# Detection of T1 phase transitions in simulated confluent cell monolayers using CNNs

Nigar Abbasova
*Department of Physics, University of Oslo*
(Dated: May 31, 2023)

We trained a LeNet-5 inspired convolutional neural network using supervised machine learning to classify images using binary classification into classes with cells undergoing T1 transitions and cells without any T1 transitions. We trained the model on simulated data of two sizes - one dataset with 10008 images, and another with 22300 images. We achieved a maximal accuracy of 94 % using the Adam optimiser with a learning rate of 0.01 with no weight decay on a model that was trained for 20 epochs.

## I. INTRODUCTION

Collective cell motion is of fundamental relevance for various biological processes, such as cancer metastasis, tissue morphogenesis and others [16]. Collective cell motion relies heavily on the active interactions of cells with each other, in contrast to single-cell motion where no neighbouring cells are considered. Thus, cells are able to exhibit emergent structures and dynamics through these interactions, and one of the fundamental processes for such emergent large-scale behaviour is the topological rearrangement of neighbouring cells [14]. Such rearrangements are called T1 transitions, which are characterised as local, dissipative events leading to changes within the tissue architecture and influencing large-scale flow properties of cell tissues [14]. During a T1 transition, two neighbouring cells move apart while their two neighbouring cells come together (see Fig.(1)).

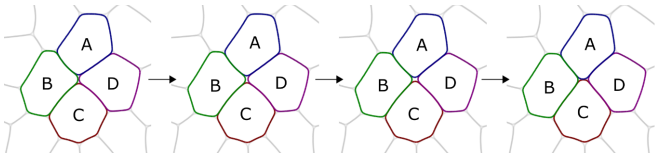



FIG. 1: Time evolution of tissue undergoing a T1 transition. The transition starts when B and D detach, and is completed when A and C come in contact. Extracellular gap is also formed between all four neighbouring cells during a transition. [14]

Such transitions have been observed in real tissues (for example, during wing morphogenesis of a Drosophilia [12]), however, it can be difficult to identify such transitions with the naked eye in a system composed of many cells. Due to various experimental drawbacks, it can be difficult to quantify T1 transitions in epithelial tissues, as the cells become very flat when confluent, and the cellular barriers are not clearly visible. In order to quantify and identify T1 transitions experimentally, one would require to fluorescently tag specific molecules in the cell membrane and image the cells using a specific microscopy mode. In addition to that, T1 transitions are temporally limited events - there exists a natural time delay between successive T1 events [11]. Given that, it would be difficult to image fluorescent cells over a long period of time as it can be damaging for the cells themselves. Otherwise, it would be very difficult to identify cell borders i.e., segment cell images and identify T1 transitions.

In order to tackle the issue of fluorescent imaging, a lot of machine learning algorithms have been developed to perform automatic cell segmentation e.g., one of the most popular ones is Cellpose 2.0 [20]. It allows users to train their own model i.e., use transfer learning and perform image segmentation on cells. Even though one is able to achieve decent cell segmentation using Cellpose, it is still challenging to acquire enough data with T1 transitions and identify them by eye. Thus, one possible way of identifying these transitions is by training a neural network. An example of a neural network that was designed to learn from visual data is a convolutional neural network. Convolutional neural networks are a variation of feed-forward neural networks used for object detection, e.g., autonomous vehicles and face detection, image captioning for images and videos and others [21]. Convolutional neural networks have also been used to identify phase transitions in other physical systems, such as phase transitions of the 2D Ising model and the q-state Potts model [15].

Due to the drawbacks associated with acquiring experimental data, the goal of this study is to train a convolutional neural network to identify T1 transitions in confluent cell monolayers using simulated data. We simulate a confluent cellular monolayer using a multi-phase field model with large shape deformations, which result in rise of spontaneous T1 transitions. This is because the cells are modelled as active incompressible droplets with emergent extracellular spaces. See Fig.(6) for an illustration of what a simulated confluent cellular monolayer look like.

By training a model to identify T1 transitions in simulated data, we are hoping to employ it in transfer learning on actual experimental images of cellular monolayers in

[1] https://www.kaggle.com/nigarabbasova/cnn-t1-transitions

the future to successfully identify T1 transitions.
For this study, we will attempt at implementing one of the earliest and simplest convolutional neural networks called LeNet-5, which was originally designed in 1998 by Yann LeCun and others for document recognition [18]. They have concluded that a multi-layered network can learn well from complex datasets, and that reducing a number of free parameters in the neural network can lead to better generalisation [4]. The orginal dataset that the LeNet-5 model was trained on is the MNIST dataset with 10 handwritten digits. Thus, the network was built to identify 10 classes of images from 1 to 10. For our problem, we will simplify the output to two classes - images with T1 transitions and images without. In order to do that, we will simulate images of confluent tissues, and will split them into smaller images with 4-6 cells each. The dataset used for training, validation and testing will consist of images with and without T1 transitions.

## II. THEORY

### A. Multi-phase field modelling

In order to model the cells in two dimensions, we implement multi-phase field model formalism. The system is comprised of N cells with an area occupying a square domain of size $[0, L] \times [0, L]$ using periodic boundary conditions. The individual cells are represented by a scalar phase-field $\phi_i(\mathbf{x}, t)$ where $i$ is the index used to label each cell ($i = 1, 2, ..., N$). The inner and outer edges of the cells are set to bulk phase values of $\phi_i \approx 1$ and $\phi_i \approx -1$ respectively. The cell boundary is defined as the transition region between the two bulk phases.
The dynamics of the phase-field $\phi_i$ is implemented using conservative dynamics given by

$$\partial_t \phi_i + \mathbf{v}_i \cdot \nabla \phi_i = \Delta \frac{\delta \mathcal{F}}{\delta \phi_i} \tag{1}$$

where $\mathbf{v}_i$ is the advection velocity of the $i$-th cell and $\Delta$ is the two-dimensional Laplacian applied to the variational derivative of a free energy functional $\mathcal{F}$ with respect to the phase-field. The free energy $\mathcal{F}$ is defined as

$$\mathcal{F} = \mathcal{F}_{CH} + \mathcal{F}_{INT} \tag{2}$$

where $\mathcal{F}_{CH}$ is the Cahn-Hilliard energy term and $\mathcal{F}_{INT}$ is the interaction term.
The Cahn-Hilliard energy accounts for cell deformability and the strength of the attraction and repulsion interactions. The Cahn-Hilliard energy also ensures the phase separation into the two bulk regions (inner and outer edges of the cells), which are separated by a thin, diffusive interface. Cells with circular shapes minimise this energy term.
The interaction energy term is comprised of a local interaction potential, which couples cells to each other, where both repulsion and attraction terms are taken into consideration.
The advection velocity $\mathbf{v}_i$ introduces cell activity into the simulation, and has been modelled to account for properties of MDCK (Madin-Darby canine kidney) cells that we are interested in. The advection velocity is expressed as

$$\mathbf{v}_i = \mathbf{v}_i(\mathbf{x}, t) = \nu_0 B(\phi_i) \mathbf{e}_i(t) \tag{3}$$

where $\nu_0$ is the parameter controlling the magnitude of the activity and $B(\phi_i) = \frac{1}{2}(\phi_i + 1)$ is the cell's interior and exterior interface. $\mathbf{e}_i(t) = [\cos\theta_i(t), \sin\theta_i(t)]$ contains $\theta_i$ term which is the orientation of self propulsion of the cells. $\theta_i$ is defined as a combination of rotational diffusion and relaxation due to the orientation of cell's shape elongation.

### B. Deep neural networks

Deep learning is a subset of machine learning which uses artificial neural networks with multiple layers to learn complex patterns and relationships in data. Deep learning can be divided into two main sub-classes - supervised and unsupervised learning. Supervised learning relies on a labelled dataset provided by the user, while unsupervised learning does not. In supervised learning, various tasks such as classification and regression need labelled data to be able to make correct predictions and learn over time. In our study, we will be focusing on supervised learning using convolutional neural networks.

A typical neural network consists of weights and biases, where weights represent the strength of the connections between neurons in different layers, and biases refer to parameters associated with neurons that adjust during the training process to enable the network to learn complex relationships between the inputs.
In supervised learning, we consider a labeled training dataset $D = (x_i, y_i)_{i=0}^n$ where $x_i$ denotes the input and $y_i$ denotes the output that the model is trying to predict. The model itself is determined by a set of parameters which we denote as $\theta$. During the training process, the model is searching for the most optimal parameters $\hat{\theta}_D$ which minimise the loss function $L(D, \theta)$ [7]. Once the training is over and the most optimal parameters have been chosen, we can use the model to make predictions on an unseen dataset to evaluate the accuracy of our model. The goal of the training process is to minimise the loss function by adjusting the weights and biases of the network, which is done over a certain number of epochs. Epochs in this context refer to the number of times that we choose to pass our data through the algorithm for learning.

The loss function can often be thought of as the cost associated with the model's predictions with respect to the input data and the expected outcome. A learning algorithm that is used to minimise the loss function is called backpropagation - it computes the gradient descent with respect to the weight values for various inputs, and tunes

the weights and biases in the direction that reduces the loss. By updating the weights and biases recursively, the network learns to make better predictions and improves its performance.
Since our problem is a binary classification problem, we will use cross-entropy as the loss function, which is defined as

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(x^{(i)}, y^{(i)}, \theta). \tag{4}$$

Here,

$$L(x, y, \theta) = -logp(y|x; \theta) \tag{5}$$

is defined as the loss for each input [13] and $m$ is the number of nodes in a given layer. A node in a neural network is an element that receives the input, applies a transformation using weights and biases which is then followed by the application of an activation function to produce an output.

### *Activation functions*

Activation functions are essential to deep learning. They determine the output of the model, the accuracy and also the computational efficiency [6]. Activation functions introduce non-linearity to the model - this is important because physical problems that we deal with in real world do not always follow linearity. This is also applicable to problems which do not have straightforward solutions, and are instead divided by a decision boundary which is not linear. Introducing non-linearity to the network enables the model to create complex mappings between the network's inputs and outputs, which is essential to solving real-life problems [1].
There exists a myriad of different activation functions that can be used depending on the problem that one wishes to solve. Some of the most widely used ones are the **sigmoid** and **hyperbolic tangent** activation functions. See Fig.(2) for a visual illustration of the different activation functions.

**ReLU** (rectified linear activation function) is another type of activation function that is becoming the common choice since it is faster to compute than some of the others, and prevents the gradient descent computations from getting stuck. This is because the output of ReLU is not saturated - it does not have a maximum value. This is often an issue with sigmoid and hyperbolic activation functions as they have little to no gradient to propagate back through the network during the learning stage [6]. In addition to that, the exponential nature of these activation functions makes them more computationally expensive [6].
In comparison to the sigmoid and hyperbolic activation functions, the ReLU function is composed of two linear pieces, with its slope either being 0 or 1. The output of this activation function does not have a maximum value,

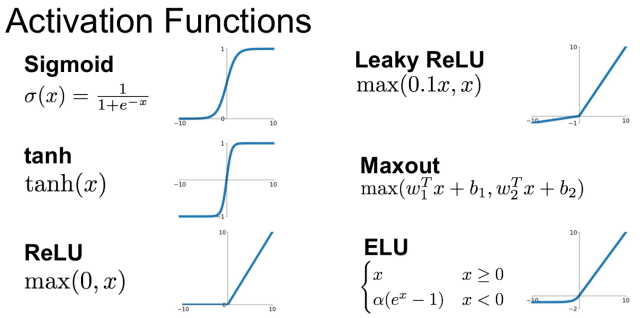



FIG. 2: Visual representation of different activation functions used in deep learning [2].

which makes it more suitable for gradient descent computations, and also much faster to compute than the sigmoid and hyperbolic activation functions [6].
However, certain issues arise when using the ReLU activation function - one of them is known as dead neuron problem, which occurs when a weight is negative and the neurons only output zero during computations (hence are not active during the learning stage). Therefore, to combat this problem, another activation function can be employed called the leaky ReLU, which allows for backpropation even if a neuron has a big negative weight associated with it [19].

### *Batch Normalisation*

Batch normalisation is a layer in a neural network included between hidden layers, which is used to normalise the outputs from one hidden layer before passing it as the input to the next hidden layer. The batch normalisation layer is often included in the architecture of a model even if the inputs are normalised during the preprocessing of data stage.
Layers with batch normalisation ensure better convergence of the gradient descent computations during training [8]. This is because batch normalisation reduces the dependence of the gradients on the initial values of the parameters. Batch normalisation can also reduce the need for regularisation, which is another tool that we can use to help with the problem of overfitting during training.
The batch normalisation transformation contains two additional trainable parameters for each dimension of the input - $\beta$ (offset) and $\gamma$ (scale), which help the model choose the optimum distribution for each hidden layer.

### *Accuracy*

Apart from computing the loss of a network, we can also compute its accuracy. Accuracy of a network is measured as a ratio of correctly predicted labels over the total

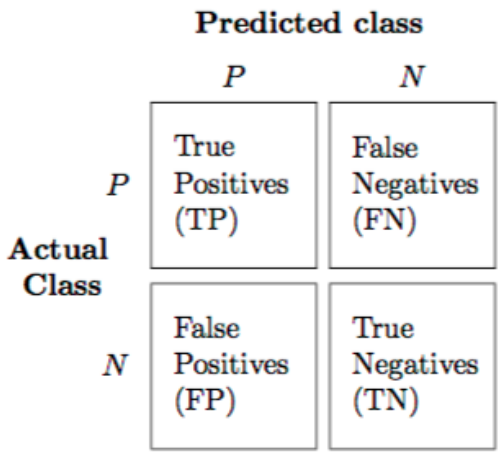



FIG. 3: Example of what a confusion matrix outputs for a binary classification problem [22].

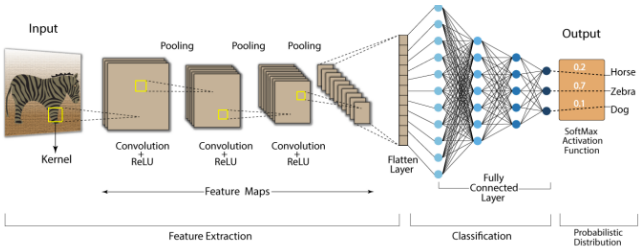



FIG. 4: Visualisation of a convolutional neural network showing the different layers that is consists of [10].

number of labels. In our case, it will be how many correct images with and without T1 transitions our network was able to identify.
To visualise the performance of the network, we can plot the **confusion matrix**. It shows us how many instances of true positives, true negatives, false positives and false negatives the network was able to identify. Ideally, we want to minimise the number of false positives and negatives, and maximise the true positives and negatives. A typical confusion matrix for a binary classification is a 2 x 2 matrix - see Fig. (3) for an illustration.

### C. Convolutional neural networks

Most of convolutional neural networks (CNNs) consist of three main building blocks (or layers, to be more precise in this context) - **convolutional layer**, **pooling layer** and a **fully connected layer**. See Fig.(4) for a visualisation of a typical convolutional neural network.

The number of convolutional layers and the choice behind the pooling layers is decided based on the task set for the network. LeNet-5 is a simple network in terms of architecture - it is comprised of two convolutional layers, followed by max pooling layers and finally the fully connected layers. Since the operations commute with each other, the ordering of the layers is not important - however, it can affect the execution time of the model. Since batch normalisation and activation functions do not commute, so the order is up to the designer of the model. We will now explain the theory behind each layer in a bit more detail.

In context of neural networks, convolutional layers refer to layers of the network that perform convolutions - linear mathematical operations that enable the network to create *feature maps*, which is an output of the network that represents certain features of the input.
A convolution is an operation defined as

$$s(t) = (x * \omega)(t), \tag{6}$$

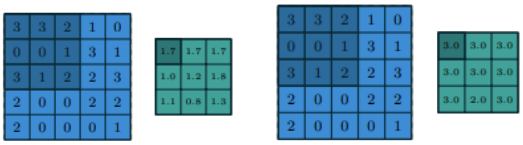
where x(t) is a function referred to as the **input**, $\omega(t)$ is the **kernel** and the output s(t) is the **feature map**. In the case of images, which are two-dimensional, we use a two-dimensional kernel K, and our convolution becomes

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n), \tag{7}$$

where I is the two-dimensional image input, and m, n are the elements of the input.
Two common techniques that can be used in convolutional layers are **padding** and **strides**.
Padding refers to a technique used when the kernel size does not match the size of the input, and as a result, the size of the output after convolution is not equal to the input. Hence, we can **pad** the edges of the input with additional pixels of a certain value to ensure that the original edge pixels are centered when the kernel is sliding across the input image. One of the more common paddings is zero-padding, where extra pixels set equal to zero.
Striding refers to skipping some of the locations that the kernel will sweep over during a convolution. This is done when we want an output with a lower size than the input. When we set a stride of 1, we choose to pick slides one pixel apart. With stride of 2, we pick slides two pixels apart and so on.

#### *Subsampling layers*

Subsampling (or downsampling) in convolutional neural networks are used to reduce the size of feature maps by implementing some function to summarise subregions of the map. Such functions can for example take the average or the maximum value of some region (average pooling or max-pooling, respectively) [9].
Pooling operation is very similar to a convolutional operation, but instead, we replace a linear combination using a kernel with another function. During pooling, we also slide a window across the input as we do with a kernel, and feed the outcome into a pooling function. During pooling, one can also implement strides just as one can




(a) (b)

FIG. 5: Visual representation of how pooling operations work on a matrix using (a) average pooling and (b) max-pooling [9].

in a convolution. See Fig.(5) for a visualisation of how the max and average pooling operations work.

### D. Data Scaling

Data scaling (also known as feature scaling) refers to either the normalisation or standardisation of data before it gets fed into a neural network for training.
There are various ways to perform feature scaling, some of which are normalisation (also known as min-max scaling), standardisation (setting values of each feature to have zero mean and unit variance) or scaling to unit length.
When the variance of input data is unknown, or there is a change that the features have a very large variance relative to each other, it is best to normalise the data to avoid insufficient performance of the model. If the input data is not normalised, the highly varying values can propagate through the layers of the network and lead to accumulation of very large error gradients. This can result in the exploding gradient problem, which consequently leads to an unstable model that is unable to learn sufficiently from the training data [5].
The min-max scaling is implemented by scaling the data to have values between 0 and 1. For each feature (attribute associated with an input, in our cases it is a pixel), the minimum value of the feature is set to 0, and maximum is set to 1. To implement it, we can use the following equation:

$$x_{scaled} = \frac{x - min(x)}{max(x) - min(x)}. \tag{8}$$

An advantage of the min-max scaling is that it does not assume anything about the statistical distribution of the data, which is particularly important when the statistical profile of the data is unknown. It is also possible to rescale data on a different interval [a,b], which would be given as [3]

$$x_{scaled} = a + \frac{(x - min(x))(b - a)}{max(x) - min(x)}. \tag{9}$$

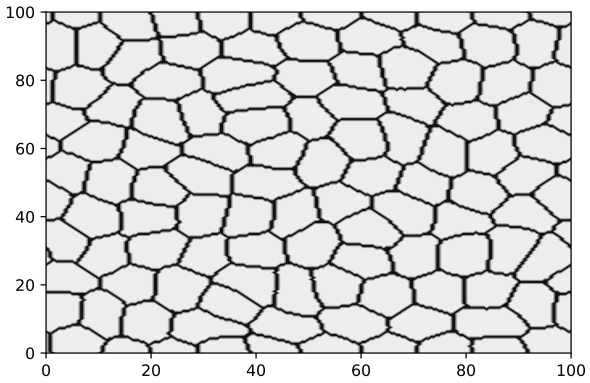



FIG. 6: Snapshot of the simulated MDCK cells at an arbitrary time point.

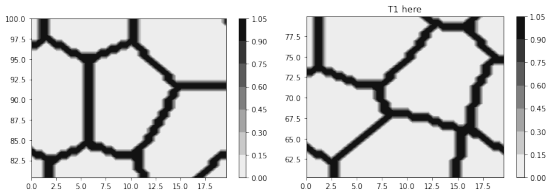
## III. METHOD

### A. Simulating data

We were given a set of experiments that were simulated using different parameters. The experimental results are contained as numpy arrays for the phase fields and the information about T1 transitions (where on the grid they occur and the time at which they occur). See Fig.(6) to see what an entire simulated tissue layer looks like. We then define a domain and split number, which we use to configure the number of cells that we want to include in each image. For a domain size of 100 and a split number of 5 (split number here refers to how many sections we choose to evenly divide the tissue image in), we get at least 5 cells per image (see Fig.(7)). Each image has an original size of 150 by 150 pixels. In order to identify a T1 transition, we need at least 4 cells in the frame, however, due to the nature of the simulation, it is also possible to have more than one T1 transition in an entire tissue layer.
During the generation of data, we acknowledge that certain frames are overlapping due to the way that we are cropping the entire tissue plane. However, this is not relevant for the training of the model, as we are randomly shuffling the data during the preprocessing stage and making sure to use images from multiple different experiments to train the model. Therefore, we assume the datapoints in the dataset (i.e the images) to be independent of each other.

### B. Preprocessing of the data

In order to improve the performance and stability of our deep learning model, we standardised our data using the min-max scaling given by Eq.(8) on the entire dataset, before splitting it into a train/validation/test




(a) (b)

FIG. 7: Example of the images fed into the network of cells (a) without any T1-transitions (b) undergoing a T1-transition. Here, we chose split number equal to 5.

set. We do this to ensure the robustness of the model and the quality of the results post-training.
We also ensure that the images are in grey-scale and resize them from their original size of $50 \times 50$ pixels down to $32 \times 32$ pixels. We resized the images to follow the LeNet-5 model's parameters, as they were introduced in the original paper [17].
Firstly, we evaluated the model on a small dataset consisting of 10008 images, and split it using a 40/30/30 ratio for train, validation and test sets respectively. We started our training with a batch size of 16, but have quickly moved on to bigger batch sizes and opted for 256 as our final batch size.
After establishing an optimal number of epochs, learning rate and weight decay, we trained the model on a bigger dataset to evaluate its performance. The bigger dataset consisted of 22300 images, which we also split using 40/30/30 ratio for train, validation and test datasets.

### C. Architecture of the network

We set up our network by following LeNet-5's original set up with two convolutional layers and three fully connected layers. We modified the original architecture by adding batch normalisation layers after each convolutional layer and a leaky ReLU activation function. For the two convolutional layers of our model, we used a kernel size of 5x5, a stride of 2 and zero padding. For the max pooling layers, we used also kernel size of 2x2 and stride equal to 2.

The two convolutional layers consists of a convolutional operation, followed by batch normalisation, a leaky ReLU activation function and max-pooling. After the two convolutional layers, the output tensor is flattened and fed into two fully connected layers, where we also apply the leaky ReLU activation function in between the layers. See Fig.(8) for an overview of the model. See the interactive notebook for a more detailed summary of the model provided by PyTorch.

Based on the input (shape of the images and 1 channel) and the chosen parameters for the convolutional and subsampling layers, our model consisted of 61070 trainable parameters in total. We chose Adam as the optimiser for our classification problem and cross-entropy as our loss function.

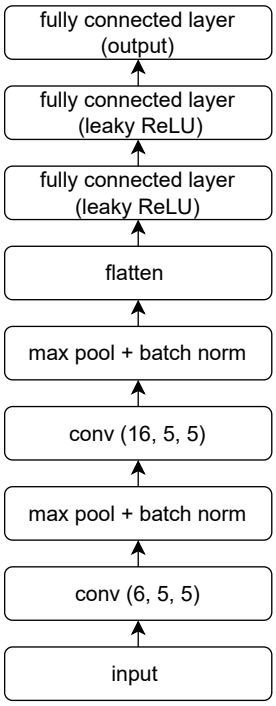



FIG. 8: Overview of our LeNet-5 inspired model architecture. The model consists of two convolutional layers, two subsampling layers and three fully connected layers with a leaky ReLU activation function. The last fully connected layer maps out the output which is the number of classes in our problem.

### D. Tuning hyperparameters

Our model consisted of three main hyperparameters: learning rate and weight decay for the optimiser, and number of epochs. We started off with the default learning rate given by PyTorch for the Adam optimiser ($lr = 0.001$), weight decay of zero ($wd = 0$) and 10 epochs. The weight decay is a type of regularisation technique, which adds a penalty term to the loss function of the network. Thus, weight decay encourages the model to learn smaller weights (as it imposes a cost for large weights) and prevents it from overfitting the training data. To investigate the effect of the hyperparameters on the model's performance, we trained the model using three different learning rates with and without a weight decay of 0.05 using 20 and 30 epochs.

## IV. RESULTS

### A. Results for data consisting of 10008 images

#### 1. Loss and accuracy

The model was trained on 4003 images, validated on 3002 images and tested on 3003 images. We computed the loss and accuracy of the train and validation datasets as functions of epochs for different numbers of epochs, learning rates and weight decays.
After investigating the effect of different learning rates, we decided to compare two specific values of learning rates with and without weight decay parameter. In Fig.(9), we evaluate the loss and accuracy for 30 epochs with learning rate $lr = 0.001$ and weight decay parameter $wd = 0.05$. We do the same in Fig.(10), but for 20 epochs.

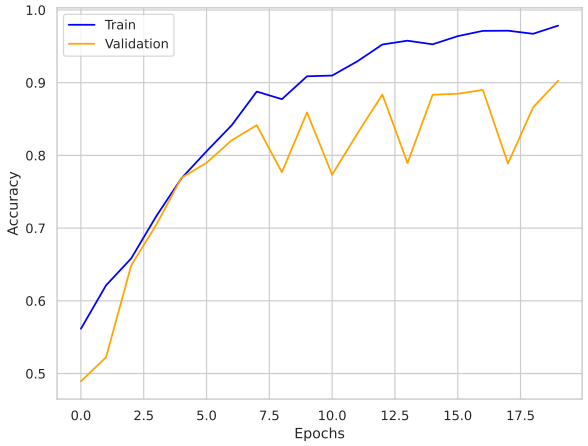
We then evaluate the model with the same learning rate of $lr = 0.001$ but without any weight decay. We do this for 30 epochs with the results illustrated in Fig. (11) and for 20 epochs in Fig. (12).

#### 2. Confusion matrices

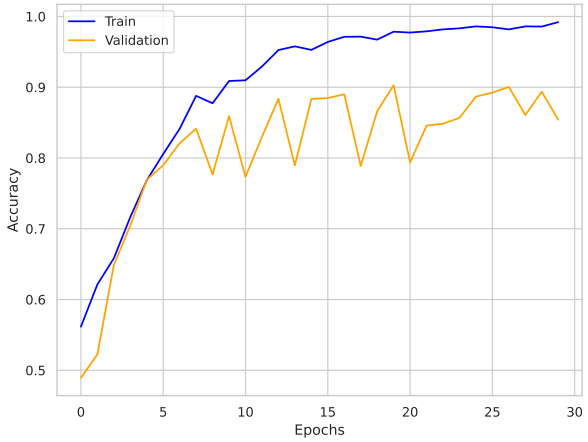
We also compare how well the models perform for different numbers of epochs using unseen test data. The accuracy of the model with a learning rate of 0.001 and weight decay 0.05 was 90% for 30 epochs, and 92% for 20 epochs. We illustrate how well the model performs by creating confusion matrices for different hyperparameters. See Figs.(13) and (15) for results with $lr = 0.001$ with and without weight decay of 0.05.
We also test the model with a larger learning rate of 0.01 with and without weight decay, with the confusion matrices shown in Figs.(14) and (16) respectively.

### B. Results for data consisting of 22300 images

After establishing the optimal number of epochs, learning rate and weight decay, we have trained our model on a larger dataset with 22300 images. We chose to train the model for 20 epochs and setting the learning rate to be lr = 0.01 with no weight decay. The model was trained on 8927 images, validated and tested on 6695 images. The loss and accuracy for the larger dataset is illustrated in Fig.(17). The confusion matrix for evaluation of the model on a bigger test dataset is illustrated in Fig.(18). We achieved an accuracy of 89% when training the model on a bigger dataset using a learning rate of 0.01 and 20 epochs with no weight decay.




(a)



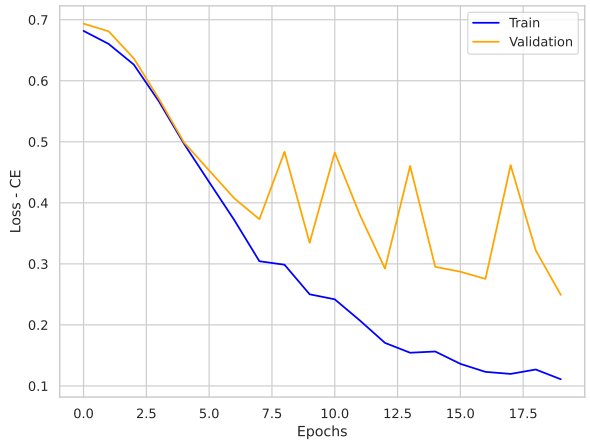

(b)

FIG. 9: Loss (a) and accuracy (b) of training and validation datasets as a function of epochs. Here, learning rate is $lr = 0.001$ and weight decay rate is $wd = 0.05$ and the total number of epochs is 30.

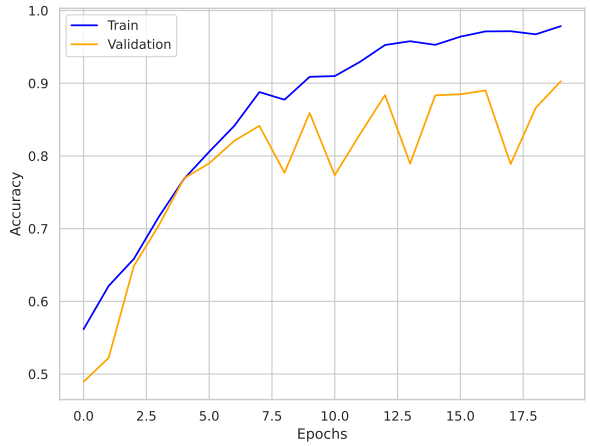
## V. DISCUSSION

### A. General remarks

We can see that the loss and accuracy of the model trained without weight decay with a learning rate of 0.001 converges much faster and in a more stable fashion than the model trained with a weight decay of 0.05. We see this clearly in when comparing Figs.(12) and (11), for both 20 and 30 epochs without any weight decay to those with weight decay of 0.05 in Figs.(10) and (9). It is perhaps important to mention that a learning rate of 0.001 is the default value set by PyTorch for the Adam optimiser, which was a good choice in terms of the conver-

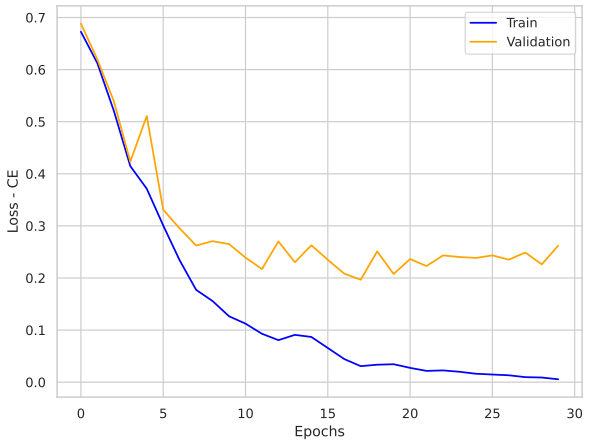



(a)

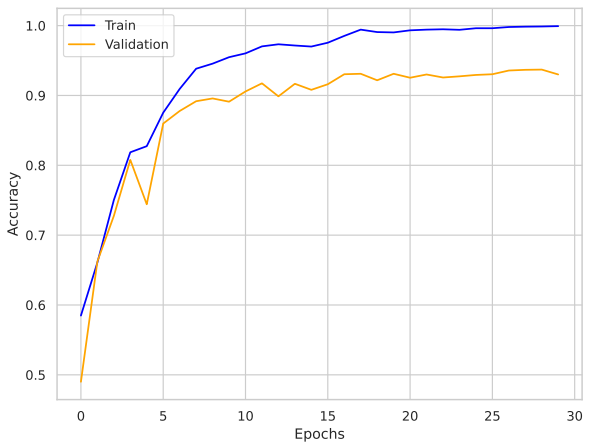



(b)

FIG. 10: Loss (a) and accuracy (b) of training and validation datasets as a function of epochs. Here, learning rate is $lr = 0.001$, the weight decay rate is $wd = 0.05$ and the total number of epochs is 20.




(a)

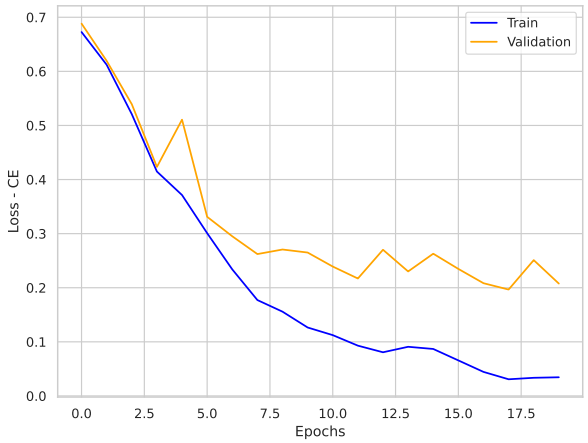



(b)

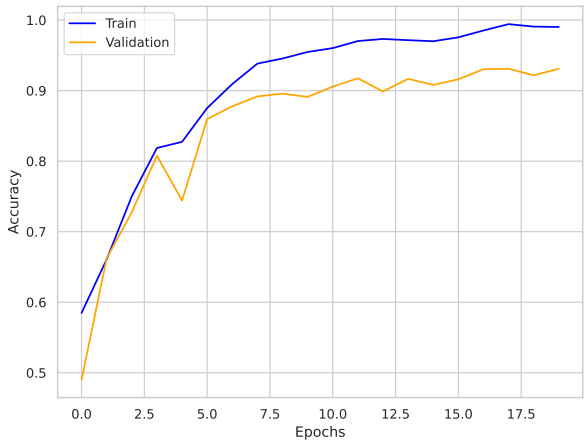
FIG. 11: Loss (a) and accuracy (b) of training and validation datasets as a function of epochs. Here, learning rate is $lr = 0.001$, weight decay rate is $wd = 0$ and the total number of epochs is 30.

gence of the loss function and the accuracy, but was not better at correctly classifying the images in comparison with a learning rate of 0.01. As we can see in Fig.(15 b), the model trained using a learning rate of 0.001 and no weight decay was able to classify the images correctly for both number of epochs (20 and 30) albeit slightly better for 30 epochs than 20. However, the same model trained using a slightly higher learning rate of 0.01 without weight decay was able to classify images equally as well for both 20 and 30 epochs, with 1% difference (see Fig. (16)). Nevertheless, a model trained with the same learning rate of 0.01 and weight decay of 0.05 was much worse at predicting the two classes for both epochs - the accuracy was 84 % for both epochs (see Fig. (14)).

Given that we want our model to be time efficient, it is perhaps useful to choose a greater learning rate of 0.01 without any weight decay and train the model for 20 epochs to achieve reasonable results.
To test this hypothesis, we trained our model using this set of hyperparameters (20 epochs and learning rate of 0.01) on a larger dataset consisting of a total 22300 images. We trained the model on 8927 images and tested it on 6695, and got an accuracy of 89% (see confusion matrix in Fig.(18)), where the model was more capable of correctly predicting images with T1 transitions than those without. Given this result, it is likely that the performance of a model can be improved by either reducing the learning rate slightly or adding

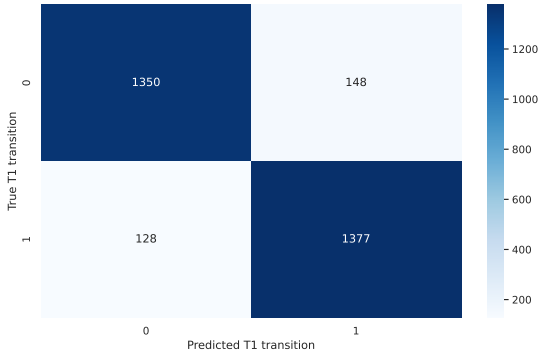



(a)

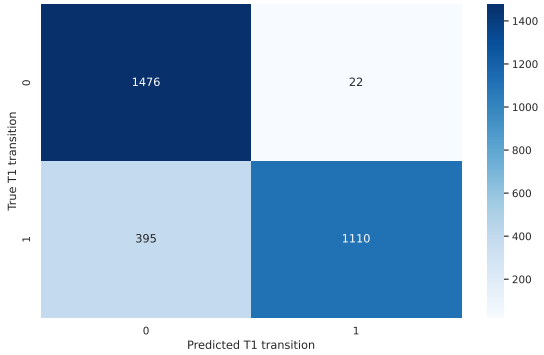



(b)

FIG. 12: Loss (a) and accuracy (b) of training and validation datasets as a function of epochs. Here, learning rate is $lr = 0.001$, the weight decay rate is $wd = 0$ and the total number of epochs is 20.




(a)




(b)

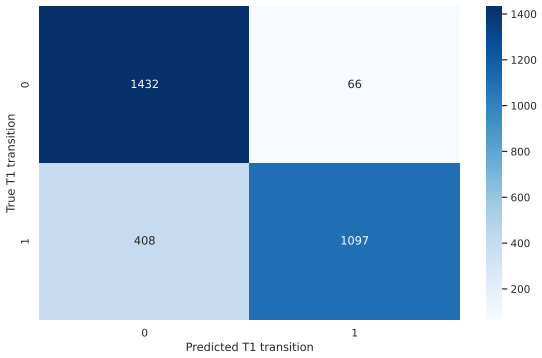
FIG. 13: Confusion matrices for unseen test data with a model trained for a) 20 epochs, 90 % accuracy and b) 30 epochs, 92 % accuracy. Learning rate was $lr = 0.001$ and weight decay rate was $wd = 0.05$.

weight decay - as seen from the loss function in Fig.(17 a)), the model might be on its way towards overfitting, as the validation loss is increasing as the training loss is decreasing.

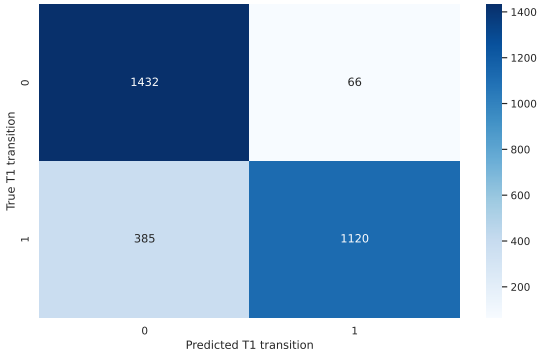
However, it is hard to conclude whether a weight decay parameter was necessary when training the model on a smaller dataset. When testing the model on a smaller, unseen dataset, we see that the accuracy of the predictions did not improve with the addition of a weight decay - this is clear when comparing accuracies for 20 and 30 epochs with a learning rate of 0.001 with and without weight decays in Figs.(13) and (15). Based on the confusion matrices, we can clearly see that the model is more balanced and is able to predict images with and without T1 transitions equally well when trained without a weight decay for both 20 and 30 epochs (see Fig.(15)) than with (see Fig.(13)). Perhaps the weight decay parameter chosen was too big, and penalises the weights more than necessary.

### B. On data normalisation

Prior to normalising the data input, we noticed that weight decay made a difference in the accuracy and provided us with better results. However, as mentioned earlier, upon the realisation of normalisation of data and its importance, we noticed that weight decay term is not necessary given that a sufficient learning rate has been chosen. For both the small and big datasets, training, validation and test data have been normalised using min-max scaling introduced in the Theory section. Opinions on whether the test data should also be normalised are very divided, as it might not be necessary to

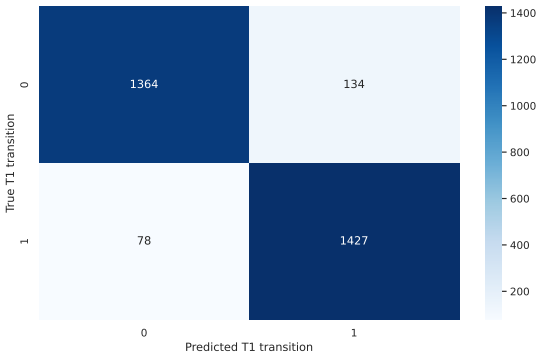



(a)

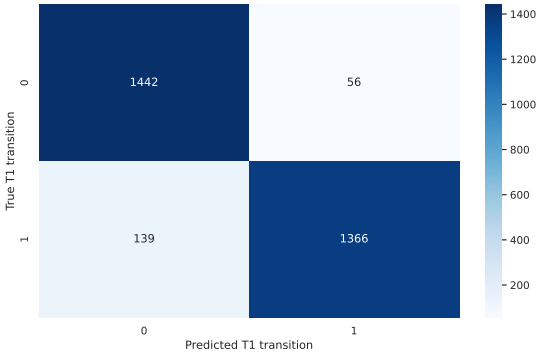



(b)

FIG. 14: Confusion matrices for unseen test data with a model trained for a) 20 epochs, 84 % accuracy and b) 30 epochs, 84 % accuracy. Learning rate was $lr = 0.01$ and weight decay rate was $wd = 0.05$.




(a)




(b)

FIG. 15: Confusion matrices for unseen test data with a model trained for a) 20 epochs, 82 % accuracy and b) 30 epochs, 93 % accuracy. Learning rate was $lr = 0.001$ and weight decay rate was $wd = 0$.

normalise the training data as the model should be generalised enough not to require that. Given the ambiguity of the conclusions, we have chosen to normalise the entire dataset including the unseen test data. It could be worth checking whether the model will perform as well with unnormalised test data and compare the results.

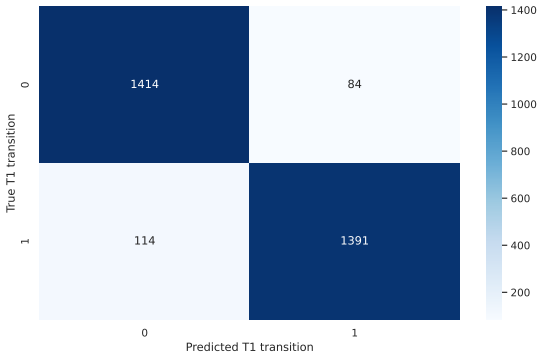
### C. Potential issues

To evaluate the performance of the model critically, we have to be mindful of potential correlations between the inputs of the model.
When generating an entire layer of cells during a simulation (see Fig.(6)), more than one T1 transitions can arise. This might become a problem when splitting the tissue image into smaller images of 4-5 cells, as multiple T1 transitions can occur next to each other. That way, some of the images might capture the same T1 transition and not be fully independent of each other in terms of spatial resolution.

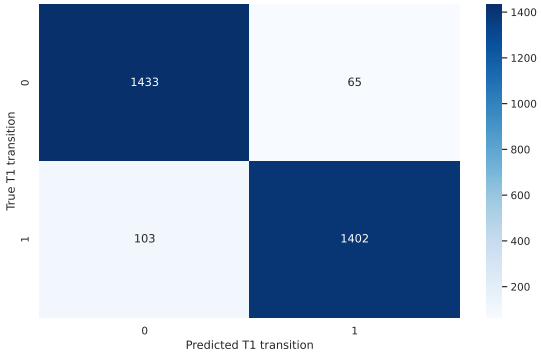
However, we ensure that the images loaded into the model are shuffled and split randomly - this is done to reduce any bias and enhance the generalisation of the model while also excluding any temporal dependencies of the consecutive images during the data simulation process.

### D. Relation to experimental results

Another important aspect that needs to be addressed is how our model can be used to evaluate actual, experimental data. Normal image sizes for experimental data are around 1000x1000 pixels, if not more. Our images have been resized to 32x32, which is the original input of LeNet-5's data, and resizing the images has not been an issue. However, what can be an issue is whether the size of the simulated cell in an image is in agreement with the size of actual real life cells, and whether it is still feasible to use transfer learning using a model trained on this




(a)



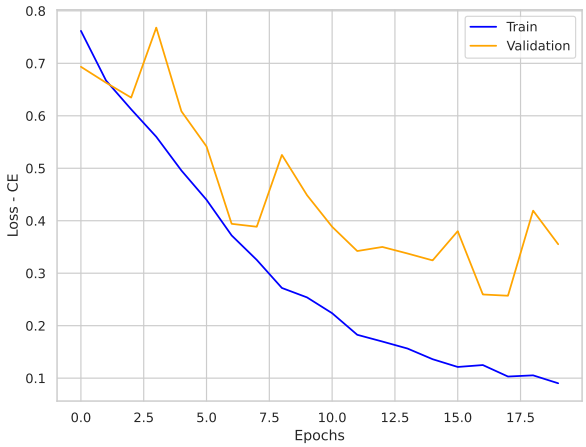

(b)

FIG. 16: Confusion matrices for unseen test data with a model trained for a) 20 epochs, 93 % accuracy and b) 30 epochs, 94 % accuracy. Learning rate was $lr = 0.01$ and weight decay rate was $wd = 0$.

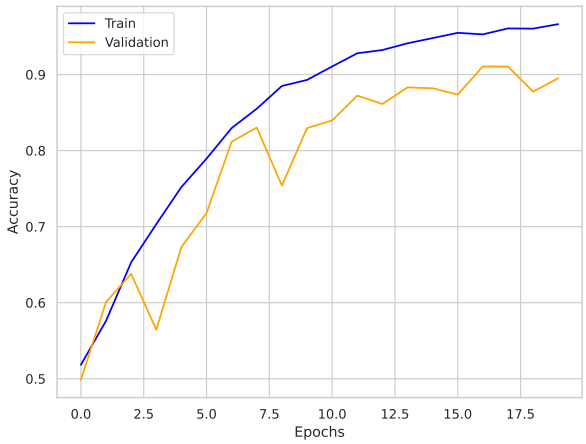
dataset.

## VI. CONCLUSION

From this study, we can conclude that for a smaller dataset consisting of 10008 images in total, a learning rate of 0.01 with no weight decay yields better results. The accuracy of the network with this set up was 94 % using 30 epochs. Based on our results, a larger dataset with the same set of hyperparameters might need adjustment for the learning rate, or an additional weight decay parameter to prevent overfitting. This is because the evaluation of our model on a larger dataset yielded an accuracy of 89 %, which is slightly lower than the one with a smaller dataset.
We have found that a weight decay of 0.05 might be too large for the smaller dataset, as the accuracies with optimal learning rates and the chosen value of weight decay did not improve the performance of the network.
Further studies can be done to evaluate the performance




(a)




(b)

FIG. 17: Loss a) and accuracy b) for big data set using learning rate lr = 0.01 and no weight decay. The model was trained for a total of 20 epochs.

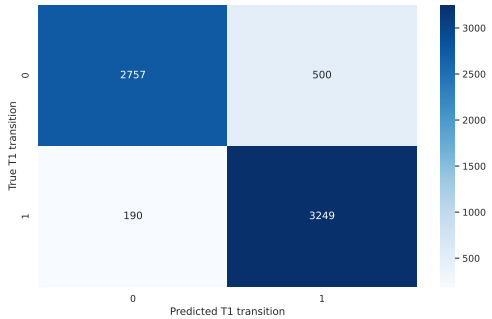



FIG. 18: Confusion matrix for a big test dataset using learning rate lr = 0.01 and no weight decay. The model was trained for a total of 20 epochs. Accuracy is 89 %

of the network on larger datasets using a weight decay to prevent overfitting. Another aspect to take into consideration is the size of the images fed into the network, and more specifically, how well the cell size in the simulated images corresponds to the real size of the cells. This is important as we want to implement the network for future recognition of cells undergoing T1 transitions in real life.

To further improve the performance of the network, one can also consider looking into other hyperparameters that can be adjusted, such as the number of layers and number of nodes in each layer of the network.

---


[1] Amor, E. (2020). Understanding non-linear activation functions in neural networks. https://medium.com/ml-cheat-sheet/understanding-non-linear-activation-functions-in-neural-networks-152f5e101eeb.

[2] Andrey, N. (2017). How to debud neural networks. manual. https://medium.com/machine-learning-world/how-to-debug-neural-networks-manual-dc2a200f10f2.

[3] Atoti (2022). When to perform a feature scaling? https://www.atoti.io/articles/when-to-perform-a-feature-scaling/.

[4] Bangar, S. (2022). Lenet 5 architecture explained. https://medium.com/@siddheshb008/lenet-5-architecture-explained-3b559cb2d52b.

[5] Brownlee, J. (2017). A gentle introduction to exploding gradients in neural networks. https://machinelearningmastery.com/exploding-gradients-in-neural-networks/.

[6] Chen, B. (2021). Why rectified linear unit (relu) in deep learning and the best practice to use it with tensorflow. https://towardsdatascience.com/why-rectified-linear-unit-relu-in-deep-learning-and-the-best-practice-to-use-it-with-tensorflow-e9880933b7ef.

[7] Dawid, A. e. a. (2020). Phase detection with neural networks: interpreting the black box.

[8] Doshi, K. (2022). Batch norm explained visually. https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739.

[9] Dumoulin, V. and Visin, F. (2018). A guide to convolution arithmetic for deep learning.

[10] E, S. K. (2023). Convolutional neural networks. https://developersbreach.com/convolution-neural-network-deep-learning/.

[11] et al, D. A. (2020). Controlled neighbor exchanges drive glassy behaviour, intermittency, and cell streaming in epithelial tissues.

[12] et al, E. R. (2015). Interplay of cell dynamics and epithelial tension during morphogenesis of the drosophila pupal wing.

[13] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[14] Jain, H. (2023). Robust statistical properties of t1 transitions in confluent cell tissues.

[15] Kimihko, F. and Sakai, K. (2021). Can a cnn trained on the ising model detect the phase transition of the q-state potts model?

[16] Ladoux Benoit, M. R.-M. (2017). Mechanobiology of collective cell behaviours.

[17] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998a). Gradient-based learning applied to document recognition. 86(11):2278–2324.

[18] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998b). Gradientbased learning applied to document recognition.

[19] Luthfi, R. (2021). Neural network: The dead neuron. https://towardsdatascience.com/neural-network-the-dead-neuron-eaa92e575748.

[20] Marius, P. and Stringer, C. (2022). Cellpose 2.0: how to train your own model.

[21] Mishra, M. (2020). Convolutional neural networks, explained. https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939.

[22] Sebastian, R. (2023). Creating a confusion matrix. https://rasbt.github.io/mlxtend/user_guide/evaluate/confusion_matrix/.