Can liquid crystal phases be identified via machine learning?

Joshua Heaton, 10133722

School of Physics and Astronomy, The University of Manchester

MPhys project report

Project performed in collaboration with James Harbon

December 30, 2020

Abstract

In this project, we investigate the application of machine learning models to the challenge of identifying liquid crystal phases directly from their texture images. Branching and sequential convolutional neural networks are applied to three distinct phase classification tasks, utilising training data in the form of liquid crystal texture images extracted from polarised microscopy videos. We obtain high mean test set accuracies, often greater than 90 percent, for sequential models classifying the isotropic, nematic, cholesteric, and smectic phases. Sequential and branching models perform very well in the binary classification of the smectic A and smectic C phases, with all models achieving a mean test set accuracy of above 90 percent. Models are also applied to the classification of the fluid smectic, hexatic smectic, and soft crystal phases. Success in this task is limited so far due to a lack of data for the hexatic smectic and soft crystal phases.

Contents

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 the 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 (ILSVRC) 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. Certain unique liquid crystal phases, often generated by 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 form the core of our investigations. Current literature on 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 the prediction of physical liquid crystal properties [6]. Our study further explores and attempts to push the classification task limits 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 Background theory

2.1 Liquid crystal phases

2.1.1 Structure

Liquid crystals are substances in a state between 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 structural variations result in numerous individual identifiable liquid crystal phases [8, 5]. Thermotropic liquid crystals exhibit phase transitions with changing temperature, whereas lyotropic liquid crystals are dissolved in a solvent with the phase de-

pending on the concentration [8]. This project will be concerned with only thermotropic liquid crystals.

When cooling a thermotropic liquid crystal, starting in the isotropic liquid (Iso) phase, it will first transition to the nematic (N) phase, which has orientational order only. If the chemical is chiral, the chiral nematic or cholesteric (N*) phase will, instead, be present. It also has no positional order and has a periodic variation of the director, resulting in helical structures. Upon further cooling, the smectic (Sm) phase is reached. This can be split into three categories, going from fluid smectic (FSm) to hexatic smectic (HSm) to soft crystal (SC) 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. Smectic B has a director perpendicular to the layer planes, whereas for smectic I it is tilted towards the vertices of the hexagons, and towards 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].

2.1.2 Polarised microscopy

The liquid crystal texture data used in this project have 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 unpolarised light is shone through the arrangement, the resulting image will be dark unless the sample rotates the plane of polarisation [5]. For liquid crystals, the isotropic liquid phase has no optical properties and produces 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.

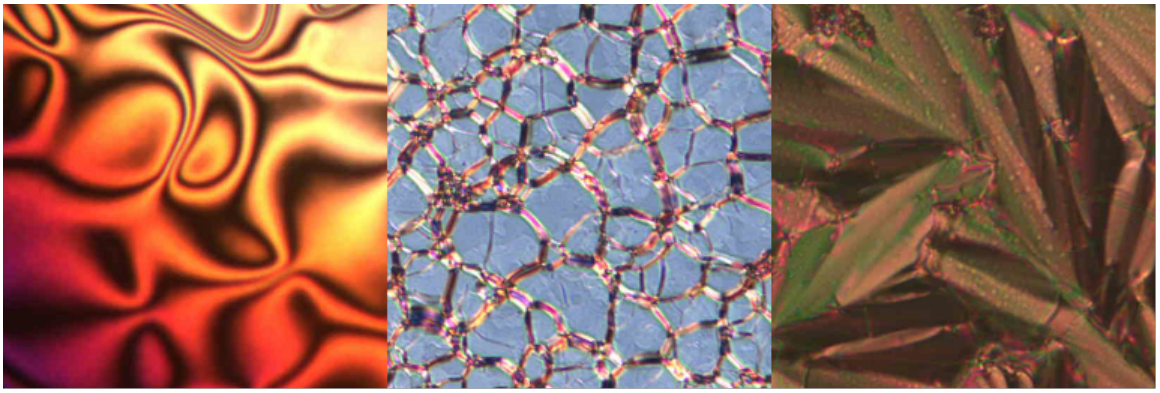

Figure 1: Liquid crystal textures from the dataset, from left to right: nematic phase compound 5CB, cholesteric phase compound D5, and smectic C phase compound M10.

2.2 General machine learning principles

A machine learning model is a computer algorithm that 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. Unsupervised learning does not use labels. 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 usually be expressed as a function of inputs and a set of parameters $\boldsymbol{\theta}$ such that $\hat{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$. The parameters are optimised by minimising a cost function $J(\boldsymbol{\theta})$, which measures the deviation of model predictions 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 model capacity and certain 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. However, the model may have a high generalisation error, which is the 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. Poor performance on both the training and validation sets indicates 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 is repeated until the model fits optimally [9, 2]. The final model is evaluated on the unseen test set, providing an estimation of its generalisation error [9]. Methods used to reduce generalisation error, such as reducing model capacity, are known as regularisation. [2].

Data leakage, with 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 occurs. This can result in a false indication of low generalisation error. Therefore, data leakage must be avoided to produce a reliable model [10].

2.3 Feedforward neural networks

2.3.1 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 simple case of 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 hyperparameters D and W_l defines the architecture of the model [13]. A schematic of an example network is presented in Figure 2.

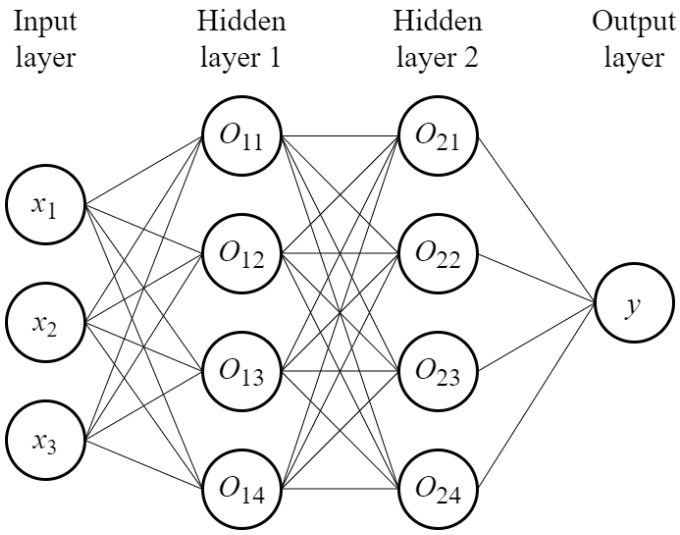

Figure 2: Diagram of a feedforward fully-connected neural network with three input values, $D = 3$, $W_1 = W_2 = 4$, and one output unit.

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

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

for $l > 0$. w_{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 input vector $\boldsymbol{x}$. In matrix form,

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

with

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

where $\boldsymbol{w}_l$ 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 hidden layer,

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

The activation function is applied element-wise. A highly effective choice of activation function for hidden units, performing 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 and $\boldsymbol{\theta}$ the combined representation of all $\boldsymbol{w}_l$ and $\boldsymbol{b}_l$. 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, $\sigma_{SM}(h)$, for which 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

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

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].

2.3.2 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 for neural networks is detailed in Appendix A.1. 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 combined with 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 "minibatch" of m examples from the training data, giving

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

In general, the loss of precision 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 minibatch sampling are known as stochastic methods. Typically, a single parameter update step is performed on each minibatch. There are no duplicate example selections until the model has seen the whole training set. After this, all examples are again available for selection. Such a cycle is known as an epoch of training, with $\lfloor N_{\text{train}}/m \rfloor$ update steps [2].

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

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

where α is a hyperparameter called the learning rate, which controls the amount by which the parameters change with each update [18]. α must be chosen carefully because it greatly affects 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 A.2. 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].

2.3.3 Regularisation

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

Augmentation involves adding altered copies of examples to the training set, increasing the effective overall number of training examples, improving generalisation. For example, image data can undergo random rotations, reflections, translations, magnifications, and other transformations. Of course, this must produce images that could still feasibly be a member of the original dataset [2].

If a model is trained for too many epochs, the training set accuracy 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 minibatch from the validation set, is monitored during training. After a specified number of epochs, if the performance has not improved by more than a tolerance value, training is stopped. The number of epochs allowed for improvement is called the patience hyperparameter [2].

When using dropout, at each training step, random hidden units are not 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 to certain individual layers or all hidden units, dropout simulates training many sub-models that share parameters, encouraging each hidden unit to learn more general and useful features. This improves generalisation by increasing the final model's robustness to noise [21].

When a layer has batch normalisation, the output vector $\boldsymbol{h}$ for a training minibatch is changed according to

$$\boldsymbol{h} \leftarrow \boldsymbol{\gamma} \odot ((\boldsymbol{h} - \boldsymbol{\mu}) \oslash \boldsymbol{\sigma}) + \boldsymbol{\beta} \tag{15}$$

where $\boldsymbol{\mu}$ is the vector of means and $\boldsymbol{\sigma}$ is the vector of standard deviations of the outputs over the whole minibatch. $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are new parameter vectors learned in conjunction with the layer parameters. $\odot$ represents element-wise multiplication and $\oslash$ represents element-wise division of the vectors. This has some desirable effects when applied to each hidden layer of the model. It improves generalisation in a similar way to dropout by introducing noise. Model training stability is increased, allowing greater learning rates to be used, reducing training

time. Running averages for each $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are recorded during training and are used when evaluating the final model on single inputs [22].

2.4 Convolutional neural networks

2.4.1 Convolutional layers

CNNs are a type of feedforward neural network with at least one layer using 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 large grid-like inputs, in particular image data [23]. CNNs are trained in the same way as standard fully connected neural networks, with modifications to the backpropagation algorithm [24].

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 values corresponding to pixel brightness, whereas colour images have three channels, corresponding to the red, green, and blue values for pixels [25, 2]. The input is convolved with N_K kernel tensors, $\boldsymbol{K}^{(p)}$, with $p \in [1..N_K]$. All kernels have 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 $\boldsymbol{K}^{(p)}$ 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

$$h_{ijk} = \sum_{m=1}^{H_K} \sum_{n=1}^{W_K} I_{(i-m)(j-n)r} K^{(p)}_{mnr} \tag{16}$$

with $r = k \bmod D_I$ and $p = \lceil \frac{k}{D_I} \rceil$ [25, 2]. Similarly to standard fully connected, or dense, hidden layers, a non-linear activation function is applied to every element of $\boldsymbol{h}$ [25, 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 learn different input features, with the type of feature related to the kernel dimensions [25, 2]. The usage of kernels in convolutional layers is a form of parameter sharing, where parameters are used for more than one operation in a model. Compared to a dense layer, far fewer parameters are needed, resulting in greatly reduced memory usage and often improved regularisation [25, 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 input bounds, which results in the output having a smaller width and height than the input depending on the kernel size [25, 2]. "Same" padding is when the width and height of the output are equal to the input, with each pixel 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 [25, 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 [25, 2].

2.4.2 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 previous layer's output. It outputs single values as a function of each rectangle of values, similarly to how kernels work in convolutional layers [25, 2]. However, the function used by a pooling layer does not contain any trainable parameters, and each channel is processed individually [25, 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 $5 \times 5 \times 3$ [25, 2]. There are numerous 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 have a pool size equal to the input's width and height, giving one output value for each input channel. The output is a vector suitable to be fed into a dense layer [25].

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 input features [25, 2].

3 Project work

3.1 Model training set-up

3.1.1 Universal training configurations and definitions

All CNN models in this project are implemented using the TensorFlow machine learning library in Python, with the Keras high-level API. They are 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 to using a standard CPU [26, 2]. Any hyperparameters corresponding to model tensor dimensions, specifically minibatch 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 minibatch size of $m = 32$ and the Adam optimiser with hyperparameters $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-7}$ is used. Early stopping monitored by validation set loss and with a patience of 100 epochs is used. During training, model parameters are saved each time a new lowest validation loss is recorded. 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, where outputs of certain layers are shared between multiple subsequent layers. We omit the depth when quoting kernel sizes. It is equal

to the number of channels in the layer. We will refer to max pooling with pool size and strides of 2×2 and same padding as a standard pooling layer.

Dataset augmentations are implemented automatically during training using the image data generator function in Keras. Minibatches of images are loaded from the training directory, and specified transformations are applied randomly to each image before being used for training, without altering the original saved image. We define "flip augmentations" as random flipping of images vertically and horizontally. "All augmentations" are 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 used to test the models is the percentage of outputs matching the true labels when a model is evaluated on a specific dataset.

3.1.2 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. Labels are assigned to each image based on the phase of the video when the image is 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 resolution, they are split into smaller sub-images with the appropriate 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 scaled down to the input dimensions of the model. Next, they are converted from three-channel RGB colour mode to one-channel grayscale, with pixel brightness values in the range $[0, 1]$. We have assumed that individual liquid crystal textural features are identifiable without colour.

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

3.2 Models I, 4-phase classifiers

3.2.1 Dataset construction I

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. The videos used in the construction of this dataset were either downloaded with permission from Vance Williams' YouTube channel and Instagram page [27, 28] or provided by project supervisor Ingo Dierking. All images were prepared as in Section 3.2.2, 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 in the range $[0, 0.1]$.

Table 1: Dataset distribution for 4-phase classifier models.

	Iso	**N**	**N***	**Sm**	**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	

3.2.2 Model architectures and training configurations I

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 input is convolved with 32 kernels, resulting in 32 channels for the first convolutional layer of each model. Two kernels are used for all other layers, doubling the number of channels with each successive convolutional layer. Each convolutional layer has kernel size 3×3 and same padding and is followed by a standard pooling layer, aside from the final convolutional layer which is followed by global average pooling. This is then followed by dense layers with 256 units, then 128 units, and 4 units for the output layer, which has a softmax activation function. As an example, an architectural diagram of the three-layer model is displayed in Figure 8 in Appendix C.

3.2.3 Results I

The final accuracy of a trained model varies each time it is retrained due to the stochastic nature of parameter initialisation, data augmentation, and minibatch selection [2]. The models were trained three times, and the accuracies were calculated for the 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, due to the small sample size. The final results are displayed in Figure 3. From the plots, we see that the validation accuracy has a slight positive correlation with the number of layers, with the opposite true for the test accuracy. However, this trend is not clear for flip augmentations with 256×256 input size. The decrease in test set accuracy with higher numbers of layers is suggestive of overfitting, which is expected as the model capacity increases greatly with each added convolutional layer. The lower capacity models from this selection are, therefore, a better choice. Potentially, the high validation accuracies at greater layer counts are because the models are saved at the point of highest validation accuracy during training, which can fluctuate to high values. The difference between some of the validation and test accuracies could also be due to the imbalance between the number of samples in each set, with almost double the number of validation samples as test.

In every case, for the same input size and number of layers, models with flip augmentations have a higher mean test set accuracy than with all augmentations. This suggests that some of

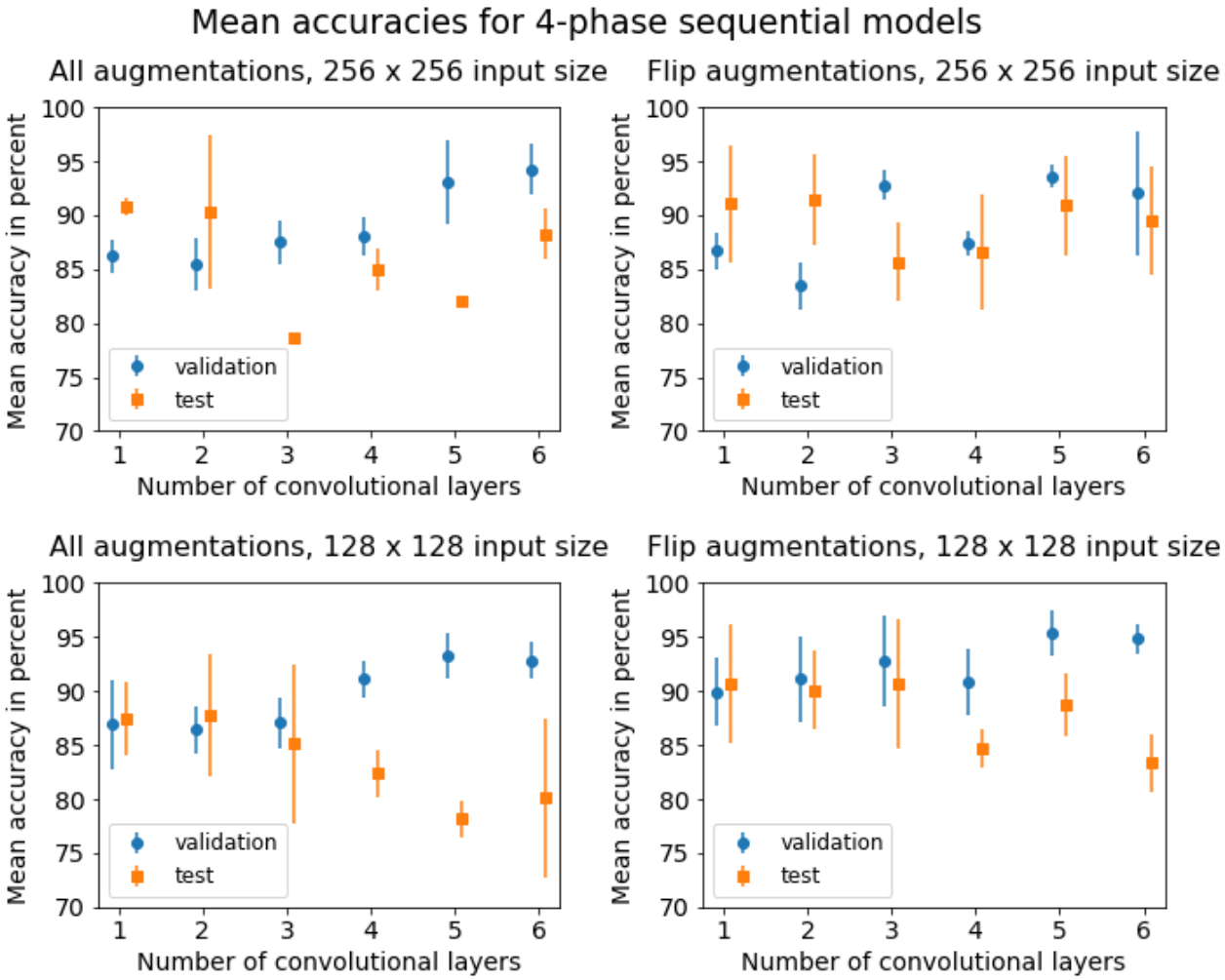

Figure 3: Plots of the mean accuracies against the number of convolutional layers for the 4-phase models over three training runs.

the images produced by the extra augmentations are dissimilar to actual texture images. When applying the rotation, scaling, and translation transformations included in all augmentations, certain edges and corners of the image are extended or distorted to fill the input space. These regions contain no liquid crystal texture features, which could be the cause of the lower performance. In addition, the models with all augmentations took approximately twice as long to train as with flip augmentations, due to the increased computing time in applying the extra transformations.

With the same augmentations and number of layers, models with an input size of 256×256 performed better on the test set than 128×128, 10 out of 12 times. Certain textural features may have been lost at the lower resolution, which could explain the lower accuracies observed. Also, there was no significant reduction in model training time with the 128×128 input size, despite smaller convolution operations and GPU memory requirements.

Overall, the models achieved good accuracies on the test set, with 10 out of 24 scoring above 90 percent. This demonstrates the viability of CNNs in the liquid crystal phase identification

task. However, the textures of the four phases these models classify are highly distinct and potentially easily identified by the human eye, meaning even the best models would perhaps not be useful at this stage. The most straightforward way to improve the results would be increasing the size of the dataset and adjusting model capacity appropriately.

The highest accuracy on the test set out of every trained model was 94.33 percent. This was one of the models with two convolutional layers, flip augmentations, and 256×256 input size. The lowest had 74.42 percent accuracy, with 6 convolutional layers, all augmentations, and 128×128 input size. Figure 4 displays the test set confusion matrices for these models. We

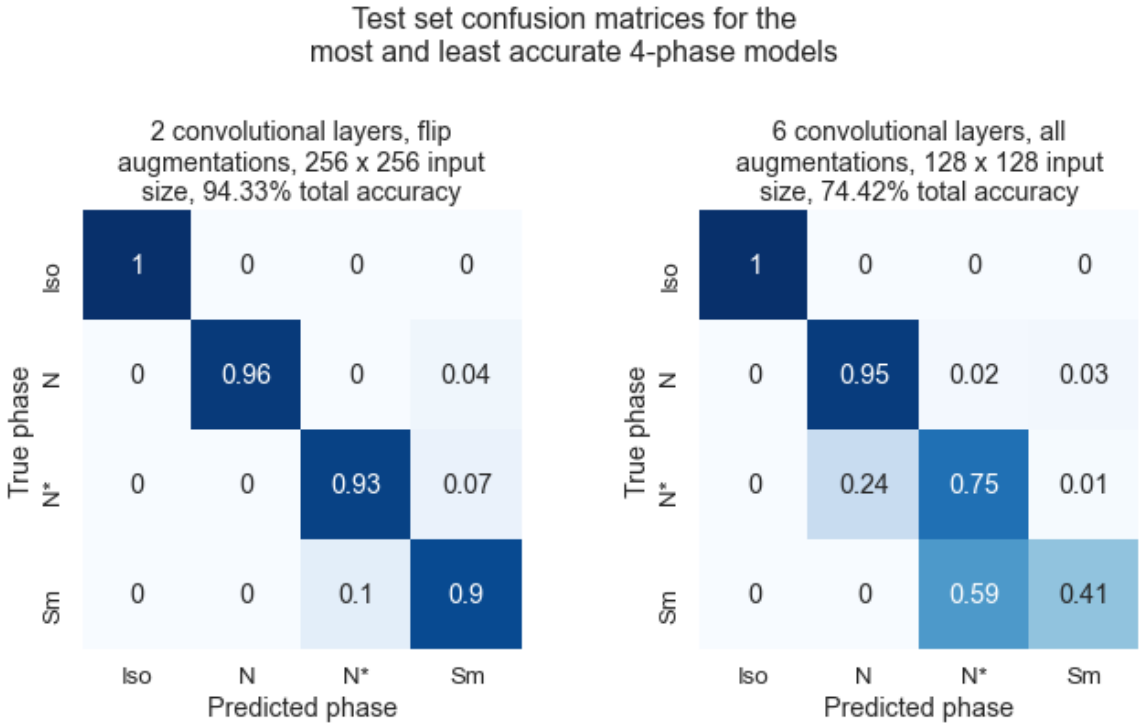

Figure 4: Test set confusion matrices for the models with the highest and lowest overall test accuracies. The value in each square represents the fraction of examples with the true phase label for which the model output was the predicted phase label. We, therefore, aim for values close to one along the downward diagonal, which corresponds to correct model outputs.

see that both models, especially the less accurate one, perform the worst on the cholesteric and smectic phases, suggesting that they have a higher degree of similarity between their textural features than the other classes. Both models correctly identify the nematic phase more than 95 percent of the time. They are both completely accurate in identifying the isotropic phase and do not miss-label any other phases as isotropic. This high isotropic success rate is potentially a result of the uniformity of the generated dark isotropic textures, which do not have highly complex features for the model to learn. Upon removal of the isotropic class from the test set, the most accurate model's accuracy is reduced to 92.57 percent, and the least accurate to 66.86. The isotropic class could, therefore, be viewed as a source of data leakage for these models. However, it does suggest that the models would easily be trained to identify real isotropic textures.

3.3 Models II, smectic A and C binary classifiers

3.3.1 Dataset construction II

The next set of models were an attempt at the binary classification task of smectic A and C phases, which can have similar textural features due to their structures only differing by a tilt in the director [5]. The distribution of data for the two classes is displayed in Table 2. The validation and test sets are more equally balanced for this set of models. All polarised

Table 2: Dataset distribution for smectic A and C classifier models.

	SmA	**SmC**	**Totals**	**% of total**
Training	719	1067	1786	70.17
Validation	174	183	357	14.03
Test	204	198	402	15.80
Totals	1097	1448	2545	

microscopy videos used in this set were provided by project supervisor Ingo Dierking, and all images were again prepared as in Section 3.2.2.

3.3.2 Model architectures and training configuration II

Based on the results of Section 3.3, flip augmentations and an input size of 256×256 were used for all smectic A and C models. The binary cross-entropy loss function is used, with one output unit for every model. The same six sequential model configurations as for the 4-phase classification task were tested, with reduced numbers of hidden units to prevent overfitting because the dataset is smaller. In this case, the final convolutional layer for each sequential model has 32 channels, with the number of channels halving with each convolutional layer going backwards through the model. Therefore, the six convolutional layer model has only one channel in its first layer. The number of hidden units in the dense layers is also reduced, with 32 in the first and 16 in the second for all sequential models.

A different type of CNN architecture was also tested for smectic A and C, inspired by Google's 22 layer GoogLeNet Inception model, the winner of the ILSVRC in 2014 [4]. An Inception model contains branching "Inception blocks", a specific arrangement of convolutional layers depicted in Figure 5 [29]. Three Inception models were used for smectic A and C, containing one, two, and three Inception blocks. Similar to GoogLeNet, each one starts with a convolutional layer with kernel size 7×7, 2 channels, and same padding, followed by a standard pooling layer. This is followed by two more convolutional layers, first with kernel size 1×1, 4 channels and valid padding, and second with kernel size 3×3, 8 channels and same padding, before another standard pooling layer. This is proceeded by the Inception blocks. The number of channels in each layer in a block is reduced by the 1×1 kernel layers and starts at 8 for the first, doubling with each successive block. The final inception block is followed by average pooling with pool size and strides of 5×5 and valid padding. This is followed by a convolutional layer with kernel size 3×3, a number of channels equal to half

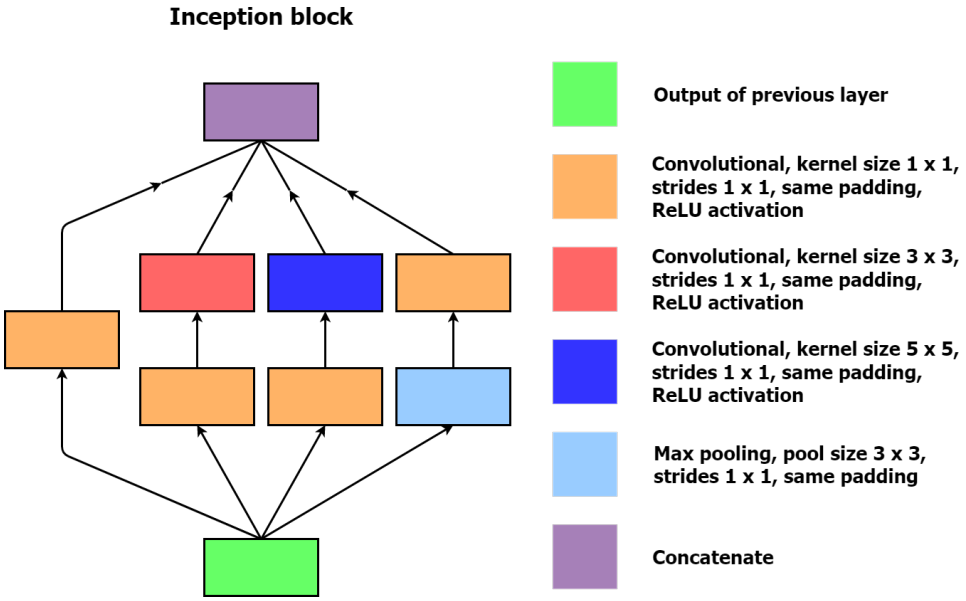

Figure 5: Architectural diagram of an Inception block. The input is shared between 4 parallel convolution operations, with their outputs being concatenated along the channel axis in the final layer. This results in the output having a number of channels equal to the sum of the number of channels of each concatenated layer. Different kernel sizes in each parallel layer can extract differently sized features from the input, propagating more information through the output of the block. The 1×1 kernel layers preceding the 3×3 and 5×5 ones can be used to reduce the number of channels and, therefore, the computational cost, if necessary. The pooling layer is there to provide some translational invariance [29].

that of the output of the final inception block, and same padding. Similarly to the sequential models, it is finished with global average pooling, followed by two dense layers, first with a number of hidden units equal to the channels in the previous convolutional layer and second with half that number, before the final single output unit. As an example, the architecture of the inception model with one block is drawn in Figure 9 in Appendix C.

3.3.3 Results II

Similar to the 4-phase models, all sequential and Inception smectic A and C models were trained three times and the mean validation and test accuracies calculated, with an uncertainty of half the range. These results are shown in Figure 6. We do not observe any noticeable trends in either validation or test accuracy, and most of the models have similar scores on both sets.

All nine models achieved a mean test accuracy above 90 percent, with four above 95. This is a strong performance, demonstrating the viability of CNNs in differentiating similar-looking liquid crystal textures. However, it is a simple binary classification problem, so larger datasets with more phases will be required to test the limits of these types of models. The Inception models have a greater minimum mean test accuracy than the sequential ones, with 94.79 percent compared to 92.71. However, the highest mean test accuracy of 97.05 percent belongs to the sequential model with four convolutional layers. More training will have to be carried

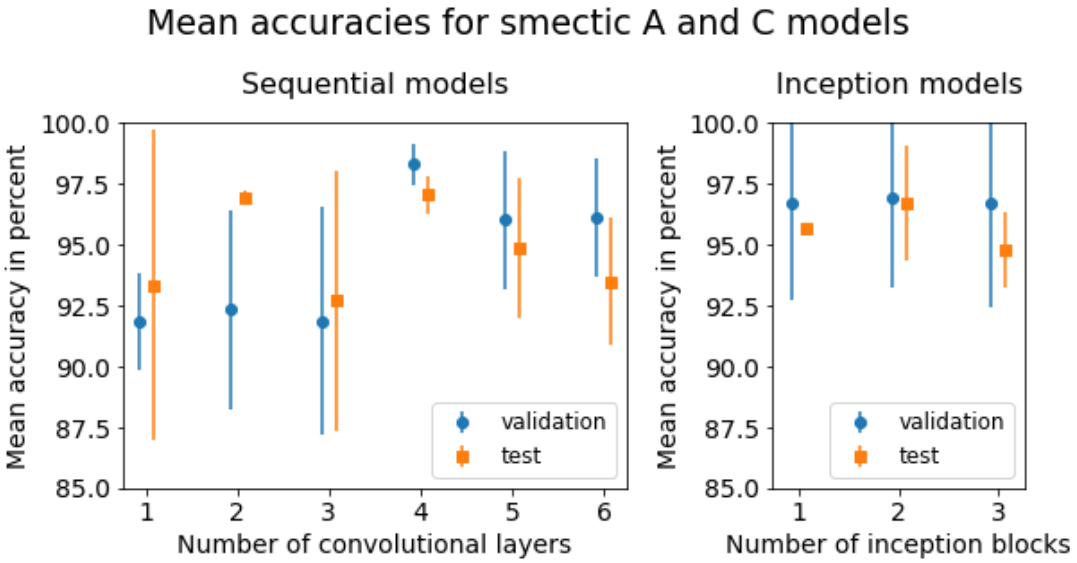

Figure 6: Plots of the mean accuracies over three training runs against the number of convolutional layers for sequential models, and the number of Inception blocks for Inception models, for the smectic A and C models.

out to draw confident conclusions about the architecture that performs better overall.

The error bars are very large on some of the data points, suggesting that certain models in this task are highly sensitive to the stochastic properties of training. The error could be reduced to provide a more confident estimation of each model's overall accuracy by increasing the dataset size and performing more training runs. A single training cycle takes a long time, approximately one hour, with the set up used to train these models, which is the limiting factor in producing more results and higher accuracies. Larger models with bigger datasets will also take longer to train.

The single saved model with the highest test accuracy at 99.22 percent was an Inception model with two blocks and the lowest was a sequential model with one convolutional layer, at 85.42 percent. The confusion matrices for these models are displayed in Figure 7. The Inception model misidentifies two percent, or four samples, of the smectic C test images. This is likely a fluctuation and may not represent the model's true generalisation error, due to the relatively small size of the dataset. The sequential model's inaccuracies mostly stem from misidentifying smectic A as smectic C, potentially due to there being more smectic C samples than A in the training set.

3.4 Models III, general smectic classifiers

3.4.1 Dataset construction III

The same models as the smectic A and C classifiers, aside from the number of output units, were tested in the identification of the fluid smectic, hexatic smectic, and soft crystal phases. The dataset distribution is presented in Table 3. Again, all videos used in this set were provided by project supervisor Ingo Dierking, and all images for this set were prepared as in

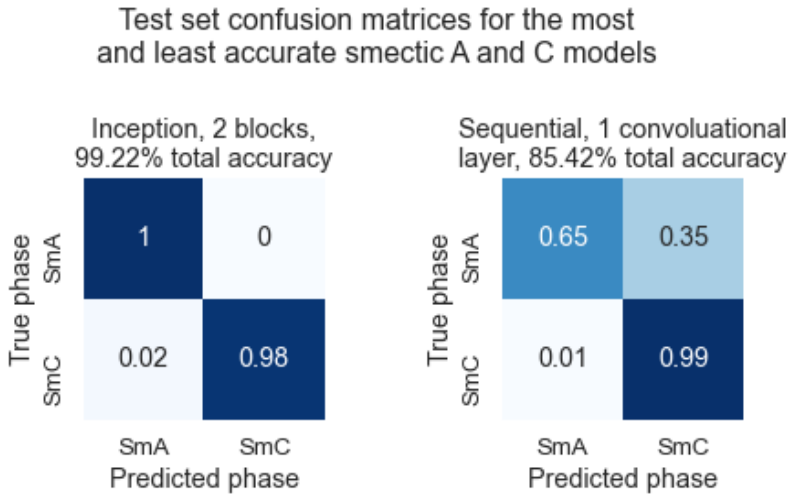

Figure 7: Test set confusion matrices for the smectic A and C models with the highest and lowest overall test accuracies.

Table 3: Dataset distribution for general smectic classifier models.

	FSm	**HSm**	**SC**	**Totals**	**% of total**
Training	1759	486	600	2845	70.67
Validation	372	90	144	606	15.05
Test	389	90	96	575	14.28
Totals	2520	666	840	4026	

Section 3.2.2. The current collection of polarised microscopy videos gives an image dataset that is greatly imbalanced in favour of the fluid smectic phases.

3.4.2 Model architectures and training configuration III

The nine models used for the general smectic phase classification have the same architectures and training configurations as those of section 3.4.2, aside from using the categorical cross-entropy loss instead of binary cross-entropy and having three final layer output units instead of one.

3.4.3 Results III

When training the sequential models on the dataset, despite some trial and error adjustments to the hidden layer sizes, every model predicted fluid smectic 100 percent of the time on the test set. This is most likely due to the large imbalance in the dataset. The sequential models are possibly unable to learn enough features from the small sets of hexatic smectic and soft crystal data, collapsing into predictions of fluid smectic every time, which results in an accuracy on the test set of 67.65 percent. This is simply equal to the percentage of fluid

smectic examples in the test set.

The Inception models were trained on this set one time each, with some more successful results. The test set accuracies were 72.24, 83.27, and 85.48 percent for one, two, and three blocks respectively. The models are still biased towards the fluid smectic class. However, they can identify the other phases, for example, the three-block model correctly classifies 62 percent of the hexatic smectic and 83 percent of the soft crystal phases. The Inception models' performance is potentially better than that of the sequential models due to the range of different kernel sizes, which can extract features of varying size [25]. More data, especially for the hexatic smectic and soft crystal phases, will be needed to conduct a more concrete investigation.

4 Conclusions

In this beginning half of the project, we have endeavoured to establish a well-formed liquid crystal texture dataset, and investigate the viability of machine learning algorithms in identifying the different liquid crystal phases from the dataset. The approach so far has considered deep convolutional neural networks, in three separate phase classification tasks.

From the 4-phase model investigations, we conclude that the flip augmentations are less computationally expensive and time-consuming, and produce more accurate models than the all augmentations setting. Also, when comparing the two input sizes of 256×256 and 128×128, the former results in higher accuracies with no increase in model training time. In general, the 4-phase models perform well on the test set, showing that CNNs have the potential to be an effective choice of machine learning algorithm type.

The extremely high performance of the smectic A and C binary classifier models further demonstrates the efficacy of CNNs in identifying liquid crystal phases, including ones with fewer distinguishable textural features. The Inception and sequential models both achieve good results. The large deviations in accuracy on repeated training runs of these models show that final model accuracy can vary greatly, depending on the parameter initialisation values and the randomly selected and augmented training data minibatches. Increased confidence in average accuracy requires further training runs, which is limited by long training times.

Despite having built a substantial overall collection of texture images, the general smectic phase model investigation highlights the requirement for balanced classes and a large training set for models to generalise well. A lack of data in the hexatic smectic and soft crystal phases results in poor performance, with the sequential models outputting a fluid smectic prediction for every input. The Inception models offer a marginal improvement in this case. However, the accuracies on the test set are still lower than would be required of a reliable model.

5 Going forward

For the second half of this project, there are numerous potential ways to expand upon the current work and take it in different directions. Other types of CNNs with branching architectures could be investigated, as well as larger models trained with more data that can classify a greater number of liquid crystal phases. Entirely different deep learning algorithms could also

be investigated, transformer networks in particular. Coloured image inputs could be tested, to determine whether or not this would lead to better generalisation or simply create data leakage. If the data and labels become available, regression models for physical properties of liquid crystals such as cholesteric pitch length could be investigated.

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. 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] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, vol. 37, p. 448–456, JMLR.org, 2015.

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

[24] 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.

[25] 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.

[26] 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.

[27] V. Williams, "Vance Williams YouTube channel." `https://www.youtube.com/channel/UCB8qnCxJbdsuXpQ5RbLNy3Q`. Accessed: 12-10-2020.

[28] V. Williams, "Vance Williams Instagram page." `https://www.instagram.com/vance.williams/`. Accessed: 12-10-2020.

[29] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

Appendices

A Training algorithms

A.1 Backpropagation

For a fully connected feedforward neural network, the backpropagation algorithm is used to compute the gradient of the cost function with respect to all trainable parameters, $\boldsymbol{g} = \boldsymbol{\nabla}_{\boldsymbol{\theta}} J$. $\boldsymbol{g}$ is composed of the gradients of J with respect to the weights and bias parameters for each layer l, $\boldsymbol{\nabla}_{\boldsymbol{w}_l} J$ and $\boldsymbol{\nabla}_{\boldsymbol{b}_l} J$. Let us define $\boldsymbol{\mathcal{G}}$, a vector with a variable size that is updated with the gradient of each layer as we move backwards through the network. At the start of backpropagation, we assign the gradient on the final layer outputs to $\boldsymbol{\mathcal{G}}$,

$$\boldsymbol{\mathcal{G}} \leftarrow \boldsymbol{\nabla}_{\boldsymbol{O}_D} J = \boldsymbol{\nabla}_{\hat{\boldsymbol{y}}} J, \tag{17}$$

obtained from the derivative of the loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ with respect to $\hat{\boldsymbol{y}}$. We then perform D steps with layer number l decreasing by one each time. For each step, we first update $\boldsymbol{\mathcal{G}}$ by changing it from a gradient on the layer's output to a gradient on the non-activated output, by the chain rule,

$$\boldsymbol{\mathcal{G}} \leftarrow \boldsymbol{\nabla}_{\boldsymbol{h}_l} J = \boldsymbol{\mathcal{G}} \odot A'(\boldsymbol{h}_l). \tag{18}$$

The parameter gradients for the layer are given at this point by

$$\boldsymbol{\nabla}_{\boldsymbol{w}_l} J = \boldsymbol{\mathcal{G}} \boldsymbol{O}_{(l-1)}^{\top}, \tag{19}$$

$$\boldsymbol{\nabla}_{\boldsymbol{b}_l} J = \boldsymbol{\mathcal{G}}. \tag{20}$$

To finish the step, the gradient is propagated onto the output of the next lower hidden layer,

$$\boldsymbol{\mathcal{G}} \leftarrow \boldsymbol{\nabla}_{\boldsymbol{O}_{(l-1)}} J = \boldsymbol{w}_l^{\top} \boldsymbol{\mathcal{G}}. \tag{21}$$

Upon reaching $l = 1$, all parameter gradients have been obtained and the algorithm is complete [12, 2].

A.2 Adam optimiser

The Adam stochastic optimiser algorithm proceeds as follows. Before the first iteration, the biased gradient first and second moment estimate variables, $\boldsymbol{s}$ and $\boldsymbol{r}$, and the timestep, t, are initialised to zero. At the start of each iteration, a minibatch of data is sampled and $\hat{\boldsymbol{g}}$ is calculated as in Equation 13. t is updated as

$$t \leftarrow t + 1, \tag{22}$$

$\boldsymbol{s}$ is updated as

$$\boldsymbol{s} \leftarrow \beta_1 \boldsymbol{s} + (1 - \beta_1) \hat{\boldsymbol{g}}, \tag{23}$$

and $\boldsymbol{r}$ is updated as

$$\boldsymbol{r} \leftarrow \beta_2 \boldsymbol{r} + (1 - \beta_2) \hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}}, \tag{24}$$

where β_1 and β_2 are the first and second moment exponential decay hyperparameters. The bias-corrected first and second moments, $\hat{\boldsymbol{s}}$ and $\hat{\boldsymbol{r}}$, are then updated as

$$\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\beta_1^t}, \tag{25}$$

and

$$\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\beta_2^t}. \tag{26}$$

In the final step of the iteration, the parameters of the model are updated as

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \epsilon}, \tag{27}$$

where α is the step size hyperparameter and ϵ is a small hyperparameter for numerical stability. Iterations are carried out until model training is complete. The recommended defaults for the hyperparameters are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ [19].

B Example architecture diagrams

Figure 8 provides a representation of the network architecture of the 4-phase sequential model with three convolutional layers and 256×256 input size. Figure 9 provides a representation of the network architecture of the smectic A and C Inception model with one block.

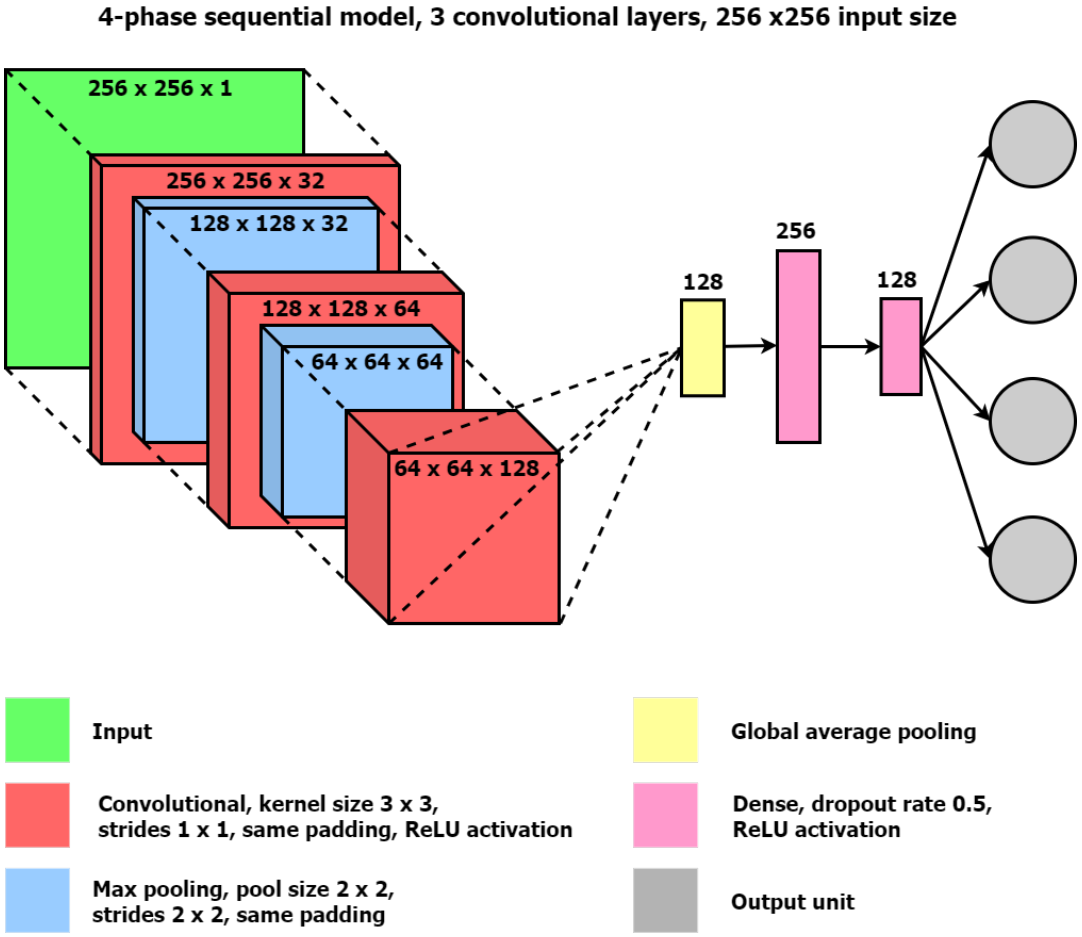

Figure 8: Architectural diagram of the sequential CNN model with input size 256×256 and three convolutional layers. The output dimensions are displayed as width $\times$ height $\times$ channels for each convolutional and max pooling layer. Not to scale.

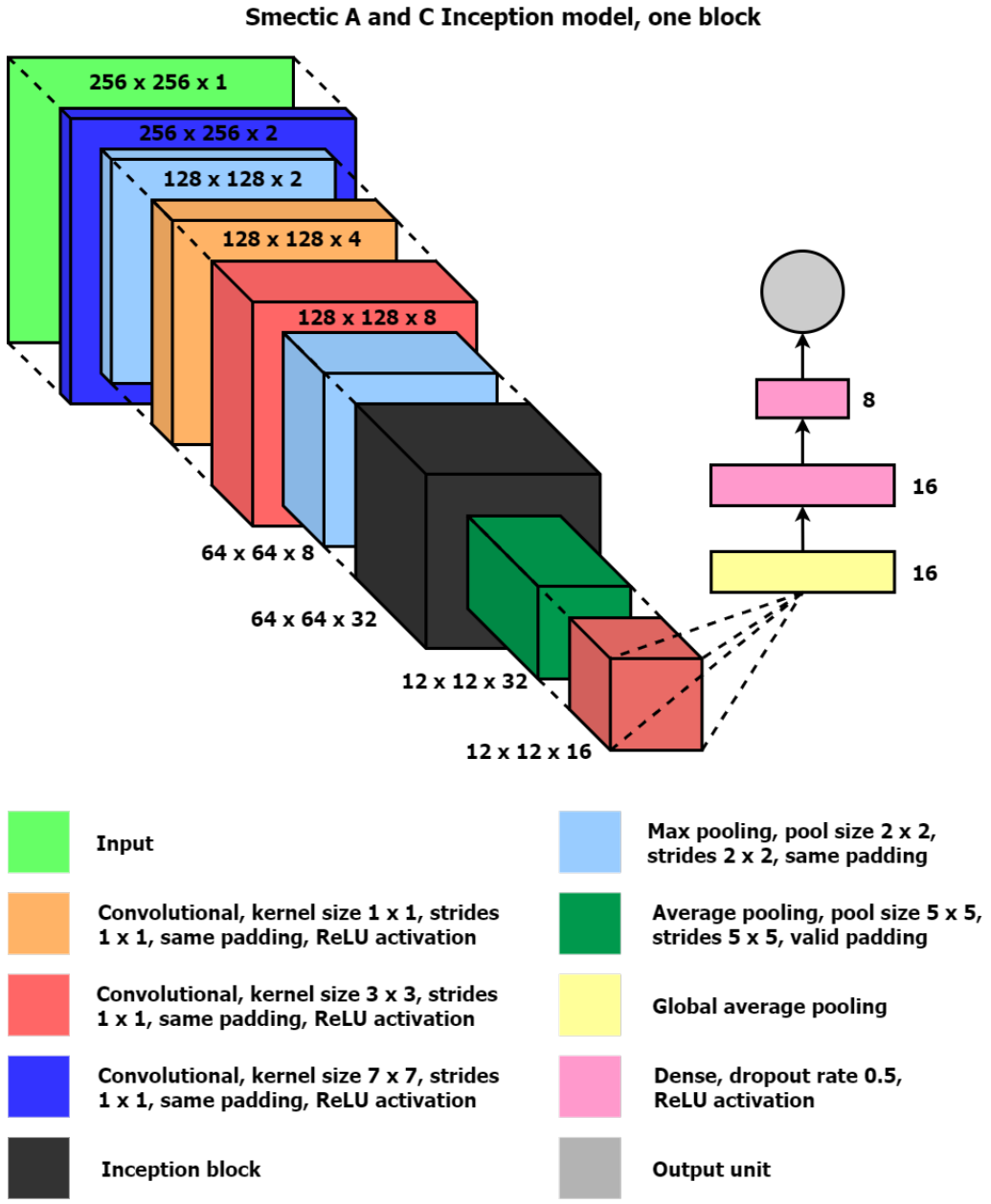

Figure 9: Architectural diagram of the Inception CNN model with one block. The output dimensions are displayed as width × height × channels for each convolutional, max pooling and Inception block layer. Not to scale.