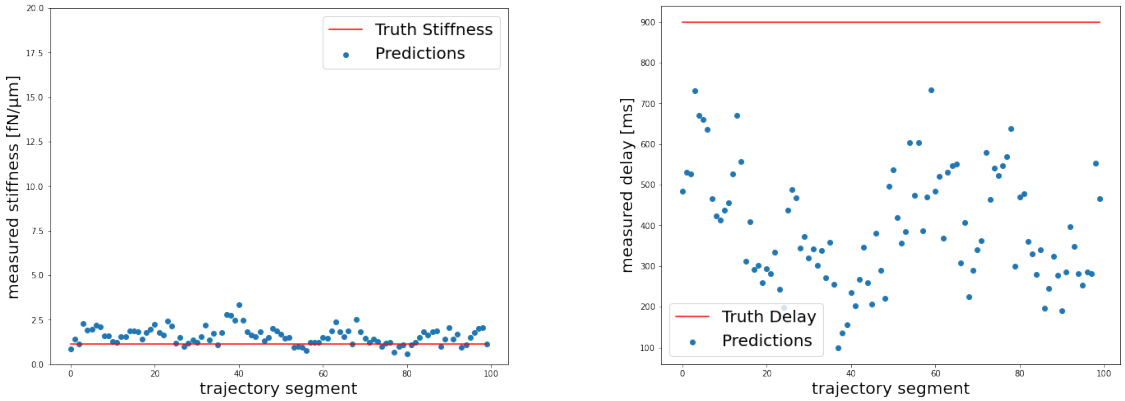


Figure 5.7: Predictions result using deep learning technique for the second experiment data with delay: 900 ms.

In Fig. 5.7 we can see that most of the trajectory segments from the experiment data with delay 900 ms predicted to trajectory with delay between 300 ms to 400 ms. By calculating the TSD, and comparing the experiment data with delay 900 ms with simulation trajectory with delay 300 ms, 350 ms, 400ms and 900 ms, we found that using these parameters gives different TSD for 900 ms delay. The generated trajectory with 900 ms delay is more similar to generated trajectory with 400 ms delay. We also found that the experiment data with 900 ms delay is more similar with generated trajectory with 350 ms compare with generated trajectory with 300 and 400 ms delay. The TSD comparison can be seen in Fig. 5.8.

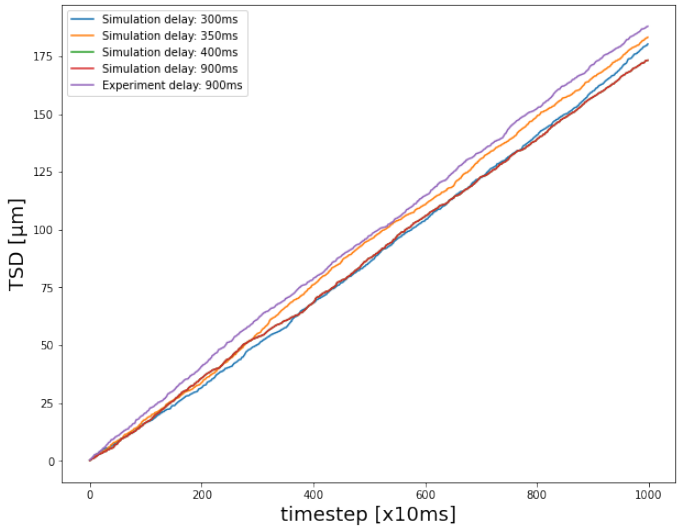


Figure 5.8: Comparing TSD of the experiment data with delay 900 ms, with generated trajectory which used experiment parameters based on variance method to calculate stiffness and autocorrelation function to estimate friction coefficient.

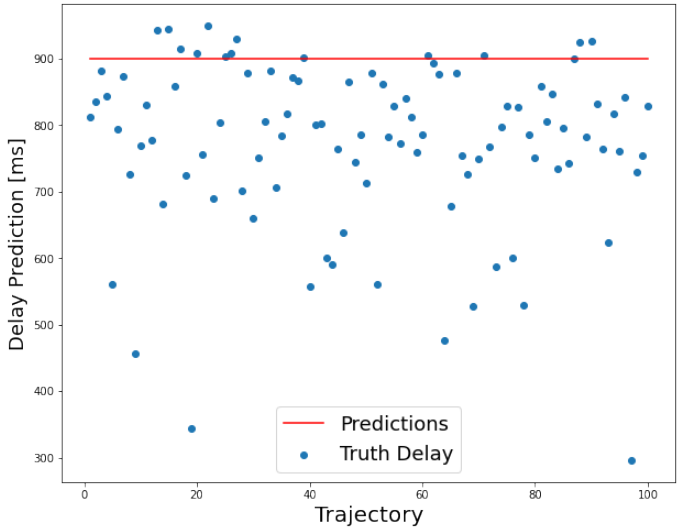


Figure 5.9: Prediction results on generated trajectories using the experiment parameters with 900 ms delay.

Then we generate 100 trajectories with the experiment parameters and delay 900 ms, then feed it into the trained network to get the predictions in order to check the network performance in measuring the delay of generated trajectories using these parameters. In general, as shown in Fig. 5.9, the delay prediction result showing similar behavior with what we got when we measure the experiment data with 900 ms delay, where the delay predicted to be smaller than 900 ms. But, the performance of the network is better in measuring the delay of generated trajectory.

Even though the result is better compared to prediction on the experiment data, with the average of predicted delay is 772.03 ms, the network still predicting the delay in wide range. As we can see in Fig. 5.9, there are some generated trajectories predicted to have delay in range [300, 500] ms. This is confirming that the TSD analysis is correct, where the generated trajectory with 900 ms delay have similarities with generated trajectory with delay in range of [300,400] ms as shown in Fig. 5.8.

6

Conclusion

After analyzing the results from all methods present in this thesis, we can conclude that:

- Deep Learning technique outperform variance method in calibrating the stiffness of optical tweezers with delayed feedback.
- Deep Learning technique outperform Approximation method in measuring the delay based on trapped particle's trajectory.
- Deep Learning delay predictions performance is decreasing when the delay is smaller than optical tweezers timescale τ, Because the displacement of this kind of trajectory is similar based on TSD analysis.
- Even though the delay predictions gives wide deviation for delay smaller than timescale, but the network did recognize some pattern. As shown in Fig. 5.1 and the top left-panel of Fig. 5.2, where the slope of the delay predictions is actually following the delay truth value.

Bibliography

[1] J. Gieseler, J. R. Gomez-Solano, A. Magazz'u, I. P. Castillo, L. P. Garc´ıa, M. Gironella-Torrent, X. ViaderGodoy, F. Ritort, G. Pesce, A. V. Arzola, et al., (2020) "Optical tweezers: A comprehensive tutorial from calibration to applications,".

[2] Aykut Argun, Tobias Thalheim, Stefano Bo, Frank Cichos and Giovanni Volpe (2020) Enhanced force-field calibration via machine learning

[3] S. M. J. Khadem and Sabine H. L. Klapp (2019) Delayed Feedback Control of Active Particles: a Controlled Journey Towards the Destination. https://pubs.rsc.org/en/content/articlehtml/2019/cp/c9cp00495e

[4] Pyragas Kestutis (2006) Delayed feedback control of chaos Phil. Trans. R. Soc. A.3642309–2334. https://doi.org/10.1098/rsta.2006.1827

[5] Göran Wahnström (2021) "Brownian Dynamics Lecture Notes".

[6] G. Volpe and G. Volpe (2013) "Simulation of a Brownian particle in an optical trap," Am. J. Phys. 81, 224–231.

[7] Giovanni Volpe, Dmitri Petrov (2006) Torque Detection using Brownian Fluctuations. DOI: 10.1103/PhysRevLett.97.210603

[8] Christian Janiesch, Patrick Zschech, Kai Heinrich (2021) Machine Learning and Deep Learning. https://doi.org/10.1007/s12525-021-00475-2

[9] Bishop, C. M. (2006). Pattern recognition and machine learning (Information science and statistics). Springer-Verlag New York, Inc.

[10] Osvaldo Simeone (2018) A Very Brief Introduction to Machine Learning With Applications to Communication Systems. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=8542764

[11] Emmert-Streib Frank, Yang Zhen, Feng Han, Tripathi Shailesh, Dehmer Matthias (2020) An Introductory Review of Deep Learning for Prediction Models With Big Data. Frontiers in Artificial Intelligence. https://doi.org/10.3389/frai.2020.00004

[12] Hinton, G. E., and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. Science 313, 504–507. doi: 10.1126/science.1127647

[13] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. Nature 521:436.

[14] Shagan Sah (2020) Machine Learning: A Review of Learning Type.

[15] Qiong Liu, Ying Wu (2012) Supervised Learning. https://doi.org/10.1007/978-1-4419-1428-6_451

[16] Farhana Sultana, Abu Sufian, Paramartha Dutta (2019) Advancements in Image Classification using Convolutional Neural Network. https://arxiv.org/pdf/1905.03288.pdf

[17] Pengfei Liu, Xipeng Qiu, Xuanjing Huang. Recurrent Neural Network for Text Classification with Multi-Task Learning. https://www.ijcai.org/Proceedings/16/Papers/408.pdf

[18] Kyunghyun Cho, Bart van Merri enboer ,Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio (2014) Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. https://arxiv.org/pdf/1406.1078.pdf

[19] Sepp Hochreiter, Jurgen Schmidhuber (1997) Long-Short Term Memory. Neural Computation 9(8): 1735-1780. https://www.bioinf.jku.at/publications/older/2604.pdf

[20] Hemanth Pedamallu (2020) RNN vs GRU vs LSTM. https://medium.com/analytics-vidhya/rnn-vs-gru-vs-lstm-863b0b7b1573

[21] Shudong Yang, Xueying Yu, Ying Zhou (2020) LSTM and GRU Neural Network Performance Comparison Study. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=9221727

[22] Tong Yu, Hong Zhu (2020) Hyper-Parameter Optimization: A Review of Algorithms and Applications. https://arxiv.org/pdf/2003.05689.pdf

[23] Andrew Ng. Improving deep neural networks: Hyperparameter tuning, regularization and optimization. Deeplearning. ai on Coursera, 2017.

[24] P. H. Jones, O. M. Marag'o, and G. Volpe, Optical tweezers: Principles and applications (Cambridge University, 2015)

[25] Pesce, G., Jones, P.H., Maragò, O.M. et al. Optical tweezers: theory and practice. Eur. Phys. J. Plus 135, 949 (2020). https://doi.org/10.1140/epjp/s13360-020-00843-5

[26] Göran Wahnström (2021) "Monte Carlo Lecture Notes".

[27] Kaichao You, Mingsheng Long, Jianmin Wang, Michael I. Jordan (2019) HOW DOES LEARNING RATE DECAY HELP MODERN NEURAL NETWORKS. https://arxiv.org/pdf/1908.01878.pdf

[28] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying Quoc V. Le (2018) DON'T DECAY THE LEARNING RATE, INCREASE THE BATCH SIZE. ICLR 2018. https://openreview.net/pdf?id=B1Yy1BxCZ

[29] A. Argun, T. Thalheim, S. Bo, F. Cichos, and G. Volpe, "DeepCalib," http://github.com/softmatterlab/DeepCalib (2020)

A

Appendix

June 10, 2022

1 HARMONIC TRAP - EXPERIMENT DATA

Now we have 4 different experiment data: 1. Delay 0 ms 2. Delay 10 ms 3. Delay 30 ms 4. Delay 50 ms

We want to try our program to predict the stiffness and the delay of the experiment data

[ ]:
```
import DeepCalib

%matplotlib inline
from math import pi
from numpy import log
from numpy import exp
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.constants import Boltzmann as kB
import scipy
from scipy import optimize
from keras.models import load_model
```

2 1. CHECKING THE DATA

First lets load the trajectory from experiment to see how the trajectory look like. Here we load the trajectory with delay 0 ms.

We can also use this trajectory to estimate the stiffness. But, from experimental set up, we dont have information on temperature so we assume the temperature is: 300 K.

to get the stiffness, we use - k = kb T/ < x2 >

[ ]:
```
path_file = "D:\Thesis\Experiment\coordinates_delay_000ms.txt"

array = np.genfromtxt(path_file, skip_header=1)
# array = list(array)
# del array[32643]
# del array[32643]
# array = np.array(array)
```

```
x = (array[:,0] - np.mean(array[:,0])) * 1e-6
y = (array[:,1] - np.mean(array[:,1])) * 1e-6

plt.plot(x)
plt.figure()
plt.plot(y)
```

3 Estimate stiffness

Since it is actually contain nan on indices 32643 and 32644 with both approachs (numpy and panda), now we can estimate stiffness using what we have

[ ]:
```
kb = 1.3806 * 1e-23
T = 300
sum_x = 0
sum_y = 0

for i in range(len(x_subs)):
    sum_x += np.square(x[i])
    sum_y += np.square(y[i])

mean_x_squared_sum = sum_x / len(x_subs)
mean_y_squared_sum = sum_y / len(y_subs)

k_estimate_x = (kb * T)/mean_x_squared_sum
k_estimate_y = (kb * T)/mean_y_squared_sum

print(k_estimate_x)
print(k_estimate_y)
```

[ ]:
```
def acf_by_hand(data, lag):
    # Slice the relevant subseries based on the lag
    y1 = x[:(len(x)-lag)]
    y2 = x[lag:]
    # Subtract the mean of the whole series x to calculate Cov
    sum_product = np.sum((y1-np.mean(x))*(y2-np.mean(x)))
    # Normalize with var of whole series
    return sum_product / ((len(x) - lag) * np.var(x))
```

[ ]:
```
ndata = len(x)
nlags = 310
acf = np.zeros(nlags)
lag_array = np.zeros(nlags)

for i in range(nlags):
    acf[i] = acf_by_hand(x, i)
    lag_array[i] = i
```

```
# lag_array = lag_array[acf>0.1]
# acf = acf[acf>0.1]
coeff = np.polyfit(lag_array, np.log(acf),1)
print(coeff)
est_acf = np.exp(coeff[1]) * np.exp(coeff[0]*lag_array)

plt.plot(lag_array, acf)
plt.plot(lag_array, est_acf, '--')
```

[ ]:
```
est_gamma = (0.01 * k_estimate_y) / (-coeff[0])

print(est_gamma)
```

[ ]:
```


l = (np.arange(100)-50)/20e6
hx = np.histogram(x,l)
hy = np.histogram(y,l)

plt.plot((l[:-1]+l[1:])/2,-np.log(hx[0]/np.max(hx[0]))*kb*T)
plt.plot((l[:-1]+l[1:])/2,-np.log(hy[0]/np.max(hx[0]))*kb*T)

k = 1.61e-9
plt.plot((l[:-1]+l[1:])/2, k*(l[:-1]+l[1:])**2/8, "--")
```

3.1 From the experiment result, we have stiffness used in this experiment is 8.49 fN.

Now we have to estimate the friction coefficient (gamma) used in this experiment. to estimate gamma, we use autocorrelation method.

[ ]:
```
ndata = len(x)
nlags = 200
x1 = x
x2 = x
acf = np.zeros(nlags)
lag_array = np.zeros(nlags)

def acf_by_hand(data, lag):
    # Slice the relevant subseries based on the lag
    y1 = x[:(len(x)-lag)]
    y2 = x[lag:]
    # Subtract the mean of the whole series x to calculate Cov
    sum_product = np.sum((y1-np.mean(x))*(y2-np.mean(x)))
    # Normalize with var of whole series
    return sum_product / ((len(x) - lag) * np.var(x))
```

[ ]:
```
for i in range(nlags):
    acf[i] = acf_by_hand(x, i)
    lag_array[i] = i

# scipy.optimize.curve_fit(lambda t,a,b: a*np.exp(b*t),  lag_array,  acf)
# /coeff = scipy.optimize.curve_fit(lambda t,a,b: a*np.exp(b*t),  lag_array,
 ↪acf,  p0=(4, 0.01))
#

lag_array = lag_array[acf>0.1]
acf = acf[acf>0.1]
coeff = np.polyfit(lag_array, np.log(acf),1)
```

[ ]:
```
plt.plot(lag_array, np.log(acf), '.',lag_array, lag_array*coeff[0] + coeff[1])

plt.legend(['ACF trajectory based', 'ACF Polyfit Approximation'])
plt.title('Autocorrelation')
plt.xlabel('lags')
plt.ylabel('Autocorrelation')
```

[ ]:
```
k
```

[ ]:
```
gamma_est = -k*0.01/coeff[0]
print(gamma_est)
```

4 2. Trajectory Generator with same parameter as experiment

Here we need to train the network with trajectories generated using a function which implement experiment parameters: - stiffness : 8.49 fN - particle's diameter : 200 nm - dt : 1 ms - temperature : 300 K

[ ]:
```
### Physical parameters
R = 1e-7                    # Radius of the Brownian particle [m]
eta = 0.001                 # Viscosity of the medium [kg m^-1 s^-1]
T = 300                     # Temperature [K]
k0 = 8.49                     # Reference stiffness [fN \mu m ^-1]
gamma0 = gamma_est #6 * pi * eta * R  # Reference friction coefficient [kg s^-1]
delay0 = 100                # Maximum delay in log10. So its = 100

## Simulation parameters
dt = 1e-3
Dt = 1e-3
Nparticle = 32
pre_simulation = 1000
Niter = 10000
oversampling = int(dt/Dt)
```

```
##scaling function
scale_inputs = lambda x_pos: x_pos * 1e+6
rescale_inputs = lambda scaled_x_pos: scaled_x_pos * 1e-6

scale_targets = lambda k, delays: [k /100, delays/delay0]
rescale_targets = lambda scaled_k, scaled_delay: [scaled_k*100, np.
 ↪round(scaled_delay*delay0)-1]

def simulate_trajectory(Nparticle=Nparticle,
                        delay0 = delay0,
                        T=T,
                        gamma0=gamma0,
                        k0=k0,
                        dt=dt,
                        pre_simulation=pre_simulation,
                        oversampling=oversampling,
                        Niter = Niter):

    import numpy as np
    from scipy.constants import Boltzmann as kB
    from math import pi
    from math import sqrt
    from numpy.random import randn as gauss
    from numpy.random import rand as uniform
    from numpy.random import randint

    k = uniform(Nparticle) * k0 * 10
#     k = 4.5 + np.random.rand(Nparticle) * k0 + np.random.rand(Nparticle) * 15
    gamma = gamma0* (uniform(Nparticle) * .1 + .95)
    tscale = (1e9*gamma)/k

    Dt = dt / oversampling
    Diff = kB * T / gamma
    first_term = - k * 1e-9 / gamma * Dt
    second_term = np.sqrt(2*Diff*Dt)
    time = np.zeros(Niter)
    n = 0

    #initialize delay array
    x_delay = np.zeros((Nparticle, delay0+1))
    delays = delay0 * uniform(Nparticle) + .95

    delays_calc = np.round(delays)                                    # [1,100]
    delays_target = delays

    #initialize position of the particle
    x_pos = np.zeros((Nparticle, Niter))
```

```
    X = x_pos[:,0]

    X_MAX = 1e-5
    X_MIN = -1e-5

    x_prev = np.zeros(Nparticle)

    for t in range(pre_simulation):

        for i in range(Nparticle):
            x_prev[i] = x_delay[i, -int(delays_calc[i])]

        X = X + first_term * x_prev + second_term * gauss(Nparticle)

        if any (X > X_MAX):
            arr = np.where(X > X_MAX)
            X[arr] = X_MAX - 1e-6
        elif any (X < X_MIN):
            arr = np.where(X < X_MIN)
            X[arr] = X_MIN + 1e-6

        #update delay_array
        x_delay = np.roll(x_delay, -1)
        x_delay[:,-1] = X

    #generate trajectory with delay
    n = 0
    for t in range(Niter * oversampling):

        for i in range(Nparticle):
            x_prev[i] = x_delay[i, -int(delays_calc[i])]

        X = X + first_term * x_prev + second_term*gauss(Nparticle)

        if any (X > X_MAX):
            arr = np.where(X > X_MAX)
            X[arr] = X_MAX - 1e-6
        elif any (X < X_MIN):
            arr = np.where(X < X_MIN)
            X[arr] = X_MIN + 1e-6

        #update delay_array
        x_delay = np.roll(x_delay, -1)
        x_delay[:,-1] = X

        #save position
#         x_pos[:,t] = X
```

```
        if t % oversampling == 0:
            x_pos[:,n] = X
            n += 1

    # Normalize trajectory and targets

    inputs = DeepCalib.trajectory(
        names=['x'],
        values=x_pos,
        scalings=['x [\u03BCm]'],
        scaled_values=scale_inputs(x_pos))

    targets = DeepCalib.targets(
        names=['k [fN/\u03BCm]', 'delay [ms]'],
        values=np.swapaxes([k, delays_calc-1],0,1),
        scalings=['k/100)', 'delay/delay0'],
        scaled_values=np.swapaxes(scale_targets(*[k, delays_target]),0,1))

    # time scale
    timescale = np.divide(gamma, k)
    return inputs, targets
```

[ ]:
```
### Show some examples of simulated trajectories

number_of_trajectories_to_show = 10
DeepCalib.plot_sample_trajectories(simulate_trajectory,␣
 ↪number_of_trajectories_to_show)
```

5 3. Build and Train the network with Experiment's Parameters

Try using ReLU for activation function in the output layer

[ ]:
```
### To make sure we use the newest network architecture, we put import code␣
 ↪again here

import DeepCalib
```

[ ]:
```
### Define parameters of the deep learning network

input_shape = (None, 10000)
lstm_layers_dimensions = (1024, 256, 64)
number_of_outputs = 2

### Create deep learning network

network = DeepCalib.create_deep_learning_network(input_shape,␣
 ↪lstm_layers_dimensions, number_of_outputs)
```

```
### Print deep learning network summary

network.summary()
```

[ ]:
```
### Define parameters of the training

sample_sizes = (32, 128, 512, 1024)
iteration_numbers = (501, 1001, 1001, 1001)
verbose = .01

### Training
training_history = DeepCalib.train_deep_learning_network(network,␣
 ↪simulate_trajectory, sample_sizes, iteration_numbers, verbose)
```

[ ]:
```
### Plot learning performance

number_of_timesteps_for_average = 100

DeepCalib.plot_learning_performance(training_history,␣
 ↪number_of_timesteps_for_average)
```

[ ]:
```
### Test the predictions of the deep learning network on some generated␣
 ↪trajectories
number_of_predictions_to_show = 1000
%matplotlib inline
DeepCalib.plot_test_performance(simulate_trajectory, network, rescale_targets,␣
 ↪number_of_predictions_to_show)
```

[ ]:
```
### Define parameters of the training

sample_sizes = (1024, 2048)
iteration_numbers = (1001, 1001)
verbose = .01

### Training
training_history = DeepCalib.train_deep_learning_network(network,␣
 ↪simulate_trajectory, sample_sizes, iteration_numbers, verbose)
```

[ ]:
```
save_file_name = 'TEST4.h5'
network.save(save_file_name)
```

6 4. PREDICT THE EXPERIMENT TRAJECTORY

Now, since we already have a great result for small k, we we try to predict the stiffness of the experiment using our trained network.

*For playing, we create test_trajectory generator which used only 1 value of delay or single value of k

[ ]:
```
model = load_model('TEST2.h5')
```

7 TRY PREDICT THE TRAJECTORY WITH DELAY FEEDBACK

[ ]:
```
path_file = "D:\Thesis\Experiment\coordinates_delay_150ms.txt"

array = np.genfromtxt(path_file, skip_header=1)
x_del = array[:,0]
y_del = array[:,1]
mean_x = np.mean(x_del)
mean_y = np.mean(y_del)
x_subs = x_del - mean_x
y_subs = y_del - mean_y
print(mean_x)
print(mean_y)

fig, axs = plt.subplots(2)
axs[0].plot(x_subs)
axs[1].plot(y_subs)
```

Now lets predict the experiment trajectory using trained network

[ ]:
```
predictions_delay = []
predictions_k = []
ndata = len(x_subs)
ninput = 1000
oversam = 1
nmeas = 100
steps = int((ndata-1000*oversam)/nmeas)
slength = oversam*ninput
points = []

for i in range(nmeas):
    x = x_subs[(i*steps):(i*steps+slength):oversam]
    predicted_k, predicted_delay = DeepCalib.predict(model, x)[0]
    predictions_k.append(predicted_k)
    predictions_delay.append(predicted_delay)
    points.append(i)

[predictions_k, predictions_delay] = rescale_targets(*np.array([predictions_k,␣
 ↪predictions_delay]))
```

[ ]:
```
k_truth = np.zeros(nmeas)
tau_truth = np.zeros(nmeas)
```

```
plt.figure(figsize=(10,8))
plt.scatter(points, predictions_k)
plt.plot(points, k_truth, 'r')
plt.xlabel('trajectory segment', fontsize=20)
plt.ylabel('measured stiffness [fN/\u03BCm]', fontsize=20)
plt.legend(['predictions', 'stiffness value'], fontsize=20)
plt.ylim([0,100])
```

[ ]:
```
plt.figure(figsize=(10,8))
plt.scatter(points, predictions_delay)
plt.plot(points, tau_truth, 'r')
plt.xlabel('trajectory segment', fontsize=20)
plt.ylabel('measured delay [ms]', fontsize=20)
plt.legend(['predictions', 'stiffness value'], fontsize=20)
plt.ylim([0,100])
```

8 PLAYING

[ ]:
```
### Physical parameters
R = 1e-7                    # Radius of the Brownian particle [m]
eta = 0.001                 # Viscosity of the medium [kg m^-1 s^-1]
T = 300                     # Temperature [K]
k0 = 8.49                     # Reference stiffness [fN \mu m ^-1]
gamma0 = gamma_est #6 * pi * eta * R  # Reference friction coefficient [kg s^-1]
delay0 = 100                # Maximum delay in log10. So its = 100

## Simulation parameters
dt = 1e-2
Nparticle = 32
pre_simulation = 1000
Niter = 1000
oversampling = 10

##scaling function
scale_inputs = lambda x_pos: x_pos * 1e+6
rescale_inputs = lambda scaled_x_pos: scaled_x_pos * 1e-6

scale_targets = lambda delays: delays/delay0
rescale_targets = lambda scaled_delay: np.round(scaled_delay*delay0)-1

def simulate_trajectory(Nparticle=32,
                        delay0 = delay0,
                        T=T,
                        gamma0=gamma0,
                        k0=k0,
```

```
                        dt=dt,
                        pre_simulation=pre_simulation,
                        oversampling=oversampling,
                        Niter = Niter):

    import numpy as np
    from scipy.constants import Boltzmann as kB
    from math import pi
    from math import sqrt
    from numpy.random import randn as gauss
    from numpy.random import rand as uniform
    from numpy.random import randint

    k = np.zeros(Nparticle) + 8.5
#     k = 4.5 + np.random.rand(Nparticle) * k0 + np.random.rand(Nparticle) * 15
    gamma = gamma0* (uniform(Nparticle) * .1 + .95)
    tscale = (1e9*gamma)/k

    Dt = dt / oversampling
    Diff = kB * T / gamma
    first_term = - k * 1e-9 / gamma * Dt
    second_term = np.sqrt(2*Diff*Dt)
    time = np.zeros(Niter)
    n = 0

    #initialize delay array
    x_delay = np.zeros((Nparticle, delay0+1))
    delays = delay0 * uniform(Nparticle) + .95

    delays_calc = np.round(delays)                                    # [1,100]
    delays_target = delays

    #initialize position of the particle
    x_pos = np.zeros((Nparticle, Niter))
    X = x_pos[:,0]

    X_MAX = 1e-5
    X_MIN = -1e-5

    x_prev = np.zeros(Nparticle)

    for t in range(pre_simulation):

        for i in range(Nparticle):
            x_prev[i] = x_delay[i, -int(delays_calc[i])]

        X = X + first_term * x_prev + second_term * gauss(Nparticle)
```

```
        if any (X > X_MAX):
            arr = np.where(X > X_MAX)
            X[arr] = X_MAX - 1e-6
        elif any (X < X_MIN):
            arr = np.where(X < X_MIN)
            X[arr] = X_MIN + 1e-6

        #update delay_array
        x_delay = np.roll(x_delay, -1)
        x_delay[:,-1] = X

    #generate trajectory with delay
    n = 0
    for t in range(Niter * oversampling):

        for i in range(Nparticle):
            x_prev[i] = x_delay[i, -int(delays_calc[i])]

        X = X + first_term * x_prev + second_term*gauss(Nparticle)

        if any (X > X_MAX):
            arr = np.where(X > X_MAX)
            X[arr] = X_MAX - 1e-6
        elif any (X < X_MIN):
            arr = np.where(X < X_MIN)
            X[arr] = X_MIN + 1e-6

        #update delay_array
        x_delay = np.roll(x_delay, -1)
        x_delay[:,-1] = X

        #save position
#         x_pos[:,t] = X
        if t % oversampling == 0:
            x_pos[:,n] = X
            n += 1

    # Normalize trajectory and targets

    inputs = DeepCalib.trajectory(
        names=['x'],
        values=x_pos,
        scalings=['x [\u03BCm]'],
        scaled_values=scale_inputs(x_pos))

    targets = DeepCalib.targets(
```

```
        names=['delays [ms]'],
        values=delays_target,
        scalings=['delays / delays0)'],
        scaled_values=scale_targets(delays_target))

    # time scale
    timescale = np.divide(gamma, k)
    return inputs, targets
```

[ ]:
```
### Show some examples of simulated trajectories

number_of_trajectories_to_show = 10
DeepCalib.plot_sample_trajectories(simulate_trajectory,␣
↪number_of_trajectories_to_show)
```

[ ]:
```
### Define parameters of the deep learning network

input_shape = (None, 1000)
lstm_layers_dimensions = (1024, 256, 64)
number_of_outputs = 1

### Create deep learning network

network = DeepCalib.create_deep_learning_network(input_shape,␣
↪lstm_layers_dimensions, number_of_outputs)

### Print deep learning network summary

network.summary()
```

[ ]:
```
### Define parameters of the training

sample_sizes = (32, 128, 512, 1024)
iteration_numbers = (501, 1001, 1001, 1001)
verbose = .01

### Training
training_history = DeepCalib.train_deep_learning_network(network,␣
↪simulate_trajectory, sample_sizes, iteration_numbers, verbose)
```

[ ]:
```
network = load_model('PLAYING.h5')
```

[ ]:
```
### Test the predictions of the deep learning network on some generated␣
↪trajectories
number_of_predictions_to_show = 1000
%matplotlib inline
```

```
DeepCalib.plot_test_performance(simulate_trajectory, network, rescale_targets,␣
 ↪number_of_predictions_to_show)
```

[ ]:
```
### Define parameters of the training

sample_sizes = (1024, 2048)
iteration_numbers = (1001, 1001)
verbose = .01

### Training
training_history = DeepCalib.train_deep_learning_network(network,␣
 ↪simulate_trajectory, sample_sizes, iteration_numbers, verbose)
```

[ ]:
```
save_file_name = 'PLAYING.h5'
network.save(save_file_name)
```

[ ]:
```

```

[ ]:
```

```

June 10, 2022

1 Rotational trap with Delay

1.1 1. INIZIALIZATION

[ ]:
```
import DeepCalib

%matplotlib inline
from numpy import log10
from numpy import log
from numpy import exp
from math import pi
import numpy as np
```

1.2 2. DEFINE TRAJECTORY SIMULATION

Here the function that simulates the motion of the Brownian particle in the force field under consideration is defined. Specifically, in this case, we consider a Brownian particle in a rotational force field, and the motion of the particle depends on the the trap radial component k and the rotational component M.

This file is used to reproduce results that are shown in the numerical section of Fig.5 and generate the pretrained network "Network_Example_3a" that is going to be needed to execute Example 3b.

Comments: 1. The function that simulates the trajectories must be called simulate_trajectory. 2. Lambda functions scale_inputs, rescale_inputs, scale_targets, and rescale_targets must also be defined. For the best performance of the learning, the rescaling of both the inputs and targets should lead to values of order 1.

[ ]:
```
### Physical parameters
R = 1e-7                    # Radius of the Brownian particle [m]
eta = 0.001                 # Viscosity of the medium [kg m^-1 s^-1]
T = 300                     # Temperature [K]
k0 = 10                     # Reference stiffness [fN m^-1]
M0 = 20                     # Reference rotational coefficient [N m^-1]
gamma0 = 1.3 * 6 * pi * eta * R  # Reference friction coefficient [kg s^-1]
delay0 = 100

### Simulation parameters
N = 1000                    # Number of samples of the trajectory
```

```
Dt = 5e-2                    # Timestep
dt = 5e-3
oversampling = int(Dt/dt)  # Simulation oversampling
t_eq = 10*gamma0/k0
offset = 1000                # Number of equilibration points

### Define functions to scale and rescale inputs
scale_inputs = lambda x, y: [x * 1e+6, y * 1e+6]     # Scales input trajectory␣
 ↪to order 1
rescale_inputs = lambda scaled_x: [scaled_x * 1e-6,
                                   scaled_y * 1e-6]  # Rescales input trajectory␣
 ↪to physical units


### Define function to scale and rescale targets
scale_targets = lambda k, M, delays: [k/delay0, (M+50)/delay0, delays/delay0]  ␣
 ↪                    # Scales targets to order 1
rescale_targets = lambda scaled_k, scaled_M, scaled_delays: [scaled_k * delay0,
                                                    (scaled_M*delay0)-50,␣
 ↪scaled_delays*delay0] # Inverse of targets_scaling

### Define the simulate_trajectory function

def simulate_trajectory(batch_size=32,
                        T=T,
                        k0=k0,
                        M0=M0,
                        gamma0=gamma0,
                        delay0=delay0,
                        N=N,
                        Dt=Dt,
                        oversampling=oversampling,
                        offset=offset,
                        scale_inputs=scale_inputs,
                        scale_targets=scale_targets):

    import numpy as np
    from scipy.constants import Boltzmann as kB
    from math import pi
    from math import sqrt
    from numpy.random import randn as gauss
    from numpy.random import rand as uniform

    ### Randomize trajectory parameters

    k = k0 * 10 * uniform(batch_size)
    M = M0 * (uniform(batch_size)*5 -2.5)
```

```
    gamma = gamma0 * (uniform(batch_size)*0.2 + .9)

    delays = delay0 * uniform(batch_size) + .95

    delay_calc = np.round(delays)

    X_MAX = 1e-5
    X_MIN = -1e-5
    Y_MAX = 1e-5
    Y_MIN = -1e-5
    ### Simulate

    dt = Dt / oversampling
    x = np.zeros((batch_size, N))
    y = np.zeros((batch_size, N))
    D = kB * T / gamma
    C1 = -k * 1e-9 / gamma * dt
    C2 = -M * 1e-9 / gamma * dt
    C3 = np.sqrt(2 * D * dt)

    x_delay = np.zeros((batch_size, delay0+2))
    y_delay = np.zeros((batch_size, delay0+2))

    X_PREV = np.zeros(batch_size)
    Y_PREV = np.zeros(batch_size)
    X = x[:,0]
    Y = y[:,0]
    n = 0

    for t in range(offset):                        # Offset

        for i in range(batch_size):
            X_PREV[i] = x_delay[i, -int(delay_calc[i])]
            Y_PREV[i] = y_delay[i, -int(delay_calc[i])]

        X = X + C1 * X_PREV - C2 * Y_PREV + C3 * gauss(batch_size)
        Y = Y + C1 * Y_PREV + C2 * X_PREV + C3 * gauss(batch_size)
#         X = X + C1 * X - C2 * Y + C3 * gauss(batch_size)
#         Y = Y + C1 * Y + C2 * X + C3 * gauss(batch_size)

        if any (X > X_MAX):
            arr = np.where(X > X_MAX)
            X[arr] = X_MAX - 1e-6
        elif any (X < X_MIN):
            arr = np.where(X < X_MIN)
            X[arr] = X_MIN + 1e-6
```

```
        if any (Y > Y_MAX):
            arr = np.where(Y > Y_MAX)
            Y[arr] = Y_MAX - 1e-6
        elif any (Y < Y_MIN):
            arr = np.where(Y < Y_MIN)
            Y[arr] = Y_MIN + 1e-6
#         print(X_PREV)
        #update delay
        x_delay = np.roll(x_delay, -1)
        x_delay[:,-1] = X
        y_delay = np.roll(y_delay, -1)
        y_delay[:,-1] = Y
#         print(x_delay)

    for t in range(N * oversampling):             # Simulation
#         print('current x', X)
#         print('current y', Y)
        for i in range(batch_size):
            X_PREV[i] = x_delay[i, -int(delay_calc[i])]
            Y_PREV[i] = y_delay[i, -int(delay_calc[i])]
#         print('x prev', X_PREV)
#         print('y prev', Y_PREV)
        X = X + C1 * X_PREV - C2 * Y_PREV + C3 * gauss(batch_size)
        Y = Y + C1 * Y_PREV + C2 * X_PREV + C3 * gauss(batch_size)
#         X = X + C1 * X - C2 * Y + C3 * gauss(batch_size)
#         Y = Y + C1 * Y + C2 * X + C3 * gauss(batch_size)
#         print('x new', X)
#         print('y new', Y)

        if any (X > X_MAX):
            arr = np.where(X > X_MAX)
            X[arr] = X_MAX - 1e-6
        elif any (X < X_MIN):
            arr = np.where(X < X_MIN)
            X[arr] = X_MIN + 1e-6

        if any (Y > Y_MAX):
            arr = np.where(Y > Y_MAX)
            Y[arr] = Y_MAX - 1e-6
        elif any (Y < Y_MIN):
            arr = np.where(Y < Y_MIN)
            Y[arr] = Y_MIN + 1e-6

        #update delay
        x_delay = np.roll(x_delay, -1)
        x_delay[:,-1] = X
        y_delay = np.roll(y_delay, -1)
```

```
        y_delay[:,-1] = Y



        if t % oversampling == 0:
            x[:,n] = X
            y[:,n] = Y
            n += 1

    # Normalize trajectory and targets

    inputs = DeepCalib.trajectory(
        names=['x', 'y'],
        values=np.swapaxes([x, y],0,1),
        scalings=['x [\u03BCm]', 'y [\u03BCm]'],
        scaled_values=np.swapaxes(scale_inputs(*[x,y]),0,1))

    targets = DeepCalib.targets(
        names=['k [fN/\u03BCm]', 'M [fN/\u03BCm]', 'delays'],
        values=np.swapaxes([k, M, delay_calc],0,1),
        scalings=['k/100', '(M+50)/100', 'delays/delay0'],
        scaled_values=np.swapaxes(scale_targets(*[k, M, delays]),0,1))

    return inputs, targets
```

[ ]:
```
trajectory, targets = simulate_trajectory(2)
```

1.3 3. CHECK TRAJECTORY SIMULATION

Checks the results of the function to simulate the trajectories by plotting some examples in rescaled units.

Have a look at the trajectories and check if they match your system, and keep an eye on different trajectories and make sure your scaled units vary in the order of 1, i.e, neither too small (0.01 or smaller) nor too large (100 or larger)

The parameter number_of_images_to_show determines the number of trajectories that are plotted.

[ ]:
```
### Show some examples of simulated trajectories

number_of_trajectories_to_show = 10
DeepCalib.plot_sample_trajectories(simulate_trajectory,␣
 ↪number_of_trajectories_to_show)
```

1.4 4. CREATE AND COMPILE DEEP LEARNING NETWORK

The parameters of the deep learning network are defined and the network created. The summary of the network is printed where the output shape and number of parameters for each layer can be

visualized.

Comments: 1. The parameter input_shape determines the shape of the input sequence, given by the number of time-steps times the number of samples in each input sequence. Make sure your input shape dimensions match the length of the input trajectory, in this example 2 x 1000 = 2000. 2. The parmameter conv_layers_dimensions determines the number and size of LSTM layers. 3. The parameter number_of_outputs determines the number of outputs, i.e. the number of force field parameters to be estimated.

[ ]:
```
### Define parameters of the deep learning network

input_shape = (None, 1000)
lstm_layers_dimensions = (1024, 256, 64)
number_of_outputs = 3

### Create deep learning network

network = DeepCalib.create_deep_learning_network(input_shape,␣
 ↪lstm_layers_dimensions, number_of_outputs)

### Print deep learning network summary

network.summary()
```

1.5 5. TRAIN DEEP LEARNING NETWORK

The parameters for the training of the deep learning network are defined and the network is trained. The sample size, iteration number, MSE, MAE and the time of each iteration is printed.

Comments: 1. The parameter sample_sizes determines the sizes of the batches of trajectories used in the training. 2. The parameter iteration_numbers determines the numbers of batches used in the training. 3. The parameter verbose determines the frequency of the update messages. It can be either a boolean value (True/False) or a number between 0 and 1.

[ ]:
```
### Define parameters of the training
model = network
sample_sizes = (32, 128, 512, 2048, 4096)
iteration_numbers = (1001, 1001, 2001, 2001, 2001)
verbose = .01

### Training
training_history = DeepCalib.train_deep_learning_network(model,␣
 ↪simulate_trajectory, sample_sizes, iteration_numbers, verbose)
```

1.6 6. PLOT LEARNING PERFORMANCE

The learning performance is plotted. The MSE, MAE, sample size, iteration number and iteration time are plotted against the number of timesteps.

Comment: 1. The parameter number_of_timesteps_for_average determines the length of the average. It must be a positive integer number.

[ ]:
```
### Plot learning performance

number_of_timesteps_for_average = 100

DeepCalib.plot_learning_performance(training_history,␣
↪number_of_timesteps_for_average)
```

1.7 7. TEST DEEP LEARNING NETWORK ON NEW SIMULATED TRAJECTORIES

The deep learning network is tested on new simulated trajectories (parameters are defined in section 2). The predicted values of the targets are plotted as function of their ground-truth values both in scaled and physical units.

Comments: 1. The parameter number_of_predictions_to_show determines the number of predictions that are shown.

[ ]:
```
### Test the predictions of the deep learning network on some generated␣
↪trajectories

number_of_predictions_to_show = 1000

%matplotlib inline
DeepCalib.plot_test_performance(simulate_trajectory, network, rescale_targets,␣
↪number_of_predictions_to_show)
```

1.8 8. SAVE DEEP LEARNING NETWORK

Comments: 1. The parameter save_file_name is the name of the file where the deep learnign network is saved. 2. By default, the network is saved in the same folder where DeepCalib is running.

[ ]:
```
save_file_name = 'delayed_3a_optimized.h5'
model.save(save_file_name)
```

2 TRY TRAIN LONGER

[ ]:
```
model.summary()
```

[ ]:
```
### Define parameters of the training

sample_sizes = (4096, 4096)
iteration_numbers = (1001, 1001)
verbose = .01

### Training
```

```
training_history = DeepCalib.train_deep_learning_network(model,␣
 ↪simulate_trajectory, sample_sizes, iteration_numbers, verbose)
```

[ ]:
```
### Plot learning performance

number_of_timesteps_for_average = 100

DeepCalib.plot_learning_performance(training_history,␣
 ↪number_of_timesteps_for_average)
```

[ ]:
```
### Test the predictions of the deep learning network on some generated␣
 ↪trajectories

number_of_predictions_to_show = 1000

%matplotlib inline
DeepCalib.plot_test_performance(simulate_trajectory, network, rescale_targets,␣
 ↪number_of_predictions_to_show)
```

[ ]:
```
save_file_name = 'delayed_3a_longer.h5'
network.save(save_file_name)
```

[ ]:
```

```

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY