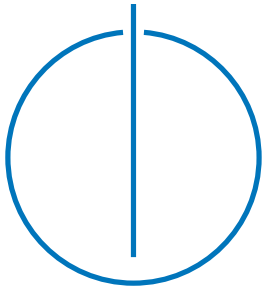
DEPARTMENT OF INFORMATICS
(Guided Research in Informatics)

Technische Universität München

DAAD WISE Internship Report

Learning Aerosol Dynamics Directly From Image Data

Siddharth Sanjay Gandhi



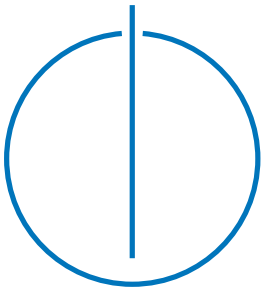
DEPARTMENT OF INFORMATICS
(Guided Research in Informatics)

Technische Universität München

DAAD WISE Internship Report

Learning Aerosol Dynamics Directly From Image Data

Author:	Siddharth Sanjay Gandhi
Advisor:	Dr. Felix Dietrich
Examiner:	Dr. Felix Dietrich
Submission Date:	August 15th, 2022



Abstract

Understanding aerosol dynamics is important for several applications such as understanding the spread of infectious diseases or how fuel injection works in a combustion engines. We generally operate aerosol modelling at coarse macroscopic scale, as it is very challenging to do a microscopic scale. Partial and Stochastic Differentials Equations can be used to model aerosol dynamics, however they are quite challenging to solve numerically, especially for large data sets. We have to consider interactions such as gas-fluid and aerosol-air interactions to model the dynamics properly. Hence, in this project, we aim to learn aerosol dynamics directly from the inputted image data. The images are first segmented using the U-Net architecture, then the aerosol dynamics (cloud growth) are modelled using Stochastic Differential Equations (SDEs), which are learnt using neural networks. As there is a lack of labelled image data, we will use crude simulations in Blender to generate synthetic data for training the models. We were able to train neural networks to learn the trajectories of the cloud sizes over time. We then looked at various ways to get a better fit and to address the imbalanced regression problem that is inherently present in the dataset. We looked at SMOGN and KDE based reweighing to solve the imbalance problem and compare the various results for different configurations.

Contents

1 Introduction

Aerosols are defined as "any colloidal suspension of solid or liquid particles in gas" [1].This can refer to many things, such as, the contents of a perfume or a spray paint, which are artificial, to more natural occurrences such as fog, mist and smoke. Understanding aerosol dynamics is important because it has uses in understanding how air flows in jet engines or how fuel injections work in combustion engines. Airborne diseases can also spread via aerosol particles in the air when a person coughs or sneezes. They also make up a significant portion of our atmosphere's particles, and can have major effect on the weather or climate of a region, depending on the properties of aerosol particles and their concentrations.

In 2020, the Severe Acute Respiratory Syndrome CoronaVirus 2 (SARS-CoV-2), a novel strain of coronavirus causing respiratory disease with high chances of severe illness and death, resulted in a global pandemic, which is still on-going as of today. The primary transmission mode of SARS-CoV-2 virus was through aerosol transmission, specifically through coughing and sneezing. Even breathing and speaking normally released respiratory aerosol particles, carrying the virus. These particles could remain suspended in the air for several hours, before settling on a surface, where it can still be transmitted via surface and contact transmission.

A good example of understanding aerosol dynamics came from testing various preventive measures to stop the spread of COVID-19 disease. In one experiment, the efficacy of face-shields was tested with humans exhaling white smoke in an otherwise dark room, and observing the growth of the white aerosol cloud. It was found that the aerosol particles rapidly spread around the room and thus, face-shield type designs were considered futile for containing the aerosol particles after coughing [2].

Figure 1.1: Experiments of humans breathing smoke against a dark background (Simulated)

[1]Source: https://en.wikipedia.org/wiki/Aerosol

[2]Prof. Christian Schwarzbauer's Study: https://www.br.de/nachrichten/wissen/pilotstudie-umstrittene-klarsichtmaske-bietet-keinen-schutz,SIZ0Tcp

Hence, modelling aerosol growth can allow us to understand how and where the particles will flow and what factors can influence them (such as emission source, surrounding air velocity, turbulence, particle size, etc). Running such simulations can allow us to better understand viral exposure in indoor spaces such as a classroom or office [10] and how ventilation may help or how aerosol particles interact in the atmosphere with other particles.

Now, there is a lack of good labelled training data for breathing experiments such as Prof. Schwarzbauer's study. So, in order to train the model with enough data, we have to do crude simulations using a 3D rendering software like Blender. These images will be quite coarse and it is not possible to observe them at the individual, microscopic scale. Hence, we will be relying on the growth of the volume of the aerosol cloud, over time.

This aerosol cloud growth is considered a non-linear discrete time sequence. These can be modelled in many ways, namely with ordinary, stochastic or partial differential equations [2], where we can take into account many initial factors such as turbulence, gravity, pose, etc. However, as there are elements of stochasticity (such as turbulence), we will be trying to model them with SDEs. Generally, to solve these equations numerically would take a long time and much computation. However there are techniques available to learning coarse Stochastic Differential Equations (SDEs) directly from fine grained particle simulations[3]. Hence, it is a natural progression to attempt to learn the dynamics of cloud growth using SDEs with neural networks, which is what we'll look at in this project: Given a series of images like Figure 1.1 , can we model the cloud growth?

My contributions are as follows:

1. Develop a script for automating the Blender Simulations of any amount of image data, with different initial parameters. We can input random *Strength* values (for the *Turbulence* object in the Blender) and the script will automatically generate the data and organize it in folders.

2. Replicate Rohan Saxena's work of the U-Net based Image Segmentation model [8], which can separate the aerosol cloud (subject) from the rest of the image (background). We then train it on the newly generated training data.

3. Use the work of Dietrich *et al.* [3] to train a neural network to learn drift and diffusivity of the SDEs over time (directly from data). This was done for both parameterized and non-parameterized cases. We then passed the *Cloud Size* data over time for all the persons and simulations to learn the SDE underlying the cloud growth.

4. Optimize the SDE model with techniques like Hyperparameter Optimization with the Keras Tuner library to get the model with best-fit.

To give a brief outline of what follows: in Section 2, we will take a look at the previous work on this topic, the background of the concepts involved and the popular techniques used. In section 3, we can see the details of our model, the project pipeline from the input images to the trained SDE model and also the results obtained. Finally, in Section 4, we will end with the conclusion and future scope of this topic.

2 State of the Art

2.1 Previous Work

Rohan Saxena's work on this topic [8] was successful at training a U-Net based image segmentation model, capable of separating the cloud or white part of the image from the rest. An example of the same is presented below:

Figure 2.1: U-Net Model Predictions Overlaid on top of original input images.

Figure 2.1 (from Rohan's Report) shows a thin red outline, which is the U-Net model's prediction of the region where the aerosol cloud is located.

He was able to achieve this by first setting up synthetic simulations in Blender, for 11 people in 7 different scenarios, with varying initial conditions. For example, the turbulence may have varying *Strength* parameter, which can change how the aerosol cloud will grow over time (which is what we will be attempting to learn in this paper). Then he used Convolutional Neural Networks in the form of a U-Net architecture (very good for image segmentation tasks), along with the dataset of images and masks generated by blender to train the model for our specific segmentation task. He also experimented training the model with different configurations during training, and tried to find the model which generalizes the best, i.e. which segments regardless of the person's color, or orientation of image.

2.2 Modelling aerosol dynamics to estimate the risk of viral exposure

As we saw in Section 1 , studying aerosol dynamics is an important problem to understand airborne transmission of diseases. This can help measure the risk of exposure to viruses, by modelling the path each viral particle can take. Vuorinen *et al.* [10] carried out various

simulations in different configurations such as coughing when in a closed room with a many walls in it or when in a supermarket walking at regular speed. They determined that how fast an aerosol cloud becomes diluted, depends largely on turbulence intensity and the size of the room. Sukrant Dhavan [2] also used a comprehensive model to monitor aerosol dynamics to come to several conclusions about how the risk of infection depends on the size of droplets, ventilation in the area, physical distance from others, and mask usage. Jones *et al.* [4] uses an aerosol model to estimate the uncertainty to viral exposure in a classroom and office setting and states the importance of good ventilation to minimize viral exposure from aerosols.

2.3 Image Segmentation

Image segmentation is the splitting or segmentation of the image into various groups, where each group is a bunch of closely related pixels. This relationship can be defined by various properties such as color, texture, etc. It finds the most use in medicine, where medical images can be segmented to identify malignant cells from benign ones.

2.3.1 Convolutional Neural Networks

There are a variety of techniques available for image segmentation, and neural networks are quite popular. Even in neural networks, we can have different types of architectures; the most common being Artificial Neural Networks (ANNs) or Feed Forward Neural Networks. However, they are not suitable for image segmentation tasks because:

1. They are not very efficient at identifying features or commonalities between groups of pixels.

2. They are unable to account for small movements in positions of the pixels (so they over-fit the training data in a way).

3. They require large amounts of training data and take a long time to train, while being computationally expensive.

Instead, another form of neural networks called Convolutional Neural Networks (CNNs) are used [5]. CNNs use a mix of perceptrons (ANNs) and convolution layers, which create feature maps to extract high level features from a portion of the image. This provides a more structured approach to the segmentation task with each layer responsible for different 'tasks'. For example, one layer may be grouping all white pixels, while another may be looking for eyes and noses. Thus, they are good at finding similarities between a group of pixels. Due to this, they can automate the task of feature extraction making training much easier as compared to previous models where feature extraction was manual, and they can even tolerate small shifts in the input images.

2.3.2 The U-Net Architecture

For image segmentation tasks, a very popular and precise convolutional neural net architecture is the U-Net architecture [7]. It is widely used in biomedical image segmentation tasks.

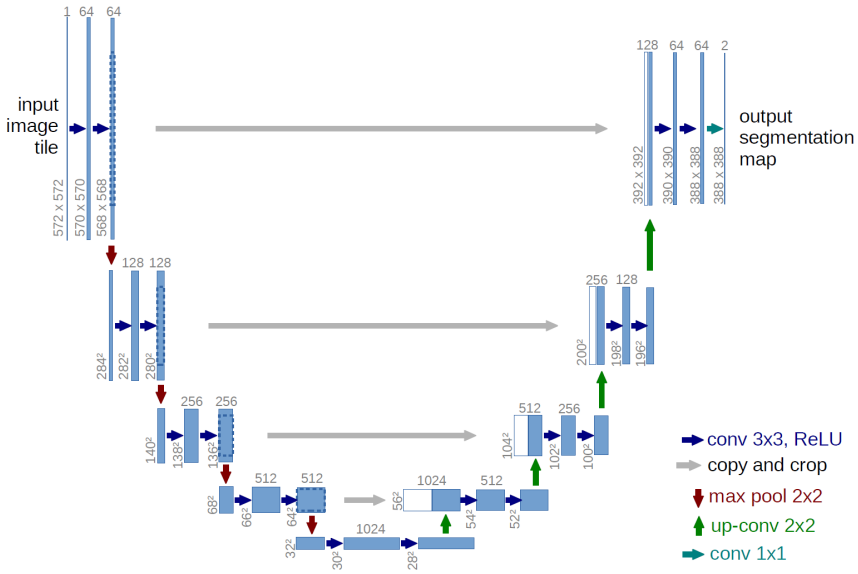


Figure 2.2: U-Net Architecture for Biomedical Image Segmentation [7]

This architecture, as demonstrated in Figurefig:unet-arch, has 2 parts, the contracting (encoder) part on the left and the expansive (decoder) part on the right.

The encoder part is a classical fully convolutional neural network, which extracts high level features from the input image to create a *feature map*. However, it drastically reduces the resolution of the image, and for image segmentation tasks, we would like to have the segmented image be the same resolution as the input image. So we add a decoder part to the end of the encoder which up-samples (opposite of convolution operation) the image repeatedly, to restore it's original resolution.

Each step level is a double convolution with size $3*3$ (kernel size). The padding same as input so as to maintain the image size. Moving down the encoder, a $2*2$ max-pooling layer with a stride of 2 is used, meaning we replace a $2*2$ portion of the image with it's maximum value. And then we slide over 2 positions and then repeat the process. During the up-sampling, we just reverse what we did in the down-sampling step. However, at each up-sampling step, we also concatenate the feature map from the encoder step of same level, which helps give localization information for semantic segmentation. Finally, to avoid over-fitting to the input data, we drop 10% of the pixels randomly at each convolution step.

All of this allows U-Net to perform much better than the previous state of the art meth-

ods, and it is efficient enough that it can do so with a small set of labelled data.

2.4 Learning Stochastic Differential Equations

Stochastic Differential Equations are used to model stochastic processes which involve random variables. They are used to describe phenomenons like stock price fluctuations or physical systems which are subject to much randomness.

Traditionally, ordinary differential equations are of the form

$$dx_t = f(x_t)dt, \tag{2.1}$$

given some initial condition $x_0 = c$ (as it is a first order derivative). Such equations are deterministic, meaning they can have only one possible value of x at a time t. However, SDEs are of the form

$$dx_t = f(x_t)dt + \sigma_t dW_t. \tag{2.2}$$

Here f is the deterministic term (called *drift*) while the σ is the stochastic or noise term (also called *diffusivity*). The noise term W is white noise, where W is a Wiener Process (standard Brownian Motion) with $\Delta W(t_i) = W(t_{i+1}) - W(t_i)$. It is normally distributed, with mean = 0 and standard deviation = $\sqrt{\Delta t_i}$ (as each increment will be a normal distribution, the variances will be linear, meaning it is added at each time step and thus, the std. deviation is the square root of time interval). It is also independent of it's previous increments, meaning it only depends on the current term and no other.

2.4.1 The Euler-Maruyama Scheme:

Now upon integrating Equation 2.2 , we get

$$\int_{t_i}^{t_{i+1}} dx_t = x_{t+1} - x_t = \int_{t_i}^{t_{i+1}} f(x_t)dt + \int_{t_i}^{t_{i+1}} \sigma_t dW_t. \tag{2.3}$$

Assuming a very small time interval Δt, we can approximate the f and σ terms as constants (at time t_i) and thus Equation 2.3 will become

$$\int_{t_i}^{t_{i+1}} dx_t = x_{t_i+1} - x_{t_i} = f(x_{t_i}) \int_{t_i}^{t_{i+1}} dt + \sigma_{t_i} d \int_{t_i}^{t_{i+1}} W_t$$

$$\therefore x_{t_i+1} = x_{t_i} + f(x_{t_i})\Delta t_i + \sigma_{t_i}\Delta W_{t_i}. \tag{2.4}$$

This is called the Euler-Maruyama Scheme of process x with $\Delta t_i = t_{i+1} - t_t$ and $\Delta W_{t_i} = W_{t_i+1} - W_{t_i}$. For simplicity, let us assume $t_i = 0$, $\Delta t_i = h$ and $\Delta W_{t_i} = \delta W_0$, so from Equation 2.4 , we will have

$$x_1 = x_0 + hf(x_0) + \sigma(x_0)\delta W_0. \tag{2.5}$$

2.4.2 Learning drift and diffusivity using Neural Networks

Now, from the work of Dietrich *et al.* [3], we can learn the drift f and diffusivity σ, directly from the tuples (x_0, x_1, h) over time. Let us assume that f and σ are neural networks with weights θ. In order to train this network from the tuples in our dataset D, we need to formulate the equation for loss function from Equation 2.5 and minimize it.

We can think of x_1 as a multivariate normal distribution conditioned on x_0 and h. We need to find out the expected value ($\mathbb{E}$) and variance (Var) to define it, so from Equation 2.5

$$\begin{aligned}
\mathbb{E}(x_1) &= \mathbb{E}(x_0 + hf(x_0) + \sigma(x_0)\delta W_0) \\
&= \mathbb{E}(x_0 + hf(x_0)) && \because\ \mathbb{E}(\sigma(x_0)\delta W_0)) = \sigma(x_0)\mathbb{E}(\delta W_0)) = 0 \text{ by definition; see Section 2.4} \\
&= x_0 + hf(x_0) && \because \text{ there is no stochastic term left, so purely deterministic}
\end{aligned} \tag{2.6}$$

$$\begin{aligned}
Var(x_1) &= Var(x_0 + hf(x_0) + \sigma(x_0)\delta W_0) \\
&= Var(\sigma(x_0)\delta W_0) && \because \text{ the other terms are deterministic, so they do not 'vary'} \\
&= \sigma^2(x_0)Var(\delta W_0) && \because\ \sigma(x_0) \text{ is a constant and } Var(cX) = c^2 * Var(X) \text{ if } c \text{ is a constant} \\
&= h\sigma^2(x_0) && \because \text{ std. dev. of } \delta W = \sqrt{\delta t} = \sqrt{h}\text{; see 2.4 and } Var = (std.\ dev)^2
\end{aligned} \tag{2.7}$$

$$\therefore x_1 \sim \mathcal{N}(x_0 + hf_\theta(x_0),\ h\sigma^2_\theta(x_0)). \tag{2.8}$$

Now we can find the probability density function for Equation 2.8 from

$$p(x) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}}\Sigma^{-1}(\mathbf{x}-\mu)\right)}{\sqrt{(2\pi)^k|\Sigma|}} \quad \text{where } \mu = \text{mean and } \Sigma = \text{variance.} \tag{2.9}$$

For convenience, we can take the natural logarithm of Equation 2.9

$$\ln(p(x)) = -\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}}\Sigma^{-1}(\mathbf{x}-\mu) - \frac{k}{2}\ln(2\pi) - \frac{1}{2}\ln(|\Sigma|). \tag{2.10}$$

Next, we can define the maximum (expected) log likelihood for new position x_1 given current position x_0 and time interval h as

$$\theta := \arg\max_\theta \ln(p_\theta(x_1|x_0, h)). \tag{2.11}$$

To repeat this for all tuples (x_1, x_0, h) in our dataset D (with N tuples), we can take the average of all the maximum (expected) log likelihood values and maximize that

$$\theta \approx \arg\max_{\theta} \frac{1}{N} \sum_{n=1}^{N} \ln \left(p_\theta(x_1^{(n)} | x_0^{(n)}, h^{(n)}) \right). \tag{2.12}$$

Finally, to get the loss function to estimate θ for drift (f_θ) and diffusivity (σ_θ), we can take negative of Equation 2.12 and substitute $p(x)$ from Equation 2.9 :

$$\mathcal{L}(\theta | x_1, x_0, h) \coloneqq \frac{1}{2} \frac{(x_1 - x_0 - h f_\theta(x_0))^2}{h \sigma_\theta^2(x_0)} + \frac{k}{2} \ln(2\pi) + \frac{1}{2} \ln\left(h \sigma_\theta^2(x_0)\right). \tag{2.13}$$

Minimizing the loss in Equation 2.13 over dataset D implies maximizing the log likelihood values in Equation 2.12 . This will allow us learn the values of drift (f_θ) and diffusivity (σ_θ) directly from the given image data.

3 Learning Aerosol Dynamics from Images

3.1 Problem Definition

The aim of this project is to train a neural network, which can model aerosol (cloud) growth over time, given a dataset of simulations of persons breathing out white smoke against a dark background (in image format). We will construct a pipeline which takes in random values for the turbulence strengths, which are used to generate the image dataset. This data is first used to train a U-Net based image segmentation model, which will separate the aerosol cloud from the rest of the image. From this we measure the cloud size of each segmented image and create another dataset consisting of cloud size at each frame for each simulation (given some input parameters such as turbulence). We will use this dataset to train the SDE Neural Network model, in order to learn the aerosol growth over time.

3.2 Pipeline

Figure 3.1 demonstrates the entire process taken during this project, from random values to dataset generation, to training the U-Net model and SDE neural network.

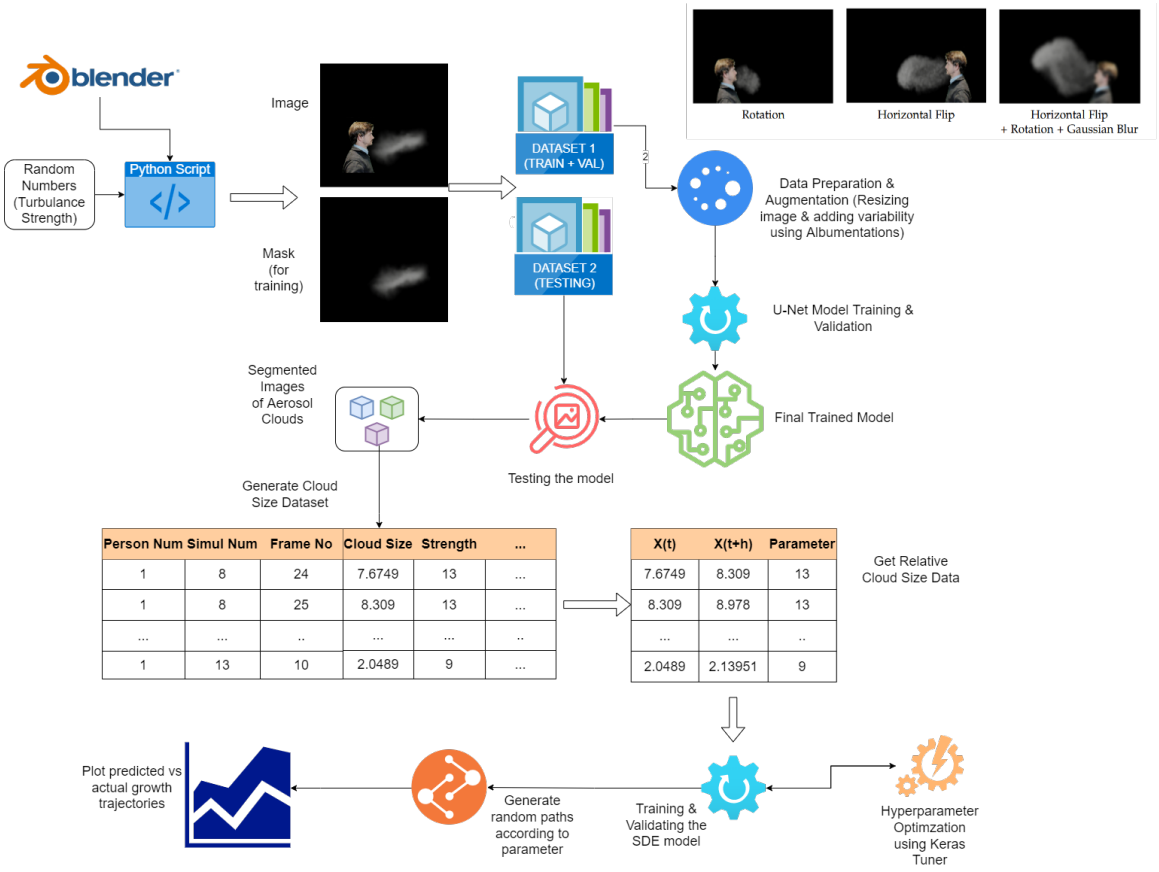


Figure 3.1: Pipeline for the project

3.3 Generating Synthetic Data Using Blender

There is not much image data available for smoke simulations, where a human is exhaling out white smoke against a dark background. However, we do not need to have perfect/real-life training data to train our image segmentation model for this task. For image segmentation, U-Net is efficient enough to use data augmentation to utilize the given annotated data effectively. Hence, crude simulations from Blender should be enough.

Rohan gathered images of humans from stock photographs on the internet, which are free to use, and with consent of the individuals involved. He then set up the Blender simulations with the person background, smoke emitter and turbulence objects. He generated image and mask data for 7 simulations of 11 different persons [8]. However, as we will see later, more data was required to better fit the SDE model, to have smoother trajectories.

For generating new data, we had to tweak the turbulence strength of the *Turbulence* object. All 11 people had their own `.blend` file, where we could do it manually. However,

it is slow to tweak the value with mouse and clicking to generate data $11n$ times (11 people and n simulations).

Hence, to automate things, we have created a simple script using the Blender Python API [1]. Using this script , we can vary any parameters in the initial simulations set up in `.blend` files of each person (right now, we are only tweaking the turbulence strength). We generate random turbulence values between 1 and 15 (as values more than 15 resulted in smoke covering the person's face significantly). Then we pass these values to the script to generate and organize the data in our original dataset according to `person_num` and `simul_num`. Now, we can generate as much data as we want from random values and all we have to do is wait for the simulations to finish.

3.4 Image Segmentation using U-Net Architecture

We wish to have a U-Net model which is able to generalize well, meaning it should work with any person regardless of their skin tone, camera angle or clothing. The model should generate a segmentation map for the image, meaning each pixel will have a particular label (binary segmentation in this case, as there are only 2 labels namely: part or not part of the aerosol cloud).

Albumentations library is used to create some randomness in the image such as horizontal/vertical flips and Gaussian blurs. The U-Net network is then trained on this modified dataset with 11 people and 7 simulations.

3.4.1 Evaluation Metrics

1. Pixel Accuracy is the percentage of pixels which have been correctly segmented.

$$Accuracy = \frac{N_{true\ positives} + N_{true\ negatives}}{N_{true\ positives} + N_{true\ negatives} + N_{false\ positives} + N_{false\ negatives}} \tag{3.1}$$

   However it may not be a very good measure as in case of a class imbalance, where ($N_{background\ pixels} \gg N_{foreground\ pixels}$), the model may not segment the background images at all, however it will still get a high accuracy as this score did not place more importance on the segmented part.

2. Dice Score defined as

$$\begin{aligned} DICE &= \frac{2 * Area\ of\ Overlap}{Total\ pixels\ in\ both\ images} \\ &= \frac{2 * N_{true\ positives}}{2 * N_{true\ positives} + N_{false\ positives} + N_{false\ negatives}} \end{aligned} \tag{3.2}$$

[1] https://docs.blender.org/api/current/info_overview.html
[1] https://albumentations.ai/

Hence, it not only finds the number of true positives found, but also penalizes the model for finding false positives. This makes it a much more robust measure than accuracy, especially for image segmentation tasks. A value closer to 1 means a perfect segmentation and vice-versa.

Figure 3.2 gives a better idea about the difference between dice score and accuracy.

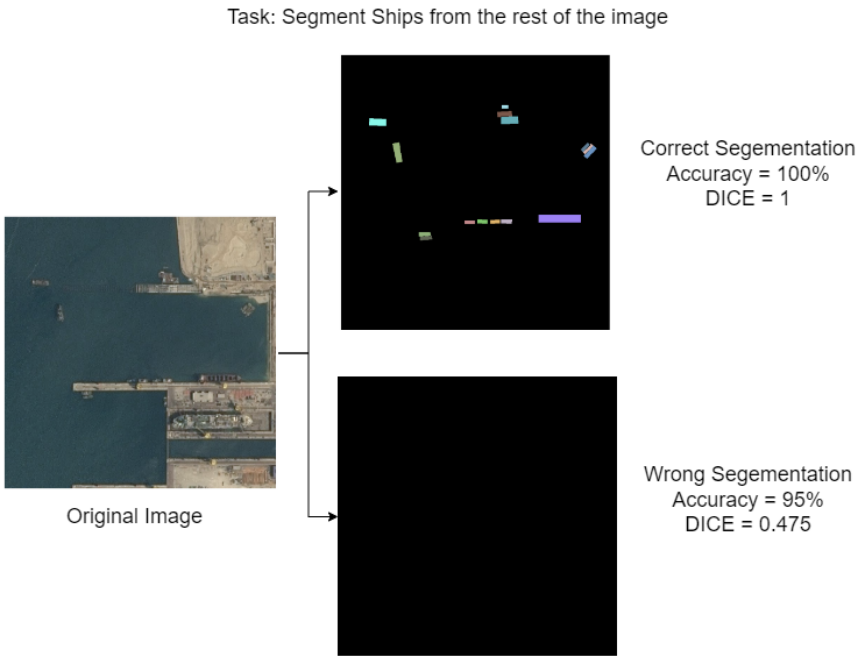


Figure 3.2: Comparison between DICE score and Pixel Accuracy [2]

3.4.2 Results

First, we recreated the results that Rohan found. A sample segmentation is visible in Figure 3.3 . We used 9 people's data for training and the remaining 2 for validation. Next, we used another dataset (never seen by the model) of 11 people with 7 additional simulations each, for testing. Here, we found an accuracy of 99.871% and a dice score of 0.985, which has about a 0.7% error/deviation from what Rohan found (99.9% accuracy and 0.992 dice score).

Next, we tried to retrain the model with more simulations. We created additional 6 simulations for all 11 persons in the first dataset. Using the same train-validation-test split, we were able to get an accuracy of 99.86%. and a dice score of 0.985.

[2] Image Source: https://towardsdatascience.com/metrics-to-evaluate-your-semantic-segmentation-model-6bcb99639aa2

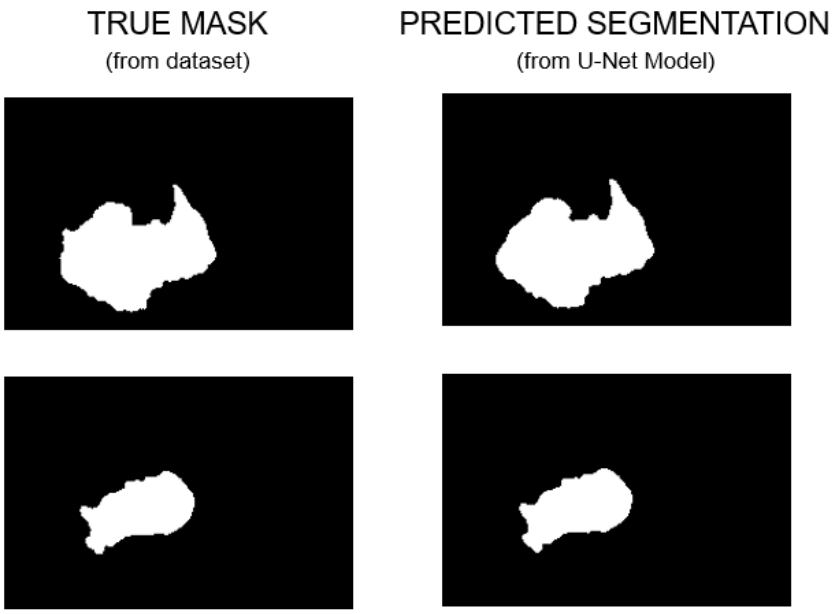


Figure 3.3: U-Net Model Predictions vs Actual Masks from dataset

This is still close to what we had before, and thus adding new data to train the model was not very useful. This verifies that U-Net is quite efficient at making good use of the limited data it might have available, and hence throwing more data at it might not improve it's performance significantly.

3.5 Modelling Cloud Sizes with SDEs using Neural Networks

Now that we have segmented the cloud part of the image from our dataset, we wish to learn the growth of this cloud over time. We will first define $Cloud\ Size$ as

$$Cloud\ Size = \%_{white\ pixels} = \frac{N_{total\ pixels} - N_{black\ pixels}}{N_{total\ pixels}}$$

$$where\ N_{black\ pixels} = N_{total\ pixels} \mid Gray\ Value\ (pixel) < 10\ [Thresholding] \tag{3.3}$$

Next, let us visualize the exact trajectories that we will be attempting to learn. From the original Blender Masks, if we plot $Cloud\ Size\ vs\ Time$ for 11 persons and 7 simulations, we get Figure 3.4 .

As we can see for each person, the graphs for the simulations are more or less the same. This would not be true in real world, as the cloud growth may depend on the person's size or lungs. However, now it makes sense as we are using simulations and thus, the aerosol

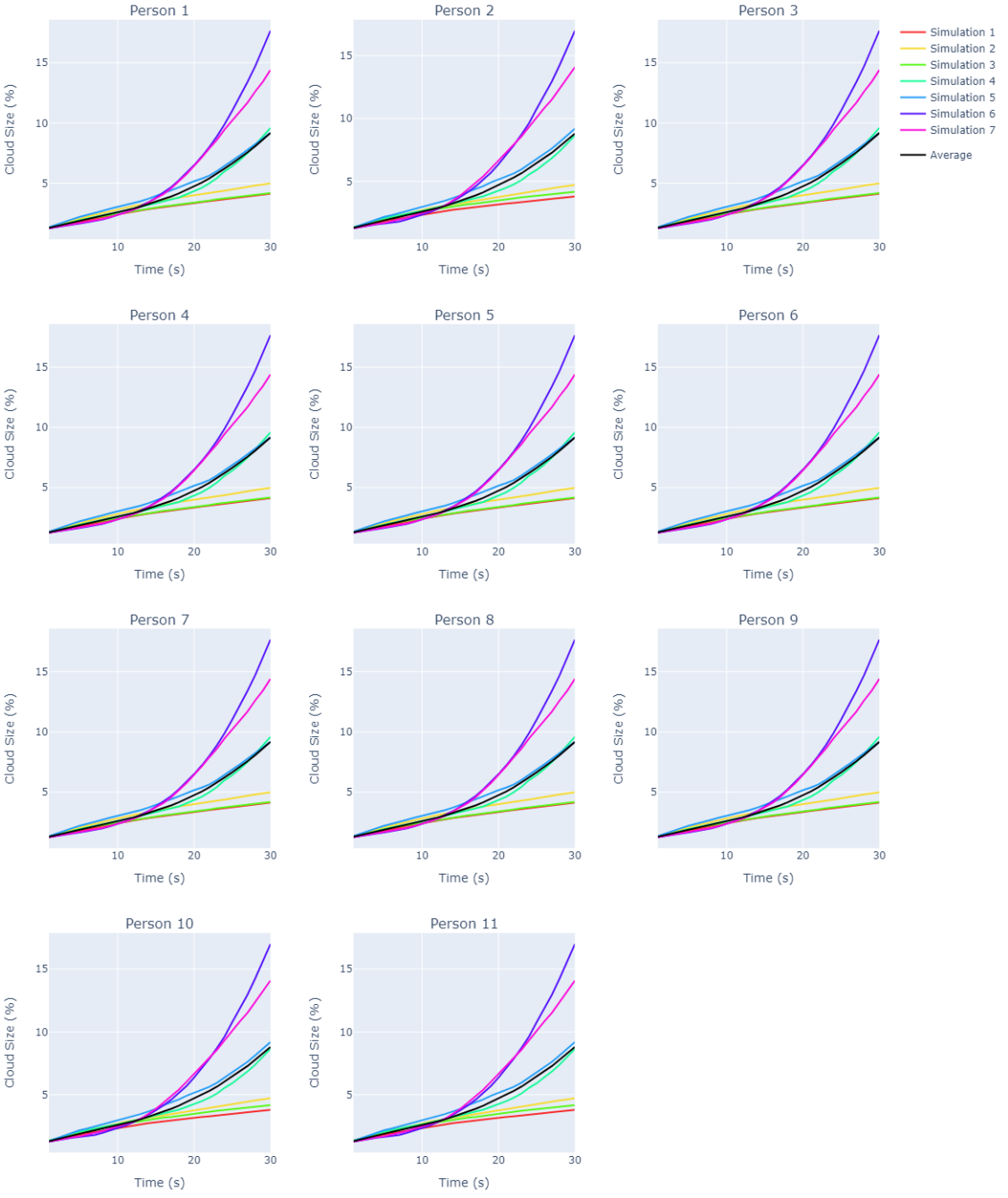


Figure 3.4: Cloud growth over time for 11 persons and 7 simulations

growth should not be dependent on the person in the image. It should only be affected by external factors such as turbulence, posture, etc. So these will be the trajectories we will be attempting to learn using SDEs.

3.5.1 Extracting data from the dataset

As we saw in Section 2.4 , we need the triplets (x_0, x_1, h) in order to train the neural network. To get these, we will first parse through the entire image dataset and their dataset info files to get a detailed dataset, as can be seen below:

	Person	Simulation	Frame No	Cloud Size	Strength	Size	Flow	Noise Amount	Seed	Wind Factor
0	1	1	1	1.258295	0.0	0.0	0.0	0.2	4	0
1	1	1	2	1.391879	0.0	0.0	0.0	0.2	4	0
2	1	1	3	1.534481	0.0	0.0	0.0	0.2	4	0
3	1	1	4	1.691020	0.0	0.0	0.0	0.2	4	0
4	1	1	5	1.809076	0.0	0.0	0.0	0.2	4	0
...	...	...	...	...	...	...	...	...	...	...
2305	11	7	26	10.836034	15.0	0.0	2.0	0.2	4	0
2306	11	7	27	11.522473	15.0	0.0	2.0	0.2	4	0
2307	11	7	28	12.363860	15.0	0.0	2.0	0.2	4	0
2308	11	7	29	13.197627	15.0	0.0	2.0	0.2	4	0
2309	11	7	30	14.084057	15.0	0.0	2.0	0.2	4	0

2310 rows × 10 columns

Figure 3.5: Cloud Size Dataset.

From Figure 3.5 , we can extract $(x_0, x_1, param)$ values (as h is implicitly defined as repeating $[1...30]$ over and over, which are the frame numbers in the simulations). Doing that we get Figure 3.6 . This is the dataset we can pass to the SDE neural network created by Dietrich *et al.* [3]. We can now test it with different configurations of this dataset to see the output.

3.5.2 Results with 11 people and 7 simulations each

Without Parameter Values

We can first try to train the model on just (x_0, x_1) values alone, i.e. without considering any of the initial parameters such as turbulence. Passing this to the model, we get a training loss of 0.7195 and a validation loss of 0.829. The resulting graphs are in Figure 3.7 .

As we can see in Figure 3.7c , the predicted paths are all over the place and not very smooth. However, this can be improved if we also consider some initial parameters from the simulations to model it.

	X(t)	X(t+h)	Parameter
0	1.258295	1.391879	4.2
1	1.391879	1.534481	4.2
2	1.534481	1.691020	4.2
3	1.691020	1.809076	4.2
4	1.809076	1.934414	4.2
...	...	...	...
2228	10.125145	10.836034	21.2
2229	10.836034	11.522473	21.2
2230	11.522473	12.363860	21.2
2231	12.363860	13.197627	21.2
2232	13.197627	14.084057	21.2

2233 rows × 3 columns

Figure 3.6: Relative cloud growth over time for 11 persons and 7 simulations

Table 3.1: Losses for non-parametrized case of (11, 7) configuration

Loss	Data	Value
Training	9 people with 7 simulations each	0.7292
Validation	20% of training data	0.8457
Testing	2 people with 7 simulations each	1.0015

With Parameter Values

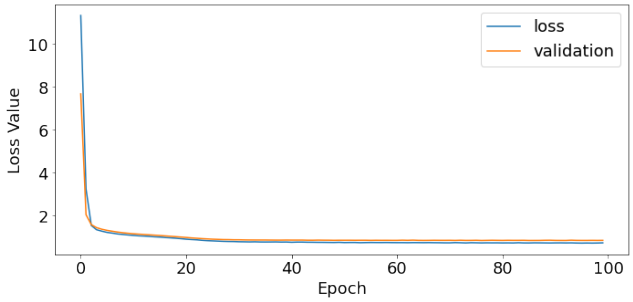
If we pass the entire triplet $(x_0, x_1, param)$ to the model, we get the a training loss of -0.323 and a validation loss of -0.117. As we can see in Figure 3.8 , with 4 parameters, it is quite hard to visualize the paths. So let us try to plot them in 3D with (x, y, z) as $(time, param, cloud\ size)$. We will plot 3 graphs, first is actual cloud growth over time, second is predicted cloud growth over time and third is comparing the actual and predicted growths.

As we can see in Figure 3.9 [C], we have multiple predicted paths for a single parameter. That is because since we have a stochastic process, it will vary because of the initial condition (which is random). Hence, we see a lot of paths for every parameter value depending on the random initial condition.

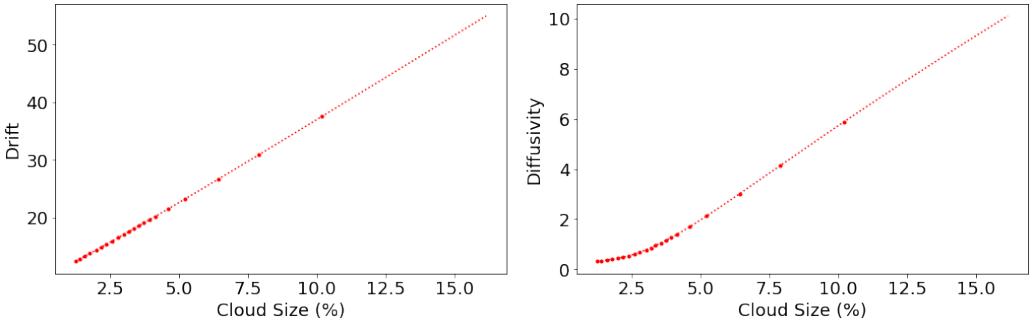
However, none of the predicted paths fit very well to the actual paths. They grow much faster in size as compared to the actual masks. So let us look at some ways in which we can get a better fit model.

Table 3.2: Losses for parameterized case of (11, 7) configuration

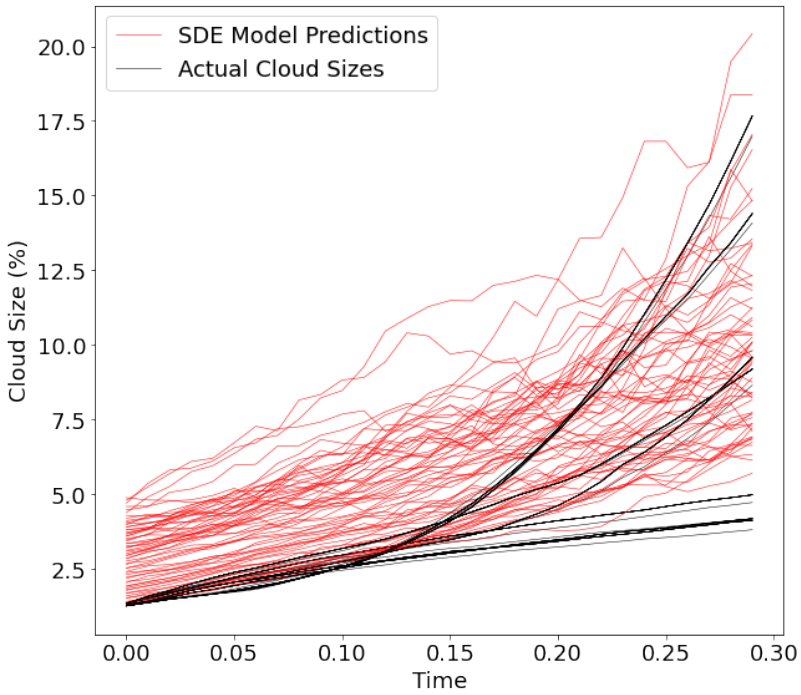
Loss	Data	Value
Training	9 people with 7 simulations each	-0.2846
Validation	20% of training data	-0.2181
Testing	2 people with 7 simulations each	-0.03773



(a) Training and validation loss

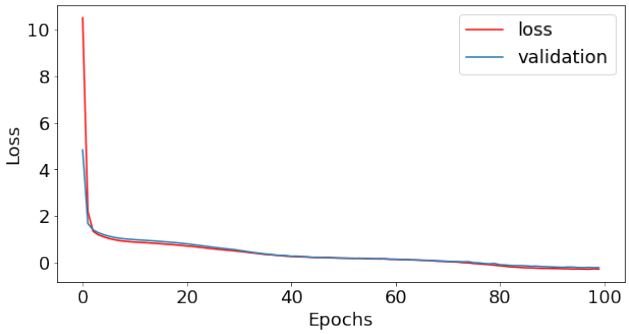


(b) Drift and Diffusivity over time

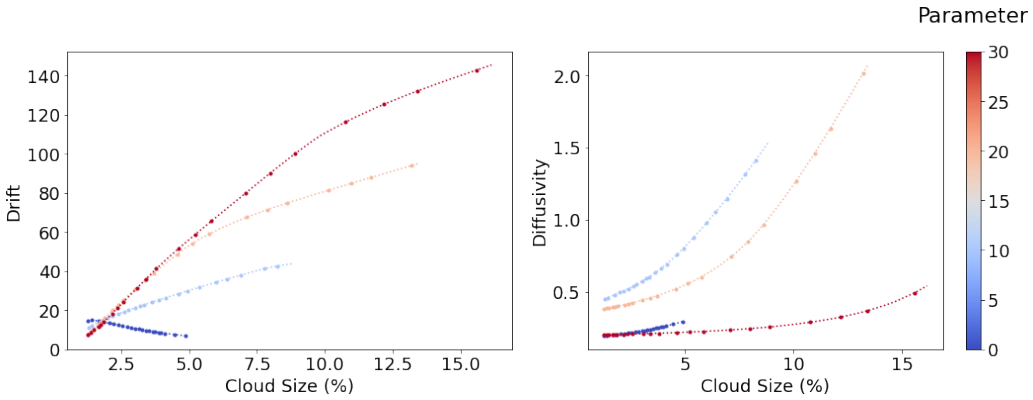


(c) Predicted vs Actual Cloud growth over time

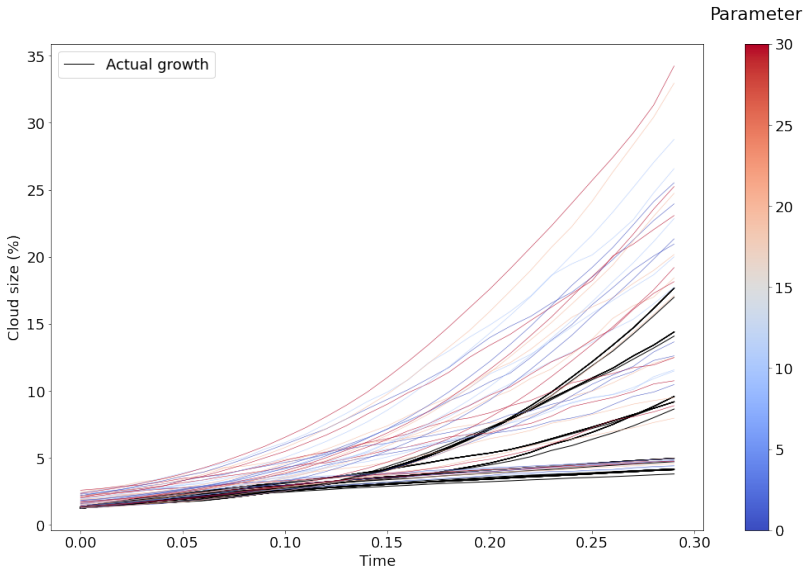
Figure 3.7: Various plots for 11 people with 7 simulations



(a) Training and validation loss



(b) Drift and Diffusivity over time



(c) Predicted vs Actual Cloud growth over time

Figure 3.8: Various plots of 11 people and 7 simulations with parameter as an input

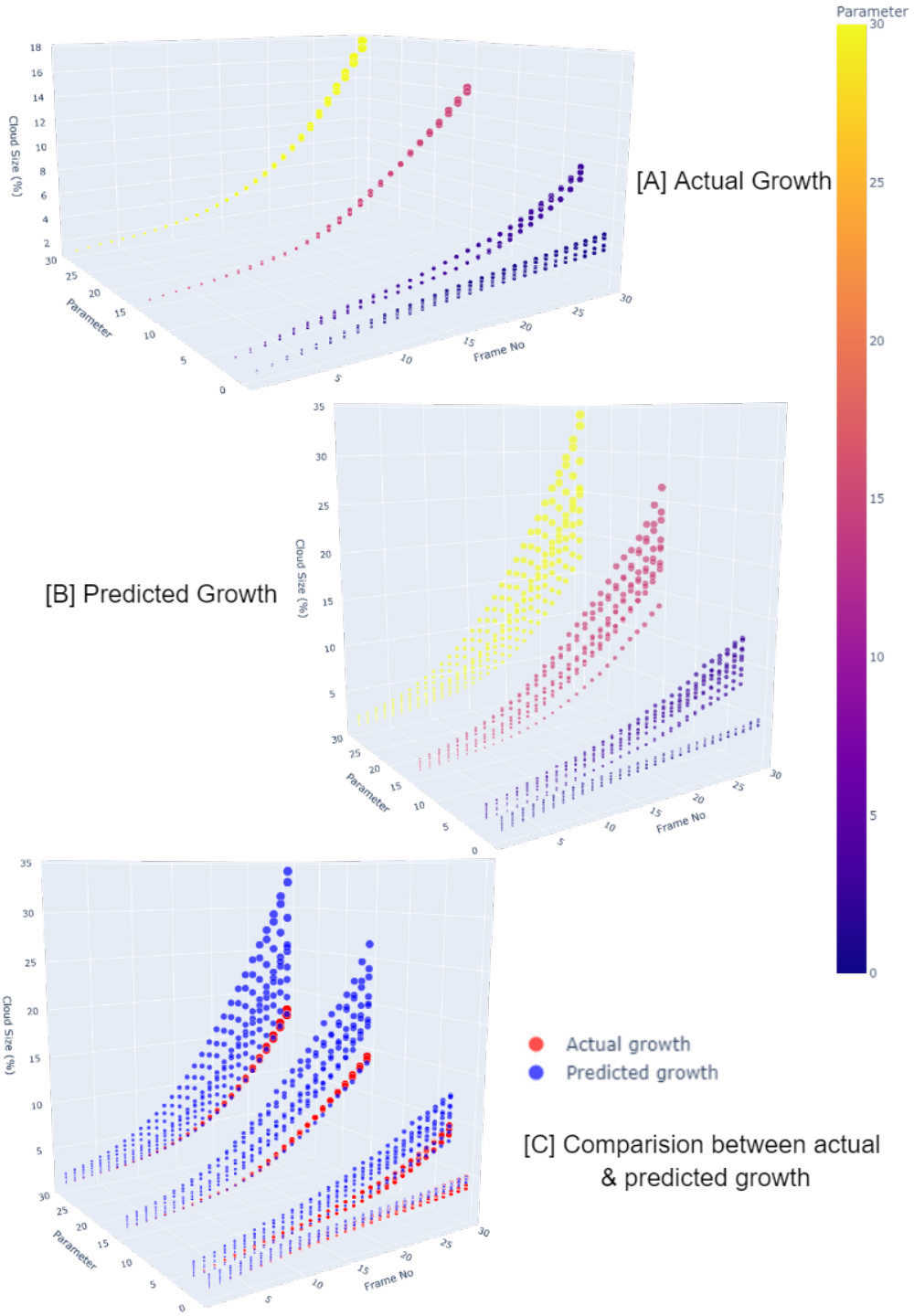


Figure 3.9: 3D plots for 11 people and 7 simulations with parameters

3.5.3 Quantifying Model Predictions

We can see that *loss* is not a good metric to tell us how good our model is at predicting the trajectories. Both parameterized and non-parameterized cases have quite low losses, but they still do not fit properly. Hence, we need to define a metric to quantify how good our model is doing. For this, we will use the Mean Square Error (MSE). To calculate it:

1. We will first consider a particular parameter. For each frame number in that parameter, we will average all of the different predicted cloud size value (which are a result of the fact that we have a stochastic process). We will repeat this for all parameters. This will give us a predicted average trajectories for each parameter.
2. We will repeat the same for original dataset as well, which will give us actual average trajectories for each parameter.
3. Finally, to calculate MSE, we can just square the difference of these 2 arrays of cloud sizes. Our goal to get a better fit is to minimize MSE.

Currently, the MSE will (11 people, 7 simulations) configurations (without optimizations) is 7.107. And the average paths can be seen in Figure 3.10 .

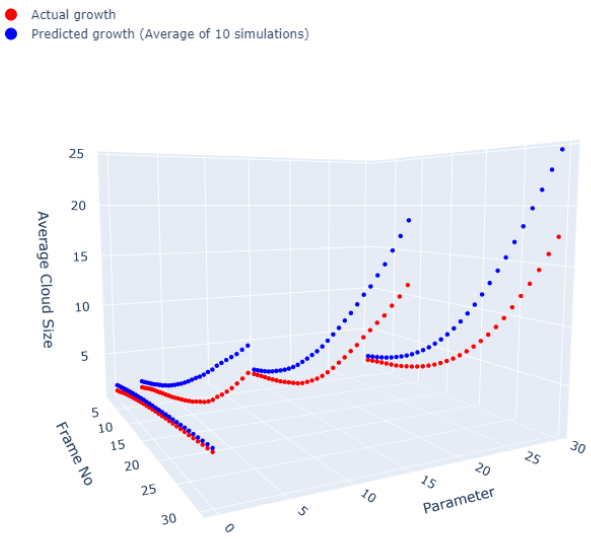


Figure 3.10: Comparison of average trajectories 11 people and 7 simulations with parameters

3.5.4 Results with fresh dataset of 11 people with 13 simulations each

In the previous results, we were using 7 pre-existing simulations (done by Rohan) and 6 new simulations, that we created with random turbulence values. However, the previous data had some problems, namely:

- The data was not consistent. Each simulation had varying values for noise amount, initial velocity, flow, etc. Since we are only trying to learn the strength parameter, all the other varying variables can hamper learning.
- There was not enough distribution in the values of the strength. A good portion of the data (the first 7 simulations) only had 4 different parameter values. This meant that the model may have had too much data for them and thus might be overfitted because of it.
- Finally, we also want variability in the $seed$ parameter. This is because it is responsible for the inherent randomness in the simulations, which no model can ever learn (also true in real life). Hence, our model should be good enough that it can learn the cloud growth trajectories despite changing seed values. However the previous dataset had same seed values for all simulations.

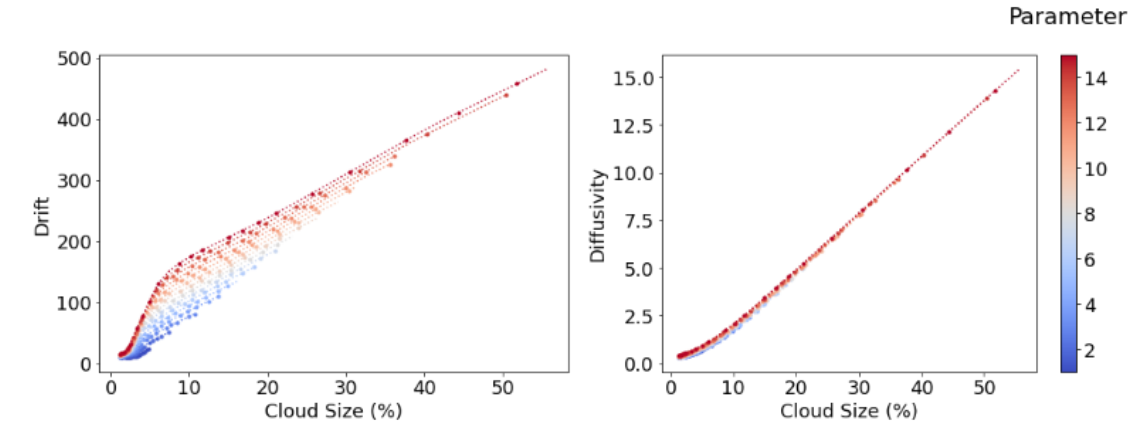
So to overcome these issues, we created a fresh dataset (11 persons with 13 simulations) with completely random turbulence strength values, from 1-15 (values higher than 15 leads to very chaotic images). We keep distinct seed for each simulation, starting from 1 to $N_{simuls} * N_{persons}$. Other parameters such as size, flow and wind factors are set to constant (0). And the noise amount is set to 0.2 (actually 0.20000000298023224 because of a floating point error in Blender-Python API, but since it is constant across all simulations, it should not be an issue).

Table 3.3: Losses for a fresh dataset of (11, 13) configuration (parameterized)

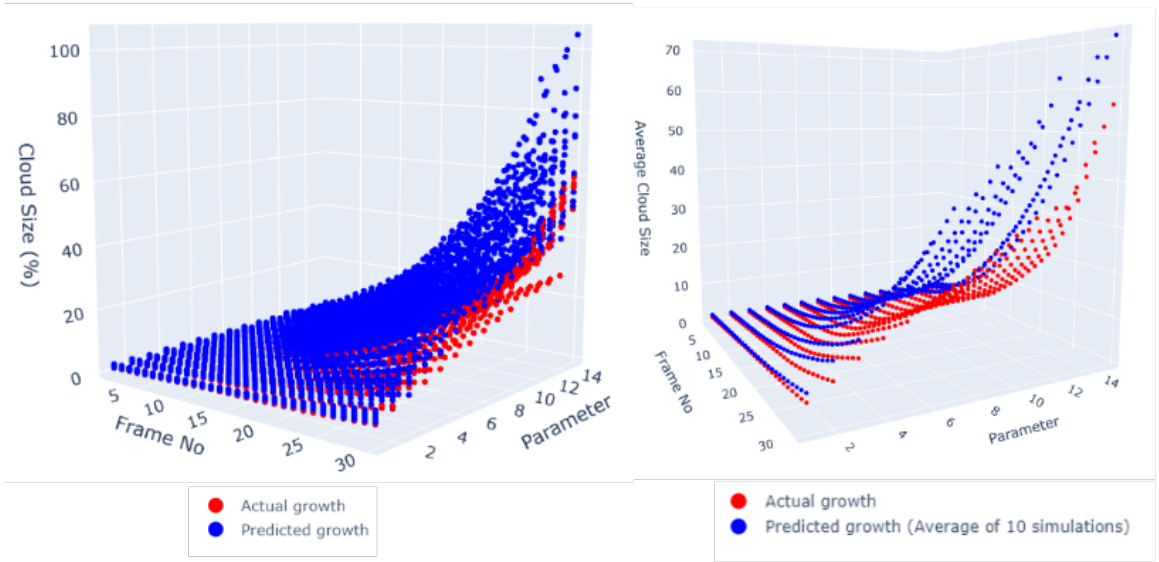
Loss	**Data**	**Value**
Training	9 people with 13 simulations each	0.6713
Validation	20% of training data	0.6750
Testing	2 people with 13 simulations each	0.4935

As we can see in Figure 3.11 [B], we have a much more uniform spread of parameters, with no peaks in between. We also have a surface like resemblance to the cloud growth (which makes sense as higher parameter values leads to higher cloud sizes). And finally, even with the varying seed parameter, the model was able to learn the cloud growth trajectories.

An interesting thing to note is that in Figure 3.11 [A], we can see that with increasing parameter (turbulence), the drift (movement from original position) increases. This makes sense as increasing turbulence would lead to more chances of the aerosol cloud spreading.



[A] Drift & Diffusivity plots over cloud size



[B] Predicted vs Original trajectories

[C] Original vs average of predicted trajectories

Figure 3.11: Various plots for fresh dataset 11 people and 13 simulations (parameterized)

However, the diffusivity (how randomly it spreads) appears to remain constant with parameter size. We would expect it to also increase, as more turbulence in the environment would lead to more randomness, however we do not see that. Perhaps the reason for this is because on the X-axis, we using *Cloud Size*, which is the percentage of the image, which is covered by white pixels. But, if there were many small clouds or one big cloud, the *Cloud Size* would still remain the same. Hence, if we perhaps trained it using some other metric (like cloud shape), we would get different results.

The MSE in this case is 123.54, which is still quite high. It is more than the less dataset case because we have more data points for the parameters (previously there were only 4, but now there are 15). However, we are still overshooting for higher cloud size values.

3.5.5 Imbalanced Dataset

In order to see what is causing the problem of overshooting in higher cloud size range, let us plot the histogram for all the cloud sizes:

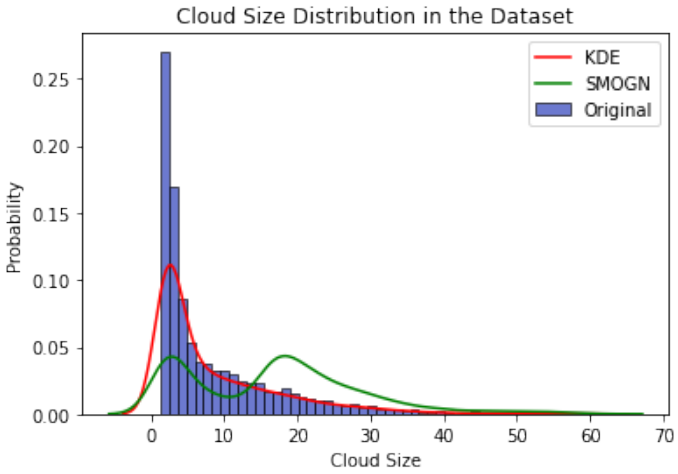


Figure 3.12: Cloud Size Distribution in the Dataset

As we can see in Figure 3.12 , there is a clear imbalance in the dataset. This means that small cloud size values (from $0 - 20\%$) are represented a lot more than the higher cloud size values (more than 30%). This leads to our model being skewed/biased towards smaller cloud sizes during training. As a result in Figure 3.11 , we see that in the predicted trajectories, the smaller cloud sizes fit much better than the higher cloud sizes. The model does not have enough training data for the minority class (higher cloud sizes) to predict properly.

Now, imbalanced datasets are not new and have been seen multiple times in real life datasets. There are various techniques to counter this imbalance such as sampling (over and under sampling) or Synthetic Minority Oversampling Technique (SMOTE). However,

all of these methods are made for classification problems (where some classes are represented more than others), but we are dealing with a regression problem (we have to predicted the next cloud size in a given trajectory).

The literature about dealing with imbalanced regression problem is limited, but we will look at 2 approaches:

Synthetic Minority Oversampling Technique for Regression (SMOGN)

SMOGN [1] is the application of traditional SMOTE algoithm (used in classification) for regression problem. It uses the current data and interpolates, to create more instances for the minority classes. It also introduces some Gaussian noise depending on the KNN distance between the underlying observations. When SMOGN is applied on our dataset, the resulting histogram can be seen in green in Figure 3.12 . As we can see, we get a much better spread in the frequencies, however, we are still missing representation from higher cloud sizes. We would expect this to not perform very well for higher cloud sizes as well, which is seen in the following graphs:

Table 3.4: Losses when trained on SMOGN dataset

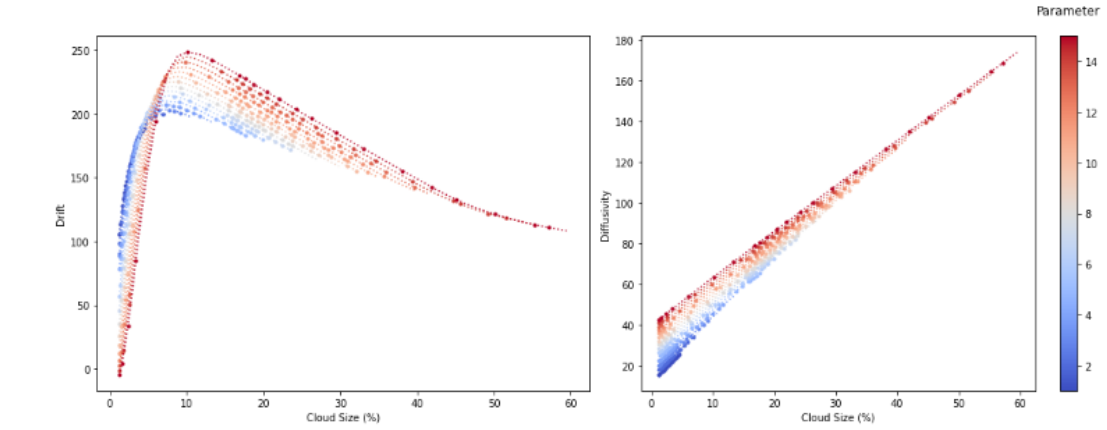
Loss	Data	Value
Training	9 people with 13 simulations each	4.8711
Validation	20% of training data	4.5430
Testing	2 people with 13 simulations each	4.389

As we can see in Figure 3.13 [C], the graphs are not very smooth and there are a lot of outlier points. The MSE is also quite high at 686.36. The reason for this is because SMOGN automatically removes certain observations (rows) and columns (features) which contain missing values, as can be seen in Figure 3.14 . However, for our task, we do not want a dataset with missing points as we are dealing with a time bound dataset where order matters. So we cannot skip certain frames in training, otherwise the model will not properly learn how it grows over 30 frames. Hence, SMOGN is not very useful for our task.

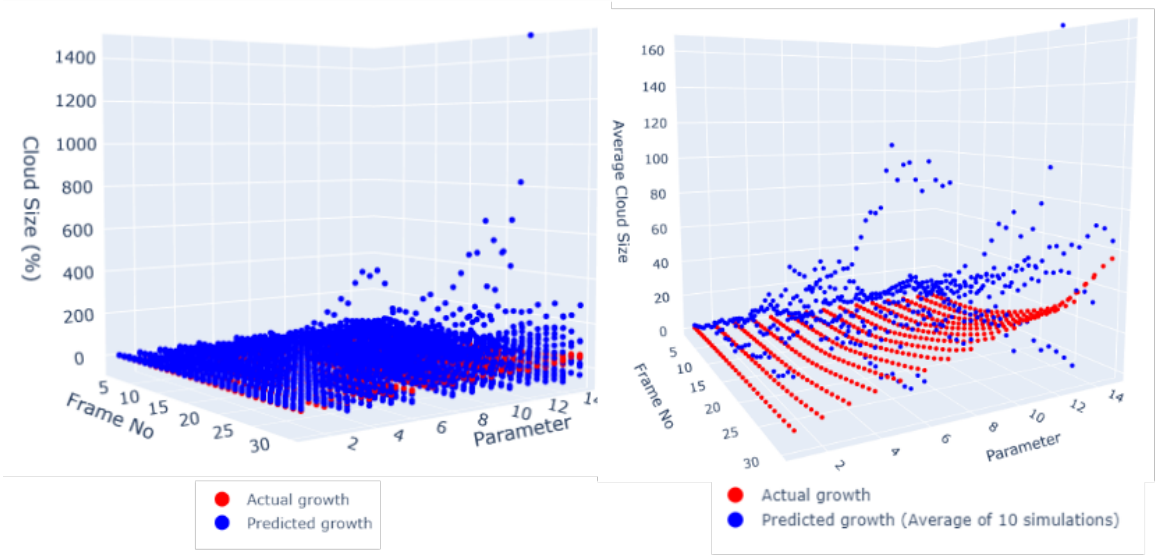
Reweighing samples using Kernel Density Estimation (KDE)

However, there is another approach to trying to solve the imbalanced dataset problem. We can reweigh the training sample so that samples occurring more frequently (smaller cloud sizes) have less weights and vice versa [9]. This would ensure that whilst the frequencies are different, we can still level the playing field by adjusting the weights given to training samples. Thus, in the ideal scenario here, larger cloud sizes (which are more scarce) would have higher weights as compared to smaller cloud sizes.

In order to achieve this, we will use a Kernel Density Estimator (KDE), which uses kernel smoothing to do probability density estimation. In other word, it returns a function which



[A] Drift & Diffusivity plots over cloud size



[B] Predicted vs Original trajectories

[C] Original vs average of predicted trajectories

Figure 3.13: Various plots for (11, 13) dataset after SMOGN

```
1 cs_smogn
[36] ✓ 0.3s
```

	Person	Simulation	Frame No	Cloud Size	Strength	Size	Flow	Noise Amount	Seed	Wind Factor	Parameter
1	1	1	2	1.480421	12	0.0	0.0	0.2	0	0	12
5	1	1	6	2.292052	12	0.0	0.0	0.2	0	0	12
6	1	1	7	2.515287	12	0.0	0.0	0.2	0	0	12
7	1	1	8	2.757186	12	0.0	0.0	0.2	0	0	12
10	1	1	11	4.188320	12	0.0	0.0	0.2	0	0	12
...	...	...	...	...	...	...	...	...	...	...	...
4279	11	13	20	5.772087	4	0.0	0.0	0.2	0	0	4
4281	11	13	22	7.173032	4	0.0	0.0	0.2	0	0	4
4284	11	13	25	9.652778	4	0.0	0.0	0.2	0	0	4
4286	11	13	27	11.371672	4	0.0	0.0	0.2	0	0	4
4289	11	13	30	14.213204	4	0.0	0.0	0.2	0	0	4

3503 rows × 11 columns

Figure 3.14: Dataset after SMOGN with certain rows missing

traces the histogram of some given data. Figure 3.12 demonstrates this quite clearly where the red line is the KDE for the original histogram. This is done by drawing a Gaussian curve at each point on the graph and adding them up to form a smooth curve like the one we see in the red.

Once we have the KDE, we can flip it upside down (with respect to it's midpoint) to get a curve which looks like Figure 3.15 .

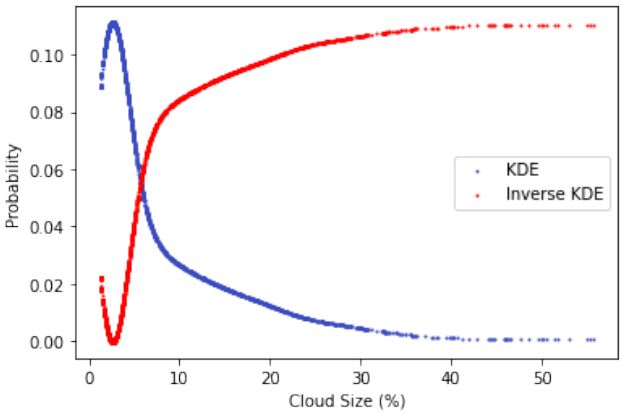


Figure 3.15: KDE vs KDE flipped with respect to mid point

If we pass these as weights during the training (by multiplying the weights with the loss values), and we test the model, we get the trajectories in Figure 3.16 .

Table 3.5: Losses after reweighing with inverse KDE weights

Loss	Data	Value
Training	9 people with 13 simulations each	1.5905
Validation	20% of training data	1.5871
Testing	2 people with 13 simulations each	1.4080

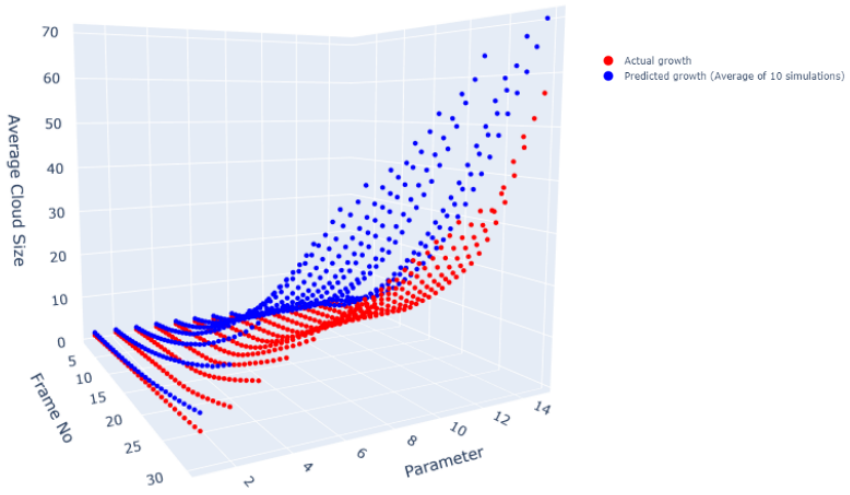


Figure 3.16: Trajectory comparision when passing inverse KDE as weights to the model

For this case, we get a MSE of 156.363. As we can see from this and the trajectories, we actually get worse performance than using no weights at all! However, this may be due to that the hyperparameters currently are meant for unweighted case. Perhaps tuning the hyperparameters with weights might improve the problem.

3.5.6 Hyperparameter Optimization

Hyperparameter means high-level parameters/variables which are selected *before* the learning starts. These are the parameters which actually *control* the learning itself. For eg., parameters like number of layers, number of neurons per layer, the learning rate, etc are hyperparameters, because changing them without changing input data, will still lead to different results. We can tune these hyperparameters in any machine learning model. Initially, these values are chosen completely at random and we then tune them over time using the validation dataset. It is not possible to know the most optimal hyperparameters (to get a perfectly fitting model) in the beginning, which is why we have to find them out using trial and error.

As we are using TensorFlow for this project, we will be using the Keras Tuner library

[3], which helps us optimize hyperparameter easily. We will be trying to vary N_{layers}, $N_{dim\ per\ layer}$ and $learning\ rate$ of the the neural network. The search space is listed in Figure 3.17 .

```
    1 tuner.search_space_summary()
[36] ✓ 0.8s
...
Search space summary
Default search space size: 3
n_layers (Int)
{'default': None, 'conditions': [], 'min_value': 1, 'max_value': 36, 'step': 1, 'sampling': None}
n_dim_per_layer (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 512, 'step': 16, 'sampling': None}
learning_rate (Choice)
{'default': 0.0001, 'conditions': [], 'values': [0.0001, 0.001, 0.01, 0.1], 'ordered': True}
```

Figure 3.17: HyperBand Tuner Search Space

The tuner will be using HyperBand search algorithm [6] to search. HyperBand is a more efficient RandomSearch algorithm, where it starts by trying random tuples for the hyperparameters, but then uses adaptive resource allocation and early stopping to optimize the process. Also, we cannot cover the entire search because with the amount of permutations we have and the fact that each trial will have 50 epochs, this process will take a long time. So we will be focusing on finding a good enough solution . We will be using $validation\ loss$ as our objective (to minimize).

Initially, the tuner will pick random combinations of N_{layers}, $N_{dim\ per\ layer}$ and $learning\ rate$ from the search space. First it will run few (4-5) epochs on it. Then it will take the best performing samples, and run it through more epochs. Meanwhile, it will also keep trying other unseen combinations.

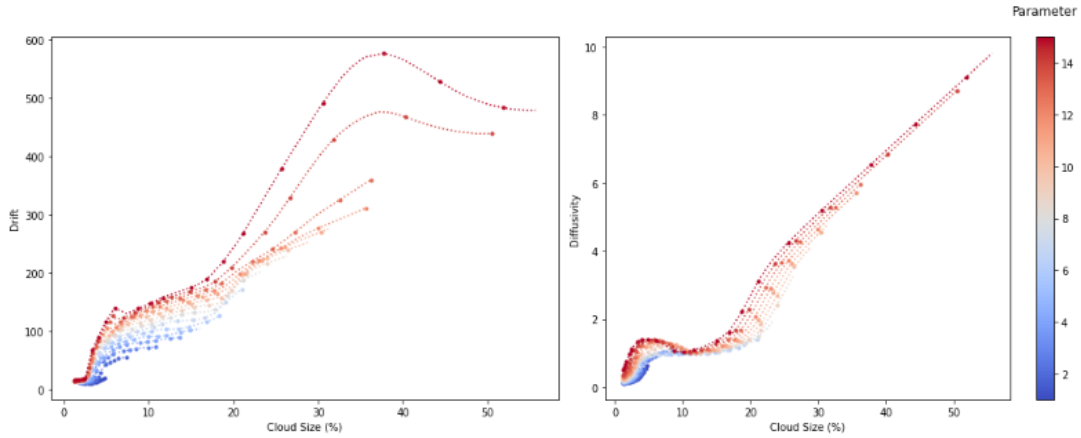
After tuning, we get the best configuration for $(N_{layers}, N_{dim\ per\ layer}, learning\ rate) = (8, 448, 0.001)$. When we train a model based on this hyperparameters with the inverse KDE weights, we get Figure 3.18 .

Table 3.6: Losses after reweighing with HP tuned inverse KDE weights

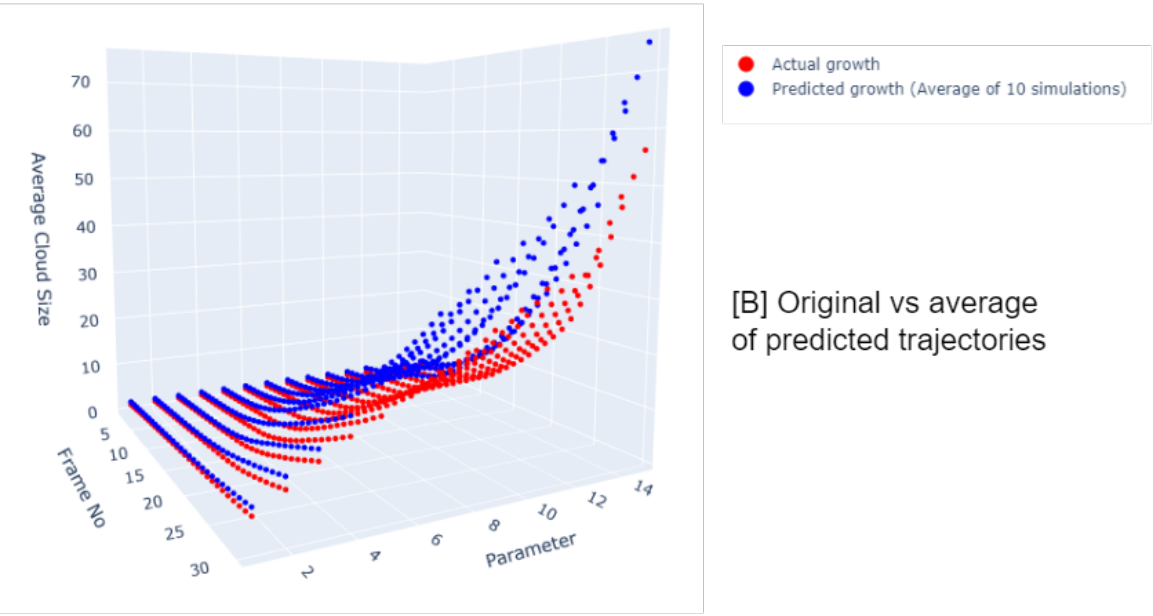
Loss	Data	Value
Training	9 people with 13 simulations each	1.5698
Validation	20% of training data	1.558
Testing	2 people with 13 simulations each	0.1112

However, the MSE in this case was 81.727, which is significantly better than the non HP tuned inverse KDE case. Hence, optimizing the hyperparameters for the weighted case certainly improved performance.

[3] https://keras.io/keras_tuner/



[A] Drift & Diffusivity plots over cloud size



[B] Original vs average of predicted trajectories

Figure 3.18: Various plots for HP tuned Inverse KDE

Comparison between different weights

Finally, let us compare how each of the different weighing strategies affect losses and MSE. We will try to compare it between unweighted, inverse KDE, HP tuned inverse KDE and KDE (passing weights directly out of KDE) as well. We use KDE just to find out what happens. Upon plotting the losses over cloud size plots for each of these configurations, we see the following:

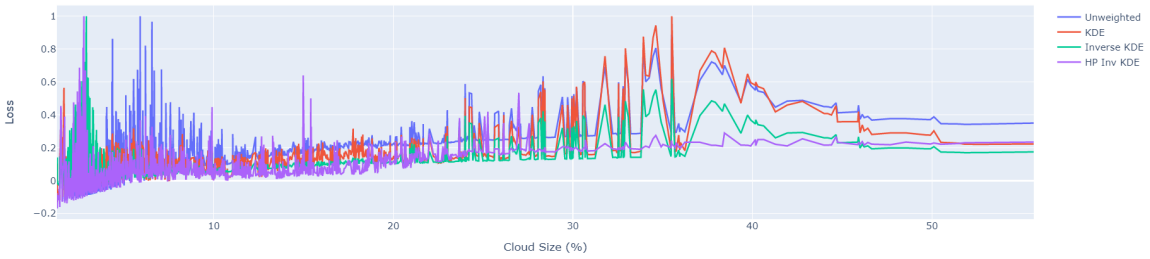


Figure 3.19: Loss vs Cloud Size for various configurations.

Table 3.7: MSE values for different configurations

Configuration	Mean Square Error (MSE)
Unweighted	123.54
KDE	68.45
Inverse KDE	156.363
HP Tuned Inverse KDE	81.727

As we can see in Figure 3.19 , there is quite a lot of fluctuations in the beginning when the model is just starting to learn the trajectories. However, over time (as time progresses, cloud sizes get bigger so going left to right can also be interpreted as over time), the fluctuations start to lessen.

First, when we pass KDE as weights, it means that we are giving more weights to the smaller cloud sizes (which are already abundant in the dataset). Hence, the model should have no problems learning the smaller cloud size growth and as a result, we see very little (the least among the 4) fluctuation among smaller cloud sizes for KDE. However, the loss does start becoming higher (the highest among the 4) once we reach higher cloud sizes.

Next, for inverse KDE, we see that small cloud sizes are a fluctuation because even though they are abundant, we are not assigning much weights to them. So the model cannot train as well as it did for KDE. However for larger cloud sizes, we see noticeable improvement as compared to KDE.

Finally, for the HP tuned inverse KDE, we again see a lot of fluctuation for smaller cloud

sizes (as we are not giving them much importance), however, for larger cloud sizes, we see the least loss values (the least among all 4), which is exactly what we would hope when we give them higher weights.

At last, when we look at the MSE for all the different configurations in Table 3.7, we see that KDE performs the best, counterintuitively! We would expect it to not perform well as we are giving less importance to the larger cloud even more as compared to the unweighted case. However, it may be that the sheer number of data points in the smaller cloud size range ($0 - 10\%$) and the lower losses there maybe enough to compensate the larger deviations in the larger cloud sizes.

4 Conclusion

To summarize, in Section 1 , we looked at the introduction of aerosols, what they are and why it may be important to learn aerosol dynamics. Next in Section 2 , we looked at the current state of the art literature for the various concepts to be used in the project. We looked at the previous work on this topic, then how aerosol modelling is used to estimate the risk of viral exposure, then how to segment and image using U-Net and finally how we can learn Stochastic Differential Equations (SDE) using neural networks. And at last, in Section 3 , we looked at recreating the U-Net model by Rohan, extracting a dataset of cloud sizes, using it to train an SDE neural network and then optimizing it to get a better fit for the trajectories, while addressing the imbalanced dataset problem which exists in the dataset.

The key takeaways from this project are that we can indeed learn the growth of an aerosol cloud using just images and a neural network (albeit in unrealistic conditions). We first generated datasets using Blender simulations with nothing more than random turbulence values. Then we segmented those images to get cloud size dataset and this was passed to the SDE model to learn their growth. This can be used to model how aerosol particles may spread inside a closed space (again, to model viral exposure). Another key takeaway is the attempt to address the imbalanced regression problem, as current solutions to addressing imbalanced datasets are generally for classification and the literature of regression is limited. However, we looked at 2 techniques to potentially address this and they worked with a varying degree of success.

Finally, in the future, this work may be improved by training on real life image data (as compared to the simulations used here) or passing in new input parameters (other than turbulence) such as pose of the person or wind speed. The imbalanced dataset problem could also be improved further to get a much better fit (again the best results we got were from KDE counterintuitively). And we could also develop a better metric to pass for model training. The current loss function (from Equation 2.13 is not a good indicator of how well our model is fitting to the original trajectories. All configurations that we tested this upon had quite low loss values, so if we could come up with something like DICE score for image segmentation, where model is penalized from straying away too much, it would yield better results. As a last idea, we can take a look at the inverse question: can we find the initial parameter values, given a model and a series of images of an aerosol cloud growing?

Bibliography

[1] Paula Branco, Luís Torgo, and Rita P Ribeiro. Smogn: a pre-processing approach for imbalanced regression. In *First international workshop on learning with imbalanced domains: Theory and applications*, pages 36–50. PMLR, 2017.

[2] Sukrant Dhawan and Pratim Biswas. Aerosol dynamics model for estimating the risk from short-range airborne transmission and inhalation of expiratory droplets of sars-cov-2. *Environmental Science & Technology*, 55(13):8987–8999, 2021.

[3] Felix Dietrich, Alexei Makeev, George Kevrekidis, Nikolaos Evangelou, Tom Bertalan, Sebastian Reich, and Ioannis G. Kevrekidis. Learning effective stochastic differential equations from microscopic simulations: Combining stochastic numerics and deep learning, June 2021.

[4] Benjamin Jones, Patrick Sharpe, Christopher Iddon, E. Abigail Hathway, Catherine J. Noakes, and Shaun Fitzgerald. Modelling uncertainty in the relative risk of exposure to the SARS-CoV-2 virus by airborne aerosol transmission in well mixed indoor air. *Building and Environment*, 191:107617, March 2021.

[5] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[6] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018.

[7] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, volume 9351, pages 234–241. Springer International Publishing, Cham, 2015.

[8] Rohan Krishna Saxena. Learning Aerosol Features From Image Data. *TUM Guided Research*, page 28, 2022.

[9] Michael Steininger, Konstantin Kobs, Padraig Davidson, Anna Krause, and Andreas Hotho. Density-based weighting for imbalanced regression. *Machine Learning*, 110(8):2187–2211, 2021.

[10] Ville Vuorinen, Mia Aarnio, Mikko Alava, Ville Alopaeus, Nina Atanasova, Mikko Auvinen, Nallannan Balasubramanian, Hadi Bordbar, Panu Erästö, Rafael Grande, Nick Hayward, Antti Hellsten, Simo Hostikka, Jyrki Hokkanen, Ossi Kaario, Aku Karvinen, Ilkka Kivistö, Marko Korhonen, Risto Kosonen, Janne Kuusela, Sami Lestinen, Erkki Laurila, Heikki J. Nieminen, Petteri Peltonen, Juho Pokki, Antti Puisto, Peter Råback, Henri Salmenjoki, Tarja Sironen, and Monika Österberg. Modelling aerosol transport and virus exposure with numerical simulations in relation to SARS-CoV-2 transmission by inhalation indoors. *Safety Science*, 130:104866, October 2020.