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 we present the workflow we have developed to fit a model to spatiotemporal data, specifically data from the RUSH experiments. The pipeline can be divided into four steps, including preprocessing:

- **Step 1 - Denoising and smoothing:** Denoising and smoothing the data.
- **Step 2 - Spatial and temporal derivatives:** Calculating the spatial and temporal derivatives which will be used for fitting.
- **Step 3 - Segmentation:** segmenting the movies into several areas of interest.
- **Step 4 - Fitting:** Actually fitting the model.

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 setup should work optimally. Instead, it's been developed while analyzing the data, adapting each step on the go.

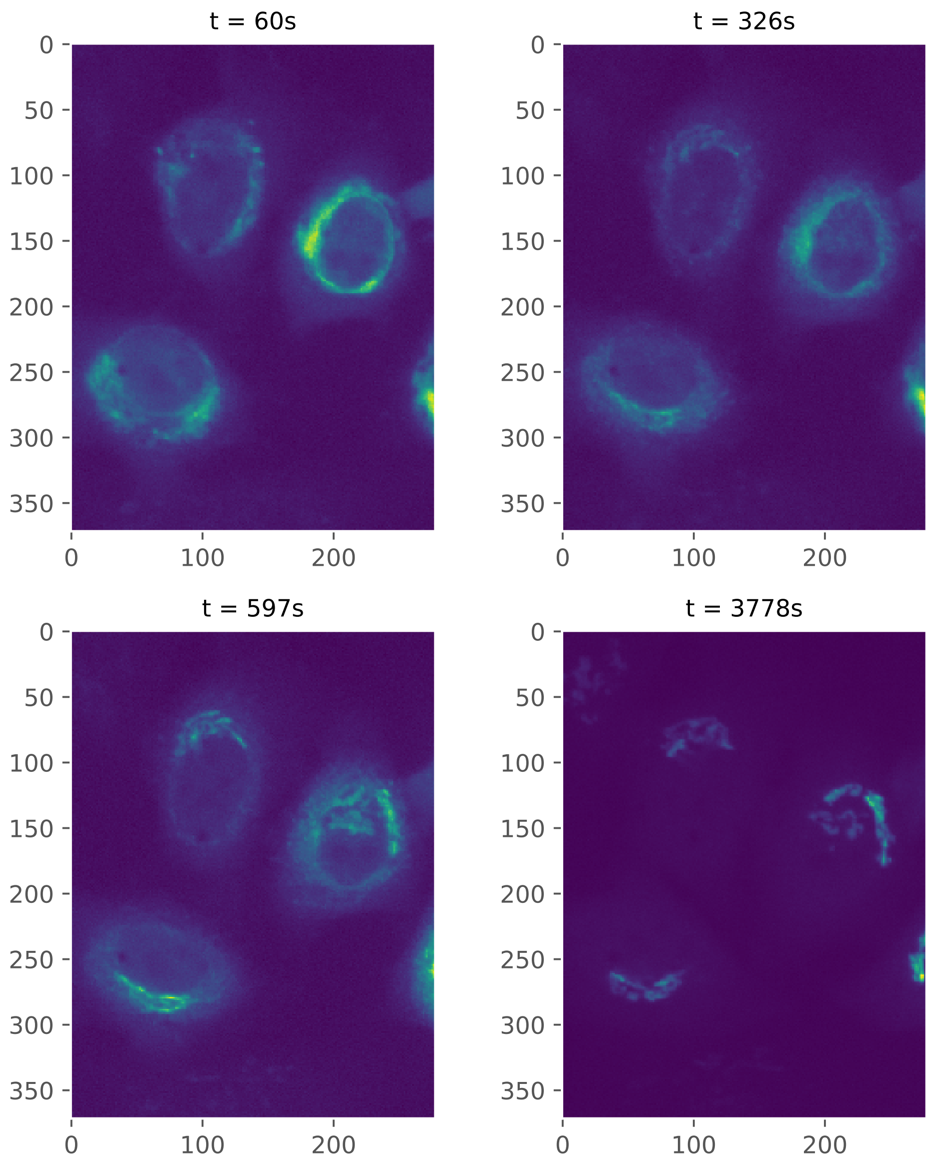


Figure 3.0.1: Four frames (t = , ,) showing a typical MANII cycle of the RUSH experiment.

3.1 Step 1 - 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.

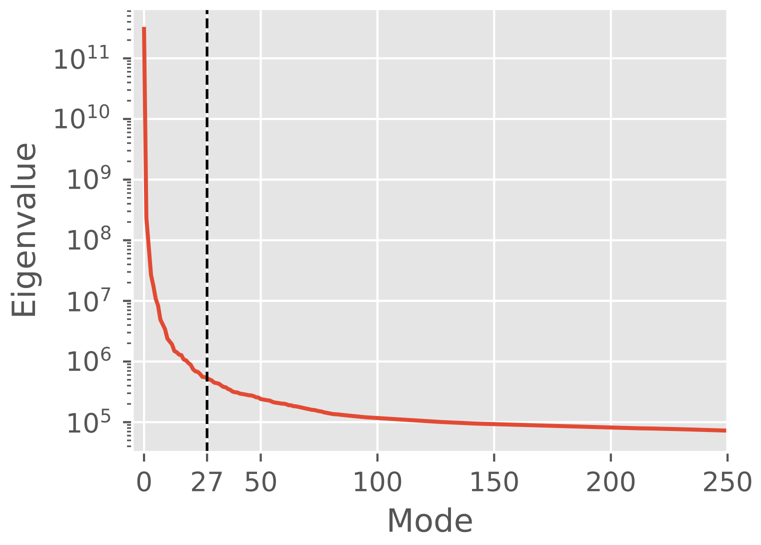
More about wavelet transform

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..

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

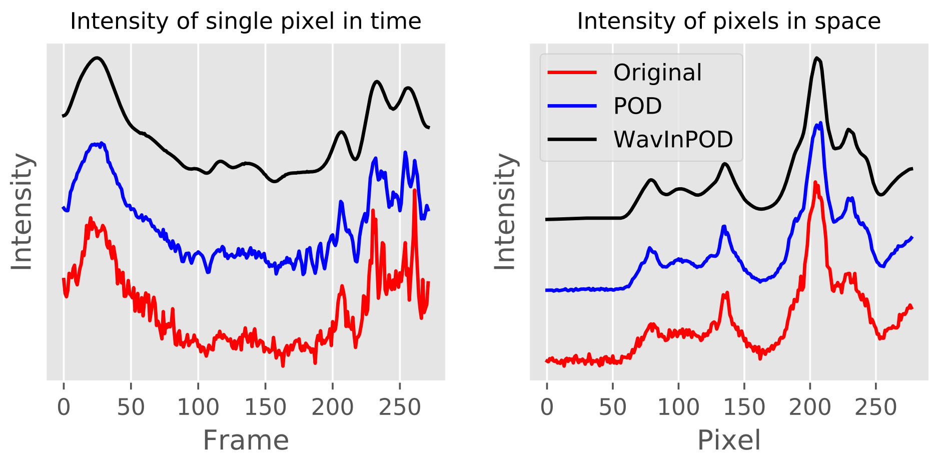
Basically we're trying to find the eigenfunctions of the data. Full explanation in paper **ref** Each eigenfunction comes with a eigenvalue, which can be interpreted as the energy of a mode. The higher the eigenvalue, the more important the mode is to the entire signal. To reduce the dimension of the 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.



3.2 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[1]. 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[2]. 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:

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

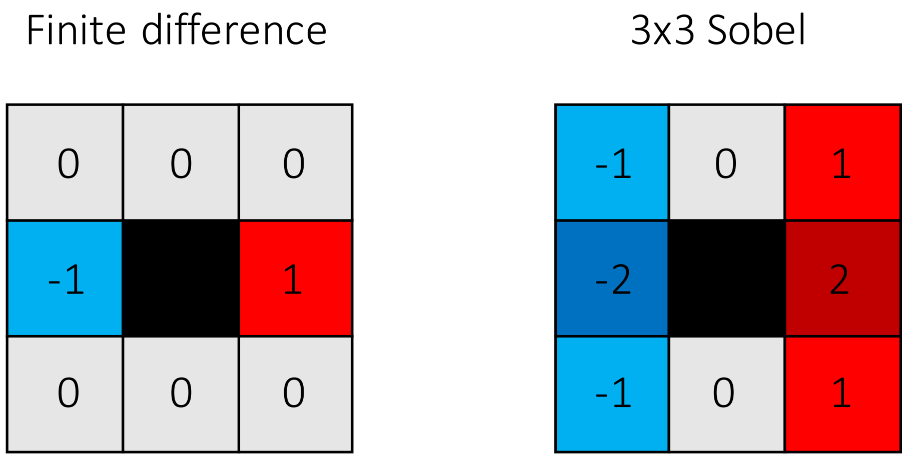


Figure 3.2.1: Left panel: One dimensional finite difference kernel. **Right panel:** Three by three Sobel filter

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. 3.2.1). 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.

3.3 Step 3 - Segmentation

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 3 unavailable. We have thus developed two methods which allow us to segment the image and the cells, based on Voronoi diagrams and the intensity.

3.3.1 Voronoi diagram

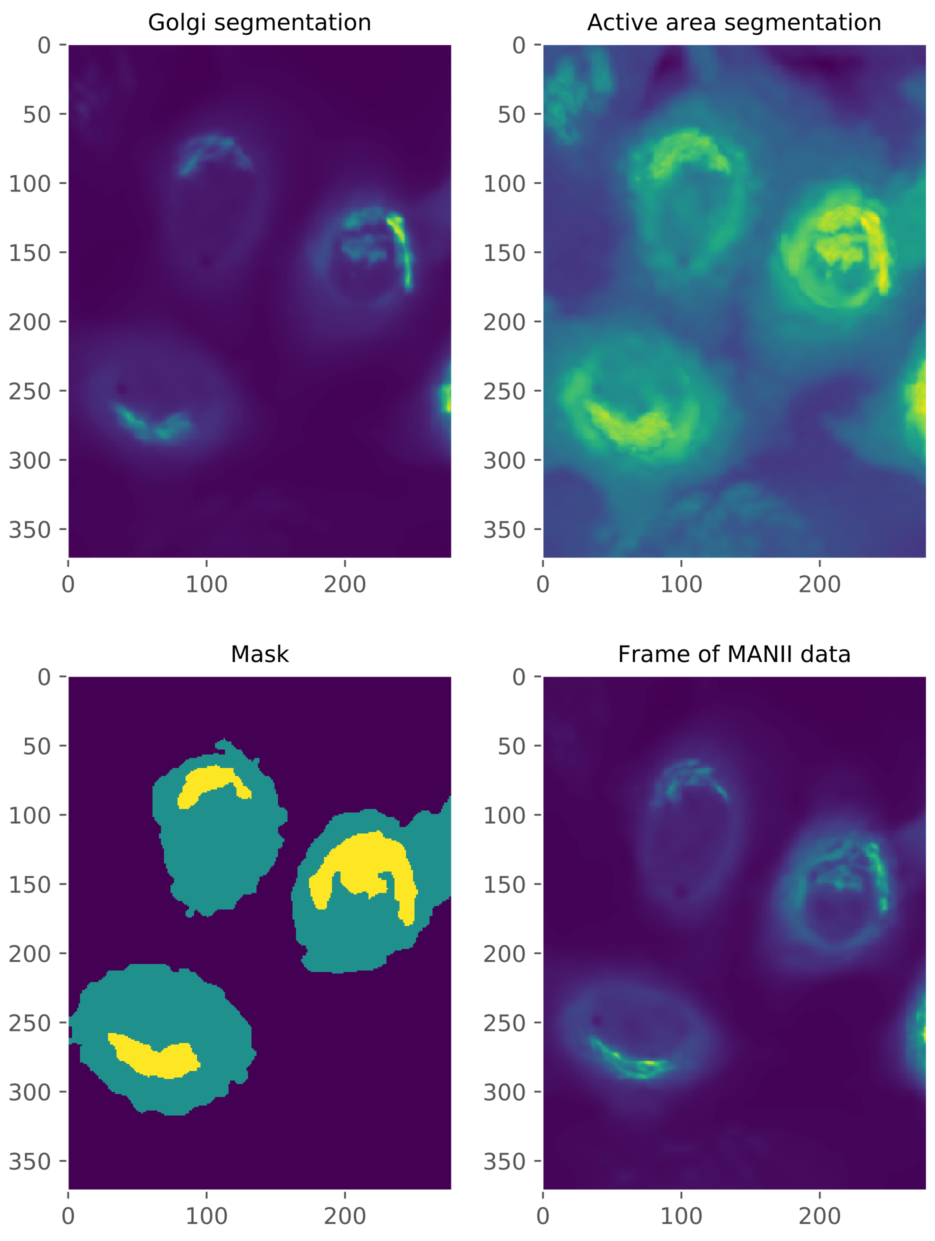
Consider again the frame on the left of figure fig. 3.0.1. Note that in early frames such as this one, the cargo (i.e. fluorescence) is spread circumnuclear. Applying a simple intensity based segmentation gives us a number of separate areas, which *very* roughly correspond to a cell. We can then pinpoint 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. **??**. Each cell centre is denoted by a dot, while the lines denote the border between each voronoi cell.

3.3.2 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. 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. **??** 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.


Golgi segmentation
Active area segmentation
Mask
Frame of MANII data
0
50
100
150
200
250
300
350
0
100
200

3.4 Step 4 - Fitting

Having gathered all the data, the final step is to fit the model. Consider the 2D advection-diffusion equation with isotropic diffusion:

$$\partial_t c = D\left(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2}\right) - v_x\frac{\partial c}{\partial x} - v_y\frac{\partial c}{\partial y} \tag{3.1}$$

where we have assumed that D, v_x and v_y are constant. We can rewrite this as:

$$y = D(x_1 + x_2) - v_x x_3 - v_y x_4$$

where $y = \partial_t c$, $x_3 = \partial_x c$ etc. Since we have determined x_i and y, we can apply least squares to obtain D, v_x and v_y. Note that physically we have $D > 0$. Constrained least squares can be used to perform a least squares fit whilst demanding $D > 0$. The results of both fits can be found in the next chapter.

We also assumed in stating eq. 3.1 that all coefficients were constant. Not making this assumption, we would end up with a model with several extra terms such as $\partial_x D$ and $\partial_x v_x$, ignoring temporal dependence. In the least squares fitting, there's no connection between a coefficient and its derivative and thus highly likely we end up with two incompatible results. To circumvent this issue, we assume the coefficients are indeed constant and fit the model to data locally, making the assumption that the coefficients in this area are roughly constant. We perform this operation for every pixel in a sliding-window manner, as shown in figure fig. 3.4.1. We thus end up with a fit for every pixel in our data, i.e. the velocity and diffusion field.

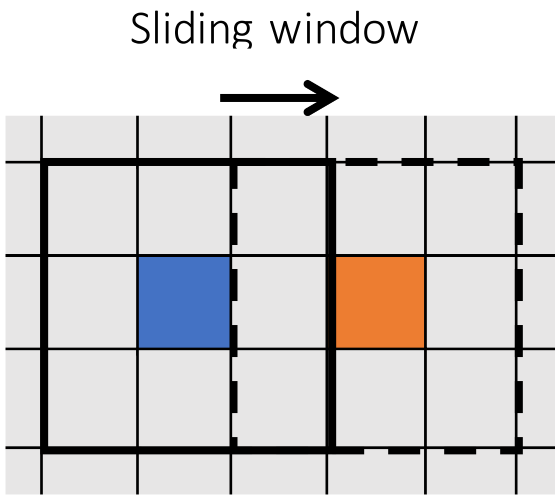


Figure 3.4.1: Schematic overview of the sliding window technique. The solid black line encompasses an area around its blue coloured central pixel and the fit output is assigned to that pixel. We then move the window (dashed black line) and perform the fit for the orange coloured pixel.

4 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 (4, 5, 6, 7, 8). 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

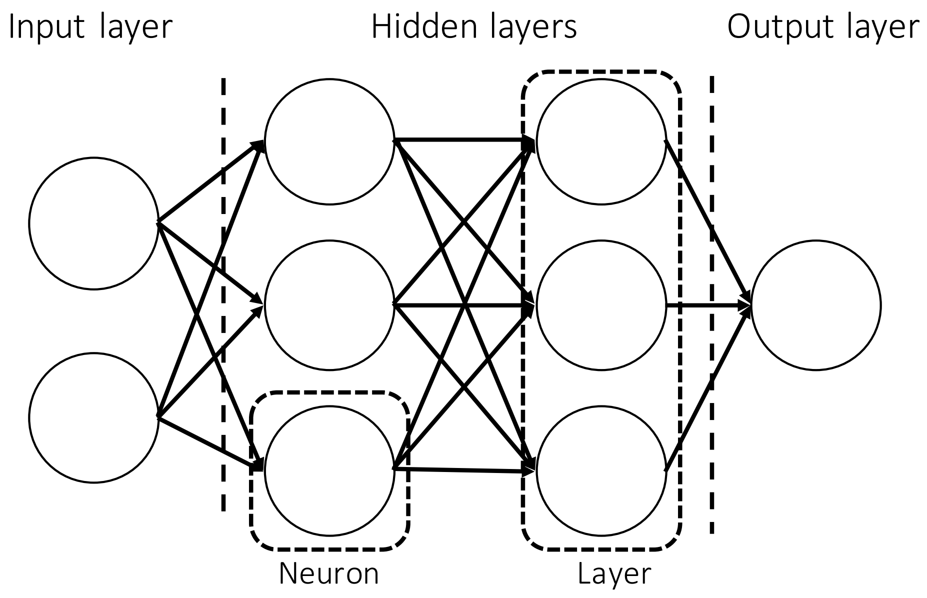


Figure 5.1.1: Schematic view of a neural network.

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:

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

so that:

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

Thus 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 (8, 7). 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, $g \rightarrow 0$. 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(8, 7). 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.

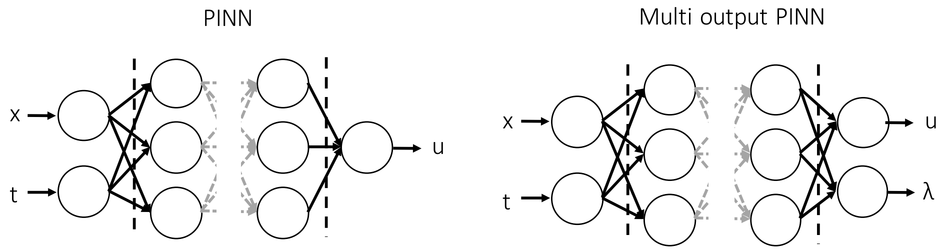


Figure 5.2.1: Left panel: a normal single output PINN. Right panel: a multi-output PINN. The network now also predicts the coefficients values at each data point.

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.

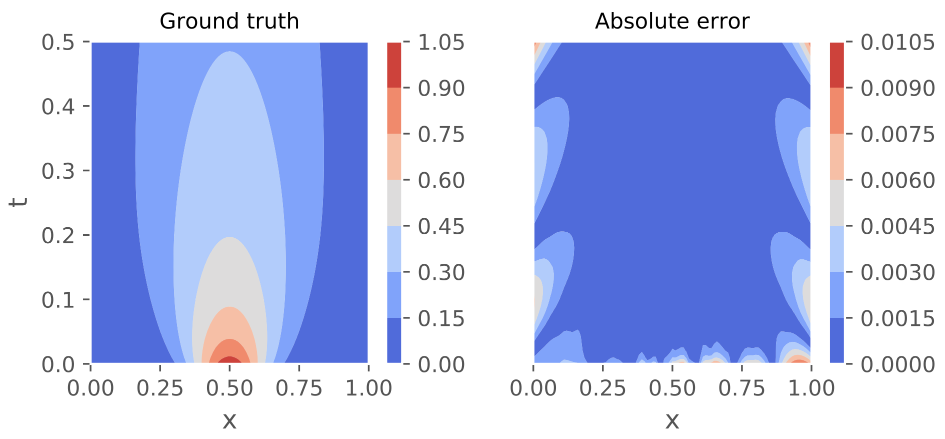


Figure 5.2.2: Left panel: Simulated ground truth of the problem. **Right panel**: The absolute error of neural network. Note that most of the error is located at areas with low concentration, i.e. signal.

The predicted diffusion coefficient is $D_{pred} = 0.100026$, giving an error of 0.026%. In 8, 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.

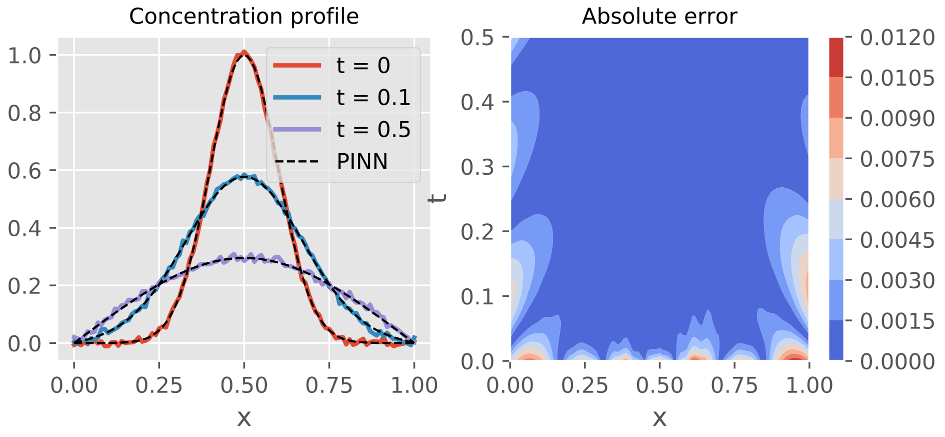


Figure 5.2.3: **Left panel**: The original noisy concentration profile with the neural network inferred denoised version super imposed. **Right panel**: The absolute error of neural network with respect to the ground truth. Note that most of the error is located at areas with low concentration.

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.

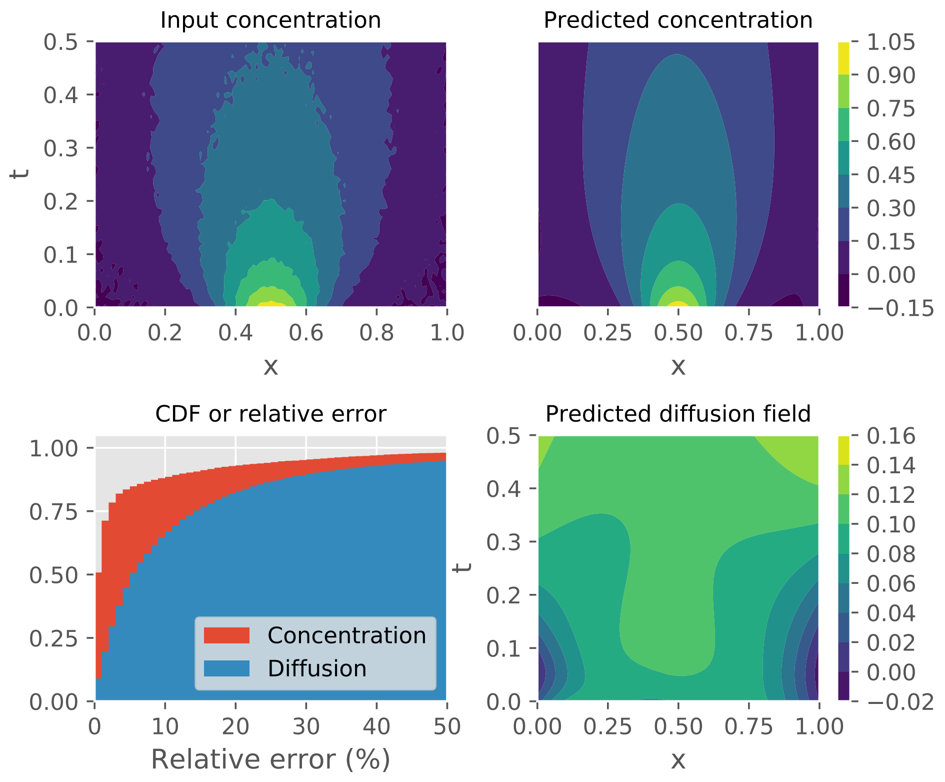


Figure 5.2.4: We show the training data and predicted concentration profile in the upper left and right panels. The lower right panel shows the inferred diffusion field while the lower left panel shows the CDF of the relatice error of the diffusion and concentration.

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

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

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

4. 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).

5. 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).

6. 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).

7. 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).

8. 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).