UNIVERSITY OF BIRMINGHAM

Validating the use of Neural Networks in the Categorisation of STEM images of Nanoparticles

William Sullivan
ID:1644901

Issue: 1
Date: April 3, 2020
Word Count: 6757

ABSTRACT

Scanning Transmission Electron Microscopes are powerful tools, providing 2D images of 3D particles at the atomic scale. However, analysing these images can be time consuming and unreliable as the methods required to infer the structure of the particle may not be obvious, especially when considering multiple images in stereo. This project aims to show that a neural network can be developed to use images in stereo and reliably categorise particles which would be too difficult identify by a human.

School of Physics and Astronomy
University of Birmingham
Birmingham, B15 2TT

Contents

1 Introduction

Scanning Transmission Electron Microscopy is a method of generating images of an nanoparticle. A highly focused beam of electrons is passed over the particle being imaged and based on the scattering of the electrons, an image can be formed, with the intensity of the image indicating the thickness of the particle at that point of the image. This method of collapsing a 3D object down to a 2D image can be very useful for identifying particles, however, it is difficult to automate the analysis of the data and can require a person, comparing the images gathered with example images of the classifications. This may not only take a large amount of time, but the results may then depend on the person evaluating the particles. This can then become more of a problem

when the categorisation is not simple, requiring analysis of multiple images and detailed comparisons between the two.

Machine Learning could provide a method of quantitatively evaluating the particles and categorising a population as $x\%$ in one category and $y\%$ in another. By inputting multiple images into a trained neural net, a categorisation for the particle can be obtained along with the certainty of the output. Across large populations of particles, any inaccurately categorised particles should be compensated for by the remaining uncertainty on correctly identified particles. For complicated categorisations, this likely won't provide a perfectly accurate output, however, it would likely still be more accurate than a human analysing the data and the analysis can be performed much quicker and with the same bias across multiple populations.

While one image of a particle can be used to infer some information, the depth of any feature is lost. By taking multiple images, it may be possible to infer more about an feature, however, for particles with many features, there may be many different ways of organising them. This project aimed to show that a neural network could be trained to take multiple images and analyse them together in order to infer the structure of specific features which would not be possible without considerable, human effort.

The Keras functional API[1] running on TensorFlow[2] was used to build and train the neural networks in python. This package contains a number of prebuilt layers along with the various loss and activation functions and the algorithms to train them. The functional API included as part of this allows for more complicated neural network structures. It was especially useful in allowing for one layer to be used multiple times for different inputs, allowing for data to be processed in parallel.

2 Neural Networks

Neural Networks consist of a series of nodes organised into layers which take a set of input data and then feed it though to consecutive layer, manipulating it based on the various parameters of the network to produce an output.

Once a network with appropriate architecture has been designed, it is initiated with random variables as the parameters. It must then be optimised through training using an optimisation algorithm. To do this, existing data must be obtained. This could be through experimental analysis or it could be through simulations. Once the data has been gathered, it must be labelled with the expected output of network. This is then often separated into two different groups, the training data which is used to train the network, and the validation data which is used to verify the network is leaning to recognise patterns and not recognise the sample data provided.

The optimisation algorithm is provided with the untrained network, the data and a set of hyperparameters. These hyperparameters define the learning and are defined by the user at the start, unlike the parameters of the network which the optimisation algorithm changes during running.

During training, small batches of data are selected in random groups from the training data. These are then used to change and improve the network based on the optimisation algorithm. After all the training data has been iterated through, one epoch has passed and the validation data will normally checked to verify progress. This process is repeated over many epochs, passing through the entire training data set multiple times.

It is important that the output of the verification data is monitored during training. If training is performed for too long, overfitting can occur. This means that the network has stopped identifying general patterns in the training data and is starting to identify specifics. This makes it less generally applicable and reduces its accuracy in other data sets, such as the validation data. There are a few methods for reducing overfitting discussed later.

2.1 Layers

Neural networks are designed with a variety of different layers, each have their own uses, from performing the majority of the calculations to increasing training efficiency. These layers are all included in the Keras package and are defined by a number of hyperparameters allocating their size and various other functions.

2.1.1 Dense Layers

Dense layers are the simplest layers in a network. Each node in this layer has its own bias parameter and is linked to each node in the layer before with a weight parameter for each node. A node will calculate its output by multiplying the output of the previous layer nodes by their corresponding weights, then summing them together with the node bias. This total is then passed through an activator function such as a sigmoid function or Rectified Linear Unit to produce an output which can be used by the next layer of nodes.

This is the most generic layer, by connecting every node to all previous nodes in a network, the network can learn its own structures. However, it requires training a disproportionately large number of parameters, increasing training time and difficulty.

2.1.2 Convolutional Layer

Convolutional layers act to recognise particular patterns or features in the previous layer. Instead of directly connecting to previous layers, a filter is passed over the previous nodes, and each node activates depending on whether specific features are detected. These layers are often used in image recognition. When considering a simple 64×64 black and white image, there are 4096 pixels of varying values organised into a square. When creating a convolutional layer, a filter size is determined, say 5×5. This filter would consist of 25 weight parameters which are multiplied by the value of the pixel underneath them and then summed together with a bias parameter for the filter before being passed through an activation function, essentially creating a dense layer connected to a specific 5×5 portion of the original image. This filter is then applied again, but shifted by one or more pixels until the entire area of the image has been covered, producing an activation map for the feature[3].

Each convolutional layer often contains many trained filters to identify many different features, each with their own specific weights and biases. There may also be additional filters if there are multiple channels. If instead of a black and white image, a RGB image is used, there are then three distinct values for each pixel in the image. One for the red value, one for the blue and one for the green. In this situation, an extra filter is learnt for each channel and the products of all three are summed with the bias before being passed through the activation function in order to produce the activation map.

Convolutional layer can also contain a some padding. By sorting adjacent nodes into groups of n for an n wide filter, there will be fewer nodes in the layer's output as the nodes on the edge get cut off. Padding adds extra zeroes around the edge of the previous layer so that the filter can be applied right up to the edges of the layer. Thi sproject used padding in all convolutional layers.

2.1.3 Pooling Layer

Convolutional layers are often followed by a pooling layer. These layers allow for some translational leeway[4]. A pooling layer isn't trained like other layers, instead, they have a defined pool size which is used to produce a filter. Similar to the filter in convolutional layers, this filter is used to separate the previous nodes into regions. Depending on the type of pooling, these regions are then reduced to one output value. Maximum pooling and average pooling are the most common methods of pooling. Maximum pooling results in the largest of the nodes in the region being used for the region's output. Average pooling simply averages all the nodes together to produce an output. Both result in the next layer having fewer nodes and allow features to shift by a few pixels and still cause the same activation after pooling. This helps to reduce overfitting and also reduces the memory required to train the neural net.

2.1.4 Backend Layers

There are a number of layers included in networks that often get overlooked as they have no direct impact on the architecture, however, they are still important for effective training and running of the network.

Dropout layers are purely to compensate for overfitting and are only active when the network is training. When running, they take the previous layer and copy it, randomly setting a proportion of the outputs to 0. This helps to reduce the dependency of the layer on a specific node and effectively distorts any data being passed through so that it appears a new, but similar set of data[5].

Batch Normalisation Layers are filters applied to entire layer. This layer normalises the outputs of the previous layer by moving the mean of the outputs to 0 and changing the standard deviation to 1, while maintaining

the proportional value of each node. This helps to increase the training rate by fixing the output range of a layer. This results in more stable inputs for the next layer to process[6].

Finally, there are a number of layers used to change the structure of the previous layer. These are the reshape, flatten and concatenate layers. The reshape layer will change the shape of the input layer to a specific shape. This is used in this project to add an extra dimension for the convolutional layer to consider the channel. The flatten layer will change all the nodes in the previous layer to be in one line in one dimension. This can be useful for removing the channel dimension in order to put all the nodes through a dense layer. Finally the concatenate layer allows for layers to be run in parallel and then be joined together along an axis before continuing with further layers.

2.2 Activation Functions

These are the functions that are applied to the summed values for a node before being passed as an output for a node. Normally, there are two function used, depending on the situation. For internal layers, Rectified Linear Unit or ReLU is used. However, the final output layer will normally use a Softmax activation.

2.2.1 Rectified Linear Unit

This is a simple activation function which has been shown to be effective in training neural nets[7]. After the previous outputs have been multiplied by the corresponding weights and summed with the bias, the ReLU function returns the value with a minimum value of zero.

2.2.2 Softmax

This is the activation function used for the output layers of categorisation tasks as it converts a list of values into a list which can be interpreted as a probability, ie, they sum to 1 and are all values between 0 and 1. To convert a list of values $x_i = (x_1, x_2, ...x_n)$ to their values after activation, $z_i = (z_1, z_2, ...z_n)$, the equation

$$z_i = \frac{e^{\beta x_i}}{\sum_{j=1}^{n} e^{\beta x_j}} \tag{1}$$

is used. Where β is a hyperparameter which can be used to favour higher or lower values, but is left as 1 for this project.

2.3 Loss Functions

The loss function is the value that the learning algorithm is attempting to minimise. It is a quantitative way of defining how accurate the prediction of the network is for a given set of data with known, correct outputs. While there are a number of different options available, categorical crossentropy is the appropriate loss function to use in almost all categorisation tasks. This loss function is defined by

$$Loss = -\sum_{i=1}^{n} t_i \log(p_i) \tag{2}$$

where t_i is the true value of the ith category and p_i is the predicted value of the ith category for n different categories.

2.4 Optimisers

The optimisation algorithm is the algorithm used to change the values of the parameters in the network. While there is no definitive conclusion on the best optimiser to use as it depends on the task, this project used the AdaDelta optimisers which is an extension on the AdaGrad optimiser, which itself expands on the Stochastic gradient descent optimiser. Through some quick experimentation at the begining of the project, it was determined that this achieved the highest accuracies with the fewest epochs.

2.4.1 Stochastic Gradient Descent

This is a simple optimisation algorithm that most other algorithms expand on, adding complexity to increase training reliability and speed. While it isn't used in this project, it is helpful in understanding the algorithms used[8].

First, a small sample of m inputs are selected randomly from the training data. A value is then estimated for each parameter in the network for the given sample. This value is the gradient of the loss with respect to the parameter, when the model has the current parameter values. Expressed as an equation, the gradient, g_n, for the parameter θ_n is

$$g_n = \frac{1}{m} \nabla_{\theta_n} \sum_{i=i}^{m} L\left(f(x_i \mid \boldsymbol{\theta}), y_i\right) \tag{3}$$

for m samples inputs, x_i, with associated labels, y_i, after the inputs have been passed through the neural net, represented by the function f, with previous, given parameter values, $\boldsymbol{\theta}$. Where the function L is the loss function being used. This will only be an estimator of the gradient and will depend on the sample taken. Taking larger sample sizes will provide a more accurate estimator, however, the return is not linear, so it is more efficient to run many smaller samples.

After estimating the gradient for each parameter, the parameter values are updated by multiplying the gradient by the learning rate, a hyperparameter set at the start of training, then subtracting this value from the current value for the parameter. As an equation,

$$\theta_n \to \theta_n - \epsilon g_n \tag{4}$$

Where ϵ is the learning rate hyperparameter. Normally, the learning rate is also decreased after each epoch. This allows for larger, more approximate learning for early epochs and then more precise learning at later epochs.

2.4.2 AdaGrad

This expands on the SGD optimiser by varying the learning rate based upon the past gradients. This way, parameters which have been updated a lot, will have smaller learning rates, increasing stability. While parameters with smaller gradients will have a larger learning rate, accelerating their learning[8]. It starts in the same way as SGD, by estimating the gradients, g_n, in the same way. It then saves the cumulative sum of the squares of the gradients for each parameter. Before updating the parameter, it then modifies the learning rate for the parameter so that,

$$\epsilon_n = \frac{\epsilon}{\delta + \sqrt{G_n^2}} \tag{5}$$

where G_n^2 is the sum of the squares of g_n and δ is a small constant of order 10^{-7} to avoid dividing by zero and increase stability. Each parameter is then updated in the same manner, but with their individual learning rate.

$$\theta_n \to \theta_n - \epsilon_n g_n \tag{6}$$

While increasing the initial learning rate of the network, this optimiser can reach a point where learning stops prematurely. By summing the square gradients for the entire duration of the run, the value of G_n^2 will eventually reach large enough numbers where the learning rate for parameters will be effectively 0, resulting in no further learning.

2.4.3 AdaDelta

This builds further on AdaGrad by attempting to continue the training process and avoiding parameter learning rates approaching zero. To do this, it also multiplies the current total, G_n^2 by a decay constant, another hyperparameter, when adding the current square of the gradient[9]. With this method, the value G_n^2 effectively becomes the sum of the squares of the gradients for a small window of previous gradients, allowing the learning rates to increase, given time.

3 Simulation

During the course of this project, large amounts of data was needed for the training of neural nets. As this data was mostly mostly for abstract particles, it was not feasible to gather this data experimentally, so the data was randomly generated from a number of defined parameters. When generating this data, it was important to ensure that there was no patterns in the generation that weren't directly linked to the structure of the particle. For example, when removing mass from the interior of the particle, it was placed on the outside so it wasn't as simple as calculating the total mass of the projection. It was also important to make the data appear real by adding several noise parameters.

This project focused on training a neural network to recognise 1D projections of 2D slices through a 3D particle. As the majority of the particle was unnecessary, it was easiest to to generate the 2D shape of a slice and then calculate the thickness of the shape at intervals to create the projection. Initially, the shapes were all rotationally symmetrical, at least as far as the random generation mattered, so they could be rotated about an axis perpendicular to the plane of the shape without having to worry about the rest of the shape, ignoring the third dimension completely. Later, the problem became more abstract and so the assumption held.

This meant that the particle could be defined by the outer perimeter of the slice. This in turn could be defined by a set of angle and radius coordinates of the corners. The angle coordinates are generated at regular angles around the perimeter depending on the parameter defining the number of angles. The radius coordinates are then generated by first determining the average radius of the shape. This is done using 2 defined parameters, one for the average value and one for the variance of the normal distribution used. Another normal distribution is then used about this value with another variance parameter set in the parameters.

This produces a rough shape which can act as a template to be filled by the interesting features which need to be classified. After that, the centre is shifted by a normal distribution, moving the entire particle so rotation can occur that isn't about the centre of the particle. A full list of parameters used to generate the perimeter can be found in table 1.

Parameter Name	Data Type	Purpose
r	float	Average radius of shapes
sig_r	float	Variance on average radius
sig	float	Variance on radius coordinates about the shape's average radius
angle_count	int	Number of radius points in a shape
centre	iterable (floats)	Average centre coordinates of a shape
sig_c	iterable (floats)	Variance on the coordinates of centre

Table 1: Parameters used in the generation of the outline perimeter of all shapes

3.1 Generating Projections

After a shape has been generated, it must then be transformed into one or more 1D projections of the thickness. It is easier to do this in Cartesian coordinates, so the shape's coordinates are first converted. A detection plane is then defined perpendicular to the plane of the shape. The line of intersect of the 2 planes is then divided into a number of data points between an upper and lower limit, defined by 3 parameters.

At each of these data points, a ray is traced perpendicularly from the detection plane and the distance to the intersect between point a and point b is calculated for every adjacent pair of points defined using the equation,

$$l = y_a + \frac{x_0 - x_a}{x_b - x_a}(y_b - y_a) \tag{7}$$

where l is the distance to the intersect, y_a and y_b are the y coordinates of point a and point b, x_a and x_b are the x coordinates of point a and point b and x_0 is the x coordinate of the data point. The derivation can be found in Appendix A. Once all the intersects are found, they are sorted based on the distance from the plane. This distance between alternating pairs is then calculated and summed to calculate the total thickness of the

shape at that point, i.e., the distance between the furthest and second furthest points of intersect are added to the distance between the third and fourth points of intersect, etc.

This process is then repeated for additional angles by converting the shape's coordinates from polar to Cartesian with an additional offset added to the angle coordinates, effectively rotating the shape. A parameter can also be used here for the variance of the angle, using a normal distribution, to make it more challenging and simulate uncertainty in the amount the particle is rotated by. Another variable also allows the detection plane to be shifted by an amount determined by a normal distribution. This was added to simulate small inaccuracies in the preprocessing which would may picking these particles out of a larger image.

After obtaining accurate projections, some noise is added to make the data appear more like experimental data. This is done based on 3 parameters. First a constant background value is add to every data value, acting as the back plate for the STEM. Next a Gaussian blur is applied to projection, where the standard deviation is defined by a parameter. Then each data point has additional noise added to it. This is done by putting each data point through a normal distribution with the true value as the average and a proportion of the true value as the variance.

Finally, after all the projections are generated, they are normalised together by subtracting the smallest data point value and dividing by the maximum. This puts the entire data set to intensities between 1 and 0, where values correspond across all projections.A full list of the parameters required for projections is included in table 2.

Parameter Name	Data Type	Purpose
data_points	int	Number of data points in a projection
lower	float	Lower limit on data points
upper	float	Upper limit on data points
limit_var	float	Variance on the centre of the detection plane
angles	iterable (floats)	Approximate angle to evaluate projection at
sig_as	float	Variance on each angle projection is taken
about	float	The point that rotation occurs about
background	float	Additional background value
gauss	float	The standard deviation for the Gaussian blur filter
noise	float	Ratio of intensity to make the variance on projection data (eg, 0.1, results in each data point having a variance of 0.1t)

Table 2: Parameters used in the generation of a projection from a defined shape

3.2 Particles with Holes

The first feature this project examined was whether or not a particle contained empty space. When generating shapes, this translated to adding a hole in them. To do this, another, smaller shape was generated and subtracted from the projection of the larger shape. However, there were a few additional details considered in order to make the small hole appear similar to a lump on the surface of the large shape so that the network would have to use multiple inputs.

The first detail considered was the slope and roughness of the smaller shape. The edges of the shape should be approximately the same as the edges of the larger shape. So that the average distance between points was approximately preserved, the angle between two points was scaled proportional to the change in radius,

$$\phi \approx \frac{R}{r}\theta \tag{8}$$

Where ϕ is the angle between coordinates for the small shape, θ is the angle for the large shape, and R and r are the radiuses of the large and small shape respectively. This is only approximate, as some rounding was required in order to keep ϕ constant for all coordinates. The variance in the surface of the large shape was then used for the small shape to keep the "roughness" the same.

The other consideration was the mass. If all the particles with a hole had lower mass, then it would simply be a case of summing the intensities of the projection and checking if it was less than some value. Therefore, when adding a hole, the same mass was placed on the outside of the larger shape. While this did then make the shapes larger, it was decided that this change would be smaller and more likely to be lost in the noise, therefore making it harder to identify. With some simple geometry, the change in radius required is given by,

$$dR = \sqrt{r^{+}R^2} - R \tag{9}$$

where dR is the required change in the radius of the large shape, and r and R are the radiuses of the small and large shapes.

Generating the hole and projections was then identical to the large shape. An additional parameter was required, defining the average radius of the hole and the variance on that average. Another parameter was also set to define the maximum distance the hole could be placed from the centre of the large shape to avoid it spilling out of it. When placing the hole in the shape, a random angle and a random radius were picked uniformly to determine the centre of the hole. A full list of parameters required for adding holes is shown in table 3.

Parameter Name	Data Type	Purpose
data_count_h	int	Total count of the data set which have holes
data_count_s	int	Total count of the data set which are solid
h_range	float	The maximum distance of hole from centre
h_r	float	Average radius of hole
sig_hr	float	Variance on average radius of hole

Table 3: Parameters used in the generation of a data set including shapes with holes and shapes without holes

3.3 Particles with spirals

It was later decided to examine spirals inside the shape and categorise the chirality of the spin. This was an abstract categorisation so it was reasonable to say that the rotation happened about an axis perpendicular to the plane of the slice, keeping the simulation 2D. When generating this feature, a spiral shape was generated and then added to the projection of the larger shape. This was meant too represent a perfect spiral of atoms with a higher atomic number inside the larger particle. While this meant that the larger particle was mostly just acting as noise, it was later useful to generate a particle with both holes and spirals. Therefore, the spirals are always generated inside a larger particle.

The first important factor was defining the chirality of the particle. It was decided to early to consider the direction of the spin moving out from the centre, i.e., if the spiral rotated clockwise as you moved along it, away from the centre, it was a clockwise spiral. The edges of the spiral were defined in the same way as the larger shape, by defining the corners in polar coordinates. The radial coordinates of the inside of the spiral were defined by,

$$r_i = s\theta \tag{10}$$

where r_i and θ are the polar coordinates of the inner side of the spiral and s is a defined "straightness" parameter. A maximum radius is then defined for each spiral, which is varied using a normal distribution with the average value and the variance defined by a parameter. Dividing this by the straightness gives the maximum angle, which can then be separated into intervals to generate the angle coordinates. Finally the outer radius of the spiral is defined by,

$$r_o = \min((t+s)\theta, s\theta + t_{max}) \tag{11}$$

where r_o and θ are the polar coordinates of the outer side of the spiral, t is an additional value to add to the straightness to cause the outer edge to move out faster and t_{max} is the maximum thickness allowed to stop the spiral expanding to the point that it begins to overlap with itself. This allows the spiral to increase gradually to the desired thickness.

While it may have been interesting to examine the effects of varying the radiuses of the spiral coordinates to create a rougher shape or examine how a varying straightness effected learning, this project didn't get a chance to, so the spirals are all steady and smooth.

Finally, there was an additional parameter to determine the positioning of the spiral's centre in the larger shape. This was used in the same way as the parameter for the centre of the hole. A full list of parameters required for adding spirals to a data set is shown in table 4.

Parameter Name	Data Type	Purpose
data_count_c	int	Total count of the data set which have clockwise spirals
data_count_a	int	Total count of the data set which have anticlockwise spirals
data_count_n	int	Total count of the data set which have no spirals
s_range	float	The maximum distance of spiral from centre
s_r	float	Average radius of spiral
sig_sr	float	Variance on average radius of spiral
s_s	float	Average straightness of spiral
sig_ss	float	Variance on straightness of spiral
s_t	float	Average value for additional straightness of spiral
sig_st	float	Variance on value for additional straightness of spiral
s_t_max	float	Average maximum thickness of spiral
sig_st_max	float	Variance on maximum thickness of spiral

Table 4: Parameters used in the generation of a data set including shapes with clockwise, anticlockwise and optionally no spirals

4 Experiments and Results

There are a large number of parameters required for generating the data sets that the models were trained on. However, most of them remained unchanged for the majority of the tasks. Therefore table 5 shows the values used for generating all of the data for the tasks described below, unless otherwise stated. The noise values were determined by generating projections with various amounts of noise and comparing to real images. It was decided that the values in table 5 generated the most realistic looking projections.

Parameter Name	Value
r	1.
sig_r	0.1
sig	0.05
angle_count	90
centre	(0.,0.)
sig_c	(0.3,0.3)
data_points	32
lower	-1.5
upper	1.5
limit_var	0.05
sig_as	0.
about	(0.,0.)
background	20.0
gauss	0.5
noise	0.001

Table 5: Default parameters used in the generation of all data sets in this project, unless otherwise stated

In all of the following experiments, training was allowed to continue for up to 200 epochs. However, very few actually reached this point as training would be stopped early if the loss of the network, when applied to the validation data, didn't decrease over a number of epochs determined the patience value. If the specified

number of epochs passed without improving the fit to the validation data, the program determined that either it was overfitting to the training data, or had progressed as far as possible. In either case, training was stopped and the model restored in case significant overfitting had occurred.

4.1 Single Input Hole Detection

The first task attempted was the categorisation of whether a particle contained a hole or not using only one projection. While a relatively simple task, it was important to establish early on in the project, what was actually possible. The neural net displayed in figure 1 was designed to learn this task. Initially, the radius of the holes was approximately twice the variation in the surface. This made it difficult to say for by eye whether a shape contained a hole, keeping it a challenge for the network to solve, while not making it too difficult. Figure 2 shows two, typical shapes from the data set, along with their projections.

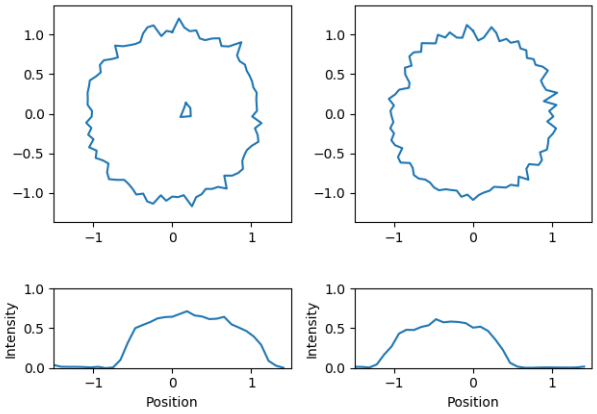


Figure 2: Sample shapes used in the categorisation of particles with and without holes
sig = 0.05 and h_r = 0.1

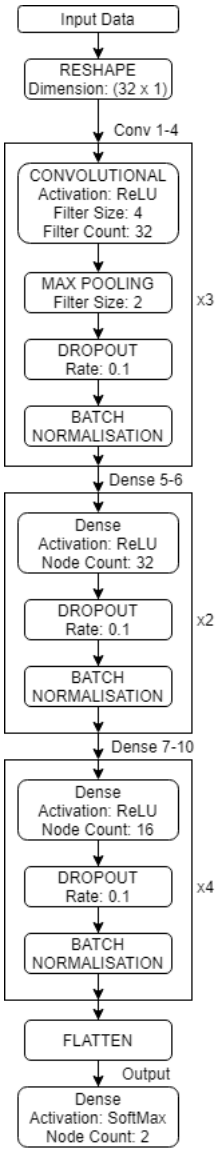


Figure 1: The Neural Network structure used to train for shapes with holes and one input.
Trainable parameters: 12,450

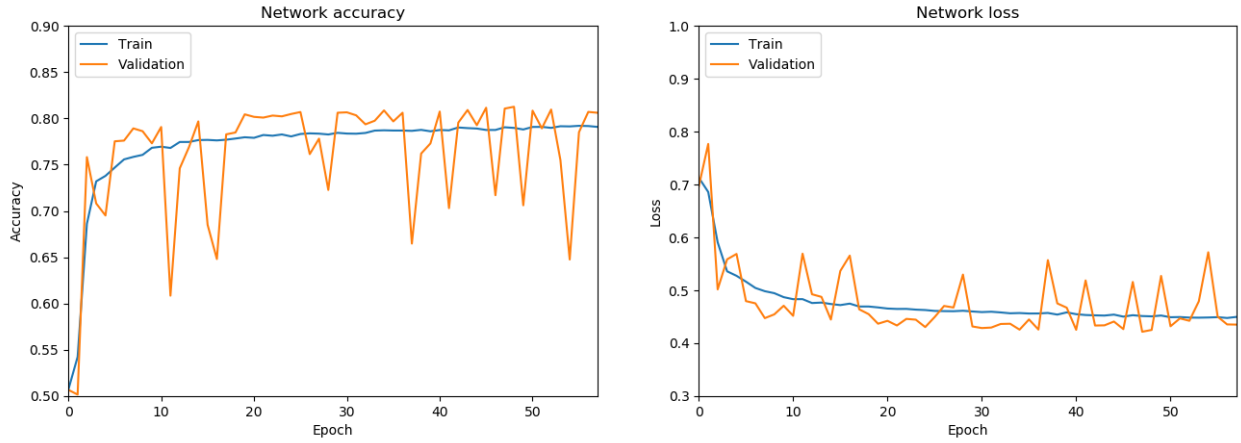
A training data set which consisted of 60,000 different shapes and a validation set with 20,000 was generated. In both, half the shapes had holes and half didn't. These holes were placed randomly in the larger shape, with a maximum range, h_range= 0.4. The radius of the holes were set to an average of, h_r= 0.1, with a variance of sig_hr= 0.01.

This was solved relatively easily and after 47 epochs, the training had achieved an accuracy of 81.1% on the validation data. Figure 3 shows how the training progressed. As can be seen, after less than 10 epochs, the network was already approaching it's solution. However, the validation accuracy was still fluctuating significantly.

Since this was still being solved relatively easily, the average hole radius was decreased to approximately the size of the surface variation, h_r= 0.05 with a variance of sig_hr= 0.005. With this change, it should be impossible to evaluate whether a hole is on the inside of the shape or on the surface. In some of the shapes, the random variance caused the hole's radial coordinate to become negative. In this case, due to the algorithms in place, it acted as multiple, smaller holes, all connected at the corners, where the perimeter intersected. It was decided that this was not a problem as the resolution on the projections would likely be too small to detect it. Alternatively, it would provide a little extra challenge.

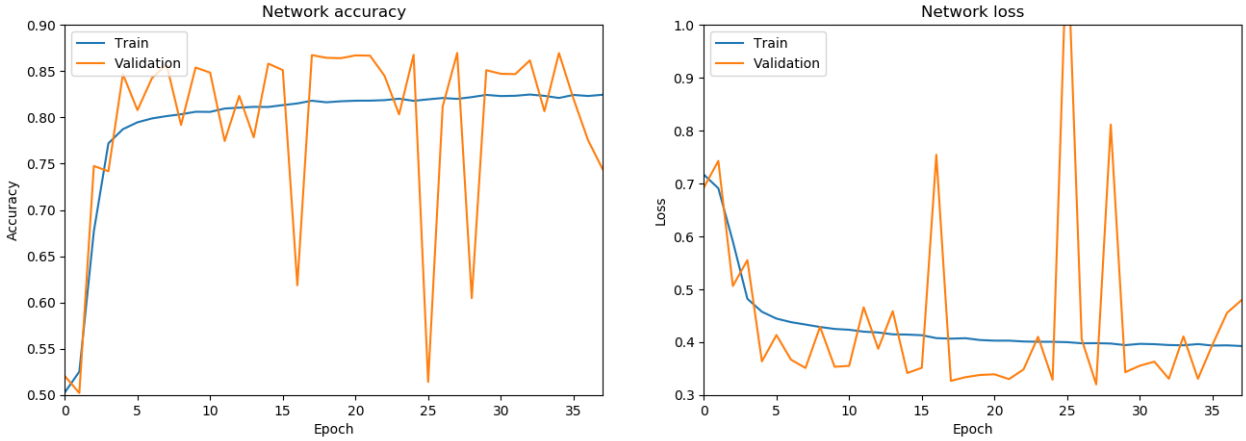
This was left to train for 50 epochs and never made any significant progress, stabilising at predicting both hole and no hole with almost equal probability. After multiple attempts at varying the network structure and

hyperparameter values, it was concluded that it would not be possible to train a network to learn this task with only one input.



(a) Network Accuracy (b) Network Loss

Figure 3: Accuracy and Loss graphs for Training and Validation data sets with h_r= 0.1 and sig_hr= 0.01 using only one projection



(a) Network Accuracy (b) Network Loss

Figure 4: Accuracy and Loss graphs for Training and Validation data sets with h_r= 0.1 and sig_hr= 0.01 using two projections

4.2 Double Input Hole Detection

These two categorisations were then repeated, however this time, the two projections of the shape were generated with a 90 degree angle between the two. This required the network structure to be modified to the structure in figure 5. This first portion of the network processed each projection in parallel. Only one set of layers was trained for this, and both projections were passed through identically. They were then concatenated along the channel axis so that they could be processed together.

When attempting the categorisation of holes with radius of approximately the surface variation, h_r = 0.05, the same result was obtained after 50 epochs. No significant training occurred and the network approached predicting both categories with equal probabilities.

However, when categorising holes with double the radius, h_r = 0.1, a higher accuracy of 87.0% was trained after only 27 epochs. The training progress is shown in figure 4. This was expected, as adding another projection provided more information for the network. The intention was that the network would use this information to interpret any surface features and distinguish them from holes in the particle.

To verify that the network was using multiple inputs, another set of data was generated, with 2,000 shapes with only one projection, then this was passed through twice, as two separate projections. If the network was using the two projections properly, two identical projections at 90 degrees to each other should make little sense and the network should struggle to identify them. The accuracy did drop to 78.0%, however, this was comparable to the accuracy achieved with one image after 27 epochs. It appeared as though the network was processing each projection individually and then average the two results in some way.

This was also repeated with the parallel section of the net removed, concatenating the layers before the convolutional layers. However, this produced slightly lower accuracies, and was less effective overall. As a result, it was determined that simply providing two inputs would not be sufficient, and the network would have to be encouraged to use both in some other manner.

4.3 Chirality Detection

Since the network wasn't learning to use multiple images, spirals were introduced as a feature which could only be recognised when using multiple images. The parameters in table 6 were used to generate 60,000 shapes, with an equal split of clockwise and anticlockwise spirals for the training data, and a further 20,000 for validation data. The spirals had an average radius of, s_r = 0.5, with a variance of sig_sr = 0.05, and they were distributed a maximum of s_range = 0.1 away from the centre of the larger perimeter.

Parameter Name	Value
s_s	0.04
sig_ss	0.008
s_t	0.01
sig_st	0.002
s_t_max	0.1
sig_st_max	0.02

Table 6: Parameters used in the generation of spirals

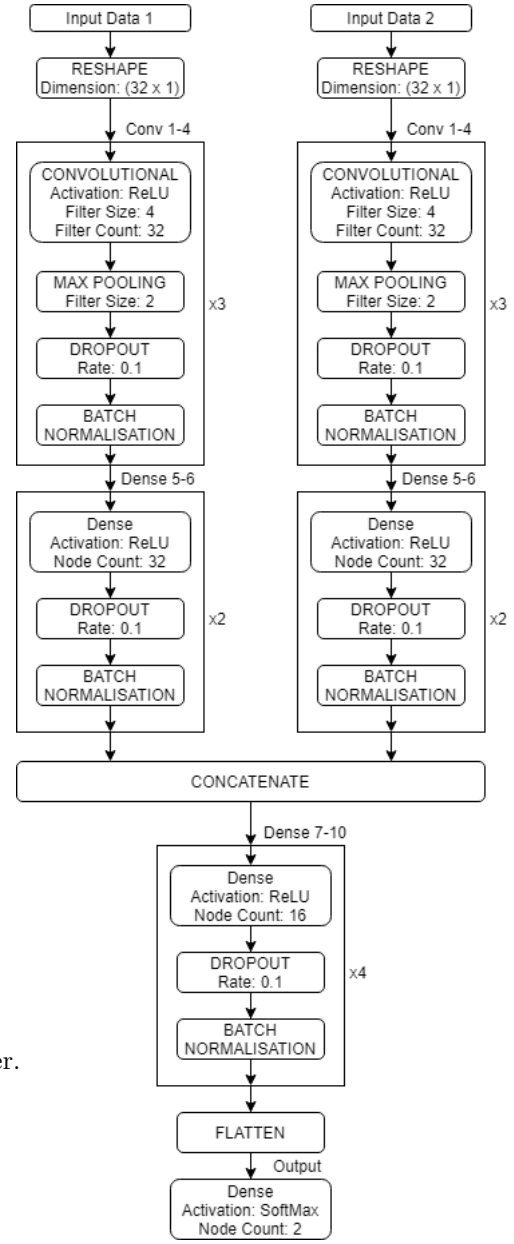


Figure 5: The Neural Network structure used to train for shapes with holes and two inputs.
Trainable parameters: 12,962

Initially, this didn't work and the networks settled on equally predicting both. Three changes were required in order for the network in figure 5 to learn this categorisation. The most significant was increasing the number of data points. With only 32 points in each projection, the network didn't have enough information to identify the chirality of the spirals, even with the other parameters set to generate extremely simple shapes.

It was also necessary to set the variance on the centre of the larger shape, sig_c = 0. Presumably, rotating the particle about an axis with a varying position created a task which was too difficult to solve. Interestingly though, this was not the case with the position of the spiral in the sphere, s_range. This had far less of an impact and the network could still solve the task, even with this set relatively high, as long as the actual particle

was rotated through its centre. This is possibly due to the particle being the more random shape which needed to be identified so it could be ignored.

The final change was to remove the pooling layers from the network. These layers were removing information by only using the highest feature activations.

After training for 37 epochs, an accuracy of 90.7% was achieved. Figure 6 shows typical shapes and projections used and figure 7 shows the training progress.

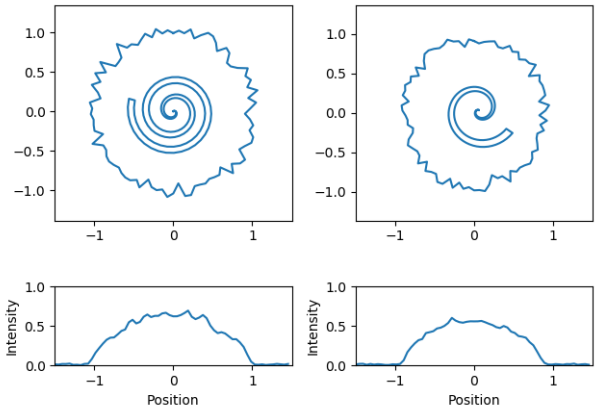
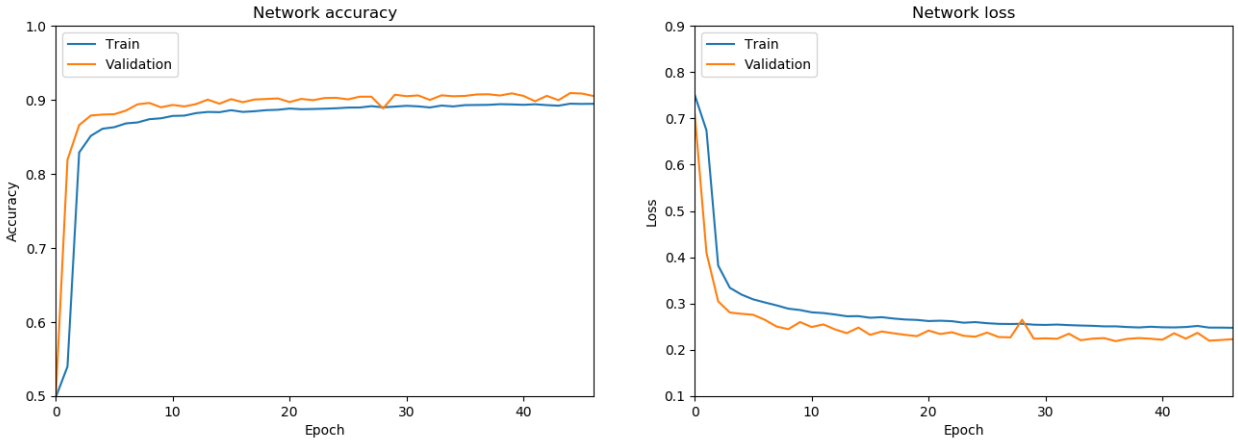


Figure 6: Sample shapes used in the categorisation of particles with spirals, with parameters defined in section 4.3

As can be seen, this trained very quickly to to use multiple images with a high accuracy and then, over the next 30 epochs made very small improvements. It was then decided to use this to train a network which would use multiple images to predict holes properly.



(a) Network Accuracy

(b) Network Loss

Figure 7: Accuracy and Loss graphs for Training and Validation data sets with spirals, with parameters defined in section 4.3

4.3.1 Side Note on Third Category

While attempting to train a network on spirals, a much larger network was used, the structure is shown in Appendix B. While changing parameters, a third output category was added to the network which was never used. It was discovered that this allowed the network to train on tasks where it otherwise failed.

When looking at the outputs of the network, this third output very quickly drops to small values of order 10^{-4} for the majority of cases. However, occasionally an output will equally predict all three categories. When looking at the shapes and projections which cause these predictions, there is nothing which makes them stand out from the rest so it is uncertain why this makes a difference. It is possible that these projections don't give enough information about the shape and this was the best way to minimise the loss function, effectively turning this into an uncertain answer for the network. Although, the network could do this and provide even answers without a third category.

4.4 Transfer Learning

Since the network could be trained to categorise spirals relatively quickly, it was hoped that some of the methods used by the network could be applied to the identification of holes via transfer learning. This is simply the method of retraining a network to complete a task after it has already been trained to complete a different one. If the network had trained to filter out the larger shape when categorising spirals, this behaviour would then already be learnt by the network when attempting to identify holes.

In practice, this was not the case and the results were almost identical to the results in section 4.2, with the variations being attributed to the randomness of the learning process. These tasks were either too dissimilar to have any method carry over, or the gradient to learn the hole identifier was too steep and caused the network to unlearn useful behaviour before it could learn to utilise it.

4.5 Dual Learning

It was therefore proposed to attempt learning of both tasks simultaneously. That way, the spiral categorisation behaviour could be reinforced while the network was training to use it. However, while a large amount of the project time was committed to this method, it proved to be ineffective. There was no consistent behaviour in the training, however, it never resulted in a more successful training. The training would vary from both tasks failing to train, to only one training or, occasionally, both tasks would successfully train, however, they never achieved a higher accuracy than either task individually, and would often achieve lower.

It was initially thought that maybe the network wasn't large enough to complete both tasks simultaneously, however, after increasing the size and altering the structure of the network, it was still ineffective. The weights of the two tasks in the loss function were varied, however, attempting to prioritise either task resulted in the same outcome. Another loss function was written which effectively decreases the impact of the spiral categorisation on the loss function as the accuracy increased. This was, again, ineffective.

Finally, there was an attempt at applying transfer learning to this method. First a network would be trained to categorise spirals. This would then be retrained on a data set with shapes which contained both spirals and holes, attempting to perform both tasks. While this did train more reliably, it still hit accuracies lower than training either individually and when performing the check with a single projection, inputted twice, it still appeared to be analysing each individually and averaging the result. It was therefore concluded that this method wasn't going to work. Either the method is flawed, or the two tasks were too different to actually compliment each other. As the transfer learning also had no impact on the training, and training seemed to start from the beginning every time, it is believed that the later is the case.

5 Further work

Due to time constraints, this is as far as the project achieved, however, repeating the transfer learning and dual learning methods described in section 4.4 and 4.5 for more similar tasks may be valuable. Had time permitted, this would have been expanded upon by training a network which identified smaller, clockwise spirals specifically from a data set containing clockwise, anticlockwise and no spirals. By incorporating both anticlockwise spirals and no spirals into one category, it is thought that the tasks would be more similar for the network.

It is also possible that the network is having a problem with the hole being a lack of material, and the spiral being extra material. The training may have been more successful had the hole been an area of denser material as well, or had the spiral actually been removing material.

6 Conclusion

Classifying particles using multiple projections proved to be easier than expected. As long as the particle was rotated about its centre, a neural network could reliably be trained to classify features which required two projections with a high degree of accuracy, 90.7%. However, when a feature could be classified with only projection, it was very difficult to encourage the proper use of multiple projections and the network appeared to simply average the predictions of the two, individual projections.It is possible that this was due to flawed methods, however, it appears more likely to be a result of a picking a task with too few similarities and further exploring these methods with other tasks which are more similar may prove to be more successful.

References

[1] F. Chollet, et al, Keras, March 2015; `https://keras.io`,

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, ...X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, Novemeber 2015. `arXiv:1603.04467[cs.DC]`. Software available from `https://tensorflow.org`.

[3] A. Karpathy, Convolutional Neural Networks for Visual Recognition, April 2020;
`http://cs231n.github.io/convolutional-networks/#conv`

[4] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning Book: Convolutional Networks, pp. 326-366, 2016;
`http://www.deeplearningbook.org/contents/convnets.html`

[5] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, June 2014;
`http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf`

[6] S. Ioffe, C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, February 2015; `arXiv:1502.03167[cs.LG]`

[7] X. Glorot, A. Bordes, Y. Bengio, "Deep Sparse Rectifier Neural Networks", Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, vol. 15, pp. 315-323, Apr 2011;
`http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf`

[8] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning Book: Optimization for Training DeepModels, pp. 290-304, 2016;
`http://www.deeplearningbook.org/contents/optimization.html#pf14`

[9] M. D. Zeiler, ADADELTA: An Adaptive Learning Rate Method, December 2012; `arXiv:1212.5701[cs.LG]`

Appendices

A Ray Tracing

Notation: $\begin{bmatrix} a \\ b \end{bmatrix}$ is a vector indication a in the x direction and b in the y direction

Start with a ray going through a data point at $\begin{bmatrix} x_0 \\ 0 \end{bmatrix}$ in the direction $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$. This is defined by,

$$\hat{r} = \begin{bmatrix} x_0 \\ 0 \end{bmatrix} + t_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{12}$$

Where t_1 is a variable to indicate distance along the line. Then consider two points A and B at $\begin{bmatrix} x_a \\ y_a \end{bmatrix}$ and $\begin{bmatrix} y_b \\ y_b \end{bmatrix}$ respectively. The vector between the two points is defined by the equation,

$$\hat{s} = A + t_2(B - A) \tag{13}$$

$$= \begin{bmatrix} x_a \\ y_a \end{bmatrix} + t_2 \begin{bmatrix} x_b - x_a \\ y_b - y_a \end{bmatrix} \tag{14}$$

Where t_2 is a variable to indicate distance along the line. An intersect occurs when $\hat{r}$ equals $\hat{s}$,

$$\begin{bmatrix} x_0 \\ 0 \end{bmatrix} + t_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_a \\ y_a \end{bmatrix} + t_2 \begin{bmatrix} x_b - x_a \\ y_b - y_a \end{bmatrix} \tag{15}$$

This gives two equations which can be solved to find t_1, the vertical distance required to reach an intersect point,

$$x_0 = x_a + t_2(x_b - x_a) \tag{16}$$

$$\frac{x_0 - x_a}{x_b - x_a} = t_2 \tag{17}$$

and

$$t_1 = y_a + t_2(y_b - y_a) \tag{18}$$

$$= y_a + \frac{x_0 - x_a}{x_b - x_a}(y_b - y_a) \tag{19}$$

B Neural Network with Strange Category Behaviour

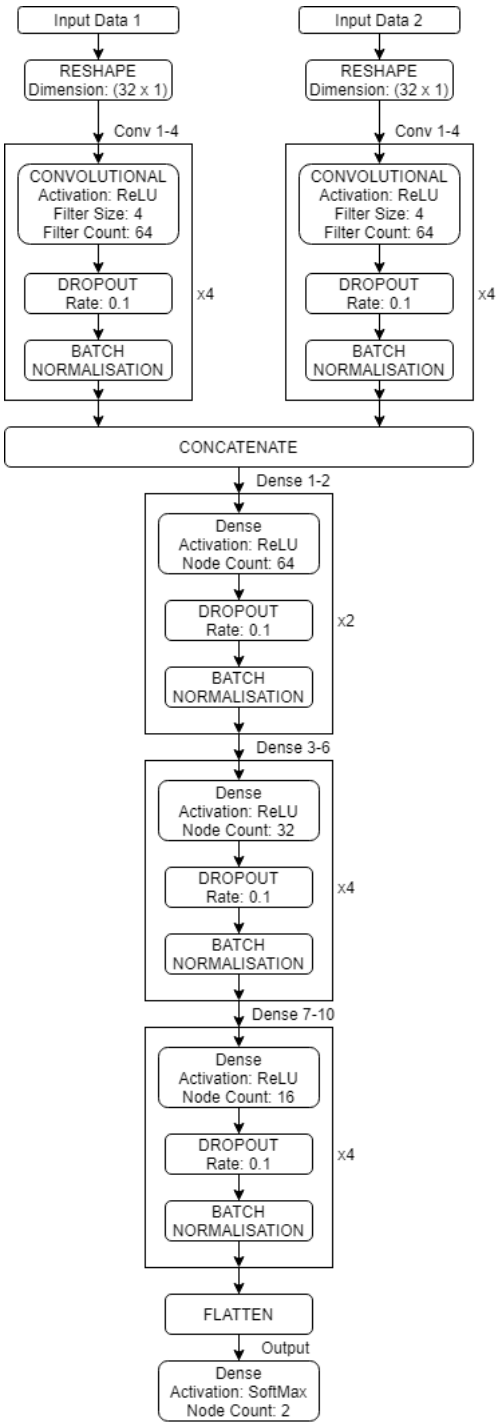


Figure 8: The Neural Network with the strange category behaviour described in Section 4.3.1