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

1.1 Quantitative work on the Golgi so far

1.2 RUSH system

1.2.1 Man II

2
Introduction

3
Data processing pipeline

In this chapter I present the work done on processing the rush movies. Several preprocssing steps hav been undertaken to improve the quality of the fit, and we present all here. Roughly, we can divide the process in four steps:

1. Segmentation and creation of masks
2. Denoising of movies
3. Calculation of spatial and temporal derivatives
4. The actual fitting

Below we describe each step separately.

3.1 Step 1: Segmentation

The images obtained from the rush experiments often contain multiple cells. Furthermore, we can also segment the image into roughly three different types: 1) the background, where nothing of interest happens. No cells are present here, 2) the cytoplasm, which is the area where we want to fit our model and 3) the Golgi itself, where we do not necessarily want to fit. Unfortunately, no bright field images were available, making segmentation significantly harder, as no clear cell boundary can be observed. Further complicating the story is the large dynamic range of the movies due to the fluorescence concentrating in the Golgi. The following procedures we present have been developed to deal with these problems. Note that they are empirical methods, i.e. there's no theoretical background as to why they *should* work. However, in practice they do and I haven't found any other method which was able to.

3.1.1 Voronoi diagram

This method is based on a technique called Voronoi tesselation and doesn't depend on ny measure of the intensity. It was developed after noting that since the cargo is spread throughout the ER in the first few frames and as the ER is roughly circumnuclear, we can use this to determine the centre of the cell (roughly). Voronoi tesselation then allows us to divide the frame into areas with just one point per area, i.e. one cell per area (theoretically). More precise, given n coordinates, voronoi tesselation divides the given area into n pieces, where every point in a piece is closest to one coordinate. In practice this means for us that each point in a cell area is closest to its the given cell centre. Figure **ref** shows this. Each calculated cell centre is a red point and the lines depict the borders between each voronoi cell. Assuming the cells don't move too much, they don't cross the cells and thus we apply the voronoi diagram calculated in the first few frames to the entire movie.

3.1.2 Intensity

For the fitting however we wish to make a slightly better approach than a voronoi diagram. As stated, we can't find the exact delineation of the cell, but looking at the intensity, we can see an 'area' of interest, separating background from the cell. Since the Golgi is quite bright in het last 200 or so frames, we consider only the intensity for the Golgi, while for the cytoplasm we consider both the intensity and its time derivative. Thus we have two analog but different processes. For the Golgi we do the following:

1. Renormalize the concentration C between 0 and 1.
2. Sum all frames. One then obtains an image such as figure **ref**

$$x \sum_{frames} C(x, y, t)$$

3. This image is thresholded, either through an otsu threshold or a manual one, until the mask roughly matches what we want. Note that extreme precision isn't required, since we just want the rough area. THis results in figure **ref**

For the cytoplasm we follow the same procedure only now we take the log of sum of the product of the intensity and its time derivative:

$$\log\left(\sum_{frames} C(x, y, t) \cdot \partial_t C(x, y, t)\right)$$

We thus obtain a complete mask for the movie as shown in figure **ref**

3.2 Step 2 - Denoising

In order to accurately calculate the derivatives and generally improve the quality of fitting, we wish to denoise and smooth the obtained movies. Denoising and smoothing is a subject about which many books have been written and there are hundreds of approaches. One oft-used technique is to Fourier transform the signal, cutoff all coefficients above a cutoff frequency and retransform back into the real domain. Next, a Savitzky-Golay filter can be used to finally smooth the result. However, a big issue with all these methods is their non locality. Since our movies have different scales, this is a big problem. Furthermore, they often smooth out sharp peaks. After evaluating several methods, I have settled on a relatively new method presented in **ref**.

The so-called WavinPOD method combines two well-known filtering techniques, known as wavelet filtering and Proper Orthogonal Decomposition. Below we explain each separately. Our explanation is adapted from **ref** and **ref**.

Wavelet filter

A wavelet filter is not really the appropriate name, as its more of a transform.

More about wavelet transform

Proper orthogonal decomposition

Proper orthogonal decomposition is a technique similar to what is known as Principal component Analysis in statistics and falls into the general category of model reduction techniques. It's often used in flow problems to extract coherent structures from turbulent flows. Simply put, in POD we wish to express data as a sum of orthogonal functions, where the basis is determined from the data, i.e. we don't impose something as a fourier basis, etc..

$$f(x, t) = \sum_k g(x)h(t)$$

Basically we're trying to find the eigenfunctions of the data. Full explanation in paper **ref** Each eigenfunction comes with a eigenvalue, which can be interpreted as the energy of a mode. The higher the eigenvalue, the more important the mode is to the entire signal. To reduce the dimension of the data, we pick a cutoff k_{max} and only use the modes $k < k_{max}$. Several methods are used to determine the cutoff, but often used is the knee-technique. When we plot the log10 spectrum in figure **ref**, one can often observe a 'knee'. Usually this point is taken as the cutoff.

WavinPOD

WavinPOD combines these two techniques in the following way. First, we decompose our problem with a POD transformation. This yields a set of temporal and spatial modes. We select the most energetic modes and wavelet filter these, before transforming them back to the real domain. As shown in **ref**, combining these techniques has an advantage over others.

In our case, we select the number of modes to be used by hand (30 in the case of MANII) and apply a 3-level db4 wavelet. We use a slightly higher than necessary level to increase smoothness. In the figure below we show the result for both a pixel in time and one time snapshot. Note that the result is significantly smoother, but that smaller details have been preserved.

3.3 Step 3 - Derivatives

Taking spatial and temporal derivatives of these images is not an entirely trivial operation due to the discreteness of the system. More specifically, taking numerical derivatives of data is extremely hard to do properly and becomes even harder in the presence of noise. Next to basic finite difference methods, one can for example use a linear-least-squares fitted polynomial, smoothing spline or a so-called tikhonov-regularizer **ref needed**. Each method comes with its strengths and weaknesses, but one particularly nasty thing for our context is that they don't scale well to higher dimensions and quickly become computationally expensive.

Another issue related to discretization is the size of the grid w.r.t. the size of the features. To see this, we plot a 2D-gaussian with $\sigma = 1$ in figure **ref**.

As expected, the derivative is normal to the isolines of the object. Now consider the discretized version of the object. Taking the naive spatial derivate w.r.t. to each direction means only considering a single row or column of and taking the derivative in that direction. Figure **ref** shows the result of this operation. An artifact is clearly visible: instead of a nice uniform derivative, we see a 'cross'. This effect is a cause of the discretization grid being too large for some smaller, often bright, objects.

To remedy this, one can for example artificially upscale the grid, interpolate the values inbetween, and take the derivates from this grid. This is not ideal however, since the upscaling requires a large amount of memory and is computationally expensive. Another solution which is common in image processing is applying a *kernel operator*. The advantage of a kernel operator is that it is extremely computationally cheap, as it involves convolving the original picture with a differentiation kernel. The differentiation kernel is an approximate version of a finite difference scheme. We use and show here the Sobel filter, which is the most commonly used one.

In a simple finite central difference scheme, we set

$$\frac{dx}{dt} \approx \frac{x_{i+1} - x_{i-1}}{2h}$$

where h is the distance between two points. In terms of a kernel operator, this would look like (the h drops out as the distance in terms of pixels is 1):

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

And applying it by convoluting it to a matrix gives the x-derivative:

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

and analogous for the y-direction. However, as we've seen, looking at just a single row introduces cross-like artifacts. To remedy this, we wish to include diagonal pixels as well. However, the distance between the diagonal pixels and the center pixel is not 1 but $\sqrt{2}$ and furthermore we need to decompose is it into $\hat{x}$ and $\hat{y}$, introducing another factor $\sqrt{2}$. Thus, one obtains the classis 3×3 Sobel filter **ref**:

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

Although not extremely accurate, the Sobel filter seems to do the tricks for us. Several other versions such as Scharr or Prewitt exist, offering several benefits such as rotational symmetry, but we have not pursued these. They just change the coefficients. Although we have shown a 3×3 filter here, the filter can take into account higher order schemes such as a 5×5 or 7×7. The major benefit of the spatial derivatives as a convolution operator is its computational efficiency: convolutional operations are performed parallel and are extremely fast.

For the time derivative, we apply a second order accurate central derivative scheme, while for the spatial derivatives (both first and second order) we apply the 5×5 Sobel filter. We analyze these in the next chapter .

3.4 Step 4 - Fitting

Now that we have gathered all our data we can use it to fit. We construct a model (advection-diffusion) and then use a 'simple' least-squares fit to obtain an estimate of the diffusion and advection coefficients. We note two extra issues. First, a diffusion coefficient defined positive, i.e. a negative diffusion coefficient is unphysical. We thus make two different fits in the next chapter as a check: one with unconstrained variables, and one where we force $D > 0$.

Secondly, it's highly plausible that the diffusion coefficient and advection are position and time dependent. One could construct the full model for this and obtain both the coefficients and their derivatives, but its highly unlikely that this will lead to consistent results and it would still need to happen locally. We thus perform a 'moving-window-fit', where we set the width of the time and position window around a central pixel and assume that the diffusion and velocity are smooth and constant enough in that window to ensure a decent fit. In this, it's quite similar to a technique known in computer vision as optical flow.

4 Results data analysis

Here we present the results from our analysis on the RUSH experiments. We only show the results of MANII because this is the only thing we studied.

4.1 General analysis

4.2 Analysis of time derivatives

4.3 Analysis of LS-fit

4.3.1 Diffusion

4.3.2 Advection

4.4 Analysis of constrained LS-fit

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 (1, 2, 3, 4, 5). 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

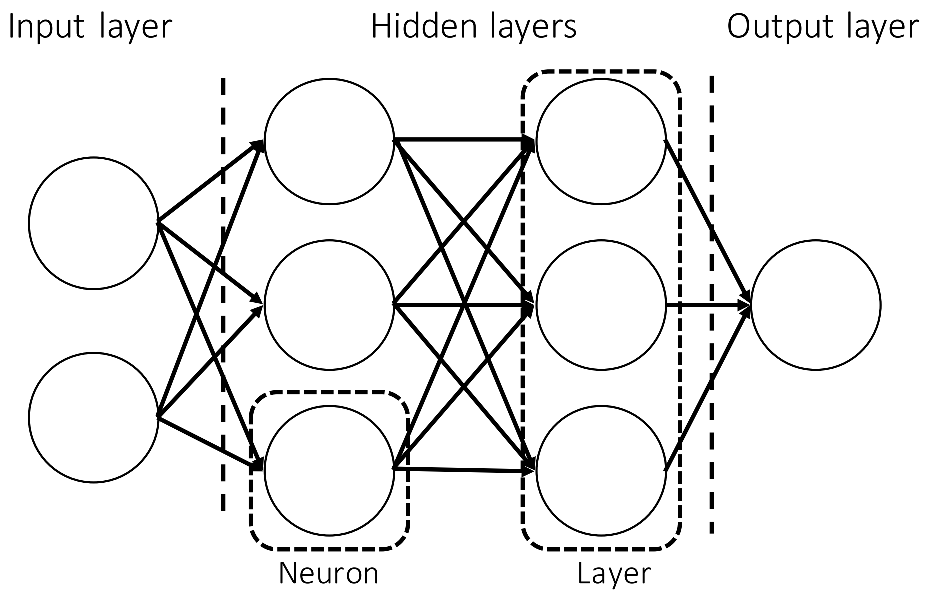

Figure 5.1.1: Schematic view of a neural network.

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 L and the most used form is a mean squared error:

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

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

In the case of neural networks we wish to minimize the cost w.r.t. to each weight w_{jk}^l and bias b_j^l. To do this computationally is not trivial, and most of the NN field uses one algorithm: back propagation. We give a short introduction below. We want to know basically two different things:

$$\frac{\partial C}{\partial w^l_{jk}}, \frac{\partial C}{\partial b^l_j}$$

the change of the cost w.r.t to each weight and bias. Let's define the error neuron j in layer l as $\delta^l_j = \partial C / \partial z^l_j$. We can rewrite this using the chain rule as:

$$\delta^l_j = \sum_k \frac{\partial C}{\partial a^l_{jk}} \frac{\partial a^l_{jk}}{\partial z^l_j}$$

However, the second term is always zero except when $j = k$, so we drop the sum. We also see that the second term is equal to $\sigma'(z^l_j)$. For the last layer L, the first term turns unto the derivative of the cost function, finally giving us:

$$\delta^L_j = |a^L_j - y_j| \sigma'(z^L_j)$$

This expression gives us the error in the output layer in terms of its weighted input. This in turn is a function of 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 calculate the errors going down the layers (from L to 0), hence the name backpropagation. Again using the chain rule gives:

$$\delta^l_j = \sum_k \frac{\partial C}{\partial z^{l+1}_{jk}} \frac{\partial z^{l+1}_{jk}}{\partial z^l_j} = \sum_k \delta^{l+1}_k \frac{\partial z^{l+1}_{jk}}{\partial z^l_j}$$

Using the definitions of z^l_{jk}, we finally after substitution obtain:

$$\delta^l_j = \sum_k \delta^{l+1}_k w^{l+1}_{kj} \sigma'(z^l_j)$$

Using these two equations, we can calculate the error in C due to each neuron. Finally, to calculate the $\partial C / \partial w^l_{jk}$ and
$\partial C / \partial b^l_j$, we relate these to the error, again via the chain rule:

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

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

These four equations together make up the the backpropagation algorithm. The complete optimization of the network goes as follows: 1. Complete a forward pass, i.e., calculate the expected outcomes with the current weights and biases. 2. Calculate the error and back propagate it to obtain the gradients in the weights and biases. 3. Adjust the weights using optimizer (e.g. gradient descent) 4. Return to step 1 and redo the cycle untill convergence.

Officialy, back propagation is a special case of a technique known as automatic differentiation, which is third type of differentiation, next to numeric and symbolic. It also 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:

$$\delta_j^0 = \frac{\partial C}{\partial x_j}\frac{\partial x_j}{\partial z_j^0}$$

so that:

$$\frac{\partial C}{\partial x_j} = a_j^0 \delta_j^0$$

Thus when learning, we also have access to high precision derivatives with regard to each coordinate!

5.2 Physics Informed Neural Networks

On the face of things, physics and neural networks seem to satisfy two completely different goals. Whereas physics tries to build (simplified) models, neural networks try to learn a general modelless mapping from the inputs to the outputs.

However, recently some approaches have emerged which fuse these seemingly opposite goals in order to do two different things:

1. Use a neural network to simulate/numerically solve equations.
2. Use a neural network to 'fit' a model to spatiotemporal data and even infer a coefficient field.

Initial results have shown very promising: using neural networks to numerically solve models doesn't require any advanced meshing and carefull handling of shocks, whereas the ability to infer coefficient fields from spatiotemporal data hasn't been shown at all to my knowledge.

5.2.1 The concept

Consider a set of 1+1D spatiotemporal data, consisting of some property u_i at coordinates (x_i, t_i). As stated before, a neural network learns the mapping $x, t \rightarrow u$ because it is a universal function approximator through minimizing the mean squared error:

$$MSE = \sum_i (u_i^{in} - u_i^{pred})^2$$

Now assume that we know that $u(x, t)$ is governed by some process which can be modeled as a partial differential equation. We can write (almost) every PDE as:

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

where f is some sort of function of u or its spatial derivatives. Now we rewrite is as:

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

To satisfy the model, the function g always has to go to zero. The idea behind Physics Informed Neural Networks (PINNs) is that we can include this function as an extra cost to be minimized:

$$cost = \sum_i (u_i^{in} - u_i^{pred})^2 + \sum_i g_i^2 = MSE + PI$$

Since to satisfy the model we need $g \rightarrow 0$, by adding our second 'Physic-informed' term, we effectively penalize solutions not satisfying the physics we put in: our new term acts as a regularizer. More concretely, this term has two effects:

1. **It prevents overfitting:** Neural Networks can be prone to overfitting. This term prevents that by penalizing variations not described by the physics.
2. **It makes the NN more data-efficient:** we can get good results with not much data.
3. **It allows fitting and prediction:** we can fit and predict based to the terms in f.

The first point is rather technical and interesting for more in-depth, so we'll leave it for now. The second point is very interesting from an experimental point of view. By presupposing some structure in the data in the form of a model, we need significantly less data to get the same result. It's also here that we see the no free lunch theorem: a price has to be paid. By encoding a model into the neural network, we lose the freedom of the neural network to map x, t u using any function. For us physicists however, this is actually a blessing but for applications where equationless modeling is usefull this is terrible. Note that we also circumvent the issue of numerical derivatives, since we can use the network provided automatic derivatives at machine precision. This combination provides the power of PINNs - a one-two punch consisting of physics regularization and automatic derivatives.

Fitting and prediction

Point three is however where PINNs really shine. How can the extra term we've included have these effects? To see this, consider the classics physics exercise of calculating the trajectory of a launched object. In this case we know the function g, i.e. $\ddot{y} = -g$. A classical numerical solver would take small steps in time, updating the position and speed of the object each step according to g. A PINN however uses a completely different approach. Given the initial state of the neural network, it calculates a first trajectory but then keeps adjusting the weights of the network until the cost is minimized, i.e. g goes to zero everywhere. In other words: the Neural network keeps launching the object and adjusting its internal weights until the physics are satisified at every step of the trajectory. The classical numerical approach goes for one correct try; the neural network just keeps trying until it converges on the correct solution. This approach allows us at the same time to circumvent complex discretization schemes and issues such as solutions blowing up. Such an approach is possible because of the backpropagation algorithm, which allows us to calculate the part of each neuron in the total cost and in which direction to change the weights and biases to minimize the cost.

Interestingly, we can also use this framework to fit models to spatiotemporal data by letting the coefficent of each term be a variable w.r.t to which we minimize too. More concretely, where before the cost was a function of the weights and biases, $cost = f(w^l_{jk}, b^l_k)$, we now let it be a function of the coefficients λ of the PDE as well: $cost = f(w^l_{jk}, b^l_k, \lambda)$. This is shown for all kinds of systems in the papers of [**ref**]. In theory however, it should also be possible to infer coefficient *fields*, both spatially and temporally varying. We can achieve this by a multi-output Neural network. Instead of outputting just c, we also output the coefficient at that spatiotemporal point, as shown in figure [**ref**]. We investigate this claim in the next section.

5.2.2 PINNs in practice

In this section we wish to evaluate the use of PINN's for our RUSH data. We start with a toy problem: a simple diffusive process. This has already been shown in the papers by Raissi, but it's just to show the reader. We then show the similar problem but with two different diffusion constants. This has not been shown by Raissi and we show here thats is possible to infer a coefficient field from the data using PINNS.

We use the following toy problem: a 1D box, started with initial condition:

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

and a diffusive process:

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

on the spatial domain (0,1) with boundary conditions in equation eq. 5.4

$$c(0, t) = c(1, t) = 0 \tag{5.4}$$

i.e. perectly absorbing boundary conditions. We used mathematica to solve these equations. The code is the appendix.

5.2.2.1 Constant D

Noiseless

Now consider the problem with a constant diffusion of $D_0 = 0.01$. We simulate the data on a domain $x : [0, 1]$ and $t : [0, 0.5]$ we use a spatial resolution of 0.01, giving the number of points 101 by 51, giving a total number of data points of 5151. Since the fitting is the training, we do not need to separate the data in a training and validation set. Figure ref shows the input data and the what the network predicts. After training, the network predicts (almost exactly) what we want it to and the average error is less than 1. Ofcourse, more interesting is the inferred diffusion parameter. With a value of 0.10034, the error is roughly 0.3. This is very good, but ofcourse our data is without noise. Note however that even still is extremely good and that in their papers Raissi shows far more complex equations such as the complex Schrodinger one. Also note that the error is mostly located at areas with low signal. This is a consistent problem and must be taken into account. Powerful as they are, neural networks seem to struggle with this fig. **??** .

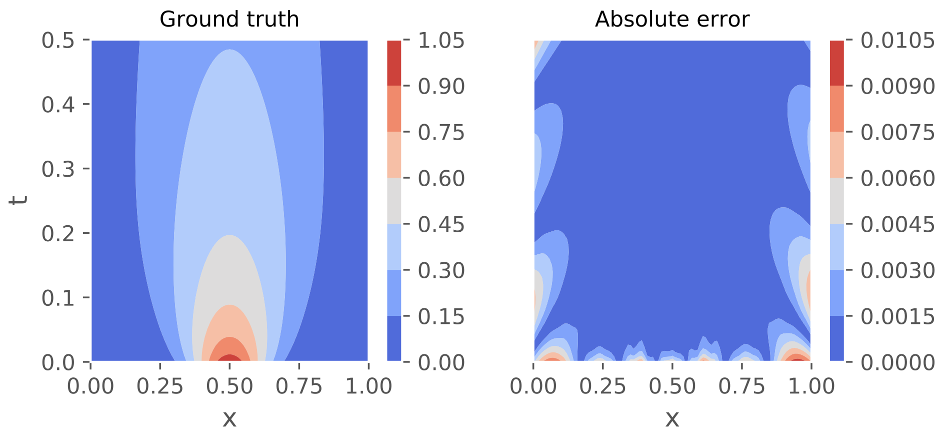

Figure 5.2.1: Left panel: something. Right panel: something else

Looks really nice.

Noisy

fig. **??** It becomes more interesting once we add noise. We take exactly the same problem, but now add 5% gaussian distributed white noise (e.g. $0.05 std(c)$) and let the neural network do it's thing again. The network is now doing two things at once: it's *denoising* the data by stating that the underlying data is of a certain model while finding the optimal model parameters. Again figure **ref** shows our original data and the fit.

The data correctly infers the ground truth. The inferred diffusion coefficient is 0.10017, an error of 0.17. I just want to state that this is almost ridiculous. We have roughly 5000 points with 5 error and the network is able to infer the coefficent within 1. We also havent optimized the network in any way: it's the most basic full connected layers with tanh activation function and we just used enough layers and neurons. It's also a very general technique: it works for whatever PDE and data multidimensional data. We also fit the model directly to the data; circumventing the need to know boundary conditions, initial conditions etc… We can now proceed to more advanced setup.

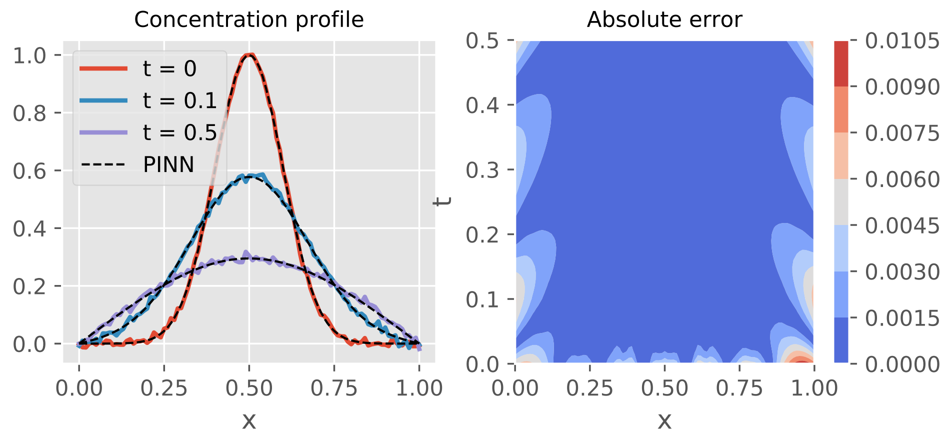

Figure 5.2.2: Left panel: something. Right panel: something else

5.2.2.2 Varying D

As stated, it should be possible to infer varying coefficient fields. Instead of using a single output network, we implement of two output neural network; outputting both the coefficient and the concentration. Note that this is slightly different from what is done in this paper **ref**. They infer the pressure field, but this is a separate added term; its not multiplied by a spatially varying term.

Non-varying

We first demonstrate this using the same data as before: so although we allow a spatially varying field, the underlying data only has a single diffusion coefficient. The coefficient is then learned through training the network and ‘corrected’ by PI-loss function. Figure **ref** shows our results.

test reference[4]

and another one

Varying

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. 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).

2. 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).

3. 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).

4. 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).

5. 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).