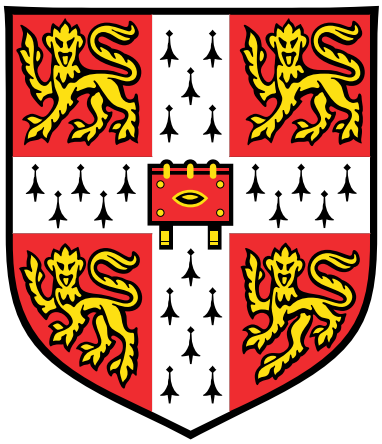
Detection of Parity Violation in Snails through Unsupervised Learning

Candidate 8212R

Supervised by Dr Christopher Lester



Cavendish Laboratory
University of Cambridge
Easter Term 2023

Declaration

Except where specific reference is made to the work of others, this work is original and has not been already submitted either wholly or in part to satisfy any degree requirement at this or any other university.

Candidate 8212R, Easter Term 2023

Online Laboratory Notebook can be found here:

https://universityofcambridgecloud-my.sharepoint.com/:w:/g/personal/jdg53_cam_ac_uk/EUQ5xMLZ2SJNhxc3l9kWkf8Bzbfmm9jCDDcx_EWARgVoAw?e=nO2ZY0

Abstract

There may be undiscovered mechanisms of parity violation that can be detected in Large Hadron Collider energy deposit data. Current theoretical models propose mechanisms of parity violation that would not be detectable in the Large Hadron Collider. Because of this, Large Hadron Collider experiments have not been used to look for any novel beyond the Standard Model parity-violating mechanisms. This project demonstrates how parity violation can be detected in multichannel data, such as Large Hadron Collider data, using a parity-odd neural network. A machine learning algorithm was trained and tested on various datasets of Helicoidea and Stylommatophora RGB colour images, as well as on an idealised 3D snail model. Extraneous sources of parity violation were removed from the image sets. It was found that the network could successfully detect parity violation in a population of snails and, when the number of channels was increased, the ability of the network to detect parity violation improved. Diluting the parity-violating snail datasets with achiral slug images reduced the level of parity violation detected. Different qualities of JPEG images were tested and it was found that JPEG compression did not introduce any parity violation. This project is an important step in the implementation of this methodology to detect parity violation in Large Hadron Collider experiments in the future.

Contents

1 Introduction

The Large Hadron Collider (LHC) is incapable of detecting any parity violation that arises from mechanisms proposed by current theoretical models [1]. The LHC lacks initial state polarisation and detector spin sensitivity [2], which has meant that it has not been used to look for any novel parity-violating mechanisms. However, there may be undiscovered mechanisms of parity violation which produce signals that are visible in LHC energy deposit data. This project aims to build on the work done in *"Using unsupervised learning to detect broken symmetries, with relevance to searches for parity violation in nature,"* by Lester and Tombs [3], to prove that searching for parity violation in LHC data is a worthwhile endeavour. Lester and Tombs used the greyscale human handwriting MNIST dataset [4] to test a symmetrised machine learning approach to the search for parity violation. Using this approach, they were successfully able to identify parity violation in the single-channelled MNIST dataset, opening the door to future investigations. However, LHC energy deposit data is multi-channelled, encompassing data gathered from multiple detectors. The primary objective of this research is to investigate the feasibility and effectiveness of using machine learning techniques to identify parity violation in multichannel datasets, aiming to develop a model applicable to LHC energy deposit data. Specifically, we focus on the detection of parity violation in images of snails, which serve as an illustrative model for studying symmetry-related phenomena.

1.1 Parity Violation and its Discovery

The action of the parity operator $\hat{P}$ on a vector introduces a sign inversion in all three spatial components. In effect, the parity operator acts as a mirror – whether some object is parity-even or parity-odd depends on whether that object is unchanged ($\psi(x) = \psi(\hat{P}x)$) or has a sign inversion ($\psi(x) = -\psi(\hat{P}x)$), respectively, under a mirror inversion. Vectors are odd under a parity transformation. For example, under this transformation, the angular momentum (an axial vector, $L \times r$) of a system is left unchanged, whereas its electric field (a vector field) is sign-inverted. In chemistry, molecules can be chiral – these molecules have non-superimposable mirror images. Certain macroscopic objects have an intrinsic chirality, such as human hands, which have a handedness, or the shells of snails, which have dextral or sinistral spirals. The momenta of final state particles produced in particle physics interactions, such as those at the Large Hadron Collider, can be likened to the position vectors of the atomic constituents of molecules. If an interaction is parity-violating, we will see a favoured arrangement of final state momenta over a population of interactions. We are interested, abstractly, in finding out whether particle physics interactions are more likely to produce final state particles that 'look' a certain way than another, in a breaking of

the symmetry of the Standard Model.

Quantum electrodynamic (QED) and quantum chromodynamic (QCD) interactions conserve parity, but the weak interaction has been found to violate parity through an unknown mechanism. However, the parity violation present in the weak interaction is not sufficient to explain the matter/anti-matter asymmetry we observe in the universe today. It is therefore important to search for new mechanisms of parity violation.

In 1957, Wu et al [5] observed that beta particles emitted from Cobalt-60 nuclei were more likely to be emitted antiparallel to the orientation of the nuclei. Both magnetic moments and magnetic fields are described by axial vectors, which are parity-even. Therefore, the likelihood of beta particle emission in one direction should be equal to the radially opposite for parity to be conserved. This was not seen. The magnetic moments of the nuclei were aligned with the field of an electromagnet, and it was found that there was a preferential direction for the beta particles to be emitted, an observation that could only be described through a parity-violating mechanism. The parity-violating mechanism behind these results is unknown [6] – knowledge of the mechanism was not required to achieve Wu's results.

1.2 Objectives

This project will use coloured RGB images of snails and slugs to determine whether parity violation can be detected in multi-channel datasets. The work of Lester and Tombs stands alone in proposing this parity violation detection methodology and as such, this project is an exciting contribution. This project promotes the number of channels in the data from one to three (from greyscale to coloured RGB), and aims to answer the following questions:

1. Can we detect parity violation in multi-channel data using a symmetrised machine learning algorithm?
2. Does increasing the number of channels in the data affect the level of detectable parity violation?
3. To what extent can this methodology be applied in future research, particularly in the context of Large Hadron Collider experiments?

These questions will be addressed through the theoretical framework and methods discussed in Section 2, followed by a description of the dataset curation process. The implementation of the neural network is explained in Section 4, followed by the publication and discussion of the results. Finally, the project will be concluded. The source code for the network can be found in the Appendix.

2 Theory and Methods

2.1 Properties of Parity-Odd Functions

Under the action of the parity operator, $\hat{P}$, the handedness of a system is flipped. For 2D images of 3D objects, $\hat{P}$ acts as a mirror, flipping the image. We can choose the orientation of the mirroring so, in this project, we defined our parity operator to flip images horizontally.

Applying a parity-odd function, $f(x)$, to a set of n images $\{x_i \mid 1 \leq i \leq n\}$, we obtain a set of values $\{f(x_i) \mid 1 \leq i \leq n\}$. To have symmetry under the parity transformation $\{f(x_i)\} \rightarrow \{f(\hat{P}x_i)\}$, we require $\{f(x_i)\}$ to consist of elements symmetrically distributed about zero. This means that our image set $\{x_i\}$ must contain either (a) solely parity-even (horizontally symmetric) images such that $f(x_i) = 0 \ \forall i$, or (b) images such that $f(x_i) = f(\hat{P}x_j)$ where $i \neq j$ for at least one pair of images. (b) can occur if an image x_j transforms into another image x_i under $\hat{P}$, so that $\hat{P}x_j = x_i$. As $\hat{P}^2 = 1$, $\hat{P}x_i = x_j$. Therefore, to achieve parity symmetry over the image set, the set must be constructed such that if we were to horizontally flip each image, it would look identical to the original set.

If a set of images is distinguishable from its mirrored set, then $\{f(x_i)\} \neq \{f(\hat{P}x_i)\}$ and the parity symmetry is broken, or violated. Therefore, $\{f(x_i)\}$ will contain elements with an asymmetric distribution about zero. Thus, if we find that $f(x_i) > 0$ for more than 50% of the images in a set, where $f(x)$ is parity-odd, the image set violates parity. This suggests that a valid way to search for parity violation is to construct different parity-odd functions and test them for asymmetry over a dataset [3].

Any asymmetry we see is evidence of parity violation in the dataset – due to our use of a parity-odd function, the idea of a false positive is irrelevant. We may test a parity-violating dataset and (due to an incorrect choice of the function $f(x)$) not detect any parity violation, but we cannot generate parity violation in a dataset that is symmetric under parity. It is possible, however, that parity violation may enter our dataset in ways extrinsic to the raw data itself. For example, certain lossy compression methods, such as compression into JPEGs, may introduce parity violation that was not present in the uncompressed images themselves. It is important to ensure that we check for this when using lossy image formats.

2.2 Parity-Odd Neural Networks and other Symmetries

Now that we have a recipe for detecting parity violation in a population, how can we construct parity-odd functions? A simple way to construct a parity-odd function $f(x)$ is to use another function $g(x)$, defining $f(x) \equiv g(x) - g(\hat{P}x)$. Here, $g(x)$ can be any arbitrary function without any specific relationship to the parity operator. Applying $\hat{P}$ to x in this

equation will give $f(x) = -f(\hat{P}x)$ as required.

In this project, a parity-odd neural network $f(x)$ was constructed using a sub-network $g(x)$. We used machine learning to find a function $g(x)$, specific to the chosen dataset, that generated asymmetry in the output set $\{f(x_i)\}$. A randomly initialised function $g(x)$ would likely produce a symmetric set $\{f(x_i)\}$ even when tested on a parity-violating dataset. However, with training on a training subset of the dataset, $g(x)$ can learn to find asymmetry. Machine learning allowed us to construct functions $g(x)$ that have the best chance of detecting parity violation in a given dataset. The construction of the network $f(x)$ out of $g(x)$ ensures that any observed asymmetry in $\{f(x_i)\}$ is due to parity violation.

By using a sub-network $g(x)$, a parity-odd network $f(x)$ can be designed to have other desirable symmetries. This project followed the approach used in Lester and Tombs [3]. We constructed $f(x)$ as follows:

$$f(x) = g(x) + g(\hat{R}_{180}x) - g(\hat{P}x) - g(\hat{R}_{180}\hat{P}x) \tag{1}$$

$\hat{R}_{180}$ represents a 180° rotation about the centre of an image. $f(x)$ is still parity-odd, even after incorporating the 2$^{\text{nd}}$ and 4$^{\text{th}}$ terms. These terms add invariance under 180° rotations about the centre of an image, as $\hat{R}_{180}^2 = 1$.

Lester and Tombs [3] used MNIST handwriting images as a toy model as a substitute for energy deposits in an LHC-type collider. Interactions have cylindrical symmetry about the beam axis and 180° rotational symmetry perpendicular to it. The cylindrical nature of the ATLAS detector at CERN is shown in Figure 1. 180° rotational symmetry arises from the isotropy of the interacting beams in the collider, and their ability to be freely interchanged.

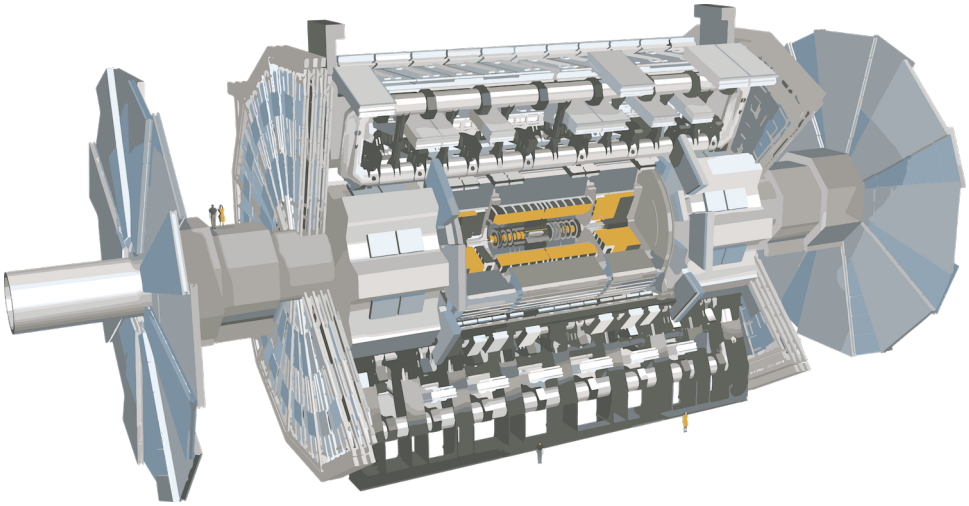


Figure 1: The ATLAS detector, displaying cylindrical symmetry [7]

The machine learning model in Lester and Tombs was designed to include both symme-

tries – $f(x)$ was constructed from $g(x)$ as in Equation 1, to include 180° rotational symmetry, and $g(x)$ was designed to be invariant under rotations about the cylindrical axis. The presence of white space at the top and bottom of MNIST images allowed for this implementation since rolling the images downwards did not create a clear horizontal line through the image. With images of snails, however, there would be a clear line. This line would not appear in an energy deposit, so it was decided that it would be more similar to an LHC-type collider to not include this symmetry in this project's network. Lester and Tombs proved that this additional symmetry could be implemented successfully, and so this invariance could easily be re-added if the dataset was more appropriate.

2.3 The Sub-Network $g(x)$

The sub-network $g(x)$ was specialised to be able to detect features of images. If the sub-network is able to 'see' the images sent to it, then the chance that $f(x)$ will be able to detect any parity violation that is present is improved. If constructed correctly, $g(x)$ may be able to find and generate asymmetry in $\{f(x_i)\}$.

Therefore, $g(x)$ was chosen to be an image classification neural network inspired by [8] and [9], with a key difference being that it outputted only one value, rather than a vector with weights predicting the class of the image. The network is copied below.

```
Net(
    (conv1): Conv2d(3, 16, kernel_size=(5, 5), stride=(1, 1))
    (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
                                            =False)
    (conv2): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1))
    (fc1): Linear(in_features=5408, out_features=120, bias=True)
    (fc2): Linear(in_features=120, out_features=84, bias=True)
    (fc3): Linear(in_features=84, out_features=1, bias=True)
)
```

The snail images used were 64 pixels by 64 pixels, compared to the 28x28 MNIST images of Lester and Tombs. It was vital to increase the image quality fed into the network because, at a resolution of 28x28, it was very hard to identify any objects in the images. A square kernel with size (5,5) was used because snails are circular in shape, and so a square kernel is best to detect them in the images. Two convolutional layers were separated by a max-pooling layer, which created a downsampled feature map. The convolutional layers raised the number of features from 3 (the RGB input) to 16, and then to 32. The final three layers were all fully connected layers, which brought the number of features down from 5408 to 120 to 84 and then to a single feature, which was fed into the calculation for $f(x)$.

The network $g(x)$ was trained in the same way as in Lester and Tombs. A batch size of $B_S = 64$ images was used, and these were fed into $g(x)$. A mean net score μ_B was calculated for each batch:

$$\mu_B = \frac{1}{B_S} \sum_{i=1}^{B_S} f(x_i) \tag{2}$$

The function $g(x)$ was trained to maximise μ_B/σ_B, where σ_B is the Possion standard deviation of the $f(x_i)$ in the batch, by setting the loss function to $-\mu_B/\sigma_B$. In this way, the network was rewarded for finding asymmetry in the dataset, by increasing the number of the $f(x_i)$ that were positive. We maximised μ_B/σ_B rather than μ_B so that the network could not reduce the loss function by simply multiplying the $f(x_i)$ by a constant.

2.4 Snails as a Parity-Violating Population

To test our machine learning algorithm, we required a source of parity violation. As previously discussed, for parity symmetry to be broken, we need a population with more examples of one handedness than the other. Hands themselves are a good source of handedness, but in a large dataset, we might expect there to be similar numbers of right and left hands. Snail shells also have an intrinsic handedness to them, as shown in Figure 2. Due to this handedness, snails are an excellent toy model to study parity violation, and they form the basis of this project.

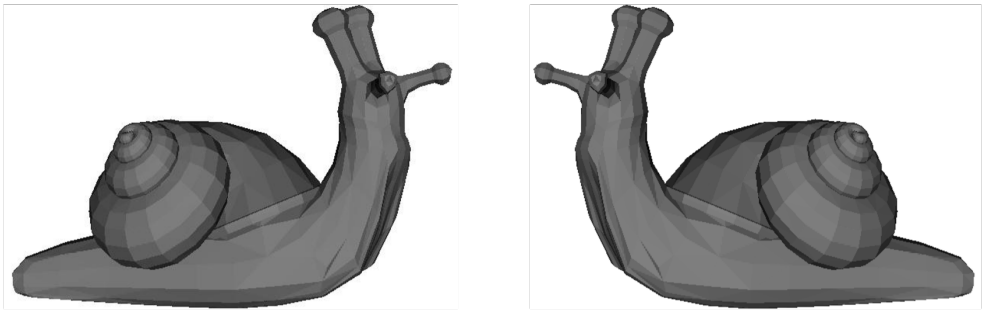


Figure 2: Snail model, showing the handedness of the shells. The snail on the left has a right-handed shell, and the snail on the right has a left-handed shell. To determine the handedness of the shell, face the spire (the 'pointy' part of the shell), with the opening to the snail's body pointing downwards. The opening will be to the left or right of the spiral: this is the handedness of the shell [10].

If we have a dataset with more snails of one handedness than the other, the dataset is parity-violating. In nature, a species of snails will almost always have one handedness [11]. Within these species there may be some individuals, called 'Snail Kings', whose shells have the opposite handedness to the vast majority of the snails in the species, caused by genetic

mutations or growth defects. Most snail species are right-handed (dextral), including the garden snail, *Cornu Aspersum*, of which about 1 in 40,000 are left-handed (sinistral). Therefore, it was expected that a dataset of snails would be parity-violating and that there would be a disparity in the number of dextral and sinistral snails in all the datasets used.

3 Dataset Curation

3.1 Living Snails Datasets

To train a machine learning algorithm to detect the parity violation of a population of snail shells, a large sample of snail images was needed. iNaturalist is a nature observation-sharing social network, with large numbers of images of many species available for use in research. Two image sets were downloaded from iNaturalist – 222,000 images of Stylommatophora [12], and 138,000 images of Helicoidea [13]. Stylommatophora is a taxonomic order which includes most land snails and slugs [14]. Helicoidea is a taxonomic superfamily within the Stylommatophora order which is predominately made up of land snails. As all the images were taken from iNaturalist, all the Helicoidea images were included in the Stylommatophora images. Comparing the results of these groups would indicate whether the network was successfully attributing parity violation to the snails, or the environment around them. Example unfiltered images are shown in Figure 3.

(a) Stylommatophora

(b) Helicoidea

Figure 3: Batches of 64 Stylommatophora and Helicoidea images. Figure 3a includes both slugs and snails, whereas Figure 3b contains only snails. These are unfiltered images taken directly from iNaturalist, and many hands can be seen in the images. The hands were removed using a parity symmetric object detection algorithm. Figure 3b forms the unfiltered first dataset, L1.

Four datasets were created out of the Helicoidea images, with a further dataset created out of the Stylommatophora images, labelled in Table 1.

Table 1: Living Snails Datasets

Dataset	Description
L1	Unfiltered Helicoidea images directly from iNaturalist
L2	Helicoidea with images with hands removed
L3	The images with hands which were removed from the Helicoidea images
L4	Greyscale Helicoidea images with images with hands removed
L5	Stylommatophora images with images with hands removed

Each of these datasets had a purpose for the objectives of the project. L1 acted as a control, against which we could compare the level of parity violation detected. L2 aimed to have snails as the only source of parity violation by removing images with hands, which are a strong source of handedness. L3 was used to test the level of parity violation in the snail images with hands in them. It was expected that the majority of hands would be left hands, as people may be inclined to hold their camera in their more commonly dominant right hand, whilst holding a snail in their left. The last Helicoidea dataset, L4, was chosen to determine whether increasing the number of channels of data changed the level of parity

violation that could be detected. It was expected that the level of parity violation would increase with the number of channels: there may be some dextral snail species that looks identical to a sinistral species when the images are in black and white, but that have different colours when the colour channels are added. In this case, a dataset made up of 50% of each species would be parity symmetric in greyscale, but maximally parity-violating in colour. Finally, the Stylommatophora dataset, L5, was created to compare to L2 – testing whether diluting the number of images of parity-violating snails with images of assumed parity symmetric slugs reduced the parity violation detected.

No extra preparation steps were required to curate L1. For L2, L4 and L5, a parity symmetric supervised machine learning algorithm was used to remove the hands. It was imperative that the 'hand remover' was parity symmetric so that no parity violation was introduced during the filtering process. The aim was to remove as many of the images with hands as possible, but it was not an issue if the hand remover wrongly removed a hand-less image, as this would not add additional parity violation into the datasets.

To train the hand remover, the ResNet-18 model [15] was implemented from scratch and trained using two hand-containing datasets, the '100 Days of Hands' dataset [16] [17] and the '11k Hands' dataset [18], and a hand-free dataset, the 'Caltech 256' dataset [19]. The Caltech 256 dataset was used to ensure that the hand remover was able to generalise well to the snail images. 100 Days of Hands contains images of hands interacting with objects, which made it a good dataset to train the hand remover on. The hand remover was parity-symmetrised at the point of implementation. Each snail image was tested for the presence of a hand at the same time as its horizontally flipped mirror image and, if either tested 'positive' for a hand, the image was flagged and removed from the dataset. The significance level of the model was tested until a suitable cutoff was reached. Both the Stylommatophora and Helicoidea image sets were passed through the hand remover.

A sample count suggested that prior to the hand removal process, approximately 9.8% of the Helicoidea, and 9.1% of the Stylommatophora dataset contained hands. 18832 hand images were removed from the Stylommatophora dataset (8.5% of the total), and 12235 from the Helicoidea dataset (8.9% of the total). Over 90% of the images detected as hands did contain hands, so it was expected that fewer than 2% of the remaining images contained hands. 64 images that were flagged as containing hands from each dataset are shown in Figure 4. Figure 4b corresponds to L3. With hands removed, L2 and L5 were created, as shown in Figure 5.

(a) Stylommatophora

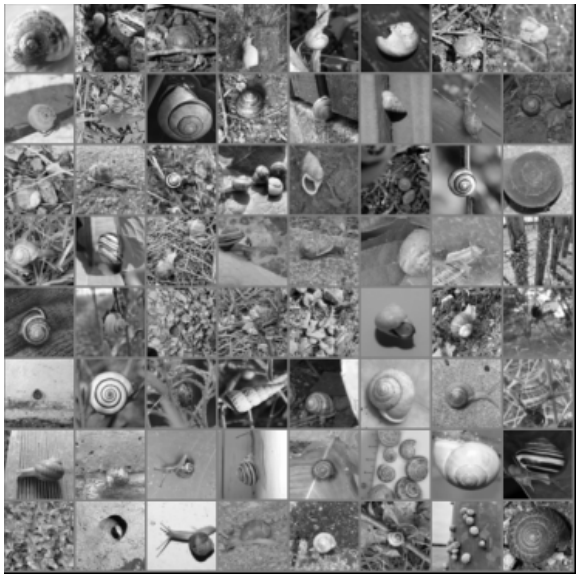
(b) Helicoidea

Figure 4: Batches of 64 Stylommatophora and Helicoidea images that were flagged as hand-containing by the symmetrised hand remover. These images demonstrate the effectiveness of the hand remover, as almost all the images contain hands. Figure 4b forms L3.

L4 was created from L2 after it had been split into test and training datasets, to be discussed in Section 4. This meant that the level of parity violation could be compared directly, as the images trained and tested on were identical except for their colour. To convert the RGB images into greyscale, the dot product of the RGB values of each pixel, `(R,G,B)`, was taken with `(0.2989,0.5870,0.1140)`, the International Telecommunications Union standard parameter for colour to greyscale shifts [20]. This parameter arises from the sensitivity of cones in the eye to different wavelengths: green appears the brightest, and blue the darkest. L4 is also shown in Figure 5.

(a) Stylommatophora with hands removed, L5

(b) Helicoidea with hands removed, L2



(c) Helicoidea with hands removed, greyscale, L4

Figure 5: Batches of 64 Stylommatophora and Helicoidea images with hands removed. No hands can be seen in these images, suggesting that the hand remover was successful.

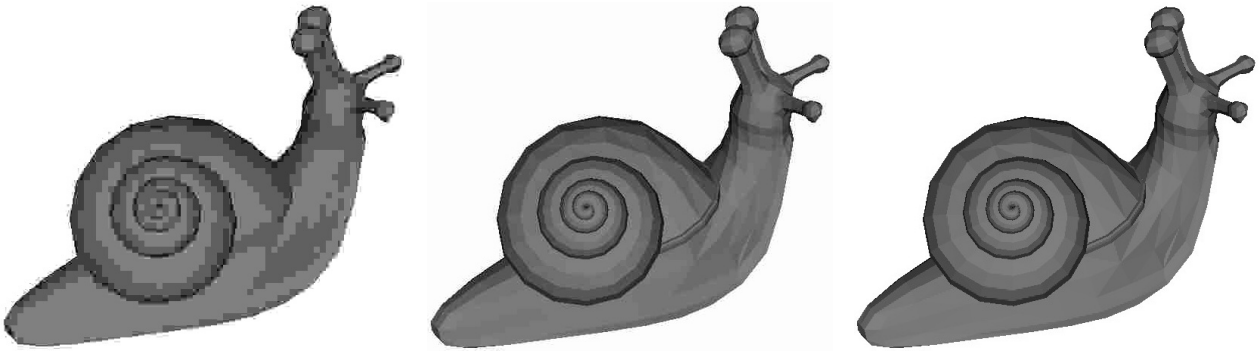
3.2 3D Snail Model Datasets

A right-handed 3D snail model [10] formed the basis of another set of datasets, visualised using PyVista [21]. This was used as a control to ensure that the network was working as designed. The model was flipped in 3D using PyVista to create the left-handed parts of the datasets. These datasets are labelled in Table 2:

Table 2: 3D Snail Model Datasets

Dataset	Description
M1	Model, greyscale, PNG: 50% right-handed, 50% left-handed
M2a	Model, greyscale, JPEG, quality 1%: 50% right-handed, 50% left-handed
M2b	Model, greyscale, JPEG, quality 20%: 50% right-handed, 50% left-handed
M2c	Model, greyscale, JPEG, quality 90%: 50% right-handed, 50% left-handed
M3	Model, coloured, PNG: 50% red right-handed , 50% blue left-handed

The results of M1 and M3 were compared to see whether increasing the number of channels of data changed the level of parity violation detected. The greyscale PNG dataset was expected to be perfectly parity symmetric, whereas the coloured PNGs were expected to be maximally parity-violating. The M2s were used to see whether lossy JPEG image compression introduced any parity violation. The quality measure of a JPEG runs from 0 to 200%. 90% is considered a high-quality JPEG, whereas 20% and 1% are low-quality JPEGs. Multiple qualities were used to test whether parity violation was introduced at different levels of compression. Right-handed snails of the three qualities are shown in Figure 6.



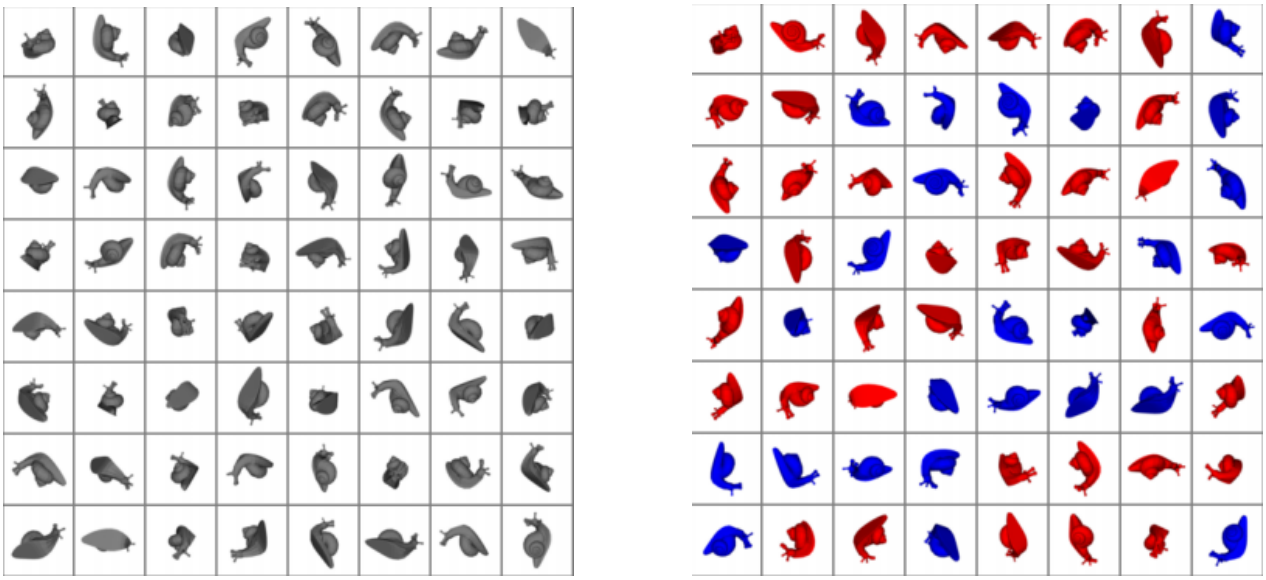
(a) JPEG Quality 1% (b) JPEG Quality 20% (c) JPEG Quality 90%

Figure 6: Screenshots of the 3D snail model at different JPEG qualities. 90% is near full quality, and no artifacts from the compression process can be seen. In the 20% image, some artifacts can be seen, and in the 1% the image is overrun by artifacts. The compression levels in Figures 6a, 6b and 6c correspond to datasets M2a, M2b and M2c respectively.

Three degrees of freedom were considered to ensure that the datasets were made up of screenshots of isotropically random orientations. The snail model was placed at the origin, and the PyVista camera was placed at a random position on a unit sphere, pointing towards the origin. This accounted for two degrees of freedom, the polar angles ϕ and θ. Then, the camera was rotated about its position vector by a random angle so that the square screenshot taken framed the snail at different rotations. This resulted in isotropically-captured images of the snails. For the greyscale images, this process was undertaken with the right-handed snail model, as well as with the model flipped in the $x - z$ plane, which

acted as a parity transformation, creating a left-handed snail model. This formed a set of greyscale PNG images which were 50% right-handed and 50% left-handed; M1. These images were compressed into JPEGs of qualities 90%, 20% and 1% to create the M2s.

Separately, the right-handed model was coloured red, and screenshots were taken as above, followed by flipping and recolouring blue to create M3. Batches of M1 and M3 are shown in Figure 7.



(a) Greyscale 3D Snail Model Dataset, M1

(b) Coloured 3D Snail Model Dataset, M3

Figure 7: Batches of 64 greyscale and coloured 3D snail models. In Figure 7a, all the images are greyscale, and so we expected M1 to be parity symmetric. In Figure 7a, however, the left-handed snails were coloured blue, and the right-handed snail red. This meant that we expected the network to detect maximal parity violation in this dataset, M3.

4 Network Implementation

The snail image datasets used were JPEGs taken from iNaturalist, and PNG and JPEG screenshots of a 3D snail model. After curation, each image dataset was split into training and test sets using an 80%-20% split, in a way that was compatible with the PyTorch data loader `ImageFolder`. Due to the source of the images (many different cameras and uploading methods), the images needed to be made uniform for the machine learning process. `ImageFolder` was used to load and transform the images using a composition of transforms. The images were converted to a PyTorch tensor using `transforms.ToTensor()`, and then normalised by the standard ImageNet [22] normalisation for RGB images, using `transforms.Normalize (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])`. Finally, each image had its shortest side resized to 64 pixels and was cropped to a 64 by 64 pixel square using `transforms.Resize(64, antialias=True)` and `transforms.CenterCrop(64)`.

This formed one of the two sets that were created from each image dataset. These normal image sets were labelled 'S', for Snail. Alongside this set, a second, achiral dataset was created by randomly flipping 50% of the images in the original test and train sets. These counterpart sets were labelled 'SLR', for 'Snail Left-Right', indicating the flip. The symmetrisation removed parity violation so that the SLR set acted as a control for the S set in each case.

The S and SLR sets were then loaded using `torch.utils.data.DataLoader()`, batch size 64 for the training sets, and batch size 2447 for the test sets. 2447 was chosen as it was the size of the smallest test set, the hands test set (L3). It meant that individual batches of testing could be compared against each other if required. For all the other datasets, the test set was larger than 2447 images, so the testing was done in batches over the whole test set.

A network $g(x)$ was trained for each dataset separately. Training different networks for each dataset allows us to compare the level of parity violation in the test set, as a network trained on similar images is in the best state to detect parity violation. The weights and biases of each network (an S and an SLR for each dataset) were saved after training each for 15 epochs. 15 epochs were chosen as the loss converged before this for all the datasets.

The test datasets were passed through these trained networks. As an additional test, L2 was tested with the S and SLR networks interchanged to create an extra set of results, L6. This tested whether a network trained on an achiral dataset could still detect parity violation in a chiral one, and vice versa.

The proportion of images attributed a positive value in each test set was recorded along with their Poisson standard deviation. These results are published in Section 5.

5 Results

5.1 Living Snails Datasets

The living snails datasets are repeated for ease:

Table 3: Living Snails Datasets

Dataset	Description
L1	Unfiltered Helicoidea images directly from iNaturalist
L2	Helicoidea with images with hands removed
L3	The images with hands which were removed from the Helicoidea images
L4	Greyscale Helicoidea images with images with hands removed
L5	Stylommatophora images with images with hands removed
L6	L2 with S and SLR trained networks interchanged

The results for all the datasets are printed below. The errors published are the Poisson standard deviation for a process with a probability equal to the positive fraction of a dataset, and a sample size equal to the test set size. Proportions lower than 50% are due to the test and training set not matching perfectly, and statistical fluctuations in the testing process.

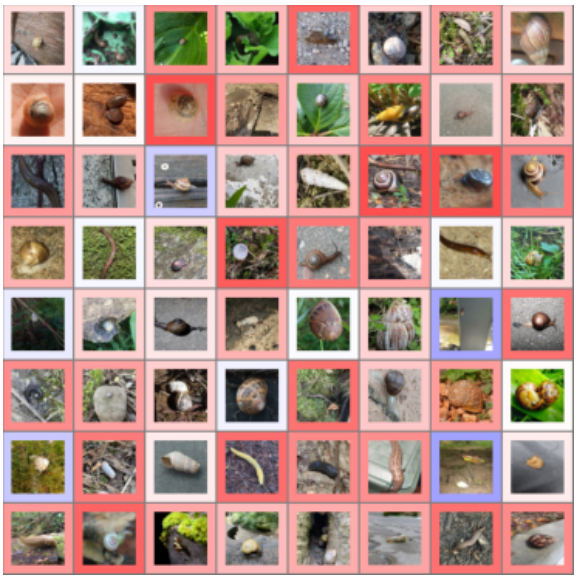
Table 4: The positive fraction of images per batch given a positive value of $f(x)$ after training. Parity-violating datasets are printed in **bold**. All the S datasets are parity-violating, having fractions much larger than 50%, whereas all the SLR datasets are not. All the SLR datasets have values consistent with 50%, suggesting achirality as expected. Even L6, where the networks were not trained on the correct dataset shows (albeit small) parity violation in the S dataset.

Living Snails Datasets

Dataset	Test Set Size	S Positive Fraction/%	SLR Positive Fraction/%
L1	26917	**70.7 ± 0.3**	49.9 ± 0.3
L2	24470	**70.7 ± 0.3**	49.7 ± 0.3
L3	2447	**61.7 ± 1.0**	49.4 ± 1.0
L4	24470	**70.0 ± 0.3**	50.1 ± 0.3
L5	39152	**58.8 ± 0.2**	49.9 ± 0.3
L6	24470	**50.8 ± 0.3**	50.0 ± 0.3

Some example batches of test data were also printed with coloured borders, ranging from blue to red, via white. Strong blue reflects a large negative network value, whilst strong red reflects a large positive network value.

(a) Helicoidea Hands Removed, L2



(b) Stylommatophora Hands Removed, L5

Figure 8: Batches of 64 Helicoidea and Stylommatophora images with hands removed (L2 and L5). Here, a coloured border has been added to highlight the network output for each image in the batch. A network output of 0 is coloured white, increasingly stronger red denotes increasingly more positive values and increasingly stronger blue denotes negative values. Both sub-figures have mostly red images, as expected, but some are coloured blue, showing that the network could not attribute them with a positive value. More blue is seen in the Stylommatophora images.

5.2 3D Snail Model Datasets

The 3D snail model datasets are repeated for ease, with their results and a bordered batch of M1 and M3:

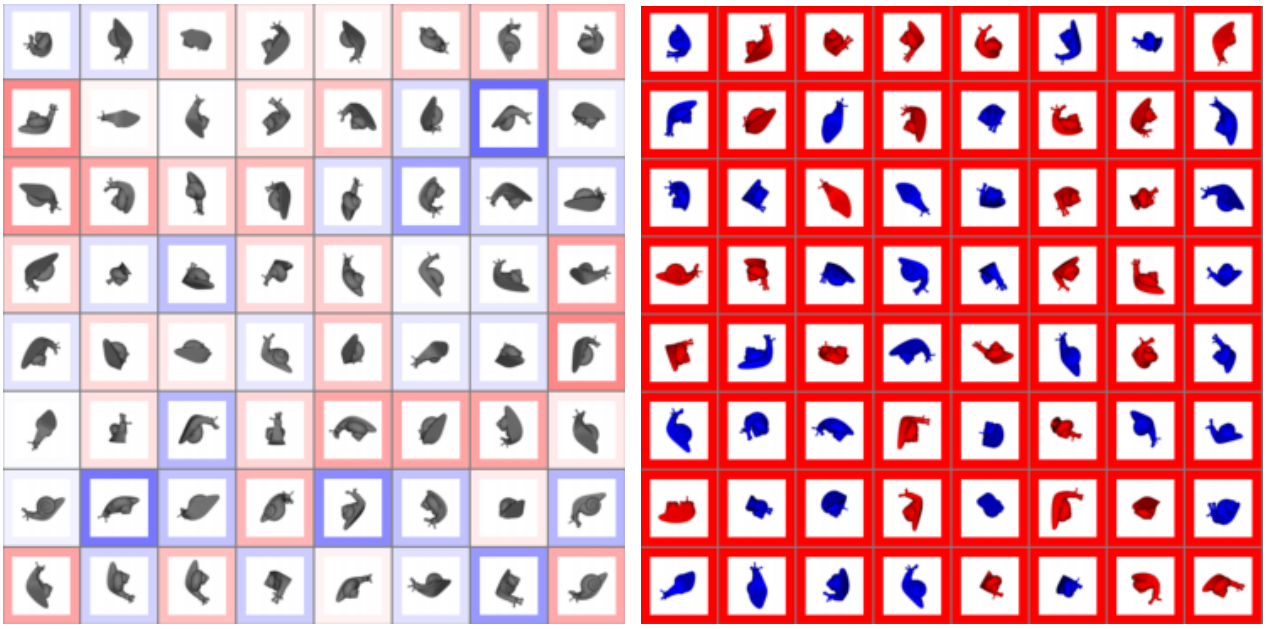
Table 5: 3D Snail Model Datasets

Dataset	Description
M1	Model, greyscale, PNG: 50% right-handed, 50% left-handed
M2a	Model, greyscale, JPEG, quality 1%: 50% right-handed, 50% left-handed
M2b	Model, greyscale, JPEG, quality 20%: 50% right-handed, 50% left-handed
M2c	Model, greyscale, JPEG, quality 90%: 50% right-handed, 50% left-handed
M3	Model, coloured, PNG: 50% red right-handed , 50% blue left-handed

Table 6: The positive fraction of images per batch given a positive value of $f(x)$ after training. Parity-violating datasets are printed in **bold**. The only parity-violating dataset was the coloured model S set, M3, which was maximally parity-violating as expected. No level of JPEG compression introduced any statistically significant parity violation, and all the SLR sets were achiral, as expected.

3D Snail Model Datasets

Dataset	Test Set Size	S Positive Fraction/%	SLR Positive Fraction/%
M1	4894	49.3 ± 0.7	49.3 ± 0.7
M2a	4894	50.3 ± 0.7	49.7 ± 0.7
M2b	4894	50.7 ± 0.7	50.0 ± 0.7
M2c	4894	49.3 ± 0.7	49.5 ± 0.7
M3	4894	**99.63 ± 0.09**	48.7 ± 0.7



(a) Greyscale 3D Model PNG, M1

(b) Coloured 3D Model PNG, M3

Figure 9: Batches of 64 greyscale and coloured 3D model PNGs (M1 and M3). The network struggles to assign positive values to the parity symmetric greyscale images, and so the batch contains an equal number of blue and red borders. The coloured images break the parity symmetry maximally, so the network can assign every image with a large positive number, bordering them all red.

6 Discussion

The project was successful in achieving its aims: it is possible to detect parity violation in multichannel data using a symmetrised machine learning algorithm. All the living snails datasets were found to be parity-violating, and the 3D model datasets that were engineered to be parity symmetric were shown to be. Furthermore, it was shown that introducing extra

channels through colouring the images increased the level of parity violation that was able to be detected. This can be seen by comparing L2 (colour) and L4 (greyscale) – L4 displayed a slightly lower level of parity violation, 70.7% to 70.0% – and by comparing M1 (greyscale) and M3 (coloured). In this case, the addition of colour, as expected, created a maximally parity-violating (99.6%) dataset as compared to its greyscale sibling, which was symmetric under parity (49.3%).

It is important to note that due to its symmetrisation, the network cannot assign values to the images in M3 based solely on the snails' colour. If there is an image of a right-handed red snail, labelled R, then the reflection of that image, L, is also red. By construction, the network $f(x)$ is parity-odd, and so $f(R) = -f(L)$. It is therefore not possible for the network to evaluate both to the same positive value. Instead, the network works at a deeper level: it is able to give a red, right-handed snail, and a blue, left-handed snail positive values because it has noticed that the colour makes them different. It is not able to do this in greyscale as there is no colour factor to differentiate them.

We can also see that diluting the parity-violating snails of L2 with previously assumed parity symmetric slugs in L5 reduced the level of parity violation from 70.7% to 58.8%. This suggests that the network is able to recognise the handedness of the snails, and not just the environment around them, which would be the same in the slug images.

The hands dataset (L3) showed a lower level of parity violation than expected – perhaps there were a more similar number of left and right hands in the images than expected, or the network struggled to detect the hands. Removing the hands (L2) from the images did not reduce the level of parity violation detected, suggesting that removing the hands did not dilute or concentrate the level of parity violation.

Furthermore, the results show that no level of lossy compression used to create JPEGs out of PNG files introduced any parity violation that could be detected by the network.

The network is unlikely to be the best or the fastest at detecting parity violation, and therefore the results achieved may not record the largest fractions possible. However, the network was trained until the loss converged, so we are satisfied that there is no under-training occurring. As the same network design and parameters were used for all the datasets, they can be compared, but there is likely a better network design for this problem. However, this is not really an issue, because the choice of a parity-odd network cannot introduce extraneous parity violation that is not present in the dataset: a better network design can only detect extra sources of parity violation invisible to our current choice. The fact that the current network may be blind to these other sources does not affect our ability to compare network results between datasets.

Even with the borders to identify specific network scores, it is challenging to determine

to what the network is attributing positive values. In Figure 9a, one might expect the different handedness of the snails to be given oppositely signed values, but this not seen – both left- and right-handed snails are seen bordered in red, for instance. It is likely, however, that the network finds smaller details that it attributes different scores to (that in turn averages out to the handedness of the snails over a large sample), rather than the handedness of a snail as an individual.

7 Conclusion

This project was successfully able to detect parity violation in multi-channel data, which is an important step in the implementation of a symmetrised machine learning algorithm to detect parity violation in LHC collisions in the future. It was shown that increasing the number of channels of data increased the parity violation detected, especially in the ideal 3D snail model datasets, which acted as an excellent simplified model to compare real snail image data against. The 3D snail model gave results that were in line with expectations, suggesting that the network was able to successfully detect the different handedness of the snails. The living snails also produced expected results, including displaying less parity violation when diluted with slugs.

Further work should explore the difficulties that may be presented when moving from the toy model of snails to real energy deposit data from LHC-type colliders. In the LHC, particles travel through different detectors including a tracking layer, an electromagnetic calorimeter, a hadronic calorimeter, and a muon detector. These detectors have varied resolutions [23] [24], so collating the data from each into a multichannel system without introducing any extraneous parity violation would pose a challenge. Furthermore, in the LHC, there is a large chance that any parity violation seen is wholly due to the collider and its measurement systems, not beyond the Standard Model physics. In this way, a parity-violating signal suggests either a new discovery or that there is an issue with the detector.

References

[1] C. G. Lester and M. Schott, "Testing non-standard sources of parity violation in jets at the LHC, trialled with CMS open data," *Journal of High Energy Physics*, vol. 2019, Dec 2019.

[2] R. Casallbuoni, P. Chiappetta, A. Deandrea, A. Fiandrino, R. Gatto, G. Nardulli, and P. Taxil, "Beam polarization at LHC and SSC. Expected asymmetries in the Bess model and comparison with other models," *Physics Letters B*, vol. 279, no. 3, pp. 397–404, 1992.

[3] C. G. Lester and R. Tombs, "Using unsupervised learning to detect broken symmetries, with relevance to searches for parity violation in nature. (Previously: "Stressed GANs snag desserts"),," 2022.

[4] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[5] C. S. Wu, E. Ambler, R. W. Hayward, D. D. Hoppes, and R. P. Hudson, "Experimental test of parity conservation in beta decay," *Phys. Rev.*, vol. 105, pp. 1413–1415, Feb 1957.

[6] A. Pich, "CP Violation. Lectures given at the ICTP Summer School in High Energy Physics and Cosmology ICTP, Trieste, Italy," 1993.

[7] "The ATLAS Detector." `https://atlas.cern/discover/detector`, June 2015.

[8] B. Neo, "Building an image classification model with pytorch from scratch." `https://medium.com/bitgrit-data-science-publication/building-an-image-classification-model-with-pytorch-from-scratch-f10452073212`, Feb 2022.

[9] "Deep Learning with PyTorch." `https://pytorch.org/`.

[10] BS3, "[FREE] snail." `https://cults3d.com/en/3d-model/various/free-snail`.

[11] H. Fletcher, I. Hickey, and P. Winter, *Instant Notes in Genetics*. Taylor and Francis, June 2002.

[12] *iNaturalist community. Observations of Stylommatophora, observed before February 2023. Exported from https://www.inaturalist.org on 22/02/2023.*

[13] *iNaturalist community. Observations of Helicoid Land Snails, observed before February 2023. Exported from https://www.inaturalist.org on 22/02/2023.*

[14] P. Bouchet, J.-P. Rocroi, B. Hausdorf, A. Kaim, Y. Kano, A. Nützel, P. Parkhaev, M. Schrödl, and E. E. Strong, "Revised Classification, Nomenclator and Typification of Gastropod and Monoplacophoran Families," *Malacologia*, vol. 61, no. 1-2, pp. 1 – 526, 2017.

[15] "ResNet-18." `https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html`.

[16] University of Michigan, "100DOH." `https://fouheylab.eecs.umich.edu/~dandans/projects/100DOH/100DOH.html`.

[17] D. Shan, J. Geng, M. Shu, and D. F. Fouhey, "Understanding human hands in contact at internet scale," *CoRR*, vol. abs/2006.06669, 2020.

[18] M. Afifi, "11K Hands: gender recognition and biometric identification using a large dataset of hand images," *Multimedia Tools and Applications*, 2019.

[19] G. Griffin, A. Holub, and P. Perona, "Caltech 256," April 2022.

[20] "BT.601 : Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios." `https://www.itu.int/rec/R-REC-BT.601-7-201103-I/en`, March 2011.

[21] C. B. Sullivan and A. A. Kaszynski, "PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK)," *Journal of Open Source Software*, vol. 4, no. 37, p. 1450, 2019.

[22] ImageNet. `https://www.image-net.org/`, March 2021.

[23] T. Lari, "Measurements of spatial resolution of atlas pixel detectors," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 465, p. 112–114, Jun 2001.

[24] J. Mašik, "Vertex and track reconstruction in ATLAS and CMS," *PoS*, vol. Beauty2013, p. 041, 2013.

Appendix - Jupyter Notebook

```
# %%
#Import matplotlib and PyTorch

import matplotlib.pyplot as plt
import numpy as np

import os

import torch
import torch.nn as nn
import torch.nn.functional as F

import torchvision
from torchvision import datasets, transforms
from torchvision.datasets import ImageFolder

mps_device = torch.device("mps")

# %% [markdown]
# In a 4D pytorch tensor, the 4 dimensions (labelled 0 to 3) are:
#
# 0. Batch size
# 1. Number of channels, e.g. 1 for grayscale, 3 for RGB
# 2. Height of each image in pixels.
# 3. Width of each image in pixels.
#
# We need certain rotations and flips to be available to us, so we define
                                        functions:
#
# rotate_180(x): to be used to add invariance with respect to 180 degree
                                        rotation.
# Reverses the 3rd and 4th dimensions ([2,3]).
# x_flip(x): flips left-right. Flips the 4th dimension.
# y_flip(x): flips top-bottom. Flips the 3rd dimension.
#
# x is our tensor.

# %%
def rotate_180(x):
    return torch.flip(x,[2,3])

def x_flip(x):
    return torch.flip(x,[3])
```

```
def y_flip(x):
    return torch.flip(x,[2])

# %% [markdown]
# As we are looking to form 2 groups of test/train datasets (non-modified
                                        chiral and
# 50% flipped achiral), we need to do some preprocessing of our data into these
                                        sets.
#
# These 2 groups will be named as follows (taking inspiration from Dr Lester's
# previous work):
#
# 1. S
# 2. SLR
#
# where 'S' is for snail, and 'LR' (Left-Right) labels the achiral group.
#
# We define the Group class, which will allow us reference different aspects of
# the objects we are dealing with, and to view our results in the future.

# %%
class Group:
    def __init__(self,name, train=True, trainloader=None, testloader=None, net=
                                            None):
        self.name=name
        self.train=train
        if trainloader is not None:
            self.set_trainloader(trainloader)
        if testloader is not None:
            self.set_testloader(testloader)
        if net is not None:
            self.set_net(net)

    def return_test_images(self):
        dataiter1 = iter(self.testloader)
        images, labels = next(dataiter1)
        images, labels = images.to(mps_device), labels.to(mps_device)
        return images

    def return_train_images(self):
        dataiter2 = iter(self.trainloader)
        images, labels = next(dataiter2)
        images, labels = images.to(mps_device), labels.to(mps_device)
        return images

    def set_trainloader(self, loader):
```

```
        self.trainloader = loader

    def set_testloader(self, loader):
        self.testloader = loader

    def set_net(self, net):
        self.net = net
        import torch.optim as optim
        self.optimizer = optim.SGD(self.net.parameters(), lr=0.001, momentum=0.
                                                    9)

    def parity_calculations(self, net_outputs, test, output):
        from math import sqrt
        num_positives=torch.count_nonzero(torch.gt(net_outputs,0).int()).item()
        num_negatives=torch.count_nonzero(torch.lt(net_outputs,0).int()).item()
        num_zeros=torch.count_nonzero(torch.eq(net_outputs,0).int()).item()
        num_total=num_positives+num_zeros+num_negatives
        poisson_p=num_positives/num_total
        poisson_q=1.0-poisson_p
        poisson_mean = num_total*poisson_p
        poisson_variance = num_total*poisson_p*poisson_q
        poisson_sd=sqrt(poisson_variance)

        fractional_mean = poisson_mean/num_total
        fractional_sd = poisson_sd/num_total

        if output == True:
            print("On",("test" if test else "train"),": Positive Fraction ",100
                                                    *fractional_mean,"+-",100*
                                                    fractional_sd,"%")
        elif output == False:
            print(100*fractional_mean,100*fractional_sd)


    def output_values(self, output, test):
        if test == True:
            self.parity_calculations(self.net(self.return_test_images()), test,
                                                     output)
        elif test == False:
            self.parity_calculations(self.net(self.return_train_images()), test
                                                    , output)



# %% [markdown]
# Create the test and train datasets:
```

```
#
# Generate a tensor from the image input. We will take the image input and
# put it through 4 transforms sequentially using transforms.Compose()
#
# transforms.ToTensor(): converts the image into 3 RGB colour channels (or 1
# greyscale channel)
#
# transforms.Normalise(mean, std): normalises the tensor according to mean and
                                        s.d. parameters
#
# transforms.Resize(x): resizes the image such that the shortest side is now x
                                        pixels
#
# transforms.CentreCrop(x): crops the image in a square at the centre with side
                                         length x

# %%
#0.5s

#normal:
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize(
                                        mean=[0.485, 0.456, 0.406],  std=[0.229
                                        , 0.224, 0.225]), transforms.Resize(64,
                                         antialias=True), transforms.CenterCrop
                                        (64)])

#for greyscale
#transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize(
                                        mean=[0.5, 0.5, 0.5],  std=[0.5, 0.5, 0
                                        .5]), transforms.Resize(64, antialias=
                                        True), transforms.CenterCrop(64)])


folder_name = "FOLDER NAME"

test_path = "PATH" + folder_name + "/test"
train_path = "PATH" + folder_name + "/train"

testset = ImageFolder(root=test_path, transform=transform)
print(f"Number of test images: {len(testset)}")

trainset = ImageFolder(root=train_path, transform=transform)
print(f"Number of training images: {len(trainset)}")


# %% [markdown]
```

```
# Create the SLR dataset: flip each image in the S dataset in turn

# %%
#4 mins for 125,000 images (17s per 10000)

def flip_dataset(dataset):
    flip_transform = transforms.RandomHorizontalFlip(p=1.0)
    flipped_dataset = []
    count = 0

    for image, label in dataset:
        if np.random.binomial(1, 0.5):
            image = flip_transform(image)

        flipped_dataset.append((image, label))
        count +=1
        if count%1000 == 0:
            print(count, "Done!")



    return flipped_dataset

testset_SLR = flip_dataset(testset)
trainset_SLR = flip_dataset(trainset)

# %% [markdown]
# Load the data using a dataloader. Create the batches for the test and train
                                        sets

# %%
batch_size=64
factor = len(testset)/2447

print("Factor is ", factor)

trainloader_S = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                        shuffle=True, pin_memory=True,
                                        num_workers=4)
testloader_S = torch.utils.data.DataLoader(testset, batch_size=int(len(testset)
                                        /factor), shuffle=True, pin_memory=True
                                        , num_workers=4)

trainloader_SLR = torch.utils.data.DataLoader(trainset_SLR, batch_size=
                                        batch_size, shuffle=True, pin_memory=
                                        True, num_workers=4)
```

```
testloader_SLR = torch.utils.data.DataLoader(testset_SLR, batch_size=int(len(
                                        testset)/factor), shuffle=True,
                                        pin_memory=True, num_workers=4)

print(trainloader_S.batch_size,testloader_S.batch_size)

# %% [markdown]
# Define the groups

# %%
groups = {
    "S" : Group("S", train=False, trainloader=trainloader_S, testloader=
                                            testloader_S),
    "SLR" : Group("SLR", train=False, trainloader=trainloader_SLR, testloader=
                                            testloader_SLR),
    }

# %%
#CHECK TRAINLOADER LENGTH

for name, group in groups.items():
    num_batches = len(group.trainloader)
    batch_size = group.trainloader.batch_size
    num_images = num_batches * batch_size
    print(f"Number of images in {name}: {num_images}")

# %% [markdown]
# Create an image viewer to check whether the trainsets look as expected

# %%
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

#PLOT IMAGES - 25s
for name, group in groups.items():
    dataiter = iter(group.trainloader)
    print(name, "(train)")
    images, labels = next(dataiter)
    imshow(torchvision.utils.make_grid(images))

# %% [markdown]
# Define the image classification net - 2 conv layers separated by a maxpool
                                        layer, followed by 3 fully connected
```

```
                                        layers
#
# Move the net to the GPU

# %%
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.tensor_size=torch.tensor([64,64]) #IMAGE SIZE

        self.kernel_1_size = (5,5)
        self.kernel_2_size = (5,5)

        self.conv1 = nn.Conv2d(3, 16, self.kernel_1_size,1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, self.kernel_1_size,1)
        self.fc1 = nn.Linear(32 * 13 * 13, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 1)

    def forward(self, x):
        answer = self.forward_worker(x)
        answer = answer + self.forward_worker(rotate_180(x))
        answer = answer - self.forward_worker(y_flip(x))
        answer = answer - self.forward_worker(x_flip(x))
        return answer/4.0

    def forward_worker(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))

        #Flatten x along "channel" dimension - we will create a 2D tensor with
                                                dimensions
        #(batch size, number of channels * height * width)
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        x = torch.sum(x, dim=1)
        return x

model = Net().to(mps_device)
print(model)

for group in groups.values():
```

```
    group.set_net(model)



# %% [markdown]
# Define a function that outputs positive fraction values for a group

# %%
def positive_fraction(groupname, message, attempts=1, output = True, test =
                                        True):
    group = groups[groupname]
    for _ in range(attempts):
        print(message,group.name)
        group.output_values(output,test)

        print()

#before training - 14s
#positive_fraction("S","Before Training",1, output = True, test = True)

# %% [markdown]
# Define the group-wise training loop

# %%
def train_group(group, no_of_epochs):

    print('Start Training',group.name)
    #Output current performance
    #group.output_values()
    print('\n')
    for epoch in range(no_of_epochs):
        #Set loss to 0
        running_loss = 0.0
        for i, data in enumerate(group.trainloader, 0):
            #Import data
            inputs, labels = data
            inputs, labels = inputs.to(mps_device), labels.to(mps_device)

            #Zero parameter gradients
            group.optimizer.zero_grad()

            #Calculate std and mean of the outputs of the net
            std = torch.std(group.net(inputs))
            mean = torch.mean(group.net(inputs))

            #we want to MAXIMISE mean/std, so we want to MINIMISE -mean/std
```

```
            loss = -mean/std

            #enforce the minimisation of the loss, and update the parameters
            #based on the gradients computed in backpropagation (using SGD)
            loss.backward()
            group.optimizer.step()


            # print statistics
            running_loss += (-loss).item()
            print_freq=100
            if i % print_freq == print_freq-1:
                print('[Epoch %d, Batch %d] loss: %.3f' %(epoch + 1, i + 1,
                                                          running_loss/print_freq
                                                           ))
                running_loss = 0.0
                #group.output_values()

        # model_dir = 'MODEL DIRECTORY'
        # epoch_plus_1 = str(1 + epoch)
        # run_identifier = "_" + epoch_plus_1 + "E_BS16_NH_1"
        # PATH = model_dir+group.name+run_identifier+'.pth'
        # torch.save(group.net.state_dict(),PATH)
        #group.output_values(output = True, test = True)
        print("Epoch ", epoch+1, " Done!" )
        print('\n')
    print('Finished Training',group.name)
    #group.output_values(output = False, test = True)
    print('\n')

# %% [markdown]
# Define train, save and load functions for the net weights and biases

# %%
def train_and_save(extension, no_of_epochs):

    for name, group in groups.items():

        train_group(group, no_of_epochs)

        folder_name = "FOLDER NAME"
        model_dir = "/PATH" + folder_name + "/"
        PATH = model_dir+name+extension+'.pth'

        torch.save(group.net.state_dict(),PATH)
```

```
def load(groupname, extension):
    folder_name = "FOLDER NAME"
    model_dir = "PATH" + folder_name + "/"

    group = groups[groupname]
    name = group.name
    NAME = name + extension + '.pth'
    group.net.load_state_dict(torch.load(model_dir+NAME))


# %% [markdown]
# Ensure the GPU is available and connected

# %%
print(torch.has_mps)
print(torch.backends.mps.is_available())
print(torch.backends.mps.is_built())

# %% [markdown]
# Train and save the weights and biases file locally

# %%
#TRAIN - takes 2 minutes per dataset (S and SLR) per epoch, running through 100
                                        ,000 images
train_and_save("EXTENSION", 15)

# %% [markdown]
# Final data for each batch: output positive fraction values

# %%
epoch_no = 15

for groupname in ["S","SLR"]:
    print("Batch Size is", testloader_S.batch_size)
    load(groupname,"_END"+str(epoch_no)+"EXTENSION")
    positive_fraction(groupname, "After training for " + str(epoch_no) + "
                                            Epochs:", attempts = int(factor),
                                            output = False, test = True)
```