Can liquid crystal phases be identified via machine learning?

Joshua Heaton 10133722

School of Physics and Astronomy, The University of Manchester

MPhys project report

December 22, 2020

Abstract

hi

1 Introduction

Machine learning methods have seen widespread utilisation across all scientific disciplines, in situations where conventional algorithms are too cumbersome to implement for specific data-based and modelling tasks [1]. Deep learning, loosely defined as machine learning with large datasets, parallel computation and scalable algorithms with many layers [2], has and continues to increase the range and complexity of possible applications of machine learning in the sciences [1]. Any task applying deep learning to data with a grid-like form, such as images, likely involves the usage of convolutional neural network (CNN) algorithms [2]. CNNs were conceived in 1989 by Yann LeCun *et al.* and successfully applied to recognition of handwritten characters [3]. However, their astounding performance in the field of computer vision would not be fully realised until after breakthroughs in deep learning starting in 2006 [2]. Their efficacy was further proven when Geoffrey Hinton *et al.* entered a CNN into the ImageNet Large Scale Visual Recognition Challenge in 2012, and won by a large margin [4].

Liquid crystal phases are in general identified by eye, directly from textures taken by polarised microscopy. Without adequate experience, this can prove a difficult task because certain unique liquid crystal phases, generated by often minor changes in structural properties, can have similar textural appearances [5]. Our project aims to test the viability of machine learning algorithms as tools to assist phase identification. CNNs are particularly suitable due to their prevalence in image classification, and so form the core of our investigations. Current literature in this specific topic is limited, and the approaches so far have mostly involved the usage of simulated textures in the training of models [6, 7]. Sigaki et al. have demonstrated the viability of CNNs in isotropic and nematic phase texture classification and in the prediction of physical liquid crystal properties [6]. Our study further explores and attempts to push the limits of the classification task across a wider range of phases, utilising real experimental data produced by polarised microscopy.

This project report will first provide a brief overview of the physics behind liquid crystals and the capturing of their textures by polarised microscopy, as well as an introduction to machine learning, neural networks and CNNs. The details and results of our investigations into phase classification will then be presented, as well as an outlook to further study.

2 Liquid crystal phases

Liquid crystals are substances in a state between that of a fully isotropic liquid and a crystal with a periodic lattice structure [8, 5]. The molecules can have varying positional order, and have orientational order over large sections. The unit vector parallel to the alignment of the molecules is called the director [8, 5]. Other details such as molecular shape and chirality affect the overall structure. These variations in structure result in numerous individual identifiable liquid crystal phases [8, 5]. Thermotropic liquid crystals exhibit phases transitions with changing temperature, whereas lyotropic liquid crystals are dissolved in a solvent with the phase depending on the concentration [8]. This project will be concerned with only thermotropic liquid crystals.

When cooling a thermotropic liquid crystal starting as an isotropic liquid, it will first transition to the nematic phase (N), which has orientational order only. The chiral nematic

(cholesteric, N*), phase also has no positional order, and has a periodic variation of the director, resulting in helical structures. Upon further cooling, the smectic phase will be reached. This can be split into three categories, going from fluid smectic to hexatic smectic to soft crystal in order of decreasing temperature. The fluid smectic phase has molecules arranged in layers, with no positional order in the plane of each layer. When the director is perpendicular to the layer planes, the phase is smectic A (SmA), with smectic C (SmC) having a director that is tilted by comparison. Hexatic smectic phases have short range positional order within the layer planes with hexagonal intermolecular arrangements interspersed with order-breaking defects. This encompasses the smectic B (SmB), I (SmI), and F (SmF) phases. Semctic B has a director perpendicular to the layer planes, whereas it is tilted towards the vertices of the hexagons for smectic I and towards to the sides of the hexagons for smectic F. The soft crystal phases are defect free within the layers and therefore exhibit long range positional order [5].

The liquid crystal texture data used in this project have all been obtained by polarised microscopy captured with a video camera. In brief terms, a polarising microscope works by placing a sample between perpendicularly aligned polarisers. When light is shone through the arrangement the resulting image will be dark, unless the sample rotates the plane of polarisation [5]. In the case of liquid crystals, the isotropic liquid phase has no optical properties so will produce completely dark textures. The nematic, cholesteric and smectic phases are anisotropic and therefore birefringent, with optical axes depending on their structures. This produces unique textural image features for each phase [5]. Some example textures taken from our dataset are displayed in Figure 1.

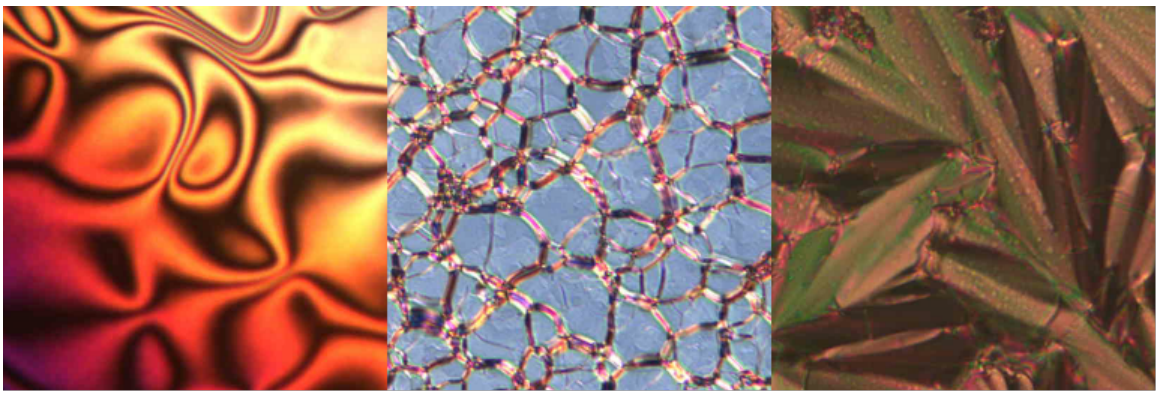

Figure 1: Liquid crystal textures from the dataset, from left to right: nematic phase compound 5CB, cholesteric phase compound D5, and smectic C phase compound M10.

3 General machine learning principles

A machine learning model is a computer algorithm which automatically improves its performance in a given task as it gains experience from a dataset [2]. It learns patterns in data and uses these patterns to make probabilistic predictions. The data normally takes the form of a set of N examples, which are usually expressed as vectors, or some other structure of features, $\{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$, containing quantitative information about example i [9]. In supervised learning the

examples are given labels $y^{(i)}$, to form a training set of pairs $\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i=1}^{N}$, and the model attempts to learn the mapping from a general input $\boldsymbol{x}$ to an output $\hat{y}$. One main type of supervised learning is regression, in which the output is a numerical scalar. The other main type is classification, in which the model predicts what the input belongs to out of a selection of classes. In unsupervised learning there are no labels, and the algorithm attempts to learn specific patterns in the dataset such as clusters of similar data points [9]. The topic of this project is a supervised classification problem.

A supervised model can be usually be expressed as a function of inputs and a set of parameters $\boldsymbol{\theta}$ such that $\hat{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$. Training involves optimisation of the parameters by minimisation of a cost function $J(\boldsymbol{\theta})$, which measures the deviation of the predictions of the model from the true labels. The most common optimisation algorithms involve computing the gradient of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$ [2].

The capacity of a model is akin to its complexity. The number of trainable parameters can give a fast indication of capacity. However, it also depends on the model's functional form. The parameters controlling the capacity of the model, as well as certain other training settings, are known as hyperparameters [2]. If the capacity is too small, the model will tend to "underfit" the training set, resulting in poor performance even when optimised well. On the other hand, too high a capacity will result in "overfitting", with high performance on the training set, but the model may have a high generalisation error, which is the model's error rate when evaluating it on new, unseen data [9, 2]. Before training begins, the entire dataset is often split into training, validation, and test sets, containing N_{train}, N_{valid} and N_{test} examples respectively. The training set, as defined previously, is used to optimise the parameters. The model's performance is then evaluated on the validation set. A poor performance on both the training and validation sets is indicative of underfitting, whereas a high performance on the training set and low on the validation set suggests overfitting. The validation set can therefore be used to tune the hyperparameters of the model before retraining. This can be repeated until the model fits optimally [9, 2]. The final model is then evaluated on the as-of-yet unseen test set to provide an estimate of its generalisation error [9]. Methods used to reduce generalisation error, such as reducing model capacity, are known as regularisation. [2].

Data leakage, in the case of supervised learning, is when there are examples in the validation or test sets with a high degree of similarity to those in the training set. The model will easily produce the correct output when evaluated on the leaked examples, especially when overfitting has occurred. This can result in a false indication of low generalisation error. Therefore, data leakage must be avoided in order to produce a reliable model [10].

4 Feedforward neural networks

4.1 Forward propagation

A neural network is a type of machine learning model that takes inspiration from the current understanding of how the brain works [11]. The inputs are forward propagated through a series of connected hidden units, akin to neurons in a brain, before reaching the output units. In the most basic form, a fully connected neural network, The units are arranged into layers, with each unit in a layer connected to every unit of the previous layer [12]. We will define

the total number of layers, excluding the input layer, as the depth, D, of the model, with the width, W_l, of layer $l \in [0..D]$ equal to the number of units it contains. $l = 0$ is the input layer. The choice of the hyperparameters D and W_l defines the architecture of the model [13]. A schematic of an example network is presented in Figure 2.

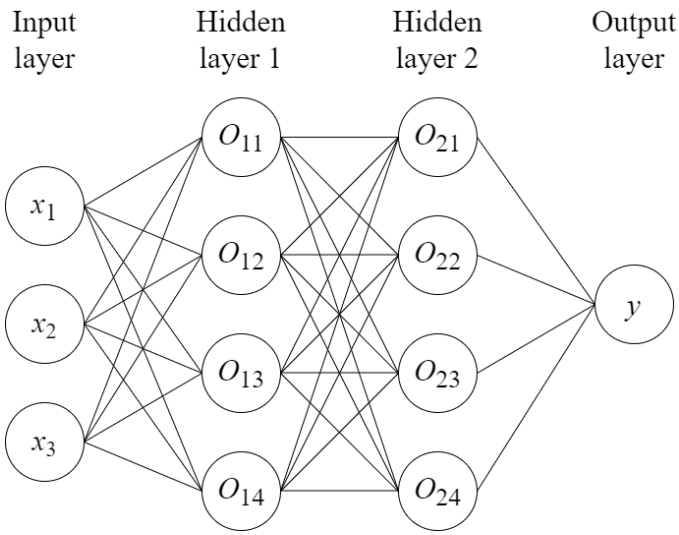

Figure 2: Diagram of a feedforward fully-connected neural network with three input values, $D = 3$, $W_1 = W_2 = 4$ and one output unit.

A single hidden unit's output value is calculated by multiplying the output of each of the previous layer's units with a weight parameter, summing these together with a bias parameter, and then passing the result through a non-linear activation function [12]. More formally, the output O_{lu} of hidden unit $u \in [1..W_l]$, of layer l, with bias parameter b_{lu} and activation function $A(h)$ is calculated as

$$O_{lu} = A(h_{lu}) \tag{1}$$

with

$$h_{lu} = b_{lu} + \sum_{v=1}^{W_{l-1}} \theta_{luv} O_{(l-1)v} \tag{2}$$

for $l > 0$. θ_{luv} is the weight parameter that hidden unit u of layer l applies to the output of unit $v \in [1..W_{l-1}]$ of the previous layer [13]. O_{0u}, is equivalent to value x_u of the the input vector $\boldsymbol{x}$. In matrix form,

$$\boldsymbol{O}_l = A(\boldsymbol{h}_l) \tag{3}$$

with

$$\boldsymbol{h}_l = \boldsymbol{\theta}_l \boldsymbol{O}_{l-1} + \boldsymbol{b}_l \tag{4}$$

where $\boldsymbol{\theta}_l$ is the $W_l \times (W_{l-1})$ dimensional matrix of weights for layer l, with rows corresponding to the weights of each hidden unit, $\boldsymbol{O}_l$ is the vector of outputs for layer l, and $\boldsymbol{b}_l$ is the vector of bias parameters for layer l. For the first layer,

$$\boldsymbol{h}_1 = \boldsymbol{\theta}_1 \boldsymbol{x} + \boldsymbol{b_1} \text{ [13].} \tag{5}$$

The activation function is applied element-wise. A highly effective choice of activation function for hidden units, that performs a similar operation to biological neurons, is the rectified linear unit (ReLU),

$$A(h) = \max(0, h) \text{ [14].} \tag{6}$$

The number of units in the final output layer depends on the type of model. For regression, there will be one unit that outputs a continuous-valued prediction [2]. For classification, the number of final layer units is equal to the number of classes, $C = W_D$, with each one outputting the predicted probability that an input $\boldsymbol{x}$ belongs to a particular class. The network is therefore summarised as $\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{\theta})$, with $\hat{\boldsymbol{y}}$ being the C dimensional vector of output probabilities. Each example in the dataset is labelled by a vector $\boldsymbol{y}$ with a value of one for the component corresponding to the true class, and zero for all other components. The component y_u, with $u \in [1..C]$, is equivalent to the Kronecker delta δ_{ut} where the index t corresponds to the true class [2]. The most common choice of final layer activation function for classification is the softmax function, $\sigma_{SM}(h)$, in which case the components $\hat{y}_u$ of the output probability prediction vector, are given by

$$\hat{y}_u = O_{Du} = \frac{e^{h_{Du}}}{\sum_{v=1}^{C} e^{h_{Dv}}} \text{ [2].} \tag{7}$$

When there are just two classes, known as binary classification, only one unit is needed in the output layer. The data labels y are equal to one or zero depending on the class that $\boldsymbol{x}$ belongs to [2]. The activation function in this case is generally the logistic sigmoid function, $\sigma(h)$, with

$$\hat{y} = O_D = \frac{1}{1 + e^{-h_D}} \tag{8}$$

where in this situation $\boldsymbol{\theta}_D$ is a W_{D-1} dimensional vector. The value of $\hat{y}$ is the predicted probability that $\boldsymbol{x}$ belongs to one of the classes, with a probability of $1 - \hat{y}$ that it belongs to the other [2].

In 1989 Kurt Hornik *et al.* mathematically proved that feedforward neural networks with multiple layers and non-linear activations can approximate any continuous function given the correct configuration [15]. Another advantage of neural networks is that they can automatically learn to extract useful higher-level features from the raw input data [13].

4.2 Training

Neural network training starts with random initialisation of the weights, for example by drawing values from a normal distribution [2]. A single update step is generally carried out by calculating the model output for each example, followed by the gradient of the cost function with respect to the model weight parameters, $\boldsymbol{g} = \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. An iterative optimisation algorithm then uses the gradient to update the parameters to reduce the cost [2]. $\boldsymbol{g}$ is obtained using the backpropagation algorithm, which calculates the gradient of $J(\boldsymbol{\theta})$ with respect to the final outputs, and then recursively applies the chain rule going backwards through the network, calculating the derivatives with respect to the outputs of each hidden unit followed by their parameters [13]. Backpropagation is detailed in Appendix I. The overall goal of

training iterations is to reduce the generalisation error of the model [2]. Generally, the cost function is the expectation value of the loss of all examples in the training set,

$$J(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \tag{9}$$

where the loss is the cross-entropy between the model's output probabilities and the input's true class label,

$$L(f(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{y}) = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{u=1}^{C} y_u \log(\hat{y}_u) \text{ [2].} \tag{10}$$

When training a classifier model, the softmax activation function of the final layer can be included in the loss function, to give the categorical cross-entropy,

$$L_{CCE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{u=1}^{C} y_u \log \sigma_{SM}(h_{Du}) = -\log\left(\frac{e^{h_{Dt}}}{\sum_{j=1}^{C} e^{h_{Dj}}}\right) \tag{11}$$

where we have used the fact that $y_u = \delta_{ut}$ for classification [16]. For binary classification in which the logistic sigmoid activation function is used, the binary cross-entropy is

$$L_{BCE}(\hat{y}, y) = \begin{cases} -\log \sigma(h_D) & \text{if } y^{(\text{true})} = 1 \\ -\log\left(1 - \sigma(h_D)\right) & \text{if } y^{(\text{true})} = 0 \end{cases} \text{ [17].} \tag{12}$$

Calculating the exact derivative of the cost function is extremely computationally expensive in most situations when N_{train} is large. Instead, an approximation of the gradient, $\hat{\boldsymbol{g}}$, is calculated by randomly sampling a small batch of m examples from the training data, giving

$$\hat{\boldsymbol{g}} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \text{ [2].} \tag{13}$$

In general the performance loss from the difference between $\hat{\boldsymbol{g}}$ and the exact gradient $\boldsymbol{g}$ is outweighed by the greatly decreased training step computation time [2]. Optimisation methods that use this random batch sampling are known as stochastic methods. Typically parameter update steps are performed batch by batch, with no duplicate example selections, until the whole training set has been seen by the model. After this all examples are again available for selection. Such a cycle is known as an epoch of training, with floor(N_{train}/m) update steps [2].

Stochastic gradient descent (SGD) is a basic batch-based optimisation algorithm in which the parameters are updated in the opposite direction of $\hat{\boldsymbol{g}}$,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}} \tag{14}$$

where ϵ is a hyperparameter called the learning rate, which controls the amount by which the parameters change with each update [18]. ϵ must be chosen carefully because it greatly affects the training stability and duration [2]. For large neural networks the cost function has a highly irregular multi-dimensional form with many local minima, meaning that in general

the final model will not find the global minimum. However, a local minimum is often enough to achieve low generalisation error [2]. For all models trained in this project, the Adam optimisation algorithm was used, detailed in Appendix II. This is a type of SGD in which the learning rate at each step is adjusted based on unbiased estimates of the first and second moments of the gradient. This algorithm is less prone to becoming stuck in shallow local minima and is not too sensitive to the choice of its hyperparameters [19].

4.3 Regularisation

Neural networks can be prone to overfitting, especially ones with a high capacity or where the dataset is small. Good regularisation is therefore a requirement for models to perform well on the test set [2]. There are a variety of strategies, from which the key ones used in this project are dataset augmentation, early stopping, dropout, and batch normalisation.

Augmentation involves adding extra examples to the training set that are altered versions of the originals, increasing the effective overall number of training examples and improving generalisation. The same effect can also be achieved by performing specific random transformations on each example as they are selected for each training batch. For example, image data can undergo random rotations, reflections, translations, magnification and other transformations. Of course, this must produces images that could still feasibly be a member of the original dataset [2].

If a model is trained for too many epochs, the performance on the training set may still be improving, but the generalisation error will start to increase due to overfitting [20, 2]. Early stopping aims to prevent this. A model performance measure, typically the cost function evaluated on a random batch from the validation set, is monitored during training. If after a specified number of epochs the performance has not improved by more than a tolerance value, the training will be stopped. The number of epochs allowed for improvement is called the patience hyperparameter [2].

When using dropout, at each training step hidden units are randomly chosen to not be included in the step, by multiplying their output by zero. The probability for a unit with dropout to not be included is equal to the dropout rate hyperparameter, which is often set to $\frac{1}{2}$. When applied applied to certain individual layers or to all hidden units, dropout simulates the training of many sub-models that all share parameters, encouraging each hidden unit to learn more general and useful features. This improves regularisation by increasing the final model's robustness to noise [21].

Batch normalisation

5 Convolutional neural networks

5.1 Convolutional layers

CNNs are a type of feedforward neural network in which at least one layer uses the convolution operation to propagate information from the previous layer, instead of the standard fully-connected configuration [2]. They have proven to be extremely effective at processing inputs with a large number of grid-like features, in particular image data [22]. CNNs are trained in

the same way as standard fully connected neural networks, with slight modifications to the backpropagation algorithm [23].

For image-based CNNs, a convolutional layer takes an input tensor $\boldsymbol{I}$ with width W_I, height H_I and depth D_I, where the depth is the number of channels. Grayscale images have one channel, with a value corresponding to the brightness of the pixel at that location, whereas colour images have three channels, corresponding to the red, green and blue values for the pixel [24, 2]. The input is convolved with N_K kernels, which are tensors $\boldsymbol{K}^{(d)}$ with width $W_K < W_I$, height $H_K < H_I$ and depth $D_K = D_I$. This produces a pre-activation output tensor $\boldsymbol{h}$, which in the most basic case has width $W_h = W_I$, height $H_h = H_I$ and depth $D_h = N_K D_I$. $\boldsymbol{I}$ is convolved with each $\boldsymbol{K}^{(l)}$ in turn, with each output stacked together depth-wise to give the total output $\boldsymbol{h}$. Having more than one kernel increases the number of channels in the next layer. This overall operation is given by

$$h_{ijk} = \sum_{m=1}^{H_K} \sum_{n=1}^{W_K} I_{(i-m)(j-n)r} K^{(d)}_{mnr} \tag{15}$$

with $r = k \bmod D_I$ and $d = \lceil \frac{k}{D_I} \rceil$ [24, 2]. Similarly to standard fully connected, or dense, hidden layers, a non-linear activation function is applied to every element of $\boldsymbol{h}$ [24, 2].

Using more than one kernel allows a convolutional layer to extract more features from its input, at the cost of increased computation time and memory usage [2]. The trainable parameters are the elements of the kernels. The kernels can individually learn different features to extract from the input, with the size of the features being related to the height and width dimensions of the kernels [24, 2]. The usage of kernels in convolutional layers is a form of parameter sharing, which is when parameters are used for more than one operation in a model. Compared to a dense layer, many less parameters are needed, resulting in greatly reduced memory consumption, and often improved regularisation [24, 2].

At the edges of the input, we have to consider the type of padding to use. Valid padding is when the kernel is kept completely within the bounds of the input, which results in the output having a smaller width and height than the input depending on the kernel size [24, 2]. The case we have discussed in which the width and height of the output are equal to the input corresponds to same padding, where each pixel is visited by the kernel the same number of times, resulting in it overlapping the boundary of the input at the furthest points. Any kernel parameters outside the input region are multiplied by zero [24, 2].

Another adjustable property of convolutional layers is the stride. This is the number of elements by which the kernel is moved with each step. The stride can be different for the horizontal and vertical directions. The basic case of Equation 15 corresponds to a stride of one in both directions. Strides greater than one result in dimensionality reduction from input to output, with less feature extraction. However, this reduces computational cost for the layer [24, 2].

5.2 Pooling layers

Convolutional layers are often directly followed by pooling. A pooling layer takes rectangles of a specific size, called the pool size, from the output of the previous layer. It outputs single values as a function of the values in each rectangle, similarly to how kernels work in

convolutional layers [24, 2]. However, the function used by a pooling layer does not contain any trainable parameters, and each channel is processed individually [24, 2]. Padding and stride are also defined for pooling layers, in the same way as for convolutional layers. As an example, a pooling layer with pool size and strides of 2×2, valid padding, and input with dimensions $10 \times 10 \times 3$ will produce an output with dimensions $5 \times 5 \times 3$ [24, 2]. There are a variety of pooling functions, such as max pooling, where the outputs are the maximum values in each rectangle, and average pooling, where the outputs are the means of the rectangles [2]. Global pooling layers are a special case with a pool size equal to the width and height of the input, resulting in one output value for each of the input channels. The output is therefore a vector, suitable to be fed into a dense layer [24].

Pooling layers are utilised for two key reasons. Firstly, they can reduce the dimensions of the network between convolutional layers, decreasing computational cost. Secondly, they can make the CNN partially invariant to translations of features in the input [24, 2].

6 Model training set-up

6.1 Universal training configurations

All CNN models in this project are implemented using the TensorFlow machine learning library with Keras high-level API, and trained using NVIDIA CUDA with cuDNN on an NVIDIA RTX 2060 graphics processing unit (GPU). GPUs are typically utilised for training deep models due to their ability to perform thousands of tensor calculations in parallel, resulting in greatly decreased model training time compared with using a standard CPU [25, 2]. Any hyperparameters corresponding to model tensor dimensions, specifically batch size, image input size, and model layer dimensions have been chosen as multiples of powers of two. This results in more efficient usage of GPU memory [2].

For every model, a batch size of $m = 32$, the Adam optimiser with hyperparameters $\epsilon = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\delta = 10^{-7}$ and early stopping monitored by validation set loss and with a patience of 100 epochs is used. Every convolutional or dense hidden layer in all models has ReLU activations and batch normalisation. Every convolutional layer has strides of 1×1, and every dense layer has dropout with rate $\frac{1}{2}$.

We will define sequential neural network model architectures as models without branching layers, and branching architectures as the opposite, in which the outputs of certain layers are shared between multiple subsequent layers. Also, when referring to kernel size, we quote only the width by the height, with the depth inferred from the number of channels in the layer.

Dataset augmentations are implemented automatically during training using the image data generator function in Keras. Batches of images are loaded from the training directory, and specified transformations are applied randomly to each image before being used for training. The original image remains unchanged in the root directory. We define flip augmentations as random flipping of images vertically and horizontally, and all augmentations as flip augmentations along with random rotations by up to 30°, image width and height shifting by up to 10%, and uniform rescaling by up to 20%.

The accuracy metric we use to test our models is the percentage of correct outputs when a model is evaluated on a specific dataset.

6.2 Image data preparation

We obtain liquid crystal texture image data by extracting images frame by frame from polarised microscopy videos, using the VLC Media Player software. A majority of the videos display more than one liquid crystal phase. Phase labels are assigned to each image based on the phase the video was displaying at the time the image was extracted. Images extracted at the point of a phase transition are either labelled with the current dominant phase or discarded if it is unclear. Depending on the pixel dimensions of the images, they are further split into smaller images with the same phase labels. Excess pixels are then cropped so that the images are square, with dimensions larger than the model input dimensions, which are square for every model. The images are then resized to the input dimensions of the model, and converted from three-channel RGB colour mode to one-channel grayscale, with pixel brightness values in the range $[0, 1]$. We have made the assumption that liquid crystal texture identification is independent of colour.

Texture images of the same phase from the same video can be very similar. Therefore, in order to prevent data leakage, these are not shared between any of the training, validation or test sets for a specific model. Conversion to grayscale could also help to prevent data leakage because different videos of the same liquid crystal compound have similar colours.

7 4-phase classifier models

7.1 Dataset construction

The first set of models were created to test the viability of CNNs in the liquid crystal phase identification task, and investigate sequential architectures, input size and dataset augmentations. The selected phases for classification were isotropic, nematic, cholesteric, and smectic. The quantities of images in all prepared sets are presented in Table 1. All images were prepared as in Section 6.2, aside from the isotropic phase, which is completely dark under a polarised microscope. Isotropic images were generated as dark noise by randomly selecting the value of each pixel from the uniform distribution $[0, 0.1]$.

Table 1: Dataset distribution for 4-phase classifier models

	Isotropic	**Nematic**	**Cholesteric**	**Smectic**	**Totals**	**% of total**
Training	1500	1691	1549	1689	6429	71.16
Validation	400	471	405	457	1733	19.18
Test	200	208	178	287	873	9.66
Totals	2100	2370	2132	2433	9035	

7.2 Model architectures and training configurations

A total of 24 models were trained on the 4-phase dataset. Two input image sizes, 256×256 and 128×128, as well as flip and all augmentations were tested for six sequential CNNs,

with an increasing number of convolutional layers from one to six. The loss function used for every model is the categorical cross-entropy. After some preliminary trial and error model training the following settings were decided on. The first convolutional layer in each model has 32 channels, with the number of channels doubling with each successive convolutional layer. Each convolutional layer has kernel size 3×3 and same padding, and is followed by max pooling with pool size and strides of 2×2 and same padding, aside from the final convolutional layer which is followed by global average pooling. This is then followed by a dense layers with 256 units, then 128 units, and 4 units for the output dense layer. As an example, an architectural diagram of one of the models is displayed in Figure 3. For each model, after early stopping,

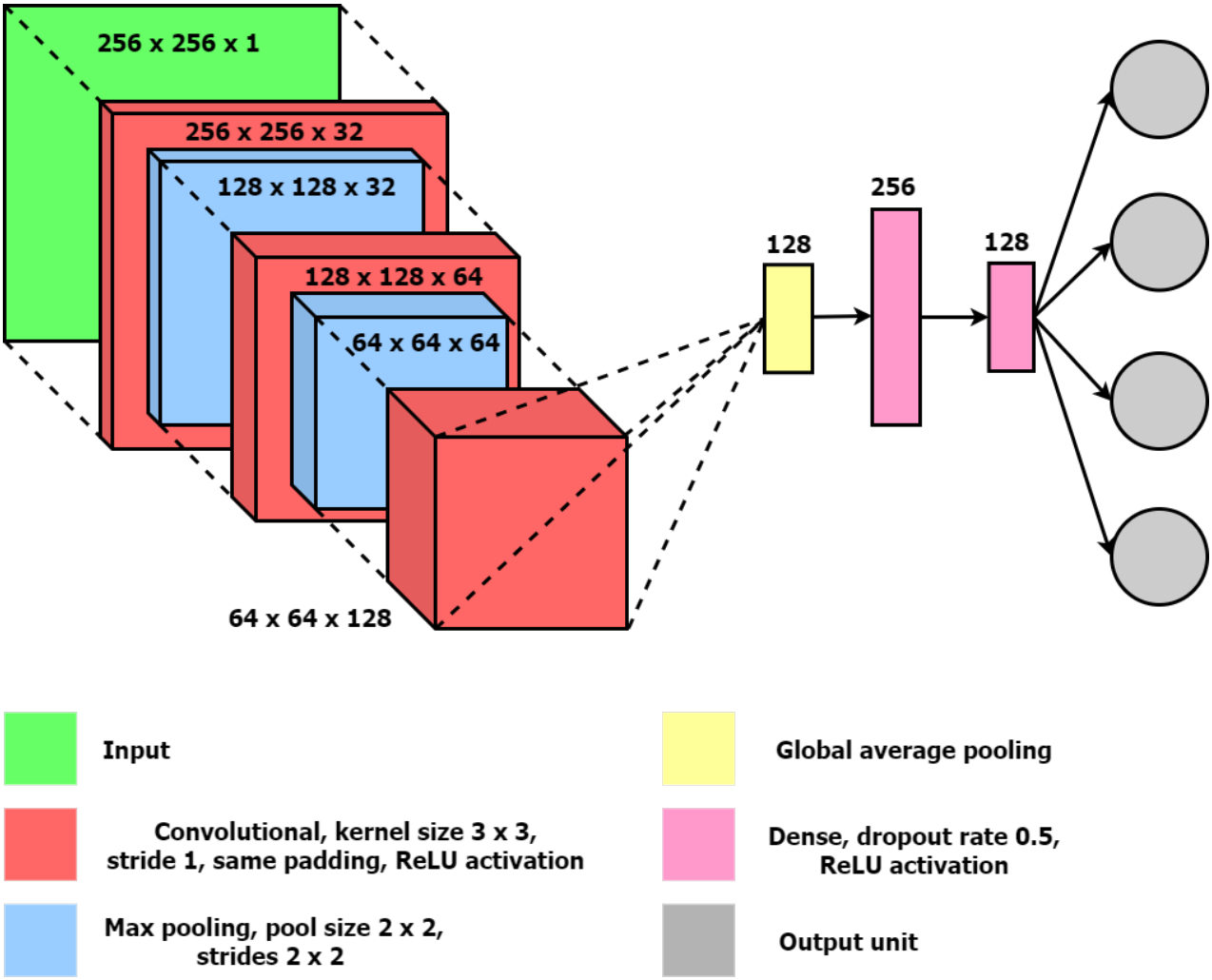

Figure 3: Architectural diagram of the sequential CNN model with input size 256×256 and three convolutional layers. The dimensions are displayed as width $\times$ height $\times$ channels for each convolutional and max pooling layer. Not to scale.

the parameters with the highest validation accuracy during training were saved as the final model.

7.3 Results

The final accuracy of a trained model will vary with each training attempt due to the stochastic nature of parameter initialisation, data augmentation and batch selection. Each of the 24 models were trained three times, and the accuracies calculated for both validation and test sets. The overall validation and test accuracies of a model are taken as the mean of the three attempts, with an uncertainty of half the range in accuracy. The final results are displayed in Figures 4 and 5.

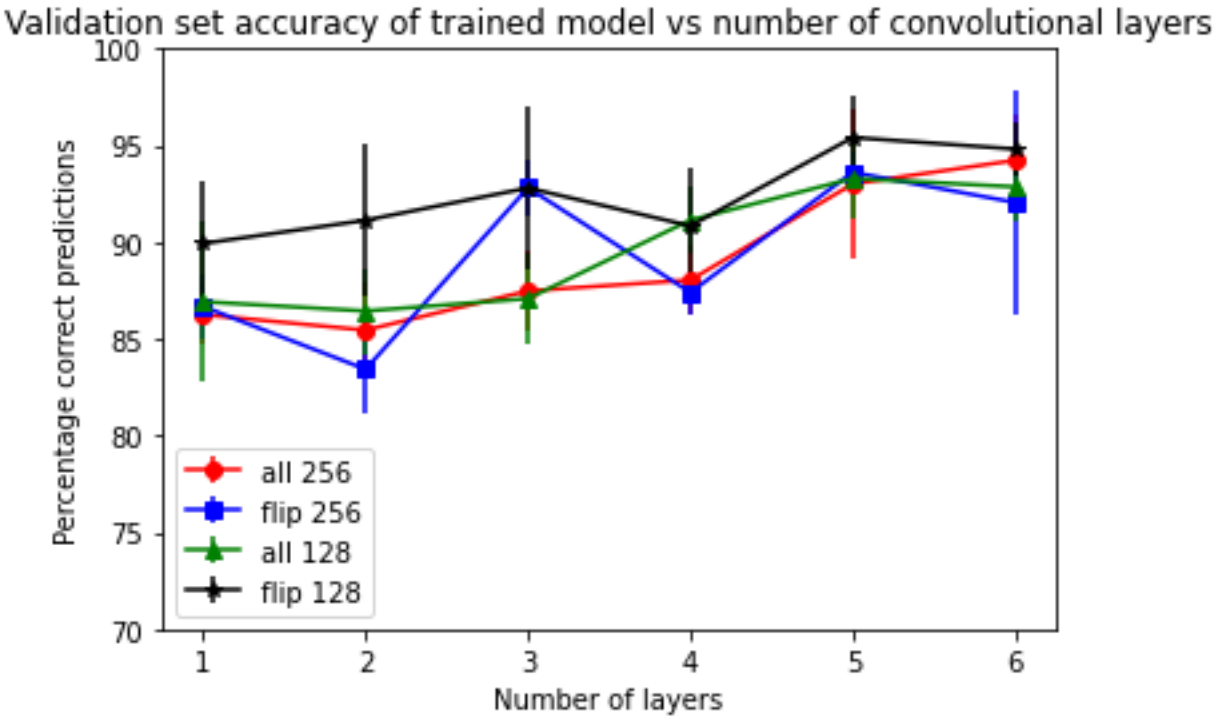

Figure 4: •

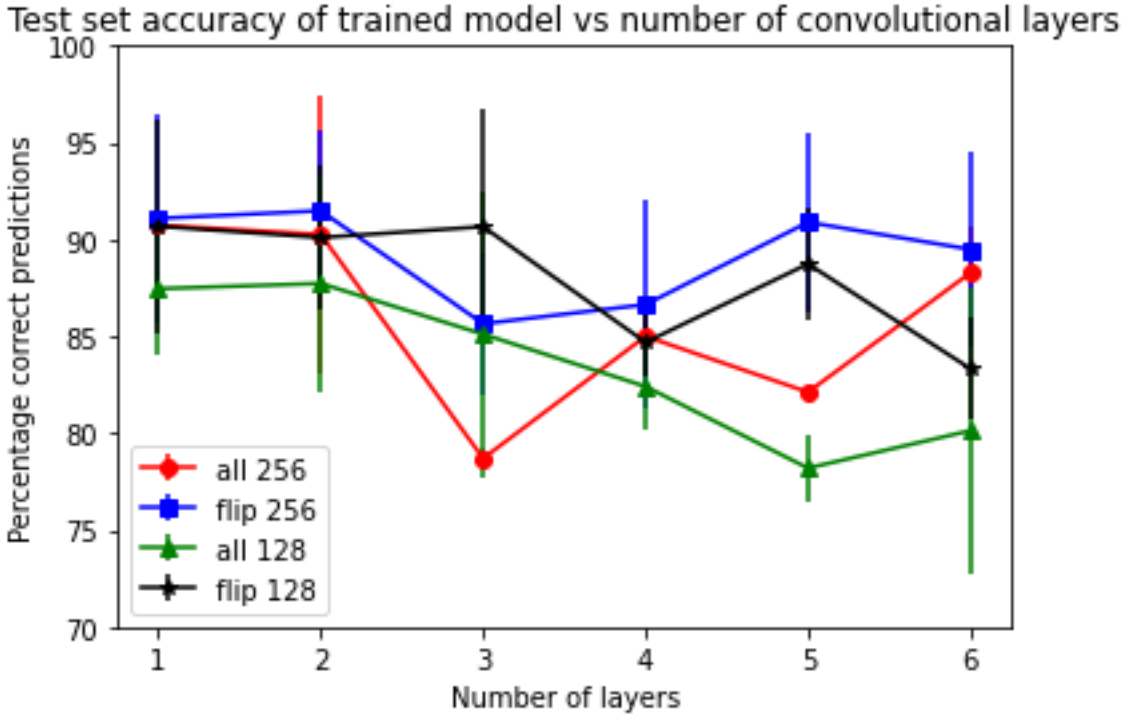

Figure 5: •

8 Smectic A and C binary classifier models

8.1 Dataset distribution

8.2 Model architectures and training configuration

8.3 Results

9 General smectic phase classifier models

9.1 Dataset distribution

9.2 Model architectures and training configuration

9.3 Results

10 Conclusions

11 Going forward

References

[1] G. Carleo *et al.*, "Machine learning and the physical sciences," *Rev. Mod. Phys*, vol. 91, p. 045002, 2019.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[3] Y. LeCun *et al.*, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[4] O. Russakovsky *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[5] I. Dierking, *Textures of Liquid Crystals*. WILEY-VCH, 2003.

[6] H. Y. D. Sigaki *et al.*, "Learning physical properties of liquid crystals with deep convolutional neural networks," *Scientific Reports*, vol. 10, p. 7664, 2020.

[7] E. N. Minor *et al.*, "End-to-end machine learning for experimental physics: using simulated data to train a neural network for object detection in video microscopy," *Soft Matter*, vol. 16, p. 1751, 2020.

[8] D. Demus *et al.*, *Physical Properties of Liquid Crystals*. WILEY-VCH, 1999.

[9] K. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[10] S. Kaufman *et al.*, "Leakage in data mining: Formulation, detection, and avoidance," *ACM Trans. Knowl. Discov. Data*, vol. 6, pp. 556–563, 2012.

[11] M. Minsky and S. A. Papert, *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Internal Representations by Error Propagation*. MIT Press, 1986.

[13] S. Haykin, *Neural Networks: A Comprehensive Foundation*. 2 ed., 1998.

[14] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," *Proc. Mach. Learn. Res*, vol. 15, pp. 315–323, 2011.

[15] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366, 1989.

[16] D. Kline and V. Berardi, "Revisiting squared-error and cross-entropy functions for training neural network classifiers," *Neur. Comp. App.*, vol. 14, pp. 310–318, 2005.

[17] M. D. Richard and R. Lippmann, "Neural network classifiers estimate bayesian a posteriori probabilities," *Neural Computation*, vol. 3, pp. 461–483, 1991.

[18] S. ichi Amari, "Backpropagation and stochastic gradient descent method," *Neurocomputing*, vol. 5, pp. 185–196, 1993.

[19] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 2014.

[20] C. Bishop, "Regularization and complexity control in feed-forward networks," in *Proceedings International Conference on Artificial Neural Networks ICANN'95*, vol. 1, pp. 141–148, EC2 et Cie, 1995.

[21] N. Srivastava *et al.*, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, 2014.

[22] A. Shrestha and A. Mahmood, "Review of deep learning algorithms and architectures," *IEEE Access*, vol. 7, pp. 53040–53065, 2019.

[23] Y. Bengio, Y. Le Cun, and D. Henderson, "Globally trained handwritten word recognizer using spatial representation, convolutional neural networks and hidden markov models," in *Proceedings of the 6th International Conference on Neural Information Processing Systems*, p. 937–944, Morgan Kaufmann Publishers Inc., 1993.

[24] H. H. Aghdam and E. J. Heravi, *Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification*. Springer Publishing Company, Incorporated, 1st ed., 2017.

[25] S. Shi *et al.*, "Benchmarking state-of-the-art deep learning software tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pp. 99–104, 2016.

Appendices

A Backpropagation

B Adam optimiser

Before the first iteration, the biased first and second moment estimate variables $\boldsymbol{s}$ and $\boldsymbol{r}$ are initialised to zero. At the start of an iteration, a batch of data is sampled and $\hat{\boldsymbol{g}}$ is calculated as in Equation 13. $\boldsymbol{s}$ is updated as

$$\boldsymbol{s} \tag{16}$$