UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Object detection at the edge

Image classification on a small embedded computer

—

Truls Mathiassen
INF-3981, November 2020

“Eg har masse data”
–Lars Vaular

“Don’t push me ’cause I’m close to the edge.”
–Grandmaster Flash

Abstract

While monitoring rodents in the Arctic Tundra to evaluate if climate changes affect the ecosystem. The camera-traps of the COAT project generates image data in large scale each year. To manually examine the data in regards to labeling is a tedious and time-consuming job, and a more efficient and automated tool for the task is required.

In this thesis we presents the architecture, design and implementation of a object classification model deployed on a small embedded computer, to be used on the gathered image data in order to classify and label the animals at the edge.

We conduct transfer-learning on the state-of-the-art pre-trained YOLOv4-tiny model by introducing a labeled COAT image set. We utilize the Convolutional Neural Network of the model to do predictions on a test image set in order to evaluate the model. The result is an application with an embedded model able to predict labels with an accuracy of 96.07% and inference time that classifies it to do so in real-time.

Acknowledgements

First and foremost, I would like to thank my family for supporting me, pushing me, and having faith, even when I was doubting.

I would like to express my gratitude to my supervisor John Markus Bjørndalen for all help and support on my terms throughout this project. A special thanks to Jan Fuglesteg for the flexibility regarding the ongoing pandemic. To Kai-Even Nilssen, thanks for all the conversations and discussions.

To all my classmates, thanks for some fantastic years at UiT. I will forever remember the good times, debugging, and late nights.

Contents

Abstract **iii**

Acknowledgements **v**

List of Figures **ix**

List of Tables **xi**

Acronyms **xiii**

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

2 Object classification **5**
2.1 Convolutional Neural Networks 6
2.1.1 Architecture 6
2.1.2 Training 9
2.1.3 Transfer learning 10

3 Related work **11**

4 Data-set preparation **13**
4.1 COAT data-set 14
4.1.1 Data-set preparation 14

5 Design And Architecture **17**
5.1 Workflow Design 18
5.1.1 Preparation 18
5.1.2 Training 18

6 Implementation **23**
6.1 Google Colab implementation 24

6.1.1 Android Application Implementation 24

7 Evaluation **27**
7.1 Experimental platforms 27
7.1.1 TensorFlow lite model platform 27
7.1.2 YOLOv4-tiny model platform 28
7.2 Experimental design 28
7.3 Classification metrics 29
7.3.1 Concepts 29
7.4 Results 31

8 Discussion **35**
8.1 Results 35
8.2 Data-set 36
8.3 Training platform 37
8.4 Edge computing 38

9 Conclusion **41**
9.1 Future Work 42

Bibliography **43**

A Roboflow API screenshots **47**

List of Figures

2.1 CNN architecture illustration from[8] 6
2.2 Basic flow in transfer learning[14] 10

4.1 Camera-trap image of a lemming 14
4.2 Annotated image of a lemming 15

5.1 Performance comparison of small vs full-scale models in regards to AP illustration from [27] 19
5.2 Architectural workflow design 22

6.1 On device screenshots of Android application for object classification 25

7.1 Colab evaluation metrics for YOLOv4-tiny with the validation data-set as input 31
7.2 Example on failure when detecting multiple objects from the data-set images 33
7.3 screenshot predictions form smartphone APP 34

8.1 screenshot predictions of bird from smartphone APP 39

1 Roboflow Dashboard 47
2 Dataset dashboard 48
3 Roboflow data-set version dashboard 49
4 Single image from data-set 50

List of Tables

4.1 Data-set image distribution. Shows the image count of each label. 14

7.1 mean Average Performance for each model and AP for each class 31

List of Abbreviations

AI Artificial Intelligence

AP Average Precision

API Application Programming Interface

CNN Convolutional Neural Network

COAT Climate-ecological Observatory for Arctic Tundra

COCO Common Objects in Context

CPU Central Processing Unit

DAO Distributed Arctic Observatory

DNN Deep Neural Network

FN False Negative

FP False Positive

GPU Graphical Processing Unit

IDE Integrated Development Environment

IOU Intersection over Union

MAP mean Average Precision

SSD Single Shot Detection

TP True Positive

TPU Tensor Processing Unit

VM Virtual Machine

YOLO You Only Look Once

1 Introduction

The effects of global warming are changing the arctic tundra dramatically, and the effects on the ecosystem is expected to greatly impact the various animal species currently living there. If temperatures are rising, which impacts the permafrost, this could lead to a whole new ecosystem, and a rapidly changing habitat with widespread consequences.

In order to keep track of changes in the wildlife and climate on in the arctic tundra, the COAT[1] was founded as a collaborating research group with members from various institutions of the FRAM center[2].

The COAT-team conducts adaptive long-term research in the face of climate change. One key monitoring objective in their research is a ground-based optical monitoring system[3]. This system uses wildlife camera-traps, which are mounted inside of artificial tunnels used by small mammals/rodents. The camera has motion detection and takes pictures when it gets triggered. With multiple traps in various areas, this generates a copious amount of data each year. Researchers spend much time conducting tedious manual labor on the data sets to produce statistics about the type, amount, frequency, and variation of animals passing through the traps.

In collaboration with the Department of Computer Science at UiT, multiple tools have been developed to make the analytic procedure available on embedded edge-computing nodes. Several image classification models have been trained and tested by researchers at UiT[4] with impressive accuracy on annotated

images.

All data produced by the camera-traps still require human interaction in order to extract the data since this is stored on a memory card in the camera module. In this thesis, we will explore the possibility of developing a lightweight image classification module that will enable edge computing. Such a model could be utilized by a small embedded computer and possibly mounted in correlation to the camera-traps to retrieve and process the data and generate statistics. The benefactor would be less human pollution of the biodiversity-ecosystem in the Arctic Tundra.

1.1 Problem definition

In this thesis, we investigate the possibility of developing a lightweight image classification model to enable edge computing on data in remote locations without the need for extensive computational power.

To find out if this is possible, we need to train a full-scale image classification model on a data-set from the COAT camera-traps. Then convert it into a lightweight model. Tensorflow Lite[1] by Google will be our final model. This will be deployed on a small embedded computer(smartphone) and tested on images from the COAT camera-traps. The results verify if the model accuracy and inference time are sufficient enough to be considered a potential candidate for real-time edge computing in the field.

The final model embedded on the smartphone should be

- Capable to detect and classify the various animals in the COAT data-set
- Classify the animals with a satisfactory accuracy to be considered a success
- Conduct the classification with an inference time that qualifies it to conduct real-time classifications
- As small as possible in order to be loaded into the edge device memory.

We outline the concept of object classification based on Convolutional Neural Networks. Describe the architecture of a CNN and how transfer-learning is conducted. We focus on the importance of data-set preparations and how this

1. https://www.tensorflow.org/lite

affects the model training. We present the implementation, design, architecture, and workflow for the diversity of components in this thesis. The model metrics and performance are evaluated and discussed before concluding and proposing future work for our project.

1.2 Contributions

This thesis makes the following contributions:

- Introduction to object detection and a description of the architecture in CNNs.
- An Detailed description of data-set preparation and transfer-learning methods for object classification tailored to classify animals from data-set collected by the COAT research group camera-traps.
- The implementation of an object classifier deployed on a small embedded computer.
- Evaluation of the models performance metrics.
- Insights in state-of-the-art cloud environment specialized for machine learning.

1.3 Outline

This thesis is structured into 9 chapters including the introduction.

Chapter 2 describes object classification, and a introduction to the design and architecture Convolutional Neural Networks.

Chapter 3 presents related work in the field of object classification, in comparison to the work done in this thesis

Chapter 4 describes the data-set used in this thesis, and how it is prepared for training.

Chapter 5 describes the design and architecture for this project with regards to data workflow.

Chapter 6 describes the implementation and dependencies in this thesis

Chapter 7 Outlines the experimental platforms in this thesis and evaluate the object classification models in regards to experimental design and results.

Chapter 8 discusses the results, dependencies, data-set and edge computing for this project.

Chapter 9 concludes this thesis, and suggest future work to improve our model and application

2

Object classification

Classification of an object in a digital image or video stream is the task of locating and processing one or more objects in order to act upon it, based on the developers desired output. This could be validation, edge detection, color adaption, face recognition, and many more. With more powerful technology, this is even standard features on many day-to-day devices used worldwide, such as the Artificial Intelligence camera feature of Huawei smartphones[5], which is just one of the many examples of where object detection and classification are used as a integrated tool.

In this chapter, an introduction to Convolutional Neural Network is given, with a thorough architecture description

2.1 Convolutional Neural Networks

Were first introduced in the 1980s by Yann LeCun. He developed a multi-layer neural network, LeNet[6], which combined convolutional neural network with backpropagation algorithms to make a computer classify hand written numbers. Splitting the problem into multi-layered networks instead of single-layer networks gained a severe performance profit. In 2012 AlexNet[7] won the ILSVRC-2012 competition working with the ImageNet dataset containing 1.2million images from 1000 different classes. Which achieved an impressive 15.3% error rate, compared with the runner up had 26.2%. This revolutionized the image classification of computer vision in regards to processing human visual input. Since then, **CNNs** has been state-of-the-art in machine learning and artificial intelligence when it comes to human visual inputs(objects, hand-writing, facial recognition, image classification, and many more) And is widely used in smartphones and AI tools worldwide.

2.1.1 Architecture

Convolutional Neural Network (CNN) are is a subcategory of Deep Neural Networks(**DDNs**), Much like **DNNs** it consists of neurons with learnable weight and biases. Instead of the hidden layers from neural networks **CNNs** stack multiple extractions as layers that are all connected, at the end of the chain, is the classification aspect, as shown in Figure 2.1. The architectural layers consist of three main components: Convolutional, Pooling, and Fully connected layers.

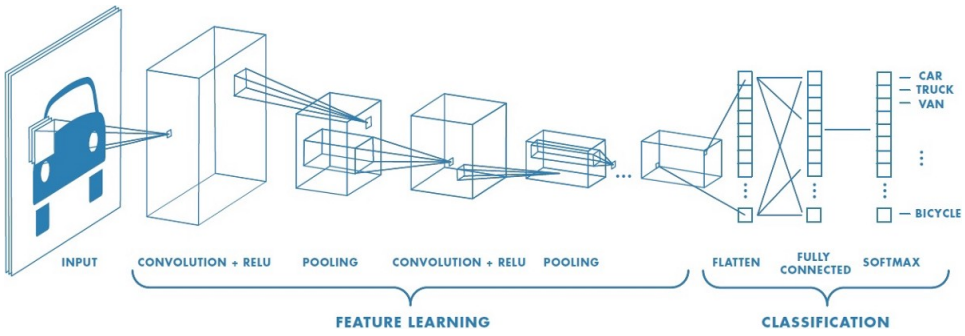


Figure 2.1: CNN architecture illustration from[8]

If we look at this from an image classification aspect, the image is submitted as an input. It gets processed and classified by categories defined in the CNN model specified for the given task. From a computer perspective, the image gets processed as an array of pixels defined by $Height \times Width \times Dimension$.

If the input image could be a standard color picture, the array would be $640 \times 480 \times 3$, where the dimension describes the RGB matrix.The procedure of processing the image is by stepping through the various layers.

Convolutional layers

The first layer in the **CNNs** architecture. It extracts features from the image by iterating over it while applying filters(known as kernels). This is done by splitting the image matrix into smaller blocks while preserving the connection between pixels. Each split is known as a feature map. The mathematical operation is:

- Image matrix $(h \times w \times d)$
- Filter $(f_h \times f_w \times d)$
- When combined outputs $(h - f_h + 1) \times (w - f_w + 1) \times 1$

Each feature map contains a filter that performs operations like edge detection, lines, intensity, sharpen, blur, shape etc. The filter size is normally of small sizes, like 3×3pixels. To keep the pixels "connected", the filter shifts 1 or 2 pixels across the image matrix, causing an overlap; this is called strides.

If a filter exceeds the image matrix, we need to apply padding. This is done either by adding zeros to the picture(zero-padding) or valid padding, which only keeps the valid parts of the image.

A supplementary step for the convolutional operation is the Rectified Linear Unit(**ReLU**) which purpose is to add non-linearity for the **CNN**. The input data would consist of non-negative linear values by adding **ReLU** the output formula is $f(x) = max(0, x)$ which iterates through the matrix and sets all negative values to 0.

Pooling layers

Between each successive convolutional layer there are pooling layers which are set to reduce variance, reduce computational complexity and extract features. This is known as spatial pooling. It retains the important features while reducing dimension of the feature maps. By reducing the number of parameters, the memory usage of the network will decrease which in turn allows for more features to be added. Spatial pooling can be divided into three different types:

- **Max Pooling**
  uses a (2×2)pixel neighbourhood matrix to iterate over the image and extracting the largest value within the matrix.

- **Average Pooling**
  uses a (2×2)pixel neighbourhood matrix to iterate over the image while calculating and keeping the average value of within the matrix

- **Sum Pooling**
  sum of all the elements in the feature map

The most commonly used techniques are max pooling and average pooling combined with a small pixel neighbourhood matrix as it reduces the most amount of data.

Fully connected layers

The last layer in a **CNN** evaluates the results from the convolutional/pooling layers by flattening them from a matrix into vectors and feeds them forward to the next fully connected layer, which applies weights to conduct predictions. The output-layer then sets the final probabilities for each label in order to classify the image. The last step is often combined with softmax, which normalizes the neural network output to fit between 0 and 1 in order to establish the probabilities.

2.1.2 Training

When training a **CNN**, the same principals apply as for other feedforward neural networks, by using the backpropagation algorithm.[9]. It utilizes in two steps, which are executed several times each batch. In the first step, a batch of images is fed through the entire network, where the softmax layer outputs a probability estimate. This step is known as **forward pass**. In the next step, a loss function measures the network's accuracy by calculating the error difference between the set training labels and the softmax estimate labels. The goal is to train the network to obtain a low loss function(low loss = high accuracy).

The loss function results are then examined by an optimizer(*adaptive learning method*)[10] that traverses back through the network and calculates how the network's weights and biases need to be tweaked to reduce loss, which is known as **backward pass**. A complete iteration for the whole batch (all images) with both forward and backward pass is called an epoch.

Training problems

Training a model to produce a classifier that conducts good and correct predictions, we could encounter two scenarios with the given data-set the model is using for training. The first is *underfitting*(high bias, low variance). This problem normally happens when the model cannot create an accurate model, normally because of a too-small or non-linear data-set, causing the model to produce wrong predictions. The other problem is *overfitting*(high bias, low variance). This means that the model trained on a too large data-set or images taken in a static environment, making the model focus on image noise or details not important for the given task. There are several ways to avoid this by using annotated images to increase the data-set, bounding boxes around the focused object, early stopping, which monitors the loss during training and stops training when the loss is increasing. The easiest way to monitor this is to split our data-set into two sets: training and testing(validation). Then we can monitor when the accuracy is peaking on the test set and know when the model is in a good fit state. For an underfitting model, the accuracy would be bad for both training and testing data-set.

For good practical examples and detailed explanation on how to handle over/underfit follow the tutorial from Tensorflow[11]

2.1.3 Transfer learning

Training a model from scratch requires enormous data-sets and is considered a high-cost operation regarding hardware requirements and actual time spent. Instead, we make use of an existing **CNN** pre-trained model that already has trained on a large scale data-set like the famous ImageNet[12] or COCO(Common Objects in Context)[13].

The standard procedure is to initiate the pre-trained model and introduce a new and smaller data-set in order to retrain it for the new task at hand. It utilizes the existing model's features and weights and fine-tunes them to make probability identifications of the new objects based on the labels submitted with the data-set.

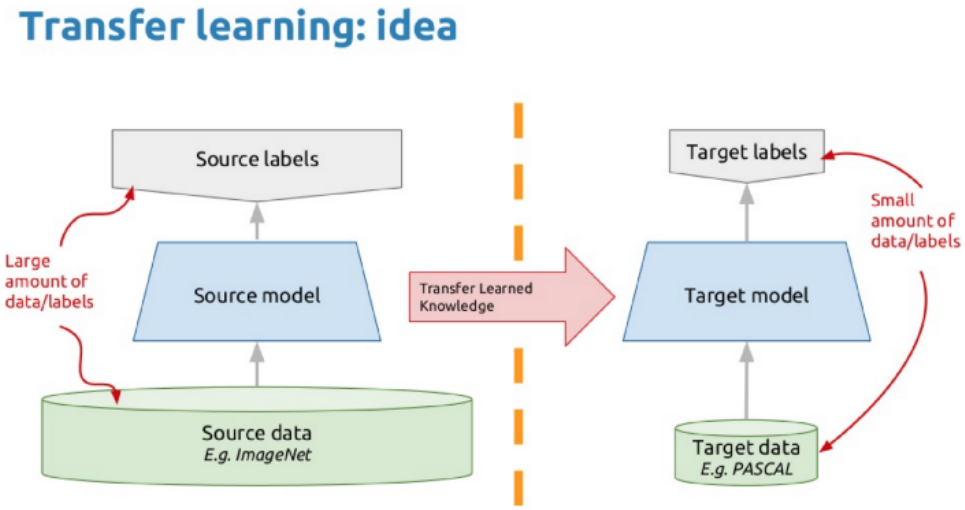


Figure 2.2: Basic flow in transfer learning[14]

3

Related work

Examining relevant literature show that there are many projects and systems in the field of object classification with a focus on animal detection. The COAT camera-trap data-set made it difficult to find similar research as the generated images is from a static environment from within a box. We have focused on the projects that utilize small or embedded DNNs, or have used camera-traps as their data-set source.

Nguyen et. al presented a solution for using DNNs for automated wildlife monitoring[15]. The research is conducted on the single-labeled data.set from Wildlife Spotter project[1], this is a data-set of grate magnitude with 107 000 images split among 15 different animal species,and images without animals. The data-set was split at a 80/20% for training and validation. The size of the data-set enables them to conduct CNN training from scratch, this is implemented in Keras[16] and trained with four Nvidia Titan X GPUs. The goal was to conduct two various types of classification: Recognizing animals vs non-animal images , and identifying the three most common species (bird, rat and bandicoot). For model architecture they choose to implement the lightweight simplified version of AlexNet[17], the medium sized VGG-16[18] and the ResNet-50[19]. Training time varies from 3-5 days for each model. The result is three models that can classify the first task with an accuracy of 92.68% for AlexNet, 95.88% for VGG-16 and 95.65% for ResNet-50. For the second task of classifying the three species the models performed with an accuracy of 87.80%, 88.03% and

1. https://volunteer.ala.org.au/wildlife-spotter

87.97% on the same models. Compared to our model performance regarding animal classification, we can conclude that taking advantage of a pre-trained model is a good approach to solving the problem at hand as we outperformed this approach with a training time of only 3 hours on a lightweight model. The data-set complexity is in our favour with regards to the static environment and gray-scale images.

H. Thom. presented an animal identifying and classification system in 2017[4]. This is based on the bait-camera data-set from COAT[1] with 8000 images of 9 different species. The system unifies three different object detection methods using CNNs. In regards to our thesis, one of the models used is YOLOv2[20], which is the second generation YOLO architecture. The accuracy of the model trained on the bait-cam data-set is measured to 92.4%. In his research, he encountered the same issues with data-set diversity and unbalanced classes as we did in our data-set. He also experienced the necessity of data-set preparation. There was a substantial precision increase by adding bounding boxes and augmentations on the data-set. This research's common goal is very similar to our approach to reducing manual labor when working with extensive image data classification processes.

S. Thommasen took on the same data-set as H Thom. in his master thesis[21]. The approach was to develop a small embedded computer capable of classifying the animals using a small neural network. The Model architecture trained to the task where several variants of small MobileNet[22], with an accuracy of 61%. Through the thesis, he explained the difficulties with data-set preparation and model conversion regarding small embedded edge computers. Based on this, we wanted to approach this task with a state-of-the-art object classifier architecture from 2020, and the results show that edge computing on small embedded devices is possible to solve the problems stated in our thesis.

4

Data-set preparation

When developing an object classification model, data-set preparation is the key to success, and several aspects need to be considered.

We need to decide whether we want to conduct transfer learning on a pre-trained model or if we want to develop a custom built model from scratch. The size of the data-set often defines this, and if any existing models conduct image classification in the area of objects we want to classify.

With a small scale data-set, it would be impossible to train a model from scratch. The aim is to develop a lightweight model to be executed through edge computing, a combination of the YOLOv4[23]model trained on the COCO data set[13] is used.

4.1 COAT data-set

In this thesis, a data-set containing 1324 images with five various labels of animals has been used. The images come from the COAT[1] camera-traps that were deployed in the arctic tundra.

Labels	**Images**
Bird	71
Lemming	376
Shrew	381
Stoat	95
Vole	400
Total:	**1323**

Table 4.1: Data-set image distribution. Shows the image count of each label.

With such a small data-set, we need to conduct some pre-processing on the images to utilize the data-set to the fullest. An example of the images in the data-set is shown in figure 4.1.

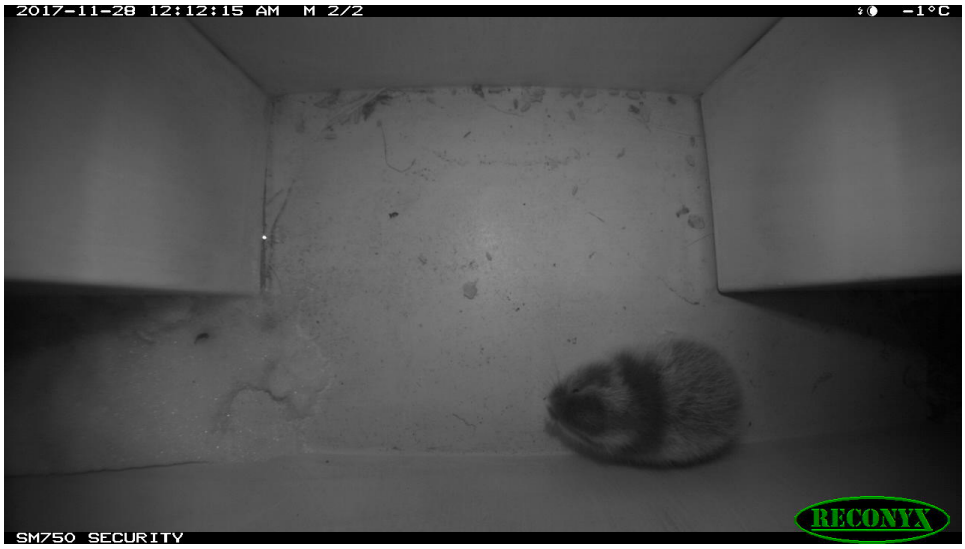


Figure 4.1: Camera-trap image of a lemming

4.1.1 Data-set preparation

There are several tools and techniques that can be applied to maximize the potential of a data-set. Since all images are taken in a static environment, the model training would create an *overfitting* outcome. With little to no variance

in the environmental background, a **CNN** model would not be able to know "what" to look for when training. To encounter this problem, we annotated all the images in the data-set. This is a tedious job that had to be done manually for each image. Annotated images have a correlating XML file connected to it, which contains information such as total image pixel size (1280×720 pixels), then the coordinates for which pixels contain the bounding box, and the label name of what is within the box parameters.

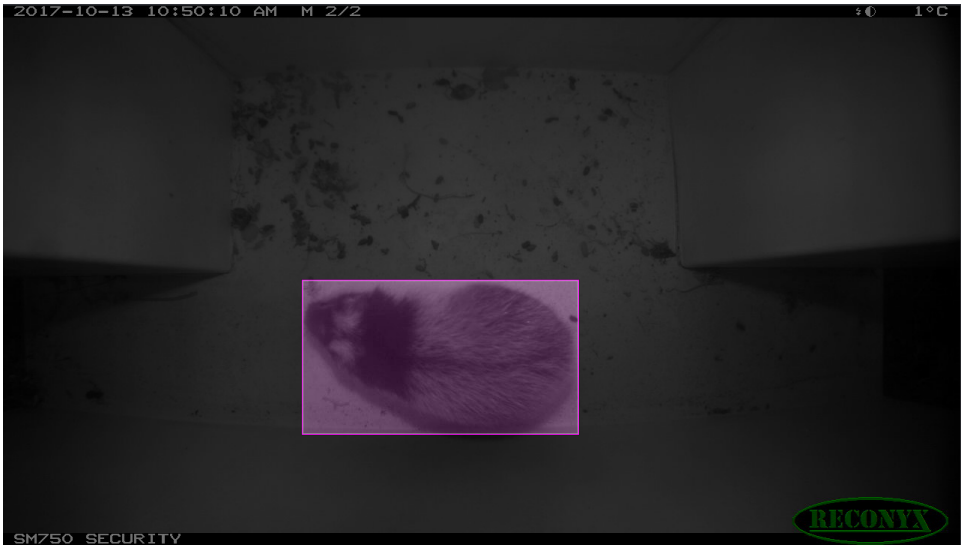


Figure 4.2: Annotated image of a lemming

Then the images with correlating XML files were uploaded to Roboflow[1]A in order to modify and create a data-set that would fit for the YOLOv4 pre-trained model. The images were resized to 416×416pixels, and a flip augmentation step was added to extend the data-set and improve model performance[24]. By flipping the images, both vertical and horizontal, the data-set size expanded from 1323 to 3175 images. The data-set was split into a training, validation, and test set where 70% is used for training. The reason to split the set is to monitor the model loss value in correlations with both training and validation data-set to ensure that we do not over-fit while training.

1. www.roboflow.com

5

Design And Architecture

During the development of the model in this project, individual design choices have been made regarding the final product. In this chapter, the architectural design for an embedded system capable of conducting image classification on a specifically labeled data-set is presented. The trained model is converted into a lightweight model that can be executed on a small and efficient micro-controller to do real-time classification and diagnostics in the Arctic Tundra. As proof of concept, an android powered smartphone is used for testing.

5.1 Workflow Design

The overall workflow design for this system is shown in figure 5.2.

The data-set images are collected from the Distributed Arctic Observatory (DAO)-store server and exported to our local machine, then uploaded to the Roboflow API for prepossessing. We Import the data-set from Roboflow, a pre-trained model, and a transformation tool from Github into Google Colab, a cloud-based virtual machine environment. The implemented solution for training and testing described in chapter 6 and chapter 7 is conducted in this environment. The lightweight model gets stored in Google Drive cloud storage and is downloaded to our local machine. Android Studio is used to develop an application with our lightweight model embedded and deployed on the smartphone edge device.

5.1.1 Preparation

Before we could utilize a GPU enabled computer to conduct transfer learning on a pre-trained object classifying **CNN** model, the data-set needs to be pre-processed as thoroughly explained in Data-set preparation.

Roboflow tool

To process the images regarding bounding box annotations, augmentations, and labeling, Roboflow[25] was chosen as the best tool to handle the task. This "all in one" platform was released in January 2020 and has streamlined the tedious process of data-set management. It supports various image and annotation formats, pre-processing(resize, orient, contrast, grayscale), augmentation(flip, rotate, crop, blur, etc.), evaluation splitting(test, validate, train), and data-set health check in order to make improvements before model training. In appendix A we have included some screenshots from the RoboFlow API.

5.1.2 Training

Before the training phase could be initiated, we need to choose which pre-trained model is best suited to solve the task at hand. This is the most crucial step in this design, which is defined by the results, establish if the model could be utilized for production.

Model selection

One of the most popular computer vision segments is object classification, and the variety of architectures to choose from is plenty. For this specific task, several points had to be taken into consideration and evaluated regarding performance, accuracy, data-set structure, and device deployment.

As we strive to work with the latest and most developed architecture, the choice fell on the cutting-edge fourth-generation You Only Look Once (YOLO)[23] released in April 2020. The performance in combination with the Common Objects in Context (COCO)[13] data-set is in the top tier. The architecture used in this thesis is YOLOv4-tiny, which is a compressed version of YOLOv4. Like the full version, it has Darknet[26] Deep Neural Network (DNN) backbone, it is trained from 29 pre-trained convolutional layers with incredible inference. When it comes to precision, it deliverers roughly 2/3 of the full-scale model when working with the COCO data-set. (see figure 5.1).

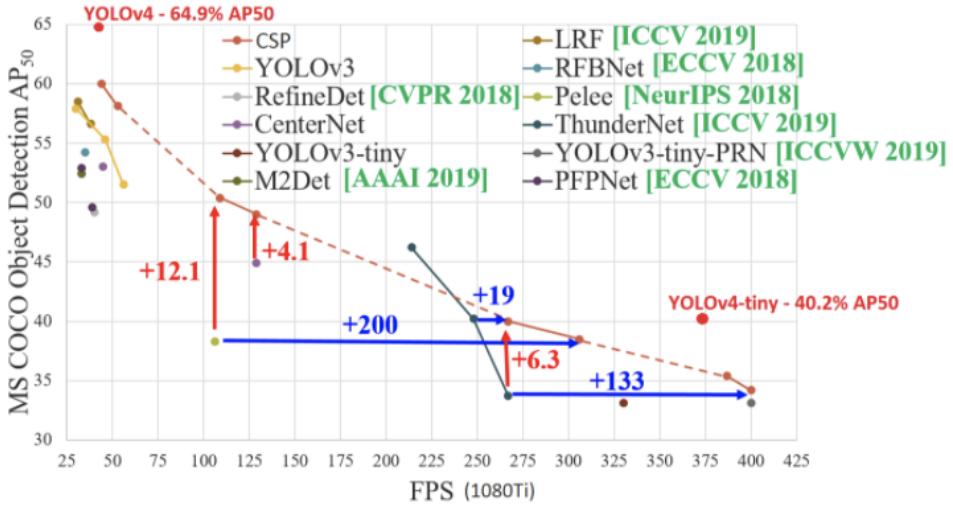


Figure 5.1: Performance comparison of small vs full-scale models in regards to AP illustration from [27]

We are training this on a data-set with only five various labels in a static environment. The accuracy trade-off should not affect the end-result in any particular way with regards to inference and probability.

As stated, we wanted to use Yolov4 since this is one of the newest and well-developed object detectors available. There are several other good candidates, and while researching, we examined both ShuffleNet[28] and the famous and well tested Single Shot Detection (SSD) MobileNet-v2[22] as possible alternatives.

GPU enabled training platform

The platform for training our model is Google Colab[1], which is a free cloud service from Google. This supports Python[2] programming language and is designed for developing deep learning applications that employ Graphical Processing Unit (GPU) or Tensor Processing Unit (TPU) to conduct calculations. The environment is built on a serverless Jupyter[3] notebook principal, supporting both text and code in a document style structure for interactive development.

The training environment is initialized, with all dependencies and modules installed in the cloud service. The data-set is converted to Darknet[26] format and compressed in RoboflowApplication Programming Interface (API). We import and extract it in Google Colab and stored as training, validation, and test subsets. The pre-trained YOLOv4-tiny model is loaded, and we introduce the labels and our custom data-set to conduct transfer learning, as shown in figure 2.2. The final product of the training is a YOLOv4-tiny model designed to conduct object classifications on the five labels in the data-set.

Model conversion

Before we can deploy our model to an edge device, this needs to be converted to TensorFlow lite[4](.tflite) format in order to be initialized on an embedded device. The operation consists of a two-step procedure: First, the Darknet .weights file needs gets converted to TensorFlow .pb file format. Then we can collapse the full-scale TensorFlow model into a lightweight .tflite model.

Device deployment

To execute the converted model on a device, we choose an Android operated smartphone. The open-source framework for an image classification app developed by Viet Hung[29] with Android Studio[5] got used as a template. The custom `.tflite` model was united with the application framework and trans-

1. https://colab.research.google.com/
2. https://www.python.org/
3. https://jupyter.org/
4. https://www.tensorflow.org/lite
5. https://developer.android.com/studio

shipped to the smartphone. The application loads the model and uses the integrated camera on the device as an input source for images classified in regards to the model's labels. The output is a bounding box with a label around the object, the model's probability accuracy, and inference time.

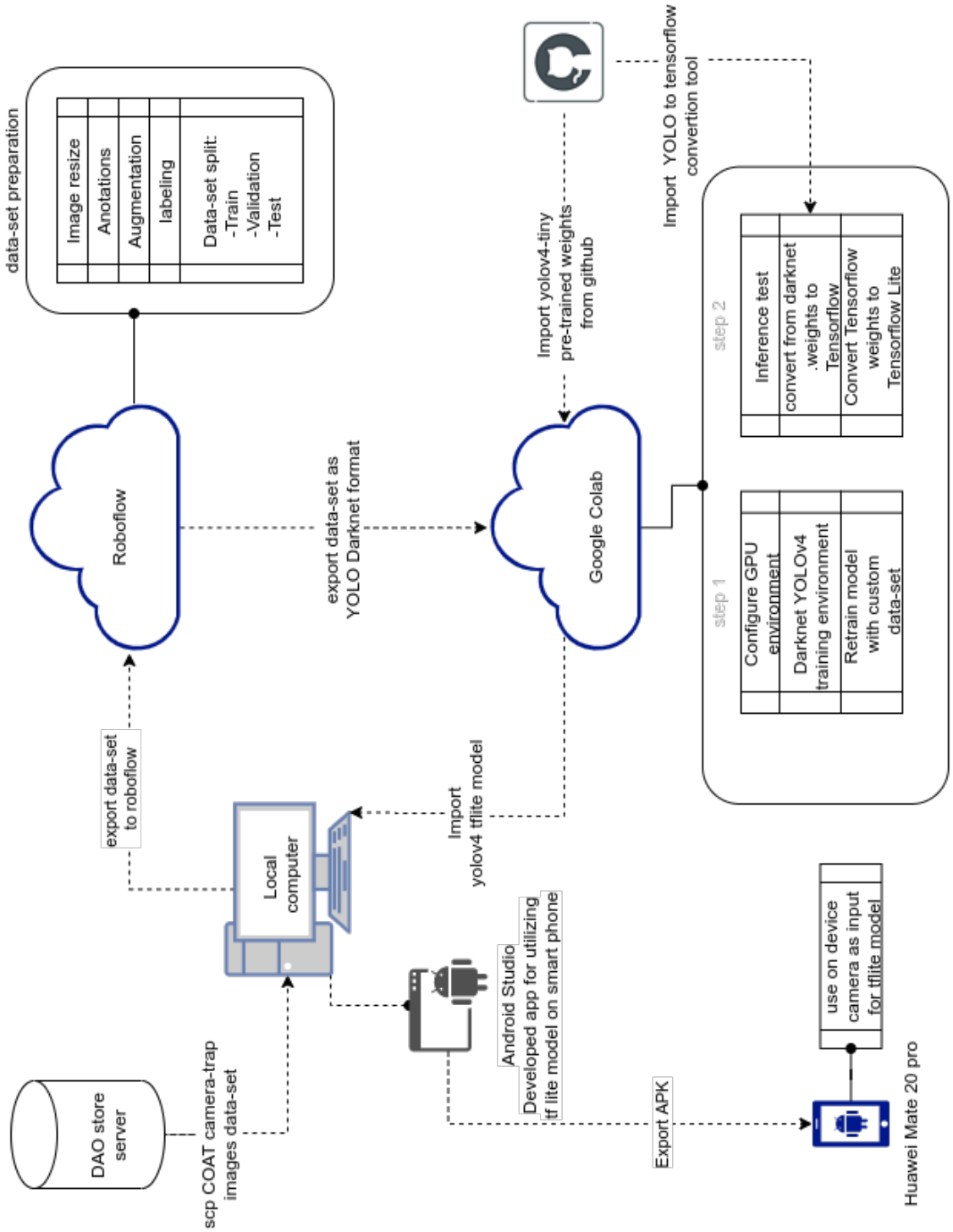


Figure 5.2: Architectural workflow design

6 Implementation

This chapter details the implementation process for a lightweight object classifier capable of accurately performing in-field classification of animals in the Arctic Tundra. The system is based on several open-source frameworks and has been developed in a cloud-based environment. We use Google Colaboratory[30] (henceforth referred to as Colab) to prototype our machine learning model. This platform provides interactive development, GPU, and TPU hardware options in a Jupyter[31] notebook-style environment. This allows us to implement, execute, and test on the same platform. By combining CNN model based on Darknet[26], YOLOv4[23], with our custom data-set4 pre-processed with the Roboflow[25] API we conducted transfer learning. The result is a Yolov4-tiny model capable of classifying the labels desired for this task.

In order to deploy this on an embedded edge computing device (smartphone for testing), we needed to convert the Darknet based Yolo model over to a TensorFlow lite model file, designed to run on such devices. For this procedure, we combined a Python implemented conversion tool[29] that generates a TensorFlow frozen graph model. This allows us to convert into a TensorFlow lite(tflite) model.

The final part of our implementation is an application for Android-based devices. This is implemented in Android Studio and based on the framework by Viet Hung[29]. The application, loaded with our pre-trained lightweight model, reads image data through the device camera module before classifying,

measuring the accuracy and performing inference in real-time.

6.1 Google Colab implementation

We start by configuring CUDA and GPU; then, we clone the Darknet repository from GitHub[1] and install all dependencies (listed in 7). Now we can clone the pre-trained YOLOv4 weights that we will rely on to conduct transfer-learning with our custom data-set.

The RoboFlow APIA offers the prepared data-set through several endpoints, which we download through an integration with Colab. The imported data-set is then divided into subsets for training and validation, and labels are linked to their respective images.

Before we start the training phase, we need to write a configuration file which describes how the training is to be conducted. We define the number of iterations we want to train for, batch size, and how often the weights should be saved to backup. With the configuration in place, we can initialize the training phase. With the Darknet model we are using there is an included script for monitoring the metrics for each iteration. We use this to avoid over-fitting, as described in 2.

After the training is completed, we mount Google Drive through another Colab integration and copy our re-trained weights before we convert them into TensorFlow format.

We initially tried to follow the guide from TensorFlow to utilize the Python API[32] in order to convert our model, but sadly, we were unable to make this work. We found a Python tool developed by Viet Hung[29] that is designed to convert from Darknet `.weights` into a TensorFlow frozen graph model. After this conversion, we could follow the TensorFlow guide to convert from regular TensorFlow into a lite model which is more suitable for our requirements.

6.1.1 Android Application Implementation

To test the converted TensorFlow lite model's performance, we decided to use a smartphone to stand in as a small embedded edge computer.

1. https://github.com/

We used the Android Studio[2] which is the official Integrated Development Environment (IDE) for Android[3] app development. The application is built on the foundation framework from the open-source GitHub[29] repository with a TensorFlow back-end. We utilized this wholesale, with some minor modifications to the visual front-end. The trained model downloaded from Google Drive is embedded within the application. We used the Android Studio virtual smartphone to test our implementation before deploying it to the actual device. The result for edge computing on the Huawei[5] smartphone is presented in figure 6.1 and shows the start screen and the application in use.

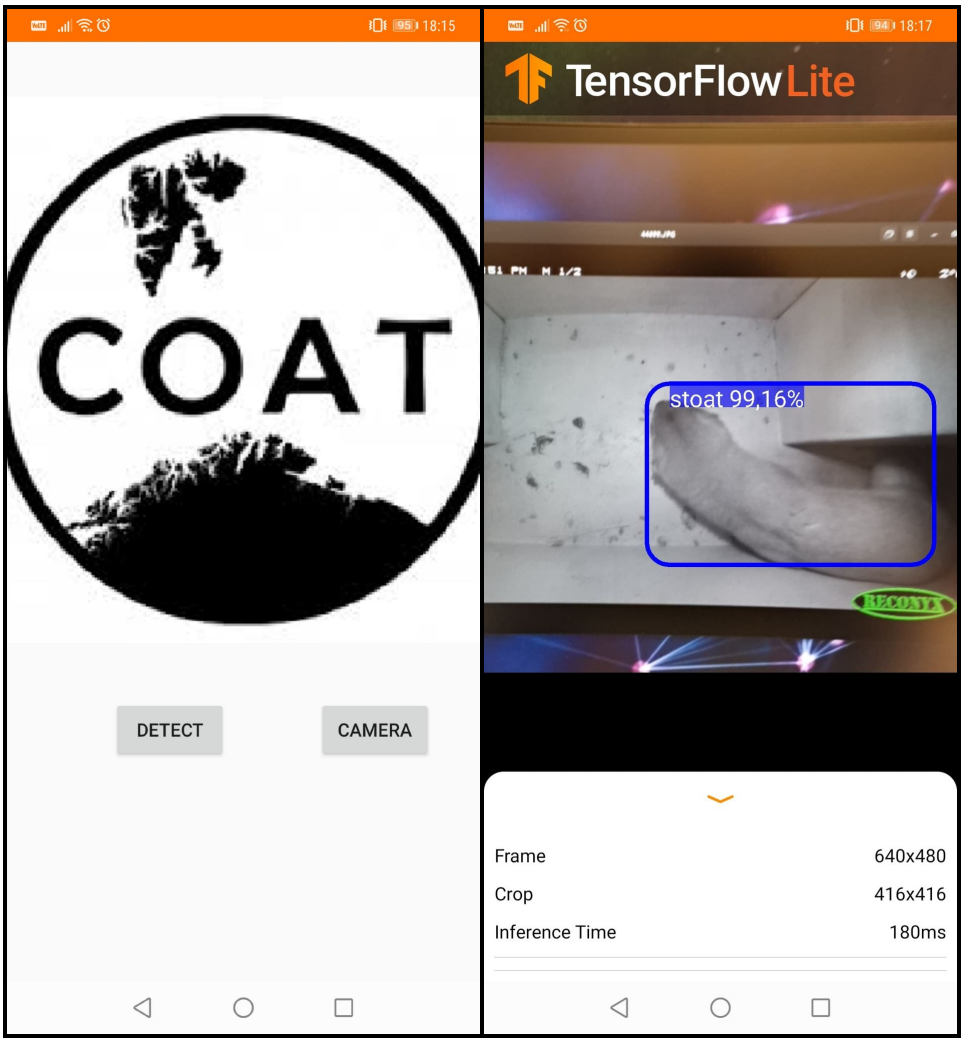


Figure 6.1: On device screenshots of Android application for object classification

2. https://developer.android.com/studio/intro/
3. https://www.android.com/intl/no_no/what-is-android/

7 Evaluation

This chapter explains the experimental setup and detection metrics used to evaluate the object classification system tailored to recognize and classify animals in images from the COAT camera trap setup[3] The evaluations are based on the YOLOv4-tiny model trained on the data-set described in 4, and the TensorFlow lite conversion of this model. T

7.1 Experimental platforms

7.1.1 TensorFlow lite model platform

The experiments for this model were run on a HUAWEI Mate 20 pro smartphone with the following spesifications:

- HiSilicon Kirin 980 octa-core CPU @ (2x2.6 GHz Cortex-A76 & 2x1.92 GHz Cortex-A76 & 4x1.8 GHz Cortex-A55)
- ARM Mali-G76 MP10 GPU @ 720Mhz (integrated into the Central Processing Unit (CPU))
- 6GB RAM @ 2133Mhz
- Operating system: Android v10.0

7.1.2 YOLOv4-tiny model platform

The experiments for this model were run in the cloud based machine learning environment Google Colaboratory. This is a Virtual Machine (VM) with the following specifications:

- Nvidia Tesla T4 16GB GPU @ 1590Mhz (2560 CUDA cores)
- Operating system: Ubuntu v18.04.5 LTS with Python v3.6.9

The testing environment is built with the following dependencies:

- CUDA v10.1.243
- cuDNN v7.6.5
- OpenCV v3.2.0
- Matplotlib v3.2.2
- NumPy v1.18.5
- Tensorflow v2.3.0

7.2 Experimental design

We are talking about two different object classification models in this thesis. The transfer learning training process is conducted for the Yolov4-tiny model since the TensorFlow lite model is a conversion of that model. The training parameters for the YOLOv4-tiny model is stated in the generated configuration(`.cfg`) file.

The code snippet show how we calculate the number of iterations and is essential to the configuration.

```
num_classes = file_len('train/_darknet.labels')
max_batches = num_classes*2000
steps1 = .8 * max_batches
steps2 = .9 * max_batches
steps_str = str(steps1) + ',' + str(steps2)
num_filters = (num_classes + 5) * 3
```

The number of labels in the training data-set is 5, this sets training to be done

with 10000 iterations (training steps). We used default batch size of 64 images. We used the data-set described in 4,and all input images where resized from $1280 \times 1024/720$ pixels to 416×416 pixels to optimize training performance. The Nvidia Tesla T4 GPU conducted all training steps in approximately 3hours.

7.3 Classification metrics

We base our evaluation methods on the most commonly used classification metric measurement techniques used in challenges such as The PASCAL VOC[33], The COCO Object detection challenge[34], and The Open Images Challenge[35]. In object classification, the mean Average Precision (MAP) is the most commonly used metric since this evaluates the overall precision of the model. AP measures how good the model precision is on a specific label(class). With an object classification model, there are almost always multiple classes(labels). For all models where labels are $N > 1$ we want to calculate the mean of the AP for each label, which is based on the precision-recall curve. The various concepts included in the evaluation are elaborated in the subsection

7.3.1 Concepts

Confidence score Is predicted by the classifier and is the probability that there is an object in the anchor box.

Intersection over Union (IOU) is defined by the intersection area divided by the union of a predicted bounding box and a ground-truth box. This is used to determine whether detection is TP or FP. In the example below we use (BB_p) for predicted bounding box, and (BB_{gt} for the ground-truth bounding box. When evaluating our model, we follow the threshold standard from PASCAL VOC[33] of 0.5(50%).

$$IoU = \frac{BB_p \bigcap BB_{gt}}{BB_p \bigcup BB_{gt}}$$

True Positive (TP). For a detection to be TP it needs to satisfy three conditions:

- Confidence score > threshold
- Predicted bounding box has an IoU greater then a threshold with the ground-truth.

- The predicted label(class) matches the label of a ground-truth.

If either of the two last conditions is not fulfilled, we get a **FP**. When the confidence score drops below the threshold, we get a **FN**.

Precision is the baseline that reveals how accurate our model is in a specific class. It is defined by the number of TP divided by the sum of TP and False Positive (FP).

$$Precision = \frac{TP}{TP + FP}$$

Recall is defined by the number of TP divided by the sum of TP and False Negative (FN) (this is known as the ground-truths).

$$Recall = \frac{TP}{TP + FN}$$

Precision and recall are inversely related. When precision increases, then recall falls and vice-versa. It is preferred to get a balance between them. This can be found by creating a precision-recall curve.

Precision-recall curve values are computed for subsets of detection. It starts with the highest detected scores, and then adding the reminding, detection scores are added in decreasing order. By plotting all the detections in a graph creates the precision-recall curve. The precision score is between 0-1 in the vertical line, and the recall score is between 0-1 on the horizontal line.

AP is the task of finding area under the precision-recall curve. Since precision and recall both are within 0-1(0-100%) AP also is within the parameters. The calculations are done by dividing the recall value from 0.0 to 1.0 into 11 points, and the mathematical formula is:

$$AP = \frac{1}{11} \times (AP_r(0) + AP_r(0.1 + ... + AP_r(1))$$

mAP is the AP for all labels(classes) in the model and is calculated:

$$MAP = \frac{\sum_q = 1^Q AveP(q)}{Q}$$

Where Q is the number of total labels, and AveP(q) is the AP for one label.

7.4 Results

In table7.1 the AP for each model is listed alongside the AP for each label in the data-set.

Model	**mAP**	Stoat	Vole	Bird	Shrew	Lemming
YOLOv4-tiny	98.19	100	98.46	93.75	100	98.72
TensorFlow lite	96.07	95.14	98.24	91.54	98.38	97,05

Table 7.1: mean Average Performance for each model and AP for each class

From the table we see that the results are satisfactory for the both models. The testing methods are conducted differently. For the YOLOv4-tiny model we have used the python script from[27] which does all calculations on the validation data-set after training is done. The metric output in Colab is shown in figure7.1.

```
 calculation mAP (mean average precision)...
268
 detections_count = 326, unique_truth_count = 265
class_id = 0, name = bird, ap = 93.75%           (TP = 14, FP = 0)
class_id = 1, name = lemming, ap = 98.72%        (TP = 79, FP = 4)
class_id = 2, name = shrew, ap = 98.46%          (TP = 64, FP = 0)
class_id = 3, name = stoat, ap = 100.00%         (TP = 15, FP = 1)
class_id = 4, name = vole, ap = 100.00%          (TP = 90, FP = 0)

 for conf_thresh = 0.25, precision = 0.98, recall = 0.99, F1-score = 0.98
 for conf_thresh = 0.25, TP = 262, FP = 5, FN = 3, average IoU = 85.50 %

 IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
 mean average precision (mAP@0.50) = 0.981860, or 98.19 %
Total Detection Time: 1 Seconds
```

Figure 7.1: Colab evaluation metrics for YOLOv4-tiny with the validation data-set as input

We initially anticipated the AP for bird and stoat to be at much lower threshold since the data-set both contained under 100 images for each class. By applying the annotations we expanded the size by 3, but the diversity in the images are still in the lower spectrum.

The environment is static on all images, and this is considered a major advantage for the end result, since this minimizes the interference. All images produced by the camera-traps is gray-scale, which makes the training process easier for the model.

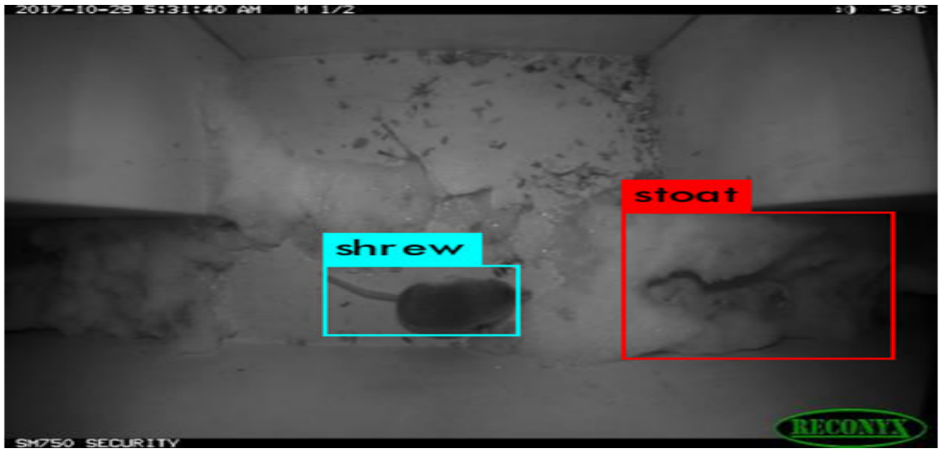
When testing with various images we encountered a problem that needs to be taken into consideration. If the model detects more than one object, the image needs to be manually checked. Due to the fact that the data-set only contains

images with one object in each image we know that the training is conducted on single objects only. Output results from multiple-detection failure shown on figure7.2.

On the Android smartphone the testing was conducted by displaying images from the same data-set on a monitor and using the mobile application deployed on the device to analyze each picture through the on-board camera. Then manually calculating the average of each class. Examples are shown in figure7.3. The inference time on each prediction conducted on the device varies from 100-300milliseconds.

```
 CUDA-version: 10010 (10010), cuDNN: 7.6.5, GPU count: 1
 OpenCV version: 3.2.0
 compute_capability = 750, cudnn_half = 0
net.optimized_memory = 0
mini_batch = 1, batch = 16, time_steps = 1, train = 0
   layer   filters  size/strd(dil)      input                output
   0 conv     32       3 x 3/ 2    416 x 416 x   3 ->  208 x 208 x  32 0.075 BF
   1 conv     64       3 x 3/ 2    208 x 208 x  32 ->  104 x 104 x  64 0.399 BF
   2 conv     64       3 x 3/ 1    104 x 104 x  64 ->  104 x 104 x  64 0.797 BF
   3 route  2                                 1/2 ->  104 x 104 x  32
   4 conv     32       3 x 3/ 1    104 x 104 x  32 ->  104 x 104 x  32 0.199 BF
   5 conv     32       3 x 3/ 1    104 x 104 x  32 ->  104 x 104 x  32 0.199 BF
   6 route  5 4                               ->  104 x 104 x  64
   7 conv     64       1 x 1/ 1    104 x 104 x  64 ->  104 x 104 x  64 0.089 BF
   8 route  2 7                               ->  104 x 104 x 128
   9 max                2x 2/ 2    104 x 104 x 128 ->   52 x  52 x 128 0.001 BF
  10 conv    128       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 128 0.797 BF
  11 route  10                                1/2 ->   52 x  52 x  64
  12 conv     64       3 x 3/ 1     52 x  52 x  64 ->   52 x  52 x  64 0.199 BF
  13 conv     64       3 x 3/ 1     52 x  52 x  64 ->   52 x  52 x  64 0.199 BF
  14 route  13 12                            ->   52 x  52 x 128
  15 conv    128       1 x 1/ 1     52 x  52 x 128 ->   52 x  52 x 128 0.089 BF
  16 route  10 15                            ->   52 x  52 x 256
  17 max               2x 2/ 2     52 x  52 x 256 ->   26 x  26 x 256 0.001 BF
  18 conv    256       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 256 0.797 BF
  19 route  18                                1/2 ->   26 x  26 x 128
  20 conv    128       3 x 3/ 1     26 x  26 x 128 ->   26 x  26 x 128 0.199 BF
  21 conv    128       3 x 3/ 1     26 x  26 x 128 ->   26 x  26 x 128 0.199 BF
  22 route  21 20                            ->   26 x  26 x 256
  23 conv    256       1 x 1/ 1     26 x  26 x 256 ->   26 x  26 x 256 0.089 BF
  24 route  18 23                            ->   26 x  26 x 512
  25 max               2x 2/ 2     26 x  26 x 512 ->   13 x  13 x 512 0.000 BF
  26 conv    512       3 x 3/ 1     13 x  13 x 512 ->   13 x  13 x 512 0.797 BF
  27 conv    256       1 x 1/ 1     13 x  13 x 512 ->   13 x  13 x 256 0.044 BF
  28 conv    512       3 x 3/ 1     13 x  13 x 256 ->   13 x  13 x 512 0.399 BF
  29 conv     30       1 x 1/ 1     13 x  13 x 512 ->   13 x  13 x  30 0.005 BF
  30 yolo
[yolo] params: iou loss: ciou (4), iou_norm: 0.07, cls_norm: 1.00, scale_x_y: 1.05
nms_kind: greedynms (1), beta = 0.600000
  31 route  27                              ->   13 x  13 x 256
  32 conv    128       1 x 1/ 1     13 x  13 x 256 ->   13 x  13 x 128 0.011 BF
  33 upsample                 2x    13 x  13 x 128 ->   26 x  26 x 128
  34 route  33 23                            ->   26 x  26 x 384
  35 conv    256       3 x 3/ 1     26 x  26 x 384 ->   26 x  26 x 256 1.196 BF
  36 conv     30       1 x 1/ 1     26 x  26 x 256 ->   26 x  26 x  30 0.010 BF
  37 yolo
[yolo] params: iou loss: ciou (4), iou_norm: 0.07, cls_norm: 1.00, scale_x_y: 1.05
nms_kind: greedynms (1), beta = 0.600000
Total BFLOPS 6.793
avg_outputs = 300197
 Allocate additional workspace_size = 26.22 MB
Loading weights from backup/custom-yolov4-tiny-detector_best.weights...
 seen 64, trained: 407 K-images (6 Kilo-batches_64)
Done! Loaded 38 layers from weights-file
test/7685_JPG.rf.736850b9e6c0e2979ae106579ba6daa7.jpg: Predicted in 5.386000 milli-seconds.
shrew: 100%
stoat: 89%
```

(a) Detection metrics



(b) Bounding box output image

Figure 7.2: Example on failure when detecting multiple objects from the data-set images

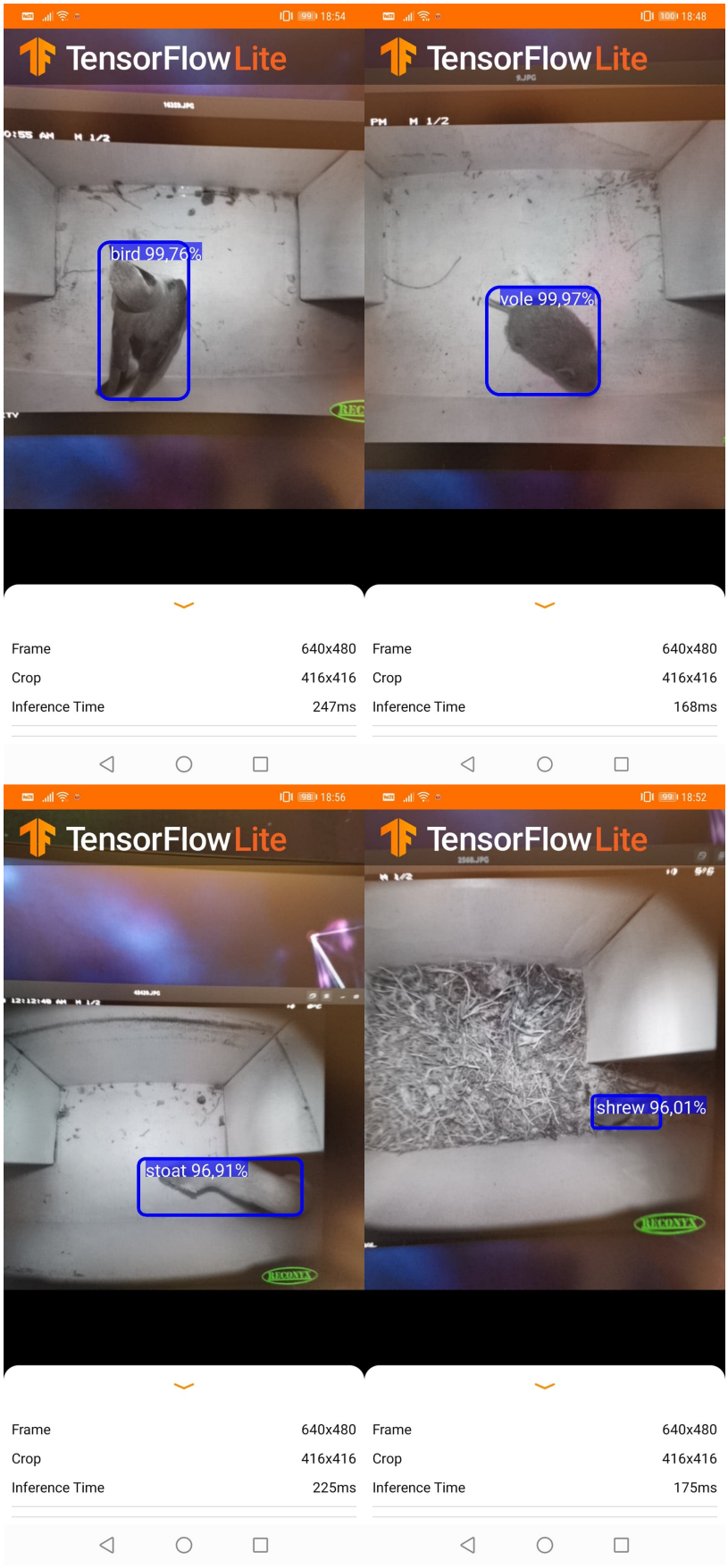


Figure 7.3: screenshot predictions form smartphone APP

8 Discussion

This chapter will discuss the results of our state-of-the-art object classification approach—how we can improve it for future integration in the COAT[3] Arctic Tundra camera traps. We will elaborate on short-comings and what needs to be developed to this "proof of concept" to conduct further testing. This thesis has used several tools and to improve our data-set, conduct transfer-learning of an existing model, and develop an application deployed and tested on a small embedded computer. We will discuss the benefits and lessons learned.

8.1 Results

Evaluation of both models shows that performance in regards to AP is above 90% on all labels of the data-set, as shown in table7.1. As suspected, the Bird class was our lowest performer, with 91.54%. The suspicion is due to the small amount and low diversity of bird images in the data-set. While manually going through the test-set with the application deployed on our smartphone, we discovered that birds' images taken from above are difficult for the model to classify. An example is shown in Figure8.1. This affects the AP for this class and indicates that the model could benefit from a more extensive training set.

As stated in chapter 7, working with images from the COAT camera-traps is very beneficial due to the static environment within the camera-trap boxes. We

suspect that the combination of bounding box annotation, gray-scale images, and the static environment are all factors that contribute to the high precision of the model.

The model is trained on resized images (416×416)pixels, as this is the recommended input image size for the YOLOv4-tiny model. We tested the model on the original size and resized image, which did not affect the model's accuracy. It performed equally on both inputs.

Classification speed is hard to define for the TensorFlow lite model, as it is tested on a device with a contaminated environment, as multiple other applications are running simultaneously in the background. The measured result is 100-300milliseconds on each classification, which indicates that an embedded device such as a modern smartphone, have computational potential to fulfill such tasks with good performance values. It would be of great interest to further the testing by rooting the device and installing a stripped version of Android[1], with only the necessary software installed. We tested the YOLOv4-tiny model in a Google Colab VM with an Nvidia Tesla V4 GPU. The prediction time for each picture varies between 5.2-5.8 milliseconds.

The model size is 23.6MB for each model, which is considered an acceptable size regarding small embedded systems; this enables it to load the whole model in memory.

8.2 Data-set

Our data-set is explained in Chapter4. In this section, we will discuss the short-comings and ways to improve it.

The data-set showed in table4.1 is imbalanced on several classes and considered a small size data-set with only 1323 images. We used RoboFlow API[2] improve our data-set by applying several features:

- Flip augmentation, which flips each image both horizontally and vertically. Which tripled the volume of images and gave a bit of diversity for the set.

- Bounding Box annotation helps the model to know where in the image the object in focus is and what type of label the object holds.

1. https://www.android.com/
2. www.roboflow.com

- Resized all images to 416x416pixels since this is the preferred input for our model
- Split the set into subsets: Training, Validation, and Test
- Converted the data-set to YOLO Darknet format in order to export it.

In retrospect, there are several short-comings with our model due to data-set-related implications. First and foremost is the set's low diversity, which gives us edge case errors like the bird example described in the previous section and shown in figure 8.1. The set does not contain any images with multiple objects, making it impossible to test how the model performs when presented with multiple object images. Because this does not exist in our set, we know this is an error, as shown in figure 7.2. If this model is to be used for evaluating large data-sets from the COAT camera-traps, the solution is to flag all images with multiple detections and manually evaluate them.

Improvement of the data-set would be to expand it by gathering more photos from the camera-traps with greater diversity and a more balanced distribution among classes. This requires more bounding-box annotations, which is tedious work since it has to be done manually. There are several useful tools for this, like the Computer Vision Annotation Tool (CVAT)[36], but it is still a time-consuming job.

8.3 Training platform

For this thesis, we choose to use a VM in the Google cloud platform, as described in 6. The choice was made from experience from previous work with DNN and Artificial Intelligence (AI) in the course INF-3910-6[37] and also described as a problem in previous master thesis in the same field of research[4][21]. When implementing DNN based system, dependencies are a significant issue. Many of the tools needed to conduct the desired work are too version-specific, and it requires that multiple systems interact with each other to execute the task at hand. The upside of implementing such systems in a VM environment makes it easier to start with "clean sheets" if we get dependency issues, and we can try various versions until we find the combination that functions as intended, without the tedious installation and removal of packets. We believe this will be a prevalent tool when working with machine learning and artificial intelligence. The design of a Jupyter notebook gives us the possibility to comment on the code snippets in a markdown environment; this makes the implementation documentation easy to follow for future users.

8.4 Edge computing

As an end product in this project, we developed an Android Application that is deployed on a Huawei[38] smartphone and is used to evaluate the object classification model. The application is built in Android Studio[3] and based on the framework developed by[29].

The result is as shown in chapter6. All experiments on our test data-set are manually done with the device. The model executes with satisfactory performance. The application's improvements would be a back-end database solution to store the detection metrics and generate statistics about the detection. The current model can also conduct detections at a high frequency, so it would be of great interest to deploy it in an environment with rodents and do real-time video classification. This could also incite how the model handles multiple objects in the same visual frame. With the current setup in the COAT camera-traps, the cameras take two pictures when the Passive IR-sensor on the camera is triggered, Then it pauses for 30 seconds to conserve battery. The leap between images is such a rate that it would not create a useful film snippet if we were to merge them into a video.

3. https://developer.android.com/studio

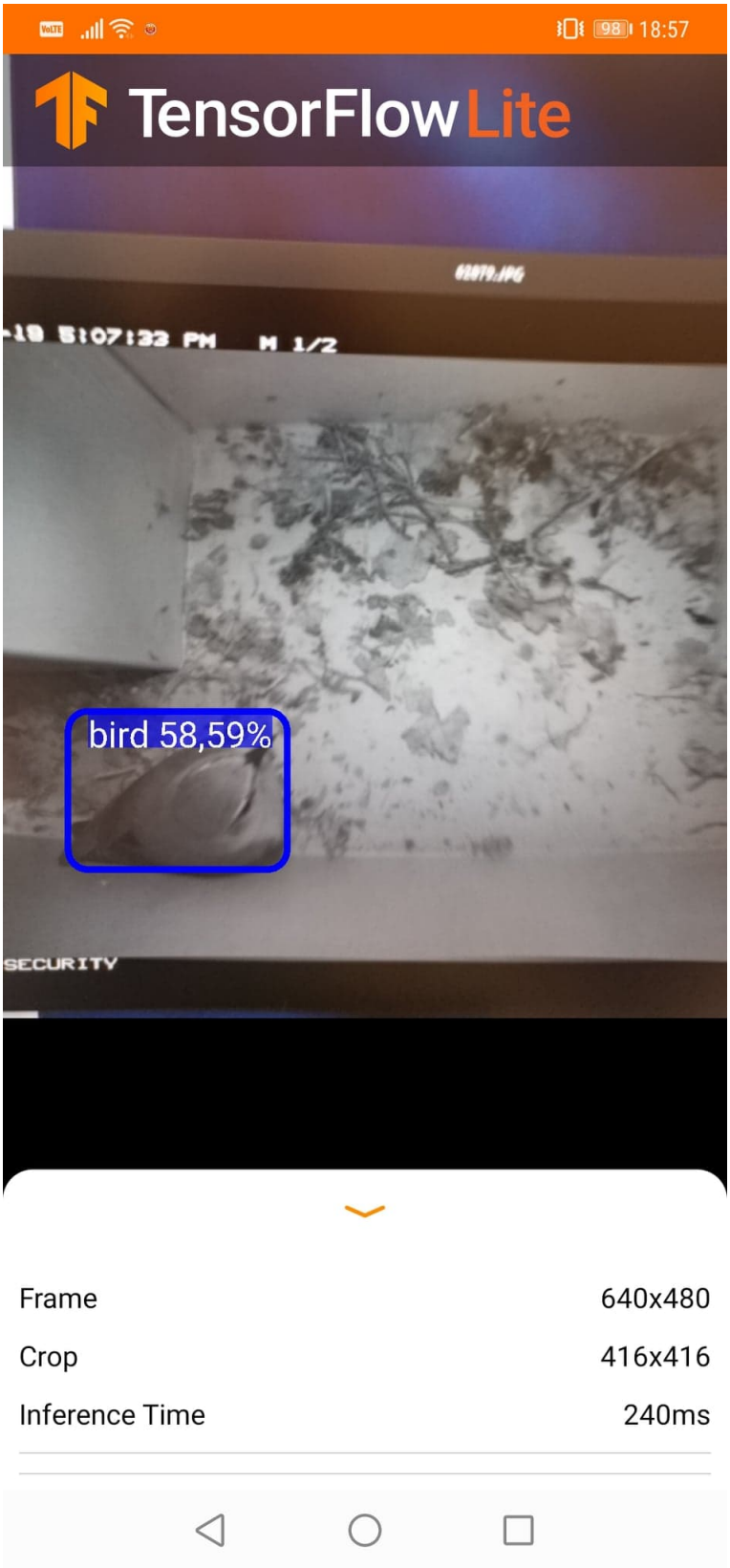


Figure 8.1: screenshot predictions of bird from smartphone APP

9 Conclusion

In this thesis, we have conducted transfer-learning on a pre-trained model by utilizing their weights in order for it to classify images of rodents from the COAT camera-trap data-set. We implemented an Android application that was deployed on a small embedded computer(smartphone) in order to conduct model testing. We have given a detailed description of how we prepared the data-set, trained the model used for classification, and the concept of CNN, which is fundamental for our model. The tools used to conduct this work are state-of-the-art platforms regarding DNN and object classification.

Our experiments showed that a small CNN model could classify the desired animals with a satisfactory accuracy even on small edge devices, with an mAP score of 96.07% for the TensorFlow lite model on the Android device and 98.19% for the YOLOv4-tiny model with a full scale GPU virtual machine. These metrics enable the model to be used as a tool for the COAT research team when classifying data-collections from the camera-traps, or used as a field testing device.

9.1 Future Work

Several improvements can be made to our model and application. We expect that the edge cases mention in chapter8 could be eliminated by expanding the size, diversity, and balance of the data-set. This could be done by annotating more images and single out each class to cope with the imbalance.

For the application, it is desired to implement a back-end functionality, handle classification of large scale image sets, and store each image's predictions on the device. This could automate the manual classification that is done by researchers on the COAT team today and enable the device to conduct field testing or even deployment at the edge.

We would also like to root an Android device and customize the software installation regarding the limitations of other software running in the background. This should generate a severe performance enhancement and measure power consumption while conducting classification on the device. The remote locations and no connection to a sustainable power source makes power consumption a crucial point in edge computing.

Bibliography

[1] "Climate-ecological observatory for arctic tundra (coat)." https://www.coat.no/en/.

[2] "Fram center homepage." https://framsenteret.no/english/.

[3] Soininen, Eeva M, Jensvoll, Ingrid, Killengreen, Siw T, & Ims, Rolf A. (2015). Under the snow: A new camera trap opens the white box of subnivean ecology. *Remote Sensing in Ecology and Conservation., 1*(1), 29-38.

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

[5] "Huawei p20's ai camera: Let artificial intelligence do the heavy lifting." https://consumer.huawei.com/en/press/media-coverage/2018/huawei-p20-ai-camera-let-artificial-intelligence-do-the-heavy-lifting/#:~:text=As%20one%20of%20the%20highly,P20%20can%20master%20photography%20intelligently. Accessed on 2020-07-07.

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *PROCEEDINGS OF THE IEEE*, pp. 2278–2324, 1998.

[7] 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.

[8] "Understanding of convolutional neural network (cnn) — deep learning." https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148. Accessed on 2020-08-20.

[9] K. P. Murphy, "Machine learning: A probabilistic perspective," pp. 569–572, The MIT Press, 2012.

[10] "Cs231 stanford vl - per-parameter adaptive learning rates(adagrad, rmsprop)." https://cs231n.github.io/neural-networks-3/#ada. Accessed on 2020-10-03.

[11] "Ml basics with keras - overfit and underfit." https://www.tensorflow.org/tutorials/keras/overfit_and_underfit. Accessed on 2020-10-20.

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

[13] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft coco: Common objects in context," 2014. cite arxiv:1405.0312Comment: 1) updated annotation pipeline description and figures; 2) added new section describing datasets splits; 3) updated author list.

[14] "Transfer learning (d2l4 insight@dcu machine learning workshop 2017)." https://www.slideshare.net/xavigiro/transfer-learning-d2l4-insightdcu-machine-learning-workshop-2017. Accessed on 2020-09-20.

[15] H. Nguyen, S. J. Maclagan, T. D. Nguyen, T. Nguyen, P. Flemons, K. Andrews, E. G. Ritchie, and D. Phung, "Animal recognition and identification with deep convolutional neural networks for automated wildlife monitoring," in *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 40–49, 2017.

[16] "Keras is a deep learning api written in python, running on top of the machine learning platform tensorflow.." https://keras.io/api/. Accessed on 2020-11-1.

[17] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, June 2015.

[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[19] "Resnet-50 convolutional neural network." https://www.mathworks.com/

help/deeplearning/ref/resnet50.html. Accessed on 2020-11-1.

[20] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," 2016.

[21] S. Thomassen. "Embedded Analytics of Animal Images," *Masters Thesis, Dec 2017*.

[22] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018.

[23] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020.

[24] "How flip augmentation improves model performance." https://blog.roboflow.com/how-flip-augmentation-improves-model-performance/. Accessed on 2020-06-15.

[25] "Overview." https://docs.roboflow.com/. Accessed on 2020-06-15.

[26] "Overview." https://github.com/roboflow-ai/darknetroboflow. Accessed on 2020-10-20.

[27] "Alexeyab github issue #6067 jun 25." https://github.com/AlexeyAB/darknet/issues/6067. Accessed on 2020-10-27.

[28] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," *CoRR*, vol. abs/1707.01083, 2017.

[29] "Tflite application framework." https://github.com/hunglc007/tensorflow-yolov4-tflite. Accessed on 2020-10-28.

[30] E. Bisong, *Google Colaboratory*, pp. 59–64. Berkeley, CA: Apress, 2019.

[31] "The jupyter notebook." https://jupyter-notebook.readthedocs.io/en/stable/notebook.html. Accessed on 2020-11-3.

[32] "Tensorflow python api." https://www.tensorflow.org/lite/convert#python_api_. Accessed on 2020-09-8.

[33] "The pascal visual object classes challenge 2012." http://host.robots.ox.ac.uk/pascal/VOC/voc2012/htmldoc/devkit_doc.html. Accessed on 2020-11-3.

[34] "Detection evaluation." https://cocodataset.org/#detection-eval. Accessed on 2020-11-3.

[35] "Open images challenge 2018 - object detection track - evaluation metric." https://storage.googleapis.com/openimages/web/object_detection_metric.html. Accessed on 2020-11-3.

[36] "Computer vision annotation tool (cvat)." https://github.com/openvinotoolkit/cvat. Accessed on 2020-06-3.

[37] "Inf-3910-6 computer science seminar: Introduction to artificial intelligence and applied methods." https://sa.uit.no/utdanning/emner/emne?p_document_id=605434&ar=2019&semester=V. Accessed on 2020-11-6.

[38] "Huawei mate 20 pro." https://consumer.huawei.com/no/support/phones/mate20-pro/. Accessed on 2020-11-8.

A

Roboflow API screenshots

The screenshots in this appendix are referred to throughout this thesis to show how the Roboflow API dashboard and usage are visualized.

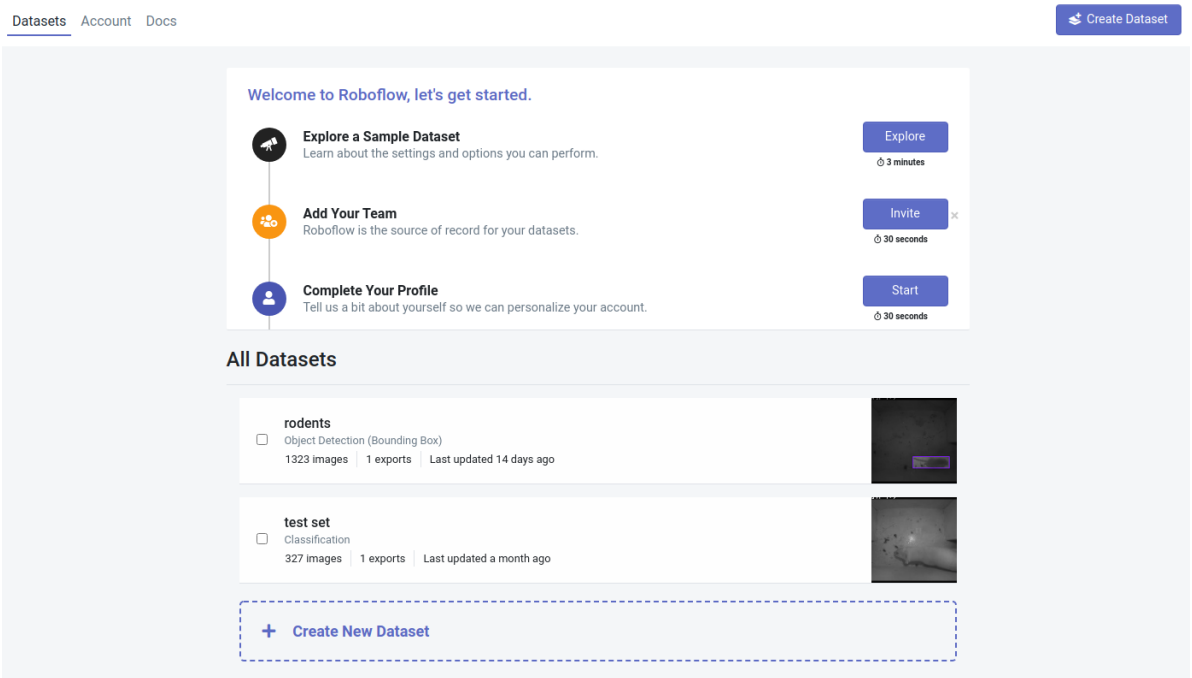


Figure 1: Roboflow Dashboard

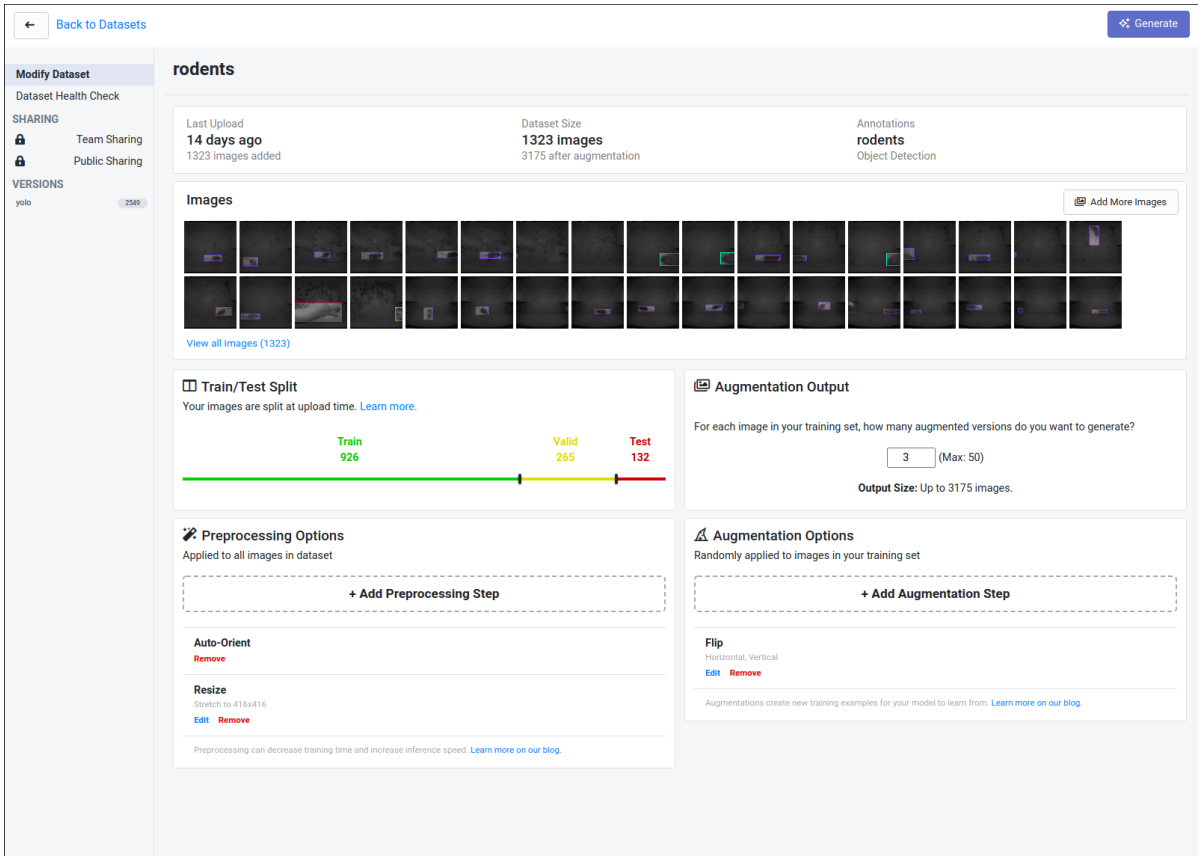


Figure 2: Dataset dashboard

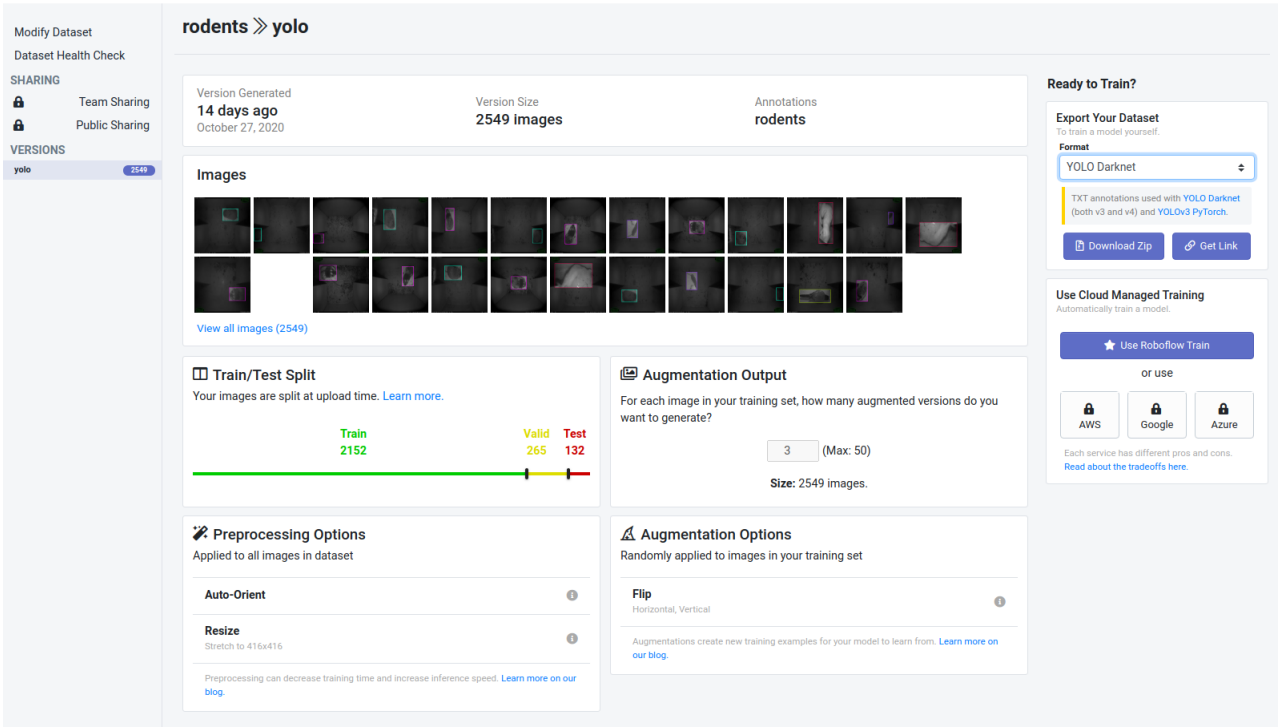


Figure 3: Roboflow data-set version dashboard

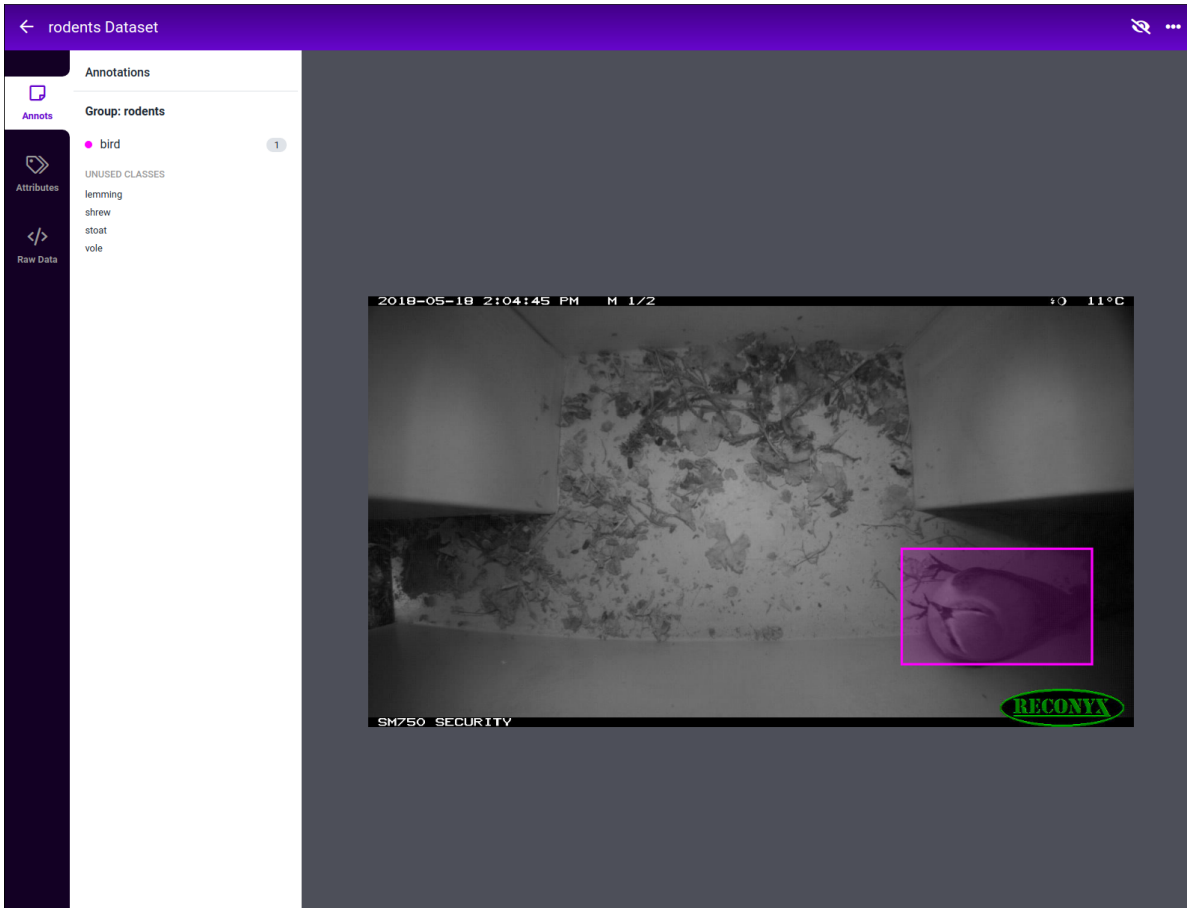


Figure 4: Single image from data-set