Bar-Ilan University

Exploring Deep Learning Approaches for Analysis of Random Walks

David Peleg

Submitted in partial fulfillment of the requirements for the
Master's Degree in the Department of Physics,
Bar-Ilan University

Ramat Gan, Israel 2022

This work was carried out under the supervision of

Dr. Stas Burov

Department of Physics, Bar-Ilan University

Acknowledgment

I would like to thank my thesis advisor Dr. Stas Burov for the opportunity to work on such challenging and fascinating project, for his boundless patience and sage advice while exploring these uncharted regions. Additionally, I would like to thank my family for all the support I have been given during the trying times of the pandemic, I would not have the strength to do this work their love and support.

Contents

Exploring deep learning approaches for analysis of random walks

David Peleg

Abstract

In the past few years, there has been a significant increase in the incorporation of machine learning into new fields due to increase in computation power as well as abundance of data to analyze. In this thesis we will incorporate machine learning into the field of statistical physics by using several neural network architectures. Our goal is to identify the underlying stochastic process based on a single short trajectory. The neural networks developed in this work show a significant increase in performance over more traditional methods such as Time Averaged Mean Squared Displacement. Furthermore, we use the trained architecture to identify the stochastic processes in trajectories that consist of several randomly intertwined random behaviors.

Chapter 1

Introduction

In the past few years, Machine Learning (ML) has been incorporated into many fields and revolutionized them as a result. The reason for this is partly due to an increase in computing power, but mainly because of the advancements made in the theoretical aspect. At the core of ML stands the Neural Network (NN) – a decision network that is capable of learning from large datasets. Much like any other learning process, we need to define a function to train the NN, and the current approach is the Maximum Likelihood Estimation (MLE) [1]. The MLE is a method of estimating the parameters of a distribution by maximizing some likelihood function so that under the assumed statistical model the observed data is most probable. In other words, since every dataset consists of samples drawn from some unknown n-dimensional distribution, the main goal of a classic NN is to implicitly approximate the generating distribution of the data by tweaking the parameters of the guessed distribution until the MLE reaches a maximum. Since it is preferable to find minima points of concave functions, instead of maximizing a likelihood function, a loss function is defined to indicate the similarity between the guessed distribution and the dataset generating distribution. This loss function can be compared to an n-dimensional energetic potential and the optimization process can be compared to finding the minimum of this potential, as will soon be shown.
In most models, there is no explicit solution to the problem of finding the minima of the loss function. Therefore, different optimization methods are used to take small steps in the direction of the gradient of the loss function until convergence to the minima. The most intuitive optimization method to implement is the Gradient Descent (GD) [2]. In this process, the entire dataset is used to calculate the derivative of the loss function with respect to the parameters of the model. The GD optimization process is defined as:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t \nabla_{\vec{\theta}} L(\vec{\theta}; x, y) \tag{1.1}$$

Where $\vec{\theta}_t$ are the parameters of the model at some step, $L(\vec{\theta}; x, y)$ is the pre-defined loss function, α_t is the learning rate at time t, x is the data and y is the classification of that data. The most popular loss function today is the Mean Squared Error (MSE) [3] function: $L(\vec{\theta}; x, y) = \frac{1}{n}\sum_i (f(x_i; \theta) - y_i)^2$. Where n is the size of the dataset, $f(x_i; \theta)$ is the output of the network (model's hypothesis function) given a set of parameters (network weights) $\vec{\theta}$ and (x_i, y_i) are samples from the dataset (the loss is computed over the entire dataset and i is the index of the current sample taken). The ability of the NN to generalize and create a prediction function for any sufficiently large dataset is the reason we are seeing a significant increase in its usage in modern physics research.

We will now investigate a more physical standpoint of the optimization process. First, we will discuss a variation of the Gradient Descent (GD) optimization method. From

a computational standpoint, the GD process is extremely time-consuming, partly since datasets today are usually extremely large. Therefore, we tweak the optimization process by calculating the derivative for a small subset of our data with every step. This new process is called Stochastic Gradient Descent (SGD) [4] and it is currently the most used process. A mathematical representation of the SGD is:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t \frac{1}{m} \sum_{i=1}^{m} \nabla_{\vec{\theta}} L(\vec{\theta}_t; x_i, y_i) \tag{1.2}$$

Where m is the size of the mini-batch of samples taken from the dataset and (x_i, y_i) are samples from the current mini-batch (this time i is the index of the current sample taken from the mini-batch m). We note, however, that calculating the gradient over a subset of samples yields the true gradient of the loss with some degree of error. It is possible to model our mini-batch gradient as:

$$\nabla L_{MB(t)} = \nabla L + \eta(t) \tag{1.3}$$

where ∇L is the true gradient and $\eta(t)$ is an error term caused by calculating the gradient over a mini-batch rather than the entire dataset. A common assumption is that $\eta(t)$ is a Gaussian random variable with a mean of 0 and a variance of α^2.The zero-mean assumption suggests that over multiple iterations of the optimization process, the noise in the gradient caused by using subsets rather than the whole dataset averages out. In other words, taking the average over time of Eq. (1.3) yields: $\mathbb{E}[\nabla L_{MB(t)}] = \nabla L$. However, as described in [5], the assumption that the Stochastic Gradient Noise (SGN) is a zero-mean Gaussian may not always be true, meaning the noise will not average out in iterations over the dataset. Therefore, a lot of research is done in modeling this noise by exploiting other distributions and coming up with new optimization processes. New research in this field has even modeled the noise distribution $\eta(t)$ as a Levy distribution which required a new modified optimization process instead of a simple SGD [6].

Let us now briefly discuss the similarities between the SGD and other physical processes. In Statistical mechanics, the model used to predict the movement of a particle in a fluid is the Langevin equation (as stated in chapter 2.1):

$$m\frac{dv}{dt} = -\lambda v + \eta(t) + \nabla U(x) \tag{1.4}$$

Where v is the velocity of the particle, m denotes the particle's mass, $-\lambda v$ is the Stokes force acting as drag depending on the fluid's viscosity, $\eta(t)$ is time-dependent noise which stems from the random collision of our tracer particle with other molecules in the fluid, and $\nabla U(x)$ is a driving force acting on our tracing particle. Taking the overdamped version of this dynamics ($\frac{dv}{dt} = 0$), we get the following stochastic differential equation [7]:

$$\frac{dx}{dt} = \frac{1}{\lambda}(\eta(t) + \nabla U(x)) \tag{1.5}$$

Discretizing this equation according to the Euler Maruyama method [8] yields:

$$x_{t+\Delta t} = x_t + \frac{1}{\lambda}[\sqrt{\Delta t}\eta(t) + \Delta t \nabla U(x)] \tag{1.6}$$

Where we take $\Delta t = 1$. Now, comparing the energy of a system of particles in some configuration to the loss function of our model (Eq. (1.3)), we see a clear similarity

between the SGD optimization process (Eq. (1.2)) and the relaxation to equilibrium in Langevin dynamics. In the SGD, the inherent random noise element appeared due to using small subsets of the dataset to calculate the gradient of the loss (mini-batching). But assuming the noise from the SGD averages out when we iterate over the entire dataset, we would like to add Gaussian noise to the optimization process in order to maintain the similarity. The variance is chosen in a manner that will make the noise dominate the inherent SGD noise. So our new Learning Algorithm is [9]:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(\vec{\theta}; x_i, y_i) + \alpha_t N(0, \sqrt{\alpha_t}) \tag{1.7}$$

Now that we have shown the similarities between the learning process and the process of a particle in a complex potential undergoing underdamped dynamics with Gaussian noise, it is reasonable to assume that in the same way that only a fraction of all physical processes can be modeled by the Langevin dynamics, the SGD is the optimal method of teaching a NN for only a fraction of the existing datasets. For example, the Cuckoo search algorithm shows that combining a local search with a combination of Levy flight behavior provides a better optimization process for some problems [10]. This algorithm, as suggested by its name, was inspired by modeling the behavior of Cuckoo birds. So, we can expand our horizon and look at other processes in Statistical Mechanics with the assumption that different optimization processes would be more suitable to tackle different ML problems. This issue will be discussed in further detail in the appendix (C).

Another area where ML could be useful is with problems that are confined to short-time measurements Regularly and irregularly spaced short-time measurements naturally occur in many scientific and industrial domains [11] [12] [13]. In observational astronomy, measurements of properties such as the spectra of celestial objects are taken at times determined by seasonal and weather conditions, as well as the availability of observation time slots. In clinical trials, a patient's state is usually observed at a low frequency, with often irregular time intervals. Another example is the time series produced by Single-Particle Tracking (SPT). SPT is mostly used to quantify the dynamics of single particles inside a disordered medium, like finding new kinetic behaviors when observing the kinetics of single molecules in live cells [14]. In order to identify the stochastic process that is responsible for the behavior of our system, as well as other statistical features (diffusion constant, presence of drift, anomalous exponent, etc.), the most commonly used method is taking the Mean Squared Displacement (MSD) of each time step in the ensemble of paths we gathered experimentally. The MSD is written as follows:

$$MSD_t = \langle \Delta x_t^2 \rangle = \langle (x_t - x_0)^2 \rangle = \langle x_t^2 \rangle + x_0^2 - 2x_0 \langle x_t^2 \rangle \tag{1.8}$$

Under the assumption that all the paths in the ensemble have a fixed starting point $x_0 = 0$ we can simply write the MSD as:

$$MSD_t = \langle x_t^2 \rangle \tag{1.9}$$

In general, we differentiate between processes according to their MSD in the following manner:

$$MSD_t \sim t^\alpha = \begin{cases} \text{sub-diffusion} & 0 < \alpha < 1 \\ \text{regular diffusion} & \alpha = 1 \\ \text{super-diffusion} & \alpha > 1 \end{cases} \tag{1.10}$$

Where the anomalous constant α is related to fundamental characteristics of the stochastic process, such as temporal correlations (for the case of Fractional Brownian Motion) and the distribution of particle steps. The main issue with the MSD is its sensitivity to small ensembles acquired from SPT. In order to deal with this issue, we commonly turn to the Time Averaged Mean Squared Displacement (TA-MSD). The TA-MSD is a method that calculates the squared displacement of a single path repeatedly, with a different time interval between samples taken.

$$\chi_{\Delta,t} = \frac{1}{t-\Delta}\int_0^{t-\Delta}(x_{t_0+\Delta} - x_{t_0})^2 dt_0 \tag{1.11}$$

We then take the time average of the resulting random variable to reduce its fluctuations around the mean.

$$TA-MSD_\Delta = \langle \chi_{\Delta,t} \rangle \tag{1.12}$$

In most (Ergodic) cases, as the length of the of the sampled path goes to infinity, the $TA-MSD_\Delta \sim MSD_t$. This means, that in cases where we are able to acquire relatively long paths from SPT, the MSD can be swapped for TA-MSD in order to classify the diffusion process. However, the experimentally produced time series are often short and sometimes unevenly spaced, mostly due to limitations of instruments and with inherent experimental constraints of biological samples [15]. Furthermore, the TA-MSD will not converge to the MSD when the process is non-Ergodic [16]. But the main issue with using these methods is the fact that we cant distinguish between different stochastic processes that exhibit the same anomalous diffusion exponent α. A different approach to gaining insights from shorter paths is to create a modified version of the Recurrent Neural Network (RNN) [17]. This network is comprised of a chain of cells where each cell contains a small Fully Connected (FC) network and every input of the cell is concatenated to the output of the previous cell (also called a hidden state) giving the network the ability to take context into account. There are many properties of stochastic processes that can be extracted using the RNN, and in this thesis, we will focus on creating a deep learning architecture that will be able to classify stochastic models that are present in experimentally obtained trajectories, e.g. SPT experiments. Furthermore, we will use our architecture to create a network that specializes in the identification of multiple models taking part in the generation of the random walk.
In Chapter 2 we will discuss the methods used to create the random walk generators. These generators will create different datasets to train and test our network. In Chapter 4 we will show how different architectures perform when used as classifiers on the constructed datasets. In Chapter 5 we will show how to use the trained networks to classify walks that contain intertwined processes, meaning different temporal segments of the random walk are governed by different dynamics.

Chapter 2

Dataset Creation

As a first step towards creating a NN that will be able to identify a random walk's generating model, we have to create a dataset that will be used to train the network and verify its accuracy. Therefore, we will be simulating different random walks created by different models to ensure the final network will be robust enough to identify a wide range of stochastic processes. Our first requirement to ensure robustness is to assign each random walk in the dataset a different diffusion constant that is sampled from a uniform distribution $U(0,1)$ (the rest of the measures taken to ensure robustness will be shown in the sections of this chapter as well as in the next chapter).

2.1 Brownian Motion

Brownian Motion (BM) is a phenomenon discovered by Robert Brown in the 19th century when he noticed the random motion of pollen grains in water. However, it was not until 1905 that Einstein formulated a comprehensive theory describing this random motion [18]. Since its discovery, the BM model has been used to analyze problems in a wide range of fields, from stock price predictions [19] and other phenomena in economic systems [20], to thermal noise in electrical circuits [21]. The BM describes the motion of a colloidal particle immersed in a fluid and the irregularity of the particle's motion is caused by random collisions between the particle and its surrounding liquid particles. In order to mathematically describe this motion, we use Newton's second law and in addition to a default driving force, we also add drag and a random force. The equation we got is called the Langevin equation [22]:

$$m\frac{dv}{dt} = F + \eta(t) - \nabla U(x) \tag{2.1}$$

Where η denotes some time-dependent noise (also referred to as a stochastic force), U is the potential acting on the particle, m is the particle's mass and F is the Stokes drag force, which is proportional to the velocity of the particle (note that we will only refer to the one-dimensional case):

$$F = -\lambda v \tag{2.2}$$

The friction coefficient λ is given as:

$$\lambda = 6\pi\mu a v \tag{2.3}$$

Where μ is the dynamic viscosity of the fluid, a is the particle radius and u is the velocity of the fluid relative to the particle.

we will take the over-damped limit of Langevin ($\frac{dv}{dt} = 0$) with no driving force ($\nabla U(x) = 0$):

$$0 = -\lambda \frac{dx}{dt} + \eta(t) \Rightarrow \qquad \frac{dx}{dt} = \frac{1}{\lambda}\eta(t) \tag{2.4}$$

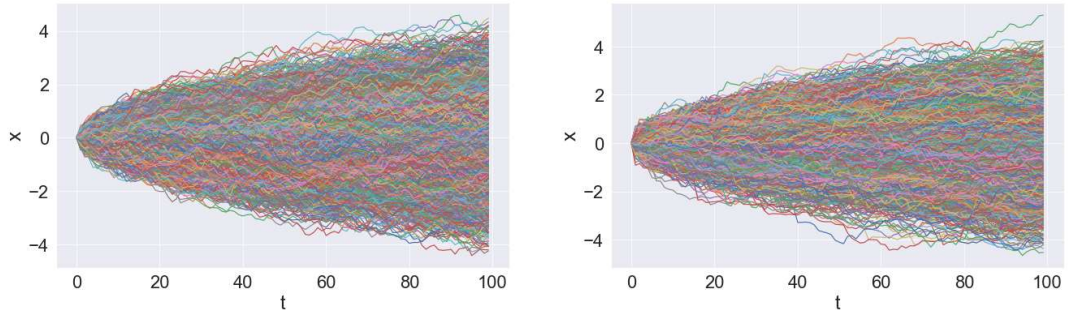
It is common to use the discretized form of (2.4) [8]:

$$x_{t+\Delta t} = x_t + \sqrt{D\Delta t}\,\eta(t) \tag{2.5}$$

Where Δt is the interval between samples and $D = \frac{1}{\lambda}$. We will use Eqn. (2.5) with two types of stochastic force: Gaussian probability distribution ($N(0,1)$) and a Laplace distribution [23] with mean 0 and variance 1 shown here as a Probability Density Function (PDF):

$$PDF(\eta) = \frac{1}{2\sqrt{2}} exp\Big(-\frac{|\eta|}{\sqrt{2}}\Big) \tag{2.6}$$

We used these schemes to generate random walks (Fig. 2.1) that will be used as part of the training dataset for our neural networks.



(a) (b)

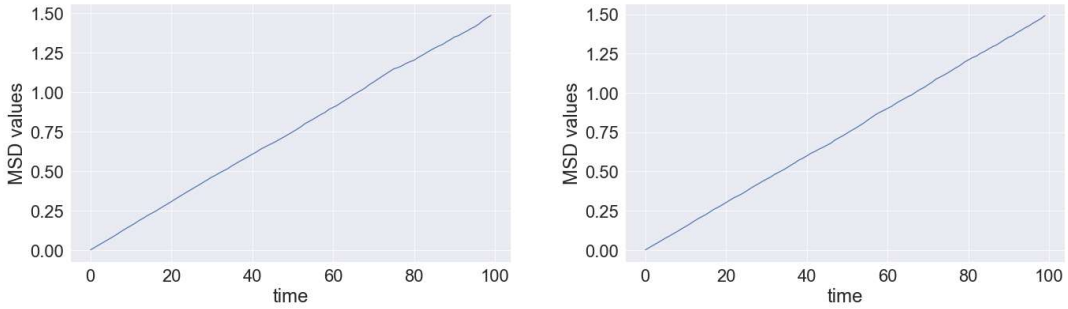
Figure 2.1: A dataset with $1 \cdot 10^4$ Brownian motion random walks with the parameters: $D = 0.3, \Delta t = 0.05, t = 5$ (100 steps). Figure 2.1a has a Gaussian distributed jump lengths $N(0,1)$, while Figure 2.1b has Laplace distributed jump lengths with a mean of 0 and variance 1

A good indication that our dataset contains normally diffusing particles is the fact that our Mean Squared Displacement (MSD) in this case is a linear function of time $\langle x_t^2 \rangle \sim t$ as shown in (Fig. 2.2).

2.2 Continuous Time Random Walk

The Continuous Time Random Walk (CTRW) [24] is the first generalization of the regular diffusion mentioned above. Unlike the regular random walk, where the time interval between every consecutive step is identical, particles following this stochastic process have their waiting times (intervals between consecutive steps) sampled from some distribution. In other words, the Probability Density Function (PDF) of finding a particle at location x in time t is given by:

$$P(x,t) = \sum_{n=0}^{\infty} P(n|t) P_n(x) \tag{2.7}$$



(a) (b)

Figure 2.2: the MSD values as function of time for the Brownian motion random walks. Figure 2.2a has a Gaussian noise $N(0,1)$, while Figure 2.2b has a Laplace distributed noise with mean= 0 and variance= 1

Where $P(n|t)$ is the probability of taking n steps in time t and $P_n(x)$ is the probability of reaching point x in n steps. Let us now define the underlying PDF's of this stochastic process. $f(x)$ is the distribution of jump lengths, and $\psi(t)$ is the distribution of our waiting times. Using the properties of the characteristic function [25], we obtain the following representation of $P_n(x)$ and $P(n|t)$ [26]

$$\hat{P}(n|s) = \frac{1-\hat{\psi}(s)}{s}\hat{\psi}^n(s) \tag{2.8}$$

$$\tilde{P}_n(k) = \tilde{f}^n(k) \tag{2.9}$$

where $\hat{\psi}$ is the Laplace transform of ψ, and $\tilde{f}$ is the Fourier transform of f.

$$\hat{\psi}(s) = \int_0^\infty e^{st}\psi(t)dt \qquad \tilde{f}(k) = \int_{-\infty}^{\infty} e^{ikx}f(x)dx \tag{2.10}$$

Plugging equations (2.8) and (2.9) into (2.7) and performing the summation yields the the Montroll-Weiss equation [24]:

$$\hat{\tilde{P}}(k,s) = \frac{1-\hat{\psi}(s)}{s}\frac{1}{1-\hat{\psi}(s)\tilde{f}(k)} \tag{2.11}$$

Since the waiting times τ_i and the displacements Δx_i are mutually independent identically distributed random variables, we can use the same generating process we used for regular diffusion with a slight adjustment to allow the possibility of samples with no movement ($\Delta x_t = 0$). We created all three CTRW's with a Gaussian distribution for jump lengths $f(x)$, the difference lays in the waiting time distributions $\psi(\tau)$. The first walk is made with a uniform waiting time distribution:

$$\tau \sim \psi(\tau) = U[2\Delta t, 10\Delta t] \tag{2.12}$$

where Δt is the interval between samples taken. This version of CTRW exhibits the same MSD as regular diffusion (Fig. 2.4a).
The second walk is made with an exponential waiting time distribution

$$\psi(\tau) = 100\Delta t e^{-100\Delta t\tau} \tag{2.13}$$

and also exhibits the same MSD as regular diffusion (Fig. 2.4b) (The constants of this function were chosen in a manner that keeps the scaling similar between all CTRW datasets).
The third walk is made with a Power-Law waiting times distribution:

$$\psi(\tau) = A_\alpha \tau^{-1-\alpha} \tag{2.14}$$

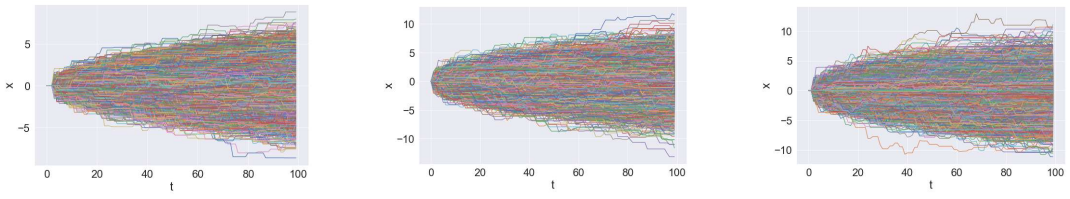
Where A_α is a normalization constant that depends on α and the starting point of the distribution τ_0. We used the following transformation from a uniform distribution to create the Power-Law (as described in the appendix A.1):

$$\tau = [-\frac{\alpha}{\Delta t}(u-1)]^{-\frac{1}{\alpha}} \tag{2.15}$$

Where τ is our desired power-law distributed random variable $\psi(\tau) = A_\alpha \tau^{-1-\alpha}$, u is a uniformly distributed sampled number $U(0,1)$ and Δt is a scaling constant which in our case is the interval between samples. The power-law waiting time distribution is common in many physical phenomena like transport in doped semiconductors or the motion of colloidal tracer particle in an entangled F-actin filament network [27].
According to [28], the MSD of our CTRW is given as

$$\langle x_t^2 \rangle \sim \begin{cases} t & 1 < \alpha \\ \\ t^\alpha & 0 < \alpha < 1 \end{cases} \tag{2.16}$$

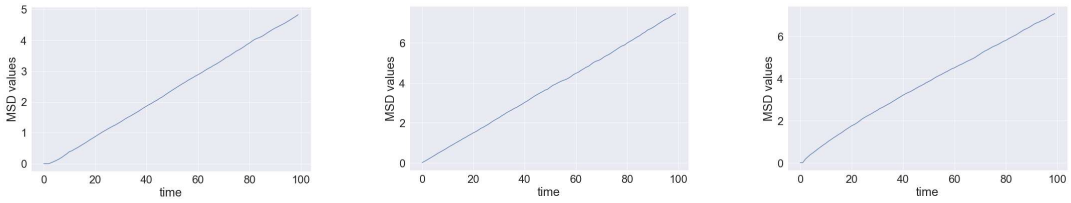
We indeed see that for the case of $\alpha \to 1^-$ the MSD starts to converge to normal diffusion (Fig. 2.4c). This means that we cannot use the MSD as a reliable tool for distinguishing between Brownian motion and CTRW with $\alpha > 1$. Furthermore, in cases where the



(a) (b) (c)

Figure 2.3: A dataset with $1 \cdot 10^4$ Continuous Time Random Walks with the parameters: $D = 0.3, \Delta t = 0.05, t = 5 (steps = 100)$, all created with Gaussian distribution of jump lengths $N(0,1)$. Figure 2.3a shows Uniformly distributed waiting times. Figure 2.3b shows Exponentially distributed waiting times. Figure 2.3c show Power-Law distributed waiting times.

CTRW exhibits subdiffusive properties (Fig. 2.5) we will not be able to use the MSD to distinguish between CTRW and other subdiffusive processes (as shown in chapter 2.3). The TA-MSD for this non-ergodic case differs from the MSD and is linear, making it useless for distinguishing between regular and subdiffusive behavior.



(a) (b) (c)

Figure 2.4: the MSD values as a function of time for the Continuous Time Random Walks. Figure 2.4a shows uniformly distributed waiting times. Figure 2.4b shows exponentially distributed waiting times. Figure 2.4c shows Power-Law distributed waiting times.

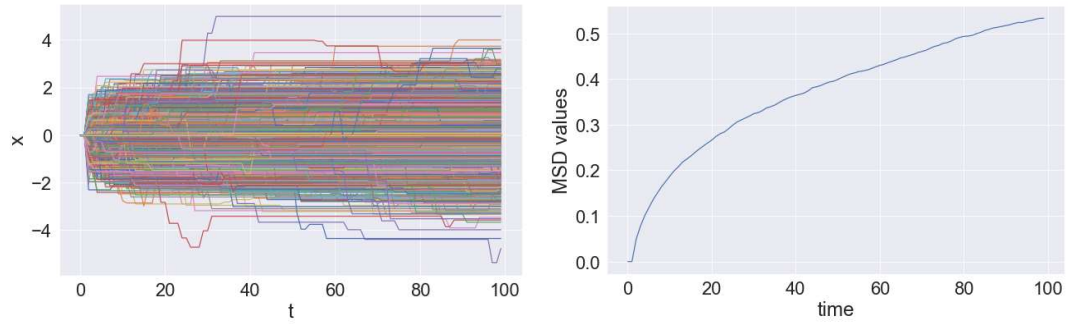


Figure 2.5: Dataset and MSD values for CTRW with powerlaw waiting time distribution with $\alpha = 0.25$

We also constructed the CTRW by using the Laplace distribution (Eqn. (2.6)) for the jump lengths generator, in order to increase the robustness of the training.

2.3 Fractional Brownian Motion

The Fractional Brownian Motion (FBM) [29] first introduced by Mandelbrot and van Ness, is a generalization of Brownian motion that includes long-range, strong spatial and temporal correlations. The position of a particle at time t is defined in terms of the Langevin equation:

$$\frac{dX(t)}{dt} = \zeta(t) \tag{2.17}$$

$$X(t) = \int_0^t \zeta(t')dt' \tag{2.18}$$

Where $\zeta(t)$ is a stationary Fractional Gaussian Noise (FGN) with $\langle\zeta(t)\rangle = 0$ and long-ranged autocovariance given as [30]:

$$\gamma(k) = \mathbb{E}[\zeta^H_{(0)}\zeta^H_{(k)}] = \frac{\sigma}{2}[(k+1)^{2H} + |k-1|^{2H} - 2k^{2H}] \qquad ,k \geq 0 \tag{2.19}$$

Where σ is the variance of the FGN and H is the Hurst exponent, determined by the long-ranged correlations in the model. This yields a simple expression of the autocorrelation function:

$$\rho^H_{(k)} = \frac{1}{2}[(k+1)^{2H} + |k-1|^{2H} - 2k^{2H}] \qquad ,k \geq 0 \tag{2.20}$$

We now note the Hurst exponent is the one responsible for the diffusion type of the random walk. An extension of the FGN is the FBM, with expected correlation between two times t and s given by Beran [30] as:

$$\rho_{(s,t)} = \frac{1}{2}(t^{2H} + s^{2H} - (t-s)^{2H}) \tag{2.21}$$

A random walk B^H can be defined as an FBM by three major properties [31]:

1. Stationary increments [32] - meaning the random variables $Y_t = B^H_{t+s} - B^H_s, \forall t \geq 0, t > s$, are identically distributed. Furthermore, B^H and Y have the same distribution. In other words, for the incremental behavior of B^H is the same for any point in time $B^H_{t+s} - B^H_s \sim B^H_t$.

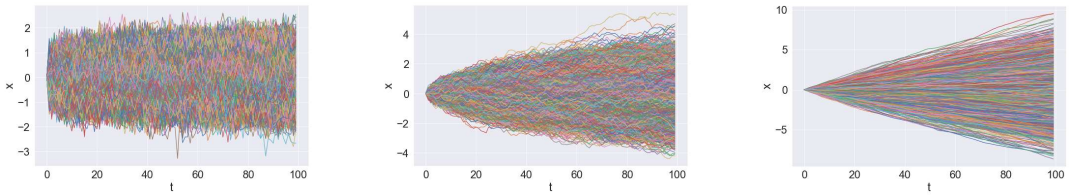
2. Self-similarity [33] - meaning the random walk has fractal behavior $B^H_{\alpha t} \sim |\alpha|^H B^H_t$

3. Long range dependence - Assuming the following times $s_1 < t_1 < s_2 < t_2$ such that the intervals $[s_1, t_1]$ and $[s_2, t_2]$ do not intersect. we get that the following dependence

$$\mathbb{E}\left[(B^H_{t_1} - B^H_{s_1})(B^H_{t_2} - B^H_{s_2})\right] < 0 \qquad ,0 < H < 0.5 \tag{2.22}$$

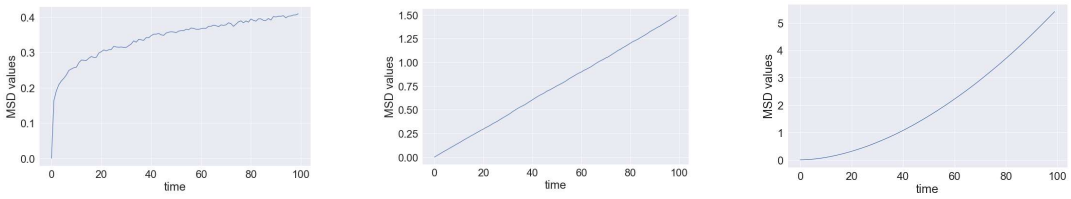
$$\mathbb{E}\left[(B^H_{t_1} - B^H_{s_1})(B^H_{t_2} - B^H_{s_2})\right] > 0 \qquad ,0.5 < H < 1 \tag{2.23}$$

To create our FBM dataset, we generated N samples of FGN using the Cholesky method [34] (as described in the appendix A.2). We then summed the samples according to (2.18) and the results can be seen in Fig. 2.6. As mentioned in chapter 2.2, we cannot distinguish between the CTRW with a heavy tailed waiting time distribution and the FBM with $H < 0.5$ (Fig. 2.7a) using MSD due to the sub diffusive nature of both processes. The MSD for the FBM can be easily calculated using equation (2.21) for the same time step: $MSD_t = \langle x_t^2 \rangle = t^{2H}$.
We note from Eqn. (2.21) that for the case of $H = 0.5$, the random walks converge to a regular Brownian motion (Fig. 2.7b). These cases mean the dataset will contain random walks that are by definition identical to Brownian motion yet labeled as FBM, which guarantees any network's failure to correctly classify. Therefore, we will remove FBM trajectories with a Hurst exponent of $H = 0.5$ from the dataset. For the case of $H > 0.5$ (Fig. 2.7c), we see a superdiffusive behavior that will be indistinguishable from the Levy walk when using the MSD.



(a) (b) (c)

Figure 2.6: A dataset with $1 \cdot 10^4$ Fractional Brownian Random Walks with the parameters: $D = 0.3, \Delta t = 0.05, t = 5$ (steps=100) for different Hurst exponents H. Figure 2.6a shows $H = 0.1$, which yields negatively correlated random walks. Figure 2.6b shows $H = 0.5$, which yields uncorrelated random walks (Brownian Motion). Figure 2.6c shows $H = 0.99$, which yields positively correlated random walks



(a) (b) (c)

Figure 2.7: the MSD values as a function of time for Fractional Brownian Random Walks with different Hurst exponent H. Figure 2.7a shows $H = 0.1$, which yields a sub-diffusive process. Figure 2.7b shows $H = 0.5$, which yields a regularly diffusive process. Figure 2.7c shows $H = 0.99$, which yields a super-diffusive process.

2.4 Levy Flights

The Levy flight refers to a random walk in which the jump lengths are independent identically distributed random variables that are sampled from a Levy distribution, which is any fat-tailed distribution (meaning it has a non-finite second moment).

In our case, Levy flights can be simulated with the same equation used for regular diffusion (Eqn. (2.5)). The only exception is that the jump length added to every time step is sampled from a heavy-tailed Levy distribution. In our case, the distribution chosen was a Power-Law distribution $x \sim x^{-1-\alpha}$. Where $0 < \alpha < 1$ would be variable at the step of training to increase robustness. For the case of Levy flights, the MSD diverges as there is a probability of observing an infinitely large step at any time step t [28]. Therefore, once again we can't rely on the MSD alone to classify random walks with Levy distributed jump lengths. A more complex and accurate approach will be presented in Chapter 4.

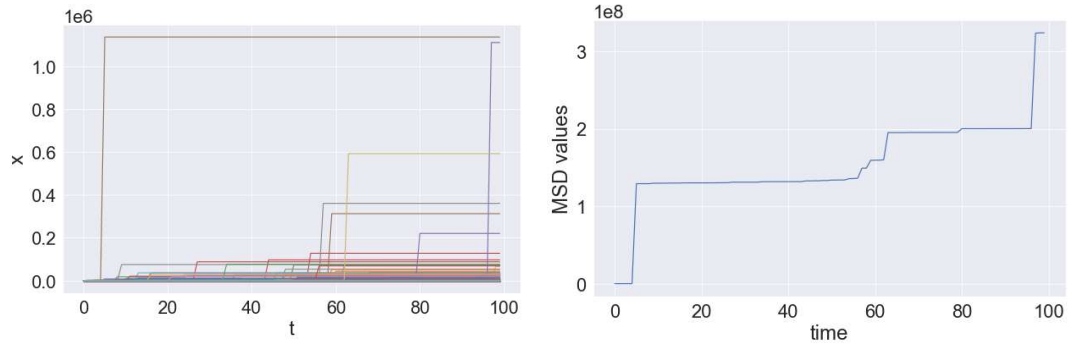


Figure 2.8: Dataset and MSD values for Levy flight random walk with $\alpha = 0.7$

Chapter 3

Neural Network Architectures

This chapter describes the Machine Learning (ML) algorithms used to classify the random walks from Chapter 2, as well as an overview of the general terms commonly used in the field of machine learning.

At the core of the modern Deep Neural Networks (DNN) stands the Perceptron [35]. The Perceptron is a simplified model of the biological neurons in our brain and is used as a building block for most DNNs. As mentioned in the introduction, any ML algorithm receives a set of values $\vec{X}$, and multiplies them by a set of weights $\vec{\theta}$. In most modern use cases, the basic Perceptron is transformed into the well-known neuron by applying a function to the product of the scalar multiplication $\sigma(\vec{X} \cdot \vec{\theta})$ called the activation function. The process of training refers to the tweaking of the weights $\vec{\theta}$ to produce the most desired result over some dataset. For regression problems, where an actual numerical value needs to be predicted by the model, this σ function's output can be taken directly. For classification problems, where the solution to the problem is binary, some threshold is defined to classify the sample $\vec{X}$ as positive or negative. An example of such a scenario is the question of whether a given random walk is BiModal or not, meaning it contains a mixture of two processes. Such problems will be tackled in Chapter 5.

Many interesting problems require multi-class classification, meaning the solution is more complex than a simple binary value (like classification of random walks). A simple solution to such problems is using multiple binary neurons with different weights:

$$y_i = \sigma(\vec{X} \cdot \vec{\theta}_i) \qquad i = 1, ..., N \tag{3.1}$$

Where σ is the activation function of the neurons, $\vec{\theta}_i$ are the parameters of each neuron i, N is the number of possible classifications, $\vec{X}$ is the input of the neurons, and y_i is the output value of each neuron i. The classification of the sample $\vec{X}$ is decided by the neuron with the maximal output value $\max(y_i)$. This concept is called a classification layer.

A common approach to the implementation of the classification layer is the use of a Softmax function on the outputs of the neurons, which yields the probability that a sample $\vec{X}$ belongs to each of the N classes. In this thesis, a variation of the Softmax function called LogSoftmax function is used. LogSoftmax is defined as:

$$\tilde{y}_i = LogSoftmax(y_i) = \log \frac{e^{y_i}}{\sum_j e^{y_j}} \tag{3.2}$$

Where y_i and y_j are the the output value of neurons i and j, and j sums over all classes N. We treat the outputs $\tilde{y}_i$ of the LogSoftmax as the likelihood of each class and therefore return the highest probability value as the ML algorithm's classification of the example.

In other words, a sample $\vec{X}$ will be classified by the ML algorithm as class i if its corresponding $\tilde{y}_i$ is maximal.
Other terms that we refer to in this thesis are:

1. Epoch - As previously established, the ML algorithm takes samples from the dataset and tweaks its parameters in order to try and match the actual value y. An iteration over the entire dataset is called an Epoch.

2. Loss function (L) - This term was extensively discussed in the Introduction (Chapter 1). The loss function is usually defined as a function of the actual value of the sample y (which in our case is the type of the random walk), and the value predicted by the ML algorithm $\tilde{y}$. The value of $\tilde{y}$ changes as the ML algorithm learns and tweaks its parameters to predict a value close to y. This term can also be called a potential due to reasons discussed in Chapter 1.

3. Backpropagation method - Also called optimization method (or optimization algorithm). This term refers to the action of taking the derivative of the Loss function with regard to the parameters of the ML algorithm $\nabla_\theta L$, and tweaking the parameters θ accordingly. A simple example is the SGD algorithm discussed in Chapter 1. The optimization method chosen in this work is the Adam algorithm, which will be briefly explained in Chapter 4.

Note that as only variations of Neural Networks are presented in this thesis, from here on ML algorithms will simply be referred to as Deep Neural Networks (DNN).
The rest of this chapter will offer a deeper dive into the different DNN architectures constructed in this work.

3.1 Fully Connected Neural Network

The first architecture tried is the Fully Connected (FC) network. This network is a simple generalization of the Perceptron [35] called Multi-Layered Perceptron (MLP) [36]. The MLP is constructed by generating an ordered set of layers, each containing multiple perceptrons. Each perceptron receives its inputs from all the perceptrons of the previous layer and transmits its outputs to all the perceptrons of the next layer. Note that as each perceptron in this work has an activation function, this network is called a Fully Connected neural network. The neurons in all layers except the classification layer will have a Rectified Linear Unit (ReLU) as activation function [37]. This function simply outputs 0 when $x < 0$ and a linear function with a slope of 1 for $x \geq 0$ ($f(x) = max(0,x)$). The FC architecture constructed in this thesis consists of four layers (Fig. 3.1):

1. Input layer with the same size as the length of the random walks.

2. First transfer layer with 30 neurons and a ReLU activation function.

3. Second transfer layer, also with 30 neurons and a ReLU activation function.

4. Classification layer with 4 neurons and a LogSoftmax function.

Although very powerful, the most notable drawback of the FC network is its inability to use the structure of the input data analyzed. In other words, since every neuron in the first transfer layer takes all the input layer values, the ordering of the input values is irrelevant.

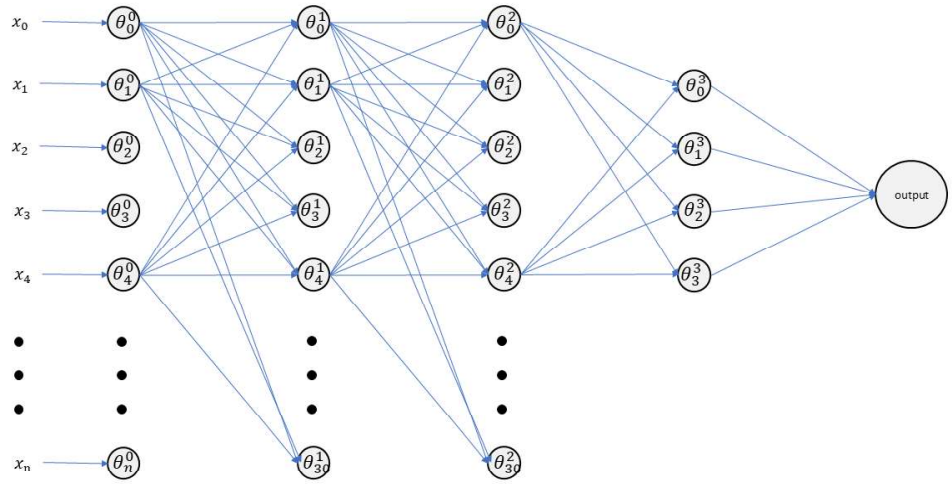


Figure 3.1: A diagram of the Fully Connected Neural Network architecture used in this work.

This means the FC network classifies based on raw data only. In the domain of random walk classification, it means that the FC architecture will not be able to infer temporal relations such as auto-correlation, as the time of every step of the random walk is not explicitly taken into account. Furthermore, the FC network requires all random walks in the dataset to be in the same length, which is a significant limitation in real-world use as particle tracking can often produce inconsistent trajectory lengths. While this issue could be solved by padding short trajectories and pruning long ones, these actions will often result in an accuracy dip. This dip could be avoided by using an ML algorithm that explicitly supports variable trajectory lengths.

3.2 Convolutional Neural Network

The Convolutional Neural Network (CNN) [38], was introduced in order to overcome the FC's inability to implicitly infer relations from the ordering of the input data (Section 3.1). The CNN is used in many fields such as classification of images ([39]; [40]), object detection [41], face detection [42], speech recognition [43], facial expression recognition [44], vehicle recognition [45] and many more.

A typical CNN operates in the following manner. An image X is given to the CNN as a matrix of numbers (images are used for simplicity, but the CNN can be easily adjusted to handle $1D$ inputs such as random walks). The CNN contains a small matrix of weights θ (meaning $|X| > |\theta|$) called a filter (or kernel). The CNN performs a convolution between its weight matrix θ and the input image X. The output matrix A is fed into an activation function σ as viewed in (3.3).

$$A_{ij} = \sigma((\theta * X)_{ij}) \tag{3.3}$$

Where A_{ij} is a cell in the output matrix, θ is the matrix of weights (kernels), X is the input, and σ is the activation function. The resulting matrix A can then be passed through more convolution layers, or get flattened and fed into an FC network. During the network

optimization process, the convolution kernels are also tweaked according to the gradient of the loss function in order to better capture the correlations. In order to increase the robustness of the CNN, a common practice is to include several different convolution kernels in each CNN layer.

The main advantage of the CNN over classic approaches to deep learning lies in its ability to capture context. In traditional FC's, when a two-dimensional matrix is given to the network as input, it simply gets reshaped to a vector. Since neurons in the input layer are independent of each other to begin with, the rearrangement of image cells done by the FC makes the recognition of patterns and correlations that much more difficult. In a CNN, this difficulty is solved due to the fact that every cell A_{ij} in the output matrix is created using only a local subset of the input X. Therefore, a convolution layer becomes helpful in capturing short time correlations between different cells in the input tensor so that the FC layer at the end will be able to make an accurate classification/prediction.

Another approach to understanding the convolution layer is to look at it as a transformation of the input that encapsulates the correlations of its input tensor. Once we have an output that behaves as a function that highlights the correlations of the input, we can train an FC to produce a function mapping between the correlation function and the desired classifications presented in our dataset. This view will be used again in the next section (section 3.3).

In this work, the architecture of the CNN is created in a manner that will attempt to imitate the TA-MSD. This means that the input layer will contain 3 types of convolution kernels, that given an input trajectory $\vec{X}$, will produce 3 output vectors $\vec{A}^i$. The first kernel type is a 2 by 1 vector that will capture nearest-neighbor correlations. The second kernel type is also a 2 by 1 vector but performs a 2-dilation convolution. This means it will capture the next-nearest-neighbor correlations. The third kernel type is also a 2 by 1 vector but performs a 3-dilation convolution. This means it will capture the next-next-nearest-neighbor correlations. In order to increase the robustness of the CNN, there are 16 convolution kernels for each kernel type. After the convolution, the results of the 3 kernels are concatenated into a single vector and fed into an FC network that learns a relation between the correlation functions and the random walk models. A summary of the architecture (Fig. 3.2) is:

1. CNN layer with 48 convolution kernels (16 for each type) and a ReLU activation function.

2. FC layer as a first transfer layer with 30 neurons and a ReLU activation function. The size of the first transfer layer depends on the output of the kernel layers, which is calculated as $(3 \cdot |\vec{X}| - 6) \cdot 16$ (where $|\vec{X}|$ is the number of measurements in a given trajectory).

3. FC layer as a second transfer layer, with 30 neurons and a ReLU activation function

4. Classification layer with 4 neurons using a LogSoftmax activation function

Even before diving into the results in Chapter 4, two possible drawbacks of using this architecture come to mind. The first is the requirement that all input trajectories must be of equal length that is known a-priory. As mentioned in Sec. 3.1, this demand greatly hinders the acquirement of experimental results, meaning real world use cases are limited. The second drawback is the fact that the size of the convolution kernels only enables short-time correlations depending on the number of kernel types contained in the CNN.

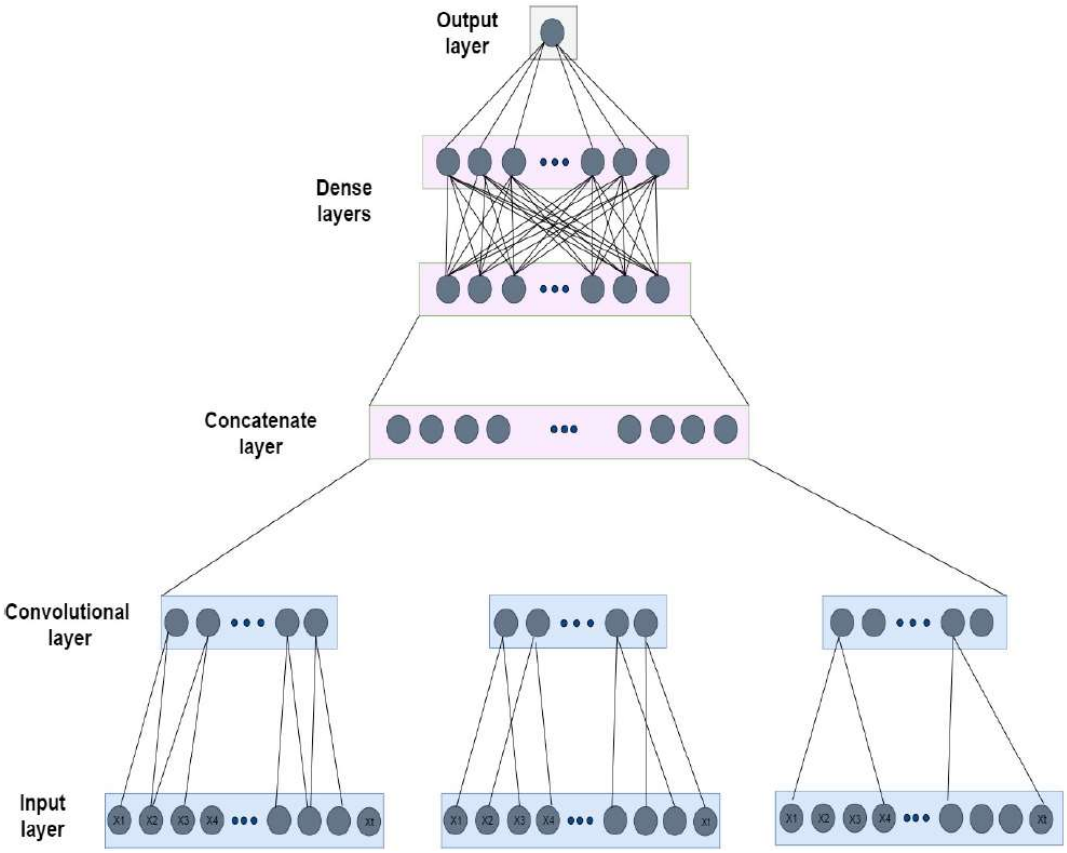


Figure 3.2: A diagram of the Convolutional Neural Network taken from [46]

In other words, in order to increase the CNN's resemblance to the TA-MSD, kernel types with larger dilation must be added to the network architecture. Therefore, ideally, we should have as many kernel types as the number of steps in the input trajectory. However, training the number of kernels required for such an ideal scenario will be computationally expensive.

The GRU-based DNN architecture offered in Section 3.3 will provide both the flexibility of dynamically sized trajectory classification, as well as fixed DNN size that is independent of the length of the trajectory.

3.3 Gated Recurrent Unit

The third and final architecture tried for this part is the Gated Recurrent Unit (GRU) [47], which is a variant of the Recurrent Neural Network (RNN) mentioned in the introduction. The concept of RNN is very commonly used to solve problems concerning time dependent data. The most popular use case is in the field of Natural Language Processing, with NN's responsible for text generation [48], machine translation [49], speech recognition [50], text summary [51] and many more. Another field that relies heavily on RNN is Time Series analysis, with solutions to problems such as: anomaly detection [52] [53], forecasting [54] [55] [56] [57], and classification [58] [59] [60].
In order to better understand the logic behind the GRU model, a short introduction of its two predecessors (the RNN and LSTM) is required.

3.3.1 brief RNN overview

The advantage of the Recurrent Neural Network (RNN) over the traditional FC lies in the fact that each input fed into the network contains additional information from previous inputs, termed hidden states. This feature essentially gives the network a sense of "memory", or as referred to in the CNN chapter, context. The mathematical representation of the hidden state output of the RNN cell (Fig. 3.3) is therefore:

$$h_t = tanh(\theta_{hh} h_{t-1} + \theta_{xh} x_t + b_h) \tag{3.4}$$

Where b_h is a bias added, θ_{xh} and θ_{hh} are two sets of network weights used to calculate the new hidden state h_t according to the input x_t and the previous hidden state h_{t-1} respectively (in the actual cell, both sets of weights are held as a single matrix). Another common practice is to let each cell generate a vector of hidden values which are then fed into another FCN for classification $\hat{y}_t$:

$$\hat{y}_t = \sigma(\theta_{yh} h_t + b_y) \tag{3.5}$$

Where σ is an activation function, θ_{yh} is a set of network weights, b_y is a bias added, h_t is a tensor of hidden states, and $\hat{y}_t$ is the classification of the input. When compared to the CNN, this added layer can be viewed as the function linking between the correlation function generated by the RNN and the classification of the temporal data. The size of the hidden state vector $|h_t|$ is a hyperparameter which in this case was 40. This use case of the RNN is referred to in the literature as a many-to-one RNN.

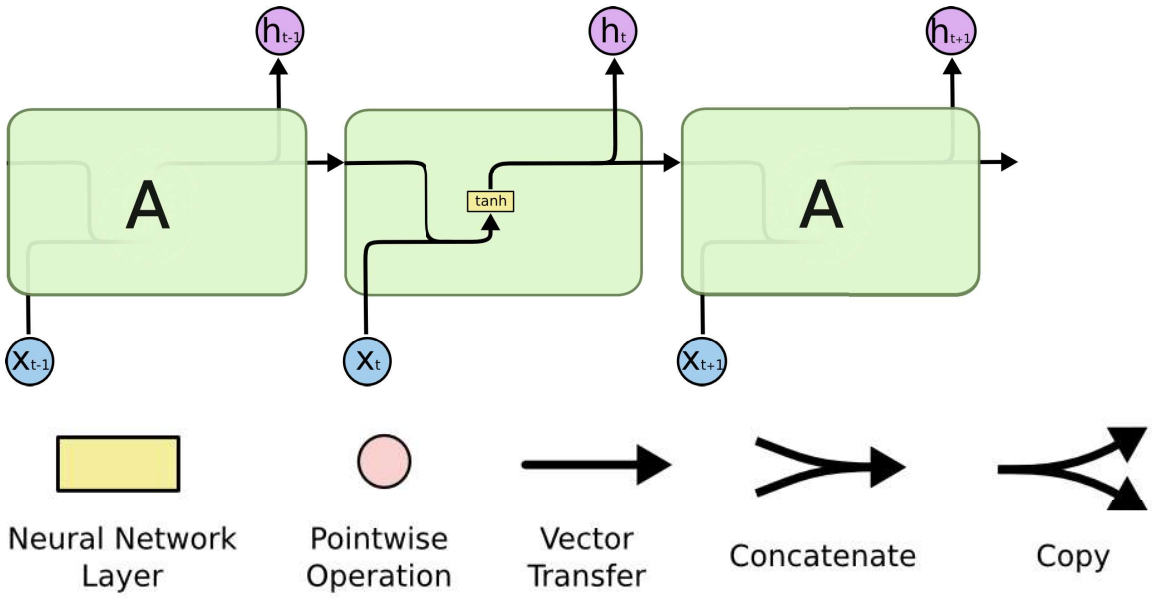


Figure 3.3: The basic cell architecture of the vanilla RNN (without bias), and the operator representations. *tanh* is a fully connected layer with a hyperbolic tangent function [61].

Note that in order to update the weights, the loss function must now take all the previous inputs in the time series into account:

$$L(\hat{y}, y) = \frac{1}{T} \sum_{t=1}^{T} L_t(\hat{y}_t, y_t) \tag{3.6}$$

The derivative of Eqn. (3.6) with respect to the weights $\theta_{hh}, \theta_{xh}, \theta_{yh}$ is called BackPropagation Through Time (BPTT) [62]. A common approach is to view this process as unrolling the network and treating every time step as an individual layer.
Although very powerful, the RNN suffers from three shortcomings.

The first, and the arguably less major issue is the difficulty of learning long-term dependencies, which will be demonstrated with the following example. Consider a dataset consisting only of CTRWs and the network's task is to predict whether the walk will make a step in the next time sample. For this case, in order to find the probability of making a step, the RNN must take into account all the previous waiting times. Or in other words, the RNN must take long-term dependencies into account. In theory, the RNN should be able to handle these types of problems. But in reality, this process involves too much hyperparameter tuning to be practical. The difficulty of RNNs to carry information over long periods of time is further explored in [63].

The second issue is commonly referred to in the literature as the vanishing/exploding gradients problem. As mentioned earlier, during the backpropagation process, the network is unrolled and each time step is treated as an individual layer. This unrolling action essentially yields a network with the same depth as the length of the time series. Therefore, the optimization process encounters the same obstacles that occur in extremely deep NNs. When optimizing deep NNs, multiplicative gradients can be exponentially decreasing/increasing with respect to the number of layers. In other words, any gradients that are slightly below/above 1 will either halt the learning process with 0 size update rules or overflow the network weights. The most common fix to the exploding gradient issue is gradient clipping - which means to simply cap the maximum value of the gradient.

Before introducing the solution to both these issues, there is a need to address the third point. In the vanilla RNN, the significance of every input step to the entire time series is equal. In cases such as Levy flights, the trajectory of the random walk looks like a simple Brownian motion (even though Levy flights are by nature fractals, the finite resolution of the measurements might hide this feature), which means a leap should have a much larger impact compared to the rest of the trajectory when classifying the random walk. Therefore, rather than simply concatenating the hidden state to the input, a more delicate system is required to take the hidden state into account.

3.3.2 brief LSTM overview

The Long Short Term Memory (LSTM) architecture [64] solves all of the RNNs previously discussed issues. It does so, by introducing a more complex unit cell that maintains a cell state C_t in addition to the traditional hidden state h_t. The cell state is transferred between time steps along with the hidden state as shown in (Fig. 3.4). The LSTM cell architecture contains four different Neural Networks (excluding the final NN added to infer the classification given the hidden state values) that can be divided into three gates that together, maintain and update the cell state C_t:

1. **forget gate** - a gate that takes into account the previous hidden state h_{t-1}, and the current input x_t, and outputs a new cell state C'_{t-1} that represents the importance of each vector slot to the new cell state C_t. As the name suggests, this gate decides which values of the cell state to keep and which ones to throw away. It does so by feeding $x_t \frown h_{t-1}$ (concatenated vectors) to a sigmoid activated NN that will output a vector f_t of values between 0 and 1. The gate then calculates the Hadamard product of the output and cell state vectors, which essentially suppresses the values of C_{t-1}

that should be forgotten. In a more mathematical notation, the output C'_{t-1} of the forget gate is defined as:

$$f_t = \sigma(\theta_f(x_t \frown h_{t-1}) + b_f) \tag{3.7}$$
$$C'_{t-1} = C_{t-1} \odot f_t \tag{3.8}$$

Where θ_f is the matrix of weights of the f NN layer and b_f is the bias.

2. **input gate** - a gate that takes into account h_{t-1}, x_t and C'_{t-1} (the output of the forget gate), and creates a new cell state C_t. The input gate uses two independent NN layers to do so: A hyperbolic tangent activated NN layer to generate a new cell state vector $\tilde{C}_t$, and another sigmoid activated NN layer to generate another forget vector i_t to decide the importance of each slot in $\tilde{C}_t$ (similar to f_t). Then, the input gate takes the Hadamard product of $\tilde{C}_t$ and i_t, and adds it to C'_{t-1} to create the new cell state C_t. In a more mathematical notation, the output C_t of the forget gate is defined as:

$$i_t = \sigma(\theta_i(x_t \frown h_{t-1}) + b_i) \tag{3.9}$$
$$\tilde{C}_t = tanh(\theta_C(x_t \frown h_{t-1}) + b_C) \tag{3.10}$$
$$C_t = C'_{t-1} + \tilde{C}_t \odot i_t \tag{3.11}$$

Where θ_i, θ_C are the weights of the i and $\tilde{C}$ NN layers respectively. And b_i, b_C are the biases of the two NNs.

3. **output gate** - a layer that uses x_t, h_{t-1} and C_t (the new cell state) to create a new hidden state h_t. First, the output gate feeds x_t and h_{t-1} to a sigmoid activated NN layer to get o_t. Then it runs an element-wise hyperbolic tangent function on the cell state C_t to push values between -1 and 1 and the new hidden state h_t will be the Hadamard product of the output and o_t. And in a more mathematical notation:

$$o_t = \sigma(\theta_o(x_t \frown h_{t-1}) + b_o) \tag{3.12}$$
$$h_t = o_t \odot tanh(C_t) \tag{3.13}$$

Where θ_o are the weights of the output gate NN layer. And b_i, b_C are the biases of the two NNs.

The key here, is that the forget gate prevents the gradients from vanishing. In contrast to vanilla RNNs, where the sum in Eqn. (3.6) is comprised of expressions with a similar behaviour (which eventually leads to vanishing/exploding gradients), the forget gate regulates the gradients of different layers (time steps) which enables the network to update parameters in such a way that different sub gradients do not behave in a similar manner.

Note that the sigmoid network of the update gate is very similar to the forget gate and that the hyperbolic tangent function is performed both as an element-wise operation and as a separate layer.

This redundancy in calculations will lead to longer computation times. So the Gated Recurrent Unit (GRU) cell architecture is suggested as a simplification of the LSTM (both comprehension and computational) that maintains the concept of variable weighted memory (variable weight for different time steps).

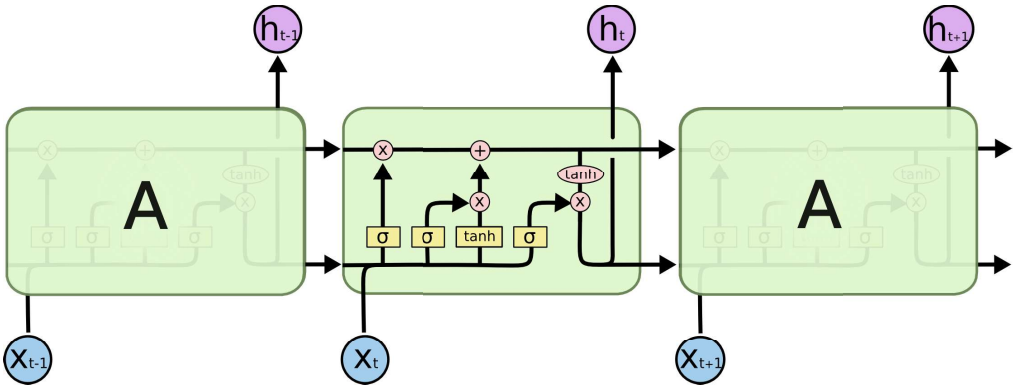


Figure 3.4: The architecture of the LSTM cell. σ is a fully connected layer with sigmoid activation function [61]

3.3.3 GRU

The Gated Recurrent Unit (GRU) [47] provides a simplification of the LSTM model by changing the cell architecture to merge the forget and input gates into a single update gate, as well as merging the cell state into the hidden state (Fig. 3.5). This variation not only provides better time complexity but will also improve the accuracy over the random walks dataset (as will be viewed in Chapters 4, 5). The output of a GRU cell architecture is calculated as follows:

1. **update gate** Determines the weight of each value in the updated hidden state vector $\tilde{h}_t$, as well as how much of the past h_{t-1} needs to be added to the updated hidden state. In a more mathematical notation, the output z_t of the update gate is defined as:

$$z_t = \sigma(\theta_z(x_t \frown h_{t-1}) + b_z)) \tag{3.14}$$

   Where θ_z is the matrix of weights of the z NN layer and b_z is the bias.

2. **reset gate** Creates a temporary updated hidden state $\tilde{h}_t$ by first calculating how much of the previous hidden state needs to be forgotten r_t. r_t is calculated by feeding the input and previous hidden state to a sigmoid activated FC network. At the next step, the updated hidden state will be calculated by once again feeding the input with the Hadamard product of the previous hidden state and the reset gate values r_t, this time to a hyperbolic tangent activated FC. In a more mathematical notation, the output $\tilde{h}_t$ of the reset gate is defined as:

$$r_t = \sigma(\theta_r(x_t \frown h_{t-1}) + b_r)) \tag{3.15}$$

$$\tilde{h}_t = tanh(\theta_h(x_t \frown (r_t \odot h_{t-1})) + b_h) \tag{3.16}$$

   Where θ_r, θ_h are the weights of the r and h NN layers respectively. And b_r, b_h are the biases of the two NNs.

3. **output** Not to be confused with a gate (has no learned parameters), this part uses the values of the update gate z_t and the inverse of the update gate $1 - z_t$, as well as the previous hidden state h_{t-1} and the updated hidden state $\tilde{h}_t$, to calculate the new hidden state h_t. Formally defined as:

$$h_t = (1 - z_t) \odot \tilde{h}_t + z_t \odot h_{t-1} \tag{3.17}$$

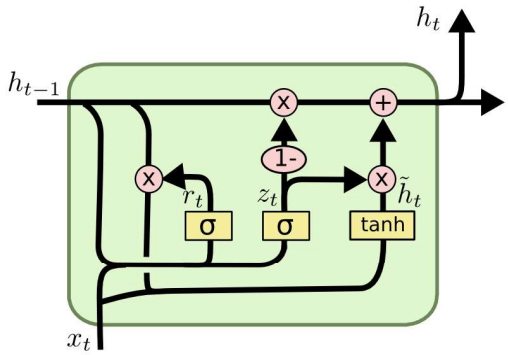


Figure 3.5: The cell architectures of the GRU [61]

The current state-of-the-art methods contain many more variations to the gate structures presented here such as: GRU1, GRU2, GRU3, and Minimal Gated Unit (MGU) [65].

A further improvement to the NN's complexity is gained by using a multi-layered GRU. This means that every GRU block consists of more than one layer, and every layer feeds its output not only to the next time step but also to the layer above it. The final classification will use the output of the final layer n at time step t, written as $y_t^{(n)}$. The custom architecture used in this work is a 3 layered network, based on the GRU. This means the output at time step t is given as:

$$y_t^{(1)} = f(x_t, h_{t-1}^{(1)}) \qquad y_t^{(2)} = f(y_t^{(1)}, h_{t-1}^{(2)}) \qquad y_t^{(3)} = f(y_t^{(2)}, h_{t-1}^{(3)}) \tag{3.18}$$

Where f is the function representing the GRU block, and the hidden state of every cell is a vector with 40 numbers.

The output of the three GRU layers at the final step $y_t^{(3)}$ is then used as the input for an FCN with the following structure (Fig. 3.6):

1. Input layer with 40 neurons
2. Dropout layer with a probability of 0.2
3. Classification layer with 4 a ReLU and LogSoftmax activation functions

The dropout layer simply ignores some of the outputs of the input layer with a probability of 0.2 in the training phase only. It is used as a form of regularization to avoid overfitting the network.

Chapter 4 presents the hyperparameters and optimization algorithms used to train the models. The next chapter also discusses the performance of each model as well as its benefits in order to determine a new state of the art for classifying random walks.

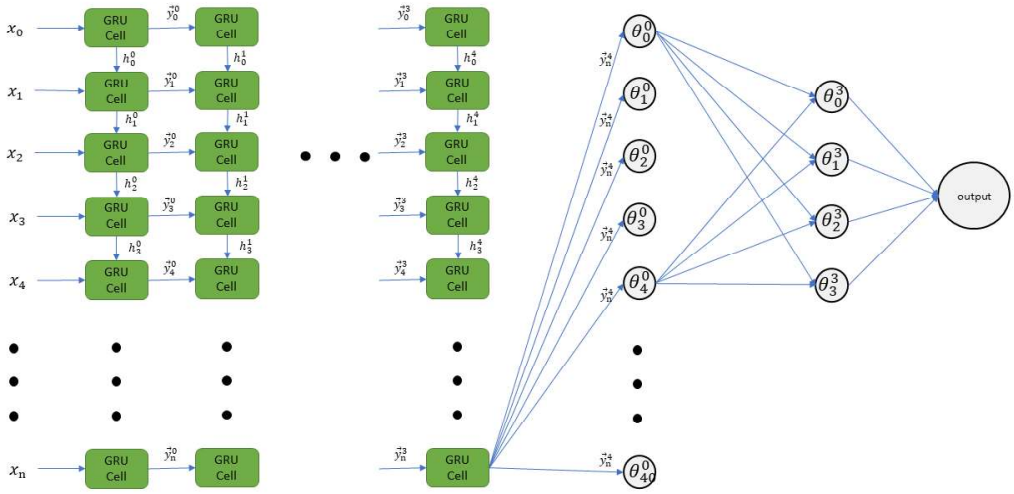


Figure 3.6: A diagram of the Gated Recurrent Unit Neural Network

Chapter 4

Identify Generating Models

In this chapter, the three NNs explained in Chapter 3 will be used to identify the generating model of a random walk based on a single trajectory sampled.

4.1 Datasets and Optimization

In order to get a comprehensive view of their strengths and weaknesses, the NNs are trained on four different datasets. The datasets are created using the methods for generating random walks explained in Chapter 2, along with a few tweaks to the generation process that ensure the datasets simulate the natural processes as closely as possible.
First, the diffusion coefficient of every trajectory generated is sampled from a uniform distribution $D \sim U(0,1)$. The reason for this tweak is to remove the possibility of bias in the trained network.
Second, the waiting time distribution of the CTRW is a power-law with a variable alpha sampled from a uniform distribution $\alpha \sim U(0,2)$ for every path generated. This tweak is added in order to make sure the resulting NNs will learn to differentiate between regular and sub-diffusive processes.
Third, the Hurst exponent of the FBM is also sampled from a uniform distribution $h \sim U(0,1)$ while H (Hurst exponent) values that yield Brownian motion are discarded $h \notin (0.4, 0.6)$.

For the sake of training and testing the networks, 4 datasets were created, all consisting of 6 different random walks, divided into 4 classes (also viewed in Table 4.1):

1. The first dataset consists of $4 \cdot 10^5$ random walks with 100 steps each, divided equally into the following classes: Brownian motion with Gaussian distributed step function, Brownian motion with Laplace distributed step function, Levy walk with a power-law distributed step function $\psi(\tau) = A_\alpha \tau^{-\alpha}$ where $1 < \alpha < 2$, FBM, CTRW with Gaussian distributed jump lengths and power-law waiting times, CTRW with Laplace distributed jump lengths and power-law distributed waiting times. The sample times are still $\Delta t = 0.05$ and the datasets are shown in (Fig. 4.1). For this dataset, the networks were trained for 30 epochs.

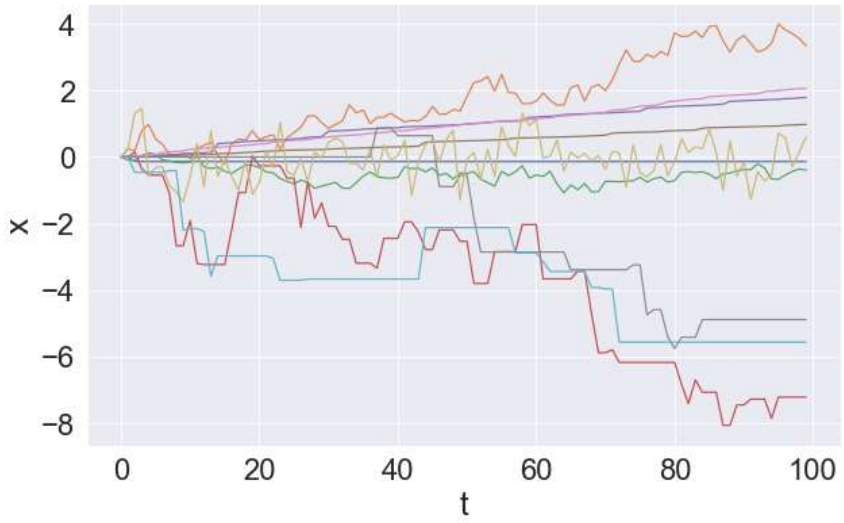
2. The second dataset used for training is the same as the first one with the exception that we have $4 \cdot 10^4$ random walks with 1000 steps each. This dataset is used to determine the NN's capability to analyze long processes. For this dataset, the networks were trained for 30 epochs.

3. The third dataset used for training is also the same as the first one with the exception that we have 5000 random walks with 100 steps each. This dataset is used to determine the network's capability of learning effectively with a limited amount of data. For this dataset, For this dataset, the networks were trained for 30 epochs.

4. The fourth dataset used for training is also the same as the first one with the exception that we now have $4 \cdot 10^4$ random walks with 20 steps each. This dataset is used to determine the NN's capability to analyze short processes, which is the most realistic use case as in most experimental setups we are only able to measure short trajectories. The ability to classify short walks will also be useful when analyzing mixed model random walks, as will be discussed in chapter 5.

Samples	Trajectory length	Training epochs
$4 \cdot 10^5$	100	30
$4 \cdot 10^4$	1000	30
$5 \cdot 10^3$	100	30
$4 \cdot 10^4$	20	100

Table 4.1: A table of the essential information about the datasets used to train and test the NNs.



(a)

Figure 4.1: A subset of the first dataset. The first dataset contains $4 \cdot 10^5$ random walks generated by different models, all with the degrees of freedom described in Chapter 2. While the CTRW and positively correlated FBM trajectories are relatively simple to identify, the distinction between Brownian Motion, Levy walks, and FBM with different Hurst exponents is a complicated task.

4.2 Optimization Method

For the optimization process, the method used is the Adaptive Moment (Adam) [66] optimization algorithm. This optimization method takes advantage of two previously common optimization methods, which will be explained in this section.

Although it introduces a large memory overhead, the Adam optimizer is an essential part of the training process, as it yields a minimum of 20% increase in accuracy across all of the networks discussed in this work.

A deeper dive into the performance of the networks and their implications is offered in the following sections of this chapter.

4.2.1 AdaGrad

The first algorithm is the Adaptive Gradient (AdaGrad) algorithm [67]. In more traditional optimization methods such as the SGD, the most common improvement to the learning rate is to make α decay over time. This was done in order to account for the fact that as the optimization process converges to the minima of the potential, smaller steps are needed in order to pinpoint the exact location of the minima. However, since the NN has many parameters, each corresponding to a different feature of the input data, the rate of convergence to the minima is varying between dimensions. Furthermore, the decay rate of α adds another hyperparameter that needs to be tweaked when optimizing the NN.

AdaGrad solves this problem by assigning each parameter of the NN its own learning rate, which would adapt during training to the rate of convergence in each dimension of the potential. The mathematical representation of the optimization step of the AdaGrad learning method is:

$$G_t = \nabla_\theta L(\theta_t; x, y) \tag{4.1}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\varepsilon + \sum_{\tau=1}^{t} G_\tau^2}} \odot G_t \tag{4.2}$$

Where ε is some small constant added to insure no division by zero occurs. The sum at the denominator corresponds to taking the sum of the squares of all past gradients with respect to all parameters θ.

The AdaGrad essentially grants the learning process per-weight learning rates that are independently decaying over time according to their respective gradient. The benefit from using the AdaGrad algorithm is that in addition to losing the hyperparameter fine-tuning needed for the time decay of α, it also greatly reduces the need to fine-tune the learning rate itself. Ultimately, the learning rate chosen for most NNs that train using AdaGrad is $\alpha = 0.01$. However, since only positive values are summed in the denominator, AdaGrad has a monotonically decreasing learning rate. This decreasing learning rate will cause the network to prematurely converge which will inhibit its learning. Early solutions to this issue include the AdaDelta [68], which restricts the window of accumulated past gradients to some fixed size w.

4.2.2 RMSprop

The Root Mean Squared Propagation (RMSprop) algorithm is a variation of the AdaDelta algorithm. As mentioned, the AdaDelta algorithm improves AdaGrad by restricting the

amount of accumulated past gradients. However, instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. In a mathematical form:

$$\mathbb{E}[G_t^2] = \gamma\mathbb{E}[G_{t-1}^2] + (1-\gamma)G_t^2 \tag{4.3}$$

RMSprop sets $\gamma = 0.9$, which yields the following optimization algorithm:

$$\mathbb{E}[G_t^2] = 0.9\mathbb{E}[G_{t-1}^2] + 0.1G_t^2 \tag{4.4}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\varepsilon + \mathbb{E}[G_t^2]}} \odot G_t \tag{4.5}$$

4.2.3 Adam

Adam takes the RMSProp and adds the concept of momentum by keeping an exponentially decaying average of past gradients in addition to the average of past squared gradients. In total, the Adam algorithm maintains the same update rule of the RMSprop (and the AdaDelta) and adds the first momentum to that process.

The maintained averages of the optimization process are now:

$$m_t = \mathbb{E}[G_t], \qquad m_t = \beta_1 m_{t-1} + (1-\beta_1)G_t \tag{4.6}$$

$$v_t = \mathbb{E}[G_t^2], \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2)G_t^2 \tag{4.7}$$

Where m_t and v_t are estimates of the first and second moments of the gradients respectively (initialized at 0). $\beta_1, \beta_2 \in [0,1)$ are hyper-parameters controlling the exponential decay rates of the moments m_t, v_t. G_t is the gradient of our potential evaluated at time t.

The first and second moments are strongly biased towards 0, especially during the initial time steps of the optimization process. Therefore, the Adam algorithm introduces a bias-corrected moments:

$$\hat{m}_t = \frac{m_t}{1-(\beta_1)^t} \tag{4.8}$$

$$\hat{v}_t = \frac{v_t}{1-(\beta_2)^t} \tag{4.9}$$

In total, our optimization step looks like:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \varepsilon}\hat{m}_t \tag{4.10}$$

Where ε is a smoothing term that avoids division by 0.

The initial learning rate of the optimization process is set to $\alpha_0 = 5 \cdot 10^{-3}$ for the training of all the networks in this section. Note that the metric chosen to evaluate performance was accuracy, and that all experiments were repeated until a maximal standard deviation of 3% accuracy was reached.

4.3 FC Results

The first network to be trained and tested is the FC and its results will now be discussed (the results can be viewed in Table 4.2). The FC exhibited the fastest training time out of all three NNs tested. However, this property was the only area where the FC triumphed over the other two NNs.

First Dataset

As shown in Fig. 4.2a, the FC reached 88% accuracy over the first dataset. This result is already acceptable for real-world use, as taking the mean over a small ensemble of trajectories sampled would increase the confidence. At a first glance, the FC's performance over the first dataset seems to indicate that this network is able to capture the correlations between time steps. However, this assumption is disproven by examining the FC's performance over the second dataset. In the following subsection, this accuracy would be used as a baseline.

Second Dataset

Training the FC on the second dataset (1000 step paths sampled in each trajectory) yields some unexpected results. The logical assumption for similarities between neural networks and more classical algorithms would be to compare the neural network to a time-averaged MSD. In such a comparison, the increasing length of the trajectory would mean more data is given, therefore the accuracy should increase. However, as seen in Fig. 4.2b, the accuracy of the FC drastically dropped to an average of 56%, which is a 34% decrease in performance from the baseline. There could be three explanations for this performance reduction:

1. The first argument would be that in addition to the increase in trajectory length, there was also a reduction in the size of the training dataset (which was essential to maintain reasonable training times). This reduction in sample amount might be the cause of the network's poor performance. This hypothesis is rejected by examining the results over the fourth dataset ($4 \cdot 10^4$ trajectories of length 20), which had the same dataset size but shorter trajectories.

2. The second hypothesis is the claim that the network is simply not large enough to capture the correlations, which raises the question of whether a wider and deeper version of the FC might be able to classify our random walks better. However, as explored in the Appendix B), no noticeable performance gain could be seen from the use of wider and deeper networks.

3. The third hypothesis, which is also the one accepted by us, is that the FCN is unable to capture the correlations in the trajectory. This option will be further explained when the performance of the CNN over the same dataset is presented (Section 4.4)

This difficulty to train a long path classifier will also be demonstrated, to a lesser extent, by the CNN (4.4).

Third Dataset

Training the FC on the third, small dataset ($5 \cdot 10^3$ samples) shows the FC's vulnerability to decreasing dataset sizes. As seen in Fig. 4.2c, the network reached an accuracy of 65% on the test set, which is a 26% decrease in performance from the baseline. This result is a little less surprising as a common requirement for training NNs is a sufficiently large dataset. However, as will be seen in the GRU results (Section 4.5), the network's ability to learn the temporal relations within the trajectory reduces its need for a large dataset. Therefore, this result provides further affirmation to our third hypothesis that the FCN is unable to capture correlations correctly.

Fourth Dataset

Training the FC on the fourth dataset ($4 \cdot 10^4$ samples, 20 steps each) somewhat surprisingly yielded the best results with an average of 89% (Fig. 4.2d). There are two noteworthy facts to take into account when looking at these results:

- The first, is that the FC performed badly on a dataset with the same amount of samples but long trajectories.
- The second, is that the FC performed better on this short trajectory dataset than it did on the first dataset, which contained more samples.

In total, both the first and second datasets contained more information about the underlying structure of the random walks, yet the accuracy of the FC dropped. These results further strengthen the claim that the FC is unable to capture the temporal correlations required to robustly classify random walks.

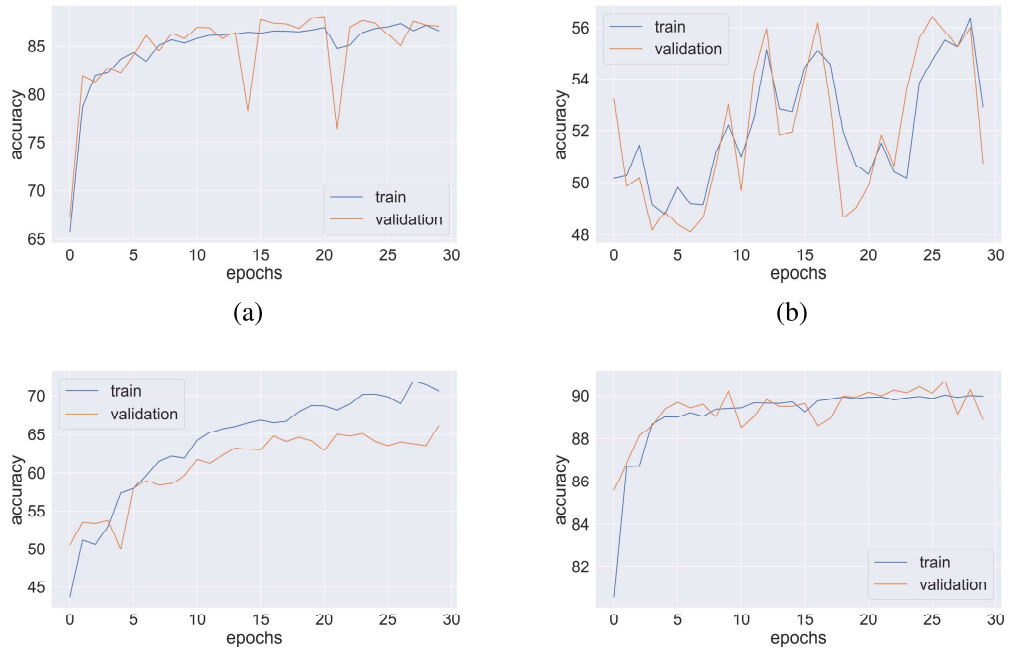


Figure 4.2: The accuracy of the FC on the training and test datasets as a function of the epochs. Figure 4.2a illustrates the case of $4 \cdot 10^5$ walks, 100 steps each. Figure 4.2b illustrates the case of $4 \cdot 10^4$ walks, 1000 steps each. Figure 4.2c illustrates the case of $5 \cdot 10^3$ walks, 100 steps each. Figure 4.2d illustrates the case of $5 \cdot 10^5$ walks, 20 steps each.

Another problem we would like to solve is the network's inability to process variable-length random walks, as the size of the first layer must always match the length of the random walk.

Samples	Trajectory length	Accuracy
$4 \cdot 10^5$	100	88%
$4 \cdot 10^4$	1000	56%
$5 \cdot 10^3$	100	65%
$4 \cdot 10^4$	20	89%

Table 4.2: The accuracy of the Fully Connected Neural Network over the four datasets

4.4 CNN Results

The second network to be trained and tested is the CNN and its results will now be discussed (the results can be viewed in Table 4.3). The CNN performed marginally better than the FC. However, the CNN still suffers from the same shortcomings the FC has.

First Dataset

As shown 4.3a), the CNN reached an average accuracy of 94% when training on the first dataset ($4 \cdot 10^5$ samples, 100 steps each). These results present a 7% improvement over the FC, which already had an acceptable performance for real-world scenarios. Also noting once again that these results were taken for single trajectory classification. And since most experimental results include particles under the same stochastic forces, would be reasonable to assume that acquiring a small ensemble of trajectories generated by the same random walk model would be feasible. Therefore, taking the mean classification probability of an ensemble of trajectories would increase the accuracy of the CNN.
However, examining the CNN's performance over the rest of the datasets reveal the same inability of capturing the underlying temporal relations present in the random walks.

Second Dataset

Training the CNN on the long trajectory dataset ($4 \cdot 10^4$ samples, 1000 steps in each trajectory sample) yields an average accuracy of 84%, As shown in Fig. 4.3b. This 10% accuracy decrease is the same performance dip exhibited by the FC that highlights the CNN's inability to capture correlations. Once again, as explained in the previous section (Section 4.3), there could be three explanations for this performance reduction. The first two explanations are rejected due to the same arguments presented in Section 4.3.

Now expanding on the third point mentioned in the previous section regarding the inability of the FCN and CNN networks to classify long walks. Our assumption is that it stems from the inherent method of classification learned by these networks. As explained in the CNN exposition (Section 3.2), the CNN trains several convolution kernels in order to extract unique features that would distinguish a certain class from the rest of the classification options. The FC network performs the same actions to a lesser extent, as unique features are represented by perceptrons. We think the networks learn to notice anomalous events within a given random walk (such as a Levy flight, or a big leap in a short interval caused by positively correlated FBM) and use these events as the unique distinctive features that differentiate between different models. However, as the trajectory length increases, the likelihood of these anomalous events existing in different models increases as well. Therefore, a dataset containing long trajectories would have no unique features that the networks would be able to use to distinguish between the random walks. A possible

solution would be to introduce more distinctive anomalous events into the random walk dataset. This could be done by simply increasing the size of the dataset in proportion to the trajectory length. But since we only have a finite amount of digital storage, we would like to have a classifier that would be based on the statistics of said anomalous events. This ability to distinguish based on the statistics of the anomalous events will be useful again in Chapter 5.

Third Dataset

Training the CNN on the third, small dataset ($5 \cdot 10^3$ samples, 100 steps in each sample) shows the CNN's vulnerability to decreasing dataset sizes. As shown in (Fig. 4.3c), the network reached an accuracy of 82% on the test set, which is a 14% decrease in performance from the baseline. This reduction is less dramatic than the one exhibited by the FC and is once again expressing a common weakness shared by most NNs. However, as mentioned in the previous section, the GRU's ability to learn temporal relations within the trajectory will reduce its dependence on dataset size. Therefore, this result provides further affirmation to our third hypothesis that the CNN is unable to capture correlations correctly.

Fourth Dataset

Training the CNN on the fourth dataset ($4 \cdot 10^4$ samples, 20 steps each) yielded an average of 91% accuracy (as shown in Fig. Fig. 4.3d).

Once again, there are two noteworthy facts to take into account when looking at these results:

- The first, is the efficiency of the CNN. Taking $Samples \cdot Length$ as a measure of the amount of information presented by the dataset means the CNN lost only 3 points when trained with 50 times less information than the baseline (first dataset).
- The second, is that the FC performed marginally better on this short trajectory dataset than it did on the long trajectory dataset (second dataset). Taking the same $Samples \cdot Length$ measure of information into account means the CNN performed better in this case than it did when trained with 50 times more information. Once again, these results prove the CNN's weakness when it comes to identifying underlying correlations.

Samples	Trajectory length	Accuracy
$4 \cdot 10^5$	100	94%
$4 \cdot 10^4$	1000	84%
$5 \cdot 10^3$	100	82%
$4 \cdot 10^4$	20	91%

Table 4.3: The accuracy of the Convolutional Neural Network over the four datasets

Overall, the CNN exhibits marginally better results than the FC, and can even be considered powerful enough to analyze actual data. However, besides overcoming the performance deficits present in both the FC and the CNN, another goal that would be achieved by the GRU in the next chapter is the ability to identify random walks with variable trajectory lengths.

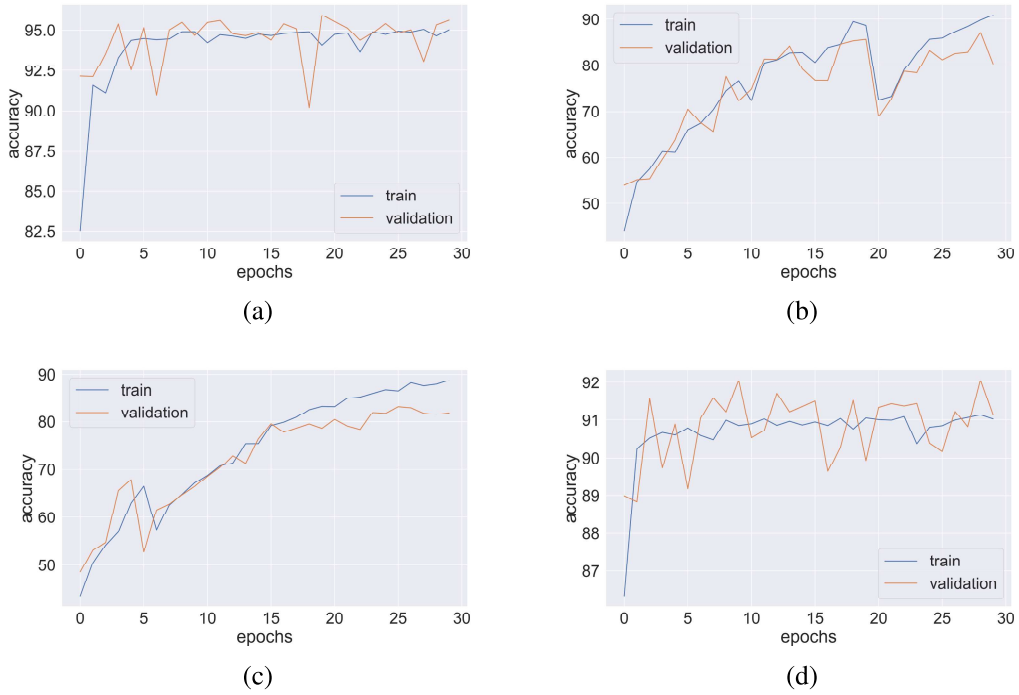


Figure 4.3: The accuracy of the CNN on the training and test datasets as a function of the epochs. Figure 4.3a illustrates the case of $4 \cdot 10^5$ walks, 100 steps each. Figure 4.3b illustrates the case of $4 \cdot 10^4$ walks, 1000 steps each. Figure 4.3c illustrates the case of $5 \cdot 10^3$ walks, 100 steps each. Figure 4.3d illustrates the case of $5 \cdot 10^5$ walks, 20 steps each.

4.5 GRU Results

The GRU network yielded better results than the previous networks on all four datasets, as seen in Fig. 4.4 (results can also be viewed in Table 4.4). As explained in Chapter 3, the GRU processes each time step in the trajectory, and in addition to outputting a classification value, will also maintain a hidden state that serves as a long term 'memory'. This unique feature grants the GRU two of the desired qualities that have been missing from the previous networks:

1. Since the GRU calculates the output for each individual time step, it is able to classify trajectories with variable-lengths.

2. The existence of a hidden state between time steps serves as the equivalent of a correlation function. Meaning that learning the underlying dynamics of the random walk is an integral part of the GRU's training process.

Examining the results in more detail reveals how these qualities are reflected in the results.

First Dataset

As shown in Fig. 4.4a, the GRU reached 97% accuracy over the first dataset. However, as previously mentioned, the performance of the networks on the first dataset is insufficient

to prove that the network learns the temporal relations in the trajectory. Yet, when taking into account the GRU's ability to classify variable-length trajectories, these results show that the GRU would be an appropriate tool for an experimental environment. Also noting again, that in real-world use, it would be possible to increase the accuracy of the GRU even further when an ensemble of trajectories of the same random walk model is sampled.

Second Dataset

Training the GRU on the second dataset ($4 \cdot 10^4$ samples, 1000 steps each) yields perhaps the most noteworthy result. As seen in Fig. 4.4b, The GRU was the only network that surpassed the baseline performance when trained on long datasets, with an average of 99%. These results show that the GRU indeed captures the temporal relations contained within the random walk rather than simply classifying based on anomalous events that distinguish each random walk. In order to further prove this point the following experiment was performed: A GRU network will be trained on the first dataset ($4 \cdot 10^5$ samples, 100 steps each) and then tested on the second dataset. Now, referring again to the point mentioned in Section 4.3, the computation process of the NN is compared to a time-averaged MSD. In addition, the final layer of the NN is compared to a function between the TA-MSD and the correlation function of the random walk. This experiment shows a basic assumption about the comparison: As the TA-MSD algorithm is calculated over longer times, the resulting function will fit better to the correlation function. Therefore, if the GRU learns to compute a correlation-aware algorithm over the random walk that is comparable to the TA-MSD, its fit to the actual correlation function of the random walk should increase in accuracy. This feature should occur regardless of the trajectory length the GRU was trained on. This experiment over the GRU yielded the exact same results as the original training over the dataset (99% accuracy), indicating that the GRU indeed learns a distinct auto-correlation function. This ability will once again be demonstrated in Chapter 5.

Third Dataset

Training the GRU on the third, small dataset ($5 \cdot 10^3$ samples, 100 steps each), shows the GRU's resilience to decreasing dataset sizes compared to the other two architectures. As seen Fig. 4.2c, the network reached an average accuracy of 93% on the test set, which is only 4 points less than the baseline (compared to a decrease of 12 and 26 points for the CNN and FC respectively). This resilience to decreasing dataset sizes could also be inferred from examining the GRU's performance over the previous two datasets. In the previous training scenarios, the GRU reached an accuracy very close to its peak accuracy on the fourth epoch (Fig. 4.4a, 4.4b). These results show that adding the ability to learn underlying temporal correlations to the network architecture reduces its dependence on dataset size.

Fourth Dataset

Training the GRU on the fourth dataset ($4 \cdot 10^4$ samples, 20 steps each) shows once again that the learning process of the GRU depends on the clarity of the auto-correlation function. It is clear that as the length of the trajectory shrinks, the temporal relations between different time steps become less apparent. Therefore, if the expectation from the GRU

is that it learns to classify random walks based on their auto-correlation function, then this decrease in trajectory length should also affect the statistical significance of the auto-correlation function belonging to a specific random walk. As shown in Fig. 4.4d, the GRU reached an average of 94% accuracy, which is 3 points lower than the baseline. As mentioned earlier, this result is to be expected.

The GRU's performance over the fourth dataset is the most beneficial to us when approaching the next chapter.

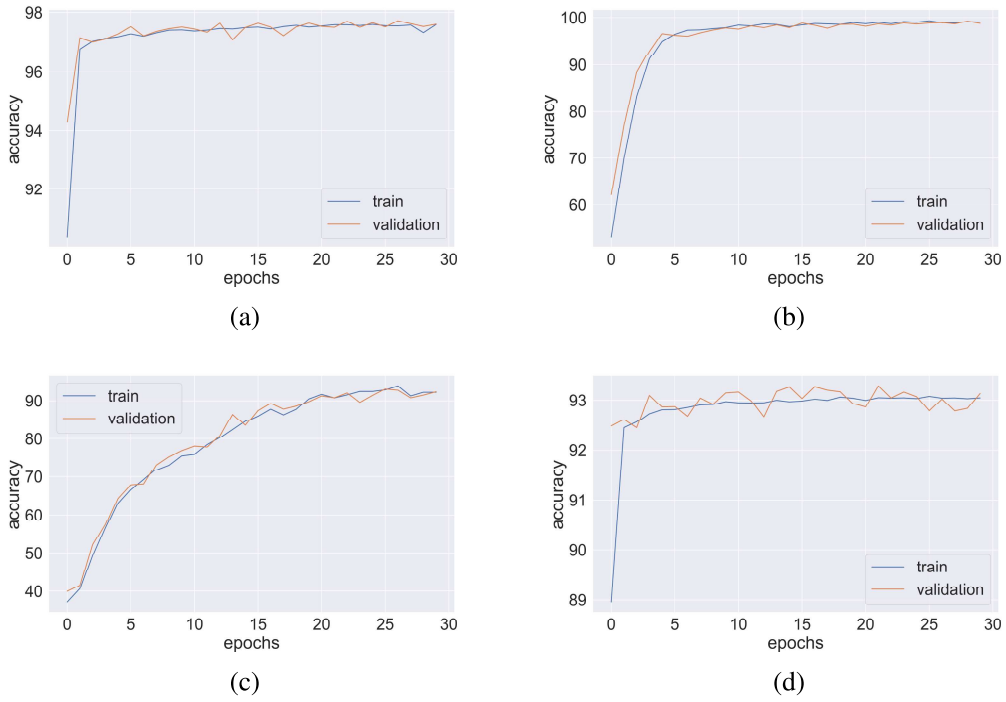


Figure 4.4: The accuracy of the GRU on the training and test datasets as a function of the epochs. Figure 4.4a illustrates the case of $5 \cdot 10^5$ walks, 100 steps each. Figure 4.4b illustrates the case of $5 \cdot 10^4$ walks, 1000 steps each. Figure 4.4c illustrates the case of $5 \cdot 10^3$ walks, 100 steps each. Figure 4.4d illustrates the case of $5 \cdot 10^5$ walks, 20 steps each.

Samples	Trajectory length	Accuracy
$4 \cdot 10^5$	100	97%
$4 \cdot 10^4$	1000	99%
$5 \cdot 10^3$	100	93%
$4 \cdot 10^4$	20	94%

Table 4.4: The accuracy of the Gated Recurrent Unit Neural Network over the four datasets

	GRU	CNN	FCN
$5 \cdot 10^5$ walks 100 steps each	97%	94%	88%
$5 \cdot 10^4$ walks 1000 steps each	99%	84%	56%
$5 \cdot 10^3$ walks 100 steps each	93%	82%	65%
$5 \cdot 10^5$ walks 20 steps each	94%	91%	89%

Table 4.5: The accuracy of the different networks described in the chapter over the single model datasets.

Chapter 5

Identify Bi-Modal Random Walks

This chapter explores scenarios where the sampled trajectory is generated using up to two models. Such random walks will be referred to throughout this work as BiModal random walks.

BiModal random walks can be observed in various physical examples. First, is the foraging patterns of wandering and black-browed albatross birds. In contrast to previous assumptions, the work in [69] shows that these birds foraging patterns are best modeled by a combination of Levy and Brownian random walks.

Another example, are the transitions between different states during sleep [70]. In this case, the duration of wake periods is characterized by a scale-free power-law distribution, while the duration of sleep periods has an exponential distribution with a characteristic time scale. Therefore, the transitions between different states of the brain during sleep are modeled as a regular Brownian motion when the subject is asleep, and a random walk moving in a logarithmic potential when the subject is awake.

In this work, the transition in the BiModal random walks will occur at random times and will involve one of the four models discussed in Chapter 2. This time-dependent transition also means that the generated trajectory might not belong to a BiModal random walk. In other words, the Deep Neural Network will now have to classify all the random walk models contained in a sampled trajectory. While this problem seems fairly similar to the one discussed in the previous chapter (4), the chosen network will have to perform better on the single trajectory classification task. Going back to the experimental use case, the assumption made in the previous chapter (4) was that repeating the experiment in order to increase the trajectory ensemble of the random walk is possible. The obvious benefit to increasing the number of trajectories sampled would be that averaging over all the network classifications in the ensemble would produce a more accurate result, in the same manner that the MSD would produce more reliable results as the ensemble size increases. However, in the BiModal case, there is another layer of complexity as not all trajectories in the ensemble contain a transition from one model to the other. Therefore, the BiModal random walks require that our classification algorithm of choice would be able to generate a prediction based on a single sampled trajectory.

The goal of this chapter is to classify the two random walks contained in some sampled trajectory (or single random walk in case where no transition happened). The naive idea would be to compare this problem to an anomaly detection problem. However, this comparison relies on the assumption that identifying a trajectory as a BiModal random walk and identifying the two random walks contained in that trajectory are two problems with similar difficulty. This assumption will be disproved later on in the chapter. In order

to achieve a good classifier for the BiModal random walk, two possible approaches could be used:

1. Training a DNN to analyze the BiModal random walk given the entire trajectory. In other words, a dataset containing BiModal random walks would be created and the network will be trained on that dataset.

2. Training a DNN to analyze the BiModal random walk given fragments of the trajectory. In other words, a network would be trained in the same way it did in Chapter 4. Then, the trained network will be given the first and last fragments of the trajectory and make its prediction based on these trajectory segments.

A deeper dive into the results of the two approaches is offered in the coming sections of this chapter.

5.1 Analysis Preliminaries

5.1.1 BiModal Dataset

The BiModal dataset is generated by extending the single trajectory dataset from Chapter 2. Therefore, all the degrees of freedom mentioned when creating the dataset for Chapter 4 remained the same. Further changes made to the trajectories generator are:

- For any trajectory, there is a 50% chance of transition. This rule is added in order to increase the complexity of the problem as well as to more closely mimic real-world scenarios.

- The time t when transition between two generating models occurs is uniformly distributed such that: $t_{transition} \sim U(\frac{1}{6}T, \frac{5}{6}T)$. This rule effectively means that samples that contain a transition that occurred too early are disregarded in order to reduce complexity.

- In a transition, there is a 50% chance that the diffusion constant will also change. This rule is added in order to remove any biases that might affect the network's classification. In other words, in order to avoid a scenario where the network learns to identify a transition between models based on a change in the diffusion constant, the diffusion constant may remain the same after the transition.

- In cases where no transitions occurred in the trajectory there is no transition between generating models, There is a 25% chance that some hyper parameter of the random walk will change. The hyper parameters that may change are: diffusion constant, anomalous diffusion exponent (for CTRW and Levy walks), and Hurst exponent (for FBM). The reason for this rule is to remove the same bias presented in the previous point. Namely, the network should be able to distinguish between a transition in generating model and any other anomalous events that might arise in the sampled trajectory.

Note that for this dataset, the amount of labels increases as a function of the amount of unique generating models. The original dataset (Chapter 2) produced trajectories with four unique generating models: Brownian, CTRW, FBM, and Levy. In the BiModal dataset, all combinations are labeled as unique trajectories as the ordering is taken into

account (Brownian to CTRW, CTRW to Brownian...). This means there are now 16 label options for any given trajectory. A subset of the BiModal dataset can be viewed in 5.1

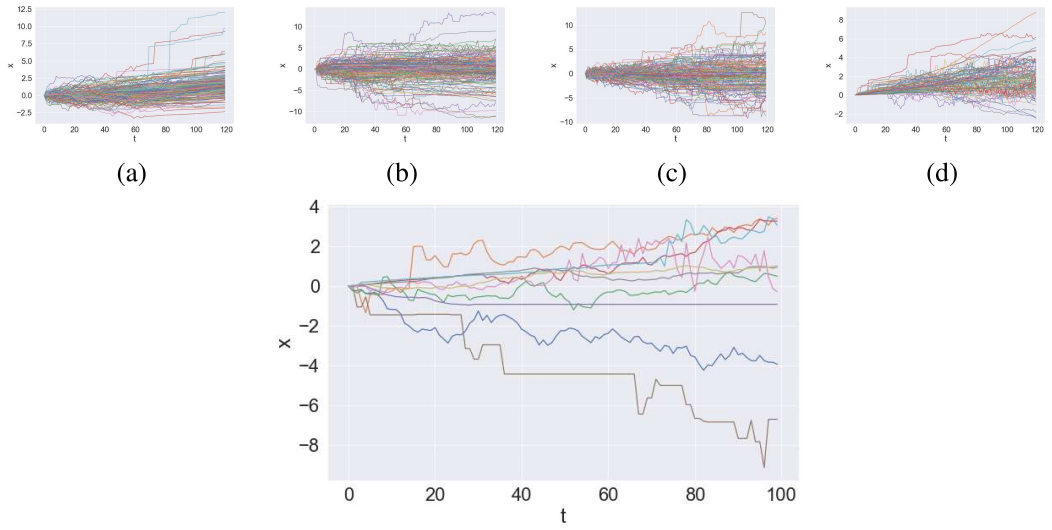


Figure 5.1: A small subset of the dataset used to train the networks featuring different transitions between processes. Fig. 5.1a shows transitions from Brownian motion to levy flights occurring at random times in the interval $[\frac{1}{6}t, \frac{5}{6}t]$. Fig. 5.1b shows transitions from CTRW to FBM. Fig. 5.1c shows transitions from FBM to CTRW. Fig. 5.1d shows transitions from Levy to FBM. Fig. 5.1e shows a subset of the entire BiModal dataset on the same scale, which highlights the difficulty to classify the models in this dataset.

5.1.2 BiGRU

We now introduce yet another network architecture called Bidirectional GRU (BiGRU) [71]. This unit cell functions similarly to GRU with the exception that the final output encapsulates both forward and backward analysis of a given time series (trajectory). This new architecture yields more stable results at the expense of longer compute times. The usefulness of the BiGRU comes partly from the relative ease of implementation. It is equivalent to training two GRU recurrent networks at once, with one getting the regular trajectory as input, and the other taking the same trajectory in reverse. The final output of the network for every time step would therefore take into account all the time steps up to the current t as well as all future steps in the given trajectory. As the random walks analyzed in this work are time-reversible, it would be logical to use a NN that is able to leverage this quality.

Another tweak added to the GRU from Chapter 3 in order to make it bi-directional, is to double the size of the FC layer that makes the classification. Since there are now 40 hidden states for each direction, the new size of the FC layer would be 80 neurons.

The BiGRU outperforms all the networks mentioned in Chapter 3, as will be shown in the next sections of this chapter.

5.2 First Approach: Full Trajectory Based Classification

Our first approach to classifying the models contained in a sampled trajectory would be to feed the entire trajectory to a neural network of choice and train it to output the correct generating models. This approach poses an obvious issue in the form of exponentially increasing classification options. As mentioned in Sec. 5.1.1, the BiModal dataset contains all combinations of the original dataset with a regard to order, which sums to 16 classification options. Therefore, in order to try and maintain the performance of all networks exhibited in Chapter 4, the number of samples per label in the training set must remain the same. This means that the size of the dataset must exponentially increase with the amount of generating models we choose to train the network with. Furthermore, a common heuristic is that the size of the neural network (the number of layers and amount of neurons per layer) must increase with the number of unique classes in the dataset. Therefore, the performance of all networks over the BiModal dataset is expected to decrease substantially when using the first approach.

5.2.1 BiModal Full Trajectory Results

We left the architectures and hyperparameters of all the networks the same as it was in Chapter 4, while the new BiGRU receives the same hyperparameters as the GRU - 3 layers and 40 hidden states per direction. Training the networks on the new dataset highlights the previously discussed shortcomings of using the first approach (Table 5.1). The FC network converged at 54% (Fig. 5.2c) which supports the claim made in Chapter 4 even further, as there are hardly any anomalous events that the network could use to distinguish between generating models.

The performance of the CNN was a bit more surprising. Converging to a result even worse than the FC with 32%, it seems like the CNN is more vulnerable to the removal of anomalous events. Furthermore, as seen in 5.2b, the CNN has shown substantial overfitting over the dataset.

Overfitting is a term in machine learning that refers to cases where the accuracy over the train set increases over time while the accuracy over the validation stagnates. This indicates that the used NN architecture no longer learns the underlying dynamics in the data, but rather simply "memorizes" the classifications of the samples in the training set. Usually, a remedy to such a case would be to decrease the size of the network (remove neurons or layers of neurons), as this is usually caused by a network being a lot more complex than the dynamics in the data it is supposed to analyze. However, since this classification problem is essentially a complication of the problem discussed in Chapter 4, the assumption that the CNN architecture used is too complex for analyzing BiModal walks is invalid. The CNN's inability to identify the underlying generating models will be further highlighted when we examine the network's performance over the transition recognition task.

The GRU and BiGRU networks once again proved the benefit of hard coding the temporal relations into the architecture by outperforming the other two networks. The GRU yielded an average of 79.5% accuracy on the BiModal dataset (as seen in Fig. 5.2d). The advantage of the BiGRU when training on the BiModal dataset can be inferred from two angles when examining the results. First, the BiGRU yielded an average accuracy of 82.5% (Fig. 5.2a), which is 3 points higher than its non-reversed counterpart. Second, as seen in Fig. 5.2a, the training process of the BiGRU was a lot more stable than the

other two networks, indicating increased resilience to perturbations in the dataset which can often occur in experimental settings.

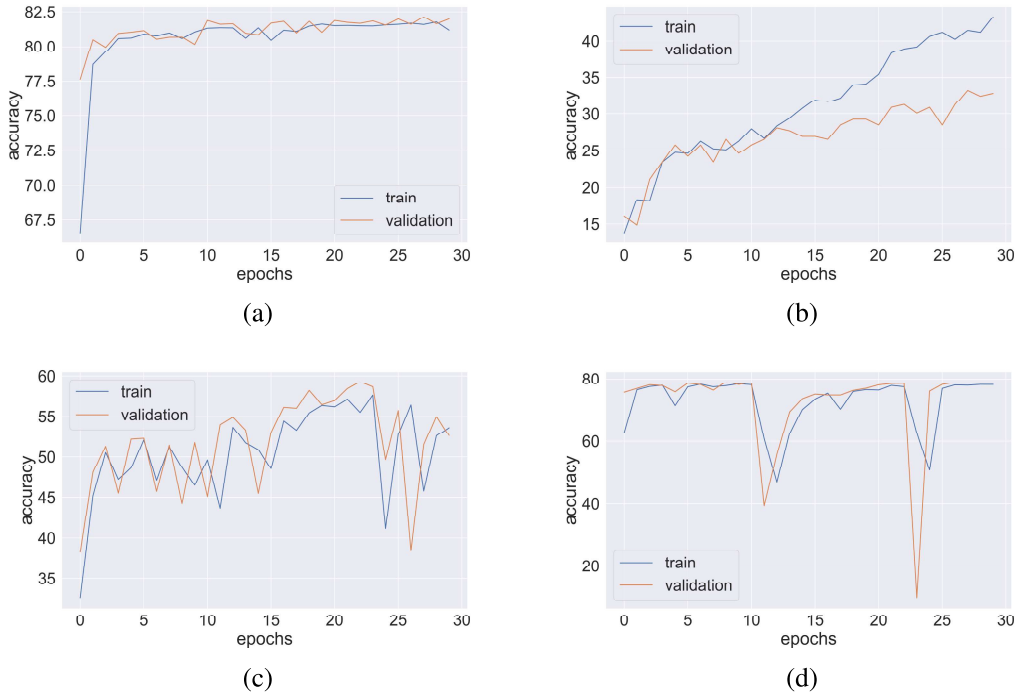


Figure 5.2: The accuracy of the networks on the training and test datasets containing BiModal trajectories, as a function of the epochs. the dataset contained $4 \cdot 10^5$ walks with 100 steps each. Figure 5.2a illustrates training the BiGRU network. Figure 5.2b illustrates training the CNN. Figure 5.2c illustrates training the FC network. Figure 5.2d illustrates training the GRU network.

However, there is no way around the fact that even the best-case scenario yields a 17% decrease in performance when compared to the performance over the single model trajectories dataset. Furthermore, we note that our RNN based architectures lose their effectiveness when training on longer walks. A possible explanation for the performance of the RNN based networks on the BiModel dataset is that the hidden states are overwritten after the transition. As mentioned before, the hidden states of the RNN capture the temporal relations between consecutive time steps. While the GRU and BiGRU are able to introduce the concept of "memory" into the hidden states, the capacity of the hidden layers might be insufficient to capture multiple dynamics at the same time. Therefore, our assumption is that the RNNs have a strong bias towards the second generating model in the trajectory, which impacts their ability to learn.

In order to solve both of these problems, we will turn to the second approach in Section 5.4.

5.3 Transition Recognition

Before attempting to solve the BiModal problem using the second approach, we now introduce a simplified version of the BiModal classification problem. Namely, rather

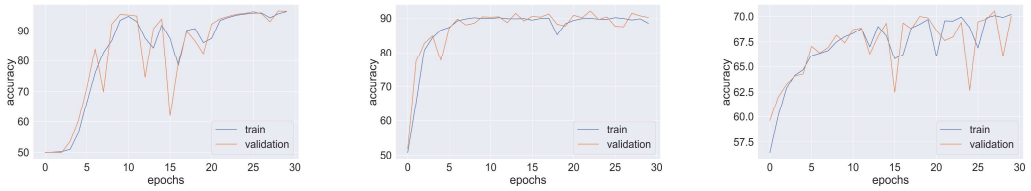
than identifying the generating models contained in the trajectory, The NN architecture's task is now to simply tell whether a transition occurred or not in a given trajectory. In order to do so, a small tweak must be added to the BiModal dataset in the form of an extra column indicating whether a transition occurred in the sample. The motivation to examine this simplified version of the problem is to further prove the claim that the CNN and FC classify trajectories based on unique anomalous events rather than based on the underlying dynamics of the random walks.

As can be seen in Fig. 5.3, all networks have the ability to distinguish between regular and BiModal random walks. With the CNN scoring 92% accuracy and the FC network scoring 71% accuracy on average on the validation set. These results show that both these networks are capable of learning to identify that an anomalous event has indeed occurred in the given sample.

The second reason for examining the transition recognition problem is to highlight the fact that detecting anomalies in sampled trajectories is a much less complex problem than the task of identifying the generating models contained within a BiModal trajectory. Therefore, we now show the performance of the BiGRU on the transition recognition dataset.

The BiGRU performed well with 96% accuracy on the validation set on average. The BiGRU performance on the transition task yields further insight into the cause for the 17% decrease in performance shown in Section 5.2.1. The first thing to take into account is the 4% error in recognizing the existence of a transition within the sample trajectory. The second thing to take into account is the 7% error in identifying single model trajectories given a small dataset ($5 \cdot 10^3$ samples, 100 steps each) shown in Chapter 4. The reason for the error for the small dataset is, as mentioned earlier, the amount of total sampled trajectories in the dataset is now divided between a large number of classification options (16). This leaves the networks with a small ratio of samples per label to learn from. Therefore, the worst-case scenario error of the BiGRU would be a multiplication of the two errors, yielding a maximum of 28% error.

However, we cannot make the same assumption when attempting to explain the discrepancy between the performance of the other two networks over the BiModal and regular datasets. Therefore, we turn to the same explanation used in Section 4.4 to explain the accuracy reduction over the long trajectories dataset. We once again assume that the CNN and FCN learn to identify anomalous events within the trajectory and base their classification on the distinctive features of the events rather than the distinctive statistics of these events. In this case, the networks can easily learn to tell the difference between a regular trajectory and a trajectory containing an anomalous event such as model transition but are completely blind to the order of participating models. In other words, the FCN and CNN can easily distinguish between a trajectory containing transition from CTRW to Brownian and a trajectory containing only Brownian motion, but cannot distinguish between the former and a trajectory containing transition from Brownian to CTRW due to the existence of the same anomalous events between the two. Furthermore, the transition points in every trajectory are independent of the two generating models contained in the trajectory. Therefore, the ability of the networks to identify transition points does not add any unique data that could be useful for the networks in differentiating between different trajectories.



(a) (b) (c)

Figure 5.3: The accuracy of the networks on the training and test datasets containing BiModal trajectories, as a function of the epochs. the dataset contained $4 \cdot 10^5$ walks with 100 steps each. Figure 5.3a illustrates training the BiGRU network. Figure 5.3b illustrates training the CNN. Figure 5.3c illustrates training the FC network.

	GRU	BiGRU	CNN	FC
Classify Models	79.5%	82.5%	32%	54%
Transition Recognition	89%	96%	92%	71%

Table 5.1: The performance of all four NN architectures over the two classification tasks.

5.4 Second Approach: Segment Classification

We now attempt to solve the problem of identifying the random walk models contained in a BiModal trajectory by using the second approach. Meaning we split the input trajectory into segments and use the NN to classify each segment individually. The motivation for using this approach is that we would like to utilize the GRU's state-of-the-art performance over the single model trajectory classification problem (Chapter 4) in order to solve the BiModal classification problem. Another reason for using the second approach is that we can train the NN on the single model dataset and then simply reuse the network for the BiModal identification problem. The main benefit in doing so lies in the reduced size of the training set required to train the NN. As mentioned earlier in the chapter, the amount of labels in the dataset scales exponentially with the number of random walk models analyzed (which in our case is 4). Therefore, in order to maintain the same ratio of samples per label, the number of samples in the training set also had to increase exponentially, which added a time and memory constraint to the training process. Overall, the process of solving the BiModal problem with the second approach includes:

1. Training a BiGRU model over the 100 step, $5 \cdot 10^5$ samples dataset (Chapter 2).
2. Taking trajectory samples from the BiModal dataset and splitting them into segments.
3. Feeding the first and last segment of each trajectory into the trained BiGRU and classifying the trajectory based on the classification of these two segments. Note that this action relies on the underlying assumption that there is at most one transition contained in the trajectory.

In order to reuse the trained BiGRU, a modified validation function is now introduced. Relying on the fact that there is one transition at any given trajectory, this new validation function splits a given trajectory into n segments. Then, the validation function generates

an output based on the model's classification of the first and last segment of the given trajectory. Ideally, the goal would be to have n individual segments that each contain samples from a unique random walk generating model. However, since the transition occurs at a random time step t, this assumption is highly improbable. Therefore, the goal would be to split the walk into as many segments as possible so that the samples arriving from different models would be disregarded by the NN as noise. n is a hyperparameter of this problem and would have to be chosen such that we minimize the interference within each trajectory while keeping each segment sufficiently large so that the NN will be able to create a proper prediction. Note that this hyperparameter n would have a strong dependence on the length of the trajectory T. Also note that the dataset contains trajectories with transitions occurring in the range of $[\frac{1}{6}T, \frac{5}{6}T]$. Therefore, in order to simulate experimental scenarios, we would like to make sure that some of the segments fed into the network during the validation process will contain a model transition. In order to make sure we get segments containing multiple models, the validation function splits each trajectory into $n = 4$ segments (in the published code repository, the user would be able to tweak the value of n freely).

However, naively applying this validation function yields a significant drop in accuracy. This aggressive reduction of accuracy is caused due to the network's training on the original dataset, containing only trajectories starting from the point $X_0 = 0$. This means the hidden states produced by the BiGRU by computing the first steps of the BiModal random walk are severely saturated. This saturation leads to the BiGRU wrongly classifying all segments that do not start from $X_0 = 0$ as Levy walks.

A possible fix for this issue would be to generate a dataset of trajectories with variable starting points, then train the BiGRU on that dataset in order to remove the starting point bias. However, since the length of the trajectory is a variable that could increase arbitrarily, the starting point of the final segment would have the same mean and variance as the random walks contained in the BiModal trajectory. This means that in the worst-case scenario, the analyzed trajectory is a random walk that transitions from an infinite mean process to a finite mean one (for example, Levy flight to Brownian). This type of transition occurring in the trajectory would still saturate the hidden states of the BiGRU, leading to incorrect classification. Even in the best-case scenario (no transition occurring in the BiModal trajectory), in order to effectively remove the starting point bias, the modified dataset would need to have a variance no smaller than the variance of the first step of the final segment $X_{\frac{n-1}{n}T}$. Since this variance is a function of time (as shown in the introduction of the MSD algorithm), this method would mean losing time independence which was the main selling point of the BiGRU.

Eventually, the method chosen to deal with the starting point bias is to subtract the first value of the segment from all values of the last trajectory segment. In a more mathematical sense:

$$X_{\frac{n-1}{n}T \to T} = X_{\frac{n-1}{n}T \to T} - X_{\frac{n-1}{n}T} \tag{5.1}$$

Where $X_{\frac{n-1}{n}T \to T}$ is the vector of locations making up the last segment of the trajectory.

In total, this modified validation method which utilized a BiGRU network that was trained on the single models dataset yields a 5 point improvement over the baseline results in the BiModal trajectory classification task, landing at a 87.5% accuracy on the validation set.

In order to gain deeper insights into these results, a confusion matrix of the specific classifications of the BiGRU over the validation set is offered (Fig. 5.4). As can be seen in the confusion matrix, the main issue contributing to the loss in classification effectiveness

is caused by trajectories containing FBM and Brownian at some stage. This phenomenon makes sense when taking into account that a large portion of the original architecture's loss stems from the difficulty distinguishing between FBM trajectories with Hurst exponent close to 0.5. In other words, some of the FBM trajectories are generated with Hurst exponents that make them indistinguishable from regular Brownian motion. Factoring in the reduced trajectory length caused by splitting the random walk (which means classifying over 30 step trajectories), and recalling the results from Chapter 4, where training over a 20 step trajectory yields 93% accuracy. We can therefore relate the reduced performance to the increased amount of classes containing these two models. A further explanation of the loss is the fact that some of the segments fed into the network are also comprised of more than one model, as discussed earlier in the section. We recall from Chapter 4 that as we extend our trajectory, the model's ability to classify it increases. Therefore, we test our new inference method with increasingly longer trajectories - $[240, 360, 480]$. As can be clearly seen in Table 5.2, the new validation function has raised the long-range accuracy significantly, with a peak of 94%. Which means we have gained back our model's ability to classify long walks.

steps	120	240	360	480
accuracy	87.5%	92.5%	93.7%	94%

Table 5.2: The accuracy of the BiGRU trained on the regular single model 100 step dataset and then tested on mixed model datasets with different walk lengths.

Before concluding this chapter, a scheme for further research is suggested in order to both presents a heuristic for estimating the optimal n segments, as well as to classify trajectories with an arbitrary amount of transitions. To do that, one can rely on the average confidence offered by the network. Recall that the way a prediction is made is by feeding the input to the network, producing a vector of likelihoods where each slot is the probability that the sample belongs to a specific class. Then, the class with the highest value of likelihood is chosen as the classification of the sample. In this work, the confidence of the network refers to the likelihood of the chosen classification. An extension of this concept would be to take the mean of the confidence of all segments of a trajectory, the averaged confidence. Therefore, in order to find the optimal number of segments n to split a trajectory, a possible method would be to split into variable amounts of segments and choose the n that produces the highest averaged confidence. Note that in this method, n would no longer have to be a hyperparameter of the network, but a variable that could be chosen for each individual trajectory. Also note that maximizing this score is essentially minimizing the probability that a segment contains multiple random walk generating models, which would in turn increase the accuracy of the BiGRU.

Another possible use case of the averaged confidence score suggested here would be in the formulation of a method to discover the number of transitions occurring in a given trajectory. In this case, rather than feeding only the first and last segment of the trajectory to the BiGRU, the averaged confidence would be calculated for all n segments of the trajectory. Then, after choosing the n that maximizes the averaged confidence, every consecutive segment that has the same classification would be treated as a single segment. After that, simply counting the number of segments created in the trajectory should yield the number of transitions in the trajectory as well as the underlying generating models

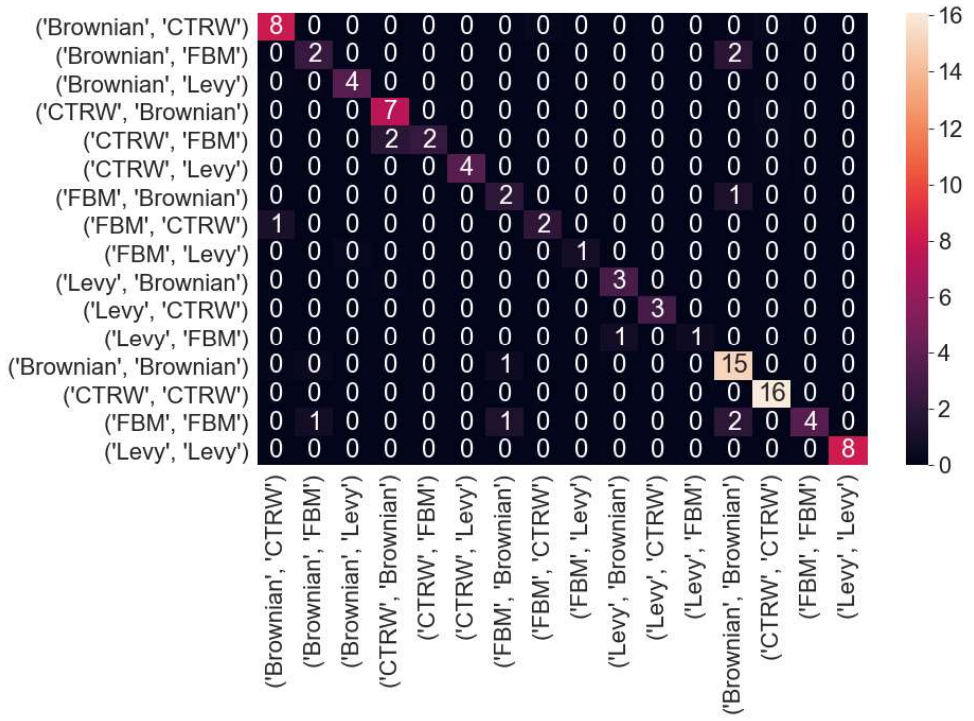


Figure 5.4: The confusion matrix of the BiGRU used to classify a validation dataset of mixed model random walks, with each trajectory containing 120 steps. each square represents the percentage of predictions of the given class. the rows of the matrix represent the ground truth, while the columns represent the network classification

contained in it.

In the next chapter, we present a summary of the contents of this work, as well as possible leads for extending it by utilizing state-of-the-art advancements in the field of machine learning.

Chapter 6

Conclusion And Further Discussion

This work offers a new approach for analyzing single trajectory samples of random walks by utilizing popular algorithms from the field of machine learning. The capabilities of three neural network paradigms were demonstrated on the problem of classifying the underlying random walk model of a single trajectory.

The problem of classifying random walks was divided into two individual tasks with increasing complexity. The first task was classifying trajectories created by single random walk generators. As mentioned, the goal was to reach an adequate classification accuracy using only a single sampled trajectory, meaning traditional methods that rely on an ensemble of samples such as the MSD are of no use. Furthermore, this classification accuracy was to be achieved using short trajectories, which would leave the TA-MSD useless as well. The second task was to classify the underlying models of a BiModal random walk, as well as the ordering of these models within the trajectory. Unlike the first problem, which was heavily researched and contains many possible approaches, as far as we know there is currently no other algorithm capable of analyzing trajectories containing multiple random walk models, which makes the NN approach all the more valuable.

In order to train the NNs, a dataset containing four distinct random walk models was created. Simulating the four random walks enabled various degrees of freedom during training such as: variable diffusion constants, variable Hurst exponents, and variable anomalous diffusion constants. Furthermore, the dataset generator allows the user to replace the probability density functions responsible for both the step length as well as the waiting times. This feature enables training the NNs on a multitude of different random walks, from Brownian motion with Laplace distributed step lengths, to CTRW with Gaussian distributed waiting times. As for the second task of classifying BiModal trajectories. The dataset generator is capable of automatically simulating trajectories with randomly distributed transition points, ranging within any segment chosen by the user (in this work the transition point chosen was uniformly distributed in range $[\frac{1}{6}, \frac{5}{6}]$) The dataset generator can be found in the attached GitHub repository and could be imported as a stand-alone module in other projects.

The first machine learning paradigm analyzed was the Fully Connected Neural Network (FCN). The FCN is the most traditional and well-known algorithm for solving machine learning problems. This slightly out-of-date method was proven to be not very useful when it comes to the task of classifying single sample trajectories. As shown in the previous chapters, out of all three NNs tested in this work, the FCN yielded the worst results on both classification tasks.

In order to improve the classification performance, the second paradigm was utilized,

the Convolutional Neural Network (CNN). The CNN learning algorithm is able to capture context, which greatly improved the NN's performance on the first classification task. However, as was proven by the second classification task, both the FCN and CNN learned to classify random walks based on anomalous events occurring in the random walk trajectory. Furthermore, both networks required constant length trajectories, which is a major constraint in experimental environments.

The third and final paradigm tried was the Recurrent Neural Network (RNN). The networks based on this paradigm were the Gated Recurrent Unit (GRU) and the Bi-directional GRU (BiGRU). As discussed in the previous chapters, these networks have both the flexibility to analyze random length trajectories, as well as the ability, to capture the underlying temporal correlations contained within the trajectory. Therefore, it was no surprise that the RNN based networks showed the best performance on both datasets. Furthermore, the RNN based models showed great improvements over the other CNN and FCN when it comes to analyzing long-ranged trajectories. With results from Chapters 4, 5 showing a direct positive correlation between the accuracy of the model and the length of the trajectory.

This thesis only scrapes the surface of the applications of neural networks in the field of analyzing different characteristics of random walks. First, one may leverage the neural network's ability for transfer learning - training an already-trained network with a custom dataset. This unique feature opens the possibility of developing a large open-source AI that can be used and enhanced by any researcher that comes across a new type of walk. Therefore, in order to allow for easy extension of this work, the entire code base for this work is open-sourced (including files of the trained networks), and an API would be provided in order to grant the ability to load and save trained networks for further training. Another lead for further research would be to incorporate new machine learning paradigms into the task of classifying random walks. The first example is the Transformer [72] network architecture which revolutionized the field of Natural Language Processing with the use of self-attention. Language modeling tasks are a subset of time series problems, with the major difference being a finite amount of variables that scales with the size of the dictionary. Therefore, multiple works in recent years have adapted the Transformer for time series classification [73], [74], which further hints at the usability of Transformers for analyzing physical stochastic processes.

Another possible enhancement to this work would be the Physics-Informed Neural Networks (PINNs) [75]. This approach replaces the generic loss functions commonly used with a loss function specifically tailored according to the dynamics of the data. In other words, the Partial Differential Equation can be solved directly by the neural network for the entire parameter space of the problem. An example for PINNs use in stochastic processes was recently demonstrated in a work applying this algorithm for weather and climate modeling [76].

Bibliography

[1] In Jae Myung. In: *Journal of Mathematical Psychology* 47.1 (2003), pp. 90–100.

[2] Sebastian Ruder. In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747.

[3] "Mean Squared Error". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 653–653. ISBN: 978-0-387-30164-8.

[4] J. Kiefer and J. Wolfowitz. In: *Ann. Math. Statist.* 23.3 (Sept. 1952), pp. 462–466.

[5] Abhishek Panigrahi et al. In: *CoRR* abs/1910.09626 (2019). arXiv: 1910.09626. URL: http://arxiv.org/abs/1910.09626.

[6] Umut Simsekli et al. In: (2020). arXiv: 2002.05685 [stat.ML].

[7] Grigorios Pavliotis. *The Langevin Equation*. Sept. 2014, pp. 181–233. ISBN: 978-1-4939-1322-0. DOI: 10.1007/978-1-4939-1323-7_6.

[8] Sunday Kayode, Akeem Ganiyu, and Adegoke Ajiboye. In: *OALib* 03 (Jan. 2016), pp. 1–15. DOI: 10.4236/oalib.1102247.

[9] Max Welling and Yee Whye Teh. In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML'11. Bellevue, Washington, USA: Omnipress, 2011, pp. 681–688. ISBN: 9781450306195.

[10] Sean Walton et al. In: *Chaos Solitons and Fractals* 44 (Sept. 2011), pp. 710–718. DOI: 10.1016/j.chaos.2011.06.004.

[11] Grant Foster. "Wavelets for period analysis of unevenly sampled time series". In: *The Astronomical Journal* 112 (1996), pp. 1709–1729.

[12] KK Gordon Lan, David M Reboussin, and David L DeMets. "Information and information fractions for design and sequential monitoring of clinical trials". In: *Communications in Statistics-Theory and Methods* 23.2 (1994), pp. 403–420.

[13] Brian Litt and Javier Echauz. "Prediction of epileptic seizures". In: *The Lancet Neurology* 1.1 (2002), pp. 22–30.

[14] Eli Barkai, Yuval Garini, and Ralf Metzler. "Strange kinetics of single molecules in living cells". In: *Physics Today* 65.8 (2012), pp. 29–35. DOI: 10.1063/PT.3.1677. eprint: https://doi.org/10.1063/PT.3.1677. URL: https://doi.org/10.1063/PT.3.1677.

[15] MN Chowdhury. In: *Journal of medicine* 17.5-6 (1986), pp. 373–396. ISSN: 0025-7850. URL: http://europepmc.org/abstract/MED/3295094.

[16] *ergodic hypothesis*. 2009. DOI: 10.1093/acref/9780199233991.013.1024. URL: https://www.oxfordreference.com/view/10.1093/acref/9780199233991.001.0001/acref-9780199233991-e-1024.

[17] Alex Sherstinsky. In: *Physica D: Nonlinear Phenomena* 404 (Mar. 2020), p. 132306. ISSN: 0167-2789. DOI: 10.1016/j.physd.2019.132306. URL: http://dx.doi.org/10.1016/j.physd.2019.132306.

[18] Albert Einstein. "Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen". In: *Annalen der physik* 4 (1905).

[19] Paul H Cootner. *The random character of stock market prices*. MIT press, 1967.

[20] Steven Roman. *Introduction to the mathematics of finance: from risk management to options pricing*. Springer Science & Business Media, 2004.

[21] Rep Kubo. "The fluctuation-dissipation theorem". In: *Reports on progress in physics* 29.1 (1966), p. 255.

[22] Don S. Lemons and Anthony Gythiel. In: *American Journal of Physics* 65.11 (1997), pp. 1079–1081. DOI: 10.1119/1.18725.

[23] "Laplace Distribution". In: *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, 2008, pp. 294–295. ISBN: 978-0-387-32833-1. DOI: 10.1007/978-0-387-32833-1_219. URL: https://doi.org/10.1007/978-0-387-32833-1_219.

[24] Elliott W. Montroll and George H. Weiss. In: *Journal of Mathematical Physics* 6.2 (1965), pp. 167–181. DOI: 10.1063/1.1704269. eprint: https://doi.org/10.1063/1.1704269. URL: https://doi.org/10.1063/1.1704269.

[25] Miljenko Huzak. "Characteristic Functions". In: *International Encyclopedia of Statistical Science*. Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 238–239. ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_636. URL: https://doi.org/10.1007/978-3-642-04898-2_636.

[26] Michael George Bulmer. *Principles of statistics*. Courier Corporation, 1979.

[27] I. Y. Wong et al. "Anomalous Diffusion Probes Microstructure Dynamics of Entangled F-Actin Networks". In: *Phys. Rev. Lett.* 92 (17 Apr. 2004), p. 178101.

[28] R Metzler. "JH jeon, AG Cherstvy and E. Barkai, Anomalous diffusion models and their properties: non-stationarity, non-ergodicity, and ageing at the centenary of single particle tracking". In: *Phys. Chem. Chem. Phys* 16 (2014), pp. 24128–24164.

[29] Benoit B. Mandelbrot and John W. Van Ness. "Fractional Brownian Motions, Fractional Noises and Applications". In: *SIAM Review* 10.4 (1968).

[30] Jan Beran. *Statistics for long-memory processes*. Vol. 61. CRC press, 1994.

[31] Georgiy Shevchenko. "Fractional Brownian motion in a nutshell". In: *arXiv preprint arXiv:1406.1956* (2014).

[32] Oliver C. Ibe. "2 - Basic Concepts in Stochastic Processes". In: *Markov Processes for Stochastic Modeling (Second Edition)*. Ed. by Oliver C. Ibe. Second Edition. Oxford: Elsevier, 2013, pp. 29–48. ISBN: 978-0-12-407795-9. DOI: https://doi.org/10.1016/B978-0-12-407795-9.00002-5. URL: https://www.sciencedirect.com/science/article/pii/B9780124077959000025.

[33] Benoit Mandelbrot. “How Long Is the Coast of Britain? Statistical Self-Similarity and Fractional Dimension”. In: *Science* 156.3775 (1967), pp. 636–638. ISSN: 0036-8075. DOI: 10.1126/science.156.3775.636. eprint: https://science.sciencemag.org/content/156/3775/636.full.pdf. URL: https://science.sciencemag.org/content/156/3775/636.

[34] “Simulation of fBm”. In: *Fractional Brownian Motion*. John Wiley Sons, Ltd, 2019. Chap. 3, pp. 239–249.

[35] F. Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain”. In: *Psychological Review* (1958), pp. 65–386.

[36] LB Almeida. *Handbook of Neural Computation, chapter C1. 2 Multilayer perceptrons, pages C1. 2: 1–C1. 2: 30*. 1997.

[37] Abien Fred Agarap. In: *CoRR* abs/1803.08375 (2018). arXiv: 1803.08375. URL: http://arxiv.org/abs/1803.08375.

[38] Sakshi Indolia et al. In: *Procedia Computer Science* 132 (2018). International Conference on Computational Intelligence and Data Science, pp. 679–688. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2018.05.069. URL: http://www.sciencedirect.com/science/article/pii/S1877050918308019.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[40] Jeff Donahue et al. “Decaf: A deep convolutional activation feature for generic visual recognition”. In: *International conference on machine learning*. PMLR. 2014, pp. 647–655.

[41] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. “Deep neural networks for object detection”. In: (2013).

[42] D Timoshenko and Valery Grishkin. “Composite face detection method for automatic moderation of user avatars”. In: *Computer Science and Information Technologies (CSIT’13)* (2013).

[43] Tara N Sainath et al. “Improvements to deep convolutional neural networks for LVCSR”. In: *2013 IEEE workshop on automatic speech recognition and understanding*. IEEE. 2013, pp. 315–320.

[44] Ayşegül Uçar. “Deep Convolutional Neural Networks for facial expression recognition”. In: *2017 IEEE International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*. IEEE. 2017, pp. 371–375.

[45] Emmanuel Maggiori et al. “Convolutional neural networks for large-scale remote-sensing image classification”. In: *IEEE Transactions on Geoscience and Remote Sensing* 55.2 (2016), pp. 645–657.

[46] Livnat Cohen. *diffusion coefficient of short transport in disordered media: the deep learning approach*. 2020.

[47] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[48] Tsung-Hsien Wen et al. “Semantically conditioned lstm-based natural language generation for spoken dialogue systems”. In: *arXiv preprint arXiv:1508.01745* (2015).

[49] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[50] Wootaek Lim, Daeyoung Jang, and Taejin Lee. “Speech emotion recognition using convolutional and recurrent neural networks”. In: *2016 Asia-Pacific signal and information processing association annual summit and conference (APSIPA)*. IEEE. 2016, pp. 1–4.

[51] Piji Li et al. “Deep recurrent generative decoder for abstractive text summarization”. In: *arXiv preprint arXiv:1708.00625* (2017).

[52] Pankaj Malhotra et al. “Long short term memory networks for anomaly detection in time series”. In: *Proceedings*. Vol. 89. 2015, pp. 89–94.

[53] Adrian Taylor, Sylvain Leblanc, and Nathalie Japkowicz. “Anomaly detection in automobile control network data with long short-term memory networks”. In: *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2016, pp. 130–139.

[54] C Lee Giles, Steve Lawrence, and Ah Chung Tsoi. “Noisy time series prediction using recurrent neural networks and grammatical inference”. In: *Machine learning* 44.1 (2001), pp. 161–183.

[55] Eugen Diaconescu. “The use of NARX neural networks to predict chaotic time series”. In: *Wseas Transactions on computer research* 3.3 (2008), pp. 182–191.

[56] Min Han et al. “Prediction of chaotic time series based on the recurrent predictor neural network”. In: *IEEE transactions on signal processing* 52.12 (2004), pp. 3409–3416.

[57] Wei Bao, Jun Yue, and Yulei Rao. “A deep learning framework for financial time series using stacked autoencoders and long-short term memory”. In: *PloS one* 12.7 (2017), e0180944.

[58] Zhengping Che et al. “Recurrent neural networks for multivariate time series with missing values”. In: *Scientific reports* 8.1 (2018), pp. 1–12.

[59] Fazle Karim et al. “LSTM fully convolutional networks for time series classification”. In: *IEEE access* 6 (2017), pp. 1662–1669.

[60] Michael Hüsken and Peter Stagge. “Recurrent neural networks for time series classification”. In: *Neurocomputing* 50 (2003), pp. 223–235.

[61] Christopher Olah. “Understanding lstm networks”. In: (2015).

[62] Klaus Greff et al. “LSTM: A search space odyssey”. In: *IEEE transactions on neural networks and learning systems* 28.10 (2016), pp. 2222–2232.

[63] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[64] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[65] Rahul Dey and Fathi M Salem. “Gate-variants of gated recurrent unit (GRU) neural networks”. In: *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*. IEEE. 2017, pp. 1597–1600.

[66] Diederik Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *International Conference on Learning Representations* (Dec. 2014).

[67] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[68] Matthew D Zeiler. “Adadelta: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701* (2012).

[69] Nicolas E Humphries et al. “Foraging success of biological Lévy flights recorded in situ”. In: *Proceedings of the National Academy of Sciences* 109.19 (2012), pp. 7169–7174.

[70] C-C Lo et al. “Dynamics of sleep-wake transitions during sleep”. In: *EPL (Europhysics Letters)* 57.5 (2002), p. 625.

[71] Htet Lynn, Sung Pan, and Pankoo Kim. “A Deep Bidirectional GRU Network Model for Biometric Electrocardiogram Classification Based on Recurrent Neural Networks”. In: *IEEE Access* PP (Sept. 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2939947.

[72] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[73] Neo Wu et al. “Deep transformer models for time series forecasting: The influenza prevalence case”. In: *arXiv preprint arXiv:2001.08317* (2020).

[74] George Zerveas et al. “A transformer-based framework for multivariate time series representation learning”. In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2021, pp. 2114–2124.

[75] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707.

[76] K Kashinath et al. “Physics-informed machine learning: case studies for weather and climate modelling”. In: *Philosophical Transactions of the Royal Society A* 379.2194 (2021), p. 20200093.

[77] William H. Press et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. USA: Cambridge University Press, 2007. ISBN: 0521880688.

[78] “Cholesky factorization”. English. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 1.2 (Sept. 2009), pp. 251–254. ISSN: 1939-5108.

Appendix A

Simulating Noise Distributions

A.1 Power-Law Distribution

We will now describe the method used in Chapter 2.2 to transform a Uniform distribution $U(0,1)$ into the Power-Law distribution $\psi(\tau) = A_\alpha \tau^\alpha$, where $\alpha > 1$ for the sake of normalization (adapted from [77]). Given a uniformly distributed random variable x, we take some prescribed function of it $y(x)$. The probability distribution of y, denoted $p(y)dy$, is simply determined by the fundamental transformation law of probabilities, which is simply

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \tag{A.1}$$

Which in our case means

$$p(y) = A_\alpha y^{1-\alpha}, \qquad p(x) = U(0,1) \tag{A.2}$$

Plugging our distributions into (A.1), and solving the consequent Ordinary Differential Equation

$$1\frac{dx}{dy} = A_\alpha y^{-\alpha} \tag{A.3}$$

$$\int 1dx = \int A_\alpha y^{-\alpha} dy \quad \Rightarrow \quad x = \frac{A_\alpha}{1-\alpha} y^{1-\alpha} + C \tag{A.4}$$

And were left with the following transformation

$$y = \left[(x-C)\frac{1-\alpha}{A_\alpha}\right]^{\frac{1}{1-\alpha}} \qquad \alpha > 1 \tag{A.5}$$

We will now use the demand for normalization to find A_α

$$\int_{y_0}^{\infty} A_\alpha y^{-\alpha} = 1 \tag{A.6}$$

$$\Rightarrow \quad -\frac{A_\alpha}{1-\alpha} y_0^{1-\alpha} = 1 \tag{A.7}$$

$$\Rightarrow \quad A_\alpha = (\alpha-1) y_0^{\alpha-1} \tag{A.8}$$

Plugging (A.8) into (A.5) yields

$$y = y_0[C-x]^{\frac{1}{1-\alpha}} \tag{A.9}$$

We note that as $x \to 1$ our transformed variable must diverge as well $y \to \infty$, and when $x = 0$ $y = y_0$. We'll find a constant C that satisfies both conditions

$$y_0[C-0]^{\frac{1}{1-\alpha}} = y_0 \quad \Rightarrow \quad C = 1 \tag{A.10}$$

Therefor, our complete transformation from Uniformly distributed variables to Power-Law is

$$y = [1-x]^{\frac{1}{1-\alpha}} y_0 \tag{A.11}$$

A.2 Fractional Gaussian Noise

The method we used in order to simulate an exact Fractional Gaussian Noise (FGN) is the Cholesky decomposition [34] of the FGN's covariance matrix. First, we will define the FGN in terms of the Fractional Brownian Motion (FBM). We note that the trajectory of the FBM in the interval $[0,T]$

$$B^H = \{B_t^H, t \in [0,T]\} \tag{A.12}$$

can only be simulated in discrete time. Therefor, we use an equidistant partition of our measurement interval $[0,T]$ to N steps and simulate the trajectory as

$$B^H_{T/N}, B^H_{2T/N}, ..., B^H_{T(N-1)/N}, B^H_T \tag{A.13}$$

Since this trajectory is a centered Gaussian vector with a defined covariance matrix, it can be simulated as a linear transformation of a standard Gaussian vector. Due to the self-similarity property of the FBM (shown in (2.3)) it will be enough for us to simulate the walk as $B^H_1, B^H_2, ..., B^H_N$ and then simply multiply each value by $(T/N)^H$. Since the increments of FBM are stationary ((2.3)), we can define each sample of FBM as a cumulative sum of FGN samples ζ

$$\zeta^H_n = B^H_n - B^H_{n-1} \tag{A.14}$$

$$B^H_N = \sum_{n=0}^{N} \zeta^H_n \tag{A.15}$$

where $\zeta^H = (\zeta^H_1, ..., \zeta^H_N)^T$ is a centered Gaussian vector with a covariance matrix (based on (2.20))

$$\Gamma_N = \begin{pmatrix} 1 & \rho(1) & \rho(2) & ... & \rho(N-1) \\ \rho(1) & 1 & \rho(1) & ... & \rho(N-2) \\ \rho(2) & \rho(1) & 1 & ... & \rho(N-3) \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ \rho(N-2) & \rho(N-3) & \rho(N-4) & ... & \rho(1) \\ \rho(N-1) & \rho(N-2) & \rho(N-3) & ... & 1 \end{pmatrix} \tag{A.16}$$

where $\rho(n) = \mathbb{E}[\zeta^H_1 \zeta^H_{n+1}]$ is the covariance between the first and n samples. Therefor, we can simulate FGN by applying the following transformation

$$\zeta = LZ \tag{A.17}$$

where Z is a standard Gaussian vector, and L is a linear transformation matrix such that $LL^T = \Gamma$. In order to find the wanted transformation, we must first find the "square root" of the covariance matrix Γ. Since Γ is a symmetric, positive, real definite matrix, we are able to decompose it into a multiplication of a lower triangular matrix L, $\big($where $L = (l_{ij})_{i,j=1,\dots,N}$ and $l_{ij} = 0$ for $i < j\big)$ and its transposed pair L^T, using the Cholesky factorization method [78]. Let us first rewrite the condition $LL^T = \Gamma$ in a coordinate-wise form:

$$\sum_{k=1}^{j} l_{ik} l_{jk} = \rho(i-j), \qquad 1 \le j \le i \le N \tag{A.18}$$

and the elements of $l_{ij}, j \le i$ can be calculated recursively. After some calculations, it is easy to see that for $n \ge 2$, the elements of the $(n+1)$ row of our transformation matrix L are determined by:

$$l_{n+1,1} = \rho(n), \tag{A.19}$$

$$l_{n+1,j} = l_{jj}^{-1}\Big(\rho(n+1-j) - \sum_{k=1}^{j-1} l_{n+1,k} l_{jk}\Big), \qquad 2 \le j \le n, \tag{A.20}$$

$$l_{n+1,n+1} = \sqrt{1 - \sum_{k=1}^{n} l_{n+1,k}^2} \tag{A.21}$$

Appendix B

Wide Networks For Long Time Series

In Chapter 4, we exhibited a reduction in the performance of the Fully Connected (FC) network and the Convolutional Neural Network (CNN) when attempting to identify the random walk's generating model based on an extremely long trajectory of 1000 steps. A possible explanation for this accuracy decrease could be that the network architecture used was not deep or wide enough to sufficiently describe long trajectory random walks. Therefore, we attempted to train wider and deeper variations of both networks on the long trajectory dataset.

B.1 Wide Fully Connected Neural Network

The first network architecture tested was the wide FC network, constructed of several layers of neurons with a ReLU activation function. The layers are as follows:

- A layer the size of the trajectory (1000 neurons)
- A layer of size 700 neurons
- A layer of size 500 neurons
- A layer of size 300 neurons
- A layer of size 100 neurons
- A layer of size 30 neurons
- An output layer with log softmax activation function

Since the network was both marginally deeper and wider, we expected it to overfit the training set. However, the performance of the network over the long trajectory dataset remained the same as the one exhibited by the smaller network. As can be seen in Fig. B.1a, the network yielded an average accuracy of 62% over the test set.

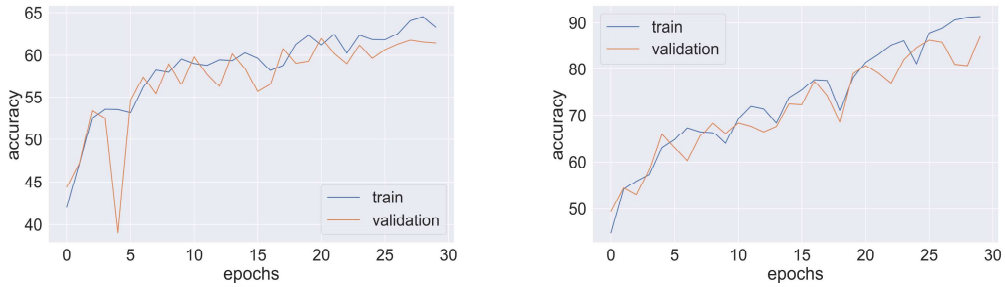
B.2 Wide Convolutional Neural Network

The second network architecture tested was the wide CNN, where the tweaks made were both to the number of kernel matrices were created for each filter type, as well as the depth of the FC layers that follow the convolutions. As discussed in Chapter 4 , the CNN

contained three types of convolution kernel matrices with a dimension of $(2,1)$ called filters. The kernels iterated over the nearest, next-nearest, next-next-nearest neighbors. But unlike the original CNN, which had 16 matrices for each filter type, the new architecture contained 64 kernels for each filter. Furthermore, the FC network that accepted the output of the convolutions was also widened and deepened as follows:

- A layer of size $(3 \cdot \text{steps} - 6) \cdot 64$
- A layer of size 200
- A layer of size 100
- A layer of size 100
- An output layer with log softmax activation function

The network yielded and averaged performance of 87% over the test set, which was not a large improvement margin over the original network (Fig. B.1b).
We thus conclude that a basic architectural tweak is not an acceptable solution, and the entire change machine learning model was indeed necessary.



(a) (b)

Figure B.1: The accuracy of the wide CNN and wide FC network on the long trajectory training and test datasets as a function of the epochs. Figure B.1a illustrates the case of training the wide FC network. Figure B.1b illustrates the case of training the wide CNN

Appendix C

Using Random Walks as Optimization Methods

As discussed in chapter 1, there is a deep relation between stochastic processes and the process of machine learning. Here we attempted to exploit this relation to come up with a new optimization method for training neural networks that will be inspired by different stochastic processes. The general approach to this problem is to take three aspects of the process into account:

The first is distribution jumps - where we will take an optimization process (SGD for example), and at constant intervals, add random noise taken from some distribution (exponential for example) to the gradient of the loss.

The second aspect is variable time interval – where will take the interval between optimization steps from some distribution.

The third aspect is memory-based jumps – where the jumps will depend on the current and previous states of the system. The first method will be to take the mean of the previous gradients over a given interval and add it to the current one. The second method will be to take the mean of the previous loss values and add it to the current gradient.

C.1 Architecture

We created three cases of networks that will be trained to identify handwritten numbers with examples taken from the MNIST database. The architectures in the three cases were identical: the first layer takes a photo of 28*28 pixels as a column vector and uses a ReLU activation function [37] on the input. The first layer is fully connected to a 64-neuron second layer. The second layer also uses ReLU and is fully connected to a 64-neuron layer. The final layer applies a linear function on the input and has a ten neuron output, on which it applies a log softmax function. We set the amount of epochs to be 10, and with each complete iteration over the entire dataset the learning rate is set to decay polynomially with each epoch t like $\alpha_t = a(b+t)^{\gamma}$, $\gamma \in (0.5, 1]$. In our case, we chose $a = 1, b = 100, \gamma = 0.8$

C.2 Regular diffusion as Optimization

The first optimization method uses the first aspect, spatial jumps. With every optimization step, in addition to taking a step in the direction of the gradient of the loss function, a noise

is added to the optimization step. The noise is applied individually to each edge of the network and is taken from an exponential distribution with the scale given by the learning rate at time t: $f(x) = s\alpha_t e^{-x}$. We trained three networks concurrently with different scale multipliers: $s = [1, 2, 10]$ and different jump intervals: $[50, 100, 200]$. We can immediately

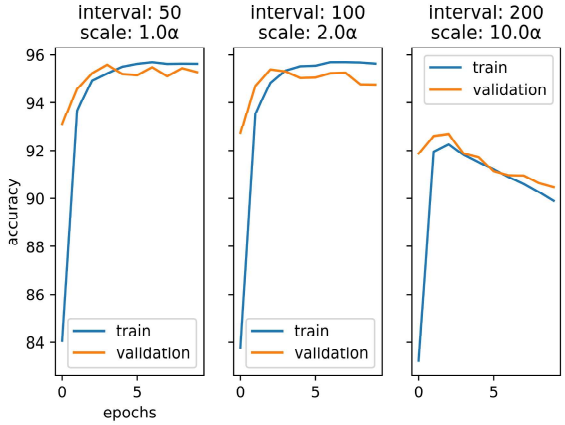


Figure C.1: The accuracy of the network trained by the exponential jumps optimizer.

see from figure C.1 that the exponential jumps optimizer has a tendency to overfit the data very quickly (at around epoch 4) which is counter-intuitive considering that the jumps are supposed to help the network avoid local minima.

C.3 Memory based Optimization

This idea is similar to the momentum learning algorithm. We take as noise the mean value of all the optimization steps made in the current interval. Our results are fairly similar to

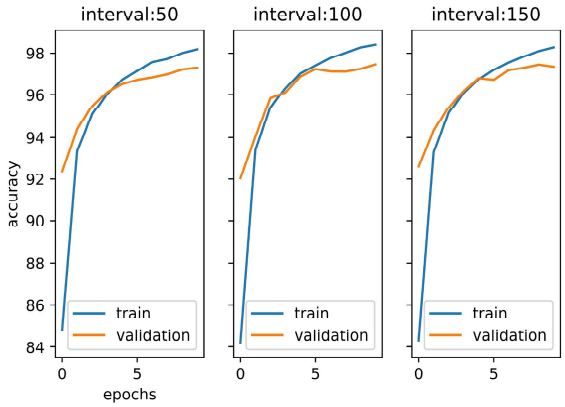


Figure C.2: The accuracy of the network trained by the mean gradient optimizer.

those of a regular SGD (Fig. C.2) which might simply suggest that the minima of our loss function is not very complicated. One might see a difference when training the model on more complex datasets.

C.4 CTRW as Optimization

The final optimization method tried uses the aspect of non-constant jump intervals. This method is similar to the continuous-time random walk [24], meaning the size of the jump is constant but the interval between jumps is taken from some random distribution. we trained the network with different hop sizes, all proportional to the learning rate. After fine-tuning the hyperparameters, we concluded that using a decaying learning rate with $b = 500$ achieves the best results with this kind of optimizer. It seems like due to the small

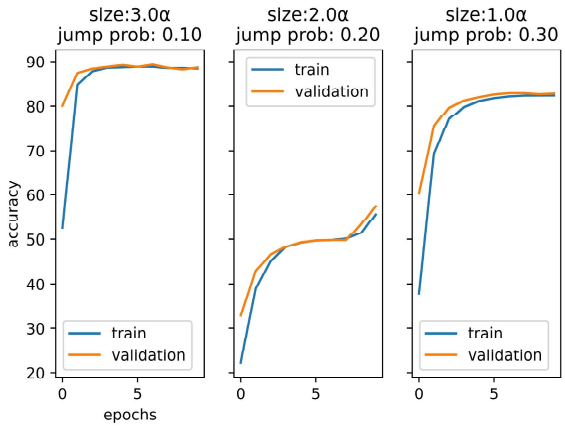


Figure C.3: The accuracy of the network trained by the variable time interval optimizer.

learning rate, the network has a harder time avoiding local minima, but gains the ability to escape local minima eventually due to the noise. However, when training the network for 100 epochs we see that avoiding some local minima means diverging loss. In other words, it seems like the addition of random noise has changed the shape of the potential.

תקציר

בשנים האחרונות חלה עלייה משמעותית בהטמעת למידת מכונה ובינה מלאכותית בתחומים חדשים. זאת בשל עלייה בכוח החישוב בנוסף לעלייה בכמות המידע הזמין לניתוח. בעבודה זו נטמיע למידת מכונה בתחום הפיזיקה הסטטיסטית על ידי ניצול של מספר ארכיטקטורות של רשתות נוירונים. המטרה היא לזהות את התהליכים הסטוכסטיים הקיימים במערכת מתוך דגימה של הלך אקראי בודד. הרשתות שפותחו בעבודה זו מראות שיפור משמעותי בביצועים בהשוואה לשיטות קלאסיות לניתוח הלכים אקראיים. בנוסף, אנו משתמשים ברשתות המאומנות על מנת לסווג תהליכים סטוכסטיים במסלולים שנוצרו על ידי מספר מודלים של הלכים אקראיים.

עבודה זו נעשתה בהדרכתו של ד"ר סטאס בורוב מן המחלקה
לפיזיקה של אוניברסיטת בר-אילן

אוניברסיטת בר-אילן

גישות חדשות לאנליזה של הלכים אקראיים באמצעות למידה עמוקה

דוד פלג

עבודה זו מוגשת כחלק מהדרישות לשם קבלת תואר מוסמך במחלקה לפיסיקה של
אוניברסיטת בר-אילן

רמת גן

תשפ״ב