UiT THE ARCTIC UNIVERSITY OF NORWAY

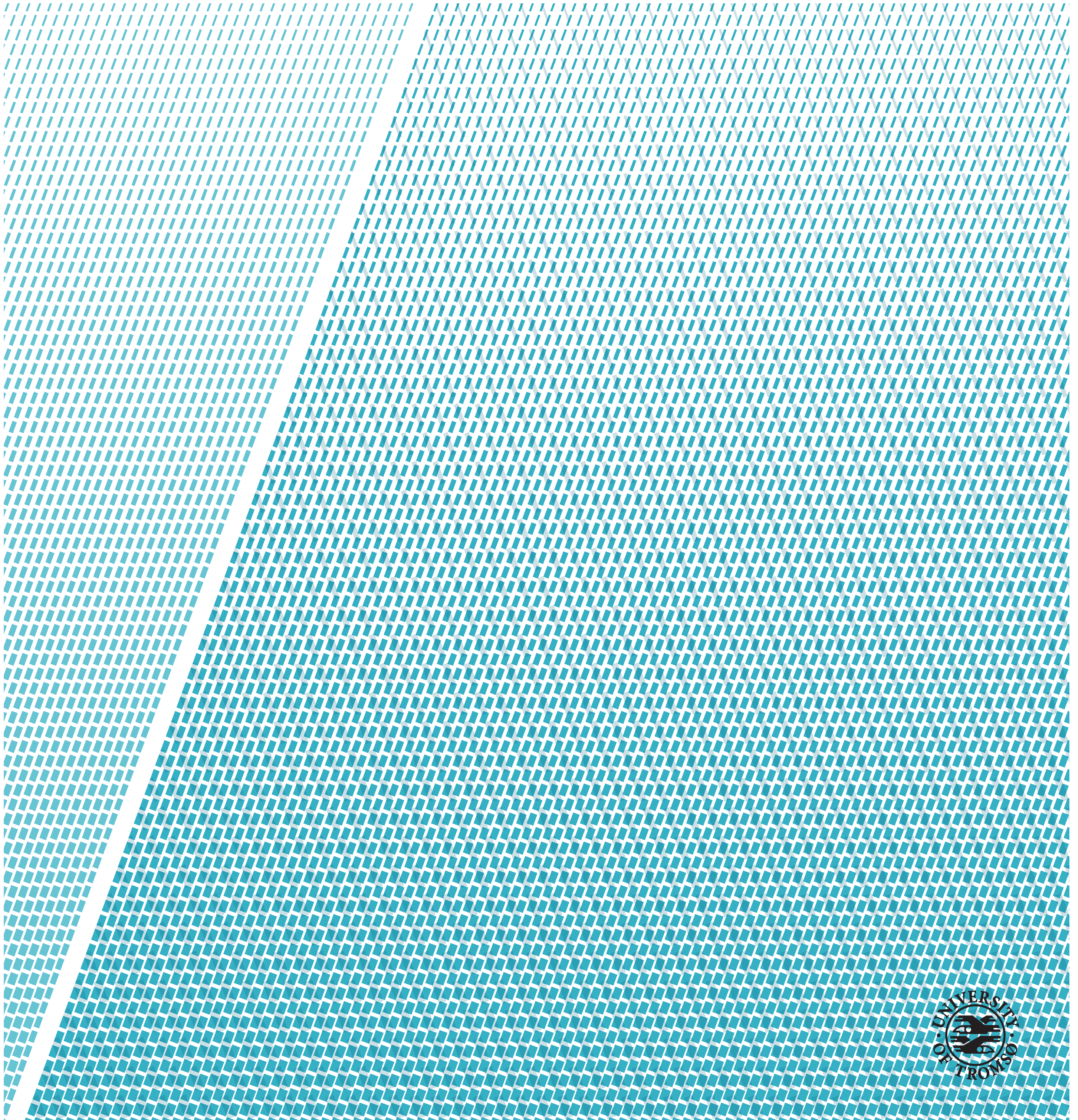
Faculty of Science and Technology
Department of Computer Science

Embedded Analytics of Animal Images

—

Sigurd Thomassen

INF-3981 Master thesis in Computer Science, December 2017



This thesis document was typeset using the *UiT Thesis LaTeX Template*.
© 2017 – http://github.com/egraff/uit-thesis

To Jesus.

Thank you very much.

“SIGURD... (͡° ͜ʖ ͡°)”
–Birgitte

“Rockets are cool. There’s no getting around that.”
–Elon Musk

Abstract

Due to the large increase of image data in animal surveillance, an effective and efficient way of labeling said data is required. Over the past few years the Climate-ecological Observatory for Arctic Tundra (COAT) project have deployed dozens of cameras in eastern Finnmark, Norway during winter, which have resulted in a large volume of wildlife images which is used to document the effects of climate change on animal ecosystems in the area. The images are manually labeled by biologists, and is a time-consuming task.

This thesis presents the architecture, design and implementation of an image classification system to be used with the camera traps for *in-situ* analytics on accumulated image data for periodical updates. The system will automatically classify and label the images taken by the cameras.

Using state-of-the-art Convolutional Neural Networks (CNNs) we train the system on previously labeled COAT image data. We train four different models based on the MobileNet architecture. The models vary in number of weights, and input image resolution.

Results show that we can automatically classify images on a small computer like the Raspberry Pi, with an accuracy of 81.1% at 1.17 FPS, and a model size of 17Mb. In comparison a GPU computer achieves the same accuracy and model size, but it has a classification speed of 12.5 FPS.

Acknowledgements

First I would like to thank my main advisor Professor Otto Anshus, and co-advisor Associate professor John Markus Bjørndalen for your great advices, ideas and feedback whenever I needed it throughout the work on this thesis.

Then I would like to thank the Department of Computer Science, its technical and administrative staff for support when needed. I would also like to thank the person(Professor Otto) who came up with the glorious idea of having an espresso machine stationed right outside my office! Thank you!

Furthermore I want express my sincerest gratitude to the *Masterinos*. Without you, I'm not sure I would have finished.

I would also like to thank my parents for encouraging me to take a higher education, and supporting me the whole way.

Most of all, thank you to my wife Ane. You have been my biggest motivation the past 5 years, and you are a great role model.

Finally I would like to express my appreciation for my best friend Tobias who have been a good sparring partner in discussions regarding the thesis, and a great coffee drinker!

And to my secretary Camilla. Here's your special mention.

Contents

Abstract **iii**

Acknowledgements **v**

List of Figures **xi**

List of Tables **xiii**

List of Abbreviations **xv**

1 Introduction **1**
1.1 Problem definition 2
1.2 Contributions 3
1.3 Outline 4

2 Image Classification **5**
2.1 Convolutional Neural Networks (CNNs) 5
2.1.1 Image representation 6
2.1.2 Looking for features 7
2.1.3 Structure 7
2.1.4 Training the network 10
2.1.5 Transfer Learning 12

3 Related Work **13**
3.1 Embedded Neural Networks 13
3.2 Animal Classification 14

4 Dataset **17**
4.1 Preprocessing 19
4.1.1 Cropping edges 19

5 Architecture **21**
5.1 Dataset 21
5.2 Preparation section 22

5.2.1 Data preparation 23
5.2.2 Training 23
5.3 Observation section 23

6 Design 25
6.1 Back-end 25
6.1.1 Datastore 25
6.1.2 GPU-enabled computer 26
6.2 Front-end 30
6.2.1 Inference on low-powered edge device 30

7 Implementation 33
7.1 TensorFlow 34
7.1.1 Dependencies 34
7.2 OpenCV 35
7.2.1 Dependencies 35
7.3 NumPy 35
7.4 Keras 35

8 Evaluation of image classification system 37
8.1 Experimental Platform 37
8.2 Experimental Design 38
8.3 Classification metrics 39
8.4 Results 41
8.4.1 MobileNet_1.0_224 41
8.4.2 MobileNet_1.0_192 43
8.4.3 MobileNet_1.0_128 45
8.4.4 MobileNet_0.75_224 47
8.4.5 Model size 49
8.4.6 Classification speed 49

9 RPI Performance evaluation 51
9.1 Experimental Plarform 51
9.2 Experimental Design 52
9.3 Power metrics 52
9.4 Energy expenditure of RPI executing MobileNet_1.0_224 . . 53
9.5 Classification speed 54

10 Discussion 55
10.1 Evaluating Results 55
10.1.1 Classification 56
10.1.2 Classification speed 57
10.1.3 RPI Energy expenditure 58
10.2 Idea 58

10.3 Dataset 59
10.4 Crowded images 60
10.5 Issues with exporting models 60
10.6 Batching classifications 61

11 Conclusion **63**
11.1 Future Work 64

Bibliography **67**

List of Figures

2.1 Human vs computer 6
2.2 Illustration of CNN Architecture 8
2.3 A classic CNN architecture example. 9
2.4 Label Vector. 10

4.1 Example images from the COAT dataset. 18
4.2 Cropped images. 19
4.3 Comparison of original and cropped image. 20

5.1 Architecture of system for *in-situ* analytics (arrows shows dataflow). 22

6.1 Design. 31
6.2 Folder Structure. 32

8.1 MobileNet_1.0_224 accuracy 41
8.2 MobileNet_1.0_224 crossentropy 41
8.3 MobileNet_1.0_224 Top 1 Confusion Matrix 42
8.4 MobileNet_1.0_224 Top 2 Confusion Matrix 42
8.5 MobileNet_1.0_192 accuracy 43
8.6 MobileNet_1.0_192 crossentropy 43
8.7 MobileNet_1.0_192 Top 1 Confusion Matrix 44
8.8 MobileNet_1.0_192 Top 2 Confusion Matrix 44
8.9 MobileNet_1.0_128 accuracy 45
8.10 MobileNet_1.0_128 crossentropy 45
8.11 MobileNet_1.0_128 Top 1 Confusion Matrix 46
8.12 MobileNet_1.0_128 Top 2 Confusion Matrix 46
8.13 MobileNet_0.75_224 accuracy 47
8.14 MobileNet_0.75_224 crossentropy 47
8.15 MobileNet_0.75_224 Top 1 Confusion Matrix 48
8.16 MobileNet_0.75_224 Top 2 Confusion Matrix 48

List of Tables

2.1 Image Classification Output 7

3.1 MobileNet vs Inception V3 for Stanford Dogs, comparing classification accuracy, extracted from [7]. 14
3.2 MobileNet vs ResNet-152 (data combined from [7] and [36]) 15

4.1 Dataset distribution. 18

6.1 MobileNet Architecture 29

8.1 Mobilenet models with input resolution. 39
8.2 MobileNet_1.0_224 Precision and Recall for the top-1 and top-2 case, extracted from 307 test images. 41
8.3 MobileNet_1.0_192 Precision and Recall for the top-1 and top-2 case, extracted from 307 test images. 43
8.4 MobileNet_1.0_128 Precision and Recall for the top-1 and top-2 case, extracted from 307 test images. 45
8.5 MobileNet_0.75_224 Precision and Recall for the top-1 and top-2 case, extracted from 307 test images. 47
8.6 Modelsize in megabytes(mb) for the four different variants of MobileNet. 49

9.1 Measurements of Raspberry Pi 3 Model B being idle and doing image classification with the MobileNet_1.0_224 model. 53
9.2 Classification speed for the four different models, measured on the RPI 10 times, and taking the average of the results. . 54

10.1 MobileNet comparison for AP and AR for the top-1 and top-2 case, extracted from tables 8.2,8.3,8.4,8.5 56
10.2 MobileNet_1.0_224 and MobileNet_1.0_128 Precision and Recall for the top-1 case. 56
10.3 Softmax example. 60

List of Abbreviations

AP Average Precision

API Application Programming Interface

AR Average Recall

CNN Convolutional Neural Network

COAT Climate-ecological Observatory for Arctic Tundra

CPU Central Processing Unit

DNN Deep Neural Network

FN False Negative

FP False Positive

FPS Frames Per Second

GPU Graphical Processing Unit

mA Milliampere

NN Neural Network

RPI Raspberry Pi

SGD Stochastic Gradient Descent

TP True Positive

Introduction

Among all ecosystems on Earth, the arctic tundra is one of the ecosystems most challenged by climate change. Due to rapid change in climate, new ecosystems arise with unknown properties. Such drastic changes calls for monitoring [1].

The Climate-Ecological Observatory for Arctic Tundra (COAT) [1] is a response to urgent international calls for scientifically robust observation systems. It consists of five institutions within the Fram Centre[1]. It is a long term project with a goal of creating this observation system, as well as documenting and understanding the climate impacts in arctic tundra ecosystems.

COAT uses camera traps to monitor the biodiversity in the ecosystem of the arctic tundra. Camera traps is a widely used method of monitoring, and have been a large factor in wildlife ecology the past two decades. Across the world, there are deployed tens of thousands camera traps [2] [3].

Images taken by COATs camera traps today, are manually examined and labeled. This process takes a lot of human labour, and can require several months of work. Today this task is often performed by COATs own biologists.

This thesis presents the architecture, design and implementation of an image classification system to be used with camera traps for *in-situ* analytics on

1. www.framsenteret.no/english

accumulated data. The system will automatically classify and label the images taken by the cameras. This is done periodically, so if important information emerges, it can be reported to the back-end as soon as possible.

1.1 Problem definition

This project has built an analytics system for small embedded computers to automatically classify animal species from images collected by COAT wildlife camera traps. The purpose of the project is to determine the architecture, design, implementation and performance characteristics of the system.

Typical platform characteristics of the small embedded computers in 2017 are:

- RAM restricted: 1GB or less RAM, which constrains the model to be small enough to fit in a unit with such limited memory.
- CPU restricted: CPU, like the ARM Cortex-A53, having low clock speeds(1.2GHz).
- Storage restricted: Where storage units are restricted to SD cards or similar technologies, currently up to 256GB.

We describe the concept of a mobile neural network for image classification, which is based on deep Convolutional Neural Network (CNN)s. We give a thorough introduction to CNNs and image classification, as well as describing the preparation of the dataset for training. We present the architecture, design and implementation of a system for embedded analytics of animal images on small embedded computers. The system, can train and evaluate CNN models which in turn can be used to classify new images of animals on the small computer. We evaluate the systems classification-accuracy and speed, by comparing the CNN-models against each other. We also evaluate the energy expenditure of the small computer when running idle, and under load. Finally we discuss the approach of this thesis, and propose future work for our embedded analytics system.

1.2 Contributions

This thesis makes the following contributions:

- A thorough description of the dataset, and the preparation of the dataset.
- An image classification system, with its architecture, design and implementation.
- An evaluation of the image classification system, with regards to recognition metrics and energy expenditure.
- A comparison of different MobileNet models.
- Insights in porting a model from a resource rich environment like a GPU-computer, to a resource scarce environment like the RPI.

1.3 Outline

The thesis is structured into eleven chapters including the introduction.

Chapter 2 describes object classification, as well as giving a thorough introduction to CNNs.

Chapter 3 presents related work in the field of animal classification, as well as related work done in embedded/mobile neural networks, comparing it to the work done in this thesis.

Chapter 4 describes the dataset in this project, as well as how it is prepared for training on a Neural Network (NN).

Chapter 5 describes the system architecture, and how a front-end and back-end work together as a complete system for animal classification.

Chapter 6 describes the design of the system, and shows how a Deep Neural Network (DNN) was trained in the back-end, yielding a model to be used for inference in the front-end.

Chapter 7 describes the implementation and dependencies of the system, as well as describing where to find said dependencies.

Chapter 8 describes the classification quality of four different variations of the MobileNet model.

Chapter 9 describes the classification speed and energy expenditure of the MobileNet model running on the Raspberry Pi.

Chapter 10 then discusses the results, and describes the process of solving the problem of doing image classification on a small embedded computer. It also describes the difficulty of keeping a high classification accuracy, while reducing the model size drastically.

Chapter 11 concludes the thesis, and suggests future work to improve the systems classification quality.

2

Image Classification

Image classification is the task of identifying different objects in digital images or video and then assign semantic labels to the image. It is not to be confused with object detection, which is the task of locating the object within an image. It has become important in computer science, and is applied in many computer systems doing; localization, detection and scene parsing [4].

In recent years, image classification as a field of research has made great progress. This is due to the use of CNNs [5] [6] [7] [8] [9] [10]. Large public datasets like ImageNet [11] and Standford Dogs [12], as well as benchmarks like the *ImageNet Large Scale Visual Recognition Challenge 2014* (ILSVRC2014) [13] has also been important for the research fields development. This chapter will introduce the concept of CNNs and data augmentation.

2.1 Convolutional Neural Networks (CNNs)

CNNs became very prominent in 2012 when Alex Krizhevsky presented his CNN AlexNet [5] which was the winner of that years "ImageNet Large Scale Visual Recognition Challenge (ILSVRC)" [13]. This resulted in a drop from 26% to 15% classification error, which at that time was an incredible feat. Since then, several companies have been using DNNs as the main part of their services.

Facebook[1] uses CNNs for their automatic tagging algorithm [14], and Google for their photo search, as well as YouTube[2] video analysis [15]. There are many uses of these CNNs, but the arguably most popular use-case for them, is for image processing [16].

2.1.1 Image representation

We have previously stated that image classification is the task of identifying objects in an image, and then assigning labels that best describes the specific image. A human is able to identify the setting it is in, as well as identifying objects within this setting. Being showed an image, a human is most of the time able to identify the setting, and label each object within [17]. These are abilities that humans are good at. A human interprets an image it sees, like the one in figure 2.1a

(a) What humans see

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

(b) What computers see

Figure 2.1: Human vs computer

When a computer "looks" at an image, it will see an array like the one in figure 2.1b. This is an array of pixel values. The dimension of the picture matters as well. If the picture had a width of 224, a height of 224, and had colors in RGB format, the matrix would have a dimension of (224 x 224 x 3). Each number in the matrix would have a number between 0 and 255 which is the color intensity of each pixel. To a human this is meaningless while performing image classification, but for a computer, it is the only thing it can interpret. The computer is shown an image represented by a matrix like the one in figure 2.1b, and through processing it will return numbers that describes the certainty of the picture belonging in a certain class. An example of this classification is shown in table 2.1.

1. www.facebook.com
2. www.YouTube.com

Class	Precision	Precision in percent
WhiteTailedEagle	0.8	80.0%
Crow	0.15	15.0%
Reindeer	0.05	5.0%
ArcticFox	0	0%

Table 2.1: Image Classification Output

2.1.2 Looking for features

How can a computer separate between an eagle and a fox? A human does this by looking at the animals features. It notices that a fox has paws, and an eagle has wings. The computer looks for features in a similar way. It starts by finding edges and curves in the image. These are found with the help of filters. A filter is usually a (1 x 1), (3 x 3), (5 x 5) or (7 x 7) matrix. These matrices could have a pattern like the one below.

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

This filter would represent a vertical edge detector. And when applied to a receptive field of the image matrix, it may or may not detect a vertical edge in the specific area of the image. A curve detector might look like the filter below.

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Using such filters by traversing the image, could eventually detect some edges and curves. Then in the next layer of the CNN, a different filter that combines the previous ones, might detect a paw or a wing. Stacking layer upon layer would eventually build more abstract features, and going on with this, the CNN would learn what an eagle and a fox looks like. It could also then separate between them. The reason it learns, is becase the weights in the CNN will tune themselves to feedback from a "run-through", which tells them if the classification on that run was better or worse than the previous.

2.1.3 Structure

CNNs take an image, runs it through a series of convolutional, nonlinear, pooling and fully connected layers [18]. It then returns an output which can be either single class, or it can be a probability of several classes that best describes

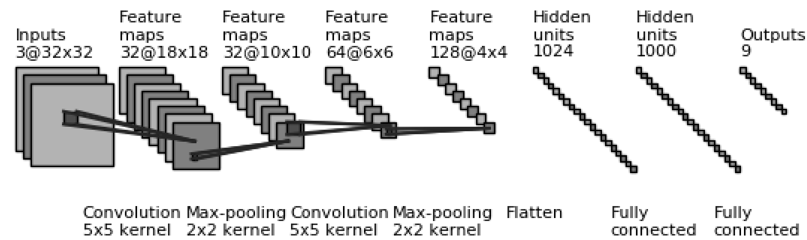


Figure 2.2: Illustration of CNN Architecture

the whole image. The structure of the CNN could look like the one in figure 2.2.

Convolutional layers

The first layer in a CNN is always a "convolutional layer" [19]. Its primary function is to extract features from an image. Imagine a flashlight shining over the picture, starting from the top left corner. Say that the light in this flashlight shines over an area that covers 5 x 5 pixels. The flashlight is a filter like the ones explained in section 2.1.2, and the area it shines on is called a receptive field [19]. The filter is also a matrix of numbers, where the numbers are called weights. As the filter is convolving across the picture, it is multiplying the weights from the filter with the pixel values from the image. The multiplications are summed up into one number. Then the filter convolves another step, which is called a stride(often one or two pixels), and repeats the multiplication on the new receptive field. One important thing to notice is that the filter needs to have the same depth as the input. If the picture had 3 dimentions in color(RGB) then the filter needs to be 5 x 5 x 3 to cover all of the pictures dimentions. This is to make the multiplications work correctly. Increasing the number of filters produce more features, leading to a larger network that is better at recognizing

patterns. It does however have a higher computational complexity in terms of memory usage, which is caused by even more multiplications.

So far, we know that filters in convolutional layers detect low level features like edges and curves. These filters are used to train new filters. The new filters might learn how a paw or a wing look like. The network keeps convolving, and learns more abstract features. Eventually some filters might trigger when they see a bird or a fox in an image.

Nonlinear layers

A nonlinear layer is a gatekeeper between each convolutional block, see figure 2.3. It is an activation function like the sigmoid function or the ReLU [20] function. It activates the output of the preceding layers, by transforming it into a value of 0 or 1 depending on the value it received. This is to let through features with a high score of confidence, and prevent features with a low score of confidence. This is done to keep features that have contributed to better validation accuracy, and discard the ones that have not.

Input->Conv->ReLU->Conv->ReLU->Pool->ReLU->Conv->ReLU->Pool->FC

Figure 2.3: A classic CNN architecture example.

Pooling layers

A pooling layer reduces the spatial dimension of the input, and retains the most distinct features. It does this by using a pooling filter which usually takes the largest(Max pooling) or average(Average pooling) value within the filters and uses that as a representation for the area covered by the filter [19] [21]. An example can be seen in the matrices below. Where the first matrix is a 4 x 4 image, and the second is the result of the filter passing over it. It started in the top left corner finding that 5 was the largest value. In the top right corner, 4 was the largest value. Bottom left, 2 was the largest, and in bottom right, 4 was the largest.

$$\begin{bmatrix} 5 & 5 & 4 & 2 \\ 3 & 4 & 3 & 3 \\ 0 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 5 & 4 \\ 2 & 4 \end{bmatrix}$$

The input is now downsampled, and the overall number of parameters is re-

duced. By doing this, we also reduce the "*memory footprint*" of the network, making it possible to add even more filters. The most common pooling techniques are max pooling and average pooling. Max pooling traverses the image with a small receptive field(2 x 2), and returns the maximum value within the field. The stride of the Max-pooling is usually 2, so that it does not cover previous pixels. Average pooling does the same, but calculates the average instead of taking the maximum value.

Fully Connected layers

The fully connected layers are the last few layers that takes the output from whatever layer was before it, be it convolutional, nonlinear or pooling, and outputs a vector [19]. If this is the last fully connected layer, it is often the classifier of the network. In a case where we are predicting 6 different species of animals, it would take the input of the layer before it, and outputting a 6 dimensional vector. If our labels and output vector look like the ones in figure 2.4, we can see that the classification layer has predicted a probability of the input image to be 10% Fox, 10% Crow 75% Eagle and 5% Raven.

$$\begin{bmatrix} Fox & Crow & Eagle & Raven & Reindeer & Owl \end{bmatrix}$$

$$\begin{bmatrix} 0.1 & 0.1 & 0.75 & 0.05 & 0.0 & 0.0 \end{bmatrix}$$

Figure 2.4: Label Vector.

2.1.4 Training the network

How does the filters know what values to keep? The computer adjusts the values(also called weights) in the filters through a process called backpropagation.

Before explaining the concepts of backpropagation we should look into what a CNN needs in order to function. As with humans, the moment it is born, the mind is fresh. A newborn human does not know what a fox or an eagle is. In a similar way the CNN does not know this either. The weights are usually randomized before training, and the filters does not know how to look for curves and edges, nor paws or wings. As a human grows older, its teachers and parents shows them pictures with assigned labels. The CNN is trained in the same way.

Backpropagation can be separated into 4 particular parts.

- Forward pass
- Loss function
- Backward pass
- Weight update

The forward pass takes a training image, which a computer sees as an array of numbers (224 x 224 x 3). It passes this image through the whole NN. If this is the first training image, and all the weights and filter values were randomly initialized, the result could look like the vector below, which is an output that does not give preference to any class.

$$\begin{bmatrix} 0.166 & 0.166 & 0.166 & 0.166 & 0.166 & 0.166 \end{bmatrix}$$

As we are dealing with a fresh CNN, it is not able to recognize the low level features like curves and edges. Hence no reasonable classification. This is where the loss function comes in. If we use the classes of animals as we used in figure 2.4, and give our CNN a picture of an eagle, the label for that picture would be like the vector below.

$$\begin{bmatrix} 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

So what the loss function does, is calculating the measure of error between the true labels, and the predicted labels. In the beginning the loss will be very high. We want our CNN to get to a point where the predicted labels are the same as the true labels. We get there by minimizing the amount of loss returned by the loss function. This is where the backward pass comes in. The backward pass is using an optimizer that goes backward through the CNN and figures out what weights contributed most to the high loss, and tweaks the weights so that the loss decreases. When the optimizer has found the best settings for the weights, we go through the last step, which is the weight update. This is where all the weights in the filters are updated to help the minimze the loss.

Optimizers which is used in the backward pass, comes in different shapes and sizes. Some of the more popular optimizers are;

- Stochastic Gradient Descent (SGD) [22]
- RMSprop [23]
- Adam [24]

The optimizers have parameters that tune the overall network, and some of these parameters are learning rate and decay. These are called hyperparameters, as they are chosen by the programmer. For example a high learning rate causes the optimizer to take bigger steps in the weight updates, it does however come with the risk of taking too large steps that are not precise enough, and it will "overshoot" and not converge. A low learning rate can result in a network that does not learn anything.

A problem in deep learning is that the DNNs tries to memorize the training data. So a network can become really good at predicting images it has allready seen, but does not generalize well from the patterns it observes. This phenomenon is called overfitting [25], and usually happens when the network is too big and complex for the task. It contains too many parameters, that causes it to overreact to unimportant details in the training data [26]. When this happens the NN will have a big problem in that it has a poor predictive performance on new data. Luckily there are techniques that can counter this behaviour of overfitting. Some of these techniques are soft weight sharing [27], dropout [28] and regularization [29].

2.1.5 Transfer Learning

Transfer learning is the concept of using existing DNN architectures, and pre-trained weights in combination with your own classification layer and dataset. The computation cost of training this NN, is only a fraction of what it would haven been when training a NN from scratch without imported weights. Training a CNN from scratch would train all of the layers in the network, but with transfer learning, you only train the last few layers. When the network allready has pretrained filters that knows what features to look for, you only need to "tune" it for your own dataset in the top layers.

ImageNet [11] is one of the more famous image databases for deep learning. Several DNN architectures has pre-trained weights for this dataset. Loading the weights from ImageNet and fine-tuning your network for your own dataset is standard practice. This exploits the advantage of ImageNets features as well as saving hours to weeks of training time. It is possible to keep a few of the earlier convolution layers fixed during fine-tuning, which reduces the possibility of overfitting.

3

Related Work

Looking through relevant literature shows that there are many systems and projects that are working on detection and classification with embedded or mobile DNNs. However there are not many of these embedded systems that focus on animal classification specifically. We can see that Animal classification with DNNs in general is done in several systems. Few however does it with embedded sytems using small mobile DNNs.

3.1 Embedded Neural Networks

According to SqueezeDet [30], object detection is a crucial task for autonomous driving, and in addition to high accuracy, object detection also need real-time inference speed to ensure prompt vehicle control. To achieve this one needs a *small model size*. Bichen et. al. had a model size of 4.8MB. One also needs energy efficiency so that deployment can happen on embedded systems.

The most energy-expencive operations involved in neural network inference is DRAM access, which has a 100 times higher energy use than SRAM access [31]. Hence the smaller DNN models, so that the whole model can fit in SRAM at the same time, thus reducing energy usage.

Small neural networks like SqueezeNet [6], which achieves AlexNet [5]-level accuracy with 50x fewer parameters and $<$ 0.5MB model size [6], Inadola .et

al. claims that smaller DNN's are more feasible to deploy on hardware with limited memory, than larger models.

MobileNets [7], a new class of efficient CNNs for mobile vision applications were presented in April, 2017. The MobileNets comes in different variants, where they differ in the resolution of the images they take, and the number of weights they have. This results in different model sizes. When testing the MobileNets on the Stanford Dogs dataset [12], they found that the MobileNets could compete with one of the most renowned DNNs, Inception V3 [9].

Model	Top-1 Accuracy	Million Mult-Adds	Million Parameters
Inception V3	84%	5000	23.2
1.0 MobileNet-224	83.3%	569	3.3
0.75 MobileNet-224	81.9%	325	1.9
1.0 MobileNet-192	81.9%	418	3.3
0.75 MobileNet-192	80.5%	239	1.9

Table 3.1: MobileNet vs Inception V3 for Stanford Dogs, comparing classification accuracy, extracted from [7].

As we can see in table 3.1, the largest MobileNet architectures can compete with Inception V3. In return the MobileNets only has a fraction of the parameters(size) that Inception has, and they also has less computation cost measured in mult-adds. The dataset has 120 different breeds of dogs with about 150 images per class, making it a total of 20 580 images. Compared to our dataset it has a lot more classes, and is designed for the task of fine-grained image classification, whereas our dataset has more diverse classes like birds and foxes.

In July, 2015. Dürr et. al. published a paper describing real-time face recognition on a Raspberry Pi using CNNs on limited computational resources. [32]. They reached a performance of approximately 2 frames per second with more than 97% recognition accuracy. Related to our work which also does inference on a Raspberry Pi, they show that CNNs can be effective on devices with limited resources.

3.2 Animal Classification

Norouzzadeh et. al. presented in April, 2017, a system for classifying different animal species on the Snapshot Serengeti dataset containing 48 species in

3.2 million images, using Deep CNNs [33] [34]. Using the CNN architectire ResNet-152 [35], they achieved a classification accuracy of 92%. They were also able to have the system classify new images which the system had a high confidence about, because of the hight accuracy in classifying specific species. This allowed for human time to be focused elsewhere. Our system does not achieve this high overall accuracy, because we sacrifice some accuracy in the advantage of a smaller model size. Where the ResNet-152 architecture has 152 layers in its neural network, we are working with a MobileNet that has only 28 layers [7]. This causes the big difference in model size. Where the ResNet-152 has a model with 60.2 million parameters, the largest MobileNet model has only 4.2 million.

Resolution	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
1.0 MobileNet-192	69.1%	418	4.2
1.0 MobileNet-160	67.2%	290	4.2
1.0 MobileNet-128	64.4%	186	4.2
ResNet-152	-	-	60.2

Table 3.2: MobileNet vs ResNet-152 (data combined from [7] and [36])

Chen et. al. presented in January, 2015, a Deep CNN based species recognition algorithm for wild animal classification on camera-trap image data [8]. They compared the Deep CNN algorithm to a visual bag-of-words [37] for classification. Where the bag-of-words model achieved an overall animal classification accuracy of 33.507%, the Deep CNN achieved a 38.315% accuracy. The camera-trap dataset contained 20 different species. They bring up the difficulty of a challenging data-set with many species of animals.

H. Thom. presented in December, 2016, an animal species identification system that can automatically identify small mammals in camera trap images [38]. The system used three different deep CNNs. The system achieved a 97.84% accuracy 97.81% precision and 93.45% recall on a dataset with 10 000 images spanning 11 classes. They show that establishing real time identification at remote camera traps could be difficult due to high computational costs of CNNs.

H. Thom. presented in June, 2017, a unified detection system that can automatically localize and identify animal species in digital images from camera traps in the Arctic tundra [39]. The system unified three object detection methos using CNNs. The system used a dataset with 8000 images containing over 12 000 animals spanning 9 different species. The system can automatically

detect animals in the Arctic tundra with a 94.1% accuracy at 21 frames per second.

WTB [40] is an end-to-end, distributed, IoT system for wildlife monitoring. It integrates recent advantages in machine learning in regards to image processing, to automatically classify animals in images from remote camera traps. WTB uses Google TensorFlow [41] and OpenCV [42] applications to perform the classification and tagging for a subset of their 1.12 million images. Using stock Google images of animals, and a small number of their own images as background, they construct a synthetic dataset for training. Due to this, the system is able to accurately identify bears, deer, coyotes, and empty images. This in turn significantly reduces the time and bandwith requirements for image transfer, as well as end-user analysis time, as WTB filters the images on-site.

This is the closest work to ours, as it is doing animal classification based on data from camera-traps. Their focus however is not minimizing the model size, as their model is 490MB. This is significantly higher than our 17MB. By doing the classification *in-situ* they decrease their network transfer by 70%, and achieves a classification error of 0.2% for coyotes, 1% for bears, and 12% for deer.

4

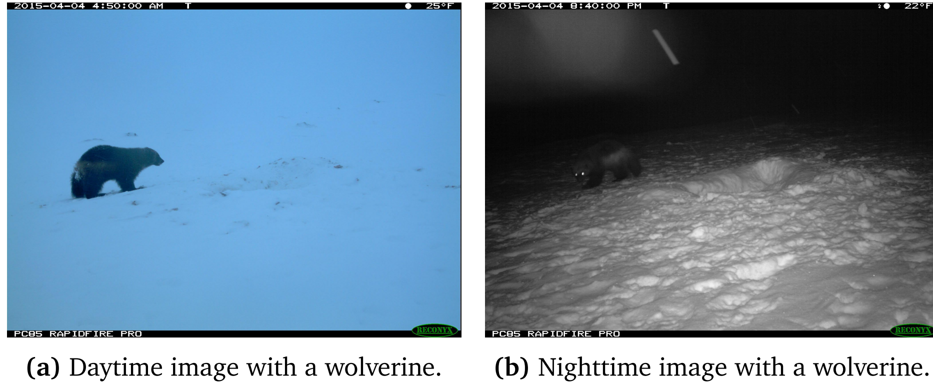
Dataset

This chapter describes the dataset we have worked with. As there have been done work similar to ours within the COAT project [38] [39] before. We base our dataset on the contributions that has allready been made.

The overall dataset provided by COAT contained 1 849 076 images taken from 2011 to 2016. The images are taken by camera traps in the northern county of Finnmark, Norway. 37 camera traps [39] are deployed every year, scattered across the five regions: Stjernevann, Komag, Ifjor, Nyborg and Gaissene.

The pictures are taken during daytime and nighttime. This is possible due to the infrared flash the cameras are equipped with. However the pictures taken during the night are in greyscale, where pictures taken during the day are in color. See figure 4.1.

In this project we use a subset of the overall dataset from COAT. Our subset of data is a modification of the dataset H. Thom [39] produced from the COAT data. The dataset consists of 9 classes of animal species, where images are labeled thereafter.



(a) Daytime image with a wolverine. **(b)** Nighttime image with a wolverine.

Figure 4.1: Example images from the COAT dataset.

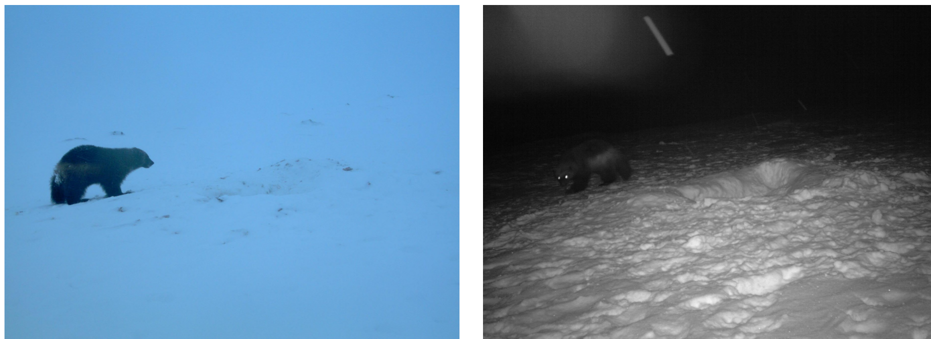
The labelled dataset suffers from heavy class imbalance. There are classes like the raven that has close to 50 000 labelled images, the snowy owl class only has 52 labelled images. A class imbalance like this can lead to an imbalanced classification [33]. Where the classifier might be really good at classifying ravens, it is really bad at classifying snowy owls. The class imbalance is reduced through pruning of classes. We decrease the number of images in a high populated class, to a more normalized amount. A new subset of the dataset emerges from this pruning. The class distribution of our dataset is based somewhat on the distribution H. Thom worked with on object detection [39]. The dataset distribution of the classes is represented in table 4.1.

Class	Images
ArcticFox	684
Crow	585
WhiteTailedEagle	1084
GoldenEagle	1577
Raven	2964
RedFox	2841
Reindeer	858
SnowyOwl	52
Wolverine	566
Total	**11211**

Table 4.1: Dataset distribution.

4.1 Preprocessing

The images in the dataset has a large size. The width, height and depth of the images in pixels are 2048 x 1536 x 3. Earlier work within the COAT group shows that removing the black borders encapsulating the picture, which contains information like date, time and temperature had a positive effect on classification [38]. Following this strategy leaves us with pictures like the ones in figure 4.2. The size of these pictures are 1844 x 1382 x 3 pixels.



(a) Cropped daytime image. **(b)** Cropped nighttime image.

Figure 4.2: Cropped images.

4.1.1 Cropping edges

Cropping the black edges off of an image is something that needs to be done only once for the dataset. Because after the data-preprocessing is done, we store a copy of the preprocessed data. Preprocessing gave us the tools to scale, crop, process and draw on images among other things. In the case of our dataset, we only needed to crop the edges off of the images, because we followed the strategy of H. Thom [38]. This was done by finding a percentage of width and height large enough to make sure the black borders would be cut away. We decided on 10 percent, which translated into 5 percent per side. By having a human visually compare the before and after image, a conclusion were made that 5 percent per side was enough to remove the edges, see figure 4.3.

(a) Original image.

(b) Cropped image.

Figure 4.3: Comparison of original and cropped image.

5

Architecture

This chapter describes the architecture of our system for doing *in-situ* analytics in the Arctic tundra on a low-power embedded computing device.

The overall architecture consists of two sections. The preparation section, and the observation section. In the preparation section we prepare our dataset for training which results in a model ready for inference. The observation section is an *in-situ* embedded device where new data is gathered, and analyzed with our pre-trained model. The two sections work together as a complete system, where the preparation section is mostly there for the setup phase, and the observation section is the operational unit. Figure 5.1 shows the abstraction of the overall system.

5.1 Dataset

The preparation section contains the dataset for training. This dataset contains several sets of images and labels from different locations and projects. Earlier projects involving camera traps have generated a set of image data from said camera traps. These images are stored on a server. Looking at chapter 4, we see the results of this data collection. Some of the data is structured, from previous projects, whereas others are not. We use a dataset which is allready labeled and sorted into folders accordingly. This is represented as the selection and fetching of dataset for training in figure 5.1. The data is a central part of

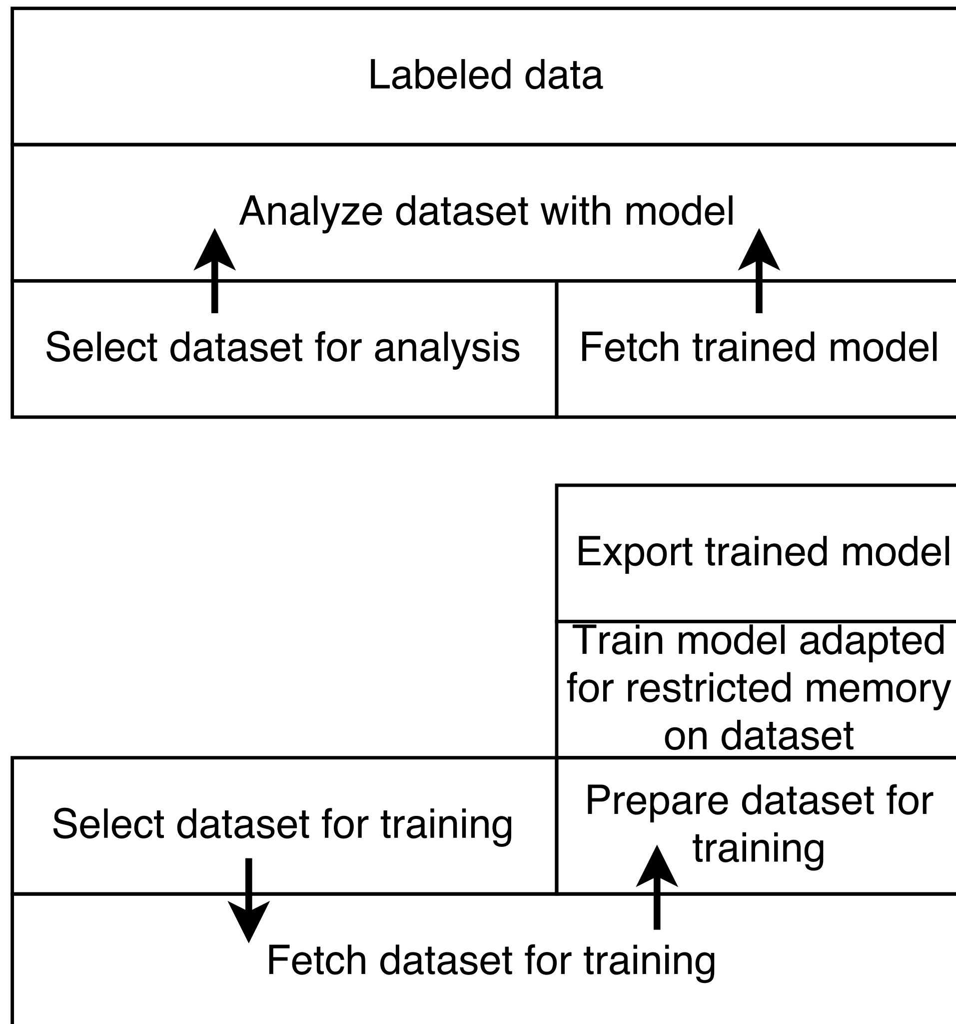


Figure 5.1: Architecture of system for *in-situ* analytics (arrows shows dataflow).

the project, but the extraction of it is not. This is because the extraction of data is done just once. This is when we extract the original dataset for further processing done in the back-end.

5.2 Preparation section

The preparation section consists of two main parts. The data preparation module, and the training module.

5.2.1 Data preparation

The data preparation is done in the preparation section. This means that the data is processed and then saved as a subset of the dataset. The purpose is to prepare the data for the training module. The data we use has some unwanted characteristics which are known to skew the analysis accuracy of the model. Because of this, we mask the unwanted characteristics from the data.

5.2.2 Training

The training is also done in the preparation section. It takes the prepared dataset and feeds it through its training algorithm. It will then learn the features of the different classes in the dataset, and by being showed enough of this data, be able to recognize the same or similar features in data it has never seen before. It can then be used for recognition and classification of new data. The training step will produce a model that can be used in the observation section on our edge device for analysis of new data. This will result in new data sorted by labels.

5.3 Observation section

The observation section is the final step of the system pipeline. It is the "product" of the previous steps. In it, is a low-powered embedded computer with limited resources and energy. It receives data from an attatched camera, and classifies the data with the model fetched from the preparation section. Both the raw and classified data is stored locally on the embedded computer until it is fetched. This data is then stored on a server for long-time storage.

/6 Design

This chapter describes the design of our system for doing *in-situ* analytics in the Arctic tundra on a low-power embedded computing device. We will show how we trained a DNN in our back-end which yielded a model we could use for inference on our embedded device. Going on, we will follow the flow of figure 6.1, and describe each component respectively.

6.1 Back-end

Our back-end is built up of two main components. The datastore and the GPU-enabled computer.

6.1.1 Datastore

The datastore resides in the back-end of our system. It contains several sets of images and labels from different locations and projects. The main datasets are pictures from camera traps focusing on two different types of traps. One dataset is from camera traps using bait to lure wild animals like eagles, foxes, reindeer, etc, close for photography, where as the other focuses on small rodents within a tunnel that has a camera trap mounted.

The dataset we decided to use is described in detail in chapter 4.

6.1.2 GPU-enabled computer

The dataset is fetched from the datastore manually, and a copy of it is stored in the local persistent memory on the Graphical Processing Unit (GPU)-enabled computer. This copy is then fed through a pipeline of data-preprocessing to refine the dataset before doing any deep learning on it. This begins with cropping of edges to remove the unwanted features from the images, as they are known to skew the accuracy of the DNN. The data is then sorted into three subfolders *Train, Test* and *Validation*. This is normal practice in data science[1], as it will give us "*unseen*" data when it is ready to test the model.

The images are then fed into a CNN, where they are used to train a model. This model is what we are after, it is the "*product*" of the GPU-enabled computer, which is then manually transfered to the edge device in our front-end for further use.

The edge device loads this model, and uses it to label new images it has stored in its local storage. After the images are labeled they can be stored alongside other images of the same class.

Data pre-processing

The pre-processing of the data consists of edge cropping, and sorting. We explained how we cropped the edges of the picture data in detail in chapter 4. When an image is cropped, it is saved in a separate location within the computer storage for cropped images. The following procedure in the pre-processing is the sorting of images.

We sort the images into three different subsets of images, called *Train, Test* and *Validation*. This is because we use a subset of the dataset as training data, which the model will see and learn from. Under training it will validate its progress on a validation set, which is also a subset of the dataset, albeit smaller than the training set. When the training is finished, it will compare its final result up against a test set, which is images that it has never seen before. If it had tested its accuracy on known images, it would have a much higher prediction accuracy, as it can learn exactly what these images looks like, but by being shown something new, it will be less biased in its prediction.

The now sorted images, allready has a label, since they were allready classified in an earlier project, see chapter 4. In that regard, each class has its own

1. https://info.salford-systems.com/blog/bid/337783/Why-Data-Scientists-Split-Data-into-Train-and-Test

folder. The folder structure is shown in figure 6.2. This means that each subfolder of *Train, Test* and *Validation* contains the folders *ArcticFox, Crow, Eagle, GoldenEagle, Raven, RedFox, Reindeer, SnowyOwl* and *Wolverine*.

Because our original dataset comes in one folder containing all the subfolders described above, and not in the *Train, Test, Validation* folder structure, we had to split/sort it like that. We made a script to do this, which splits one folder structure containing the different classes in subfolders, into three folders of the same structure. Even though the structure of the folders are the same, there is a difference in the amount of data in the split. The *train* folder contains most of the data, in our case about 60 percent, where as the *test and validation* sets has about 20 percent each.

Before the data is split however, the images are randomly shuffled within their respected class folder. This is to ensure that we do not have pictures from only one location in a split. If we do not shuffle the images, we would for example have taken the first 200 pictures from the class *ArcticFox* and moved them into the "*Validation -> ArcticFox*" folder. Most likely the first 200 pictures would come from the same area, and would be very similar, as the background is most likely the same. So if the model was to train on pictures from one location, and validate against pictures from a different location, it would be bad at generalizing, and this most likely would result in a bad classification accuracy. The reason for this is overfitting. It means that the algorithm will become very good at recognizing the specific images it have been shown, and not so good recognizing anything else. This is why we randomly shuffle the images within the folders, to ensure that we do not get pictures from just one location in one split.

When the pictures are shuffled, we create lists for each of the *train, test* and *validation* folders. These lists contains the path to the pictures that should now be copied from the old folder into the new. The script traverses the lists and copies each file into the new folder structure.

This, like the cropping of the pictures, would only need to be done once. Because when it is done, you would have a dataset that is ready for training.

Training model

When training the DNN model we are using, which is a CNN. We base it on a DNN architecture called MobileNet [7]. The MobileNet body architecture is defined in table 6.1. All layers in the DNN are followed by a batch-normalization [43] and a ReLU [20] nonlinearity, except the last fully connected layer. This last layer does not have a nonlinearity, and instead feeds into the final layer

which is a softmax classifier.

Using this DNN we add our own top layer to it. So instead of using MobileNets softmax layer, which is tuned for 1000 classes, we remove that, and add our own softmax layer to the network. As we have 9 classes, our softmax layer has an output size of 9.

As there exists a lot of pre-trained neural networks [5] [6] [7] [35], we take advantage of this, and use the pre-trained weights for MobileNet from the *ImageNet* [11] image database. These weights are allready trained and tuned on a big set of image data, and we will use them as a foundation for training on our own dataset. We do this by freezing all of the layers in our CNN, except the final fully connected layer and our softmax classifier. This means that we set all of the layers, except the last two to untrainable. The parameters are not allowed to change in those layers, as we allready have pre-trained weights in there from the ImageNet database. This in turn will force our final layers to tune themselves to our dataset, using the previous layers knowledge from ImageNet. This is what is called *Transfer learning*, which is explained in section 2.1.5.

The result of the training is a model which is tuned for images like the ones we showed it, and this model is used on the test set to check its final accuracy. We also get a file with the labels of the classes.

The model is then frozen as it is and saved to be used later on the edge device in the front-end alongside the labels.

Type / Stride	Filter Shape	Input Size
Conv / s2	3 x 3 x 3 x 32	224 x 224 x 3
Conv dw / s1	3 x 3 x 32 dw	112 x 112 x 32
Conv / s1	1 x 1 x 32 x 64	112 x 112 x 32
Conv dw / s2	3 x 3 x 64 dw	112 x 112 x 64
Conv / s1	1 x 1 x 64 x 128	56 x 56 x 64
Conv dw / s1	3 x 3 x 128 dw	56 x 56 x 128
Conv / s1	1 x 1 x 128 x 128	56 x 56 x 128
Conv dw / s2	3 x 3 x 128 dw	56 x 56 x 128
Conv / s1	1 x 1 x 128 x 256	28 x 28 x 128
Conv dw / s1	3 x 3 x 256 dw	28 x 28 x 256
Conv / s1	1 x 1 x 256 x 256	28 x 28 x 256
Conv dw / s2	3 x 3 x 256 dw	28 x 28 x 256
Conv / s1	1 x 1 x 256 x 512	14 x 14 x 256
Conv dw / s1	3 x 3 x 512 dw	14 x 14 x 512
Conv / s1	1 x 1 x 512 x 512	14 x 14 x 512
Conv dw / s1	3 x 3 x 512 dw	14 x 14 x 512
Conv / s1	1 x 1 x 512 x 512	14 x 14 x 512
Conv dw / s1	3 x 3 x 512 dw	14 x 14 x 512
Conv / s1	1 x 1 x 512 x 512	14 x 14 x 512
Conv dw / s1	3 x 3 x 512 dw	14 x 14 x 512
Conv / s1	1 x 1 x 512 x 512	14 x 14 x 512
Conv dw / s1	3 x 3 x 512 dw	14 x 14 x 512
Conv / s1	1 x 1 x 512 x 512	14 x 14 x 512
Conv dw / s2	3 x 3 x 512 dw	14 x 14 x 512
Conv / s1	1 x 1 x 512 x 1024	7 x 7 x 512
Conv dw / s2	3 x 3 x 1024 dw	7 x 7 x 1024
Conv / s1	1 x 1 x 1024 x 1024	7 x 7 x 1024
Avg Pool / s1	Pool 7 x 7	7 x 7 x 1024
FC / s1	1024 x 1000	1 x 1 x 1024
Softmax / s1	Classifier	1 x 1 x 1000

Table 6.1: MobileNet Architecture

6.2 Front-end

Our front-end is the low-powered edge device reciding in the Arctic tundra. It is where the object classification on images is done, and we call this operation; inference.

6.2.1 Inference on low-powered edge device

As the low-powered edge device is supposed to have its own storage of a small set of images, it will need to classify these. By being given the trained model beforehand, it is capable of doing this. It does so by loading the model, then transforming each image into something called a *tensor*, which is a multidimentional matrix [41]. Running the tensor through the model will result in a softmax output. This output will then determine which animal is in the picture(if any). Combined with the labels from the label file, this output will be human-readable. After the inference is done, the images should be stored accordingly.

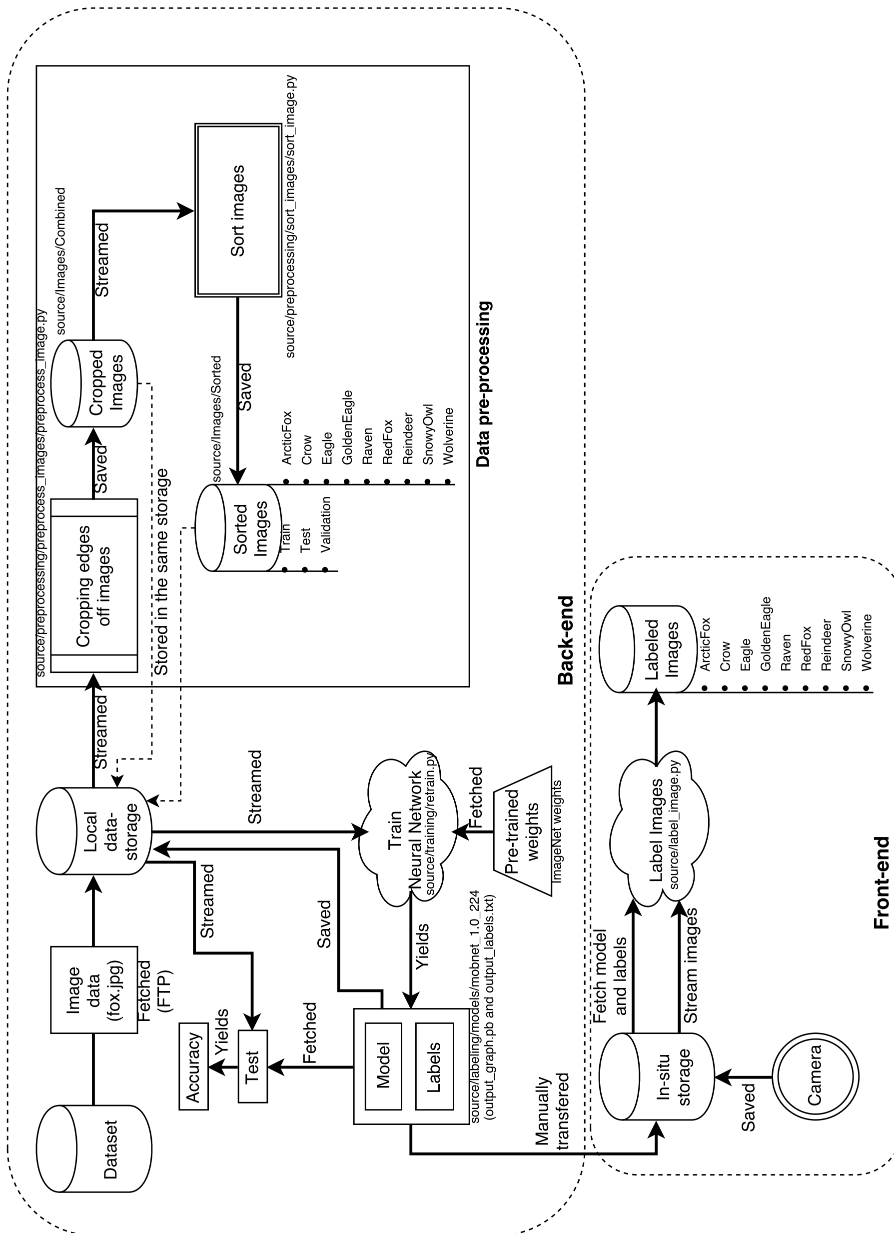


Figure 6.1: Design.

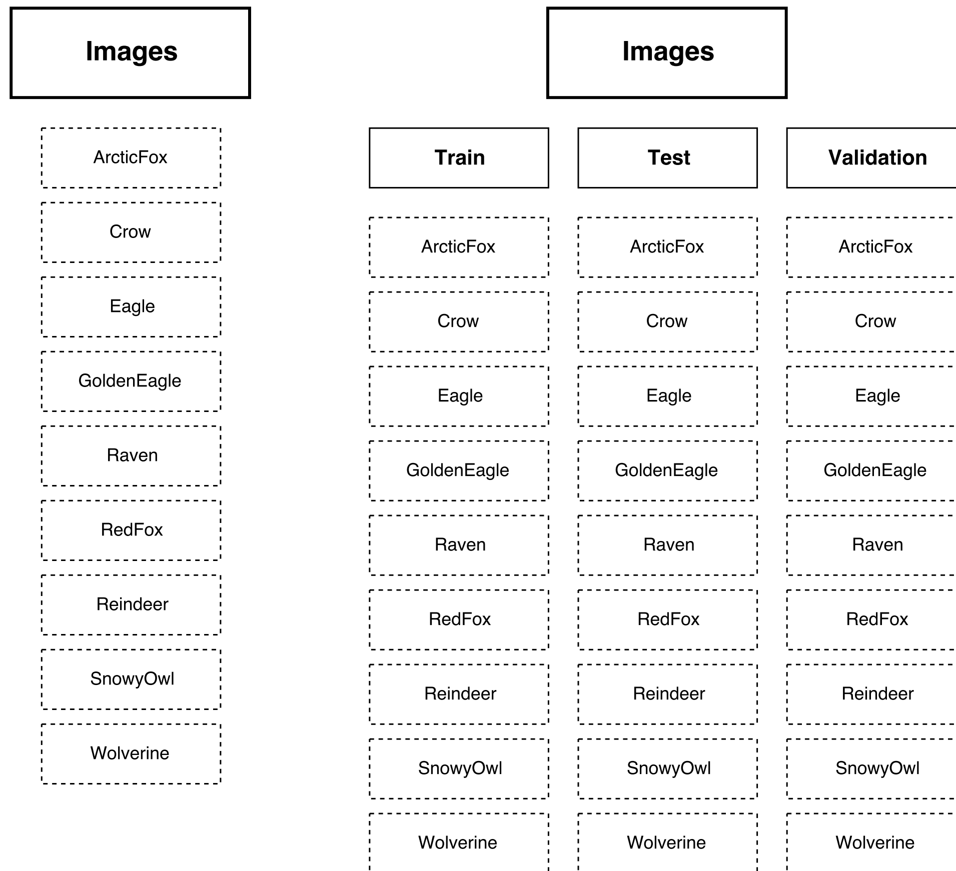


Figure 6.2: Folder Structure.

7 Implementation

Our classification system is mostly based on the open source deep learning platform TensorFlow [41]. TensorFlow is made by a team at Google, and is currently the most popular repository on GitHub[1]. Combining TensorFlow with the programming language; Python[2], a TensorFlow-wrapper called Keras[3] and a computer vision library called OpenCV[4], we get a deep learning framework for image classification.

We implemented a prototype of our classifier in Python, using Keras as a wrapper for TensorFlow. Keras allows for easy and fast prototyping compared to TensorFlow which is more tedious. Alongside the classifier we also implemented the script for trimming the edges off of the pictures, and a script for shuffling and then splitting the images into train, test, validation folders. These supplementary scripts used in pre-processing were also written in Python with the help of OpenCV.

To produce a final model, ready for inference, we used one of TensorFlows example scripts for transfer-learning on images (Per version 1.4.0 of TensorFlow, the script is included in the core library under "examples/image_retraining/retrain.py"). This was done so that we could

1. www.github.com
2. www.python.org
3. https://keras.io/
4. https://opencv.org/

produce a frozen graph, which is what we call a model in TensorFlow. The reason we used TensorFlows script for this, is because we were not able to freeze our own custom CNN model that we built within the Keras library. We will discuss this more in chapter 10.

The final part of the system is the script that labels, or classifies an image. TensorFlow has an example of this as well, but it was not sufficient enough for our use, which we will explain in depth in chapter 10. We used the example as a foundation, and customized it to our needs. This was also written in Python and uses a Python math library called NumPy[5] which is often used for matrix operations, to process the TensorFlow tensors.

Our system is built and deployed on the Ubuntu 16.04 operating system, and is executed on the Raspbian Stretch 4.9 operating system.

7.1 TensorFlow

TensorFlow [41] is the deep learning platform we used to create the deep learning model we required. It is developed by the Google Brain team at Google, and is a system that operates at large scale and in heterogeneous environments. It is also a deep learning library. Benefits with TensorFlow is that it is very popular, and has a lot of maintainers. It also achieves shorter step times than Caffe [10], and performance within 6% of the latest version of Torch [44].

7.1.1 Dependencies

TensorFlow has support for both Central Processing Unit (CPU) and GPU. Using the GPU support one needs to have an NVIDIA GPU. We installed TensorFlow with GPU support. TensorFlow using GPU, has the following dependencies:

- CUDA Toolkit 8.0
- NVIDIA drivers associated with CUDA Toolkit 8.0
- cuDNN v6.0 [45]
- GPU card with CUDA Compute Capability 3.0 or higher

5. http://www.numpy.org/

- The libcupti-dev library

Any other dependencies should be Python libraries, that will be installed alongside TensorFlow, if installing TensorFlow through Pythons package manager "*pip*".

7.2 OpenCV

OpenCV is an open source library for image and video analysis. It was originally released by Intel, and since then, programmers have worked and contributed to it as an open source project [42]. It has a plethora of "extra modules" that can be compiled into it. These modules can be found in the "opnecv_contrib" repository on OpenCVs github repository.

7.2.1 Dependencies

Depending on what version of OpenCV is wanted, and what extra modules is needed, it can have a lot of dependencies. However we did not use any special features so the "basic" version was sufficient enough for us. It is installed through Pythons package manager.

7.3 NumPy

NumPy is a library for scientific Python programming and computing. It can be installed through Python package manager.

7.4 Keras

Keras is a high-level neural networks Application Programming Interface (API), which is written in Python and can run on top of TensorFlow [41], CNTK [46], or Theano [47]. The idea with Keras was to enable fast experminentation, and be able to go from idea to result with as little delay as possible.

8 Evaluation of image classification system

This chapter describes the experimental setup and classification metrics used to evaluate the image classification system using different variations of MobileNet models. All models are trained and validated on the same set of camera trap data described in chapter 4. We compare the quality(accuracy, precision and recall) of classifications of the models.

8.1 Experimental Platform

The experiments (described in section 8.2) regarding image classification metrics were run on a desktop computer with the following specifications:

- Intel(R) Pentium(R) CPU G4400 @ 3.30GHz x 2
- GeForce GTX 960 4GB GPU @ 1241 MHz (1024 CUDA cores)
- 8GB DDR4 RAM @ 2400MHz
- Operating System: Ubuntu 16.04 LTS 64-bit with Python 3.5

The TensorFlow testing environment in our system is built with the following dependencies:

- CUDA Toolkit 8.0
- cuDNN v6.0
- TensorFlow 1.3.0
- OpenCV 3.3.0
- NumPy 1.13.3
- Matplotlib 2.1.0
- Sklearn 0.19.1

We also used a RPI for experiments regarding classification-speed and energy-consumption, this is explained in chapter 9. We did not measure classification accuracy on the RPI as it takes about 10 times longer, compared to the desktop computer. The classification accuracy is the same on the desktop computer and the RPI. This is because the model and code is the same. The difference in results of the desktop computer and the RPI, is the classification speed.

8.2 Experimental Design

We use identical training parameters for each object classification model. As they are all similar in architecture, and only differs in number of weights and image resolution input. All of the models are trained with a learning rate of 0.001 using the Stochastic Gradient Descent (SGD) optimizer, and having 4000 training steps.

We measure each model only once. The models will give the same results every time they are run, as long as the parameters and input are the same every time. The metrics we are measuring is accuracy, cross-entropy, precision, and recall.

We train each model with default input image size, which is the cropped images with size 1844 x 1382 x 3. Depending on the models input image resolution, the images will be scaled to the specific resolution. See table 8.1.

Model	Input resolution
1.0 MobileNet-224	224 x 224 x 3
1.0 MobileNet-192	192 x 192 x 3
1.0 MobileNet-128	128 x 128 x 3
0.75 MobileNet-224	224 x 224 x 3

Table 8.1: Mobilenet models with input resolution.

For each model we have two cases of classification. The first case is where the model has the classified animal as first prediction. An example would be an image containing an eagle and a crow. The model recognizes them both, but can only classify the image as one class. It is most confident on the eagle, and classifies the image as such. Even though there was a crow in the image, it is not recognized, because we only register the most confident class.

The second case is more forgiving and has the classified animal as either first or second prediction. In the example described above, this method would classify both the eagle and the crow.

This method was applied to give more leeway to the classifier, as there might be more than one animal species in an image, resulting in a higher confidence and classification score. We will discuss this further in section 10.1.1 when evaluating the classification scores.

8.3 Classification metrics

We use accuracy, crossentropy, precision and recall to evaluate the MobileNet object classification models.

Accuracy is the score we get from correctly classifying images from a test set concisting of all classes.

Cross-entropy is the score of the loss function. Both accuracy and cross-entropy is extracted from TensorFlows built-in TensorBoard application. The figures show the batch-by-batch accuracy and cross-entropy of the models. Looking at the figures we will see the moving average smoothing the measurements.

Precision is defined as the ratio of True Positive (TP) classifications to all positive classifications (TP + False Positive (FP)). Precision captures how accurate the classification model is.

Recall is defined as the ratio of TP classifications to ground truth instances (TP + False Negative (FN)) and captures how many relevant classifications are found by the classification model.

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN}$$

Precision and recall usually act like they are inversely related. When recall increases, precision falls, and vice-versa. Usually a balance between the two is preferred.

We use precision and recall for each class, but to get an average precision and recall for the model, we take the average of the metrics over all classes.

To measure the classification-speed of our model we use frames-per-second(FPS), and CPU-utilization in percent to see how effective each model is.

The precision and recall of the results are calculated by using the confusion matrices displayed under each models results, e.g(figure 8.3, 8.4). When reading a confusion matrix, the diagonal of the matrix is the TP, where as the column-wise sums without the diagonal value is the FP, and the row-wise sums without the diagonal is the FN. Using these numbers and the formulas for calculating precision and recall, we get the values in the tables in the following pages.

8.4 Results

8.4.1 MobileNet_1.0_224

In figure 8.1 we can see that the accuracy of the MobileNet_1.0_224 model is 81.1%, and figure 8.2 shows the test cross-entropy is 0.59. Table 8.2 shows the precision and recall for all the classes, as well as the average for the whole model. It takes in to consideration the case where the class is the top 1 classification, or if the class is within the top 2 classifications. Figure 8.3 and 8.4 shows the confusion matrices for the top 1 and top 2 cases.

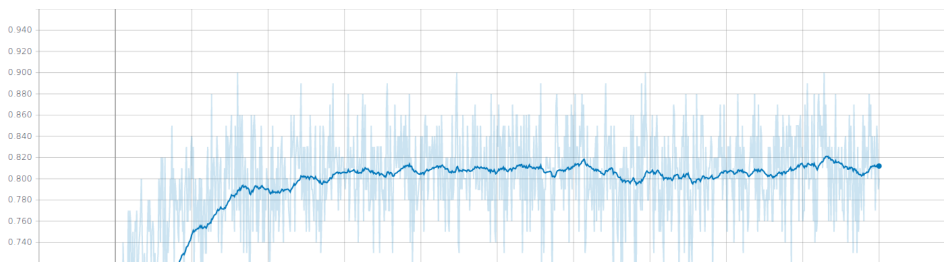


Figure 8.1: MobileNet_1.0_224 accuracy

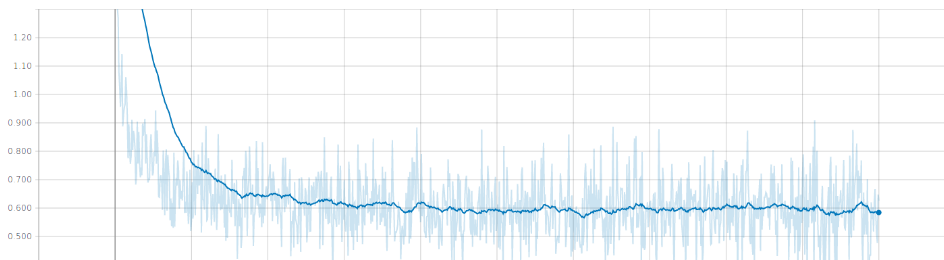


Figure 8.2: MobileNet_1.0_224 crossentropy

Class	Top 1 Precision(%)	Top 1 Recall(%)	Top 2 Precision(%)	Top 2 Recall(%)
ArcticFox	58.8	69.7	76.9	93.0
Crow	71.8	80.0	94.1	91.4
WhiteTailedEagle	68.7	82.5	82.9	97.5
GoldenEagle	92.3	30.0	100.0	67.5
Raven	88.8	20.0	95.0	47.5
RedFox	29.4	75.0	44.8	87.5
Reindeer	39.4	75.0	54.8	85.0
SnowyOwl	0.0	0.0	100.0	22.2
Wolverine	100.0	15.0	100.0	40.0
Model	61.0	49.7	83.1	70.1

Table 8.2: MobileNet_1.0_224 Precision and Recall for the top-1 and top-2 case, extracted from 307 test images.

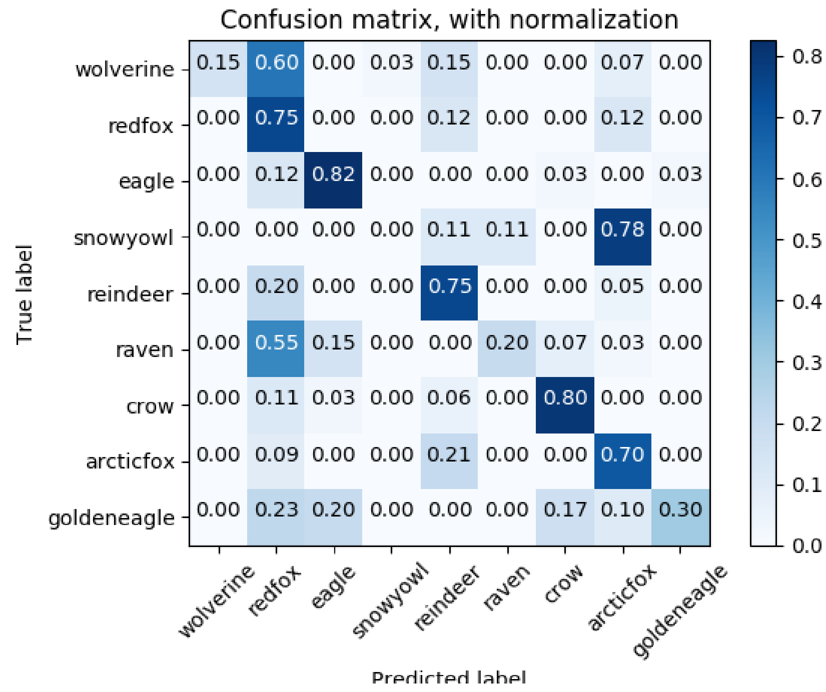


Figure 8.3: MobileNet_1.0_224 Top 1 Confusion Matrix

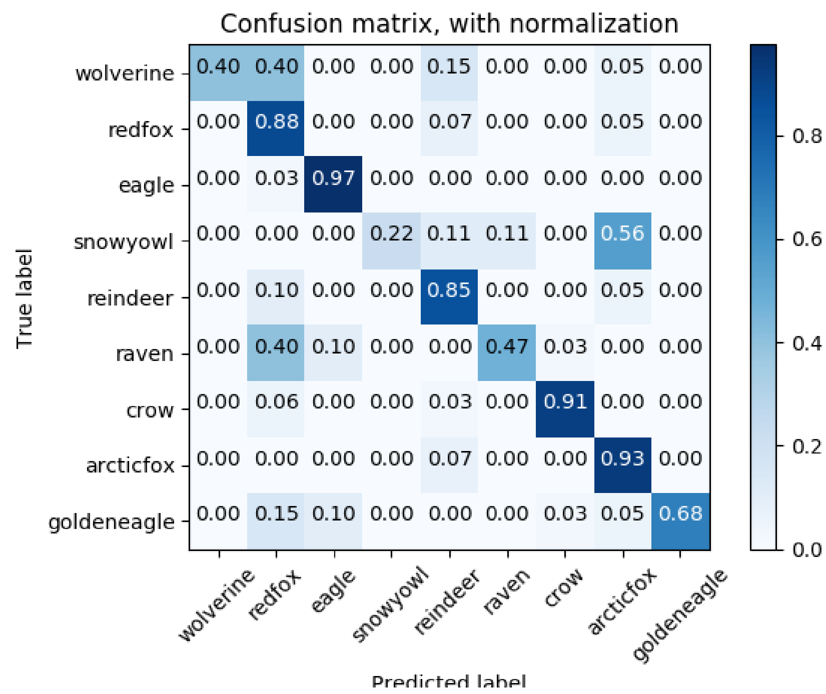


Figure 8.4: MobileNet_1.0_224 Top 2 Confusion Matrix

8.4.2 MobileNet_1.0_192

In figure 8.5 we can see that the accuracy of the MobileNet_1.0_192 model is 76.3%, and figure 8.6 shows the test cross-entropy is 0.72. Table 8.3 shows the precision and recall for all the classes, as well as the average for the whole model. It takes in to consideration the case where the class is the top 1 classification, or if the class is within the top 2 classifications. Figure 8.7 and 8.8 shows the confusion matrices for the top 1 and top 2 cases.

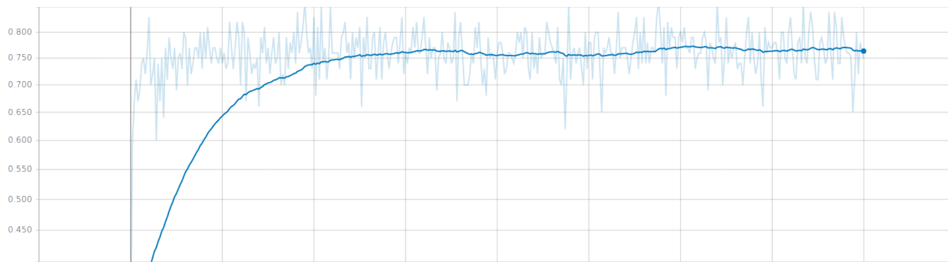


Figure 8.5: MobileNet_1.0_192 accuracy

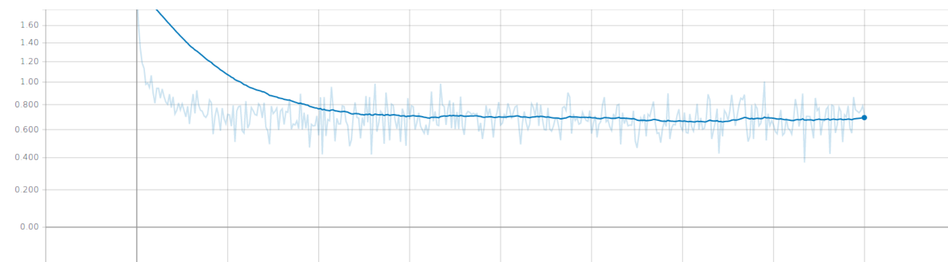


Figure 8.6: MobileNet_1.0_192 crossentropy

Class	Top 1 Precision(%)	Top 1 Recall(%)	Top 2 Precision(%)	Top 2 Recall(%)
ArcticFox	60.8	32.5	89.4	79.0
Crow	76.0	54.3	85.3	82.8
WhiteTailedEagle	46.8	92.5	61.5	100.0
GoldenEagle	100.0	5.0	100.0	37.5
Raven	66.6	5.0	87.5	17.5
RedFox	23.7	82.5	40.6	97.5
Reindeer	71.4	75.0	84.2	80.0
SnowyOwl	0.0	0.0	0.0	0.0
Wolverine	60.0	22.5	81.2	65.0
Model	56.1	41.0	69.9	62.1

Table 8.3: MobileNet_1.0_192 Precision and Recall for the top-1 and top-2 case, extracted from 307 test images.

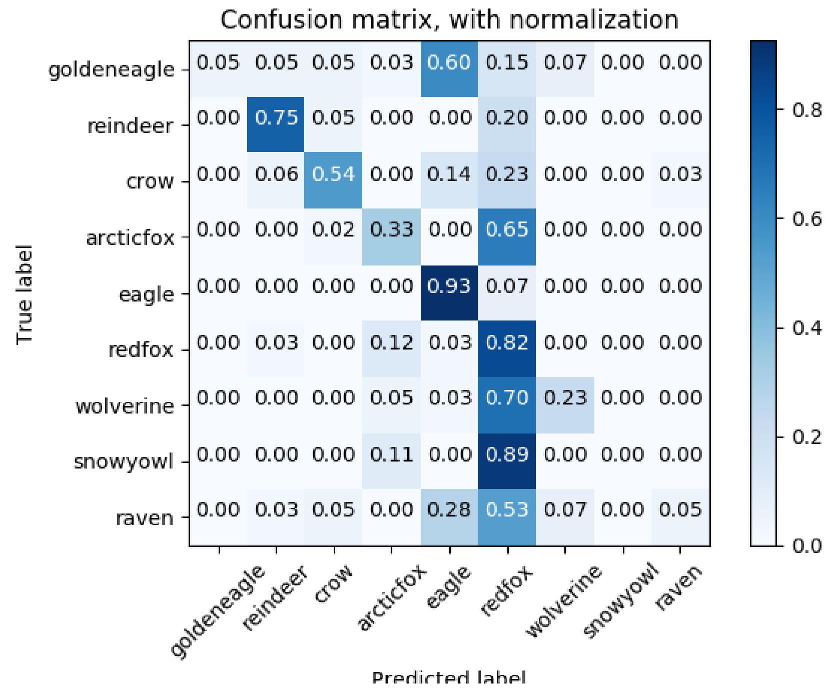


Figure 8.7: MobileNet_1.0_192 Top 1 Confusion Matrix

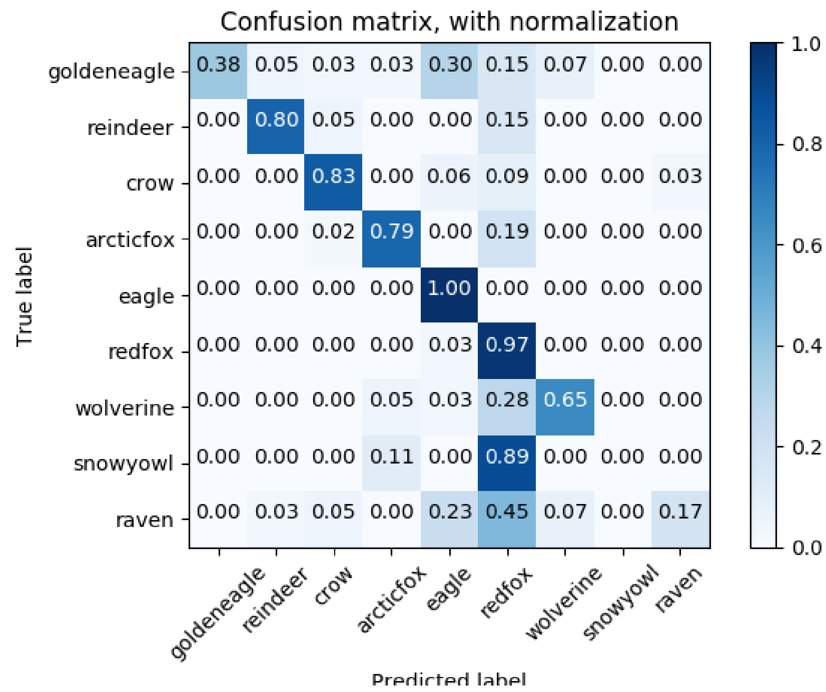


Figure 8.8: MobileNet_1.0_192 Top 2 Confusion Matrix

8.4.3 MobileNet_1.0_128

In figure 8.9 we can see that the test accuracy of the MobileNet_1.0_128 model is 75.1%, and figure 8.10 shows the test cross-entropy is 0.64. Table 8.4 shows the precision and recall for all the classes, as well as the average for the whole model. It takes in to consideration the case where the class is the top 1 classification, or if the class is within the top 2 classifications. Figure 8.11 and 8.12 shows the confusion matrices for the top 1 and top 2 cases.

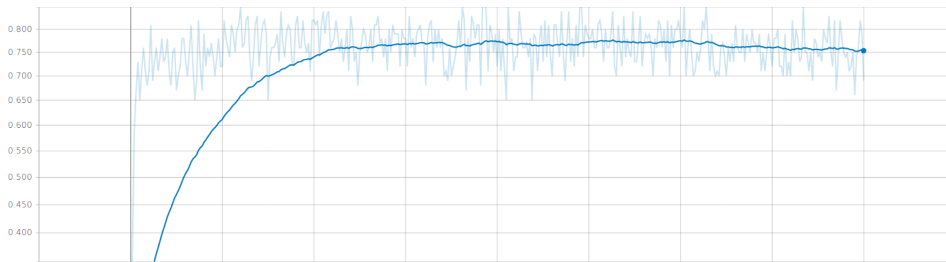


Figure 8.9: MobileNet_1.0_128 accuracy

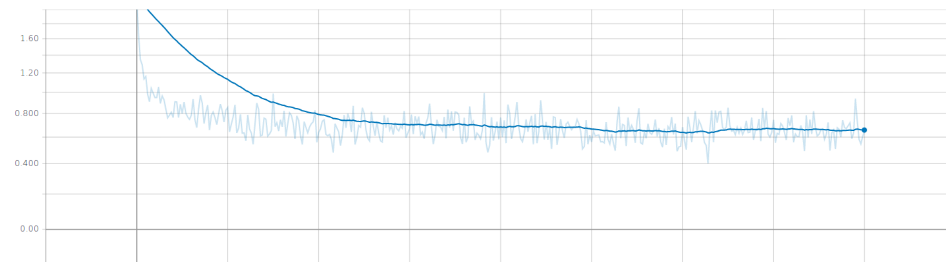


Figure 8.10: MobileNet_1.0_128 crossentropy

Class	Top 1 Precision(%)	Top 1 Recall(%)	Top 2 Precision(%)	Top 2 Recall(%)
ArcticFox	59.0	30.2	81.5	72.1
Crow	100.0	2.8	100.0	20.0
WhiteTailedEagle	41.8	82.5	51.3	97.5
GoldenEagle	100.0	7.5	100.0	12.5
Raven	83.3	12.5	92.8	32.5
RedFox	33.0	82.5	49.3	92.5
Reindeer	43.3	65.0	56.6	85.0
SnowyOwl	33.3	44.4	58.3	77.7
Wolverine	62.9	85.0	72.0	90.0
Model	61.8	45.8	73.5	64.4

Table 8.4: MobileNet_1.0_128 Precision and Recall for the top-1 and top-2 case, extracted from 307 test images.

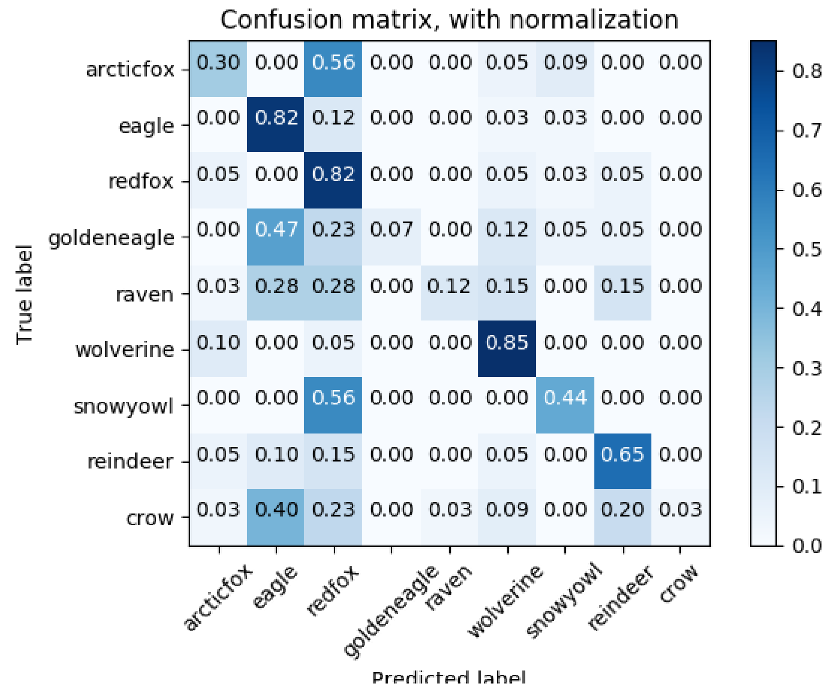


Figure 8.11: MobileNet_1.0_128 Top 1 Confusion Matrix

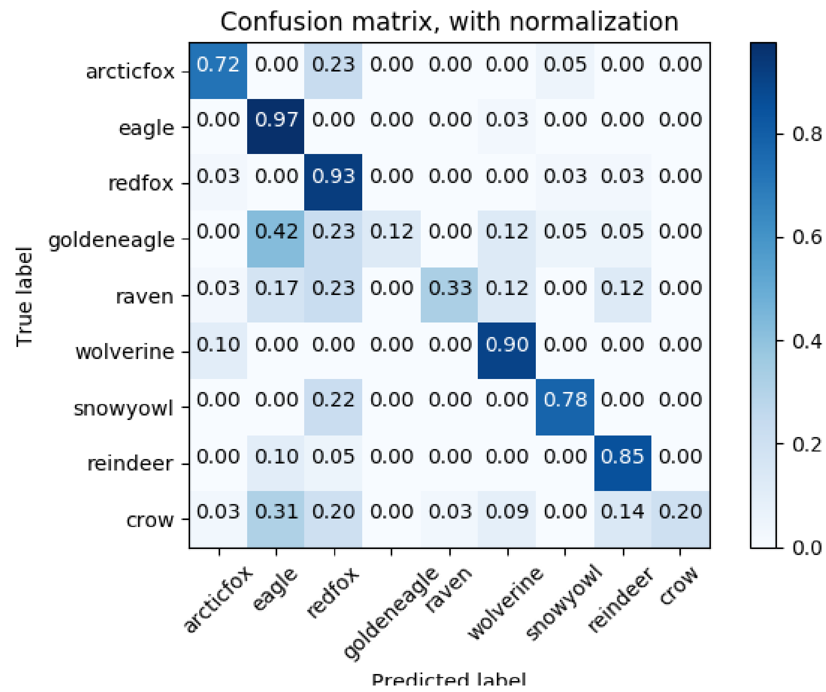


Figure 8.12: MobileNet_1.0_128 Top 2 Confusion Matrix

8.4.4 MobileNet_0.75_224

In figure 8.13 we can see that the accuracy of the MobileNet_0.75_224 model is 75.0%, and figure 8.14 shows the test cross-entropy is 0.72. Table 8.5 shows the precision and recall for all the classes, as well as the average for the whole model. It takes in to consideration the case where the class is the top 1 classification, or if the class is within the top 2 classifications. Figure 8.15 and 8.16 shows the confusion matrices for the top 1 and top 2 cases.

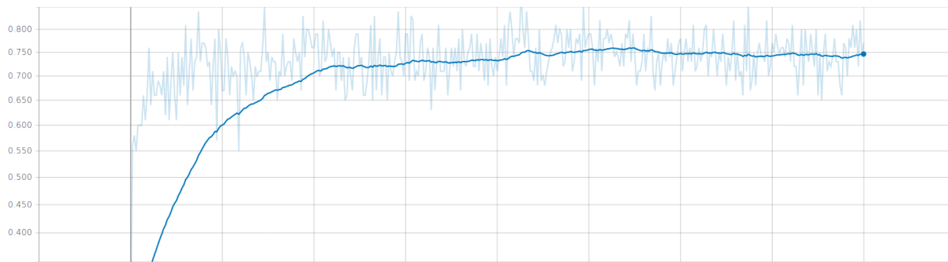


Figure 8.13: MobileNet_0.75_224 accuracy

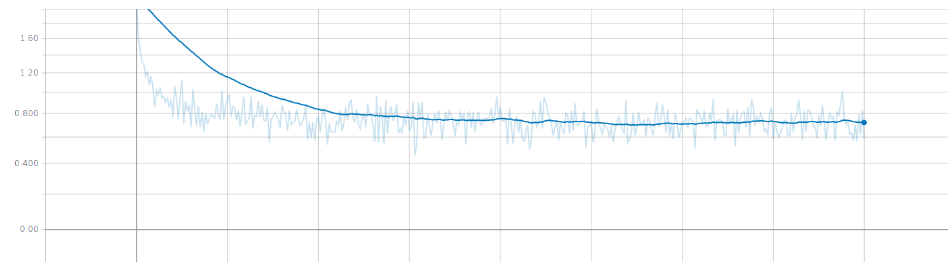


Figure 8.14: MobileNet_0.75_224 crossentropy

Class	Top 1 Precision(%)	Top 1 Recall(%)	Top 2 Precision(%)	Top 2 Recall(%)
ArcticFox	0.0	0.0	20.0	4.6
Crow	16.0	22.8	31.5	48.6
WhiteTailedEagle	67.5	67.5	75.5	85.0
GoldenEagle	100.0	2.5	100.0	10.0
Raven	87.5	17.5	92.8	32.5
RedFox	26.0	45.0	36.1	65.0
Reindeer	13.7	80.0	20.0	90.0
SnowyOwl	0.0	0.0	0.0	0.0
Wolverine	33.3	5.0	78.5	27.5
Model	38.2	26.7	50.5	40.3

Table 8.5: MobileNet_0.75_224 Precision and Recall for the top-1 and top-2 case, extracted from 307 test images.

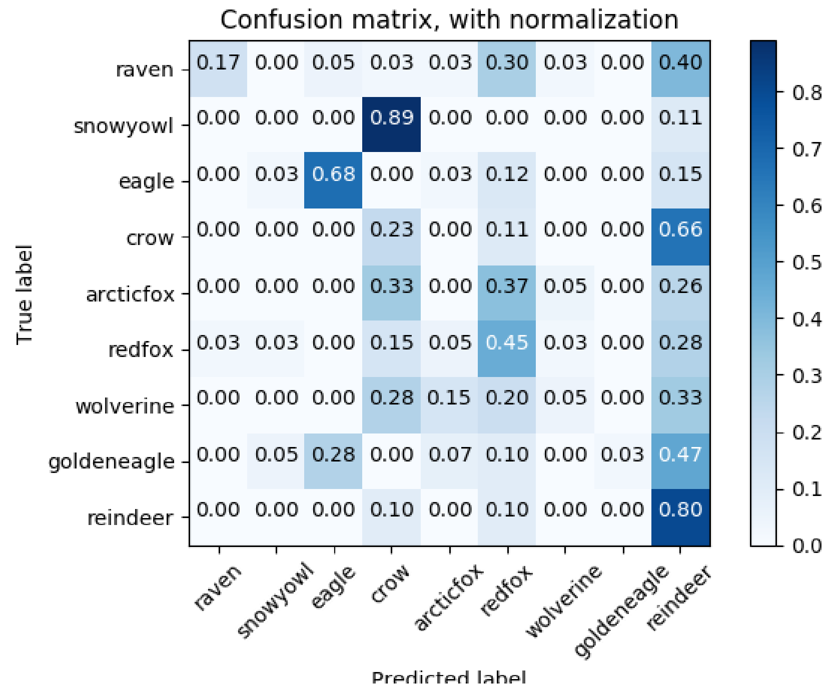


Figure 8.15: MobileNet_0.75_224 Top 1 Confusion Matrix

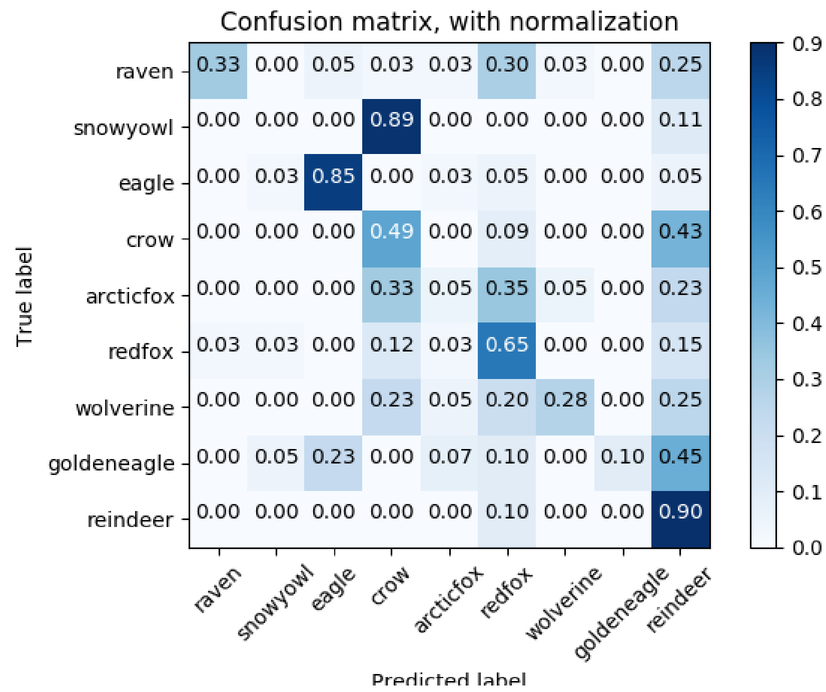


Figure 8.16: MobileNet_0.75_224 Top 2 Confusion Matrix

8.4.5 Model size

The size of the models does not vary by a lot, except for one model. The models with 100% of weights are all the same size, which we can see in table 8.6. The 0.75_224 model differs from the others, as it has 25% less weights. Looking at the table we see that it has 11 mb compared to the others 17 mb. This is explained by the reduced weights. Even though the number of weights is 25% less, the size of the model is 35% less.

Model	Size in megabytes(mb)
MobileNet_1.0_224	17
MobileNet_1.0_192	17
MobileNet_1.0_128	17
MobileNet_0.75_224	11

Table 8.6: Modelsize in megabytes(mb) for the four different variants of MobileNet.

8.4.6 Classification speed

When testing the models on the GPU computer environment, we achieved a classification speed of 12.5 images per second.

9

RPI Performance evaluation

This chapter describes the experimental setup, power metrics, and classification-speed used to evaluate the performance of the RPI for four different MobileNet models. We also see the effect the difference in model-size and image resolution has on classification speed.

9.1 Experimental Plarform

The measurements of classification-speed were run on a RPI with the following specifications:

- Quad Core 1.2GHz Broadcom BCM2837 64 bit CPU
- 1GB RAM
- Operating System: Debian Jessie With Raspberry Pi Desktop

The TensorFlow testing environment for our RPI is almost the same as on the desktop, but it does not have support for CUDA or OpenCV. There was no need for OpenCV on the RPI, as we do not do preprocessing on data, or training of

models on the RPI. The dependencies are listed below:

- TensorFlow 1.3.1
- NumPy 1.13.3
- Matplotlib 2.1.0
- Sklearn 0.19.1

9.2 Experimental Design

For measuring classification speed on the RPI, we measure the time it takes for the model to classify 50 images. We measure this 10 times, and take the average of this. The same is done for CPU-utilization. As in section 8.2 we use the models described in table 8.1.

The current of the RPI was measured with an amperemeter integrated in the circuit between the power-outlet and the RPI. We measured the range of the current over a period of 7 minutes. The RPI was idle for the first 2 minutes, and under load for the next 5 minutes. Everything external(Graphical interface, HDMI, ethernet, WiFi, Bluetooth, ...) was disabled on the RPI. It started a script on bootup, and this launched the whole process of idle state and working state. Table 9.1 shows the details of the measurement. We used the largest MobileNet model(1.0_224) for this task, as it was the most accurate model, and the one likely to be used in a production environment. This experiment was done only once.

9.3 Power metrics

Energy-consumption is measured in watt-hours(Wh), which is an energy unit equivalent to one watt of power expended over one hour of time. Energy is equivalent to power multiplied by time. Determining energy in watt-hours, means that power must be expressed in watts and time must be expressed in hours.

Watts are composed of volts multiplied with ampere.

$$Watt = Volt * Ampere \qquad Watthours = Watt * Hours$$

We measured the range of Milliampere (mA) the RPI used for the period of measuring.

When knowing the average of mA, and the voltage, we can calculate the watt, and in turn calculate the watt-hour, which is our measurment for energy-consumption.

9.4 Energy expenditure of RPI executing MobileNet_1.0_224

Device	Raspberry Pi 3 Model B
Idle(mA)	103
Inference(mA)	156-430 (mostly between 180-260)
CPU	ARM Cortex-A53
#cores	4
CPU clock	1.2GHz
RAM	1GB
Storage(SD)	16GB
VDD(V)	5

Table 9.1: Measurements of Raspberry Pi 3 Model B being idle and doing image classification with the MobileNet_1.0_224 model.

We only measured the power consumption of the MobileNet_1.0_224 model, as it was the best performing model for image classification. Due to restricted access to equipment for measuring power consumption, we were only able to measure the aforementioned model.

By taking the average of the mA for the RPI under load, we get an average current of 293 mA, which is 0.293 amperes. Knowing that the RPI uses 5 volts as input, we get the watt by multiplying volts with amperes. Following the formula described in 9.3 we get:

$$1.465W = 5V * 0.293A$$

The energy expenditure for 5 minutes under load, the RPI uses

$$0.122Wh = \frac{5minutes}{60minutes} * 1.465W$$

We can expect the other MobileNet models of the same size(1.0) to have about

the same consumption, as they have the same amount of weights. The 0.75 model however could differ, as it has less weights to compute.

9.5 Classification speed

When testing the models on the RPI we achieved a speed of 1.17 to 1.54 images per second on average depending on the model. The RPI does not have any form of cooling mounted.

Model	FPS	Time in sec per 50 frames	Minimum CPU load(%)	Maximum CPU load(%)
MobileNet_1.0_224	1.17	42.66	48.8	57.9
MobileNet_1.0_192	1.27	39.26	45.5	52.0
MobileNet_1.0_128	1.54	32.35	35.7	43.9
MobileNet_0.75_224	1.31	38.09	45.9	51.4

Table 9.2: Classification speed for the four different models, measured on the RPI 10 times, and taking the average of the results.

Looking at table 9.2, we can see that the fastest model is the one with 128 x 128 pixel resolution. and the slowest is the one with 224 x 224 and 100% of the weights. The model with reduced weights (0.75) and 224 x 224 pixels, is the second fastest. As we saw in table 8.5, the reduced weights model had significantly worse classification score compared to the other models, so even though it is slightly faster than the 1.0_224 model, it still has close to half the precision and recall.

10 Discussion

This chapter will discuss our results and how we solved the problem of doing image classification on a small embedded computer. We describe the difficulty of keeping a high classification accuracy, while reducing the model size drastically. We also discuss how we worked to solve this challenge by training different variants of our classificaiton model, investigating their results, and suggesting ways to improve the system.

10.1 Evaluating Results

We compare 4 different models of MobileNet, see table 8.1. The results show that different resolution in input images can affect both precision and recall for the different classes. Some classes benefit from a higher resolution where others benefit from a lower resolution. Reducing the number of weights seems to only affect the classifications in a negative way, this is discussed below.

10.1.1 Classification

Model	Top 1 AP(%)	Top 1 AR(%)	Top 2 AP(%)	Top 2 AR(%)
MobileNet_1.0_224	61.0	49.7	83.1	70.1
MobileNet_1.0_192	56.1	41.0	69.9	62.1
MobileNet_1.0_128	61.8	45.8	73.5	64.4
MobileNet_0.75_224	38.2	26.7	50.5	40.3

Table 10.1: MobileNet comparison for AP and AR for the top-1 and top-2 case, extracted from tables 8.2,8.3,8.4,8.5

Looking at classification results of our four different models in table 10.1, it is clear that resolution and number of weights have an effect on precision and recall. Comparing models, we see that the MobileNet_1.0_224 model is the highest performing one, having an AP for the whole model of 61.0% and AR of 49.7%. This is for the case where we only register a correct classification if the classified animal is the highest ranked classification. If we register a correct classification where the animal is either first or second, we get better results. The AP and AR are then 83.1% and 70.1% respectively.

The second best performing model is the MobileNet_1.0_128. It has less resolution than MobileNet_1.0_192, but has better scores. Comparing the 128 model up against the 224 model for the top-1 case, we can see that the 128 model are better at recalling the wolverine, snowyowl and redfox classes than the 224 model, see table 10.2. The balance between precision and recall in these classes for model 128, is preferred to the precision/recall balance in model 224. In the remaining classes (arcticfox, crow, whitetailedeagle, goldeneagle, raven and reindeer) the 224 model has better recall and precision/recall balance.

Class	224 Top 1 Precision(%)	224 Top 1 Recall(%)	128 Top 1 Precision(%)	128 Top 1 Recall(%)
ArcticFox	58.8	69.7	59.0	30.2
Crow	71.8	80.0	100.0	2.8
WhiteTailedEagle	68.7	82.5	41.8	82.5
GoldenEagle	92.3	30.0	100.0	7.5
Raven	88.8	20.0	83.3	12.5
RedFox	29.4	75.0	33.0	82.5
Reindeer	39.4	75.0	43.3	65.0
SnowyOwl	0.0	0.0	33.3	44.4
Wolverine	100.0	15.0	62.9	85.0
Model	61.0	49.7	61.8	45.8

Table 10.2: MobileNet_1.0_224 and MobileNet_1.0_128 Precision and Recall for the top-1 case.

The 192 model scores worse than the 128 model. A reason for this might be that 128 x 128 pixel inputs correspond better with the 1844 x 1382 pixel resolution, which is the resolution we use at runtime.

The 224 model with reduced weights (0.75) was the worst performing model. The reason for the worse result is less weights. Having 25% less weights than the other models, it will have less capacity to know features, which in turn makes it less capable of classifying accurately.

MobileNet_1.0_224 and MobileNet_1.0_128 have a similar score in both AP and AR when looking at the top-1 case. The difference in score increases if we take top-2 into consideration. It is clear that the difference in input resolution for the models affects the score for individual classes. Where the 128 model is better at wolverine, snowyowl and redfox, the 224 model performs better on the rest. This applies for both top-1 and top-2. A reason for the difference in classification score on individual species might be the size of the animal itself.

An example could be; an image with a redfox contains a lot of crows surrounding it. When reducing the size of the image down to 128 x 128 pixels from 1844 x 1382, the crows might have shrinked in size so much that they become unrecognizable. A crow with features like wings and a beak, could become 3 unrecognizable pixels in a blob with such downsampling of image resolution. What remains in the image would be the more distinct features of the redfox. With no "obvious" crows in the image, the more distinct redfox would be easier to classify as there is a lower amount of disturbing features in the image.

This is speculation, but could be an explanation of why the smaller resolution model is better at predicting some species compared to the larger resolution model.

10.1.2 Classification speed

As briefly discussed in section 9.5, we see that the 1.0_128 model is the fastest one. The second fastest is the 0.75_224, and this is due to the reduced number of weights compared to the other three models. As previously mentioned, the 0.75_224 model is the second fastest model, but it is the worst model in classification quality by far. Compared to our best performing model, the 1.0_224, the 0.75_224 has almost half the classification score.

Comparing the 1.0_128 and the 1.0_224, the first one has a classification speed of 1.54 FPS where the second has a speed of 1.17 FPS. That is 0.37 FPS in difference. As we discussed in section 10.1.1, these are the two best performing

models when it comes to classification score.

To decide which model to use in a production environment, a tradeoff have to be made between classification score and classification time. As we can see from table 9.2, the 1.0_128 model uses about 10% less CPU load to do its classification. This could be connected to less power consumption overall. As it also uses less overall time to classify a set of images, it is "working" less than the other models.

If battery capacity is restricted, then the best choice might be the 1.0_128 model, even though it was worse than the 1.0_224 at classifying overall, it was better in some classes. If however classification score is more important, and the power consumption of the 1.0_224 model is not too much higher, that might be the best one. Choosing any of the other two models would be a worse tradeoff no matter what, as the 1.0_128 model performs better than both of them, and is faster to classify, and uses less CPU load.

When measuring the classification speed on the RPI we used 50 images. When testing with 300 images, the RPI crashed due to overheating. In our experience the RPI is able to handle a small batch like 50 images or less, without cooling.

10.1.3 RPI Energy expenditure

In section 9.4 we saw that the RPI used 0.122Wh for 5 minutes of work. A common car-battery has about 1000 watt-hours. As previously calculated, the RPI uses 1.465W, and with 1000 watt-hours this becomes 682.6 hours of continuous work under load. With 1.17 FPS, our system would be able to analyze 2.1 million images on a car-battery with 1000 watt-hours.

10.2 Idea

Within the COAT project, previous work related to ours, have involved image classification [38] and object detection [39]. The resulting NNs in these projects have had significant model sizes (1000MB), and high accuracy scores (93%). These projects have used high-end computers for their training and analysis. With "unlimited" electrical power, and heavy-duty graphic cards for processing, model size would be no problem. As a result of large models containing 10's of millions of weights, the accuracy of the models increase. Exporting said models to a small embedded computer with limited electrical power capacity, and little computing power compared to high-end GPUs, would not work. In order to

classify images *in-situ* on such low-end devices, we need a small model which fits in the embedded computers memory, and is fast enough in computation to not consume too much of the battery power, which in turn would decrease its lifespan.

Looking into new research [7] [6] on small neural networks, we found Googles MobileNet [7] a good candidate for image classification on low-end embedded devices.

As previous work within the COAT project had resulted in a labeled dataset, we took advantage of this and used it as our training data.

10.3 Dataset

Our dataset is described in chapter 4. Here we will discuss the shortcommings with the dataset, and ways to improve it.

As our dataset is imbalanced, in the sense where some classes are over-represented compared to others, our model will become good at recognizing certain classes, and bad at recognizing others. The ideal would be to have an even distribution of classes.

As an attempt to tackle this imbalance, we synthesized more images in some under-represented classes. This was done through data-augmentation. In the SnowyOwl class, which was severly under-represented, we created 25 images out of 1 image. By horizontally flipping it, as well as slight rotations we could do this. To keep the aspect ratio of the image after rotating, we needed to synthesize some pixels. This resulted in un-natural looking images, and by testing this on a dataset containing original images, and augmented ones, we achieved a worse test accuracy than the result we got with only the original images. Due to this we discarded the idea of augmenting the images in this way.

Another attempt at augmenting images for better performance, was to get rid of as much of the background as possible. Using an algorithm to find key-points(interesting clusters of pixels) in an image, we were able to find the most distinct animals in an image, and cut them out to create a new image. This resulted in a large variation of image sizes, and were not very accurate for all species, where it in some cases would ignore the animals completely, and focus on a tree or a rock. We abandoned this idea as well, and settled with the idea of cutting away the black borders of the images, as explained in chapter 4.

Even though we were unsuccessful with some ways of augmenting the data, we believe that there are ways to do it successfully. We will describe this further in chapter 11.1.

10.4 Crowded images

As we are doing image classification, not detection, we have inherent difficulties when it comes to classifying several animal species at once. Our classifier uses a softmax layer, which have a 100% confidence in total. It distributes this confidence over several classes. An example could look like table 10.3.

Class	**Prediction (%)**
WhiteTailedEagle	99.8
GoldenEagle	0.1
Raven	0.05
RedFox	0.04
Reindeer	0.01

Table 10.3: Softmax example.

Say there are both an eagle and a raven in the image of the example above. It is not wrongly classifying the eagle which is the highest prediction, but it is missing out on the raven. Because there are images with several species, it is hard to predict all of them, as the model will favor some class, in this case the eagle.

We experimented with a way of classifying several species in one go. It was to consider it a correct classification for a class if it was within the top 2 predicted classes. It increases the accuracy of the model by a significant amount, but we think a better way to handle it would be with object detection. Object detection is done in the COAT project, but not for small models like ours. It would be an obvious next step in the improvement of this project.

10.5 Issues with exporting models

The development of the CNN was done in a simulated environment. This is where we created our model architecture, tuned parameters, and trained the model. When achieving a satisfactory result of the model, we decided to export it for testing on a RPI. Due to problems still unknown to us, we were not able to

"freeze" the model and export it in a usable format for loading into the RPI. Luckily TensorFlow [41] had support scripts for training the MobileNet architecture. We used their open-source script located in the TensorFlow core library under filepath: `tensorflow/tensorflow/examples/image_retraining/retrain.py` to export a MobileNet model. The script let us train a model on the training data we gave it, using the desired MobileNet architecture.

10.6 Batching classifications

The classification system on the RPI takes a few seconds in setting up. If this is done for every image, it will quickly add up, and over time have used more energy. Due to this, we batch our images. The RPI can then initate the system once per day, and then run inference on the batched images. When the classification is done, it could wait until the next day for a new batch of accumulated images.

11 Conclusion

In this thesis, we have implemented a system that classifies images of animals from the Arctic tundra on a small embedded computer using a small neural network. We have given detailed description of the dataset and the way it was prepared, as well as describing the concept and training of CNNs which is the type of NN we used for our model. We described and analyzed four different variations of the MobileNet NN, comparing their classification metrics, as well as power usage and classification speed.

Our experiments showed that a small mobile NN like MobileNet, can classify a range of animal classes, doing the inference on a small embedded computer like the RPI with a classification speed of 1.17 FPS and an AP and AR of 61.0% and 49.7% respectively for the case of only classifying the top prediction as a hit. Where as accepting the classification of the top 2 predictions, we have an AP and AR of 83.1% and 70.1% respectively.

11.1 Future Work

Several improvements can be made for our image classification system. We expect the classification accuracy to increase for several classes of animal species if more work is put into dataset preparation. [40] shows that image synthesizing, in a way where they take their own background image, and synthesizes new ones with different stock photos of animals taken from Google image search is possible, and gives good results in classification.

Applying object detection to the system like [30] and [39] did, could possibly improve the recognition of the animals, as well as be able to recognize more than one animal in an image.

With power consumption being a central part in *in-situ* systems without continous power supply, a method for hibernating the small computer could save power. In our case where we use a RPI as the small computer, a Sleepy Pi[1] could be a candidate for integrating a hibernation mode.

1. https://spellfoundry.com/product/sleepy-pi-2/

Bibliography

[1] Åshild Ø. Pedersen, A. Stien, E. Soininen, and R. A. Ims, "Climate-ecological observatory for arctic tundra - status 2016," in *Fram Forum 2016*, pp. 36–43, Mar 2016.

[2] A. F. O'Connell, J. D. Nichols, and K. U. Karanth, eds., *Camera Traps in Animal Ecology: Methods and Analyses*. Springer US, 2011.

[3] S. R., M. Hebblewhite, R. Kays, J. Ahumada, J. Fisher, C. Burton, S. Townsend, C. Carbone, J. M. Rowcliffe, J. Whittington, J. Brodie, J. A. Royle, A. Switalski, A. Clevenger, N. Heim, and L. Rich, "Scaling up camera traps: monitoring the planets biodiversity with networks of remote sensors," in *Frontiers in Ecology and Environment*, vol. 15, pp. 26–34, Dec 2016.

[4] J. Deng, N. Ding, Y. Jia, A. Frome, K. Murphy, S. Bengio, Y. Li, H. Neven, and H. Adam, *Large-Scale Object Classification Using Label Relation Graphs*, pp. 48–64. Cham: Springer International Publishing, 2014.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[6] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016.

[7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[8] G. Chen, T. X. Han, Z. He, R. Kays, and T. Forrester, "Deep convolutional neural network based species recognition for wild animal monitoring,"

pp. 858–862, 01 2015.

[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[10] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *CoRR*, vol. abs/1408.5093, 2014.

[11] J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, June 2009.

[12] A. Khosla, N. Jayadevaprakash, B. Yao, and L. Fei-Fei, "Novel dataset for fine-grained image categorization," in *First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition*, (Colorado Springs, CO), June 2011.

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "Imagenet large scale visual recognition challenge," *CoRR*, vol. abs/1409.0575, 2014.

[14] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," *CoRR*, vol. abs/1705.03122, 2017.

[15] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1725–1732, June 2014.

[16] M. Browne and S. S. Ghidary, *Convolutional Neural Networks for Image Processing: An Application in Robot Vision*, pp. 641–652. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[17] I. Biederman, "Recognition-by-components: A theory of human image understanding," *Psychological Review*, vol. 94, pp. 115–147, 1987.

[18] A. Ng, K. Katanforoosh, and Y. B. Mourri, "Case studies - classic networks, week 2, convolutional neural networks," *COURSERA: Convolutional Neural Networks*, 2017.

[19] A. Deshpande, "A beginner's guide to understanding convolutional neural networks," 2016. 20.11.2017.

[20] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee, “Understanding deep neural networks with rectified linear units,” *CoRR*, vol. abs/1611.01491, 2016.

[21] A. Ng, K. Katanforoosh, and Y. B. Mourri, “Convolutional neural networks - pooling layers, week 1, convolutional neural networks,” *COURSERA: Convolutional Neural Networks*, 2017.

[22] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, *Efficient BackProp*, pp. 9–48. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[23] T. T. and H. G., “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude,” *COURSERA: Neural Networks for Machine Learning*, pp. 26–30, 2012.

[24] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2014.

[25] A. Y. Ng, “Preventing "overfitting" of cross-validation data,” in *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML ’97, (San Francisco, CA, USA), pp. 245–253, Morgan Kaufmann Publishers Inc., 1997.

[26] D. M. Hawkins, “The problem of overfitting,” *Journal of Chemical Information and Computer Sciences*, vol. 44, no. 1, pp. 1–12, 2004. PMID: 14741005.

[27] S. J. Nowlan and G. E. Hinton, “Simplifying neural networks by soft weight-sharing,” *Neural Computation*, vol. 4, no. 4, pp. 473–493, 1992.

[28] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[29] B. Neyshabur, “Implicit regularization in deep learning,” *CoRR*, vol. abs/1709.01953, 2017.

[30] B. Wu, F. N. Iandola, P. H. Jin, and K. Keutzer, “Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving,” *CoRR*, vol. abs/1612.01051, 2016.

[31] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: efficient inference engine on compressed deep neural network,” *CoRR*, vol. abs/1602.01528, 2016.

[32] O. Dürr, Y. Pauchard, D. Browarnik, R. Axthelm, and M. Loeser, "Deep learning on a raspberry pi for real time face recognition," 01 2015.

[33] M. S. Norouzzadeh, A. Nguyen, M. Kosmala, A. Swanson, C. Packer, and J. Clune, "Automatically identifying wild animals in camera trap images with deep learning," *CoRR*, vol. abs/1703.05830, 2017.

[34] A. Swanson, M. Kosmala, C. Lintott, R. Simpson, A. Smith, and C. Packer, "Data from: Snapshot serengeti, high-frequency annotated camera trap images of 40 mammalian species in an african savanna," 2015.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[36] A. Boulch, "Sharesnet: reducing residual network parameter number by sharing weights," *CoRR*, vol. abs/1702.08782, 2017.

[37] H. Gao, L. Dou, W. Chen, and J. Sun, *Image classification with Bag-of-Words model based on improved SIFT algorithm*, pp. 1–6. 06 2013.

[38] H. Thom, "Automatic rodent identification in camera trap images using deep convolutional neural networks.," *Capstone Project*, Dec 2016.

[39] H. Thom, "Unified detection system for automatic, real-time, accurate animal detection in camera trap images from the arctic tundra.," *Masters Thesis*, Jun 2017.

[40] A. R. Elias, N. Golubovic, C. Krintz, and R. Wolski, "Where's the bear? - automating wildlife image processing using iot and edge cloud systems," in *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pp. 247–258, April 2017.

[41] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zhang, "Tensorflow: A system for large-scale machine learning," *CoRR*, vol. abs/1605.08695, 2016.

[42] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek, "A brief introduction to opencv," in *2012 Proceedings of the 35th International Convention MIPRO*, pp. 1725–1730, May 2012.

[43] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167,

2015.

[44] R. Collobert, S. Bengio, and J. Marithoz, "Torch: A modular machine learning software library," 2002.

[45] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014.

[46] S. H. Hashemi, S. A. Noghabi, W. Gropp, and R. H. Campbell, "Performance modeling of distributed deep neural networks," *CoRR*, vol. abs/1612.00521, 2016.

[47] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermüller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. B. Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P. L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M. Côté, M. Côté, A. C. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. E. Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. J. Goodfellow, M. Graham, Ç. Gülçehre, P. Hamel, I. Harlouchet, J. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrançois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P. Manzagol, O. Mastropietro, R. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. J. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, É. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. P. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, "Theano: A python framework for fast computation of mathematical expressions," *CoRR*, vol. abs/1605.02688, 2016.