The thing with the Golgi apparatus

Gert-Jan Both

Supervised by:
P. Sens
C. Storm

Technical university of Eindhoven
January-November 2018

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam et turpis gravida, lacinia ante sit amet, sollicitudin erat. Aliquam efficitur vehicula leo sed condimentum. Phasellus lobortis eros vitae rutrum egestas. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Donec at urna imperdiet, vulputate orci eu, sollicitudin leo. Donec nec dui sagittis, malesuada erat eget, vulputate tellus. Nam ullamcorper efficitur iaculis. Mauris eu vehicula nibh. In lectus turpis, tempor at felis a, egestas fermentum massa.

Contents

1

Introduction

2

Introduction

3

Model fitting

In the introduction we stated the question:

How can we fit a model to spatiotemporal data if that model is a PDE?

In this chapter we present the answer in the form of a system which allows us to fit PDE-models to data. We start of with a section describing the general idea and subsequent sections elaborate on each step.

The system principally works for any type of data and model, but was developed originally to analyze data from the RUSH experiments. We illustrate our system with several examples and have chosen to use real data from the RUSH experiments (specifically, MANII) over synthetic data. In figure fig. 3.0.1 we show four typical frames of the MANII movie. Characteristically, MANII moves from the ER to the Golgi and remains there. The upper left frame shows the first

frame with all the cargo in the ER, in the lower left and upper right the intermediate regime is shown (i.e. cargo moving) and in the lower right frame all the cargo has moved into the Golgi.

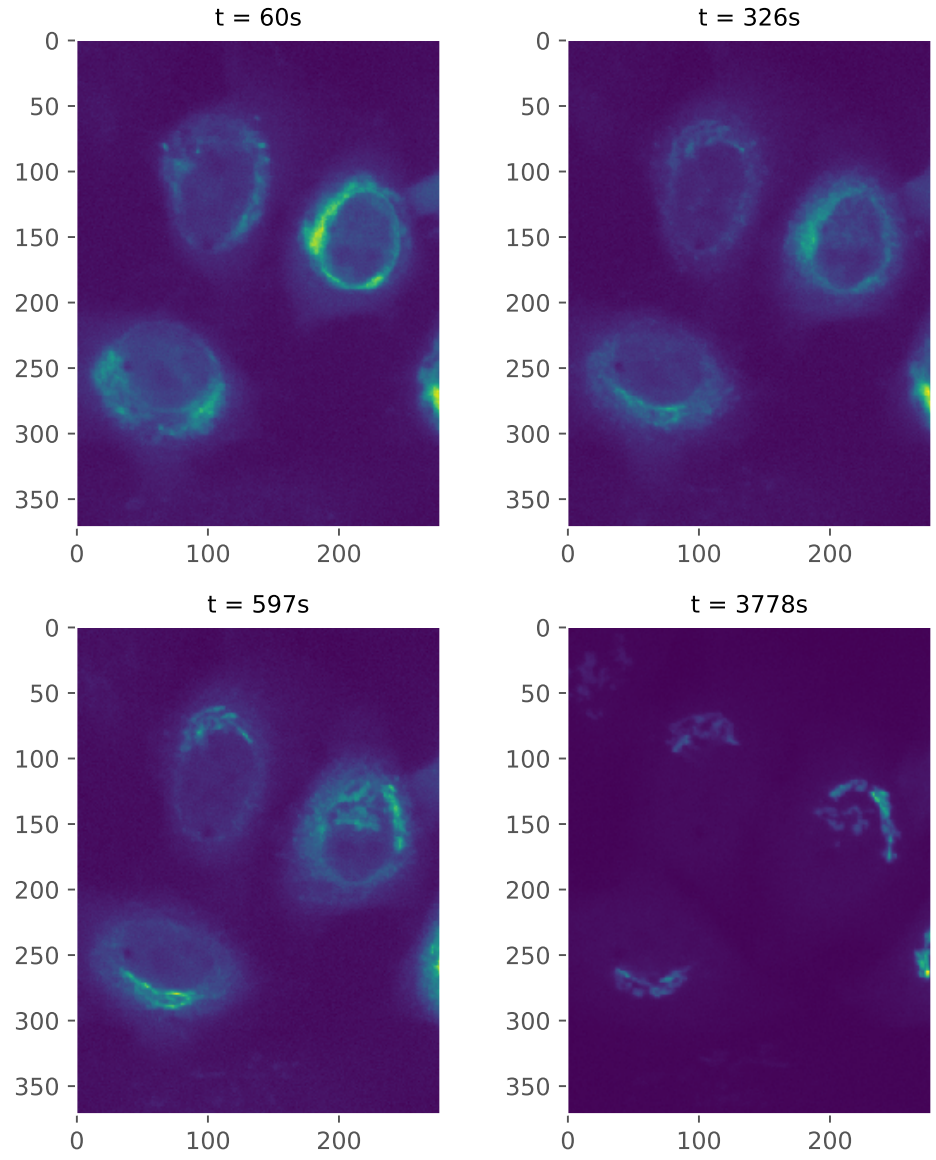


Figure 3.0.1: Four frames (t = , ,) showing a typical MANII cycle of the RUSH experiment.

3.1 The concept

Assume we have some experimental data describing some process $f(x, t)$. We have also developed a model for this process in the form of a PDE:

$$\partial_t f(x,t) = \lambda_1 \nabla^2 f(x,t) + \lambda_2 \nabla f(x,t) + \lambda_3 f(x,t) + \lambda_4 \tag{3.1}$$

We now wish to know if this model fits the data $f(x, t)$ and the value of coefficients λ_i. To do so, consider each spatial term of $f(x, t)$ in eq. 3.1 as some variable x_i and $\partial_t f$ as y, so that we can rewrite it as:

$$y = \lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3 + \lambda_4$$

If we have access to the variables x_i and y, we can perform least squares fitting to obtain the coefficients λ_i. In other words, if we can calculate the spatial and temporal derivatives, we can fit the model to the data. Once we know the coefficients, we can use our model to propagate the first frame of the data and compare our model to the experimental data.

Although the concept seems trivial, its implementation is all but. Data is rarely noiseless and obtaining accurate derivatives from noisy data is notoriously hard. We can also have different models in different areas of the data, so that we need to segment the data or the coefficients λ_i might not be constant but space- and time-dependent. The process of fitting the data thus has several steps:

1. Denoising and smoothing
2. Calculating derivatives
3. Segmenting
4. Fitting

In the next sections, we describe each step separately. Note that the method we present here has been developed empirically: there's no theoretical background

as to why this particular combination should work optimally. Instead, it's been developed while analyzing the data, adapting each step on the go.

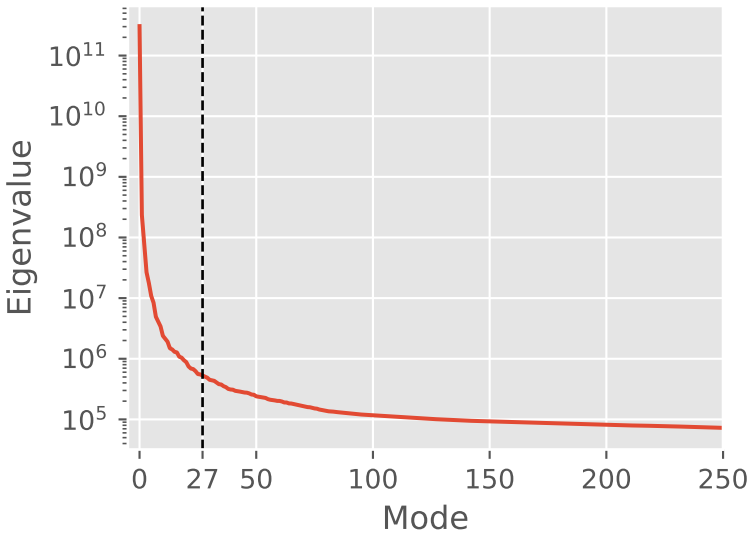
3.2 Step 1 - Smoothing and denoising

The first step in our pipeline is to denoise and smooth the data. The smoothing is necessary for accurately calculating the derivatives. Denoising still is a very active area of research (especially in life sciences) and dozens of different methods exist[1]. For example, one could Fourier transform the signal and use a high pass filter, but this would also get rid of small and sharp features. After evaluating several methods, we have settled on the so-called 'WavInPOD' method, introduced by 3 in 2016. In 4 they show that this methods outperforms several other advanced methods. WavInPOD combines Proper Orthogonal Decomposition (POD) with Wavelet filtering (Wav). Both subjects are vast (especially Wavelet transform) and we're only interested in the result of the technique, so we only present a short introduction here, adapted from 3.

POD is a so-called dimensionality reduction technique and is very closely related to the more well-known Principal Component Analysis (PCA) in statistics. In physics it's ofen used to analyze turbulent flows[5]. In POD we wish to describe a function as a sum over its variables:

$$f(x,t) = \sum_{n}^{r} a_n(x)\varphi_n(t)$$

where a_n and φ_n are called respectively the spatial and temporal modes. Associated with each mode n is an energy-like quantity E_n. Modes with a higher 'energy' E_n contribute more to the signal f than modes with a lower energy and we can thus approximate the signal by selecting the k modes with the highest energy. A typical log10 energy spectrum has a 'knee' in the values, as shown in figure fig. **??**. Modes with an energy below the knee are noise, and modes above signal.



The wavelet transform is very similar to the Fourier transform, but uses wavelets as its basis. A fourier transform gives the frequency domain with infinite precision, but tells nothing about the locality of the frequencies (.i.e when each frequency is present in a signal). By using a wavelet (a wave whose amplitude is only non-zero for a finite time), we sacrifice precision in the frequency domain but gain information on the locality instead. Performing a wavelet transform transforms the signal into the sum of an approximation and its details and we can filter this analogous to a fourier filter. Due to its locality however, noise is filtered out, by sharpness is retained.

WavinPOD combines these two techniques by applying wavelet filtering to the POD modes. In detail, one first decomposes the problem with a POD transformation. The energy spectrum of this transformation is shown in figure fig. **??** and we select a cutoff of 27. All retained modes are wavelet filtered and are then retransformed to give the denoised and smoothed signal. In figure fig. 3.2.1 we show the results of the smoothing in the time and spatial domain. In the left panel we show the signal of a single pixel in time, while we plot a line of pixels in a single frame in the right panel. The red lines denote the original (unfiltered) signal, the blue line the effect of just applying a POD and the black one the result

of the WavInPOD technique. Note that the effect of the wavelet filtering is to smooth the signal significantly and in comparing the original data to the filtered data that we've retained the sharpness of the features whilst obtaining a smooth signal.

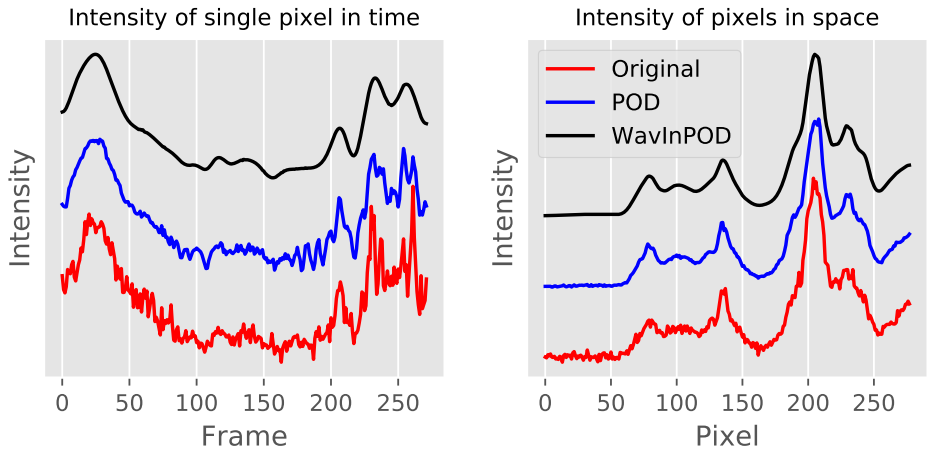


Figure 3.2.1: Effect of POD with a cutoff of 27 and wavelet filtering with a level 3 db4 wavelet. Left panel shows the result in the time domain, right panel in the spatial domain. Lines have been offset for clarity.

3.3 Step 2 - Derivatives

After having denoised the images, we calculate the spatial and temporal derivatives. Obtaining correct numerical derivatives is hard and becomes much more so in the presence of noise[7]. Next to a finite-difference scheme, one can for example (locally) fit a polynomial and take its derivative or use a so-called tikhonov-regularizer[8]. The computational cost of these methods is high however and don't scale well to dimensions higher than one. For our spatial derivatives these methods are thus not available. In fact, obtaining the gradient of a 2D discrete grid has another subtlety which we need to adress.

Naively, one could obtain the gradient of a 2D grid by taking the derivative using a finite difference scheme with respect to the first and second axis. If there are

features on the scale of the discretization ($\sim$ few pixels), such an operation will lead to artifacts and underestimate the gradient. These issues have long been known and several techniques have been developed to accurately calculate the gradient of an 'image'. The most-used one is the so-called Sobel operator. It belongs to a set of operations known as *kernel operators*. Kernel operators are expressed as a matrix and by convolving this matrix with the matrix on which the operation is to be performed, we obtain the result of the operator. We show this for the Sobel operator.

Consider a basic central finite difference scheme:

$$\frac{df}{dx} \approx \frac{f(x_{i+1}) - f(x_{i-1})}{2h}$$

where h is defined as $x_{i+1} - x_i$. In terms of a kernel operator, we rewrite this as:

$$\frac{1}{2} \cdot \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

where we have set $h = 1$, as the distance between pixels is one by definition. By convolving this matrix with the matrix A we obtain the derivative of A:

$$\partial_x A \approx A * \frac{1}{2} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

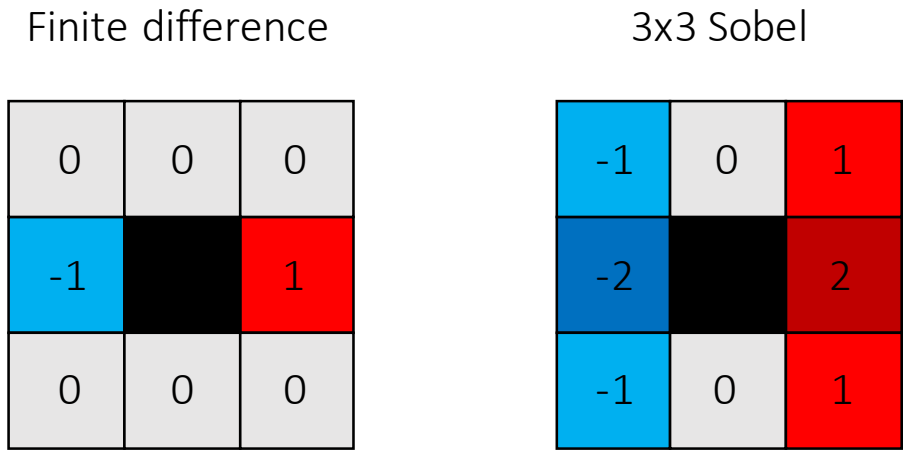


Figure 3.3.1: Left panel: One dimensional finite difference kernel. **Right panel:** Three by three Sobel filter

As stated, this operation is inaccurate and introduces artifacts. To improve this, we wish to include the pixels on the diagonal of the pixel we're performing the operation on as well (see figure fig. 3.3.1). The distance between the diagonal pixels and the center pixel is not 1 but $\sqrt{2}$ and the diagonal gradient also needs to be decomposed into $\hat{x}$ and $\hat{y}$, introducing another factor $\sqrt{2}$. The kernel thus obtained is the classic 3×3 Sobel filter:

$$\mathbf{G}_x = \frac{1}{8} \cdot \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Increasing the size of the Sobel filter increases its accuracy and we've implemented a 5x5 operator. Implementing the derivative operation as a kernel method is also beneficial from a computational standpoint, as convolutional operations are very efficient.

As the time derivative involves just one dimension, we make use of a second order accurate central difference scheme implemented in Numpy.

3.4 Step 3 - Segmentation

In the case of the RUSH data, obtained images and movies often contain multiple cells. Each of these cells can be further segmented into two more areas of interest: the cytoplasm, which is were we want to fit our model and the Golgi apparatus. We wish to make a mask which allows us to separate the cells from the backgroud and themselves and divide each cell into cytoplasm or Golgi.

Figure fig. 3.0.1 shows four typical frames in the MANII transport cycle. Note that no sharp edges can be observed, especially once the MANII localizes in the Golgi. No bright field images were available as well, together making use of techniques such as described in 9 unavailable. We have thus developed two methods which allow us to segment the image and the cells, based on Voronoi diagrams and the intensity.

3.4.1 Voronoi diagram

Consider again the frame on the left of figure fig. 3.0.1. Note that in early frames such as this one, the cargo (i.e. fluorescence) is spread circumnuclear. Applying a simple intensity based segmentation gives us a number of separate areas, which *very* roughly correspond to a cell. We can then pinpoint each cells' respective center. Given n points, Voronoi tesselation divides the frame into n areas, where point i is the closest point for each position in area A_i. The hidden assumption here is thus that each pixel belongs to the cell center it's closest too. Although this is a very big assumption, in practice we've found this to be reasonable. Furthermore, one can add 'empty' points to make the diagram match observations. Assuming small movements of the cell, this isn't an issue either for this technique, as we are assigning an area to each cell instead of very precisely bounding it. This also allows us to calculate the Voronoi diagram in the early frames and apply the segmentation to the entire movie. The result of this segmentation for MANII is shown in figure fig. 3.4.1. Each cell centre is denoted

by a dot, while the lines denote the border between each voronoi cell.

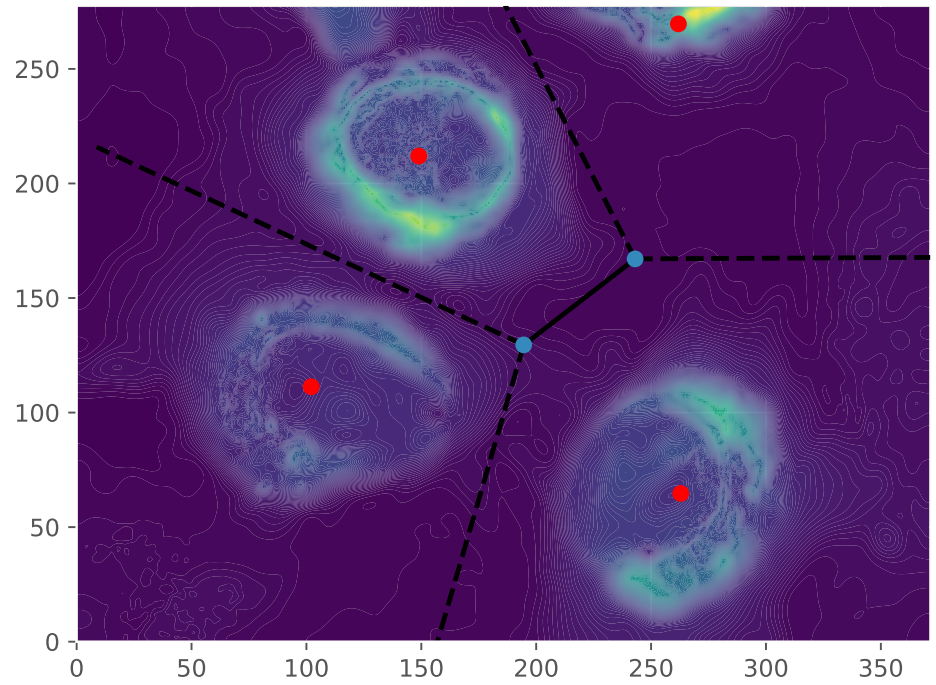


Figure 3.4.1: The obtained mask. Red dots are cell centers, dashed lines infinite edges and solid lines finite edges.

3.4.2 Intensity

The Voronoi technique works very well for an area-based approach, but for analyzing our fitting data we would like a more precise mask - although we still don't require pixel-level accuracy. From the movies, the Golgi is clearly visible and we can separate the cytoplasm from the background, with a big 'gray' area inbetween. We thus turn to an intensity based approach. We have developed the following approach for localizing the Golgi:

1. Normalize the intensity I between 0 and 1.

2. Sum all the frames in time: $\sum_n I(x, y, t_n)$. A typical result is shown in figure fig. 3.4.2 .
3. Threshold the image to obtain the mask. This is either done automatically through an Otsu threshold or by manually adjusting the threshold until desired result.
4. The mask is postprocessed by filling any potential holes inside the mask.

This procedure was unable to reliably separate the background from the cytoplasm. Noting that while the cytoplasm might not have the intensity as the golgi, its time derivative should be higher than the rest of the areas. We replace step two by $\log_{10}\left(\sum_n I(x, y, t_n) \cdot \partial_t I(x, y, t_n)\right)$, where the time derivative has been normalized between 0 and 1. Figure fig. 3.4.2 shows our final results. The upper two panels show the images obtained after performing the summing operation for the Golgi and cytoplasm respectively, while the lower left panel shows the final mask obtained after thresholding these two images. For comparison, we plot frame to compare the mask to.

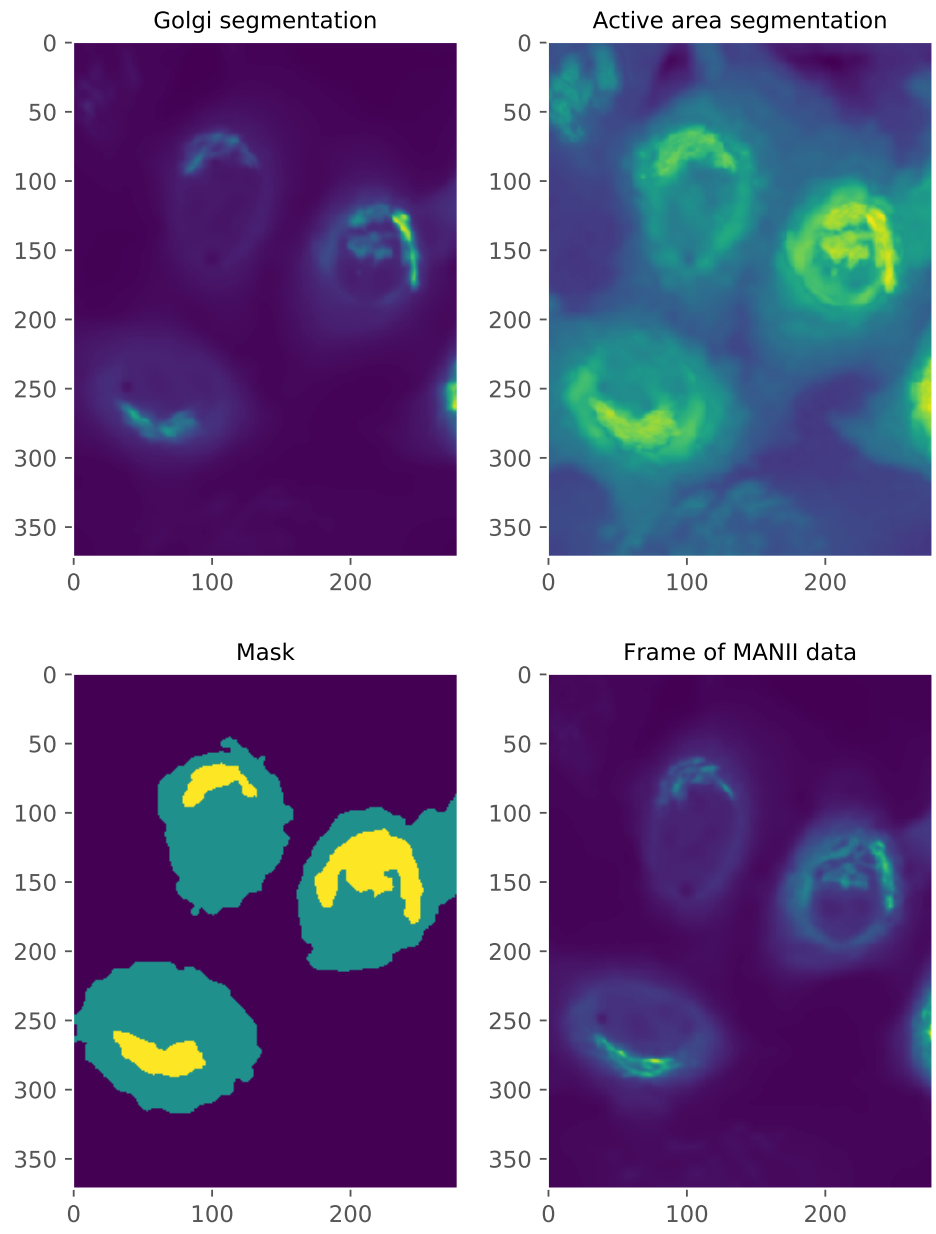


Figure 3.4.2: Four panels showing the different stages of making the mask. From segmenting the upper two panels we determine the golgi and active area, leading to the mask in the lower left. Compare the to the lower right.

3.5 Step 4 - Fitting

Having gathered all the data, the final step is to fit the model. Equation eq. 3.1 assumes a model with constant coefficients. In reality, coefficients will be spatially and even temporally varying. To circumvent this issue, we assume the coefficients can be assumed to be locally constant. We thus assume that for a small area we can fit the model using constant coefficients. We perform this operation for every datapoint in a sliding-window manner, as shown in figure fig. 3.5.1, thus ending up with a coefficient field.

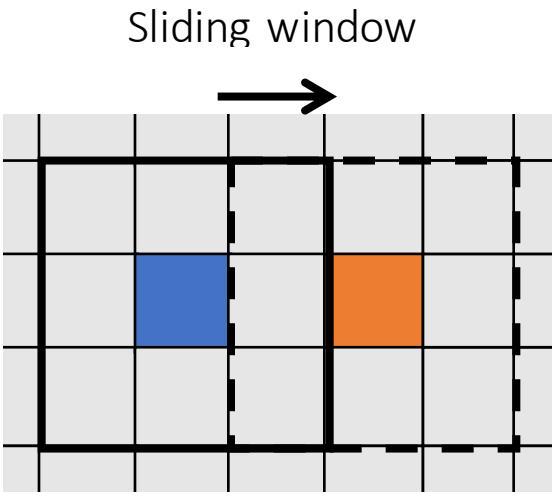


Figure 3.5.1: Schematic overview of the sliding window technique. The solid black line encompasses an area around its blue coloured central pixel and the fit output is assigned to that pixel. We then move the window (dashed black line) and perform the fit for the orange coloured pixel.

4

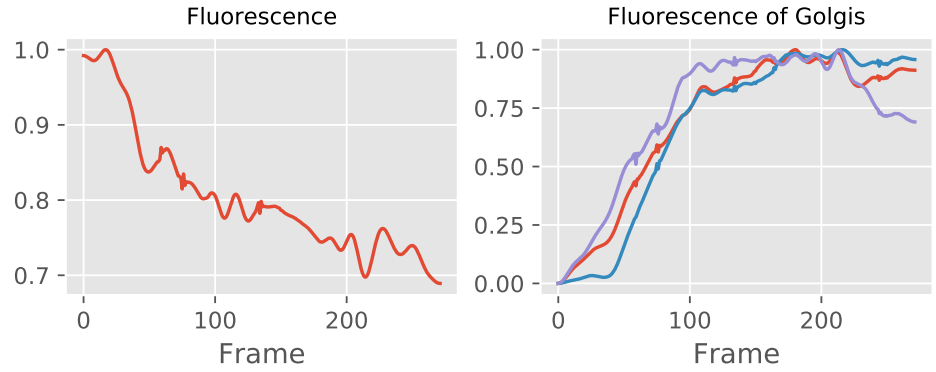
Data analysis

In the previous chapter we developed a general system to fit our advection-diffusion model to the data obtained from the RUSH experiments. In this chapter we apply said system to the MANII movies. MANII has been chosen as after arriving at the Golgi, it remains there. Dealing with only one transport direction makes everything a lot easier for us. This chapter consists of five sections:

- **Section 1** presents a general analysis of the fluorescence data of the movies
- In **section 2** we discuss the time derivative of the fluorescence, $\partial_t I$, as this also has some interesting points.
- **Section 3** contains our analysis of the least squares fit.
- **Section 4** is similar to section 3, only now we analyse a constrained fit with $D > 0$.

- **Section 5** summarizes our findings and presents possible improvements for the system.

4.1 Analysis of fluorescence



4.2 Analysis of time derivative

4.3 Analysis of LS-fit

4.3.1 Diffusion

4.3.2 Advection

4.4 Analysis of constrained LS-fit

4.5 Conclusion

Bayesian heeeeee

5

Physics Informed Neural Networks

In the previous chapters we showed the difficulties in fitting a model in the form of a partial differential equation to spatio-temporal data. The method we developed was a classical numerical approach, separating the problem into several substeps such as denoising, smoothing and numerical differentiating. In the last few years machine learning has been slowly making its way into physics. Very recently, a technique generally referred to as Physics Informed Neural Networks (PINNs) have shown great promise as both tools for simulation and model fitting (10, 11, 12, 13, 14). In this chapter, I will evaluate the use of this technique to fit the model to the RUSH data. I've divided the chapter into three parts:

- **Neural Networks** - This part will cover the basics of neural networks: their inner workings, how to train them and other general features.

- **Physics Informed Neural networks** - In this second part we introduce the concept behind PINNs, use it to solve a toy problem and apply it to our RUSH data.
- **Conclusion** - Finally we summarize the results and observations from the previous sections.

5.1 Neural Networks

Artificial Neural Networks (ANNs) are networks inspired by biological neural networks. Contrary to other ways of computing, ANNs are not specifically programmed for a task - instead, ANNs are *trained* using a set of data. Research on artificial neural networks started in the '40s but never gained any critical mass, as no efficient training algorithm was known. Once an efficient training algorithm was found in 1975 by Werbos, interest resurged but it wasn't until the late '00s that deep learning started gaining widespread traction. The use of GPU's allowed ANNs to be efficiently trained and widely deployed at reasonable cost.

The advancements in machine learning in general and especially neural networks in the last ten years have yielded a wealth of techniques and approaches. In supervised learning, the network is given pre-labeled data so that it is trained by learning the mapping from the given inputs to the given outputs. Other types such as supervised learning, where the network needs to learns to discriminate between unlabeled data, and reinforcement learning don't have any obvious use for PINNs yet and I've thus chosen to omit them. In the next sections, I'll present the mathematics of an ANN and show how they are trained using the so-called *backpropagation* algorithm.

5.1.1 Architecture

An excellent introduction is given by Michael Nielsen in his freely available book "Neural networks and deep learning." The following section has been strongly inspired

by his presentation.

At the basis of each neural network lies the neuron. It transforms several inputs non-linearly into an output and we can use several neurons in parallel to create a *layer*. In turn, we several layers in series make up a network. The layers in the middle of the network are known as *hidden layers*, as shown in figure fig. 5.1.1

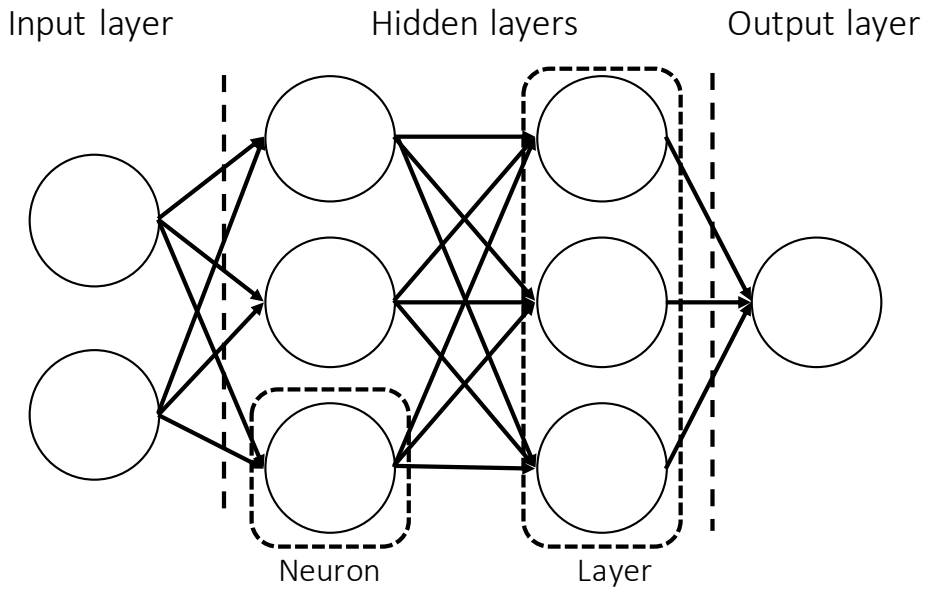


Figure 5.1.1: Schematic view of a neural network.

In the schematic shown in fig. 5.1.1, each neuron is connected to every neuron of the previous and next layer. This is known as a *fully connected* layer. Using only this type of layers, we've created a feed-forward network and it has been proven that a single hidden layer with enough neurons is a *universal function approximator*, i.e. a neural network can represent any continuous function using enough neurons.

As stated, a neuron takes several inputs and transforms them into an output. This is a two step process, where in the first step the neuron multiplies the input vector

$\mathbf{x}$ with a weight vector w and adds a bias b:

$$z = w\mathbf{x} + b \tag{5.1}$$

z is called the weighted input and is transformed in the second step by the neuron *activation function* σ. This in turn gives the output of the neuron a, also known as the activation:

$$a = \sigma(z) = \sigma(w\mathbf{x} + b) \tag{5.2}$$

The role of the activation function is to introduce non-linearity into the system. The classical and often used activation function is the *tanh*, as it is bounded between +1 and -1. Since we're working with multiple layers, it is useful to rewrite function eq. 5.2 in terms of the activation a^l of layer l:

$$a^l = \sigma(z^l) = \sigma(w^l a^{l-1} + b^l)$$

where w^l and b^l are respectively the weight matrix and bias of layer l.

5.1.2 Training

In supervised learning the task of training a machine means adjusting the weights and biases until the neural network predictions match the desired outputs. We thus need some sort of metric to define this 'distance' between prediction and desired output. Training the network than means minimizing the metric with respect to the weights and biases of the network. This metric is known as the cost function $\mathcal{L}$ and the most used form is a mean squared error:

$$\mathcal{L} = \frac{1}{2n} \sum_i |y_i - a_i^L|^2 \tag{5.3}$$

where n is the number of samples, y_i the desired output of sample i and a_i^L the activation of the last function - the prediction of the network. Minimizing this is not trivial, as the problem can have many local minima. A solution can be found however using gradient descent techniques.

Gradient descent techniques are based on the fact that given an initial position, the fastest way to reach the minimum from that position is by following the steepest gradient. Thus, given a function $f(\mathbf{x})$ to minimize w.r.t to $\mathbf{x}$, we guess an initial position x_n and iteratively change until it convergences:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma \nabla f(\mathbf{x}_n)$$

where γ is known as the learning rate. If a global minimum exists, this technique will converge on it. More advanced versions of this technique exist which are able to deal with local minima as well, since convexity of the cost function is not at all guaranteed.

Making use of gradient descent requires knowledge of the derivatives of the cost function w.r.t to the variables to be optimized. In the case of neural networks, we thus need to know the derivative w.r.t to each weight and bias. A naive finite difference scheme would quickly grow computationally untractable for even shallow networks. A solution to this problem was found by Werbos in the form of the backpropagation algorithm. Despite many years of ongoing research, it is still the go-to algorithm for each neural network implementation.

Back propagation and automatic differentiation

As we wish to minimize the cost function w.r.t. to each weight w and bias b using gradient descent, we need to find the derivative of the cost function w.r.t to each. Our argument simplifies if we move away from vector notation and introduce w_{jk}^l, the weight of the j-th neuron in layer $l-1$ to neuron k in layer l and b_j^l, the bias of the neuron j in the l-th layer. We introduce the error of neuron j in layer l as:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

We can rewrite this using the chain rule as:

$$\delta_j^l = \sum_k \frac{\partial C}{\partial a_{jk}^l} \frac{\partial a_{jk}^l}{\partial z_j^l}$$

However, the second term is always zero except when $j = k$, so the summation can be dropped. Remembering eq. 5.2, we note that $\partial a_{jk}^l / \partial z_j^l = \sigma'(z_j^l)$. For the last layer $l = L$, the first term turns into the derivative of the cost function, finally giving us:

$$\delta_j^L = |a_j^L - y_j| \sigma'(z_j^L) \tag{5.4}$$

Equation eq. 5.4 relates the error in the output layer to its inputs. This in turn is a function of all the previous inputs and errors and we thus need to find an expression relating the error in layer l with the error in an layer $l + 1$. Since we have an expression for the error in the last layer, we propagate the error going down the layers, hence the name *back*propagation. Again using the chain rule gives:

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_{jk}^{l+1}} \frac{\partial z_{jk}^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_{jk}^{l+1}}{\partial z_j^l}$$

Using equation eq. 5.1, we obtain after substitution:

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l) \tag{5.5}$$

Using equations eq. 5.4 and eq. 5.5 , we can calculate the error in C due to each neuron. Finally, we need to relate the error in each error to $\partial C / \partial w_{jk}^l$ and $\partial C / \partial b_j^l$.

Making use yet again gives us the last two backpropagation relations:

$$\frac{\partial C}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{5.6}$$

$$\sum_k \frac{\partial C}{\partial w_{jk}^l}\frac{\partial w_{jk}^l}{\partial z_j^l} = \delta_j^l \rightarrow \frac{\partial C}{\partial w_{jk}^l} = a_j^{l-1}\delta_j^l \tag{5.7}$$

Now that we now that all back propagation equations, we state the algorithm. It consists of four steps:

1. Complete a forward pass, i.e., calculate the expected outcomes with the current weights and biases.
2. Calculate the error using eq. 5.4 and do a backward pass to obtain the error in each neuron using eq. 5.5. This can be used to calculate the gradients using eq. 5.6 and eq. 5.7
3. Adjust the weights and biases using the choosen optimizer (e.g. gradient descent)
4. Return to step 1 until the optimization problem converges.

Mathematically, back propagation is a special case of a technique known as automatic differentiation. Automatic differentiation is a third type of differentiation, next to numeric and symbolic. It allows for machine precision calculation of derivatives by writing it as a chain of simple operations combined with the chain rule, similar to backpropagation. Note that:

$$\delta_j^\circ = \frac{\partial C}{\partial x_j}\frac{\partial x_j}{\partial z_j^\circ}$$

so that:

$$\frac{\partial C}{\partial x_j} = a_j^\circ \delta_j^\circ$$

Thus neural networks also give us access to high precision derivatives with regard to each coordinate.

5.2 Physics Informed Neural Networks

On the face of things, the goal of physics and neural networks are oppsite: whereas physics tries to build an understanding of things using models to make predictions, neural networks learn a *modelless* mapping to make predictions. Recent advancements however have merged the two approaches together in a concept known as Physics Informed Neural Networks (14, 13). In this approach, we encode physical laws into the network, so that the network respects the physics. This can be used to both numerically solve equations or fit a model to spatiotemporal data. Even more so, it should allow us to infer coefficient fields.

5.2.1 The concept

Consider a set of 1D+1 spatiotemporal data, consisting of some property $u(x,t)$ at coordinates (x,t). The neural network can be learned the underlying physics by minimizing the cost function:

$$\mathcal{L} = \frac{1}{2n}\sum_i |u_i - a_i^L|^2$$

The process of learning requires a lot of data and is prone to overfitting. Now assume that we know that $u(x,t)$ is governed by some process which is written as a partial differential equation:

$$\partial_t u = f(1, u, u_x, u_x x, u^2, \ldots)$$

where f is a function of u or its spatial derivatives. Rewriting it as:

$$g = 0 = -\partial_t u + f(1, u, u_x, u_x x, u^2, \ldots) \tag{5.8}$$

we see that in order to satisfy the PDE, $g \to 0$. The idea of PINNs is to add this function g to the costfunction of the neural network:

$$\mathcal{L} = \frac{1}{2n} \sum_i |u_i - a_i^L|^2 + \lambda \sum_i |g_i|^2 = MSE + \lambda PI$$

where λ sets the effective strength of the two terms. Observe that the cost function is higher if the PDE is not satisfied. Minimizing the costfunction will thus mean minimizing g and hence satisfying the PDE. We effectively penalize solutions not satisfying the physics we put in equation eq. 5.8; the added term acts a 'physics-regularizer'. Concretely, the adding of physics contrains the solution space, preventing overfitting and making the neural network much more data efficient. The most useful feature however is that we don't need a vast set of training data to train the network, as we solve the problem *by* training the network.

We can also remove the mean squared error term from the cost function and add initial and boundary conditions, similar to the PI term. If we now train the network, it will learn the solution to the given PDE whilst respecting the given boundary and initial conditions. This alternative means of numerically solving a model doesn't need advanced meshing of the problem domain or carefully constructed (unstable) discretization schemes, as it requires the physics to be fullfilled at every point in the spatiotemporal domain. A useful analogy here is calculating the trajectory of a launched object. A classical numerical solver would take small steps in time, updating the position and speed of the object each step. A PINN however uses a completely different approach. Given the initial (random) state of the neural network, it calculates a first trajectory and keeps adjusting the weights of the network until the cost is minimized, i.e. until we obtain a solution

satisfying the included physics and initial and boundary conditions. A classical numerical approach tries once using a correct and methodical approach, whereas a PINN tries many times until the result satisfies its constraints.

We can also use this framework to fit models to spatiotemporal data by letting the coefficient of each term be a variable to be minimized as well. More concretely, where before the cost was a function of the weights and biases, $\mathcal{L} = f(w^l, b^l)$, we now let it be a function of the coefficients λ of the PDE as well: $\mathcal{L} = f(w^l, b^l, \lambda_1, \lambda_2, \ldots)$. This is shown for several PDEs such as the Burgers, Schrodinger or Navier-Stokes equation in the papers of M. Raissi(14, 13). In the case of the Navier-Stokes equation, it's shown that it's also possible to infer the pressure field, which appears as a separate term. This is achieved by adding another output neuron to the PINN (shown in figure fig. 5.2.1), so that it predicts both the pressure and the flow. In theory it should also be possible to infer spatially and temporally varying *coefficient* fields. We investigate this claim in the next section.

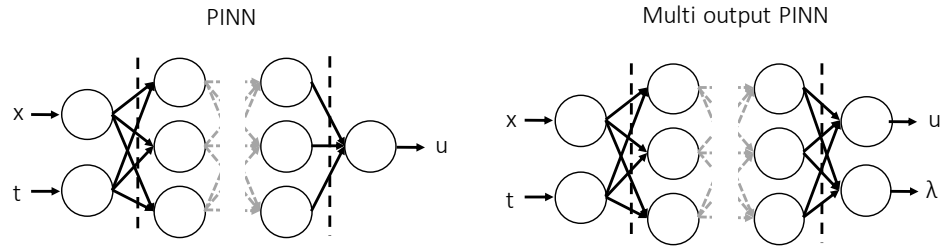


Figure 5.2.1: Left panel: a normal single output PINN. Right panel: a multi-output PINN. The network now also predicts the coefficients values at each data point.

5.2.2 PINNs in practice

We now wish to evaluate the use of PINNs to analyze the RUSH data. Using a diffusive process as a toy problem, we first show how PINNs are able to accurately determine the diffusion constant, even in the presence of noise. Next,

we prove that PINNs are indeed capable of inferring coefficient fields and finish by analyzing some parts of the RUSH data.

In our toy problem we have an initial concentration profile:

$$c(x, 0) = e^{-\frac{(x-0.5)^2}{0.02}}$$

diffusing in a 1D box according to:

$$\frac{\partial c(x,t)}{\partial t} = \nabla \cdot [D(x)\nabla c(x,t)]$$

on the spatial domain $[0, 1]$ with perfectly absorbing boundaries at the edges of the domain:

$$c(0, t) = c(1, t) = 0$$

If $D(x) = D$, this problem has an analytical solution through a Greens function. If the diffusion coefficient is spatially dependent though, the problem needs to be solved numerically. The code used to generate our data can be found in the appendix. Although this toy problem is simple and in 1D, our results easily generalize to higher dimenions and complexity at the cost of higher computational cost.

5.2.2.1 Constant diffusion coefficient

We now consider the mentioned problem with a diffusion coefficient of $D(x) = D_0 = 0.1$ and simulate it between $t = 0$ and $t = 0.5$. Using a spatial and temporal resolution of 0.01, we have a datagrid of 101 by 51, so that our total dataset consists of 5151 samples. The neural network consists of 6 hidden layers of 20 neurons each and $\lambda = 1$. Figure fig. 5.2.2 shows the ground truth for the problem and the absolute error of the neural network.

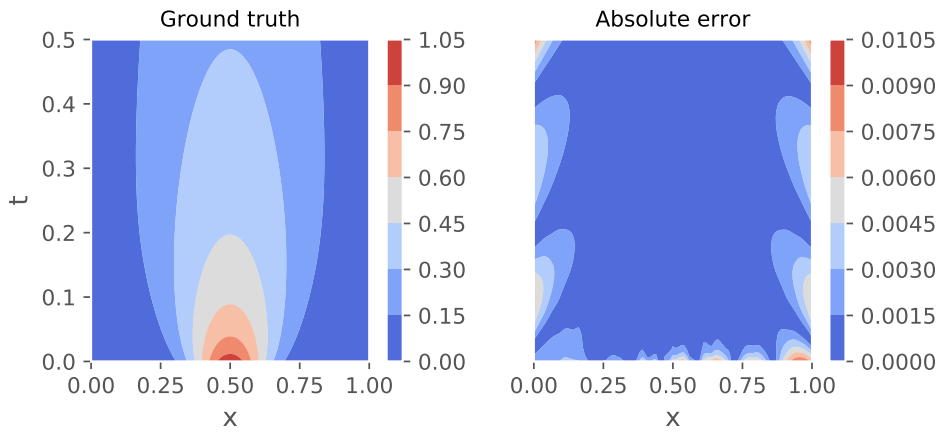


Figure 5.2.2: Left panel: Simulated ground truth of the problem. **Right panel**: The absolute error of neural network. Note that most of the error is located at areas with low concentration, i.e. signal.

The predicted diffusion coefficient is $D_{pred} = 0.100026$, giving an error of 0.026%. In 14, the authors obtain similar accuracies for significantly more complex problems such as the Schrodinger equation, which means that our accurate inference is not just due to the simplicity of the problem. Furthermore, Raissi et al. show that the result is robust w.r.t the architecture of the network. From the absolute error we observe that the error seems to be higher in areas with low concentration. This is a feature we've consistently observed: in areas with low 'signal', the neural network seems to struggle.

As good as these results are, the input data is noiseless and thus of limited practical interest. We now show that PINNs perform equally well with noisy data by adding 5% white noise to the data and performing the same procedure. The network is now doing two tasks in parallel: it's both denoising the data and performing a model fit. In the left panel of figure fig. 5.2.3 we show the concentration profile at times $t = 0, 0.1$ and 0.5, with the prediction of the PINN superimposed in black dashed lines at each time. On the right panel we show again the absolute error from the ground truth. Observe the similarities with the noiseless case: most of the error localizes in areas wit low concentration.

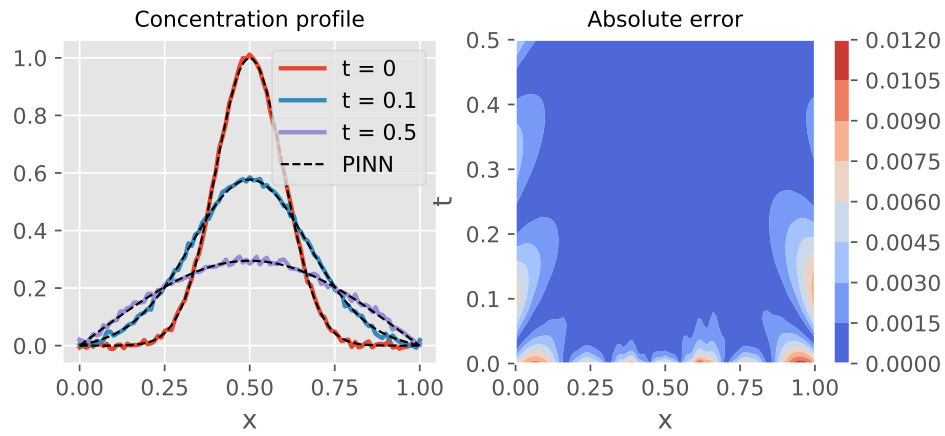


Figure 5.2.3: Left panel: The original noisy concentration profile with the neural network inferred denoised version super imposed. **Right panel**: The absolute error of neural network with respect to the ground truth. Note that most of the error is located at areas with low concentration.

The inferred diffusion constant is $D_0 = 0.10052$, giving an error of 0.52%. Although the error is slightly higher than in the noiseless version, it's extremely impressive that we obtain the diffusion constant to this precision.

5.2.2.2 Varying D

As stated, it should be possible to infer coefficient fields by using a two output neural network. One output predicts the concentration while the other predicts the diffusion coefficient. Such a network is indeed capable of generating the right coefficient field as shown in figure fig. 5.2.4 . Here the network has been trained on the constant diffusion coefficient data we used before including 5% white noise, so that we should observe a diffusion field constant at $D(x,t) = D_0 = 0.1$. In the upper left we show the data on which the network is trained, with the upper right panel the predicted concentration profile, which shows a very good match. In the lower right panel we show the inferred diffusion field. We observe a good match in the middle of the plot, but the neural network again struggles in areas with low concentration, such as the lower left and right area. A more quantitative

analysis of the predicted diffusion and concentration is presented in the lower left corner. Here we plot the Cumulative Distribution Function (CDF) of the absolute relative error. Note that the PINN predicts the concentration very well, but struggles more with the diffusion coefficient. This is expected, as the mean squared error of the cost function is quite explicit in its use of the concentration, whereas the diffusion coefficient is determined self-consistently in the PI part. We also observed similar but distinctive results in different runs, owing to the non-convexity of the problem. Overall the result is still remarkable, given that we've inferred a diffusion field from just concentration data with 5% noise.

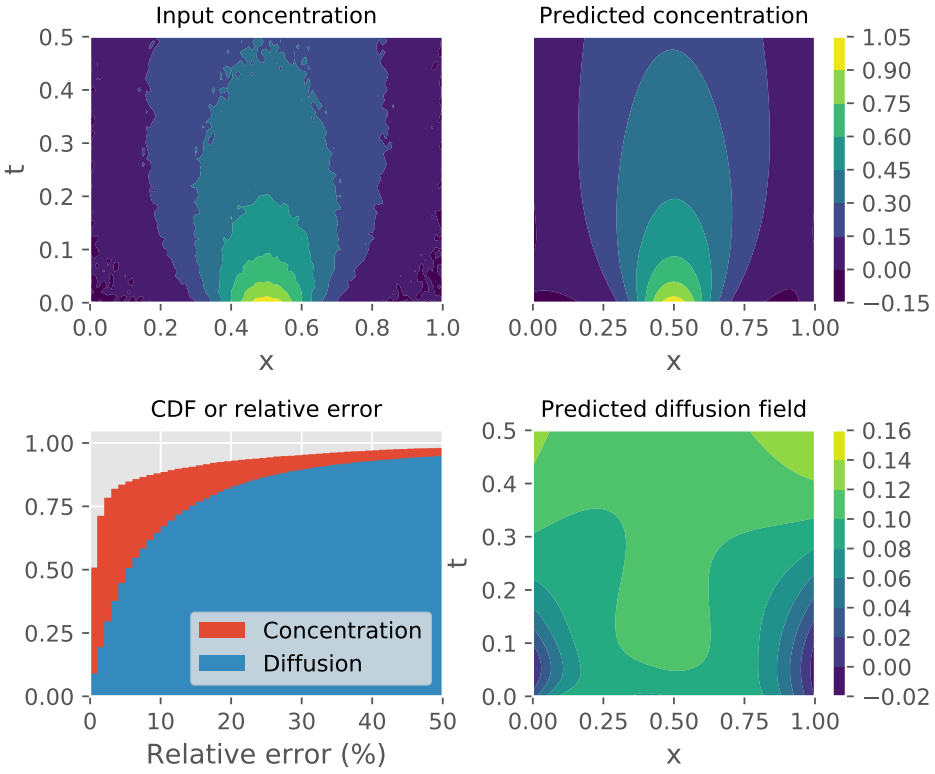
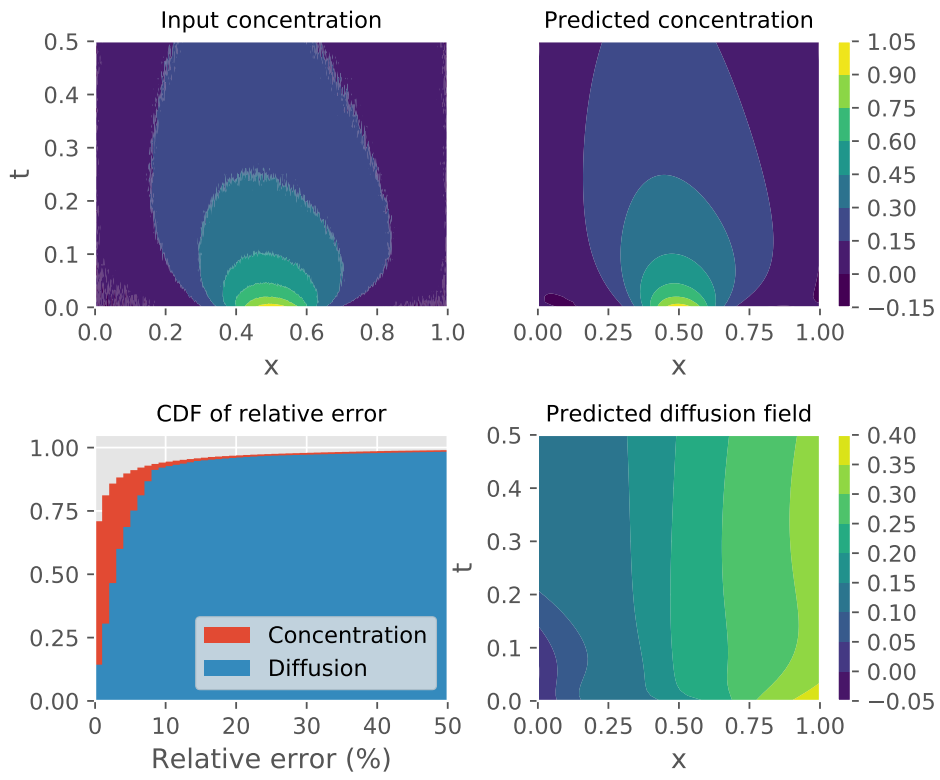
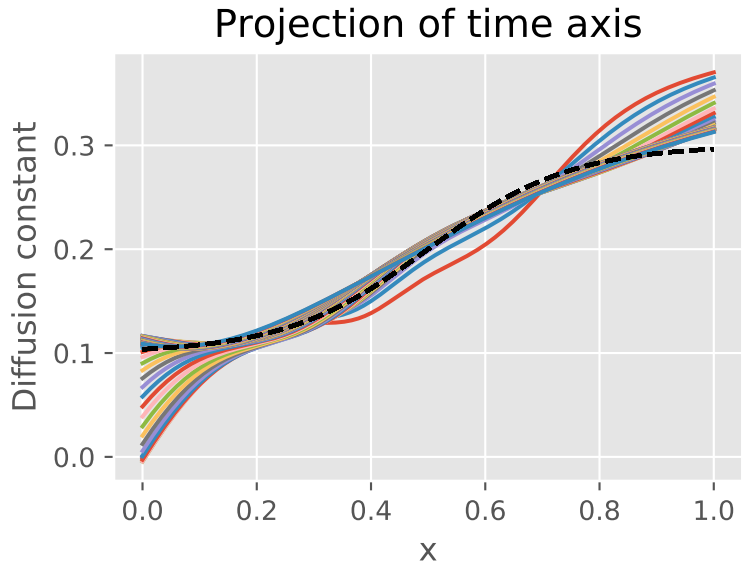


Figure 5.2.4: We show the training data and predicted concentration profile in the upper left and right panels. The lower right panel shows the inferred diffusion field while the lower left panel shows the CDF of the relatice error of the diffusion and concentration.


Input concentration
Predicted concentration
t
x
CDF of relative error
Concentration
Diffusion
Relative error (%)
Predicted diffusion field


Projection of time axis
Diffusion constant
x

5.2.2.3 Real cell

5.3 Conclusion

5.3.1 Weak points and how to improve

6

Conclusion

Appendix 1: Some extra stuff

Add appendix 1 here. Vivamus hendrerit rhoncus interdum. Sed ullamcorper et augue at porta. Suspendisse facilisis imperdiet urna, eu pellentesque purus suscipit in. Integer dignissim mattis ex aliquam blandit. Curabitur lobortis quam varius turpis ultrices egestas.

References

1. Garcia, D. Robust smoothing of gridded data in one and higher dimensions with missing values. *Computational Statistics & Data Analysis* **54,** 1167–1178 (2010).

2. Sbalzarini, I. F. Seeing Is Believing: Quantifying Is Convincing: Computational Image Analysis in Biology. in *Focus on Bio-Image Informatics* (eds. De Vos, W. H., Munck, S. & Timmermans, J.-P.) **219,** 1–39 (Springer International Publishing, 2016).

3. Zimoń, M., Reese, J. & Emerson, D. A novel coupling of noise reduction algorithms for particle flow simulations. *Journal of Computational Physics* **321,** 169–190 (2016).

4. Zimoń, M. *et al.* An evaluation of noise reduction algorithms for particle-based fluid simulations in multi-scale applications. *Journal of Computational Physics* **325,** 380–394 (2016).

5. Grinberg, L., Yakhot, A. & Karniadakis, G. E. Analyzing Transient Turbulence in a Stenosed Carotid Artery by Proper Orthogonal Decomposition. *Annals of Biomedical Engineering* **37,** 2200–2217 (2009).

6. Grinberg, L. Proper orthogonal decomposition of atomistic flow simulations. *Journal of Computational Physics* **231,** 5542–5556 (2012).

7. Bruno, O. & Hoch, D. Numerical Differentiation of Approximated Functions with Limited Order-of-Accuracy Deterioration. *SIAM Journal on Numerical Analysis* **50,** 1581–1603 (2012).

8. Knowles, I. & Renka, R. J. METHODS FOR NUMERICAL DIFFERENTIATION OF NOISY DATA. 12

9. Rizk, A. *et al.* Segmentation and quantification of subcellular structures in fluorescence microscopy images using Squassh. *Nature Protocols* **9,** 586–596 (2014).

10. Karpatne, A., Watkins, W., Read, J. & Kumar, V. Physics-guided Neural Networks (PGNN): An Application in Lake Temperature Modeling. *arXiv:1710.11431 [physics, stat]* (2017).

11. Sharma, R., Farimani, A. B., Gomes, J., Eastman, P. & Pande, V. Weakly-Supervised Deep Learning of Heat Transport via Physics Informed Loss. *arXiv:1807.11374 [cs, stat]* (2018).

12. Pun, G. P. P., Batra, R., Ramprasad, R. & Mishin, Y. Physically-informed artificial neural networks for atomistic modeling of materials. *arXiv:1808.01696 [cond-mat]* (2018).

13. Raissi, M., Perdikaris, P. & Karniadakis, G. E. Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations. *arXiv:1711.10566 [cs, math, stat]* (2017).

14. Raissi, M., Perdikaris, P. & Karniadakis, G. E. Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations. *arXiv:1711.10561 [cs, math, stat]* (2017).