DEPARTMENT OF COMPUTER SCIENCE

Deep Behavioural Action Recognition for European Roe Deer in the Wild

Joel Grimmer

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Tuesday 10th October, 2023

Abstract

This project introduces a deep learning behaviour recognition model designed specifically for Roe deer, with the aim of detecting a set of basic behaviours. Whilst steps have been taken in the research of automated behaviour recognition models for non-human species, notably great apes [37] [37], such a study has not been conducted on deer populations. Thus, performing such a study presents a novel species for behaviour recognition. These behaviours include standing, grooming, and a set of motion behaviours based on the direction of motion. It is common to find multiple deer in one video clip, so the model has been designed to identify behaviours for a single Roe deer from a video clip possibly containing multiple deer displaying many different behaviours. More than 50,000 frames from the Brandenburg video archive, developed in [17], have been annotated with Roe deer behaviour information, constituting a significant contribution to a preexisting video dataset.

The development of the model involved contrasting the performance of 3D Resnet-18 and Multi-scale Vision Transformer (MViT) networks on the behaviour dataset. The final model incorporates two 3D Resnet-18 networks to learn two distinct input streams. The first stream consists of RGB video frames, and the second consists of optical flow frames, with the aim of capturing spacial and temporal information for a given behaviour. A long tail class distribution was identified within the data-set, which hindered the learning of less frequently displayed behaviour classes. A balanced weighted random sampling approach was investigated, which improved the average class accuracy of the model.

This thesis falls within the field of animal biometric systems, which aim to automate the extraction of animal biometric information from often very large visual datasets. A model such as this aims to automate the manual extraction of information that would otherwise require a significant time investment from expert ecologists.

A concise list of achievements are as follows:

- I spent approximately 38 hours annotating a subset of the Brandenburg video archive with action behaviour classes, producing over 50,000 annotated frames.
- I wrote approximately 400 lines of code in Python to automate the pre-processing of the Brandenburg video archive, involving but not limited to the copying and arranging of video files by a unique ID, generating video frames, generating optical flow frames, and bounding box generation.
- I have researched various deep learning and concepts, including but not limited to 3D convolution, attention-based mechanisms, two stream action recognition, optical flow, class imbalance, and deep learning model evaluation.
- I have written approximately 1500 lines of Python code to implement different deep learning mechanisms using the library PyTorch. This includes the implementation of various behaviour recognition models, the implementation of dataset logic, the implementation of model training, and the implementation of model evaluation.
- I have carried out various experiments to ascertain the performance of different model architectures and training approaches by collecting different metrics, and have used these values to construct a new deep learning model that is suited to the project domain.

Dedication and Acknowledgements

I would like to acknowledge the support provided by my supervisor Dr. Tilo Burghardt throughout this project. His expert knowledge and guidance helped focus my efforts and motivated me to produce a piece of work of which I feel truly proud. His passion both for computer science and for the natural world is admirable and I have been left inspired by the potential of work in the field of animal biometrics to drive real change in our approaches to the environment.

I would also like to acknowledge the support provided by my friends and family, which has proven invaluable during difficult times in this project. This project has required a significant time investment and I appreciate the effort taken by friends and family to reach out and lend an ear when I was stuck on a particularly difficult challenge.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Joel Grimmer, Tuesday 10^{th} October, 2023

Contents

List of Figures

List of Tables

Ethics Statement

A compulsory section

This project did not require ethical review, as determined by my supervisor, Dr Tilo Burghardt

Supporting Technologies

A number of libraries and technologies were required for the successful completion of the project. These are listed below, and will be referenced throughout the thesis.

- I used the PyTorch [35] deep learning library throughout the project to facilitate the implementation of key deep learning operations, including the implementation of deep learning models, dataset logic, model training, and model evaluation.
- I the OpenCV [33] computer vision library to perform pre-processing operations on the dataset, including as splitting a video into frames and loading video frames for image processing.
- I used the PIL [34] image processing library to load video frames during model training and evaluation and to crop video frames to a bounding box.
- I used The University of Bristol's supercomputer Blue Crystal Phase 4 (BC4) [32]. BC4 provides access to Nvidia P100 GPUs and up to 100GB of RAM. This allowed for training and evaluation to take advantage of CUDA [31] acceleration. BC4 also hosted the dataset, which is larger than 2TB, saving valuable disk space on my local development machine.
- I used TensorBoard [42] to track model performance during training and evaluation. The logs helped me to collect performance metrics for different model architectures and training strategies, and TensorBoard's scalar functionality helped to visualise model performance during training.
- I used Git [15] for for version control throughout the project. I hosted the project code-base on GitHub [16].

Chapter 1

Introduction

1.1 Action Recognition

Action recognition is the task of developing automated methods to recognise and classify actions in visual data to a discrete, predefined set of action classes [18]. Action recognition is most commonly applied to human subjects, however there is a growing body of research into recognising actions taken by non-human subjects [22][29][12]. Such research has great potential in a variety of applications, including conservation work [19], veterinary research [26], and agriculture [14].

The field of computer vision has existed since the late 1960s, and historically has been approached from two distinct perspectives. A biological approach to computer vision is concerned with developing a computational model of human visual capabilities while an engineering perspective aims to automate tasks normally undertaken by the human visual system through the extraction of relevant features from visual information [20].

Action recognition takes an engineering perspective to computer vision by attempting to automate the annotation of behaviours from visual information. A feature, in the context of computer vision, is information derived from an image or video. The task of feature detection has been central to computer vision tasks since its inception. An early example is the Sobel-Feldman operator, which uses a predefined kernel to detect edge features within an image [41]. The development of increasingly powerful Graphics Processing Units (GPUs) has increased the viability of training complex feature detection models on large sets of image and video data. Convolutional Neural Networks (CNNs), a type of deep learning architecture, aim to learn kernels that can then extract relevant features, most commonly from image data, for machine learning tasks. CNNs allow for the automated generation of complex, hierarchical kernels, which can generalise complex visual stimuli. The 2012 ImageNet Large Scale Visual Recognition Challenge represented the greatest step forward taken in Deep Learning for Computer Vision, where the winning model AlexNet achieved unprecedented accuracy and loss rates within the field [1]. Since then Deep Learning has seen rapid performance increases as big tech corporations in partnership with academic institutions have poured financial resources into the research of Deep Learning for Computer Vision and the compute power necessary for their training.

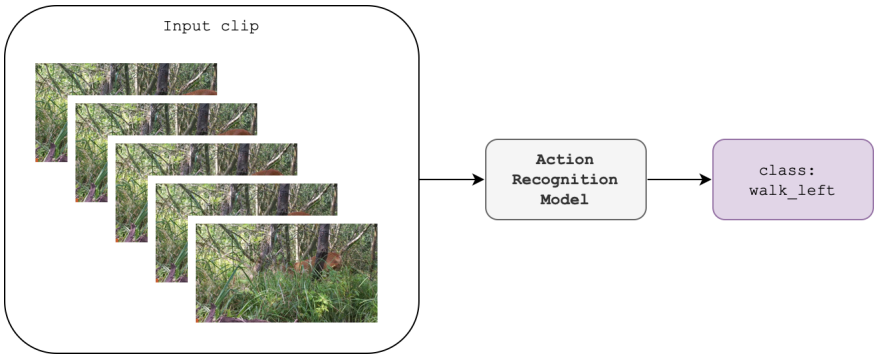

Figure 1.1: An action recognition task aims to produce a model that can accurately classify actions based on a stream on input frames.

Existing video recognition research is largely driven by improvements in image recognition models. The video recognition model 3D Resnet-18 [43] is based on the pre-existing Resnet-18 network, expanding the two-dimensional convolutional layers found within the original model by adding an additional temporal dimension in order to produce features which encode spatiotemporal data. However, the increased dimensionality of temporal information can result in high levels of information redundancy and the need to model motion dynamics. A number of solutions to these issues have been investigated [48], including processing optical flow data streams [39], which encode motion within a single frame, and the use of Video Transformers [2] which evolve input representations based on interactions along a sequence of frames.

1.2 Motivations

The European Roe Deer (Capreolus capreolus) are a species of deer. They range throughout Asia minor to Europe. They are primarily found in dense woodland, meaning their detection is often impeded by environmental occlusion [7]. There are no existing studies relating to action recognition of Roe Deer, so such an undertaking is novel. Their long limbs and expressive head movement present an opportunity for investigation into how Deep Neural Networks classify behaviour of non-bipedal species.

1.2.1 Animal Biometrics

Animal Biometrics is the study of quantified approaches to the representation of the phenotype appearance of species, individuals, behaviours, and morphological traits [23]. Animal Biometrics has benefited from advances in machine learning technology that allow for the automation of biometric labelling from visual data that would normally require time and effort from a large number of ecologists [24]. Generalised detection models, such as MegaDetector, developed by Microsoft's Ecology for AI department, allow for the automated generation of bounding boxes for humans, animals, and vehicles [46]. Behaviour detection can automate time intensive labelling that would normally be undertaken by experienced ecologists.

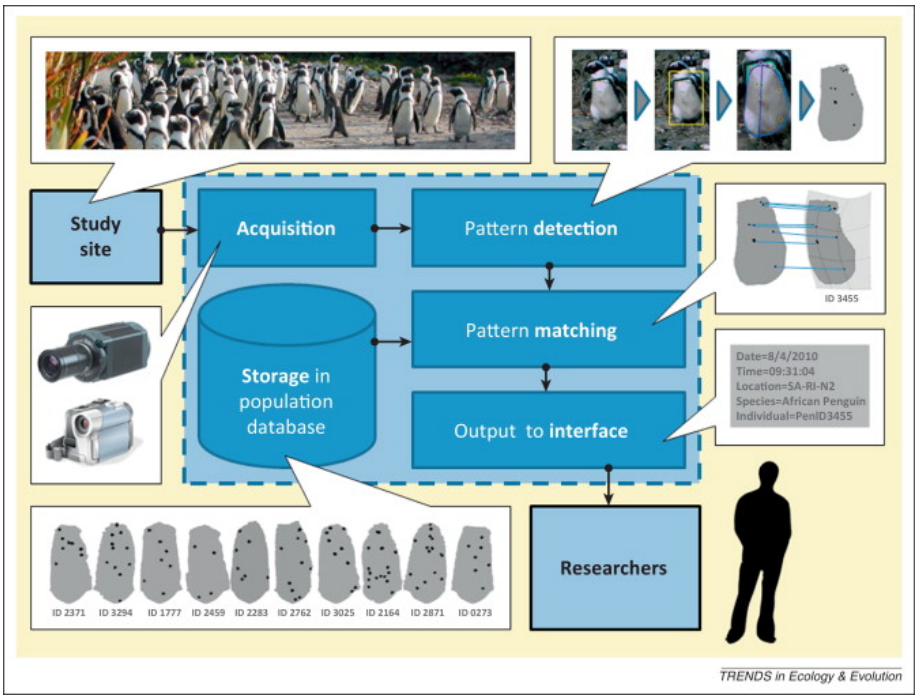

Figure 1.2: Components of an animal biometric system proposed in *Animal biometrics: quantifying and detecting phenotypic appearance*[23], using individual African penguin recognition as an example.

Animal Action Recognition has found recent applications in several fields. Behaviour analysis of wild felines has been used to protect grassland environments [12], and action recognition of cows can allow for farmers to design a scheme to promote the quality and productivity of their cattle [14]. In a similar vein, overpopulation of deer can often lead to negative impacts on crops and other wildlife [25], and behaviour analysis of Roe Deer has the potential to protect such resources by monitoring the frequency and location of behaviours of ecological interest.

1.2.2 Brandenburg Video Archive

The Brandenburg video archive[8], which constitutes the visual data for this project, was created by the German Centre for Integrative Biodiversity Research (iDiv). iDiv aims to advance research on the sustainable management of the Earth's biodiversity. The dataset is currently un-released to the public, so access to this dataset for this project was a privilege. Data collection took place in the conservation area 'Hintenteiche bei Biesenbrow' located in the Biosphere Reserve Schorfheide-Chorin. The videos consist of either greyscale infrared frames (captured during the night) at 18.5 frames per second (FPS), or RGB frames (captured during daytime) at 30 FPS, both at a resolution of 1920 × 1080 pixels. The dataset uses 39 Bushnell Trophy Cam HD Aggressor camera traps placed at a variety of locations. These camera traps record 1 minute of footage once motion has been detected in the environment via an infrared sensor. The video archive was manually annotated by ecologists to specify which, if any, species had appeared within each 1-minute recording alongside other pieces of meta-data for the video.

The video archive has been used to construct the dataset for this task. First, only videos featuring European Roe Deer were copied to the dataset. Each video was then split into its constituent frames to allow for further image processing if necessary. A subset of these videos were then manually annotated by hand on a frame by frame basis.

1.3 Related Work

There has been research into the use of computer vision techniques to identify animal biometric data. Kühl, Hjalmar S and Burghardt, Tilo laid the groundwork in *Animal biometrics: quantifying and detecting phenotypic appearance*[23] demonstrates how applying recent advances in computerised human recognition to animal populations can aid the work of ecological and evolutionary researchers. Since then, systems

to detect and identify a variety of biometric properties have been developed.

Figure 1.3: Visualisation of MegaDetector output on the Brandenburg data-set including bounding box and object classification.

An example of such a system, Microsoft AI4Earth MegaDetector, is a widely used tool to detect animals, humans, and vehicles in a variety of settings. The tool is used by ecological organisations spanning the globe and has been used for tasks such as the detection of threats to wildlife and the monitoring of protected animal populations. The underlying model is based on version 5 of the YOLO [36] architecture for object detection. MegaDetector empowers ecological organisation to automate terabytes of images, tasks that can take many years for humans to complete [44].

Animal detection and recognition has seen significant research in recent years for a variety of uses. Chen et al. [6] developed two deep learning frameworks based on Convolutional Neural Networks that analysed 8,368 images in order to automate the identification of species in the wild. The first model, which the authors called CNN-1, was a self trained network based on a new wildlife dataset. The second model, called CNN-2, was based on an AlexNet network pre-trained on the ImageNet dataset, and was then fine tuned for the wildlife dataset. The authors used a resampling process to increase the likelihood of uncommon classes being sampled from the training set during training, which improved the accuracy of both models. CNN-2 displayed superior accuracy to CNN-1, suggesting that a model instantiated with pre-trained weights from a large public dataset can improve model performance within the domain of wildlife detection. This research aimed to minimise the expense of monitoring the population of various wild species through the use of deep learning networks instead of manual counting. One of the frameworks was specifically designed to differentiate badgers from other animals, while the other could identify six distinct animal species. The reported accuracies for binary classification and multi-classification were 98.05% and 90.32%, respectively.

Favorskaya et al [11] investigated the effectiveness of joint CNN network in species recognition based on the features of an animals muzzle and their shape. The network employed three separate streams; the first two analysed the animal's muzzle and employed part-of-shape recognition and the third recognised the entire shape of the animal. The joint CNN achieved 80.1% Top1 and 94.1% Top5 accuracy results on a balanced training dataset, and 38.7% Top-1 and 54.8% Top-5 accuracy results on an un-blanaced training dataset, highlighting the importance of balancing a training dataset.

Norouzzadeh et al. [30] used footage from motion sensor camera traps to train a deep convolutional neural network to count animals, identify them, and describe their behaviour. The dataset comprised 3.2 million images featuring 48 different species. The research identified six "additional attributes" which described generic species behaviours, these being `Standing`, `Not Resting`, `Not Moving`, `Eating`, `Not Interacting`, and `No Babies`. The behaviour taxonomy chosen by the researchers resulted in multiple classes being present at once, and so the researchers used to a multi-label classification approach. The

best performing model, a ResNet-152 network, obtained 75.6% accuracy, 84.5% precision, and 80.9% recall.

Simonyan and Zisserman [40] introduced a two-stream approach behavioural action recognition from video data. This approach aims to incorporate spatial and temporal information from two separate streams. The researchers found that training a convolutional neural network on multi-frame optical flow data can lead to high performance despite a limited training set. The researchers achieved an accuracy of 87.5% when emplying an SVM fusion method to combine the two streams.

Sakib et al. [37] used a two-stream approach to detect 9 different great ape behaviours in the wild. The project addressed challenges such as a long-tail class distribution and the need to identify the behaviours of multiple apes within one scene. A model was developed that extracted features from a stream of RGB frames and a stream of optical flow frames, and various fusion methods were explored for combining the outputs of these streams. Brookes et al. [4] recently approached the same task by proposing a metric learning system which used body part segmentation to complement the use of RGB and Optical Flow streams. Again, different feature-fusion and long-tail recognition approaches were explored, and the researchers found an improvement of 12% in top-1 accuracy over previous approaches to identifying behaviours within this dataset.

1.4 Objectives

The aims of this dissertation are to investigate the effectiveness of existing deep learning approaches to action recognition on recognising deer behaviours and to extend these approaches to find more effective means of extracting behaviour information from video footage of deer. This work investigates how best to capture spatial and temporal information from video footage for the task of action classification on a novel data-set.

The objectives are:

- Label a suitably-sized subset of the Brandenburg cam-trap data-set with behaviour information
- Compare the performance of different video classification backbones
- Evaluate the performance of a novel deep learning model with the use of a variety of performance metrics, stratified across different hyper-parameter values
- Undertake class-based analysis on model performance to understand where and why a given model may perform poorly

1.4.1 Challenges

Class Distribution

The relative frequency of each behaviour within the data corpus reveals a large imbalance between different classes. The frequency distribution is loosely approximated by a long tail distribution. Long tail class distributions can result in models that are biased towards dominant classes, and that perform poorly on less frequently occurring classes [13]. The problem of training performant deep neural networks on long tail distributions is known as Deep Long-Tailed Learning, and three main solutions exist to this problem: class-rebalancing, information augmentation, and module improvement.

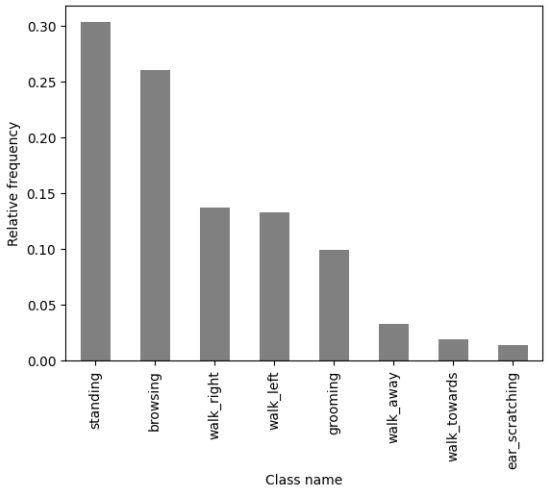

Figure 1.4: Relative frequency of each behavior within the labelled subset of the data-set

Class re-balancing aims to rebalance the training set by either re-sampling tail classes, altering the loss function to penalise incorrectly classifying tail classes to a greater degree than common classes, or adjusting the per class logits produced by the model. Information augmentation introduces additional information to the model through strategies such as transfer learning and data-augmentation. Module improvement describes strategies that augment the training of a model either through forms of representation learning, altering the design of the classifier, or decoupling training into separate representation learning and classification learning steps.

Manual Labelling

Labelling a subset of the Brandenburg Dataset required a considerable time investment at the start of this project. Time pressures limited the total numbers of annotations that could be made, and ideally all deer videos would feature detailed annotations. Each label consists of a unique video id, an index referring to which deer in a given video is being labelled (when multiple deer are present), a start frame for a given behaviour, and a corresponding end frame. Thus annotations are segmented by time and individual deer within each labelled video. This level of detail was essential for any meaningful inference to take place, as a variety of displayed behaviours were almost always present for each video. A deer would rarely just stand, browse, or groom during a video and instead most videos contained a complex series of behaviours each of which had to be labelled.

It was important to choose which behaviours to classify before annotating the data-set. A simple set of behaviours were set that required as little expert insight or mind reading of a deer as possible. For instance, a distinction between a deer standing still and looking around was found to be somewhat arbitrary, and distinctions of feeding on trees or shrubbery were not made. Keeping the class list small helps to increase the sample count per class which it was hoped would lead to increased training performance.

Natural Data

There are a number of challenges associated with performing behaviour action recognition on actions taken by wild animals within the natural environment. Cameras were located in a number of habitats, including wetlands, forests, grassland, river banks, agricultural areas, and next to streams. As such, the backdrop for each individual camera varied significantly. Behaviour recognition requires a model distinguish the subject from its surroundings, and a consistent backdrop both within the train set and across the train and test set partitions helps the model to more easily generate features based on the behaviour of the subject. The backdrop of every scene is comparatively more complex than those found in datasets for human action recognition. These datasets consist of footage of humans often in a man-made environment, which provide backgrounds that do not feature significant spatial variance such as walls and buildings. In the natural environment, many different elements such as tree trunks, grass, leaves and streams can coexist within a scene, creating a complex spatial pattern which a model may struggle to differentiate from the subject of the video. Video backdrops within a man-made environment are often

static, whilst elements such as trees and grass will move based on weather aspects such as wind and the movement of the subject of the video. Distinguishing between a moving subject and, for instance, grass that moves as a result of this subject, is a difficult task for a deep learning model that expects to encode all movement within a scene as a feature. The movement of these background elements, either due to the environment or the subject, is often random and as such this increases noise within the dataset. Footage recorded within the natural environment can also suffer due to increased occlusion from the scene. Within grassland, grass can often significantly obscure a deer. Other features of the environment such as trees can also occlude the subject of the video. The performance of a behavioural action recognition system will partially depend on the ability of the model to overcome these environmental challenges. A model that expects a consistent background for all behaviours will perform very poorly on a natural dataset such as the Brandenburg video archive, while a model that can successfully extract the behaviour of the subject of the video from the environment will perform significantly better.

Wild animals, such as wild Roe deer, may display behavioural variances between different individuals. The spatial properties and intensity of motion of actions such as grooming and walking can vary between individuals. A single individual may also exhibit the same behaviour in a different manner at different times due to differing levels of characteristics such as stress and energy. This property behavioural variance requires a large number of samples within the training set in order to provide a model with enough examples of each behaviour being performed in different ways. This property also requires a model that can generalise similar properties from samples of a given behaviour to a single behaviour class.

Chapter 2

Background

This chapter provides a comprehensive technical background of the concepts necessary to understand the remaining sections of the project.

2.1 Machine Learning

Machine Learning studies the use of computational and mathematical models to learn insights and predictions from data. Machine Learning is commonly thought of as being comprised of two separate disciplines, that of supervised and unsupervised learning.

2.1.1 Supervised Learning

Supervised learning is the process of teaching a model to make a prediction based on input data. Supervised learning usually employs training and test sets, consisting of input data and ground truth values. Models are trained to predict correct ground truth values from input data within the training data, and their performance is evaluated by their ability to generalise this learning to the unseen test set. Supervised learning can take the form of two sets of predictions, that of classification and regression. Classification problems require the model to predict one or more discrete labels from a finite set of possible labels, whilst regression requires the model to produce a prediction comprised of one or more continuous responses. Within the context of this project, a set of video frames constitutes the input data and the labelled list of behaviour classes constitutes the ground truth. This project is a classification problem as the model is designed to predict a class for a given input sample from a finite set of options.

2.2 Artifical Neural Networks and Deep Learning

An Artifical Neural Network (ANN) consists of one or more layers comprised of one or more idealised neurons. These neurons most commonly take the form of a perceptron, which takes takes a series of inputs, multiplies each by a learned weight, and then applies a given activation function to this output to produce the final output.

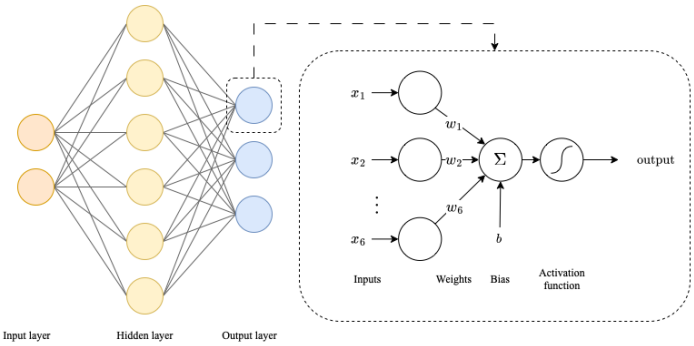

Figure 2.1: Visualisation of an ANN, including a decomposition of one of the constituent perceptrons within its output layer

The computation of the output can be given as

$$y = \phi\left(\Sigma x_i w_i + b\right)$$

where x is the input vector, w is the corresponding weight vector, b is the bias and ϕ is the activation function. Common activation functions include Sigmoid, TanH, and ReLU, each suited to different tasks. Perceptrons can be arranged as a layer. A fully connected layer refers to a layer where each perceptron within that layer receives an input from every perceptron from the previous layer. One or more layers can be stacked to form an ANN. The output layer is the final stage of the ANN and its output corresponds to the models prediction given the input data. For a classification model, the size of the output layer corresponds to the number of output classes that the model is choosing from. A Sigmoid activation function is commonly used in the output layer, which bounds the output between 0 and 1 for each class, so that each output corresponds to the network's predicted probability that the input data corresponds to that output class. Artificial Neural Networks can feature a single hidden layer and a single output layer, and a network featuring more than one hidden layer is referred to as a Deep Neural Network (DNN). All models used within this project feature more than one layer and are as such DNNs.

Loss

The error between the model's predictions and the ground truth is called loss, and represents the penalty for bad predictions. Various loss functions exist for different machine learning problems, with Cross-Entropy Loss well suited to classification problems. Cross-Entropy measures the performance of a classifier that represents a prediction by the probability that the input data belongs to each class, i.e., a series of values each between 0 and 1. For multiclass classification problems a separate loss is calculated for each class label, and the total sum represents the overall loss

$$L = -\sum_{i=1}^{M} y_i \log(\hat{y}_i) \tag{2.1}$$

where M represents the number of classes, log is natural log, y_i is the true lass label for i (1 if the class is i, otherwise 0), and $\hat{y}_i$ represents the predicted probability of class i. This loss function penalises a prediction that gives a low probability to the true class. This project involves classifying input data from a set of predefined behaviours, and as such Cross-Entropy Loss to calculate the error between the model's predictions and the ground truth.

Optimisation

Neural Networks are optimised through a process known as gradient descent. Gradient descent finds a local minimum of the loss through repeated steps opposite to the gradient at the current loss point. The process of updating weights is shown below.

$$W_i \leftarrow W_i - \eta \frac{\partial \mathcal{L}}{\partial W_i} \tag{2.2}$$

where W_i is the weight matrix at the ith layer, η is the learning rate, and L is the loss function. The gradient of the error function with respect to the weights, $\frac{\partial \mathcal{L}}{\partial W_i}$, is calculated using the chain rule.

The learning rate η is set before training begins, and optimising this hyper-parameter can lead to large gains in model performance. Strategies such as Adam allow for per-weight learning rates and adaptive learning rates, which makes tweaking this hyper-parameter less of a significant concern although still import.

The gradient descent algorithm has been extended to deal with a number of issues found when training a model through pure gradient descent. Stochastic Gradient Descent (SGD) represents a stochastic approximation of gradient descent optimisation algorithm by replacing the real gradient by an estimate randomly calculated from a subset of data. This reduces computational burden of gradient descent and prevents the network from stopping at a sub-optimal local minima.

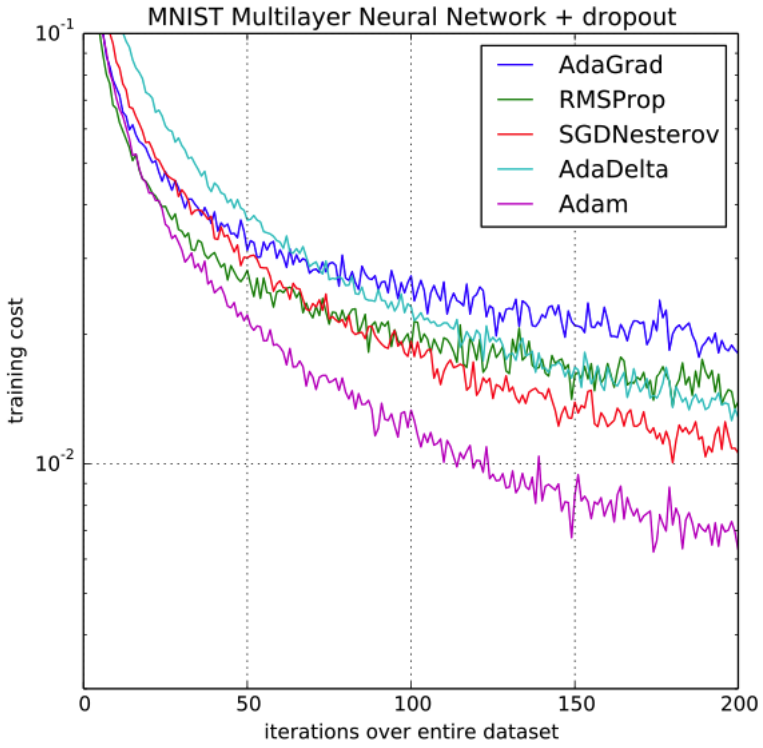

Figure 2.2: Adam outperforms other popular optimisation algorithms when training a multilayer perceptron. From *Adam: A Method for Stochastic Optimization*[21]

This project uses the Adam optimisation algorithm in order to achieve gradient descent optimisation. The algorithm [21] combines the strengths of two other contemporary optimisation algorithms: Adaptive Gradient Algorithm (AdaGrad) [27] and Root Mean Square Propagation (RMSProp). AdaGrad uses per-parameter learning rate as opposed to a single learning rate for all parameters. This can improve performance for tasks such as natural language processing and computer vision that often produce a sparse gradients. RMSProp adapts per parameter learning weights by the rate at which they are changing. This helps the algorithm excel in problems containing noisy data. This is especially useful in this project where video frames often feature a large noise-to-signal ratio due to the inherent noisiness of visual information from the natural environment, as discussed in section 4.1. Adam compares well to other optimisation algorithms and is widely used in the field of deep learning.

2.3 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a type of Neural Network that contains convolutional layers. A traditional CNN, such as AlexNet or a Resnet features interspersed convolutional and max-pooling layers to generate a complex hierarchy of features which are then processed by one or more fully connected

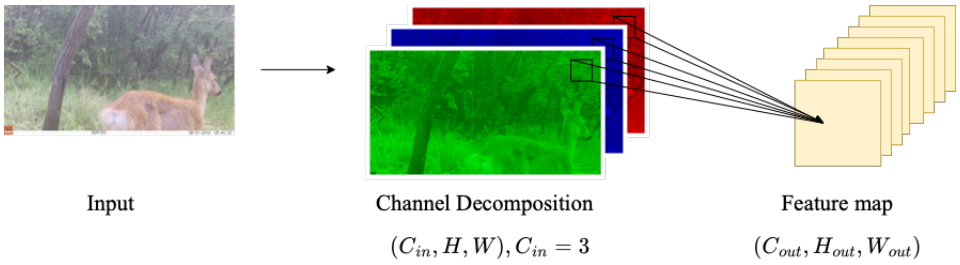

Figure 2.3: Visualisation of 2D convolution of an RGB input image

layers. The feature producing part of the network is shift invarient, meaning CNNs are well suited to recognition tasks on image and video input data. Shift invariance refers to the ability to learn patterns in multi-diemsional data regardless of the location of the pattern within a given dimension [49].

Convolutional Layer A convolutional layer applies multidimensional discrete convolution between its input and a number of learned filters during a forward pass through the network to produce a feature map. The value of the output of a convolutional layer with input size (N, C_{in}, H, W) and output size $(N, C_{out}, H_{out}, W_{out})$ can represented by

$$\text{out}(N_i, C_{out_j} = \text{bias}(C_{out_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out_j}, k) \star \text{input}(N_i, k) \tag{2.3}$$

where N is the batch size, C is the channel count (3 for an RGB image), H is the height of the input planes and W is the width of the input planes. $\star$ is the 2D cross-correlation operator. H_{out}, W_{out} are determined by the kernel size, stride and dilation of the layer.

Feature maps can be used as input to other convolutional layer, resulting in a hierarcy of learned filters, and can also be passed to fully connected layers to solve regression and classification problems. Many famous convolutional neural networks feature multiple convolutional layers to produce complex features, which are then passed to at least one fully connected layer.

Three Dimensional Convolution A three dimensional convolution involves a kernel comprised of three dimensions that slides over an input of three dimensions. Three dimensional input can be used to represent medical data, topographic data or in our case video data with the added dimension representing the temporal dimension.

Pooling Layer A pooling layers down-sample inputs along a given dimension. They are frequently used to down-sample feature maps produced by convolution layers to improve shift in variance within the feature producing part of a CNN.

2.4 Resnet

ResNet (short for Residual Network) is a deep CNN architecture. It was developed to address the issue of vanishing gradients in very deep neural networks, which occurs when gradients in the inital layers of a neural network are very small making it difficult to update their weights.

Residual blocks ResNet uses residual blocks to allow networks to learn residual functions rather than just filter maps, which mitigates the issue of vanishing gradients during back-propagation. Residual blocks utilise skip connections which bypass layers in the network, so the result of a block becomes a combination of the feature maps and the input to that block. Skip connections allow gradients to propagate directly from the output to the input of the block, rather than through the block, meaning gradients can more easily flow back to the initial layers of the network.

Downsampling residual blocks Downsampling residual blocks are a varient of residual blocks that produce an output consisting of more channels but each of which with smaller spatial dimensions. This

is achieved with convolution operations with a large kernel stride. Downsampling spatial dimensions reduces the number of network parameters and increases the number of channels to be passed a fully connected layer after feature extraction.

ResNet-18 ResNet networks are prepended by the number of layers within the network. The two most popular networks ResNet-18 and Resnet-50, are 18 layers and 50 layers deep respectively. A deeper architecture results in more model parameters, each of which need to be trained. Thus, training a larger ResNet results in epochs which take longer to complete.

3D Resnet-18 and (2+1)D Resnet-18 The network 3D Resnet-18 replaces the two-dimensional convolutional layers found in Resnet-18 with three-dimensional convolutional layers. The network (2+1)D Resnet-18 replaces two-dimensional convolutional layers with a convolution layer that performs a two-dimensional convolution followed by a one-dimensional convolution. Both models were introduced in the paper *A Closer Look at Spatiotemporal Convolutions for Action Recognition* [43], where the researchers aimed to develop 3D CNNs using a residual learning network architecture for the task of action recognition. The project will analyse the efficacy of these two models in detecting deer behaviour from visual data.

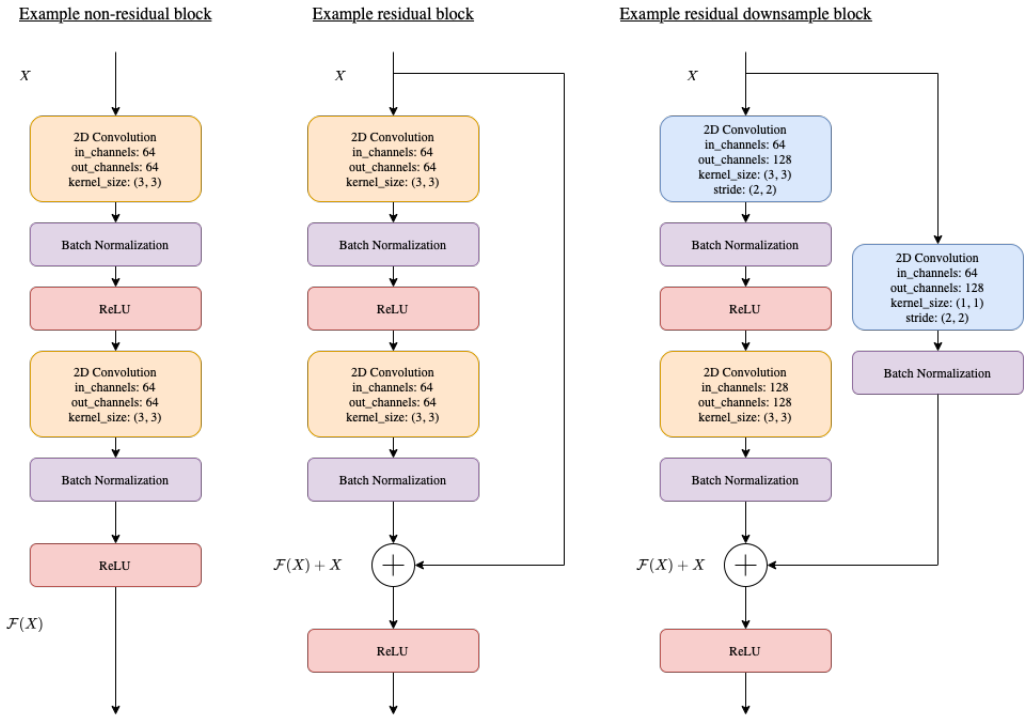

Figure 2.4: The example non-residual block (left) demonstrates a normal stacking of convolution layers. The example residual block (centre) shows the use of a skip connection. The example residual downsample block (right) demonstrates convolutional downsampling of the main convolution block and of the skip connection. Convolution layers that result in downsampled features are coloured in blue.

2.5 Transformers

A transformer is a complex deep learning model which applies an attention mechanism based on an encoder and a decoder [45].

Attention Attention assigns a weight to each part of an input sequence based on its relevance to a given output This is then used to compute a weighted sum of input elements to generate a prediction. Self-attention allows for long-term dependencies and relationships between aspects of a sequence to be encoded. Multi-head processes different parts of the input sequence simultaneously, using different sets of weights to perform multiple attention operations. Encoder-Decoder Attention is used by sequence-to-sequence models to encode a representation of the inputs and then decode this representation to a

meaningful output [45]. This is a high level description of the attention mechanism; going into further details on the intricacies of attention is not in the scope of the project.

Video Transformers Video Transformers are transformer networks that process video frames as their input sequence to be used in the attention mechanism. Video transformer networks often address the high dimensionality of video data by down-sampling through compression of video frames to reduce dimensionality and/or using multi-head attention to capture various aspects of video data. Video transformers often apply recurrent connections between the encoder and decoder modules in order to encode long-term dependencies within video data [38].

MViT The Multi-scale Vision Transformer (MViT) network [10] combines the concept of multiscale feature hierarchies often found in CNNs with a transformer model. The network learns a hierarchy from spatial and channel-based features to coarse and complex features. The second incarnation of this network, MViT V2, introduces positional embedding and residual pooling connections, to the attention mechanism found in MViT. The introduction of residual pooling connections reduces the compute and memory load incurred when training a transformer network. The project will test the efficacy of MViT V2 in detecting deer behaviour.

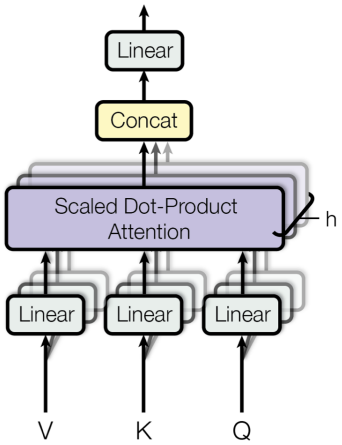

Figure 2.5: Visualisation of the Multi-Head Attention mechanism given a set of queries Q, a set of keys K, and a set of values V.

2.6 Two Stream Models

The challenge associated with behavioural action recognition is to capture both the spatial and temporal properties of a behaviour from video data. Simonyan and Zisserman [40] proposed a two stream model comprised of a spatial network and a temporal network. The model uses two separate ResNet networks, which are described as streams. A spatial stream attempts to capture the positional properties of an action. For example, for a deer this may constitute the positions of different limbs during standing or the position of the head during grooming. The temporal stream attempts to capture the properties of motion associated with a behaviour. Within this project, this could constitute the movement of the deer's head during browsing or the lack of motion associated with a deer standing still. Thus, a two stream model can capture distinct properties of a behavioura and combine these properties for a more accurate prediction.

Simonyan and Zisserman [40] used a single RGB frame as the input to the spacial channel and stacked alternating horizontal and vertical optical flow displacement frames for the temporal stream. Optical flow represents the displacement of each pixel between two frames. Vertical displacement represents the vertical motion between the two frames, and horizontal displacement represents the horizontal motion between the two frames. Carreira and Zisserman [5] proposed an expanded set of two stream models. One of the models proposed, Two-Stream 3d-ConvNet, uses a set of RGB frames and a set of optical flow frames, both of equal length, as inputs for the respective spatial and temporal model streams. The spacial and temporal streams used 3D convolutional networks to encode behavioural features. Both the 3D ResNet based models and the MViT models used in this project follow this concept of incorporating two streams of video information of equal length.

Fusion Fusion within the context of deep learning refers to the process of integrating outputs produced by distinct networks. The combined outputs can be further processed by subsequent layers or used to generate the model's prediction. Early fusion involves fusing early layers in the network whilst late fusion involves the concatenation of the output of streams. Logical fusion involves averaging the responses of two output layers of two seperate networks.

2.7 Pretrained Models

Transfer learning refers to improving a model trained on one domain by using information within a related domain. This concept can be applied to deep learning by training a model on a large, general data-set to create a set of optimal model weights and then instantiating models with these weights to then be trained on a task of narrower scope [47][28]. When successfully applied, transfer learning allows the pre-trained model to converge more quickly and to a greater accuracy during training than a model that has not been pre-trained. This can improve accuracy for a model performing recognition on a comparatively small dataset, such as the one in this project, by transferring learning from a larger more general dataset to the task of identifying a smaller set of classes from a smaller number of samples.

There are two main models of transfer learning within deep learning, those being fine-tuning and feature extraction.

Fine-tuning A pre-trained model with a new classifier suiting the domain problem is trained on the new domain. All weights within the network are updated during training, allowing for knowledge gained from training to be built upon by further training in the new domain.

Feature Extraction A pre-trained model has its original classifier changed to one suiting the domain problem, and all network weights except those of the classifier are frozen. The pre-trained model's weights do not change and only the classifier is updated during training.

2.8 Overfitting

The problem of over-fitting occurs when a model encodes overly complex patterns within input data that do not represent meaningful learning within the domain. This causes the model to generalise to unseen data very poorly, usually shown by a large disparity between training loss, and validation and test loss. This can be the result of an overly complex model, such too many layers, too many units per layer, or certain weights given too much influence within an entire model. During training, if a model's train accuracy continues to increase but its accuracy on unseen data decreases then it is learning patterns within the training data that are not present within validation and test data. This is of particular concern when training a model on a comparatively small dataset such as the one in this project, where only having a small number of samples in the training set can prevent the model generalising the target classes to the test set. Thus, the model may be very effective on the train set but may not be able to generalise this learning to the test set if the properties learnt from the train set are too specific for the effective classification of samples from the test set.

L_2 Regularisation and Weight Decay L_2 regularisation penalises large model weights in order to reduce model complexity. The strength of the penalty is determined by the term λ which is set before training. Loss with L_2 regularisation is computed by the following equation

$$\text{Loss} = \mathcal{L}(y, \hat{y}) + \lambda \sum_i w_i^2 \tag{2.4}$$

where $\mathcal{L}(y, \hat{y})$ represents a true loss function such a Cross Entropy Loss or Mean Squared Error Loss, λ represents the penalty coefficient, and $\sum_i w_i^2$ represents the sum of the square of all network weights.

Weight decay, a form of regularisation used in optimisation algorithms such as Adam, which is used in this project, applies a system similar to L_2 regularisation, but updates the update step as opposed to the loss function in order to prevent overfitting, using a penalty similar to that of L_2 regularisation. Per weight updates with weight decay is shown by the following equation

$$w_i \leftarrow w_i - \eta \frac{\partial \mathcal{L}}{\partial w_i} - \eta \lambda w_i \tag{2.5}$$

with λ again representing the penalty coefficient, $\eta\lambda w_i$ representing the penalty for large weights. Employing weight decay during training can help to prevent a model fitting too tightly to the training set on a smaller dataset such as the one in this project.

Dropout Randomly dropping connections between layers during training, with a probability p to each connection, can help to prevent networks relying on few, large, weights to make predictions on data. Dropout results in weights being more evenly distributed between two layers, helping to reduce model complexity and in turn reduces over-fitting [3].

2.9 Object Detection

Object detection is computer vision problem of detecting and locating the presence of instances of a given class. Object detection is foundational for many related computer vision tasks, such as object tracking, image captioning, and instance segmentation [50]. For this project, object detection is applied to our data corpus to generate bounding co-ordinates of Roe Deer for all video samples. Knowing the location of an agent performing an action can help in instances of multiple agents performing different actions within a scene so that action recognition can take place on a per-object rather than per-scene basis.

2.10 Metrics

A variety of statistical methods are used to evaluate the performance of different models and training techniques during this project. These metrics are applied during training on the train set and on the test set.

Top 1 Accuracy Top 1 accuracy is the number of correct behaviour predictions for a given set of data.

Top 3 Accuracy Top 3 accuracy measures whether a model correctly predicts the behaviour within the top three most likely behaviors it predicts. This metric demonstrates whether the second or third guess of a model is correct and a high top 3 accuracy measures whether a model generally predicts correct classes even if it cannot tell the difference between similar classes. A bound of 3 is considered appropriate for a model predicting between 8 total classes.

Loss The loss between predicted and ground-truth values, as computed in 2.2.

2.11 Dataset

As mentioned in section 1.2.2, the Brandenburg video archive provides the underlying data that will be used in the project. The video archive is provided alongside annotations created during the undertaking of *"Overcoming the Distance Estimation Bottleneck in Estimating Animal Abundance with Camera Traps", H Kuehl et al.* [17]. These annotations include:

- `language_vernacular_name`: The species present in the corresponding 1-minute video sample. This includes "European roe deer" alongside other species such as "European Beaver" and "North American raccoon".
- `ID`: A unique identifier for the corresponding 1-minute video sample. This is distinct from the video file name, which is not guaranteed to be unique. This identifer is composed of as such `<CameraID><Video Index>`. For instance, the 40th video from camera T01 has the ID `T0100040`. Knowing the camera ID from the unique ID makes it easy to partition datasets by camera.

This allows for relevant video samples to be pulled from the Brandenburg video archive to construct the video dataset necessary for the project. However, there are no annotations detailing behaviours within the video clips, requiring additional annotations to be added to the dataset:

- `behaviour`: The behaviour being displayed by a given deer. This only constitutes a single deer as the model is designed the return a single behaviour classification for a given input.

- `idx`: The deer index during a given behaviour. There may be multiple deer present during a video so it is important to label which deer within a given time-frame is being labelled. The index is ordered by left-to-right for the x coordinate of the top left point in the deer bounding box.
- `start`: The frame at which the deer has began displaying the annotated behaviour.
- `end`: The frame at which the deer was no longer displaying the annotated behaviour. This may be due to the deer leaving the frame, or due to the deer changing its behaviour to a different behaviour.

Start and end frame indexes are used instead of repeated incremental frame behaviour annotations in order to save space within the annotations file and to speed up the annotations process. It is common for a deer to exhibit a behaviour for at least 50 frames so annotating every consecutive frame with the same behaviour for the same deer produces a large amount of redundant data.

2.12 PyTorch, Torchvision

PyTorch is an open-source machine framework library for Python based on the Torch library. PyTorch provides a number of important high-level abstractions on top of normal python programming that allow for fast development of complex neural networks.

torch.Tensor Tensors are a key component of working with PyTorch. Tensors in PyTorch represent an n-dimensional space, and tensors can be loaded into GPU virtual memory in order to improve performance for large computations. PyTorch supports CUDA acceleration allowing machine-learning related computations to be computed on GPUs, which is achieved by loading tensors onto GPU memory.

torch.nn The torch.nn package, `nn` short for Neural Network, provides the basic building blocks for the construction of Neural Network graphs. torch.nn provides implementations of a number of key deep learning network components such as fully connected layers and convolution layers. Users can define their own building blocks by inheriting the `nn.Module` class and implementing the `__init__` and `__forward__` functions which represent initialising module values and performing a forward pass respectively. Custom `nn.Module` classes allow for blocks of deep learning layers to be reused and composed within other custom blocks in order to create large, complex networks.

torch.autograd Autograd is an automatic differentiation engine that allows for automatic generation of network gradients in order to perform gradient-descent learning. Autograd automatically collects gradients on the results of a forward pass through a neural network through a call to `.backward` on the tensor. Autograd removes the need to manually calculate backpropagation gradients during training, increasing developer productivity.

Torchvision Torchvision is a package from the PyTorch team that extends PyTorch's capabilities in computer vision tasks by providing implementations of popular model architectures, data-sets, and video and image transformations. Torchvision provides pre-trained weights for many of its model implementations which can be used for transfer learning purposes.

Dataset PyTorch provides dataset abstractions under the module `torch.utils.data`. Developers can implement a map dataset by inheriting the `torch.utils.data.Dataset` class and implementing required functions. Dataset parameters for initialisation are passed to `__init__`, `__len__` returns the length of the dataset, and `__getitem__` returns an item at a specific index.

Figure 2.6: An overview of the Brandenburg video archive [8]. The video data is sourced from a variety of locations at different times of day, with deer featured at different distances from the camera. There can be between one and three deer present per frame, and deer of different ages are featured in the video data.

Chapter 3

Project Execution

This chapter describes the main activities taken to achieve the goals of the project. Successful completion of these goals required completing various components within the project, the details of which will be described in detail in this chapter. This chapter shall thus explain decisions that were made with regards to the dataset, the model implementation, training, evaluation and experimentation.

3.1 Data

3.1.1 Dataset Restructuring

The original Brandenburg dataset, provided by the project supervisor, was comprised of a folder containing all videos taken by the camera traps over the course of data collection, and a csv file annotating key metadata details for each video. The folder containing all videos was comprised of multiple nested folders detailing the camera used and the time at which the footage was captured. This folder structure meant that a complex path to the video had to be saved within the annotations file in order to find the video, which was not ideal. The annotations csv file contained 18,899 rows and 37 columns. Most importantly, the file contained manual annotations detailing which species were present within each video (`species`) and a unique identifier for each video (`ID`). A python script was used to filter the videos to only those containing deer, which were then copied to a new shallow dataset folder where each video was contained within one directory named after the video ID. This folder is considerably smaller than the original dataset folder, and allowed for python scripts to be easily able to access a video by just its video ID.

3.1.2 Dataset preprocessing

A variety of tasks are required before the data can be labelled, and before the data can be used by models to complete the goals of this task.

Frame Generation

Generating and saving each individual frame for each video allows for more accurate labelling and provides performance improvements during training and evaluation. Access to each video on a frame by frame basis allows for behaviours to be labelled with a start and end frame. Deer can perform many of our behaviour classes in less than a second, so using a frame precision can help us label behaviours more accurately. Furthermore, daytime RGB videos are recorded at 30 frames per second whilst nighttime heat detection videos are recorded at 18.5 frames per second. Using second or millisecond based start and end times to slice videos to the correct sample would require understanding of the frame rate of each video, adding overhead during data loading. Further operations such as optical flow generation can be sped up by already having each videos frames accessible as opposed to recapturing each frame from a video before each operation on a video. Thus, access to a sequential list of individual frames is key to the execution of this task. Frames are extracted using the cv2 library within Python, and a script to extract frames is run over the entire data-set.

Optical Flow Generation

It became apparent that optical flow representation might provide a performance benefit during the execution of the project. Manual optical flow generation using the TV-L1 algorithm and cv2 proved incredibly time consuming, and due to the time constrains of the project a quicker means of obtaining optical flow information was needed. The RAFT Model, from RAFT: Recurrent All-Pairs Field Transforms for Optical Flow, was instead used to generate optical flow between each successive frame in the dataset. Using a neural net to predict optical flow as opposed to calculating the exact optical flow between each pair of frames provides a significant speed-up in frame generation.

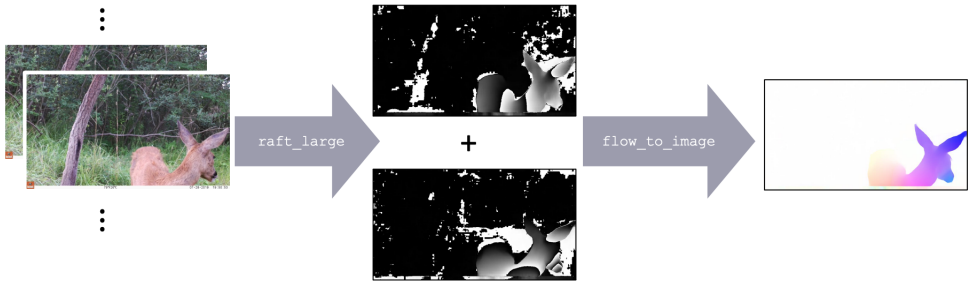

Figure 3.1: Demonstration of optical flow generation, consisting of horizontal displacement frames, vertical displacement frames, and RGB optical flow frames.

The model predicts the horizontal and vertical displacement of each pixel between a pair of successive frames. Frames representing horizontal and vertical displacement are saved in the directories `horizontal_flow/` and `vertical_flow/` respectively. These frames are combined using the PyTorch utility `flow_to_image` to form an RGB representation of the optical flow, saved in the `flow` folder. `flow_to_image` normalises the horizontal and vertical flow, based on the implementation from the RAFT paper, using a color wheel to represent flow direction and intensity. The operation in the paper is used purely for visualisation however this project will investigate the benefit of using this RGB representation as opposed to integrating horizontal and vertical flow directly.

Optical flow helps to highlight behaviours that may not be detected from purely spatial analysis. For instance, a deer standing or walking could look very similar from a still frame but the optical flow of the deer can reveal movement

Annotation Generation

The original dataset was batch processed by Microsoft Megadetector before being made accessable for the project. Each video was accompanied by a `results.json` file containing detected bounding boxes, a video visualising these bounding boxes, and a folder containing cropped frames of said bounding boxes. Unfortunately, Megadetector had been applied with a frame skip of 5 frames, meaning that significant temporal detail was lost for each video clip. Furthermore, when more than one deer was detected, the folder of cropped frames would only include one detection, so frames would often alternate between different deer randomly, and there was no way to tell based on frame names or file structure which deer belonged to which frame. After taking these limitations into account it was decided to re-run Megadetector over the entire dataset.

Megadetector provides a python script `process_video.py` which splits a video, or folder of videos, into frames and applies the model as a batch to these frames. Megadetector attempts to load all videos into memory at once when processing a folder of videos, which is both time consuming and enduces significant memory consumption. Instead of taking this approach, a custom script was written, `annotation_generation.py`, which directly accessed megadetector data structures and functions and called them directly. This allowed for custom optimisation logic and sanity checks. Within this script each video was individually loaded into memory and processed, and then removed from memory using Python's garbage collector, significantly reducing memory usage. The script can either be passed an annotations csv file and only annotate videos found within that list, or can receive a list of camera IDs from a text file and only process videos from those cameras. The order in which videos are processed is randomised, so running multiple batch jobs on the same dataset at once can provide significant

performance improvements by concurrently processing different videos. After some experimentation a confidence threshold of 0.7 was chosen for bounding box generation, the results of which were stored as a `detections.json` file, with cropping being done during data initialisation so that crops from different locations within the frame could all be used.

Figure 3.2: Visualisation of Megadetector bounding boxes. Bounding boxes are indexed left to right based on the relative x coordinate of the top-left corner of the box.

3.1.3 Dataset structure

After restructuring the dataset and generating video frames, annotations, and optical flow frames, the dataset assumes the following structure. Each folder contains the original video, a `detections.json` file containing bounding boxes, and folders containing original video's frames, frames of an RGB representation of optical flow, frames of horizontal flow, and frames of vertical flow. This directory structure made managing the data-set much easier and a multi-folder path to information was not needed in the annotations folder, replaced by a single video id.

```
.dataset
|-- T0100001
|   |-- T0100001.MP4
|   |-- detections.json
|   |-- flow
|   |   |-- 1.jpg
|   |   |-- 2.jpg
|   |   ...
|   |-- frames
|   |   |-- 1.jpg
|   |   |-- 2.jpg
|   |   ...
|   |-- horizontal_flow
|   |   |-- 1.jpg
|   |   |-- 2.jpg
|   |   ...
|   |-- vertical_flow
|   |   |-- 1.jpg
|   |   |-- 2.jpg
|   |   ...
|-- T0100008
|   ...
...
```

3.1.4 Class Definition

A subsection of the dataset was initially assessed to understand which behaviours occur most frequently. The aim was to devise a small set of behaviour classes that can accurately represent the visual behaviour

of a deer to a user without formal training in animal behaviour. As such, complex behaviours such as avoiding predators, looking for a mate, or finding shelter were avoided, as it requires expert knowledge to understand when such behaviours are taking place, and they are not always very common. Instead, simple behaviours were chosen that could be used to analyse the species' relation to a particular location. Behaviours such as browsing can represent an area where deer prefer to feed, and an area where deer mostly walk during a clip can be seen as a transitional space for the species.

Data is labelled on a frame-by-frame basis rather than a video by video due to the simple nature of these behaviours, as many different behaviours often occur for a single deer for a single clip. Labelling with an exact frame start and end point presents its own set of challenges. For instance, assessing the exact frame at which a deer stops walking and starts standing is subjective, and the same holds for many other behaviours.

In order to maintain consistency, behaviours are said to begin when the deer makes any motion that indicates the action it is about to undertake, such as readying itself to begin walking or moving its head to begin grooming itself. This can present challenges as for instance a deer may move its head in a similar manner but not groom itself, however categorising behaviours in such a way provides greater consistency than waiting until a few more frames until the behaviour is already in motion, as the previous behaviour annotation would suffer from unusual motion.

Deer can sometimes undertake more than one action at once, such as quickly alternating between walking and browsing. In instances where behaviours quickly alternate annotations aim to capture each individual behaviour. When a deer is partially obscured, either by the environment or being outside of the field of view, annotations are avoided. It is difficult to discern whether a deer is for instance grooming itself, browsing, or just standing, and so in order to prevent inaccuracies such occurrences are excluded from labelling.

Encoding movement as a behaviour can also pose challenges. Using a simple directional classification (towards the camera, away from the camera, right to left, left to right) will work for most deer movement however deer can often move both perpendicular to the camera and towards or away from the camera at the same time. There are a few solutions to this issue, each with their own caveats. Either the number of directions can be expanded to include 8 angles of motion rather than four, although this significantly increases the class count, and can decrease label accuracy as it can be hard to discern which of these categories each movement would fit into. Forward and backwards motion could be removed as classes in their entirety, with instances of diagonal movement just assigned based on left to right and right to left, although this removes the ability to label a deer walking directly towards or directly away from the camera. We can only classify forwards and backwards motion when the deer is only moving forwards and backwards and not diagonally, although this introduces human biases into the labelling as motion in a three dimensional space almost always involves moving in multiple axis, and finding the cut-off point between diagonal motion and moving forwards or backwards would be difficult to make precise. The chosen solution involves thinking of motion as within four different 90 deg segments around the deer, one pointing left, one right, one towards the camera and one away. This provides clear cutoff points, and represents the multidimensional nature of deer motion.

Figure 3.3: The dataset features deer undertaking multiple behaviours at once, deer where their behaviour is obscured, or where their pattern of movement does not easily fit into classification. Deer can be found to be browsing while walking (left), standing slightly out of frame and thus obscuring their behaviour (centre), or walking both towards/away and left/right at once (right)

Given these considerations, all behaviours within the subset of the dataset can be annotated to as accurate and consistent a degree as is possible. Inconsistent and/or inaccurate labels can lead to incorrect results

during experiments, and subsequent poor decisions when developing models, so ensuring the integrity of labelling is key for this project. Some issues may persist, such as human error incorrectly labelling walking in one direction as walking in another, however double and triple checking the labels can help to rectify any inaccuracies.

After examining a number of videos, the following classes were chosen:

- `standing` : The deer is standing in one place.
  - The deer may be moving their head or legs, but is not moving in a particular direction
  - The deer is not undertaking any of the other predefined behaviours.
- `grooming` : The deer is grooming itself.
  - This involves a deer licking the side of its body, with its head bent back around itself
  - Its head if often obscured if it is standing parallel to the camera and it is grooming the side of itself not visible to the camera
- `browsing` : The deer is eating vegetation.
  - Deer browse as well as graze - this means they eat both leaves, twigs and soft shoots and also grass and plants. These behaviours have been rolled into a single class to avoid expert labelling.
  - This represents a deer actively obtaining and eating food from the environment. A deer standing up and chewing something it has already obtained from the environment is classified as standing.
- `ear_scratching` : The deer is using one of their legs to scratch their ear and the region surrounding it.
  - This is a distinct behaviour from grooming, in which a deer uses their tongue (in our definition)
- `walk_towards` : The deer is walking towards the camera.
  - This includes any behaviour where the deer is walking "more towards" the camera rather than more to the side, based on the segment classification described above.
  - This includes all intensities of motion, including walking, running, and jumping.
- `walk_away` : The deer is walking away from the camera.
  - This includes any behaviour where the deer is walking "more away" from the camera rather than more to the side, based on the segment classification described above.
  - This includes all intensities of motion, including walking, running, and jumping.
- `walk_left` : The deer is walking right to left within the perspective of the camera.
  - This includes any behaviour where the deer is walking "more left" relative to the camera rather than forwards or backwards, based on the segment classification described above.
  - This includes all intensities of motion, including walking, running, and jumping.
- `walk_right` : The deer is walking right to left within the perspective of the camera.
  - This includes any behaviour where the deer is walking "more right" relative to the camera rather than forwards or backwards, based on the segment classification described above.
  - This includes all intensities of motion, including walking, running, and jumping.

These simple behaviours allow the model to classify one behaviour at a time in a way that is easy to interpret for the average user. Furthermore, these classes allow for feedback on where the model does well and where it does not, for instance if it can spatially tell the difference between standing and grooming and if it can understand different directions of motion.

Figure 3.4: Examples of deer performing each of the eight behaviour classes. These were found to be the most frequent behaviours within the dataset.

3.1.5 Behaviour Annotation Format

Behaviour annotations were written in one large csv file. This avoided complex logic for initialising and updating a large number of structued files, such as XML or JSON. The `behaviour_annotations.csv` file contains all annotations for the data-set.

video_id	index	target	start	end
T0100029	0	walk_away	1	58
T0100029	0	walk_left	58	111
T0100029	1	standing	1	222
T0100468	0	walk_left	1	148
...	...	...	...	...

The headings are defined as

- `video_id` refers to the unique id of the video
- `index` refers to which deer within a frame is being annotated. Bounding boxes are ordered by left to right from the top-left point of relative bounding box co-ordinates, so an index of `0` is the bounding box that begins furthest to the left in the video.
- `target` is the behaviour being taken by deer during the sample
- `start` is the frame where the behaviour starts
- `end` is the frame where the behaviour ends

Frame files are 1-indexed, as they begin with `1.jpg`, so `start` and `end` are also 1-indexed. The `end` values are exclusive while `start` values are inclusive. This follows the convention for list slicing in Python.

3.2 FrameDataset

Generating input samples required a number of decisions to be made with regards to efficiency of sample generation and the efficacy of a sample representing a behaviour in such a way as to be understandable to a DNN. The key task of sample generation involves converting a labelled subsection of a video, stored in `.MP4` or `.AVI` format into one or more of `torch.Tensor` samples representing the corresponding visual information. This involves loading frames, cropping frames, converting frames to `torch.Tensor` objects and applying transformations to these objects.

PyTorch provides the `torch.utils.data.Dataset` class to implement sample generation logic and the `torch.utils.data.DataLoader` class to implement batch generation during training and evaluation. As discussed in section 2.12 the `torch.utils.data.Dataset` class can be inherited in order to incorporate custom dataset logic into a project by implementing a number of key functions. The `FrameDataset}`class implements the `torch.utils.data.Dataset` base class to introduce custom data-set logic required for the successful completion of the project. Implementing this base class requires implementing the `__init__`, `__getitem__`, and `__len__` class methods. `__init__` is a requirement for writing any Python class, and is called when the dataset object is initialised. Within the context of a PyTorch dataset is responsible for constructing a hash-map of samples that can each be sampled by a given numerical index. `__getitem__` returns a unique sample for a given index, and is used by PyTorch data loaders to produce sample batches during training and evaluation. `__len__` returns the total number of samples that can be returned by the dataset object.

3.2.1 Data Transformations

Video frames must be transformed to a representation that can be used as an input to the model. Both RGB and optical flow frames are resized by the same operations that transformed the pretraining data-set. Frames are resized to $(128, 171)$ and then centrally cropped to size $(112, 112)$. RGB frames are normalised to mean $(0.43216, 0.394666, 0.37645)$ and standard deviation $0.22803, 0.22145, 0.216989$. This matches the mean and standard deviation values of the Kinetics-400 pretraining set, which means the distribution of intensities of the different colour channels matches those of the pretraining dataset. This allows for transfer learning to take place, as the distribution of feature maps produced by the model are dependent on the distribution of the colour channels of the training set. Optical flow frames are normalised to mean $(0.5, 0.5, 0.5)$ and standard deviation $(0.25, 0.25, 0.25)$.

3.2.2 Data Pre-loading

Loading a 1080p video from hard disk to memory is a time consuming task within the context of deep learning. The basic PyTorch `Dataset` materials suggest loading data when accessed every epoch in the `__getitem__` function. This introduces significant performance overhead when dealing with high resolution, high frame-rate data, as loading identical data every epoch quickly becomes incredibly time consuming. Performance can be improved by generating all samples before training and evaluation loops, and accessing these pre-generated samples when needed. The helper function `init_dataset` is used in the `__init__` implementation to produce a `self.data` hash-map which can be accessed directly by the `__getitem__` function during training and evaluation, meaning samples only need to be generated once. The hash-map contains all samples in a format ready to be used as input to a neural network, so no data processing needs to take place during training and evaluation iterations.

3.2.3 Splitting Samples

Processing a number of samples as a batch means that using samples of differing lengths would require padding, introducing extraneous information into samples and skewing clips towards the longest possible sample. Furthermore, the MViT backbone expects 16 frames per sample so for consistency both MViT and 3D Resnet-18 receive 16 frame samples.

The models investigated within this project aim to encode some form of temporal dependency between frames, either through a convolution or an attention-based relationship. It is therefore important that actions taken at a similar speed have similar characteristics between frames. It is therefore inappropriate to uniformly interpolate a behaviour consisting of, for example 200 frames, and another of the same class consisting of 30 frames, and expect a model to find temporal similarities between the two as the difference between sequential frames will be significantly greater for the longer sample than the shorter.

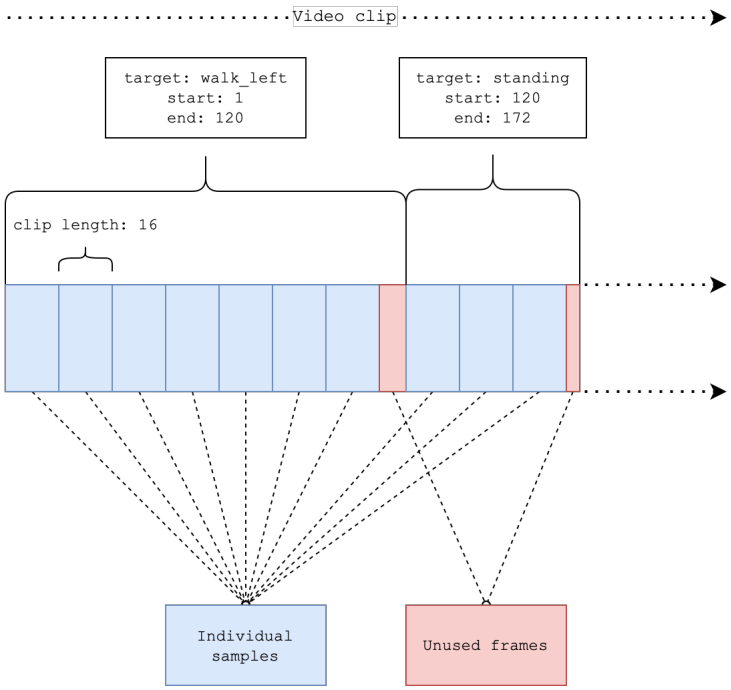

Figure 3.5: Labelled behaviours are split into multiple samples of a given sample length. This increases the number of samples available and helps the model learn different phases of a given behaviour.

Given these considerations, each behaviour annotation is split into consecutive 16 frame samples, as demonstrated by figure 3.5. Most behaviours include multiple phases; for instance, a deer will move its head towards the side of its body before grooming itself, and then it will begin to lick its fur. The hope is that splitting behaviours into successive samples can encode many different states of a behaviour. Furthermore, breaking down annotations into multiple smaller samples increases the size of the train and test sets.

3.2.4 Sample Cropping

Producing a sample consisting of frames cropped to the subject requires a series of frames each with a corresponding bounding box. As mentioned in section 3.1.2 bounding box information is stored in a `detections.json` file. Despite tuning the detection threshold to encourage object detection in areas of high occlusion and to discourage false detections, Megadetector still misses the occasional frame within a given behaviour. In order to deal with missing bounding boxes we skip frames where there is no corresponding bounding box. There are very few instances of missing bounding boxes and almost all incidents are isolated to a single bounding box, so replacing a frame by the next frame with a corresponding bounding box does not interfere with temporal encoding of samples. Individual frames from a sample are cropped during `init_dataset` using the crop functionality of the PIL library. Megadetector provides relative x and y coordinates for its bounding boxes which are converted to absolute values by multiplying the relative values by the width and height of the frame respectively.

3.2.5 Caching

Repeating the process of constructing a processed dataset for every training or evaluation run can increase the time required to make experiments. The `__init__` function is passed the keyword argument `cache_name` upon initialisation. The dataset checks whether a cache of this name is found in a pre-specified cache location, and if found it is loaded by calling `self.data = torch.load(cache_path)` where cache path is the path to the named cache file. Cache files are stored as `.pt` files in order to use the `torch.load` function to load the data efficiently. If no cache is found then the helper function `init_dataset` is called and once complete a new cache file is saved in the specified location by calling `torch.save(self.data, cache_path)`. Processing the dataset can take upwards of two hours on the GPU partition of The University of Bristol's Blue Crystal Phase 4 supercomputer, whereas loading a cache file takes less than five minutes, producing immediate and significant performance benefits.

3.2.6 Train-Test Split

Due to time constrains the number of annotations is smaller than ideal for a deep learning task. Due to this constraint, we do not distinguish between a validation and test set, as there would be too few samples in each set for evaluation to be meaningful. Instead, at every N given training epochs the model's performance is evaluated on the unseen test set. In order to gauge how well the model can generalise to unseen data different cameras are used for the train and test sets. Over optimisation of the test set is achieved by selecting between a small subset of possible hyper-parameter values. Learning rates and weight decay alphas are either increased or decreased by a power of ten in order to prevent highly specific hyper parameters from overfitting the model to the test set. A batch size of 8, expended to an effective batch size of 16 through gradient accumilation, is maintained throughout the project.

The subset of labelled cameras is comprised of the cameras T01, T02, T03, T09L, T09R, T28, T30L, and T30R. The test set is comprised of footage taken from T01, T02, T09L, T09R, and T30R, and the train set is comprised of footage from T03, T28, and T30L.

Partitioning the data into train and test sets based on location reveals the accuracy of the model when applied to new locations. This practice is known as Leave-One-Out partitioning when dealing with camera trap data and it is the standard for work in this field. The intention of the project is to produce a model that generalises to unseen environments, and as such is a useful factor when considering the potential effectiveness of the model being used by ecologists in the future. A model that is only effective in a predefined set of camera locations has little use for future ecology work so ensuring this seperation of locations is essential. As discussed in section a annotation can be split into multiple samples. Samples from the same annotation can often be highly similar, and stratifying samples from a single annotation across the train and test set can approximate to identical data within the train and test set. In fact, taking this approach the model quickly approaches a 90% test accuracy within a few epochs whilst avoiding this approach significantly diminishes this test accuracy. This suggests that, when stratifying samples from the same annotation across train and test sets, the model is learning specific sample details instead of a generalised understanding of behaviour. The most obvious solution would be to assign each sample to either the train or test set however partitioning by camera already achieves this partition whilst also interrogating the model's ability to generalise behaviours across locations.

Both the train and test sets feature a long tail class distribution after extracting all possible samples from the set of annotations. Figure 3.6 demonstrates that the relative frequency of different classes differs between train and test sets. This reflects the fact that different locations feature different patterns of behaviour from deer populations.

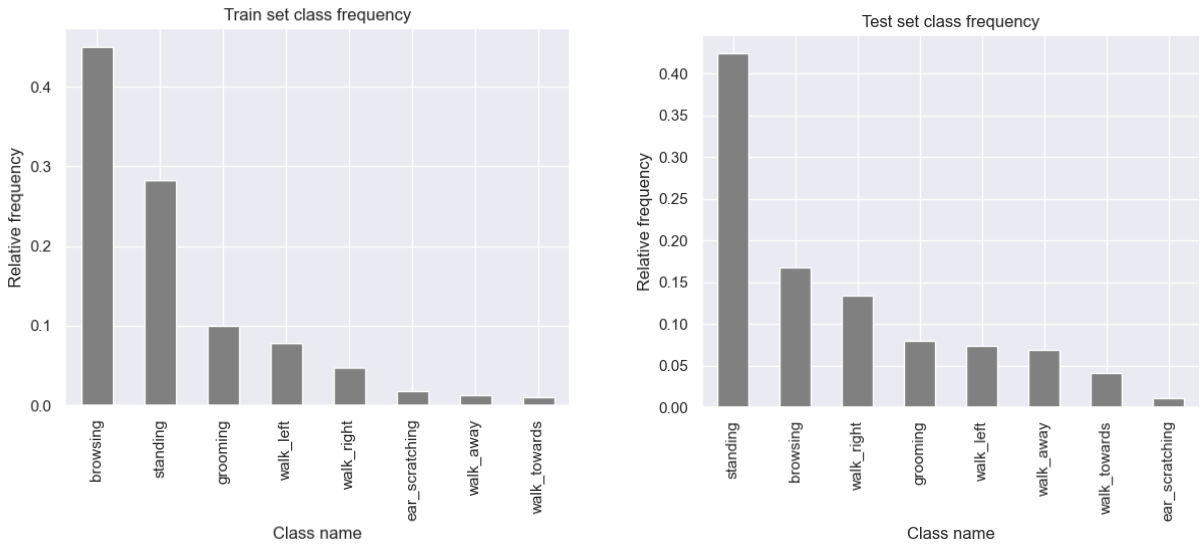

Figure 3.6: Visualisation of the class frequency in the train and test sets.

Section 3.7 will discuss steps taken to rectify the class imbalance during training. The differing class distribution is an asset during evaluation as the model is intended to be used in different camera locations

that may feature different frequencies of different behaviours.

3.3 Project Management

Good coding practices were followed during the project. The codebase follows the Python module structure, where different directories refer to different submodules, each responsible for a seperate aspect of a project. Python docstrings are used to document classes, methods, and utility functions throughout the codebase. Operations that are not obviously self documenting, such as reshaping a tensor, are accompanied by explanatory comments to help a reader understand the operations taking place.

3.3.1 User interface

All python files that should be directly run by an end user are made configurable using the `argparse` library. The library allows for the parsing of command-line options, arguments, and sub-commands when running a python file. This functionality is most useful when training a model due to the number of parameters that can be configured prior to training. Running any script with the flags `-h` or `--help` provides documentation as to the use of different command line arguments and what their default values are. Running the `train.py` and `eval.py` scripts with appropriate command line arguments is the intended way for a user to interact with the project's codebase.

```
usage: train.py [-h] [--video-root VIDEO_ROOT]
                [--annotation-root ANNOTATION_ROOT]
                [--log-dir LOG_DIR] [--fig-dir FIG_DIR]
                [--learning-rate LEARNING_RATE]
                [--l2-alpha L2_ALPHA]
                [--batch-size BATCH_SIZE] [--epochs EPOCHS]
                [--val-frequency VAL_FREQUENCY]
                [--log-frequency LOG_FREQUENCY]
                [--print-frequency PRINT_FREQUENCY]
                [-j WORKER_COUNT]

Train model

optional arguments:
  -h, --help
    show this help message and exit
  --video-root VIDEO_ROOT
    Path to a directory containing all data-set input information
  --annotation-root ANNOTATION_ROOT
    Path to a csv file containing annotations for the video data-
    set
  --log-dir LOG_DIR
    Directory to save tensorboard logs (default: tensorboard_logs)
  --fig-dir FIG_DIR
    Directory to save figures during training, such as confusion
    matrices (default: figs)
  --learning-rate LEARNING_RATE
    Learning rate (default: 5e-05)
  --l2-alpha L2_ALPHA
    L2 alpha (default: 1e-05)
  --batch-size BATCH_SIZE
    Number of videos within each mini-batch (default: 128)
  --epochs EPOCHS
    Number of epochs to train for (default: 200)
  --val-frequency VAL_FREQUENCY
    Frequency at which to test the model on the validation set in
    number of epochs (default: 2)
  --log-frequency LOG_FREQUENCY
    Frequency at which to save logs to tensorboard in number of
    steps (default: 10)
  --print-frequency PRINT_FREQUENCY
    Frequency at which to print progress to the command line in
    number of steps (default: 10)
  -j WORKER_COUNT, --worker-count WORKER_COUNT
    Number of worker processes used to load data. (default: 8)
```

Listing 3.1: Calling `train.py --help` lists all command line arguments, and provides an explanation for what each of them do and what their default value is, if one exists.

3.3.2 Version Control

Version control is achieved through Git, and the repository is hosted on GitHub. Git is helpful when tracking changes and rolling back the project if needed. Furthermore, hosting the repository on GitHub allows for the repository to be updated and run on different machines. The code base is kept identical between a local development machine and BC4 by pushing changes to GitHub and then pulling them to BC4. GitHub provides project management tools such as an integrated Kanban board which helped when managing feature developments and bug fixes in the code-base.

3.4 Models

The implementation of the models used within the project required taking into consideration how to instantiate back-bones, load pre-trained weights, freeze certain weights, arrange components into larger, more complex models,

3.4.1 Model Backbones

A backbone in deep learning refers to a section of a model that is concerned with feature extraction of input data into a given representation. When using an off-the-shelf model such as a Resnet, the final output layer must be replaced so that the number of output neurons reflects the class count. Everything before this final layer is referred to as a backbone. We test the effectiveness of two model backbones within this study, and then may combine these backbones as part of a larger network with the aim of achieving further performance improvements.

PyTorch provides the `torchvision.models.video` sub-module as part of the `torchvision` module. The module provides all operations necessary for the loading of off-the-shelf models despite the sub-module being in beta stage as of writing. The model provides implementations of popular video classification models, including MViT and ResNet, which are the models studied for this project.

3.4.2 Pretraining

PyTorch provides corresponding weights objects for its model implementations; for `MVIT_V2_S` the weights object is `MViT_V2_S_Weights` and for `r3d_18` the weights object is `R3D_18_Weights`. These weights are the result of training the models on the Kinetics-400 data-set, and instantiating the models with these corresponding weights constitutes pre-training.

The helper functions `fine_tune_resnet` and `fine_tune_mvit` help to easily instantiate the backbones of these models within a new model. The structure of the underlying model differs, so accessing and replacing the output layer of the models requires accessing different model components, so two different functions are used. The output layers are replaced by an `nn.Module` implementation called `Identity` which outputs the results of the previous layer. For `fine_tune_resnet`, the adaptive average pool layer is also replaced with `Identity` to allow for different fusion strategies to be taken for a model consisting of two convolutional streams. Both functions return the backbone model and the number of features produced by the backbone, which will need to match the number of input features for a layer after the backbone.

```
resnets = {
    "r2plus1d_18": models.video.r2plus1d_18,
    "r3d_18": models.video.r3d_18,
    "mc3_18": models.video.mc3_18,
}


class Identity(nn.Module):
    """
    Representation of the identity operation as a PyTorch `nn.Module` class.
    """

    def __init__(self):
        super(Identity, self).__init__()

    def forward(self, x):
        return x
```

```
def fine_tune_resnet(feature_extract: bool, pretrained=False, resnet_type="r3d_18"):
    """
    Parameters
    ----------
    feature_extract : bool
        If true then all model weights are frozen
    pretrained :
        If true then instantiate model with default pretrained weights
    """
    model = resnets[resnet_type](weights="DEFAULT" if pretrained else None)
    set_parameter_requires_grad(model, feature_extract)
    num_ftrs = model.fc.in_features
    model.avgpool = Identity()
    model.fc = Identity()
    return model, num_ftrs


def fine_tune_mvit(feature_extract, pretrained=False):
    """
    Parameters
    ----------
    feature_extract : bool
        If true then all model weights are frozen
    pretrained :
        If true then instantiate model with default pretrained weights
    """
    model = mvit_v2_s(weights=MViT_V2_S_Weights.DEFAULT if pretrained else None)
    set_parameter_requires_grad(model, feature_extract)
    num_ftrs = model.head[1].in_features
    model.head = Identity()
    return model, num_ftrs


def set_parameter_requires_grad(model, feature_extracting):
    """
    Parameters
    ----------
    model : nn.Module
        Module which will have its weights adjusted
    feature_extracting :
        If true then all model weights are frozen
    """
    if feature_extracting:
        for param in model.parameters():
            param.requires_grad = False
```

3.4.3 Fusion

Fusion of two CNNs is attempted in three seperate ways in this project. Figure 3.7 visualises the different ways of approaching the fusion of two 3D CNN networks.

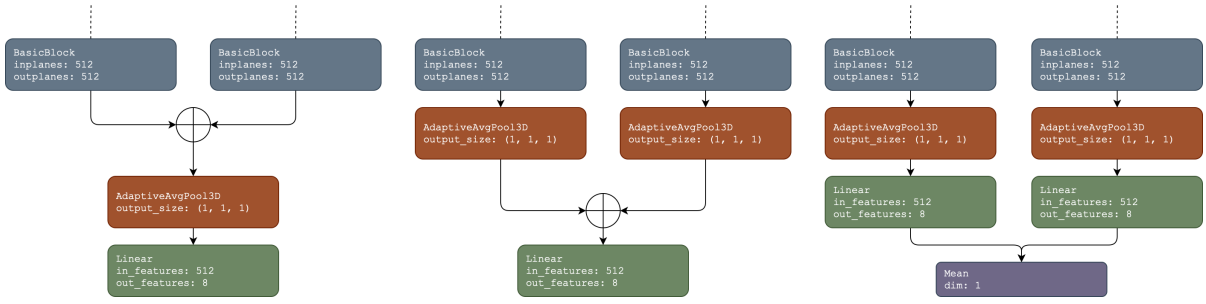

Figure 3.7: Three separate fusion methods are used to combine CNN networks. The first method "StackFuse" (Left) stacks the convolutional feature maps from both streams before a global average pool layer. The second method "AdaptiveFuse" (centre) applies a global average pool to each stream and then concatenates the two channels together. The third method "AvgFuse" (right) averages the logit values for the output layer of both networks to produce the final logit values.

All three forms of fusion are examples of late fusion, where features are combined near the end of the CNN. MViT V2 networks are combined by concatenating the 768 features produced by the MViT V2 backbone. The resulting 1,536 features are processed by a linear output layer, which constitutes the model output.

3.5 Optical Flow Frames

Optical Flow Frames are constructed by combining the horizontal displacement frame with the corresponding vertical displacement frame and an empty frame of 0 values. This produces a tensor with 3 channels, which matches that of the RGB frames. The stems for both the ResNet models and MViT expect 3 channel input data. Altering the number of channels in the stem risks breaking transfer learning as either the model weights would have to be removed from the stem or only some model weights would be kept.

3.6 Training and Evaluation

The Python script `train.py` is responsible for training the behaviour recognition model. As mentioned in section 3.3.1, command line arguments are used to specify the configuration of training parameters. Once the model, optimiser, loss function, and all other required components have been initialised the `train.py` script instantiates the `Trainer` class. The class method `train` is then used to begin the training loop. The method is responsible for all functionality that takes place once the training process has begin, including optimising model parameters, logging metrics, and validating model performance.

Models are trained for 200 epochs. Validation metrics stop improving at around 100 epochs and remain flat after this point. All metrics recorded during the project correspond the the last evaluation of the model on the validation set during training.

An epoch involves training the model on the entire training set. Mini-batches are used for training, which reduces the amount of video memory required to train the model. All training data and labels from a batch are moved to video memory when training the model on a mini-batch. This allows for CUDA performance acceleration, which improves training performance. The command line arguments specify the frequency at which validation is performed. A validation frequency of five means that the model's performance will be evaluated on the unseen dataset every five epochs. This allows for the model's performance on unseen data to be tracked during training. The model is not trained on this unseen data, and as such no back-propagation takes place during this phase.

3.6.1 Gradient Accumulation

Gradient accumulation [9] is employed in order to train the model on larger batch sizes than would normally fit in video memory. Rather than optimise the model every time a batch is processed, gradients are accumulated every batch for a given number of batches, and then only that point is the optimisation step taken. Thus, batch sizes are effectively multiplied by the number of batches processed before the optimisation step is taken, which can be referred to as the accumulation frequency. For this project a batch size of 8 is used alongside an accumulation frequency of 2, leading to a virtual batch size of 16. A batch size of 16 could not be used during the project due to virtual memory limitations so gradient accumulation was employed in order to artificially increase the batch size.

```
...
logits = self.model(batch_rgb_cropped, batch_flow_cropped)
loss = self.criterion(logits, labels)
loss = loss / accumulation_steps
loss.backward()

if (i + 1) % accumulation_steps == 0:
    self.optimizer.step()
    self.optimizer.zero_grad()
...
```

The above code snippet, which is run every time the model is trained on a mini-batch, demonstrates how gradient accumulation can be achieved in PyTorch.

3.6.2 Logging

Model performance metrics are collected during and after training. These metrics are Top1 accuracy, Top3 accuracy, classification loss, and average class accuracy. Per-class accuracy is also completed. These metrics are calculated for a given batch frequency on the training set and for every evaluation of the test set. At a user specified interval confusion matrices are saved to a pre-defined sub-directory. These confusion matrices are calculated based on the performance of the model on the test set. The Top1 accuracy and classification loss scores for the performance on the train and test set are logged to TensorBoard at a regular interval. The TensorBoard dashboard allows for these metrics to be visualised during training, which can reveal properties such as over-fitting, where the validation loss curve stops improving whilst the training loss curve keeps improving.

3.7 Balanced Sampling

Class imbalance was identified early on in the project as a potential hindrance to model performance. Much of the research covered in section 1.3 employed some form of class re-balancing in order to improve model performance. The frequency of deer behaviour in the dataset follows a long-tail distribution. This means that during training, tail classes appear less frequently in batches than more common classes. Thus, the model is not trained on enough samples in order for it to learn the properties of less common classes, resulting in lower accuracy for these classes.

A PyTorch `Sampler` class can be passed to a `Dataloader` in order to control the manner in which specific samples are sampled from the dataset. The `Sampler` implementation `WeightedRandomSampler` takes a list of weights, each corresponding to a sample in the dataset. The weight corresponds to the probability of that particular sample being sampled by the `Dataloader`. A `WeightedRandomSampler` can be employed to ensure that all classes within a dataset have an equal probability of being sampled during training.

```
def balanced_wrsampler(dataset):
    labels = np.array([sample["target"] for sample in dataset])
    class_count = np.array([len(np.where(labels == t)[0]) for t in np.unique(labels)])
    weight = 1.0 / class_count
    samples_weight = np.array([weight[t] for t in labels])
    samples_weight = torch.from_numpy(samples_weight)
    samples_weight = samples_weight.double()
    sampler = WeightedRandomSampler(samples_weight, len(samples_weight))
    return sampler
```

The above code snippet shows the implementation of the helper function `balanced_wrsampler`, written for this project. The function first counts the frequency of each class within the dataset. The function then assigns a weight to each class equal to the inverse of the frequency of each class within the dataset. This class weight is used as the probability weight for each sample belonging to the class. Finally, this list of sample weights is provided to the `WeightedRandomSampler` to produce a sampler that gives an equal probability of sampling a sample belonging any class in the dataset.

Chapter 4

Critical Evaluation

This chapter lays out the results of the different deer behaviour recognition models. Performance is evaluated across different data streams. The chapter will also articulate how different sets of results guided the choice of a final model architecture.

4.1 Single Stream Comparison

It is important to compare the efficacy of the two potential model backbones under consideration in the project. The two backbones networks are each constructed as stand-alone models incorporating a single input stream at a time, as discussed in section 3.4.1. This allows for their performance as a spatial or temporal network to be independently analysed. The final model will involve combining the best performing spatial and temporal networks.

All three model architectures achieved learning on input data to a certain degree, with both achieving a class average above 12.5% which would represent the expected average class accuracy for a model randomly predicting test set labels over 8 possible classes. All possible configurations of models and input streams obtain a Top1 accuracy over 50% and a Top3 accuracy over 75%, demonstrating that the models are able to extract meaningful behaviour representation from video information. However, class average significantly lags behind the two other metrics. This is to be expected for a data-set with a long tail class distribution, as the majority of the accuracy score comes from achieving very high accuracies on the most common classes.

Input Stream	Model	Top1	Top3	Class Average	Average Step Time
RGB	3D Resnet-18	**64.44%**	86.62%	**42.96%**	**0.318s**
	(2+1)D Resnet-18	52.42%	87.11%	37.41%	0.491s
	MViT V2	56.26%	**90.21%**	36.70%	1.225s
Optical flow	3D Resnet-18	**59.98%**	84.27%	**40.96%**	**0.318s**
	(2+1)D Resnet-18	58.12%	**84.51%**	40.15%	0.491s
	MViT V2	51.55%	79.06%	31.44%	1.225s

Table 4.1: Comparison between 3D ResNet-18, (2+1)D ResNet-18, and MViT V2 architectures, applied to RGB and Optical Flow streams. 3D Resnet-18 outperforms both other networks on Top1 accuracy and average class accuracy across both streams.

The results demonstrate that in this instance 3D Resnet-18 is better suited to the task of Roe Deer behaviour action recognition. This result is not in line with the current literature on the relative performance of the different networks; MViT V2 achieves an 80.76% Top1 accuracy on the Kinetics-400 pre-training dataset, significantly higher than the 67.46% and 63.2% Top1 accuracy scores achieved respectively by the (2+1)D ResNet-18 and 3D ResNet-18 architectures on the same dataset. The video data features different properties compared to the Kinetics-400 dataset, as mentioned in . 3D Resnet-18 has a lower average step time of 0.318s compared to the 0.491s and 1.225s step times for the two other networks. Thus, opting for 3D Resnet-18 over the other two networks also results in faster model training that thus faster experiment iteration.

4.2 Pretraining Evaluation

Both input streams made use of pre-training when comparing different model architectures. In this section we examine the potential performance benefits of loading pre-trained weights before training on the 3D Resnet-18 architecture. "Pre-trained" transfer learning refers to loading pre-trained weights and then training all model parameters on the new set. "Feature Extraction" refers to loading pre-trained weights and freezing all model parameters, except for output layer parameters, before training. We compare both approaches to instantiating random weights as opposed to pre-training.

Input Stream	Transfer learning	Top1	Top3	Class Average
RGB	None	50.56%	77.20%	30.63%
	Pre-trained	**64.44%**	**86.62%**	**42.96%**
	Feature Extraction	31.10%	82.03%	25.26
Optical flow	None	47.46%	78.19%	30.17%
	Pre-trained	**59.95%**	**84.27%**	**40.96%**
	Feature Extraction	47.96%	80.80%	24.66%

Table 4.2: Comparison between transfer learning strategy on RGB and Optical Flow input streams for the 3D Resnet-18 model. Pre-training results in significant performance gains, especially in the RGB stream.

The results demonstrate that pre-training provides significant performance benefits for both input streams. Feature extraction performs worse than random weight initialisation, suggesting that in order for pre-trained weights to be effective all parameters must be updated during training. This suggests that the features needed for the effective recognition of deer behaviour deviate compared to features generated for human action recognition from the Kinetics-400 data-set. However, the fact that loading pre-trained weights performs better than randomly initialised weights suggests that the features produced by the pre-trained model provide a good foundation for further training to take place. The task of deer behaviour recognition therefore needs specialised features to be learnt that are not present when using those generated by a model trained on Kinetics-400.

4.3 Fusion

The three convolutional stream fusion approaches covered in section 3.4.3 are compared for a two stream model consisting of two 3D Resnet-18 streams.

Fusion mechanism	Top1	Top3	Class Average
StackFuse	**66.67%**	89.55%	**40.21%**
AdaptiveFuse	65.18%	**90.46%**	38.93%
AvgFuse	65.55%	90.45%	39.14%

Table 4.3: Comparison between three different fusion techniques. The results suggest that the "StackFuse" approach performs best on this dataset.

The "StackFuse" approach refers to the stacking of convolutional feature maps from both streams before being applied to a global average pooling layer, which is then fed to an output layer. The "AdaptiveFuse" technique refers to the use of a global average pooling layer by both streams independently, the output features of which are then flattened and concatenated before being processed by the output layer. The "AvgFuse" approach consists of taking the element-wise mean of the output layer logits for each stream as the network output.

Testing suggests that the "StackFuse" approach is most effective, scoring the highest Top1 and Class Average scores, even if the "AdaptiveFuse" approach produces better Top3 scores. All three fusion methods perform better in terms of Top1 accuracy compared to a single stream model, demonstrating the performance advantages afforded by capturing both spatial and temporal dependencies within a single model architecture.

4.4 Initial Model

The initial model consists of a two-stream architecture using two pre-trained 3D ResNet-18 networks, using a fusion technique ("StackFuse") that combines convolutional feature maps before a global average pool layer and finally the output layer. This model has been has the highest Top1 accuracy score out of all the models attempted up to this point in the project.

Results on Unseen Data		
Top1	Top3	Class Average
66.67%	87.36%	42.70%

Table 4.4: Initial model performance. Performance is calculated by the final evaluation of the model on the test set during training

Action	browsing	ear_scratching	grooming	standing	w_away	w_left	w_right	w_towards
Top1	94.78%	0%	53.73%	78.27%	3.65%	36.54%	71.29%	3.33%

Table 4.5: Per-class Top1 accuracy scores

The model's Top1 is greater than any of the previously attempted models in this study. The model performs particular well on the classes browsing and standing, achieving over than 80% accuracy on each. The model performs poorly on walk_away and walk_towards classes, achieveing less than 10% on each. The model achieves 0% accuarcy on the ear scratching class, which can be attributed to its rarity.

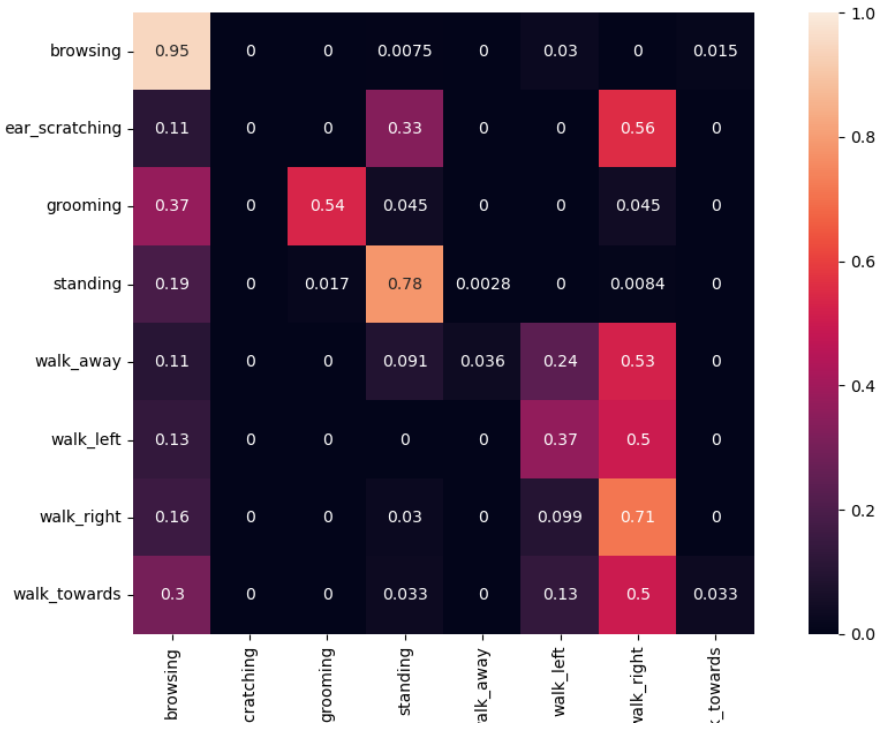

Figure 4.1: The confusion matrix of the initial model's evaluation on the test set.

4.5 Balanced Training Set

Using a balanced weighted random sampler improves all model metrics. The behaviours grooming, standing, and walk_right increase in accuracy.

Results on Unseen Data		
Top1	Top3	Class Average
72.99%	89.21%	46.86%

Table 4.6: Final validation scores taken during training for the balanced model.

The overall class average is increased from the previous model. The confusion matrix below shows that whilst the same patterns of false positives exist within both training frameworks using a balanced sampling approach can mitigate many of these issues.

	browsing	ear_scratching	grooming	standing
Top1	88.06% -6.72	0%	76.12% +22.39	89.14% +10.87

	walk_away	walk_left	walk_right	walk_towards
Top1	7.27% -3.62	34.32% -2.22	76.24% + 4.95	3.33% +0

Table 4.7: Per-class accuracy scores for the final model. The table is decomposed into two layers in order to fit onto the page

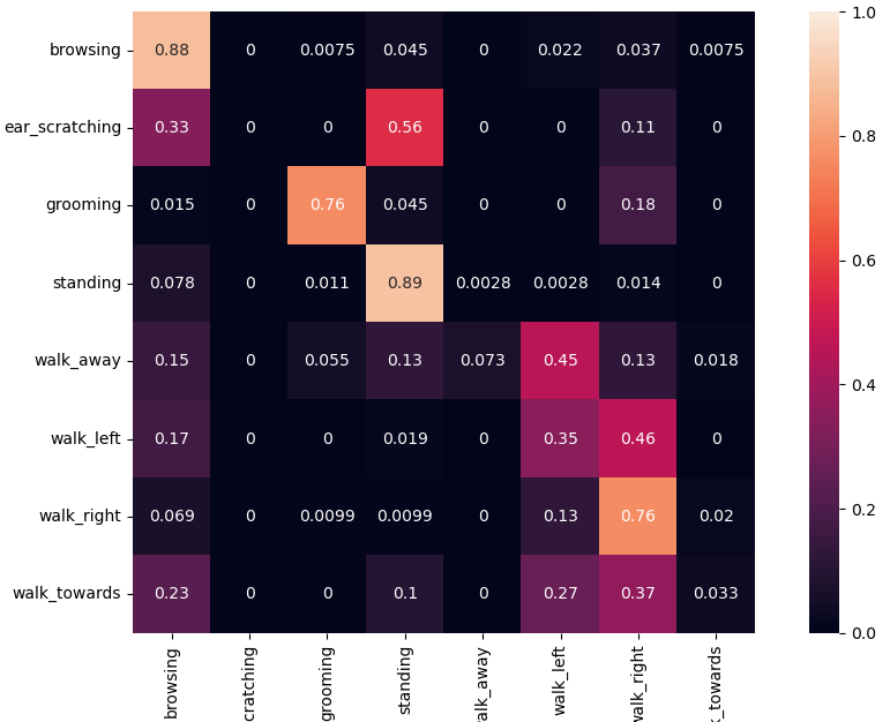

Figure 4.2: The confusion matrix of the final model's evaluation on the test set.

4.6 Final Model

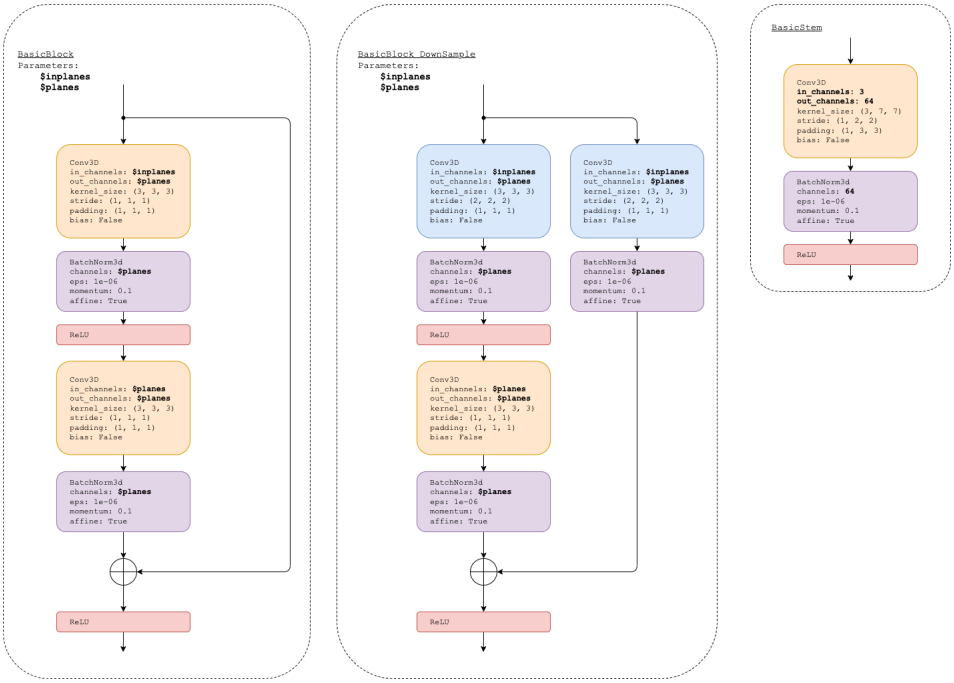

Figure 4.3: The components constituting the final model of the project

This above diagram demonstrates the different residual components used to create the final model for the project. The structure of the network is a ResNet however it is helpful to get an overview of all of the moving parts within the final network. The below diagram provides an overview of the structure of the network as a whole, and shows how fusion is achieved for the final network.

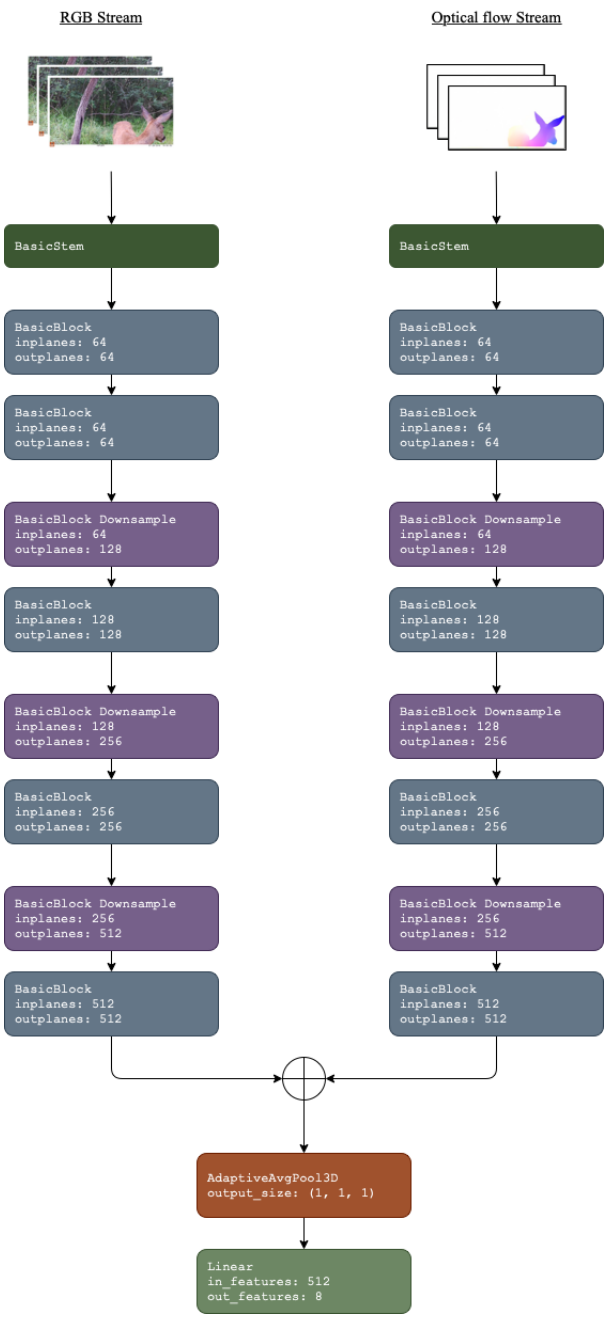

Figure 4.4: The entire network architecture of the final model of the project

4.7 Final Results

The final project results are demonstrated below. This provides a comprehensive overview of all experiments taken during the course of this project.

Input Stream	Backbone	Transfer Learning	Fusion	Top1	Top3	Class Average	Sampling
RGB	3D R-18	Pre-trained	n/a	64.44%	86.62%	42.96%	Imbalanced
RGB	(2+1)D R-18	Pre-trained	n/a	52.42%	87.11%	37.41%	Imbalanced
RGB	MViT V2	Pre-trained	n/a	51.55%	79.06%	31.44%	Imbalanced
Optical flow	3D R-18	Pre-trained	n/a	59.98%	84.27%	40.96%	Imbalanced
Optical flow	(2+1)D R-18	Pre-trained	n/a	58.12%	84.51%	40.15%	Imbalanced
Optical flow	MViT V2	Pre-trained	n/a	51.55%	79.06%	31.44%	Imbalanced
RGB	3D R-18	None	n/a	50.56%	77.20%	30.63%	Imbalanced
RGB	3D R-18	Feature Extraction	n/a	31.10%	82.03%	25.26%	Imbalanced
Optical flow	3D R-18	None	n/a	47.46%	78.19%	30.17%	Imbalanced
Optical flow	3D R-18	Feature Extraction	n/a	47.96%	80.80%	24.66%	Imbalanced
RGB + Flow	3D R-18	Pre-trained	StackFuse	66.67	89.55%	40.21%	Imbalanced
RGB + Flow	3D R-18	Pre-trained	AdaptiveFuse	65.18%	90.46%	38.93%	Imbalanced
RGB + Flow	3D R-18	Pre-trained	AvgFuse	65.55%	90.45%	39.14%	Imbalanced
RGB + Flow	3D R-18	Pre-trained	StackFuse	72.99%	89.21%	46.86%	Balanced

Chapter 5

Conclusion

Research into the development of models designed for tasks of behavioural action recognition of non-human subjects has been gaining traction. However, such a study has not been conducted in deer populations. This project has attempted to develop and evaluate deep learning techniques and architectures within the context of learning deer behaviours. Pre-existing behavioural action recognition work often focuses on the task of identifying a single behaviour from a video clip, and if multiple subjects are present within the video then all subjects are expected to be performing the same action. The properties of the dataset required that a model capable of identifying deer behaviour would need to be able to identify behaviour on a per-individual basis and take into account the possibility of an individual performing more than one behaviour within a video. Computer vision tasks relating to the natural world often have to contend with a number of factors that can impede model performance, such as complex and non-static backgrounds, partial occlusion of the behaviour, and individual behavioural differences. Thus, the project required the development of a model that could contend with the specialised requirements associated with the dataset and the issues found more broadly within computer vision tasks involving the natural world.

Results on Unseen Data		
Top1	Top3	Class Average
72.99%	89.21%	46.86%

Table 5.1: Final Model Performance. The model achieves 72.99% Top1 accuracy, 89.21% Top3 accuracy, and a 46.86% class average performance on the test set

A number of separate tasks had to be undertaken in order for the successful completion of the project. A subset of all relevant videos in the dataset was annotated, eventually constituting more than 50,000 labelled behaviour frames. This produced a dataset that could be used to successfully attempt and explore different approaches within deep learning to recognising deer behaviour.

The project initially explored the comparative advantages provided by three separate pre-existing video recognition architectures: 3D Resnet-18, (2+1)D Resnet-18, and MViT V2. A two-stream approach was then adopted with the aim to use the best performing architecture for the spatial and temporal streams. 3D Resnet-18 produced the best Top1 accuracy scores on both RGB and optical flow streams and was adopted as the achitecture for each stream within the two stream model. Three different appraoches to stream fusion were examined, with a method employing the concatenation of feature maps before global pooling adopted.

Heavy class imbalance was noted during the labelling phase of the task. Convolution matrices revealed that the model was making a significant number of false positive predictions for the most common classes. A balanced weighted random sampling approach was adopted, which partially dealt with the issue. This revealed which classes were suffering poor performance scores purely due to their lack of frequent enough sapling, and which classes were suffering poor performance for other reasons. A variety of different appearances and motions that can represent a behaviour class. The size of the labelled subsection of the dataset is comparatively small due to the significant time commitment required for a student to manually annotate each frame, possibly comprising multiple deer and thus multiple samples, of hours of footage. A

significant amount of time was invested into the labelling of the data-set however the fact that a behaviour can appear very differently due to a variety of different factors resulted in some behaviours generalising poorly whilst others generalised well. Static behaviours performed far better than behaviours involving motion. The fact the model could differentiate between a deer feeding, grooming itself, and standing, to a high degree of accuracy, demonstrates the potential of such a model to be a useful tool for ecologists in the future, especially when applied to a larger dataset.

The project's objectives have been largely met. A subset of the Brandenburg video archive has been labelled, although a larger subset of the video archive would allow for greater insights into training the model on tail classes. The performance of different video classification backbones has been evaluated, and a novel domain specific model has been constructed for the project. Individual class scores and confusion matrices have provided insights into where and why the model performs well, and where and why it does not.

5.1 Future Work

Training a multi class classification model would allow the use of a taxonomy that more accurately fits the actual behaviour of a deer. Deer can often be found taking more than one action at once, or in quick succession, and a model with the ability to encode these overlapping behaviours could prove beneficial. Much of the issues with regards to classification in this problem can be attributed to the fact that behaviours are often ambiguous to the human eye so best guesses have to be made. Labelling multiple concurrent behaviours can more accurately represent a deer's behaviour to a non-expert.

A visualisation could be produced around the model in order to visualise and label the output of the model on fresh data. The model would be continuously applied on the last 16 frames of video data and as such would be able to continuously update its behaviour predictions for each given deer. Such a system could be used to batch label large quantities of data by outputting predictions per frame to a CSV file. This would further realise the aim of this project which was to produce a model that could automate labourious and time consuming tasks.

Bibliography

[1] Md Zahangir Alom, Tarek M Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S Awwal, and Vijayan K Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *arXiv preprint arXiv:1803.01164*, 2018.

[2] Anurag Arnab, Mostafa Dehghani, Georg Heigold, Chen Sun, Mario Lučić, and Cordelia Schmid. Vivit: A video vision transformer. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6836–6846, 2021.

[3] Pierre Baldi and Peter J Sadowski. Understanding dropout. *Advances in neural information processing systems*, 26, 2013.

[4] Otto Brookes, Majid Mirmehdi, Hjalmar Kühl, and Tilo Burghardt. Triple-stream deep metric learning of great ape behavioural actions, 2023.

[5] Joao Carreira and Andrew Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[6] Ruilong Chen, Ruth Little, Lyudmila Mihaylova, Richard Delahay, and Ruth Cox. Wildlife surveillance using deep learning methods. *Ecology and evolution*, 9(17):9453–9466, 2019.

[7] Alekseĭ Danilkin and AJ Mark Hewison. *Behavioural ecology of Siberian and European roe deer*. Springer, 1996.

[8] H Kuehl et al. Brandenburg dataset - German Centre for Integrative Biodiversity Research (iDiv).

[9] Hugging Face. Performance and scalability: How to fit a bigger model and train it faster. https://huggingface.co/docs/transformers/v4.18.0/en/performance.

[10] Haoqi Fan, Bo Xiong, Karttikeya Mangalam, Yanghao Li, Zhicheng Yan, Jitendra Malik, and Christoph Feichtenhofer. Multiscale vision transformers. *CoRR*, abs/2104.11227, 2021.

[11] Margarita Favorskaya and Andrey Pakhirka. Animal species recognition in the wildlife based on muzzle and shape features using joint cnn. *Procedia Computer Science*, 159:933–942, 2019.

[12] Liqi Feng, Yaqin Zhao, Yichao Sun, Wenxuan Zhao, and Jiaxi Tang. Action recognition using a spatial-temporal network for wild felines. *Animals*, 11(2):485, 2021.

[13] Yu Fu, Liuyu Xiang, Yumna Zahid, Guiguang Ding, Tao Mei, Qiang Shen, and Jungong Han. Long-tailed visual recognition with deep models: A methodological survey and evaluation. *Neurocomputing*, 2022.

[14] Alvaro Fuentes, Sook Yoon, Jongbin Park, and Dong Sun Park. Deep learning-based hierarchical cattle behavior recognition with spatio-temporal information. *Computers and Electronics in Agriculture*, 177:105627, 2020.

[15] Git. Git documentation. https://git-scm.com/doc.

[16] GitHub. GitHub homepage. https://github.com.

[17] Timm Haucke, Hjalmar S Kühl, Jacqueline Hoyer, and Volker Steinhage. Overcoming the distance estimation bottleneck in estimating animal abundance with camera traps. *Ecological Informatics*, 68:101536, 2022.

[18] Samitha Herath, Mehrtash Harandi, and Fatih Porikli. Going deeper into action recognition: A survey. *Image and Vision Computing*, 60:4–21, 2017.

[19] Ivan Himawan, Michael Towsey, Bradley Law, and Paul Roe. Deep learning techniques for koala activity detection. In *INTERSPEECH*, pages 2107–2111, 2018.

[20] Thomas Huang. Computer vision: Evolution and promise. 1996.

[21] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[22] Volker Krüger, Danica Kragic, Aleš Ude, and Christopher Geib. The meaning of action: A review on action recognition and mapping. *Advanced robotics*, 21(13):1473–1501, 2007.

[23] Hjalmar S Kühl and Tilo Burghardt. Animal biometrics: quantifying and detecting phenotypic appearance. *Trends in ecology & evolution*, 28(7):432–441, 2013.

[24] Santosh Kumar and Sanjay Kumar Singh. Visual animal biometrics: survey. *IET Biometrics*, 6(3):139–156, 2017.

[25] Sharon Levy. A plague of deer. *BioScience*, 56(9):718–721, 2006.

[26] Dong Liu, Maciej Oczak, Kristina Maschat, Johannes Baumgartner, Bernadette Pletzer, Dongjian He, and Tomas Norton. A computer vision-based method for spatial-temporal action recognition of tail-biting behaviour in group-housed pigs. *Biosystems Engineering*, 195:27–41, 2020.

[27] Agnes Lydia and Sagayaraj Francis. Adagrad—an optimizer for stochastic gradient descent. *Int. J. Inf. Comput. Sci*, 6(5):566–568, 2019.

[28] Behnam Neyshabur, Hanie Sedghi, and Chiyuan Zhang. What is being transferred in transfer learning? In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 512–523. Curran Associates, Inc., 2020.

[29] Xun Long Ng, Kian Eng Ong, Qichen Zheng, Yun Ni, Si Yong Yeo, and Jun Liu. Animal kingdom: A large and diverse dataset for animal behavior understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 19023–19034, 2022.

[30] Mohammad Sadegh Norouzzadeh, Anh Nguyen, Margaret Kosmala, Alexandra Swanson, Meredith S Palmer, Craig Packer, and Jeff Clune. Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning. *Proceedings of the National Academy of Sciences*, 115(25):E5716–E5725, 2018.

[31] NVIDIA. CUDA Python documentation. https://nvidia.github.io/cuda-python/overview.html.

[32] University of Bristol Advanced Computing Research Centre. Blue crystal phase 4. https://www.acrc.bris.ac.uk/acrc/index.htm.

[33] OpenCV. OpenCV documentation. https://docs.opencv.org/4.7.0/.

[34] Pillow. Pillow (PIL Fork) documentation. https://pillow.readthedocs.io/en/stable/.

[35] PyTorch. PyTorch documentation. https://pytorch.org/docs/stable/index.html.

[36] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[37] Faizaan Sakib and Tilo Burghardt. Visual recognition of great ape behaviours in the wild. *CoRR*, abs/2011.10759, 2020.

[38] Javier Selva, Anders Skaarup Johansen, Sergio Escalera, Kamal Nasrollahi, Thomas B. Moeslund, and Albert Clapés. Video transformers: A survey. *CoRR*, abs/2201.05991, 2022.

[39] Laura Sevilla-Lara, Yiyi Liao, Fatma Güney, Varun Jampani, Andreas Geiger, and Michael J Black. On the integration of optical flow and action recognition. In *Pattern Recognition: 40th German Conference, GCPR 2018, Stuttgart, Germany, October 9-12, 2018, Proceedings 40*, pages 281–297. Springer, 2019.

[40] Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos. *Advances in neural information processing systems*, 27, 2014.

[41] Irwin Sobel. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, 02 2014.

[42] TensorFlow. TensorBoard: Tensorflow's visualization toolkit. https://www.tensorflow.org/tensorboard.

[43] Du Tran, Heng Wang, Lorenzo Torresani, Jamie Ray, Yann LeCun, and Manohar Paluri. A closer look at spatiotemporal convolutions for action recognition, 2018.

[44] Devis Tuia, Benjamin Kellenberger, Sara Beery, Blair R. Costelloe, Silvia Zuffi, Benjamin Risse, Alexander Mathis, Mackenzie W. Mathis, Frank van Langevelde, Tilo Burghardt, Roland Kays, Holger Klinck, Martin Wikelski, Iain D. Couzin, Grant van Horn, Margaret C. Crofoot, Charles V. Stewart, and Tanya Berger-Wolf. Perspectives in machine learning for wildlife conservation. *Nature Communications*, 13(1):792, 2022.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[46] Yizhen Wang, Yang Zhang, Yuan Feng, and Yi Shang. Deep learning methods for animal counting in camera trap images. In *2022 IEEE 34th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 939–943. IEEE, 2022.

[47] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):1–40, 2016.

[48] Guangle Yao, Tao Lei, and Jiandan Zhong. A review of convolutional-neural-network-based action recognition. *Pattern Recognition Letters*, 118:14–22, 2019.

[49] Wei Zhang, Kazuyoshi Itoh, Jun Tanida, and Yoshiki Ichioka. Parallel distributed processing model with local space-invariant interconnections and its optical architecture. *Applied optics*, 29(32):4790–4797, 1990.

[50] Zhengxia Zou, Keyan Chen, Zhenwei Shi, Yuhong Guo, and Jieping Ye. Object detection in 20 years: A survey. *Proceedings of the IEEE*, 2023.