B8-01 Alpha Particle Identification Project Report

Candidate Number: 1073415

April 22, 2025

Abstract

Detecting alpha particles in the result of Rutherford scattering experiments requires hand counting of alphas in thousands of images, taking large amounts of time as well as likely leading to unusually high human error. The task is simple and repetitive but alpha particles and background noise vary in shape and intensity meaning that peak detection algorithms perform poorly. However, the large amount of data available makes the task promising for machine learning. In this report a model is proposed for counting alpha particles in these images which has a 98.8% accuracy on low angle data counts and can reduce counting time by around 90% for high angle runs again with 79.0% accuracy as long as the camera settings are held constant. The model uses convolutional neural network approach with supervised learning training on both synthetic images and real labelled images. However for high angles the model generalises poorly to different camera settings and reasons for this as well as potential improvements to the model are explored.

1 Introduction

In 1911, Geiger and Marsden performed the now famous Rutherford experiment, verifying experimentally that the angular distribution of alpha particles scattered by thin sheets of foil was in agreement with the the Rutherford formula predicted by theory [20].

$$\frac{\mathrm{d}\sigma}{\mathrm{d}\Omega} = (\frac{Zze^2}{16\pi\epsilon_0 T})^2 \frac{1}{\sin^4(\frac{\theta}{2})} \tag{1}$$

In order to achieve this result, Geiger and Marsden spent hours in a dark lab, hand counting the flashes caused by alpha particles on their apparatus. Nowadays we can collect data to be processed later by recording charge deposits at the angle of interest in images, however the most accurate method of detecting alpha particles remains hand counting. Hand counting through thousands of images is a strong limiting factor on the amount of data it is possible to label, but an algorithm must be able to differentiate between alpha particles, which can vary in exact shape and brightness, and the large amounts background noise, which appears randomly in many shapes. The large variation in the appearance of images makes the task poorly suited to a normal algorithm but potentially well equipped to be treated as a machine learning task.

In this report I outline an effective approach to achieve automated counting with a machine learning algorithm and its results, as well as the limitations of the algorithm and ideas for how to improve its performance.

The counting problem is treated as a supervised learning problem - labelled data is used to train the model - and as a regression problem - the model aims to output a number, the count. The model maps a $256 \times 256 \times 1$ pixel image to a single number output using a convolutional neural network. Convolutional neural networks have achieved high accuracy in computer vision tasks such as image classification - labelling an image from a pre-specified set of labels - and counting objects in images for tasks such as crowd counting or cell counting in microscopy images [22]. However, the images outputted by the experiment are different to that from microscopy or a CCTV camera. The images are sparse - most pixels have value 0 - and the shape of alpha particles is simple, potentially making them harder for a model to distinguish from background. Due to the time consuming nature of hand counting images, we also lack a large labelled dataset for training.

The model uses convolutional layers to extract features as well as residual connections to combat the sparsity of the data. It is trained in two stages, first on synthetic data and then it is fine-tuned on a smaller dataset of real images with the dataset size increased by data augmentation. Images are preprocessed to set each pixel value to either 0 or 1 before being inputted to the model, with the aim of reducing the dependency of the predicted count on the overall brightness of an image. Results on an independent dataset are presented as well as an analysis of performance and ideas for improvement.

2 Neural Networks and Machine Learning

Machine learning is the idea that a computer can use data to learn how to complete a task, this task is to map an input to an output. When there is no clear and simple function to map between the two, machine learning attempts to approximate a hypothetical function by learning from the data.

One subclass of machine learning algorithms is a neural network. Historically inspired by the function of neurons in the brain, each neuron in a layer takes a number of inputs x_i and applies weights W_i to each of them as well as a bias b to obtain an output $\sum_i W_i x_i + b$ which is usually then passed through a non-linear activation function - such as the sigmoid or hyperbolic tangent function - to obtain an output y [7].

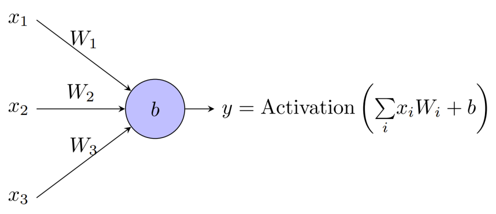

Figure 1: Diagram of an artificial neuron for machine learning, reproduced from [19]

In a feed forward neural network, or what is often known as a fully connected layer when part of a larger model architecture, each hidden layer (layers in-between the input and output) has neurons, each of which takes inputs from every neuron in the previous layer and passes its output to every neuron in the next layer.

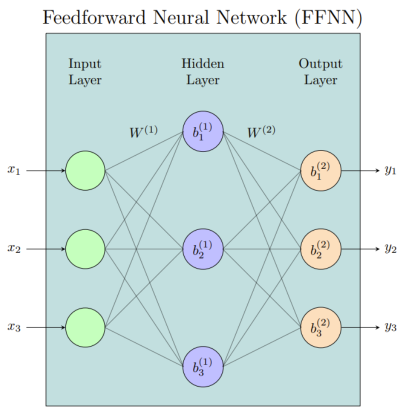

Figure 2: Diagram of a Feed Forward Neural Network, reproduced from [19]. This network has an input and output layer and one hidden layer with three neurons between them.

In supervised learning, training is done with a labelled dataset, where the model is given both an input and its expected output - the label. The model starts with random initial weights and biases (parameters) and provides an output which is compared to label using a cost or loss function to quantify the difference. Gradient descent is then used to update the weights and biases of each neuron, the gradient of the loss with respect to the parameters, θ, is calculated at each step and the parameters are then stepped in the direction of steepest gradient descent [7]. This is repeated many times in the hope that improvements in performance on the training set generalise well to new data.

$$\theta^{t+1} = \theta^t - \alpha \nabla_\theta \text{Loss}(\theta^t) \tag{2}$$

As well as a large number of parameters which the model sets through training, there are also so-called hyper-parameters which are not set by training but which we must specify initially. For example the learning rate - how large a step the model takes each time it updates its parameters. Hyper-parameters are set through experimentation in order to optimise the model's performance.

Neural networks are an extremely powerful tool which have been shown to perform well on a large and growing number of tasks and it has even been proven that a neural network with only one internal hidden layer using sigmoidal non-linear activation functions can approximate any continuous function [5]. However this proof does not tell us how to find this approximation, and the no free lunch theorem for optimisation proves that there is no universal algorithm which is always optimal to find an intended solution: there is no one optimal way to train neural networks for every problem [21]. Therefore, we must use intuition for the problem at hand and test different methods - loss function, learning rate and other hyper-parameters - of training to improve performance. Here I will give an overview of the theory behind the main techniques and tools which proved effective in model testing.

2.1 Convolutional Neural Networks

Feed forward neural networks take a vector of inputs which are mapped to a vector of outputs, this means that in order to use an image as an input the image must first be flattened to a vector. The loss of spatial information caused by the flattening of the 2D image to a 1D vector and the extremely large size of this vector meant that tasks such as image classification which require image inputs - so called computer vision tasks - proved very difficult for neural networks.

One of the first model architectures to make large improvements to this was LeCunn's 1989 model which used a new technique called convolutional neural networks to achieve high accuracy in identifying handwritten digits for US Zip-code reading [15].

A Convolutional Neural Network (CNN) uses small filters which are convolved with the image to build many feature maps which are of smaller dimensions than the original image. A small filter, say 3x3 pixels, is moved across the image. The value of the pixel in the corresponding feature map is the dot product of the filter with that section of the image, with zero-padding - surrounding the edges of the image with 0s - each output feature map can be the same 2D shape as the input. In any given convolutional layer, the number of filters and their size can be specified. A larger filter will take less computation power as it requires fewer steps to sweep across the whole image, but may miss small details. Using more filters requires more computational power but builds up more feature maps, each of which can become sensitive to different possible patterns in your images. As the model trains, the parameter values for the filters are updated and learnt.

The other common layer which is used with Convolutional layers are Pooling layers. Pooling layers act to reduce the spatial dimensions of their input and are usually either max pooling or average pooling. Looking at, for example, a 2×2 block of pixels at a time, the pooling layer turns this into one pixel by either taking the maximum value in the block, or the average (mean) value across the block.

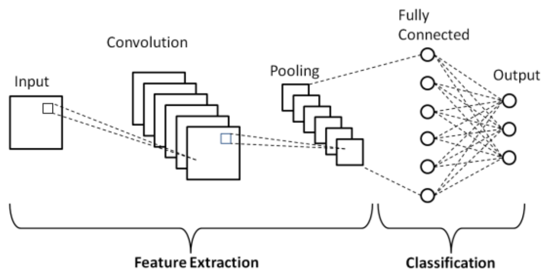

Figure 3: Diagram of a simple convolutional neural network architecture with a convolutional layer followed by pooling flattening and a fully connected layer before the output layer. Reproduced from [1]

Together, sequences of convolutional and pooling layers turn an image with larger spatial dimensions into many feature maps of smaller dimensions, each with the ability to highlight different features in the image. Say the input is a $256 \times 256 \times 1$ image - with the 1 representing a black and white image with only one channel - a convolutional layer with 32 filters would output an object of shape $256 \times 256 \times 32$, a following pooling layer with a 2×2 filter would make it $128 \times 128 \times 32$.

In order to then process these feature maps to a final non-image output, after rounds of convolutional and pooling layers, the resulting output is flattened into a vector which is then inputted to a feed forward neural network - normally known as dense or fully connected (when every neuron in one layer is connected to every neuron in the next) layers [6].

2.2 Over-fitting

Over-fitting is the idea that the model can become so good at learning the training data that performance deteriorates on new data. If the model is complex enough and the training set not suitably large, we can image a scenario where the model develops the ability throughout training not to be able to identify, say, an apple by its shape and colour but through simply remembering which images in the training set have an apple in them. This will then lead to very poor performance on new images even though the apples in them might look very similar to those in the training set.

When training a model, data is usually split into three categories: training, validation and test [7]. Training data is used to train the model and update the weights. After each round of training, performance on the validation set is evaluated and monitored but the parameters are not updated, normally the final model selected is the one that performs best on the validation set. The final model is then used on the test set which has not been used at all up until this point and this performance is the final quoted performance of a model. This is all to try to avoid over-fitting and to test the model's ability to generalise to data that it has not been trained on. The difference between the model's performance on the training and the validation datasets gives a measure of the extent to which the model is over-fitting and this is what we want to reduce.

There are many ways to do this such as penalising the model for being overly complex - known as normalisation or regularisation [7] - but one very simple and powerful tool to help with over-fitting in dense layers is known as dropout. One can imagine training many models on different datasets and averaging the outputs on all of them when using them on new data to try to improve predictions, in essence dropout aims to implement a version of this in the model itself [17].

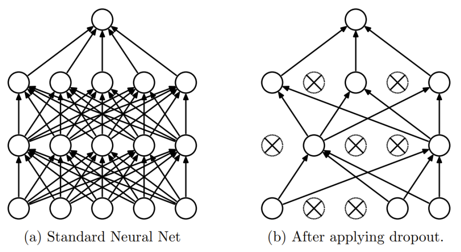

Figure 4: Diagram of dense layers in a neural network before (a) and after (b) dropout, reproduced from [17]

A dropout layer adds no parameters to the model and only one hyper-parameter: a probability. In training this dropout parameter p represents the probability for a node to be present in training. Therefore every time the model is trained on an image, dropout means there is a slightly different model being used which must learn independently to achieve the task. The aim is that neurons do not depend on each other too much and so become more robust to variations in the data. After training, the probability becomes an additional weight, which is applied to all of the neurons so the total weight is pW, effectively averaging over all of the different models that have been trained. Decreasing p reduces over-fitting more but since more nodes drop out each time, the performance of the model also deteriorates and so a balance between the two must be found. It is worth noting that in TensorFlow, when using a dropout layer the hyper-parameter that is set is actually $(1 - p)$.

2.3 Gradient Descent and Convergence

Machine learning uses gradient descent to step parameters in the direction which minimises the loss function and to improve performance on data, however as gradients are propagated backwards through the network they become sometimes exponentially small and so parameter updates stagnate and learning stops. The

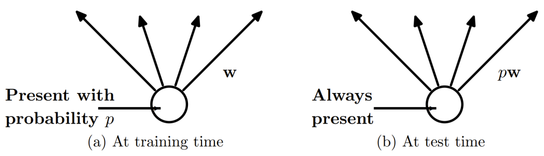

Figure 5: Effect of dropout (a) during training and (b) during prediction, reproduced from [17]

non-linear functions which are applied to neuron outputs allow the model to learn complex patterns in data but some previously common activation functions are susceptible to what is known as the problem of vanishing gradients [4].

The sigmoid and hyperbolic tangent (tanh) functions were common activation functions but both have the problem that they only work well for values around 0 or 0.5 respectively. Otherwise the functions saturate and so their gradient becomes exponentially small.

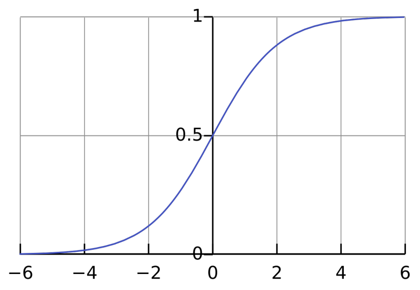

Figure 6: The Sigmoid function - reproduced from http://commons.wikimedia.org/wiki/File:Logistic-curve.svg.
The function only changes substantially when inputs are close to 0.

Currently, rectified linear units - the ReLU function - are more common, where the activation $f(x) = \max(0, x)$ is 0 for negative values and linear for positive [2]. ReLU has the benefit that it does not saturate and that its derivative is a constant (1). For a given input, a subset of the neurons will take negative values and so will be 0, on the remaining set of neurons, computation is linear and so gradients flow well and the process is less computationally expensive - as put by [8]: we can see the model as an exponential number of linear models that share parameters.

Another milestone for machine learning was the introduction of the ADAM (adaptive moment estimation) algorithm for optimisation in 2015 [11]. Normal gradient descent uses the product of the learning rate and the gradient of the loss function so if the gradient is small, the steps are small.

ADAM combats this in two ways. Firstly with the idea of of gradient 'momentum' which depends on the previous gradients and so can push the parameters out of local areas of low gradients. The momentum vector is updated each step: $\vec{m} \leftarrow \beta_1\vec{m}+(1-\beta_1)\nabla J(\theta)$, where α is the learning rate and β is known as the friction and corresponds to how quickly older gradients decay in the momentum vector.

ADAM also uses the second moment of the gradient which is used the scale down the gradient in the direction in which it is the largest with the aim that the parameters will step towards the minima rather than just in the direction of the steepest slope [7]. The variance is calculated $v \leftarrow \beta_2 v + (1-\beta_2)\nabla J \otimes \nabla J$ where $\otimes$ represents element-wise multiplication. When parameters are updated the gradient is then scaled (element-wise) by $\sqrt{v+\epsilon}$ where ϵ is a small smoothness term.

Putting both of these ideas together (along with an initial boost to $\vec{m}$ and $\vec{v}$ so they are not initially 0), ADAM updates parameters with momentum scaled down by the variance such that $\theta \leftarrow \theta + \alpha\vec{m} \oslash \sqrt{\vec{v}+\epsilon}$ (where $\oslash$ represents element-wise division) [11].

2.4 Residual Neural Networks

Another method of combatting the problem of vanishing gradients, and that of degrading model performance for models with a very large number of layers, is with residual neural networks [9]. Residual networks have skip connections where the input to a residual block x is added to the output of multiple layers - for example convolutional layers - in the block, making the output $f(x)+x$ rather than simply $f(x)$. The hypothesis is that if the required mapping of the block is $H(x)$, it is easier for the block to learn the mapping $f(x) := H(x) - x$ which is then outputted as $f(x)+x$, especially if $f(x) = I$ (the identity mapping) thereby stopping the layers from degrading performance [9].

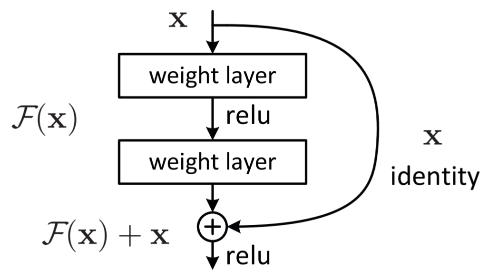
x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
$\mathbf{x}$
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 7: A diagram of a residual block with two layers and a skip connection, reproduced from [9]

3 The Model

The model aims to output just one number which is then rounded to the nearest integer to be used as the predicted count for the image. It uses a simple convolutional neural network architecture, with the addition of a residual block to aid performance. There are a total of three convolutional layers, all with 32 filters, and two max pooling layers. After the first convolutional and pooling layers, a skip connection is implemented, meaning their output later added to that of the following two convolutional layers, as can be seen in the diagram in Fig.8. The sum is then flattened into a vector

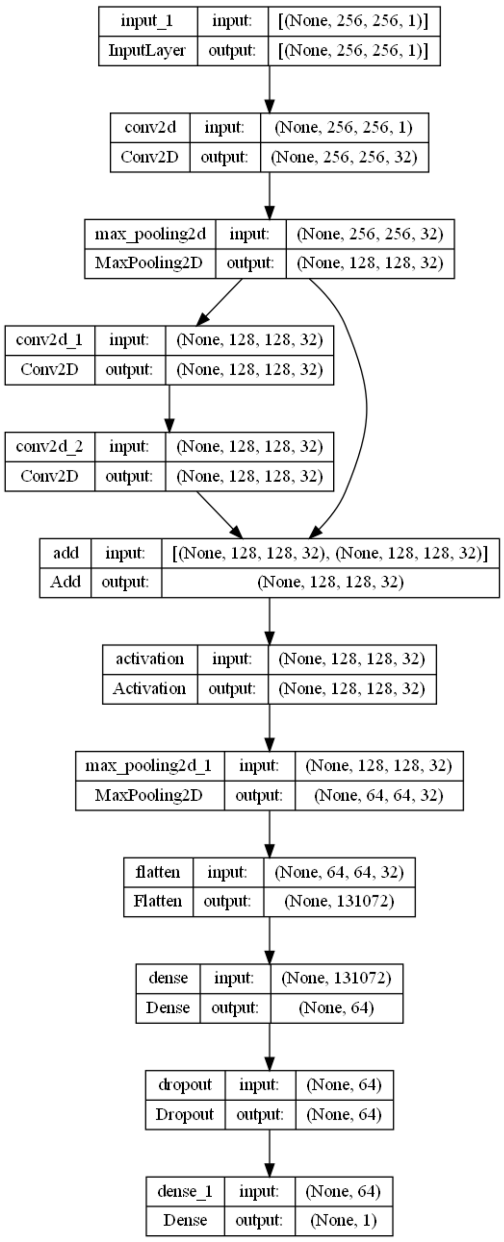
input_1 | InputLayer | input: [(None, 256, 256, 1)] | output: [(None, 256, 256, 1)]
conv2d | Conv2D | input: (None, 256, 256, 1) | output: (None, 256, 256, 32)
max_pooling2d | MaxPooling2D | input: (None, 256, 256, 32) | output: (None, 128, 128, 32)
conv2d_1 | Conv2D | input: (None, 128, 128, 32) | output: (None, 128, 128, 32)
conv2d_2 | Conv2D | input: (None, 128, 128, 32) | output: (None, 128, 128, 32)
add | Add | input: [(None, 128, 128, 32), (None, 128, 128, 32)] | output: (None, 128, 128, 32)
activation | Activation | input: (None, 128, 128, 32) | output: (None, 128, 128, 32)
max_pooling2d_1 | MaxPooling2D | input: (None, 128, 128, 32) | output: (None, 64, 64, 32)
flatten | Flatten | input: (None, 64, 64, 32) | output: (None, 131072)
dense | Dense | input: (None, 131072) | output: (None, 64)
dropout | Dropout | input: (None, 64) | output: (None, 64)
dense_1 | Dense | input: (None, 64) | output: (None, 1)

Figure 8: Diagram of the model architecture, produced by the code in Appendix A

before a dense layer of 64 neurons, with dropout set to 0.15, and a final output layer of size 1 - the predicted count. The total model has 8,440,833 parameters, a relatively large model due to the large size of the input images - the data is still of size 64×64 when flattened which leads to a large number of parameters in the first fully connected layer.

The model stems from tests of different numbers of layers and hyper-parameters. The number of layers and filters had little effect on performance. Adding a residual block caused a significant increase in performance, despite this not being a deep model. However residual blocks are more computationally intensive and so I was unable to test the effect of having multiple blocks. Similarly, varying the number of dense layers and their nodes had little effect although dropout helped to limit overtraining and the 0.15 hyper-parameter was set through testing to minimise over-fitting while maximising performance.

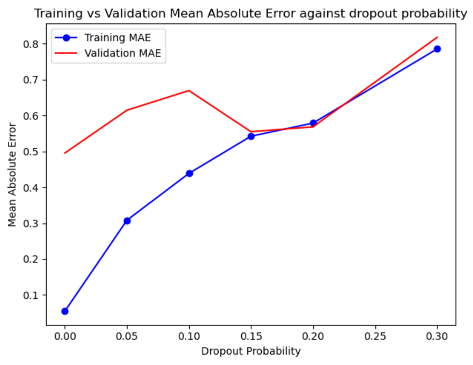

Figure 9: Plot of training and validation error compared to dropout hyper-parameter for a test model trained on 2000 synthetic images for 20 epochs

There is much discussion within the field of machine learning and particularly computer vision about whether improvements in performance come largely from model architectures or from the data and specifically the amount of the training data being used [23] [18]. Improving performance is not as simple as just adding more layers to a model or increasing the amount of data in the training set. Since we have found that the increasing the complexity of the model architecture has relatively little impact on the performance it might be argued that another architecture altogether would be better suited the problem - this is discussed in section 7 - however it also means that we must look carefully at the data that the model is being trained on in order to improve performance, this is the subject of the following section.

4 The Data

The experimental project obtains data in the form of black and white (single channel) $256 \times 256 \times 1$ pixel images, from an ADVACAM MiniPIX where charge deposits create bright spots on a black background. The alpha particles themselves are generally much brighter than background and take up around 9 pixels. These need to be distinguished from the background noise, most commonly in the form of gamma rays which are very frequent and cause single isolated white pixels. The occasional cosmic muon passing through the camera can also be easy seen as they cause distinct straight thin lines in the image.

Obtaining many of these images is easy, experimental runs can last hours with photos being taken continuously throughout, producing thousands of images. However, creating a labelled dataset is much more difficult. Labelling the data is time consuming, and it is easy to miss or over-count alphas when looking through over a thousand images which could lead to poor quality training data and so poor model performance.

The distribution of alphas across images is also not uniform. There is a probability associated with an alpha being scattered to a certain angle and so in the fixed unit of time of the exposure setting, we can describe the distribution of alphas in each image with a Poisson distribution. Normally, models are trained on data who's distribution reflects that of the entire sample in order to improve the model's capacity to generalise [13]. However the rate λ of the distribution varies with the angle, and with the exposure of the image. As we aim for a model which is robust to these variations in conditions, this raises the question of how to collate a training dataset. For example, if the model is trained exclusively on high angle images which have almost no alpha particles, it will struggle to be accurate when applied to a set of images taken at $0°$ which have an expected value of around 4.5 alphas per photo. These high angle images are also much easier to hand count as most of them have no alpha particles and so there is an overabundance of high angled labelled data compared to low angle where labelled data is most limited.

All these factors mean that despite having an overabundance of images, we are still very constrained by the amount of labelled data when training a machine learning model. This is a common issue facing datasets in many different fields, such as in medical imaging where [22] works around this issue with techniques such as data augmentation and using synthetic data. Here we take a similar approach.

4.1 Preprocessing

Data has to be preprocessed to allow the model to learn optimally and to highlight features correctly. For example if one input has a much larger range, it will dominate the models performance. Here we divide the input values by 255 to normalise them between 0 and 1 as the camera outputs pixel values between 0 and 255. The model was still very sensitive to brightness which varied lots between different angles and runs and so each pixel in the image is then set to 0 or 1 based on if it is above or below a threshold set - 0.05 was used in training. This stabilised the model and improved generalisation as it made it less sensitive to intensity and (presumably) more focused on the shapes of alpha particles. The preprocessing also has the benefit of making alphas look more alike, depending on the overall brightness of an image, intensities are scaled up or down by the camera and so the outer pixels of an alpha can appear very dim, distorting their shape. By setting all values to 0 or 1, alphas appear larger and more uniform in shape, it also accounts for varying brightness of alpha particles.

It is worth noting that although data preprocessing can be used to highlight the features in the data and improve learning - feature engineering - representation learning aims to use models to learn representations of the data which make it easier for another model to then understand and learn from the data [3]. The difficulty with both of these techniques for this data is the very large number of pixels with value 0. If these were to be included in the calculation of average value or standard deviation then the values of bright pixels would make very little difference.

4.2 Data Augmentation

The model needs to be able to distinguish alpha particles from background noise. However the location or orientation of alpha particles within this random noise does not matter. In order to increase the size of the labelled dataset, count-preserving and shape preserving transformations are applied to the images in order to produce new distinct images with a known count. These include rotations of the image by increments of $90°$, reflections horizontally or vertically, and the horizontal or vertical translation of the image by a number of pixels, wrapping the image around to the other side of the image.

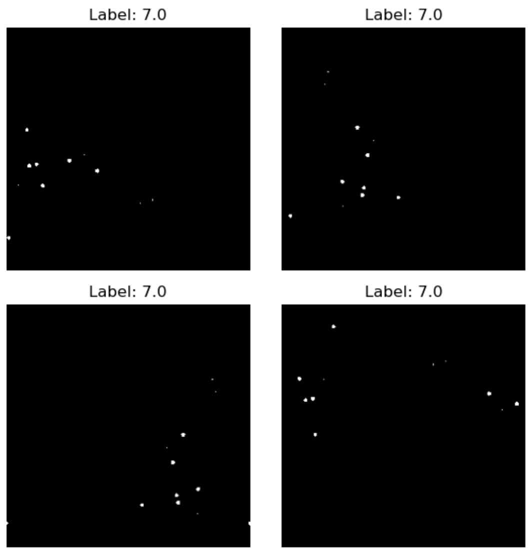

Figure 10: Image and three augmentations, all preprocessed with binary processing and intensity scaled up $100\times$, produced by the code in Appendix D

These transformations serve to modify the appearance of the image and the location of alpha particles within it, all while keeping the number the same so the model can be trained on a larger number of labelled examples. The code in appendix D takes in an image and randomly applies a circular shift of a random amount in both the x and y direction, then a random flit horizontally and vertically before a rotation by a random multiple of $90°$. These random transformations can be applied any number of times to make as many new images. However, in practice it is best not to create too many images from one as it becomes more likely that images are similar and also the model will end up being trained on the same base image many times which can lead to poor generalisation as it becomes very good at labelling the specific image and its variations. The field of data augmentation is still expanding as explored in [16], and there is much more that could be done to edit the brightness and contrast of the images. Here

I have only explored editing the locations of alphas in the image with the aim that the model improves at identifying alpha particles regardless of their position.

4.3 Synthetic data

Even with data augmentation, the lack of alpha-dense labelled data at low angles makes assembling a large dataset difficult. Augmenting the existing data also does nothing to change the distribution of alpha particles across the dataset. Since the model is aiming to be able to identify alphas no matter the angle, rather than ensuring the distribution matches that of a specific angle, it is best to ensure a uniform distribution in the hope that the model will remain accurate no matter the number of alphas. Creating a uniformly distributed - i.e. the frequency of images with one alpha is the same as the frequency of images with any other number of alphas up to a maximum - would mean discarding a large number of labelled images and so reduce the dataset even further.

To try to improve upon this, as inspired by [22], the model can be initially trained on synthetic data. Using the abundance of high angle images with no alpha particles as background samples and then adding synthetic alphas over the top, we can create many synthetic images with a uniform distribution of alpha particles to train the model on.

Figure 11: Two synthetic images produced by the code in Appendix C and their labels, preprocessed using binary processing and scaled up by 100×

The code in appendix C shows the creation of a synthetic dataset for training the model. A background image is chosen randomly from a set of 541 alpha particle-free images, then normalised (all pixel values divided by 255). Then a random number of alpha particles are added to it, with the number chosen from a uniform distribution between 0 and 19 - 19 was chosen as the maximum value to encompass all values, with a small buffer, found in 0° images. The alphas are placed at with their centre at random coordinates, a maximum intensity between 0.75 and 1 and then a Gaussian shape with standard deviation of 2 to match the observed size of real alpha particles in images.

More background samples would increase the variation seen by the model. However, even in a 10,000 image dataset, each sample is repeated around 20 times. Since there are 20 different possible values for the number of alpha particles in the image, and they are randomly dispersed in the image, there is still large variation in the training set.

5 Training

The model was trained in two stages, first on a uniformly distributed set of synthetic data, and then on real data. Due to the large difference in alpha particle distribution between low angle and high angle images, two separate models are fine-tuned, one for high angle runs and one for low angles.

5.1 Pre-training

The model was first trained for 50 epochs on a dataset of 10,000 synthetic images. A mean absolute error loss function was used so as to not place a large importance on few outliers which could have been labelled incorrectly.

$$\text{Loss} = \sum_{i}^{N} \frac{|\text{Label}_i - \text{Prediction}_i|}{N} \tag{3}$$

The learning rate started at 0.001 for 10 epochs, decayed exponentially from epochs 10 to 20 to settle at 0.0001 where it stays until epoch 35 when it halves again. This learning rate scheduling was created based on when training began to stagnate, smaller steps slows learning but also allows the model to find and settle in small minima of optimal performance [7]. The chosen model from this round of training was the epoch with the lowest validation mean absolute error.

5.2 Fine-tuning

The model was then fine-tuned on a real augmented dataset. The difficulty in this was training the model to accurately predict on high angle datasets. High angle images very rarely have an alpha particle, making accurate prediction very difficult for the model. Increasing the number of high angle samples in the fine-tuning set improved performance at these angles but degraded it at lower angles and so the model was split in two, one fine-tuned for high angles, with very few alphas, and one for low angles, with many.

It was clear through testing that low angle models performed best when trained on alpha-dense images and so the model fine-tuned for low angles was trained on a set of 153 0° and 22° images - the 22° images with label 0 were removed to ensure the model was trained entirely on dense images - augmented 7 times to increase the dataset size to 1224.

High angles required more specific testing as including only sparse high angle images in the training set gave the model too few examples with alphas and led the model to count alphas very rarely - if the training set is 500 images with only 10 alphas, just labelling everything as 0 leads to a mean absolute error of 0.02. The

High Angle Models			
Dataset	Accuracy	Count	Real Alphas Found
50/50 Ones and Zeros	89.00%	64/29	24/29
33% Sparse	84.25%	82	25
40% Sparse	90.50%	54	24
90% Sparse	93.25%	40	21
95% Sparse	93.50%	39	21
All Sparse	94.75%	10	9

Table 1: Different metrics of performance on a high angle test set vary when the fine-tuning dataset is changed. Sparse refers to high angle images.

model was trained with varying ratios of low and high angle images and then the resulting model tested on a small, independent high angle set of 400 images to compare the accuracy - number of images with correct count - the total alphas found - predicted count - and the total number of real alphas found.

The chosen high angle fine-tuning set was the 60% Sparse - made up of 153 low angle dense images augmented to 1224 and 900 sparse images forming a set of 2124 images - since it had a balance of finding a high number of the real alphas whilst keeping the count low meaning it had few false positives.

The fine-tuning was performed for 40 epochs, with an initial learning rate of 0.0005 which halved upon a plateau in the validation loss.

This combination of pre-training and fine-tuning improved performance significantly over simply training on only synthetic or only real data. Before fine-tuning the model identified significantly fewer alpha particles in a 0° test than the low angle fine-tuned model. Similarly, if the pre-training on synthetic data is skipped, the model fails to generalise well to new data.

Table 2 shows the performance of a model with no pre-training (trained only on a dataset of real high and low angle images) and the model before fine-tuning compared to the performance of the high and low angle specialised models after fine-tuning. The high angle test (115°) uses the metric of count as a percentage of real count, the low angle test (0°) uses model accuracy (number of correct images). It is clear that the training on synthetic data teaches the model to count well, with performance being improved significantly by subsequent fine-tuning on real data. However simply training on real data does not allow the model to make progress on the task.

6 Results

Previous testing used data from extreme angles, 0° for low angle and 115° for high angles. It is necessary however to define a boundary between low and high angles based on which model performs best at each

Model	0°	115°
No Pre-training	0.75%	25%
Before Fine-tuning	81%	72%
High Angle Model	98%	—
Low Angle Model	—	91%

Table 2: Different Models' Performance on a high and low angle dataset.

angle. Table 3 shows testing of each model at a range of angles in order to determine that the low angle model performs best at angles of 45° and below and the high angle above this threshold. This made the low angle test set 385 images at angles of 0°, 25°, 45° and the high angle test set 1900 images at angles between 60° and 160°.

Among these 385 low angle images there were a total of 740 real alpha particles, of which the low angle model found 687 (92.8%). The model outputted a count of 731 giving it an accuracy on the count of 98.8%.

The high angle model has a tendency to over-count, as it has very few images that it can under-count on, finding 228 alpha particles in the 1900 images when there were only 100. However it found 79 (79%) of the real alpha particles. There were 130 images which the model miscounted as having an alpha particles when the image did not, and 209 images total counted to have an alpha particles. Therefore, by making the model output the images which it counts to have an alpha particle (11%), the number of images to be hand counted can be cut down by 89% and the resulting count accuracy is 79% (21% error). It is worth noting that by doing the same calculations for using the low angle model, it finds 83% of the real alphas but 26% of the images must be hand counted.

7 Limitations and Improvements

The exposure time of the low angle images vary between 0.1 and 10 seconds with little noticeable effect on performance. The high angle images that the model was trained on as background in the synthetic images and the real images in fine-tuning were 10 second exposures. Later, the experimental project gathered high angle data at an exposure of 20 seconds. When the high angle model predicted on this data, it produced total counts in the hundreds of alphas for 500 images rather than the less than 50 that would be expected. Although the low angle model is robust against changes in exposure, the high angle model is not and fails to generalise to longer exposures, despite the images appearing similar.

We can conclude that it is likely the increase in the amount of background noise that is effecting the accuracy of predictions, meaning the model does poorly at looking for alpha particles based on their shape despite efforts to encourage this such as the binary pre-

	Low Angle Model			High Angle Model		
Angle	Accuracy	Count	Real Alphas Found	Accuracy	Count	Real Alphas Found
0	79%	460/466	449	80%	450	445
25	69%	227/234	209	75%	221	210
45	79%	44/40	29	86%	32	27
60	81%	47/16	10	91%	25	9
100	78%	92/14	10	92%	42	10
115	80%	94/29	25	92%	54	24
130	72%	72/12	12	90%	35	12
145	65%	98/17	16	89%	35	16
160	66%	86/12	10	86%	34	8

Table 3: Performance of the high and low angle optimised models on a test set varies based on the angle the images were taken at, the colours represent the chosen ranges to be designated as high or low angles.

processing. There are two reasons which may play a part in this difficulty. Firstly, the variation in camera settings, exposure time varies but also the intensity of pixels. High angle images which had no alphas have large amounts of noise present due to the long exposure times but those with alphas often had little to none as the camera likely scales intensities to fit the full range of values available to it. So when a bright alpha particle is incident, noise gets scaled down to be much dimmer, but variations in the brightness of alphas lead to variations in the brightness of the background. In future it would be preferable from the point of view of a counting algorithm to keep camera settings constant so intensities of background and alphas remain more constant to aid machine counting.

Secondly, the sparsity of the data. CNNs have been developed to work with dense data, tasks such as image classification assume the object takes up most of the image [14] and counting algorithms for tasks such as crowd counting or cell counting assume many subjects are being counted across the image [22]. In our images, over 90% of the pixels are 0, and at high angles the model is searching for just one alpha particle which takes up less than 10 pixels in amongst over 65,000. Successful counting models often attempt to map an image to a density map which is then integrated over the image to find a count [22]. When this was attempted for our data, although the model could reduce the background significantly, estimating a count from the density map proved difficult as the count was so low and there were so many pixels which should be 0, that even just small variations from 0 in these 65,000 pixels can lead to huge variations in the count, even when a threshold was set to ignore low pixel values. Even a second model trained to count from these maps performed worse the initial model.

When the model was simply trained on a mix of 10 and 20 second exposures for high angles it failed to count either effectively. Other attempts to work around the variations in brightness, or to standardise images such as normalisation by the average non-zero pixel value in an image or stacking two 10 second exposure images together to attempt to standardise all images in training and testing to 20 second exposures showed no improvement over the model presented in this report. It is likely that a more complex approach is needed to improve upon these results, particularly in the case of high angle images.

One method could be to input the exposure (or additional camera settings) to the model, to be added as another node after the convolutional layers are flattened. This however would require the abandoning of pre-training on synthetic images as it would not be possible to assign an exposure to them and so a larger real dataset labelled with the count and camera settings would need to be collated.

Abandoning the aim to directly output a count and instead localising alpha particles in an image by placing bounding boxes around them seems like the most promising avenue to test as the boxes can then be counted using an algorithm, such as in [10]. Localising before counting would aim to help the model to use the shape of alphas to identify them, hopefully making the model more robust to variations in the amount of background. Localising models usually struggle when objects overlap however the nature of the high angle images makes this extremely unlikely.

Relabelling real images with boundary boxes would take substantial time and effort but [12] uses a new architecture based on self-supervised learning where the model is trained on data with no labels, which could improve this problem. By making the model create 'collages' using examples of images to be counted and background, the model trains itself to produce a density map of objects in the image. This could be adapted for by creating samples of real alpha particles which the model makes into input collages and output labels of the collages with bounding boxes to train itself on, exploiting the large amount of alpha particle examples in low angle images, and background in high angle images.

8 Conclusion

This report presents a model for counting alpha particles in images based on an AlexNet style convolutional neural network [14] which outputs a single number count for an image input. By pre-training the model on synthetic data and then fine-tuning on real data, it performs well on alpha-rich low angle images with an error of only 1.2%. These low angle images are more suited to normal counting algorithms due to their high number of alpha particles. However, in the high angle regime where most images have no alphas, the model struggles, even when fine-tuned specifically for this regime. The highly sparse nature of the images causes the model to struggle, and its inability to undercount, since there are so few alpha particles present, leads to very high error rates when counting large numbers of images. It is also very sensitive to variations in the image exposure, where despite images appearing similar, the model performs very poorly when the exposure is increased. Simply training the high angle model on more data with more variation is not enough to improve this. It is likely necessary for the data collection to be optimised to help the model in counting through standardising camera settings. In this high angle regime when working with extremely sparse images it is also possible that the biggest improvements could come from localising alpha particles within images to make the model more robust to changes in background.

References

[1] A. Abdelrahman and S. Viriri. Kidney tumor semantic segmentation using deep learning: A survey of state-of-the-art. *Journal of Imaging*, 8:55, 02 2022.

[2] A. F. Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[3] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[4] J. Brownlee. A gentle introduction to the rectified linear unit (relu), 2020.

[5] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[6] E. A. Fei-Fei Li. Cs231n convolutional neural networks for visual recognition: Module 2 convolutional neural networks.

[7] A. Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow.* " O'Reilly Media, Inc.", 2022.

[8] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[10] P. Jiang, D. Ergu, F. Liu, Y. Cai, and B. Ma. A review of yolo algorithm developments. *Procedia computer science*, 199:1066–1073, 2022.

[11] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] L. Knobel, T. Han, and Y. M. Asano. Learning to count without annotations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 22924–22934, 2024.

[13] W. M. Kouw and M. Loog. An introduction to domain adaptation and transfer learning. *arXiv preprint arXiv:1812.11806*, 2018.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[15] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems*, 2, 1989.

[16] C. Shorten and T. M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

[17] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[18] C. Sun, A. Shrivastava, S. Singh, and A. Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE international conference on computer vision*, pages 843–852, 2017.

[19] Y. H. Teoh. Machine learning and optimization techniques for trapped-ion quantum simulators. Master's thesis, University of Waterloo, 2020.

[20] W. S. Williams. *Nuclear and particle physics.* Oxford: Clarendon Press, 1991.

[21] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[22] W. Xie, J. A. Noble, and A. Zisserman. Microscopy cell counting and detection with fully convolutional regression networks. *Computer methods in biomechanics and biomedical engineering: Imaging & Visualization*, 6(3):283–292, 2018.

[23] X. Zhu, C. Vondrick, D. Ramanan, and C. C. Fowlkes. Do we need more training data or better models for object detection?. In *BMVC*, volume 3. Citeseer, 2012.

A Model Training Code

```python
import matplotlib.pyplot as plt
import keras
import tensorflow as tf
import numpy as np
import os
from keras.layers import Dense, Flatten, Dropout, Activation
from keras.layers import Conv2D, MaxPooling2D, Add, Input
from tensorflow.keras.callbacks import LearningRateScheduler

"""
Trains base model on synthetic data and model with best validation mean absolute
    error
Defines model architecture and then trains it on a dataset which has been loaded
    into images and labels
Images are preprocessed so all values are 0 or 1
"""


# Takes in an image and make all pixels either 0 or 1
def binary_image(image, threshold=0.05):
    binary = tf.cast(image > threshold, tf.float32)
    # image > threshold gives a Boolean value at each pixel
    # Casting it as a tf.float32 makes True=1 and False=0
    return binary


# Load tfrecord dataset into np.arrays - written with help of Chat-GPT
# Takes in path to the dataset file, and the size of the dataset
# Outputs arrays images (shape: dataset_size, 256, 256, 1) and labels (shape:
    dataset_size, 1)
def load_tfrecord_to_numpy(tfrecord_path, dataset_size):
    # Initialize empty arrays for images and labels
    images = np.zeros((dataset_size, 256, 256, 1), dtype=np.float32)
    labels = np.zeros((dataset_size, 1), dtype=np.float32)

    # Load dataset and shuffle
    dataset = tf.data.Dataset.load(tfrecord_path).shuffle(buffer_size=
        dataset_size)

    # Iterate over the dataset to collect images and labels
    for i, (image, label) in enumerate(dataset.take(dataset_size)):

        if image.shape[-1] == 3:  # Check if it's RGB (3 channels)
            image = tf.image.rgb_to_grayscale(image)  # Convert to grayscale (1
                channel)

        image = tf.image.resize(image, [256, 256])  # Ensure matches expected
            image shape
        image = tf.cast(image, tf.float32)
        image = image/255  # Normalise values to between 0 and 1
        image = binary_image(image, 0.05)  # Process all values to 0 or 1

        # Ensure output is an array
        if isinstance(image, tf.Tensor):
            image = image.numpy()
        if isinstance(label, tf.Tensor):
            label = label.numpy()

        images[i] = image
        labels[i] = label

    return images, labels


# Load data into numpy arrays
loaded_images, loaded_labels = load_tfrecord_to_numpy("Datasets/
    improved_large_synthetic_data.tfrecord", dataset_size=10000)
print(loaded_labels[0])
print(loaded_images.shape)


# the model:

# Input layer
inputs = Input(shape=(256, 256, 1))

# First conv + pooling
x = Conv2D(32, (5, 5), activation='relu', padding='same')(inputs)
x = MaxPooling2D(pool_size=(2, 2))(x)

# Residual block with two convolutional layers
residual = x
x = Conv2D(32, (5, 5), padding='same', activation='relu')(x)
x = Conv2D(32, (5, 5), padding='same')(x)
x = Add()([x, residual])  # Skip connection
x = Activation('relu')(x)
x = MaxPooling2D(pool_size=(2, 2))(x)

# Flatten and Dense layers
x = Flatten()(x)
x = Dense(64)(x)
x = Dropout(0.15)(x)  # Dropout to reduce overfitting
outputs = Dense(1)(x)  # Output of shape (1)

# Compile model with Adam optimiser and mean absolute error as loss function
model = keras.Model(inputs=inputs, outputs=outputs)
model.compile(loss="mae", optimizer='adam', metrics=['mae'])

model.summary()


# Define function to decay learning rate
def exponential_decay(epoch, lr):
    if epoch < 5:  # start big as mae starts very large
        return 0.005
    if epoch > 35:  # half lr again for final epochs
        return 0.00005
    if epoch > 20:  # eventually settle at 0.0001
        return 0.0001
    if epoch >= 12:  # after 12 epochs start reducing exponentially
        return lr * tf.math.exp(-0.05 * (epoch-11))
    return 0.001


# Create the scheduler
lr_scheduler = LearningRateScheduler(exponential_decay)


# Define model checkpoint to save the model with the best validation MAE
checkpoint_best = tf.keras.callbacks.ModelCheckpoint(
    "Base-Residual-Model",  # Filename
    monitor="val_mae",
    save_best_only=True,
    mode="min"  # Lower MAE is better
)


# Train the model for 50 epochs, with 20% of the data kept aside for validation
history = model.fit(loaded_images, loaded_labels, validation_split=0.2,
    batch_size=32, epochs=50,
                        callbacks=[lr_scheduler, checkpoint_best])
```

B Model Fine-tuning Code

```
    import matplotlib.pyplot as plt
import keras
import tensorflow as tf
import numpy as np
import os
from tensorflow.keras.callbacks import ReduceLROnPlateau

# Loads pretrained model and finetunes it on real data

# Processing functions as in CNN Training
def binary_image(image, threshold):
    binary = tf.cast(image > threshold, tf.float32)
    return binary


def load_tfrecord_to_numpy(tfrecord_path, dataset_size=350):  # need to edit
   dataset size as I increase it
    # Initialize empty arrays for images and labels
    images = np.zeros((dataset_size, 256, 256, 1), dtype=np.float32)  # Image
        shape (1200, 256, 256, 1)
    labels = np.zeros((dataset_size, 1), dtype=np.float32)  # Labels shape
        (1200, 1)

    # Load dataset using tf.data.Dataset.load() (This assumes your dataset is
        already serialized)
    dataset = tf.data.Dataset.load(tfrecord_path).shuffle(buffer_size=
        dataset_size)

    # Iterate over the dataset to collect images and labels
    for i, (image, label) in enumerate(dataset.take(dataset_size)):
        # Ensure the image has shape (256, 256, 3) for RGB
        if image.shape[-1] == 3:  # Check if it's RGB (3 channels)
            image = tf.image.rgb_to_grayscale(image)  # Convert to grayscale (1
                channel)

        # Handle the case where the image might not have the expected shape
            (256, 256, 1)
        image = tf.image.resize(image, [256, 256])  # Resize to ensure it
            matches expected shape
        #image = preprocessing(image)  # need to normalise real data
        # if i remove this i need to divide by 255!!
        image = image/255  # normalise
        image = binary_image(image, 0.05)  # normalise to all 0s and 1s

        # Handle eager execution
        if isinstance(image, tf.Tensor):
            image = image.numpy()  # Convert image tensor to NumPy array
        if isinstance(label, tf.Tensor):
            label = label.numpy()  # Convert label tensor to NumPy array

        # Assign the numpy arrays to the corresponding index
        images[i] = image
        labels[i] = label

    return images, labels


# Load data into numpy arrays
loaded_images, loaded_labels = load_tfrecord_to_numpy("Datasets/Augmented-High-
    Angle-Training-Set.tfrecord"
                                                                          , dataset_size=2124)
print(loaded_labels[0])
print(loaded_images.shape)


# schedule learning rate to reduce after a plateau in learning
reduce_lr = ReduceLROnPlateau(
     monitor='val_loss',    # Metric to watch
     factor=0.5,            # Reduce LR by this factor (e.g., 0.5 = half)
     patience=3,            # Number of epochs with no improvement before reducing
          LR
     min_lr=1e-7,           # Lower bound for LR
     verbose=1              # Print when LR is reduced
)

# Create the scheduler
lr_scheduler = reduce_lr

# Define model checkpoint to save the model with the best validation MAE
checkpoint_best = tf.keras.callbacks.ModelCheckpoint(
     "Finetuned-High-Angle-Model",  # Filename for best model
     monitor="val_mae",  # Monitor validation MAE
     save_best_only=True,  # Save only the best model
     mode="min"  # Lower MAE is better
)


model = keras.models.load_model("Base-Residual-Model")  # Load model to be
    finetuned

# Train Model
history = model.fit(loaded_images, loaded_labels, validation_split=0.2,
    batch_size=32, epochs=40,
                        callbacks=[lr_scheduler, checkpoint_best])
```

C Synthetic Data Generation Code

```
import numpy as np
import tensorflow as tf
from PIL import Image
import os
import random

"""
Generates a synthetic dataset by adding alphas to background images
"""

# Generates an image with a number of alphas added randomly to a sample of
    background
# load a random real image with no alphas to use as the background then add in a
     random number of alphas
def generate_noisy_image(size=256, num_particles=0, intensity=1, spread=1):

    folder_path = "Data/Background"
    image_files = [f for f in os.listdir(folder_path)]
    random_image = random.choice(image_files)
    image_path = os.path.join(folder_path, random_image)
    image = Image.open(image_path)
    # open image and use as background to add alpha over
    image = np.array(image)/255  # normalise and convert to np array

    # repeat so double background
    random_image2 = random.choice(image_files)
    image_path = os.path.join(folder_path, random_image2)
    image2 = Image.open(image_path)
    # open image and use as background to add alpha over
    image2 = np.array(image2) / 255  # normalise and convert to np array

    image = image + image2

    for _ in range(num_particles):
        x, y = np.random.randint(0, size, size=2)  # Random center position
        intensity = np.random.uniform(0.75, 1)
        # make intensity high but not always 1

        for i in range(-spread * 2, spread * 2):
            for j in range(-spread * 2, spread * 2):
                xi, yj = x + i, y + j
                if 0 <= xi < size and 0 <= yj < size:
                    distance = np.exp(-((i ** 2 + j ** 2) / (2 * spread ** 2)))
                    image[yj, xi] += intensity * distance  # Apply Gaussian
                        shape

    image = np.clip(image, 0, 1)  # Ensure values are within [0,1]
    return image

num_samples = 2000

dataset = tf.data.Dataset.from_tensor_slices(([], []))

for i in range(num_samples):
    num_particles = np.random.randint(0, 20)  # Random count 0-19
    img = generate_noisy_image(num_particles=num_particles)
    # Generate an synthetic image
    img = img.reshape(256, 256, 1)
    image = tf.cast(img, tf.float32)

    num_particles = tf.cast(num_particles, tf.float32)

    # Add image and label to dataset
    new_data = tf.data.Dataset.from_tensors((image, num_particles))
    dataset = dataset.concatenate(new_data)

tf.data.Dataset.save(dataset, "improved_large_synthetic_data.tfrecord")
```

D Data Augmentation Code

```
import os
import pandas as pd
import tensorflow as tf

"""
Takes a dataset and augments each image a certain number of times
Creates a new dataset of the old images and new augmented images
"""

num_images = 100  # need to edit based on data being loaded
num_transforms = 8  # number of different transformations including original
image_shape = (256, 256)
image_dir = "Data/Will-Run6/tao05_0deg"

# Create an empty tf dataset or load a dataset to append to
dataset = tf.data.Dataset.from_tensor_slices(([], []))
# dataset = tf.data.Dataset.load(".tfrecord")

# Read labels from csv
df = pd.read_csv("Data/Will-Run6/ta05_0deg-labels.csv", header=None)
Labels = df.to_numpy().reshape(num_images, 1).astype('int')


# Does the data augmentation - takes an image as an input
# Randomly applies a transformation such as wraps and flips
# Outputs a new image which has been augmented
def apply_random_transformations(image, max_shift=64):

    # Randomly translates an image with wrap-around
    shift_x = tf.random.uniform([], minval=-max_shift, maxval=max_shift, dtype=
        tf.int32)
    shift_y = tf.random.uniform([], minval=-max_shift, maxval=max_shift, dtype=
        tf.int32)

    # apply random reflection
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_flip_up_down(image)

    # Circular shift using tf.roll
    image = tf.roll(image, shift=[shift_y, shift_x], axis=[0, 1])

    # apply random rotation
    k = tf.random.uniform(shape=[], minval=0, maxval=4, dtype=tf.int32)
    image = tf.image.rot90(image, k=k)

    return image


# Generates a certain number of augmented images from an input image and outputs
    them in a tensor
def generate_augmented_images(image):
    augmented_images = [image]  # Keep original
    for _ in range(num_transforms-1):  # Generate 7 different augmentations to
        add
        augmented_images.append(apply_random_transformations(image))

    return augmented_images  # Returns a tensor of shape (num_transforms, 256,
        256, 1)
```

```
# Load images in the folder and augments them to create new dataset
for i, filename in enumerate(os.listdir(image_dir)):
    if i >= num_images:
        break
    filepath = os.path.join(image_dir, filename)
    image = tf.io.read_file(filepath)
    image = tf.image.decode_png(image)
    image = tf.cast(image, tf.float32)  # Reads image into a tensor

    # Save images in list of tensors
    aug_images = generate_augmented_images(image)

    # now have augmented images saved in aug_images, need their label
    label = tf.cast(Labels[i], tf.float32)
    labels = tf.repeat(label, num_transforms)  # Repeat old label as many times
        as needed
    # create a tf dataset with images
    new_data = tf.data.Dataset.from_tensor_slices((aug_images, labels))
    # Concatenate new data to the existing dataset
    dataset = dataset.concatenate(new_data)

tf.data.Dataset.save(dataset, "run6_0_augmented_8.tfrecord")


# Dataset checks - chatgpt
loaded_dataset = tf.data.Dataset.load("run6_0_augmented_8.tfrecord")

# Check the first few elements
for img, lbl in loaded_dataset.take(3):
    print("Image shape:", img.shape, "Label:", lbl.numpy())

# Check the dataset sizes match
original_count = sum(1 for _ in dataset)
loaded_count = sum(1 for _ in loaded_dataset)
print(f"Original dataset size: {original_count}, Loaded dataset size: {
    loaded_count}")
```

E Framework for Prediction on Large Dataset

```python
import keras
import numpy as np
import os
import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image
import tensorflow as tf
import re

"""
Predicts on entire runs
Save data input in one folder, with subfolders for each angle
Inputs are the folder path and names for the output files
Outputs the count and variance for each angle saved in a csv file
For high angle it will also output the file names which is claims to have alphas
    in a csv
High Angle threshold set at 46 based on testing, batch size can be increased if
    lots of data
"""

# Variables to set:
folder_path = "Data/Will-Run7"  # Input name of folder with subfolders of images
    at each angle
output_predictions_file = "Run7-Predictions.csv"  # Filename.csv to save outputs
    to
output_alpha_images = "Run7-Alphas-test.csv"  # Filename.csv to save paths of
    high angle images with alphas for checking

high_angle_threshold = 46  # angle at which runs designated high angle
high_angle_model = "Finetuned-Model---High-Angle-Optimised"  # model for high
    angle runs
low_angle_model = "Finetuned-Model---Low-Angle-Optimised"  # model for low angle
    runs

image_shape = (256, 256)
batch_size = 128  # Number of images to be processed at a time by the model


# function to turn images into all 0s and 1s to remove issues with brightness
# input and output both images, sets any value in image greater than threshold
    to 1, any below to 0
def binary_image(image, threshold=0.05):
    binary = tf.cast(image > threshold, tf.float32)
    return binary


# preprocessing code to open image path, convert to array and normalise
# Outputs an image as np.array
def image_preprocessing(image_path):
    image = Image.open(image_path)  # Convert to grayscale
    image = image.resize(image_shape)
    image = np.array(image)
    image = image/255  # normalise
    image = binary_image(image, 0.05)  # set every value to 0 or 1
    image = np.expand_dims(image, axis=-1)
    return image
```

```
folder_results = []
high_angle_ones = []

# Loop through subfolders - processing each individually
for subfolder in sorted(os.listdir(folder_path)):
    subfolder_path = os.path.join(folder_path, subfolder)

    if not os.path.isdir(subfolder_path):
        continue  # Check this is a folder

    match = re.search(r"(\d+)", subfolder)  # Extract degrees the run was
        performed at
    angle = int(match.group(1)) if match else None  # Convert to int only if
        there's a match

    # Different models for high and low angles - defaults to low angle
    if angle >= high_angle_threshold:
        model = keras.models.load_model(high_angle_model)
        model.trainable = False
        high_angle = True
        print(f"Processing high angle folder: {subfolder}")
    else:
        model = keras.models.load_model(low_angle_model)
        model.trainable = False
        high_angle = False
        print(f"Processing low angle folder: {subfolder}")

    # Collect images in batches
    img_arrays = []
    predictions_list = []
    img_paths = []

    # Now in a subfolder, process each image in it
    for img_name in sorted(os.listdir(subfolder_path)):
        img_path = os.path.join(subfolder_path, img_name)

        if not img_name.lower().endswith((".png", ".jpg", ".jpeg")):
            continue  # Skip non-image files

        img_arrays.append(image_preprocessing(img_path))
        img_paths.append(img_path)

    # Take images and input to model for predictions
    batch_array = np.array(img_arrays)
    print(batch_array.shape)
    batch_predictions = model.predict(batch_array, batch_size=batch_size)
    batch_predictions = np.round(batch_predictions)  # Round to nearest integer
    # Store all predictions
    predictions_list.extend([max(0, int(item))] for item in batch_predictions)

    #  If a high angle run, save the names of files it claims to have alphas to
        be checked by hand
    if high_angle:
        # Save image paths where prediction > 0
        for path, prediction in zip(img_paths, batch_predictions):
            if prediction > 0:
                high_angle_ones.append([subfolder, path, prediction])
                # For high angle runs, save names of images with alphas to be
                    checked

    # Compute total prediction sum and variance for the subfolder
    print(predictions_list)
    total_prediction = np.sum(predictions_list)
    variance_prediction = np.var(predictions_list)

    # Save the results per folder
    folder_results.append([angle, total_prediction, variance_prediction])


# Write the results from each folder into a dataframe, then a csv
df_folders = pd.DataFrame(folder_results, columns=["Degree", "Sum", "Variance"])
df_folders.to_csv(output_predictions_file, index=False)

# Save alpha image paths
if high_angle_ones:
    df_alpha_images = pd.DataFrame(high_angle_ones, columns=["Folder", "Image 
       Path", "Prediction"])
    df_alpha_images.to_csv(output_alpha_images, index=False)

print("Outputs saved to: " + output_predictions_file)

# Plot prediction sum against angle
# Load the CSV file with folder predictions
df = pd.read_csv(output_predictions_file)

# Ensure sorted by angle
df = df.sort_values(by="Degree")

# Plot total prediction sum vs. angle
plt.figure(figsize=(8, 5))
plt.plot(df["Degree"], df["Sum"], marker="o", linestyle="-", color="b", label="
   Total Prediction")

plt.xlabel("Angle (degrees)")
plt.ylabel("Total Alphas detected")
plt.title("Alphas detected vs. Angle")
plt.legend()

plt.grid(True)

plt.show()
```