UiT THE ARCTIC UNIVERSITY OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Unified Detection System for Automatic, Real-Time, Accurate Animal Detection in Camera Trap Images from the Arctic Tundra

—

Håvard Thom

INF-3981 Master's Thesis in Computer Science . . . June 2017

Abstract

A more efficient and effective approach for detecting animal species in digital images is required. Every winter, the Climate-ecological Observatory for Arctic Tundra (COAT) project deploys several dozen camera traps in eastern Finnmark, Norway. These cameras capture large volumes of images that are used to study and document the impact of climate changes on animal populations. Currently, the images are examined and annotated manually by ecologists, hired technicians, or crowdsourced teams of volunteers. This process is expensive, time-consuming and error-prone, acting as a bottleneck that hinders development in the COAT project.

This thesis describes and implements a unified detection system that can automatically localize and identify animal species in digital images from camera traps in the Arctic tundra. The system unifies three state-of-the-art object detection methods that use deep Convolutional Neural Networks (CNNs), called Faster Region-based CNN, Single Shot Multibox Detector and You Only Look Once v2. With each object detection method, the system can train CNN models, evaluate their detection accuracy, and subsequently use them to detect objects in images.

Using data provided by COAT, we create an object detection dataset with 8000 images containing over 12000 animals of nine different species. We evaluate the performance of the system experimentally, by comparing the detection accuracy and computational complexity of each object detection method. By experimenting in an iterative fashion, we derive and apply several training methods to improve animal detection in camera trap images. These training methods include custom anchor boxes, image preprocessing and Online Hard Example Mining.

Results show that we can automatically detect animals in the Arctic tundra with 94.1% accuracy at 21 frames per second, exceeding the performance of related work. Moreover, we show that the training methods are successful, improving animal detection accuracy by 6.8%.

Acknowledgements

First and foremost, I would like to thank my head-advisor Associate Professor John Markus Bjørndalen for providing guidance, support, and feedback whenever I needed it throughout this thesis. Appreciation is also extended to my co-advisors, Professor Otto Anshus and Professor Alexander Horsch for sharing their knowledge and constructive feedback.

Furthermore, I want to thank the people involved in the COAT project for their help and for the opportunity to work on an interesting project.

I want to express my sincerest gratitude to my fellow students and friends, Frode Opdahl, Johan Ravn, Preben Bruvold Johansen, Kasper Utne, Simen Bakke, Nicolai Bakkeli and Tim Teige. Thank you for all your help and for five great years both inside and outside of the university. You will be missed!

Finally, I want to thank my family and my girlfriend for always being encouraging and supportive of me, with special thanks to my grandparents Arne and Elsa.

Contents

List of Figures

List of Tables

List of Code Listings

List of Abbreviations

AP	Average Precision
CNN	Convolutional Neural Network
COAT	Climate-ecological Observatory for Arctic Tundra
COCO	Common Objects in Context
CPU	Central Processing Unit
CSV	Comma-separated Values
DSSD	Deconvolutional Single Shot Detector
EXIF	Exchangeable Image File Format
FCN	Fully Convolutional Network
FN	False Negative
FP	False Positive
FPS	frames per second
GPU	Graphics Processing Unit
ILSVRC	ImageNet Large Scale Visual Recognition Competition
IOU	Intersection over Union
LMDB	Lightning Memory-Mapped Database
MAP	Mean Average Precision

NMS	Non-Maximum Suppression
OHEM	Online Hard Example Mining
R-CNN	Region-based Convolutional Neural Network
R-FCN	Region-based Fully Convolutional Network
RAM	Random Access Memory
ROI	Region of Interest
RPN	Region Proposal Network
SGD	Stochastic Gradient Descent
SS	Selective Search
SSD	Single Shot MultiBox Detector
SVM	Support Vector Machine
TP	True Positive
VOC	Visual Object Classes
XML	Extensible Markup Language
YOLO	You Only Look Once

1 Introduction

With the climate changes occurring in the world today, it is important to study and document the impact it has on animals and their environments. The Arctic tundra in the far northern hemisphere is one of the ecosystems that are most affected by these changes. Melting of the tundra's permafrost could radically change the landscapes and give rise to new ecosystems with unknown properties [1].

As a response to these realizations, five Fram Centre[1] institutions developed the Climate-ecological Observatory for Arctic Tundra (COAT) project. COAT is a long-term research project with the goal of creating robust observation systems which enable documentation and understanding of climate impacts on arctic tundra ecosystems. In autumn 2015 COAT was granted substantial funding, allowing them to establish a research infrastructure during 2016-2020 [1]. Part of this infrastructure includes the creation of a real-time animal detection system, which is presented in this thesis.

To monitor biodiversity in the Arctic tundra, COAT uses the well-known method of camera traps. This method has revolutionized wildlife ecology over the last two decades and there are currently tens of thousands of camera traps deployed around the planet [2][3]. Camera traps are considered far more cost-effective than direct observations or animal tagging, which are generally extremely labor intensive and invasive. The remotely activated cameras are equipped

1. http://www.framsenteret.no/english

with motion sensors and infrared flash, which enables them to capture images of animals in a non-invasive manner. These images can then be used to record the presence of animals at a site or in some cases suggest the absence of an animal, which could indicate the arrival of a predatorial species [4].

Every year, COAT deploys several dozen camera traps in eastern Finnmark, Norway for approximately one month during the winter. The main purpose of these cameras is to study scavenger populations, with a particular focus on the arctic fox which is critically endangered in Norway. Warmer winters are expected to negatively impact arctic fox population through decreased availability of lemming prey and increased abundance of generalist predators, such as the red fox [5]. The camera traps are programmed to take a time-lapse photo every fifth minute during day and night which accumulates to over 300 000 images per year [6]. Collecting such high volumes of images give rise to Big Data challenges in the ecology field, where usual data tools and practices might not suffice.

Currently, the images are manually examined and annotated, which is an extremely tedious approach that requires months of human labor and resources. This expensive and time-consuming task is often performed by ecologists, hired technicians, or crowdsourced teams of volunteers [7][8]. There is no doubt that this workforce could be more useful elsewhere.

The quality of manual annotations also has to be considered, as several psychological factors affect human performance when sorting objects in visual tasks. These include short-term memory which has a limit of five to nine objects and recency effects where new annotations are biased toward the most recently used labels [9]. Highly repetitive tasks are additionally known to increase fatigue and boredom, causing more annotation errors [10]. With the large amounts of image data piling up from camera traps, this slow and error-prone manual annotation is a bottleneck that hinders development in the ecology field. The advantages of camera traps are clearly not being fully exploited and the demand for automated tools to address these issues are present in both the COAT project and the ecology research community in general.

Previous work presents a system for automatic identification of small mammals in COAT camera trap images with near-human performance [11]. The system is a clear improvement over manual identification and a step in the right direction, but it still has flaws. It does not take into account the possibility of multiple animal species in one image, being unable to individually localize and identify each animal. Naturally, this is an important requirement if the system is to be reliable for animal population studies. It is particularly important when dealing with images of scavengers such as crows and ravens since they often tend to travel in pairs or flocks [12].

This thesis presents a unified detection system as the next step, with these challenges in mind. The system unifies three state-of-the-art object detection methods and is used to automatically localize and identify animals in camera trap images from the Arctic tundra.

1.1 Problem Definition

In this thesis, we consider the problem of detecting animal species in digital images from camera traps in the Arctic tundra. We state that it is possible to create an *automatic*, *real-time*, *accurate* animal detection system using cutting-edge object detection technology.

The system should be

- *automatic* by detecting animals in images with minimal human intervention.
- *real-time* by performing animal detection in images at the same, or a faster rate, than the camera traps captures and supplies images. We measure detection speed with frames per second (FPS).
- *accurate* by correctly localizing and identifying animals in images. We measure accuracy with Mean Average Precision (MAP) which provides a single-figure measure of detection quality. A detailed description of the metric is given in Section 7.3.

To test our statement we present the design and implementation of a unified detection system that detects scavengers in camera trap images from the Arctic tundra. We study and describe three state-of-the-art object detection methods, which are all based on deep Convolutional Neural Networks (CNNs) and unified through our system. The system can train and evaluate CNN models with each object detection method that, in turn, can be used to perform detection on images. We give a detailed description of dataset preparation for object detection and training methods to improve detection of wild animals in the Arctic tundra. In our evaluation, we compare the detection accuracy and speed of each object detection method on our dataset and analyze the effects of our training methods. Finally, we discuss the work that has been done in this thesis and suggest future work for our unified detection system.

1.2 Contributions

This thesis makes the following contributions:

- An introduction to CNNs and a description of three state-of-the-art methods used for object detection in digital images.
- A detailed description of dataset preparation and training methods for animal detection on real world data gathered from camera traps in the Arctic tundra.
- An implementation and description of a system that unifies three state-of-the-art object detection methods.
- A working system for automatic, real-time, accurate animal detection in camera trap images from the Arctic tundra.
- An evaluation of the system comparing the quality of detections and computational complexity of three different object detection methods on our dataset.

1.3 Outline

The thesis is structured into nine chapters including the introduction.

Chapter 2 describes cutting-edge research that has been done in the field of object detection over the past few years. It gives an introduction to CNNs and a description of three state-of-the-art detection methods that are used in our unified detection system.

Chapter 3 presents related work in the field of animal detection, comparing it to the work done in this thesis.

Chapter 4 details dataset preparation and characteristics, together with training methods used for animal detection in the Arctic tundra. It covers challenges and decisions made in bounding box annotation, dataset split, and data formats. Then moves on to describe techniques used in training, including custom anchor boxes, image preprocessing and Online Hard Example Mining (OHEM).

Chapter 5 gives a description and overview of system design and the CNN architectures used by each object detection method in our system.

Chapter 6 describes the implementation and dependencies of our unified detection system, including modifications and adaptions made to open source frameworks that are used.

Chapter 7 evaluates the system by comparing the quality of detections and computational complexity of three different object detection methods on our dataset, and show the effects of our training methods. It includes a description of our experimental setup, detection metrics used and results.

Chapter 8 discusses the process of deriving our training methods and possible deployment of our unified detection system for animal detection in the Arctic tundra.

Chapter 9 concludes the thesis and suggests future work to make improvements in our unified detection system.

2 Object Detection

Object detection is the task of localizing and identifying different objects in digital images or video. It is required in many computer systems and applications, and has become a fundamental technology in computer science. People use object detection every day through technologies such as smart phones [13], industrial robotics [14], and self-driving cars [15].

The research in the field of object detection has made great progress over the past few years, due to the use of Convolutional Neural Networks (CNNs) [16][17][18]. Access to large public datasets from object detection benchmarks such as The PASCAL Visual Object Classes (VOC) Challenge [19] and The Microsoft Common Objects in Context (COCO) Challenge [20] has also been a key factor in its development. This chapter will give an introduction to CNNs and describe three cutting-edge object detection methods that are unified through the system presented in this thesis. The methods are all based on CNNs and can be divided into two major categories:

- Region-based Convolutional Neural Networks
- Single Shot Detectors

2.1 Convolutional Neural Networks

In 2012, the ImageNet Large Scale Visual Recognition Competition (ILSVRC) was won by a huge margin, dropping the image classification error record from 26% to 15%. The winning entry presented AlexNet, a “large, deep convolutional neural network” which revolutionized the field of computer vision [21]. Since then, CNNs has been regarded as state of the art in machine learning and are used in companies such as Facebook and Google [22][23]. A CNN can be seen as a network of learning units that can be trained for a specific task, such as image classification. The network use training data to learn (e.g. image of an eagle) and can subsequently output class predictions (e.g. the label “Eagle”) on new data, forming the basis of automated recognition.

2.1.1 Architecture

CNNs stack multiple layers of feature extractors in a connected structure with a classification layer at the end. These layers form a complete deep CNN architecture for image classification. Figure 2.1 show the three main types of layers in CNN architectures: Convolutional layer, Pooling layer and Fully Connected layer.

- **Convolutional layers** primary function is extracting features from the image. They convolve the input image by sliding over it with a set of filters (also called kernels or feature detectors), each producing a feature map of the image which contains key features like edges, lines, shape, intensity etc. The pixel area or window size of the filters are usually set to a small number like 3×3 pixels, while the amount by which the filter shifts or slides across the image, also known as stride, is set to one or two pixels. Increasing the number of filters produce more image features, leading to a network that is better at recognizing patterns in unseen images, but has the downside of higher computational complexity in terms of memory usage.

- **Pooling layers** reduce the spatial dimensions and retain the most distinct features in the feature maps with a downsampling technique. This reduces the number of parameters, which in turn reduces the memory usage of the network, allowing more filters to be added. Additionally, it makes the convolution process invariant to translation, rotation, and shifting. Max-pooling and Average-pooling are commonly used. Max-pooling iterates over the image with a small pixel neighborhood (usually 2×2) and keeps the maximum value within the window. Average-pooling calculates and keeps the average value of the pixels in the window.

- **Fully Connected layers** do the high level reasoning of the features that are output from the previous convolution and pooling layers. It produces probabilities for all classes, such as “Eagle“, based on how the high-level features correlate to each particular class. The last fully connected layer is thus known as the output layer which gives the final class probabilities.

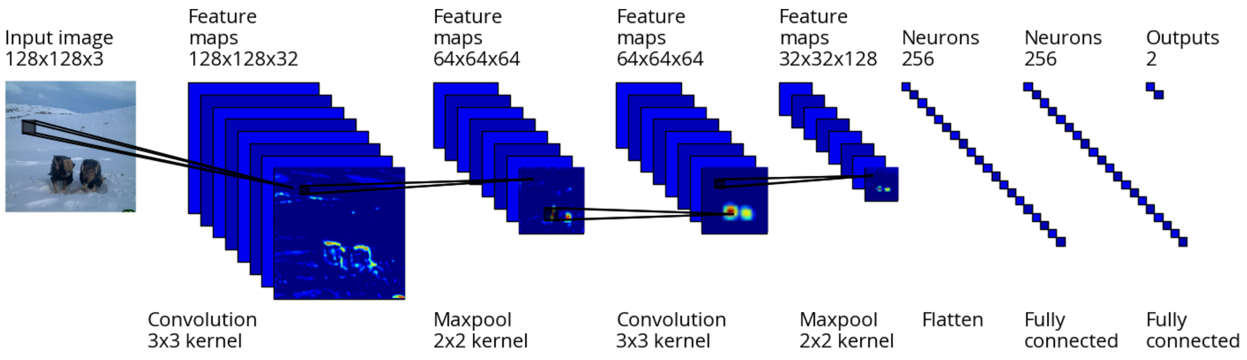


Figure 2.1: Illustration of a CNN architecture.

2.1.2 Training

A CNN is trained using the *backpropagation algorithm*, which finds parameters called *weights*, that minimizes the error between the ground truth training labels and the predicted labels. The algorithm can be explained in two steps that are iterated several times:

- **Feedforward (forward pass):** A *batch* of training images are sent through the CNN which generates a set of predicted labels using its *weight* parameters.

- **Calculate error and propagate back (backward pass):** A *loss function* calculates an error measurement between the true training labels and the predicted labels. This error measurement is thereafter used by an *optimizer* that goes backward through the network tweaking the *weight* parameters. The *optimizer’s* job is to minimize the *loss function*, so the network can make better predictions in the next feedforward step.

An *epoch* is defined as one forward pass and one backward pass (one iteration) of all the training images.

Some commonly used *optimizers* are Stochastic Gradient Descent (SGD) [24], RMSprop [25] and Adam [26], which all have parameters that can be tuned to improve the learning process. The most important parameter is the *learning rate* which controls the rate of change in *weight* parameters during training.

A high learning rate can change the *weights* to aggressively, while a small learning rate can change them too conservatively, resulting in a network that does not learn.

A problem that occurs when training a neural network model is that it tries to memorize the training images instead of trying to generalize from the patterns it observes. This is called *overfitting* and often happens when the network is too complex by having too many parameters, making it overreact to insignificant details in the training data [27]. If the model overfits and loses its ability to generalize, it will have very poor predictive performance on unseen test images. Fortunately, there are several techniques developed in order to minimize overfitting such as soft weight sharing [28] and dropout [29].

2.1.3 Transfer Learning

When working with a small dataset it is common to take advantage of existing CNNs that are already trained on very large datasets, such as ImageNet which contain 1.2 million images and 1000 classes [30]. This concept is called transfer learning, where learning in a dataset is done through the transfer of knowledge from a related dataset that has already been learned.

The standard practice is to load weights from a pre-trained network trained on ImageNet, then fine-tune the weights by continuing training with a smaller dataset. This gives the advantage of exploiting features learned on ImageNet, while adapting the weights to the new dataset. It is possible to keep some of the earlier convolution layers fixed during fine-tuning, reducing the possibility of overfitting from having too many weight parameters.

2.2 Region-based Convolutional Neural Networks

In 2013, Grishrik et. al. presented Region-based Convolutional Neural Networks (R-CNNs), achieving state-of-the-art results on the PASCAL VOC 2012 object detection challenge using CNNs [31]. Object detection introduces the challenge of drawing bounding boxes over all of the objects in an image, in addition to classifying the objects. R-CNN bridged the gap between image classification and object detection by splitting the process into three general steps: the region proposal step, the feature extraction step, and the classification step.

R-CNN uses an external region proposal algorithm called Selective Search (SS) [32] to generate 2000 class-independent region proposals from each image. SS find Regions of Interest (RoIs) in an image by exploring pixel areas of different sizes and grouping together adjacent pixels by texture, color, or intensity to identify objects. These proposals have the highest probability of containing an object and are sent through a trained CNN to extract a fixed feature vector from each region. R-CNN adds a set of linear Support Vector Machines (SVMs) at the end of the CNN to classify whether the region contains an object, and if so what object. After the region has been classified, the feature vector is also used in a regression model to obtain more accurate coordinates for the bounding box. As a final step, a greedy Non-Maximum Suppression (NMS) algorithm is used to remove bounding boxes that have a significant overlap with each other and refer to the same object [33].

At the time, R-CNN was the best in terms of detection accuracy but had the downside of being very slow. It took 84 hours to train on the relatively small PASCAL VOC 2007 dataset and detection took around 53 seconds per image [34]. The slow training can be attributed to the complicated training pipeline where three different models had to be trained separately (the CNN, SVM and regression model). Detection was slow because it required a forward pass of the CNN for every single region proposal for every single image (2000 forward passes per image). Grishrik et. al. solved these problems and presented an improved version of his method in 2015, called Fast R-CNN [34].

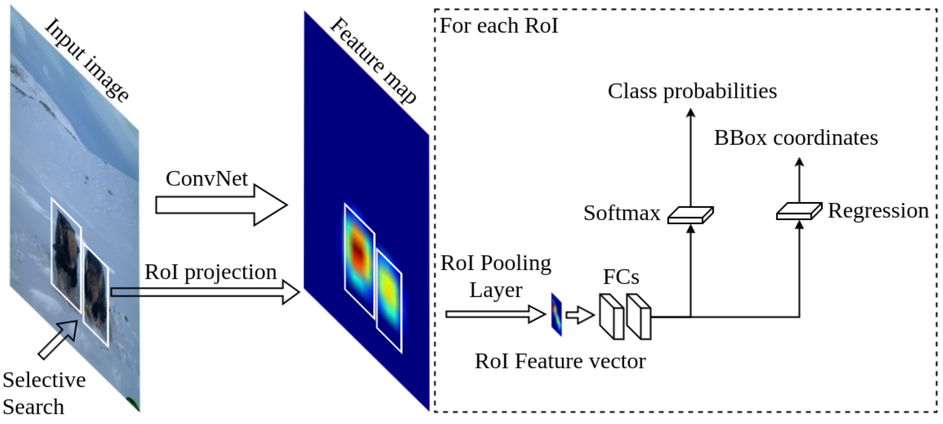


Figure 2.2: Illustration of the Fast R-CNN method. Region proposals with high object probability are shown in white bounding boxes.

Fast R-CNN introduces a technique called Region of Interest (ROI) Pooling that enables shared computations across all 2000 region proposals. Figure 2.2 show the Fast R-CNN object detection method. Instead of sending region proposals through the CNN individually, the entire image is used as input to generate a convolutional feature map. Region proposals from SS are projected onto the feature map and ROI pooling extracts a fixed-length feature vector for each ROI. The feature vector is subsequently sent through a set of fully connected layers, then a softmax layer outputs a probability for all object classes, including a negative "background" class if the region does not contain an object. Bounding box regression is also integrated in the CNN resulting in a single end-to-end architecture, mitigating the complicated training procedure in the previous version.

Training and detection time with Fast R-CNN was reduced to 10 hours and 0.32 seconds respectively, yielding a significant speedup compared to the original R-CNN. It also achieved better detection accuracy on the PASCAL VOC challenge by fine-tuning using pre-trained models from ImageNet [34].

Even with all these improvements, there was still one bottleneck remaining - the external region proposal algorithm. The slowest step in the object detection method was the SS algorithm that generates potential bounding boxes or ROIs in the image. A few months after the release of Fast R-CNN, this bottleneck was removed with the implementation of a novel Region Proposal Network (RPN) which was presented in the newest and current version, Faster R-CNN.

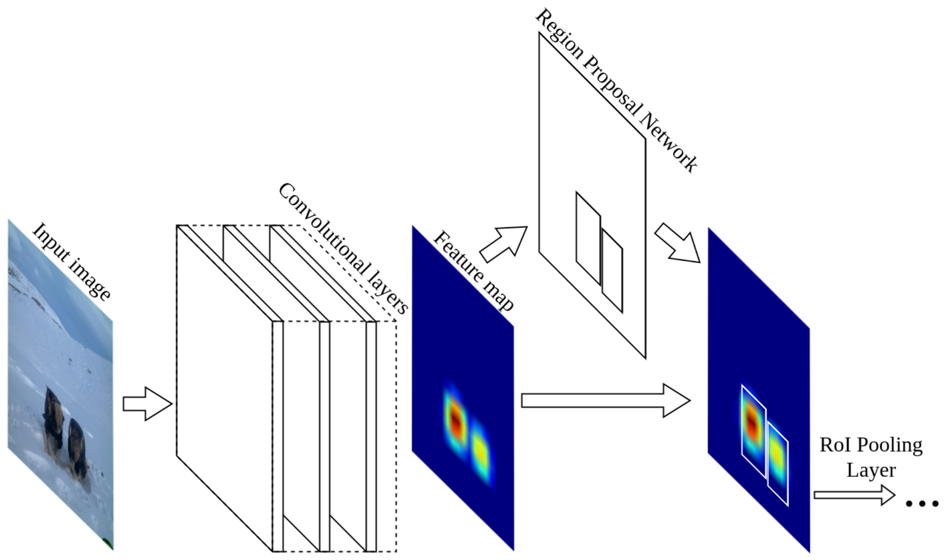


Figure 2.3: Illustration of the Faster R-CNN method. Region proposals with high object probability are shown in white bounding boxes.

The Faster R-CNN object detection method is illustrated in Figure 2.3. It shows that the previous region proposal method has been removed and replaced by the RPN, which is merged with the Fast R-CNN object detection network. This gives the benefit of shared computation on the feature map generated by the initial convolutional layers of the network, allowing nearly cost-free region proposals.

Faster R-CNN is based on the deep learning framework Caffe [35], has been made publicly available [36] and is used in our unified detection system. The implementation has previously been used as the foundation of several winning entries in the ILSVRC and Microsoft COCO 2015 competitions and is considered state of the art in object detection [16]. A more in-depth description of the RPN is given in the following subsection.

2.2.1 Region Proposal Network

Region proposals is a vital part of object detection. Too many region proposals increase the chance of False Positives (FPs), e.g. detecting objects that are not present. While having too few can lead to more False Negatives (FNs), e.g. not detecting objects that are present. Using the RPN, Faster R-CNN managed to reduce the number of proposals needed at test time from 2000 to 300, with little to no difference in detection accuracy [16]. This demonstrates its ability to find good proposals.

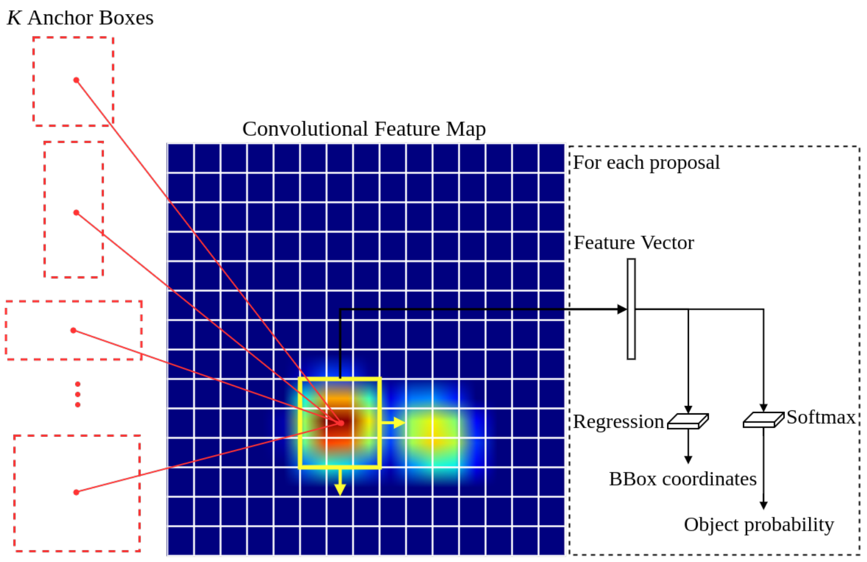


Figure 2.4: Illustration of the RPN. Sliding window is shown in yellow and anchor boxes are shown in red.

The RPN finds region proposals by sliding a window over the shared CNN feature map as shown in Figure 2.4. Each sliding window is mapped to a lower-dimensional feature vector, which is subsequently used in a box-regression layer and a box-classification layer. The regression layer outputs bounding box coordinates while the classification layer outputs an objectness score, which is the estimated probability of “object” or “not object”.

To generate a multitude of bounding boxes, the RPN simultaneously outputs K bounding box proposals at each sliding position. These proposals are computed relative to K reference boxes, called anchor boxes. For each proposal, the regressor computes 4 offset values (xcenter, ycenter, width, height) to its corresponding anchor box. Using anchor boxes, region proposals can be made over multiple scales and aspect ratios, while only relying on images and feature maps of a single scale. The authors of Faster R-CNN hand-picked 9 anchor boxes to cover the most common object scales and aspect ratios.

2.3 Single Shot Detectors

More recent object detection methods take inspiration from Faster R-CNN and has made further improvements, achieving better detection accuracy and real-time detection speed on PASCAL VOC datasets. Single shot detectors provide object detection in a single shot eliminating the bounding box proposal stage and the subsequent feature resampling stage found in Faster R-CNN. We use two popular single shot detectors in our unified detection system, called You Only Look Once (YOLO) v2 and Single Shot MultiBox Detector (SSD).

YOLOv2 is an object detection method released in late 2016 by Redmon et. al. [17]. It is implemented on top of Darknet, an open source neural network framework written in C and CUDA by the same author [37]. The method predicts bounding boxes, objectness score and object class probabilities with a region layer that uses features from the entire image in one evaluation, instead of generating and classifying proposals. All bounding boxes are predicted across all classes for the image simultaneously, allowing the network get a global view on all of the objects in the full image [38].

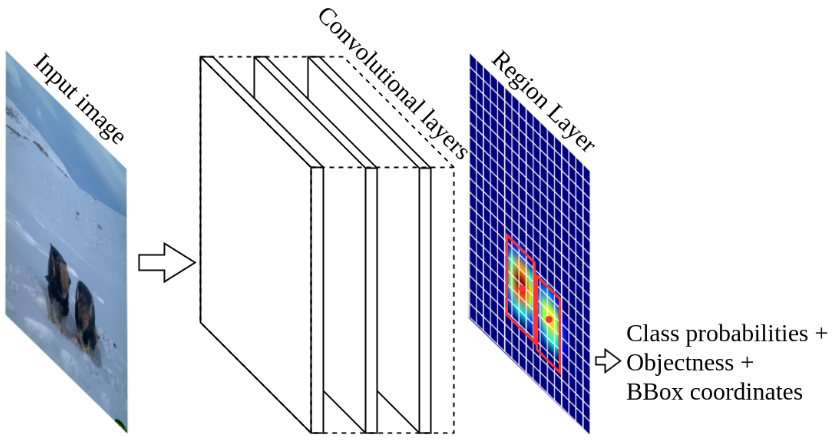


Figure 2.5: Illustration of the YOLOv2 method. Refined anchor boxes are shown in red.

The input image is sent through a convolutional network where it is divided into an S×S grid as shown in Figure 2.5. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. YOLOv2 adopts the anchor box scheme from the RPN in Faster R-CNN and makes some small modifications to predict bounding boxes. Instead of predicting unconstrained offsets allowing an anchor box to end up anywhere in the image, it predicts constrained bounding box coordinates relative to the location of the grid cell. This simplifies the bounding box prediction leading to a more stable network [17].

To generate additional and more diverse training data, YOLOv2 applies various transformations to the input images, also known as data augmentation. The image transformations include random crops, rotations, and hue, saturation, and exposure shifts [17]. This data augmentation strategy improved detection accuracy considerably compared to Faster R-CNN, which only use the original image and a horizontal flip to train.

YOLOv2 outperforms state-of-the-art methods like Faster R-CNN on the PASCAL VOC 2007 dataset while having real-time detection at an impressive 40 frames per second (FPS). Its main issue is that it struggles with small objects because of low input image resolution, resulting in very coarse features for predicting bounding boxes. Redmon et. al. made small improvements on this issue by retrieving and merging a larger feature map from early stages of the network with the later coarser feature map [17].

The SSD method addresses the small object issue in a different way while being very similar to YOLOv2. Instead of operating on a single-scale feature map, SSD adds several extra convolutional layers to the end of the network and predict on multi-scale feature maps as illustrated in Figure 2.6.

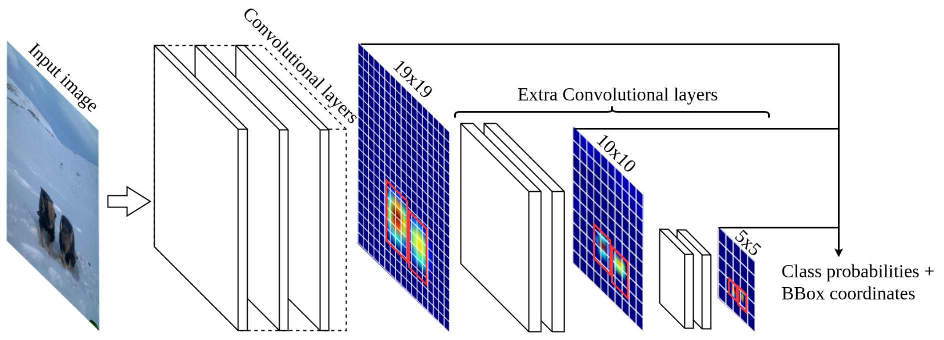


Figure 2.6: Illustration of the SSD method. Refined anchor boxes are shown in red.

Detection is done using Multibox layers which compute bounding box offsets relative to anchor box shapes (called default box in their paper) for each feature map grid cell in each feature map. SSD adopts the data augmentation strategy from YOLOv2 and add techniques to handle objects of multiple scales, including random crops acting as a “zoom in“ operation and an expansion scheme as a “zoom out“ operation [18]. It is implemented using Caffe and is released as open source [39].

YOLOv2 and SSD are very close in terms of detection accuracy and speed using similar low-resolution input images. It is safe to say that they both contributed to getting object detection on a new level, achieving reliable and fast detection.

3 Related Work

A thorough search of relevant literature show several systems that automate animal identification in camera trap images, but there are few that focus on detection, where the animals are localized in addition to being identified. The scarce research done on wild animal detection frequently use small and exclusive datasets containing only one or few animal species. It is clear that larger public camera trap datasets with bounding box annotations and several animal species are needed for further advancement in the field.

Norouzzadeh et. al. recently presented a system for automatic animal identification on the Snapshot Serengeti dataset using deep CNNs [40]. The Snapshot Serengeti dataset is a large public dataset of wild animals containing 3.2 million annotated camera trap images and 48 different animal species [8]. From 757 000 annotated images that contained an animal, they created a training set of 707 000 images, and a test set consisting of the remaining 50000 images. Using a CNN architecture called ResNet-152 [41] they achieved 92% classification accuracy on their test set, exceeding the performance of previous methods. This work shows promise in the classification of images with a single animal, but it does not address the challenge of localizing several animals. Our system can automatically detect multiple animals and multiple different species in images, providing more information for reliable animal study and documentation.

In 2009 Wawerla et. al. described a novel "motion shapelet" algorithm for automatically detecting wild bears in video frames captured by cameras at the arctic circle [42]. The algorithm is an extension of the shapelet features used for pedestrian detection, described in [43]. They combine several low-level features into mid-level ”motion shapelet“ features that are more informative and descriptive with regard to their object class. For training, they manually cropped 451 bounding boxes with a bear and 8000 negative bounding boxes with background from images. Furthermore, they used 405 positive images containing at least one bear and 16000 negative images not containing bears as a test set. Their results show that Wawerla et. al. can detect 76% of the images containing bears at 0.001 false positive images per image examined. This is similar to our work in detecting animals in camera trap images from the Arctic tundra, but their experimentation is more focused on detecting the presence, rather than correctly localizing and counting multiple occurrences of bears in images. Moreover, they do not use CNNs and only detect two object classes, ”bear“ or ”not bear“.

A more recent paper by Parham et. al. tackles the problem of zebra detection in real world images using Faster R-CNN and YOLOv1 [44]. The paper presents several challenges in animal detection, such as difficult viewpoints of the zebras and occlusion from multiple overlapping zebras. They create a manually labeled dataset of 2500 images, containing 3541 bounding boxes of plains zebras and 2672 bounding boxes of Grevy’s zebras. YOLOv1 was the best detector in their evaluation with a detection accuracy of 55.6% for plains zebras and 56.6% for Grevy’s zebras. Parham et. al. is closely related to our work in that it compares Faster R-CNN and YOLOv1 on a dataset containing wild animals. However, we use a newer, improved version of YOLO in YOLOv2 and additionally use SSD in our comparison. Our dataset which is described in Chapter 4, can also be seen as more challenging since it is more imbalanced and contains nine different animal species, where several are very similar in appearance.

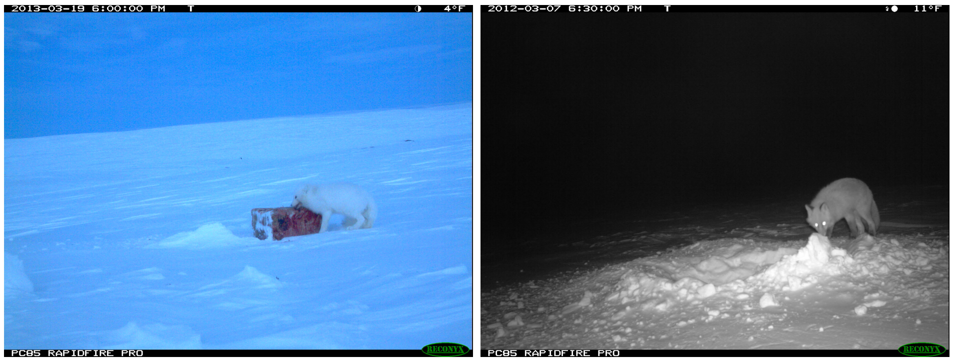
The closest work to ours is that of Zhang et. al., who propose a new method for animal detection in highly cluttered camera trap images [45]. The method uses joint deep CNN features and histogram of oriented gradient features encoded with Fisher vectors to get an efficient feature description for animal detection. For evaluation, they create a dataset with 800 camera-trap image sequences containing 6493 animals of 23 different species. Similar to our dataset, the images are in both daytime color and nighttime grayscale formats. Their experimentation compares YOLOv1, Fast R-CNN, Faster R-CNN and the proposed method, which achieved an average F-measure score of 82.1%. Instead of developing new detection methods, this thesis show that there is much potential in experimenting and adapting existing cutting-edge methods to the animal detection problem.

4 Training Methods

This chapter describes methods used when training an object detection model in our system. It covers challenges and decisions made when preparing our dataset for object detection, including bounding box annotation, dataset split, and data format. Furthermore, it explains techniques used to improve training for our specific dataset, including custom anchor boxes, image preprocessing and Online Hard Example Mining (OHEM).

4.1 Dataset Preparation

The dataset provided by COAT contained 1 849 076 time-lapse images taken from 2011 to 2016 by their camera traps in Finnmark, Norway. 37 camera traps was deployed every year, spread out over five different areas: Stjernevann, Komag, Ifjord, Nyborg and Gaissene. It is important to be aware of these areas since they will affect how we split our data into training and validation sets for our experiments. The cameras have an infrared flash so all pictures taken during the night are without color, while pictures taken during the day are with color, as seen in Figure 4.1. We decided against splitting night and day images in separate datasets because we wanted to assess the object detection methods ability to handle a mixture of greyscale and color images.



(a) Daytime image with an arctic fox. **(b)** Nighttime image with a red fox.

Figure 4.1: Example images from the COAT dataset. It contains a mixture of color and greyscale images.

COAT also provided Comma-separated Values (CSV) files with annotations for half of the images in the dataset. The CSV files did not contain any filenames, only image metadata and animal classifications, so we had to get creative to find out which images the annotations corresponded to. Fortunately, each image had metadata stored in Exchangeable Image File Format (EXIF). We created a Python script to extract the date and time from each image, along with camera information, to match them with the annotations. This was quite a time-consuming task because of the number of images to process.

For our object detection task, we were only interested in images containing animals, so we used the annotations to sort images into folders representing each class. We will denote our dataset as Baitcam, since the camera traps were designed to attract scavengers with the use of bait. The initial Baitcam class distribution can be seen in Table 4.1.

Class	Images
ArcticFox	724
Crow	732
WhiteTailedEagle	832
GoldenEagle	2050
Raven	49472 (2050)
RedFox	8870 (2050)
Reindeer	1286
SnowyOwl	56
Wolverine	704
Total	64726

Table 4.1: Initial Baitcam class distribution. It shows the number of images for each class.

Only 7% of the annotated images contained animals, the rest were either empty or had bad quality due to environmental or camera factors. The massive amount of empty images makes sense since the camera traps capture images in 5-minute intervals, regardless of animal presence. Keep in mind that this distribution might contain duplicate images since there could be several different animal species in one image.

To reduce the heavy class imbalance we decided to decrease the majority classes Raven and RedFox to 2050 images, in line with the GoldenEagle class. The Baitcam dataset was now ready for the tedious part of our dataset preparation, bounding box annotation.

4.1.1 Animal Bounding Box Annotation

In order to obtain training and validation data, we manually annotated all the animals in the Baitcam dataset with ground truth bounding boxes. We used LabelImg [46] for this task, a graphical annotation tool for labeling object bounding boxes in images. For each image, the annotations were saved as an Extensible Markup Language (XML) file in PASCAL VOC format, the format used by ImageNet. The XML file stores information for each bounding box in the image, including class name and bounding box pixel coordinates as (xmin,ymin,xmax,ymax).

Manual bounding box annotation involved several challenges. Firstly, we had to be able to correctly identify the animals present in the images. This could be difficult with animals that are similar such as white-tailed eagles and golden eagles or ravens and crows. Fortunately, we already sorted the images in class folders using COATs annotations, which made the identification process much easier. Secondly, we had to be rational when choosing what animals to include or exclude from our annotation. Images of animals that was unrecognizable because of size or position were excluded, while animals that were fully or partially visible and recognizable was included. The Reindeer class was particularly hard to annotate because they were often distant from the camera, making them very small. This is apparent in the number of excluded images in Table 4.2.

To be consistent in our annotation, we followed the PASCAL VOC guidelines [47] for bounding box annotation as best we could. Our final Baitcam animal object distribution after bounding box annotation can be seen in Table 4.2. As indicated by the initial distribution, the dataset is still quite unbalanced with Raven being the majority class by far. The number of included and excluded images shows how the Reindeer class was more challenging to annotate than the rest.

Class	Objects	Images used	Images not used
ArcticFox	577	535	177
Crow	841	444	44
WhiteTailedEagle	804	396	26
GoldenEagle	2191	1652	108
Raven	4956	1739	244
RedFox	1574	1388	418
Reindeer	643	260	973
SnowyOwl	45	31	1
Wolverine	589	571	107
Multiple Species	-	983	-
Total	12220	7999	2098

Table 4.2: Final Baitcam object distribution after bounding box annotation. It shows the number of objects, images used and images not used for each class.

4.1.2 Training and Validation Imagesets

We split the Baitcam dataset into two parts. A training imageset that contain images and ground truth annotations used for training a model, and a validation imageset with images and ground truth annotations used for evaluating the trained models. By looking at how the model performs on the validation set, we can reiterate our training methods and try to improve them for the Baitcam dataset. All images in COATs full dataset, beside the 7999 images in the Baitcam dataset, can be used to see if the model performs well on new images.

Since we are dealing with stationary time-lapse images taken at a high sampling rate, it is important to be aware of potential pitfalls when splitting the dataset into training and validation sets. A common mistake in the dataset split phase is when close to identical images gets mixed in both the training and validation set. Getting a correct detection on a validation image that is identical to a training image is not very useful, and will not help us determine if the network is generalizing well to unseen images. This could happen if we chose to split the data randomly and the dataset contained 50 identical images of a sleeping red fox taken five minutes apart.

Instead, we take a more sensible approach and select validation images for each class according to the areas in which the camera traps are placed, while all other images are used as the training set. This means that a sequence of identical images will not be mixed in the training and validation set since they will contain images from different areas. Table 4.3 shows the validation set and the camera trap area that each class was chosen from. We try to balance the dataset distribution as best we could by choosing areas with 20-30% objects,

Class	Objects	Camera Trap Area
ArcticFox	113/577 (19.58%)	Gaissene
Crow	247/841 (29.37%)	Ifjord
WhiteTailedEagle	266/804 (33.08%)	Komag
GoldenEagle	575/2191 (26.24%)	Stjernevann
Raven	1366/4956 (27.56%)	Gaissene
RedFox	289/1574 (18.36%)	Komag
Reindeer	150/643 (23.33%)	Stjernevann + Ifjord
SnowyOwl	9/45 (20.00%)	Random
Wolverine	140/589 (23.77%)	Ifjord + Nyborg
Total	3155/12220 (25.81%)	

Table 4.3: Baitcam validation set distribution based on camera trap area. It shows the number of objects chosen from each class and which camera trap area they were chosen from.

following the common training/validation split ratio of 75/25%. SnowyOwl images was chosen randomly because they were only present in the Komag area. Picking a single camera trap area as the validation set was also considered, but it would not cover all the classes sufficiently because of varying animal presence in the different areas.

4.1.3 Dataset Format Conversion

The three object detection methods in our unified detection system require different data formats when processing the dataset. Faster R-CNN use the standard PASCAL VOC data format where each image has an XML annotation file and each imageset (train and validation) is defined by a text file. XML annotations contain ground truth bounding box information as described in Section 4.1.1, while an imageset text file list all image filenames in the respective imageset. These files, along with the images, are used to create a custom image database and ROI database when training or evaluating a model with Faster R-CNN.

Since YOLOv2 is implemented on top of the Darknet framework, it requires data in a different format. Annotations in Darknet are text files instead of XML, and Darknet imageset files contain absolute paths to images instead of filenames. Furthermore, Darknet represents bounding box pixel coordinates as (xcenter,ycenter,width,height) instead of (xmin,ymin,xmax,ymax) and use pixel coordinates that are normalized between 0 and 1.

SSD on the other hand, store images and XML annotations in Lightning Memory-Mapped Databases (LMDBs), which can be more efficiently processed by the

Caffe framework. Caffe has a tool which can generate LMDBs, by using imageset files that contain relative paths to the images and XML annotations.

We create two Python scripts as part of our data utilities, to conveniently convert existing PASCAL VOC imagesets and annotations to the data formats described above. The conversion scripts require a directory with data in PASCAL VOC format and a label map file which contains the different class labels. After conversion, the dataset is ready to be used by all three object detection methods in the system.

4.2 Custom Anchor Boxes with *k*-means Clustering

As described in Chapter 2, all of the detection methods in our system use anchor boxes to make bounding box predictions. While Faster R-CNN and SSD use hand-picked anchor boxes to accommodate more general sizes and aspect ratios of objects, YOLOv2 use anchor boxes that are optimized for their PASCAL VOC training set [17]. By picking good anchor boxes related to the objects in Baitcam we will make it easier for the networks to make good detections. We implemented a k-means clustering algorithm with inspiration from Redmon et. al. [17], to create custom anchor boxes for the objects in the Baitcam training set.

k-means clustering is an iterative algorithm that attempts to assign data points into groups, called clusters, such that similar data points are put together in the same cluster. It makes use of a distance measure, often Euclidean distance, to generate k optimal cluster center points, called centroids. The best clusters containing similar data points are found by minimizing the total sum of distances between the data points and their closest centroid.

We modify the original algorithm by using inverse Intersection over Union (IOU) as our distance measure and the ground truth bounding boxes from the Baitcam training set as data points. IOU is an evaluation metric used in object detection measuring an overlap ratio between two bounding boxes. An IOU of 1 indicates that the boxes overlap completely, while an IOU of 0 means that the boxes do not overlap at all. A more detailed description of the IOU metric is given in Section 7.3

Minimizing the inverse IOU is equivalent to maximizing the IOU, so our k-means algorithm attempts to cluster bounding boxes that are similar in size and aspect ratio. The effectiveness of the modified algorithm is measured by

Code Listing 4.1: Implementation of k-means clustering to find custom anchor boxes for a set of bounding boxes.

```python
import numpy as np

def k_means_anchor_boxes(k, centroids, bboxes, iteration_cutoff=25):
    anchor_boxes = []
    best_avg_iou = 0
    best_avg_iou_iteration = 0
    iter_count = 0

    while True:
        clusters = [[] for _ in range(k)]
        clusters_iou = []

        for bbox in bboxes:
            idx, distance = find_closest_centroid(bbox, centroids)
            clusters[idx].append(bbox)
            clusters_iou.append(1. - distance)

        centroids = [np.mean(cluster, axis=0) for cluster in clusters]

        avg_iou = np.mean(clusters_iou)
        if avg_iou > best_avg_iou:
            anchor_boxes = centroids
            best_avg_iou = avg_iou
            best_avg_iou_iteration = iter_count

        if iter_count >= best_avg_iou_iteration + iteration_cutoff:
            break

        iter_count+=1

    return anchor_boxes, best_avg_iou
```

taking the average of all bounding box IOU to their closest centroid. Code Listing 4.1 show our *k_means_anchor_boxes* procedure and how it converges when the average IOU stops improving. The best average IOU is returned, along with its respective cluster centroids as our custom anchor boxes.

Figure 4.2 shows the average IOU of all bounding boxes in the Baitcam training set to their closest anchor box. We compare the default anchor boxes from YOLOv2 and Faster R-CNN to our custom anchor boxes generated with *k*-means clustering. The results show that the algorithm successfully finds anchor boxes that are optimized for our Baitcam dataset and should improve training compared to the default anchor boxes. Default anchor boxes for SSD are omitted from this comparison because it has several sets of anchor boxes with different scales, depending on the size of each feature map used in training.

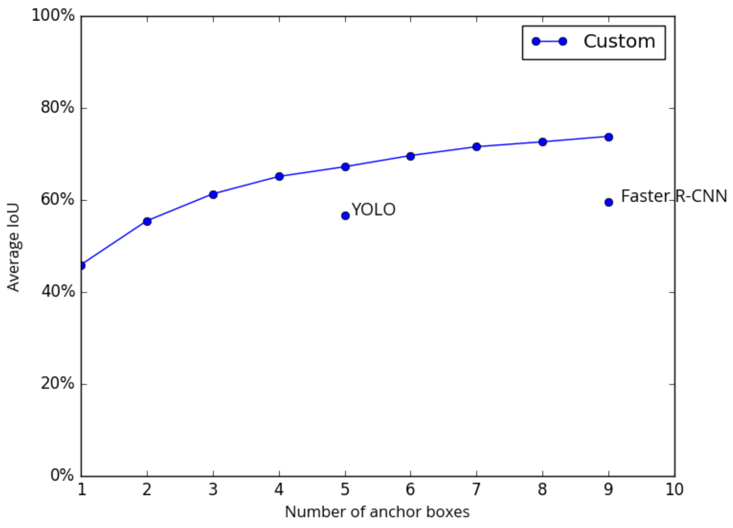


Figure 4.2: Custom vs. default anchor boxes effectiveness graph. It shows the average IOU overlap of all bounding boxes in the Baitcam training set to their closest anchor box.

The relative size difference between our custom anchor boxes and the default YOLOv2 anchor boxes which are specialized for the PASCAL VOC dataset are shown in Figure 4.3. We see that the objects in Baitcam are generally very small with similar aspect ratios, having much more square shapes with some predominance in width. These characteristic makes sense because the original Baitcam images have a 4:3 aspect ratio and are very high resolution, causing most objects to be small-scale. We show the effects of using custom anchor boxes in our evaluation.

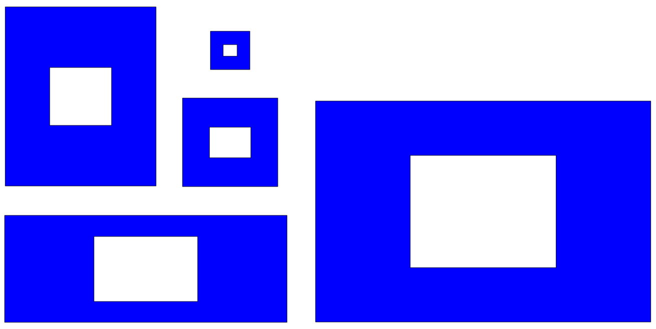


Figure 4.3: Illustration of custom and default anchor boxes. It shows the relative size of Baitcam custom anchor boxes (white) compared to default YOLOv2 anchor boxes (blue).

4.3 Image Preprocessing

The images in the Baitcam dataset have a large size of $2048 \times 1536 \times 3$ (width $\times$ height $\times$ depth) pixels. Previous work shows that cropping the black borders which contain information like date, time and temperature had a positive effect on classification accuracy [11]. We follow this strategy and do the same on Baitcam, resulting in $2043 \times 1472 \times 3$ images. They also need to be resized for training to be able to fit in memory and to avoid the curse of dimensionality [48].

Each object detection methods respective framework will resize the input image to a specified resolution when training. We train with the default low-resolution setting and a custom high-resolution setting. Since most objects in our images are very small-scale, we feel that a higher input image resolution would be beneficial on the Baitcam dataset.

With default resolution, Faster R-CNN train on images where the shorter side is scaled down to 600 pixels, keeping the image aspect ratio. This means our Baitcam images are resized to $833{\times}600$. Whereas YOLOv2 and SSD resize input images to a fixed square shape of 416×416 and 300×300 respectively. These default sizes worked as a reference point for further experimentation.

Training a few Faster R-CNN models with default resolution showed that it was having a hard time with smaller objects as previously expected. We decided to investigate the ground truth bounding boxes in the training set and found that the smallest object size was $\approx 20 \times 20$ pixels. Resizing the image to 833×600 would make this object $\approx 8 \times 8$ pixels, which would only correspond to an area of $\approx 0.5 \times 0.5$ pixels on the convolutional feature map used by the RPN. Two options could help detection on small objects in Baitcam: change the CNN architecture for detection on larger feature maps, or increase the input image size. We tried both options and found that increasing input image size gave the best results.

Based on the previously mentioned investigation and some experimentation, we chose to approximately double the default input image size for each object detection method. Our custom high-resolution training size is 1644×1184 for Faster R-CNN, 832×832 for YOLOv2 and 608×608 for SSD. We show the effects of training on default low-resolution and custom high-resolution input images in our evaluation.

4.4 Online Hard Example Mining

A problem when training an object detection model is the large imbalance between the number of ground truth objects and the number of background regions in an image. OHEM is a technique that works to solve this challenge by choosing hard ROIs to train on. It is more useful for a model to train on ROIs that it struggles on than training on easy ROIs containing only background. This technique is already implemented in the SSD object detection method and we add the option to use it when training a Faster R-CNN model as well. OHEM was implemented for Fast R-CNN and achieved significant improvements in detection accuracy compared to the original implementation [49]. Open source code of the implementation was made available [50] and we include it in the Faster R-CNN method in our system with some small adjustments.

A standard Faster R-CNN model is trained on ROI mini-batches extracted from $N = 2$ training images that are chosen randomly and uniformly. The mini-batch consist of 64 ROIs that are uniformly sampled from the object proposals in each image, giving a total mini-batch size of $B = 128$. Since foreground regions are extremely rare compared to background regions, Faster R-CNN samples the mini-batch as 25% foreground ROIs and 75% background ROIs. A ROI is labeled as a foreground object class if it has an IOU overlap of at least 0.5 with a ground truth bounding box in the image. It is labeled as background if it has an IOU in the interval $[0.1, 0.5)$ with a ground truth bounding box. The lower threshold is set to 0.1 to avoid pure background examples and behaves as an approximation to hard negative mining, assuming that regions having some overlap with the ground truth are more likely to be hard or confusing [34].

OHEM completely removes the foreground-to-background ratio heuristics and explicitly choose the ROIs that are most difficult for the mini-batch, making training more effective and efficient. This is done by adding a read-only copy of the ROI network, which runs a forward pass on all ROIs proposed by the RPN as seen in Figure 4.4. At the end of the read-only network, a Hard ROI Module picks the B/N ROIs with the highest loss as the mini-batch. These proposals represent the regions that are hardest to learn for the network. Subsequently, the mini-batch is sent through the normal forward-backward pass network for training. The downside of using OHEM is slower training due to more computation from adding the read-only network. We show the effects of training Faster R-CNN with OHEM in our evaluation.

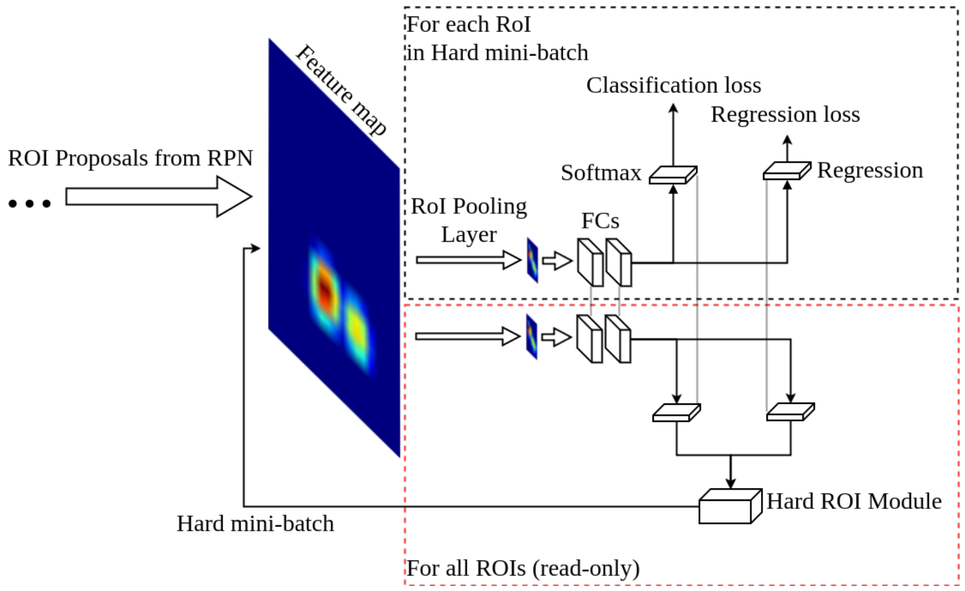


Figure 4.4: Illustration of the OHEM technique when training Faster R-CNN. The forward-pass only network (dashed red) shares weights (grey) with the forward-backward-pass network (dashed black).

5 Design and Architecture

This chapter describes the design of our unified detection system with a focus on three main actions that it can perform. We explain how the system trains and evaluates an object detection model that, in turn, can be used to perform detection on new images. Additionally, we show and describe the CNN architectures used by each object detection method in our system.

5.1 System Design

Figure 5.1 show the design of our system that unifies three different object detection methods. The dashed boxes with bolded text represent directories and indicate the system's directory structure. There are three types of actions when running the system: train a neural network model, evaluate a trained model and detect with a trained model. Before starting any of the actions, the dataset has to be created and prepared as described in Section 4.1. The resulting Baitcam dataset directory is shown with all subdirectories containing the necessary data for each object detection method.

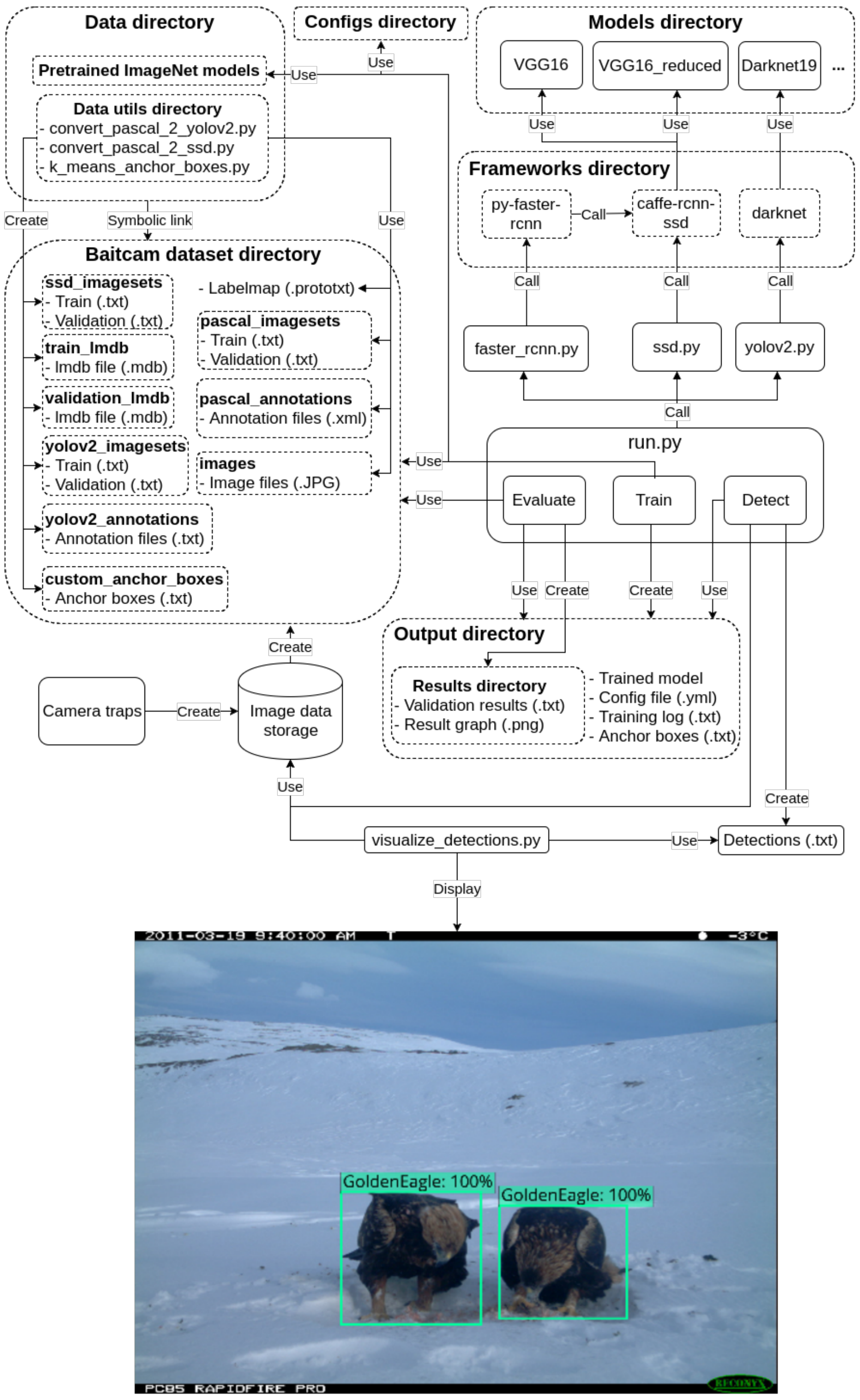


Figure 5.1: Design of our unified detection system.

- **Train**: The default action for the system is training a model. Through command-line arguments, it is possible to specify which object detection method, model and dataset to use, along with the number of training iterations. These options are added to the default configuration settings for further use in the system. A custom configuration file from the configs directory can also be given to override default settings.

  The configuration and custom anchor boxes are saved in the output folder. This allows the system environment to easily be restored in the evaluate and detect actions. Subsequently, run.py will call a wrapper for the appropriate object detection method, which in turn calls its respective framework that loads the selected model and its pre-trained ImageNet weights, then starts the training process. The wrapper acts as an intermediate stage that prepares the training process by initializing frameworks and creating or adapting model definition files for the given dataset.

  During training, the frameworks will print stats to the terminal, including iteration count and average loss. This output is logged by redirecting stdout and stderr to a log file that can be used to study the loss trend of the model. Snapshots of model weights are saved to the output directory at a user specified iteration interval, along with the final model weights when training is finished.

- **Evaluate**: When training is finished, the model can be evaluated on a validation imageset. Given an output directory, the system will load settings from the configuration file, the anchor boxes used, and the trained model weights. The wrapper calls the relevant framework to load the model and run forward passes on the validation data, creating a set of output result files (one file for each class). Each result file will list all detections for its particular class in the following PASCAL VOC format: (image filename, probability score, xmin, ymin, xmax, ymax). When all detections have been made, they are compared with ground truth annotations, giving the detection accuracy of the model. Section 7.3 gives a detailed description of the detection metrics used for evaluating the models.

- **Detect**: When the model's detection accuracy has been deemed satisfactory, it is ready to be used on new images. Detect needs a directory containing images and an output directory with the trained model. Similar to evaluate, the model is loaded and runs a forward pass on each image. A text file is created for the image directory where the resulting detections are saved as: (image path, class label, probability score, xmin, ymin, xmax, ymax). Afterward, the text file can be used in a visualization script that draws the bounding boxes with its respective class and probability score on the image, to be displayed or saved on disk.

Optional command-line arguments can be given for training without a pre-trained ImageNet model and to specify which GPU to use for training. It is also possible to set two commonly used object detection thresholds in evaluate and detect. The first is a confidence threshold, telling the system to only keep detections with a probability higher than the given threshold. The second is a NMS threshold used to remove duplicate detections. If several detections contain the same object and have an IOU overlap higher than the threshold, they will be suppressed by NMS, only keeping the highest scoring detection.

5.2 Convolutional Neural Network Architectures

Figure 5.2 show an overview of the CNN architectures used by each object detection method in our system.

As described by Girshick et. al., Faster R-CNN use the very deep VGG16 [51] as its base network and modifies it by adding the RPN and ROI pooling layer [16]. It has a total of 13 convolutional layers with filter size 3×3 pixels, and 4 max-pooling layers with filter size 2×2 and stride 2. Each max-pooling filter divides the spatial dimensions in half, allowing the number of convolutional filters to be doubled. With default input image size, the network will generate a 53×38 pixel feature map that is used by both the RPN and ROI pooling layer. The ROI feature vectors that are extracted from the ROI pooling layer has a size of 7×7, and are used for classification and bounding box regression. Table 5.1 show that Faster R-CNN with VGG16 has 136.8 million weight parameters, where most of them are in the last fully connected layers.

YOLOv2 is extended from a base network called Darknet19, which has 19 convolutional layers and 5 max-pooling layers [17]. Similar to VGG16, it mostly uses filter size 3 and doubles the number of filters after every pooling layer. The base network is extended with 3 additional 3×3×1024 convolutional layers at the end, instead of using fully connected layers like VGG16, making it a Fully Convolutional Network (FCN). Related FCNs has shown state-of-the-art results, while significantly reducing the number of parameters and computation needed by the network [52]. This reduction is also apparent in YOLOv2 when comparing the number of parameters with Faster R-CNN, shown in Table 5.1.

A 1×1×64 pass-through layer is added to include more fine-grained features from previous layers, as described in Section 2.3. Figure 5.2 shows that the pass-through layer retrieves and merges features from the final 3×3×512 layer with the second to last convolutional layer. With default input image size, the network will generate a 13×13 pixel feature map that is used for detection by the region layer.

SSD is also based on the VGG16 network but is modified to a reduced VGG16. The fully connected layers are converted to convolutional layers and the last max pool layer is changed to size 3×3 and stride 1. It is additionally reduced by subsampling the parameters from the converted fully connected layers [18]. Table 5.1 show that SSD with VGG16_reduced has the fewest number of parameters of all the object detection methods used in our system.

As described in Section 2.3, several extra convolutional layers are added to the end of the SSD network to support multi-scale training. With default input image size, the network generates multiple feature maps with a size between 38×38 and 1×1 pixels that are used by the multibox layer for object detection, as shown in Figure 5.2.

	Faster R-CNN (VGG16)	YOLOv2 (Darknet19)	SSD (VGG16_reduced)
Number of parameters	136.8	50.6	24.8

Table 5.1: Number of weight parameters (millions) in the CNNs used by each object detection method in our system.

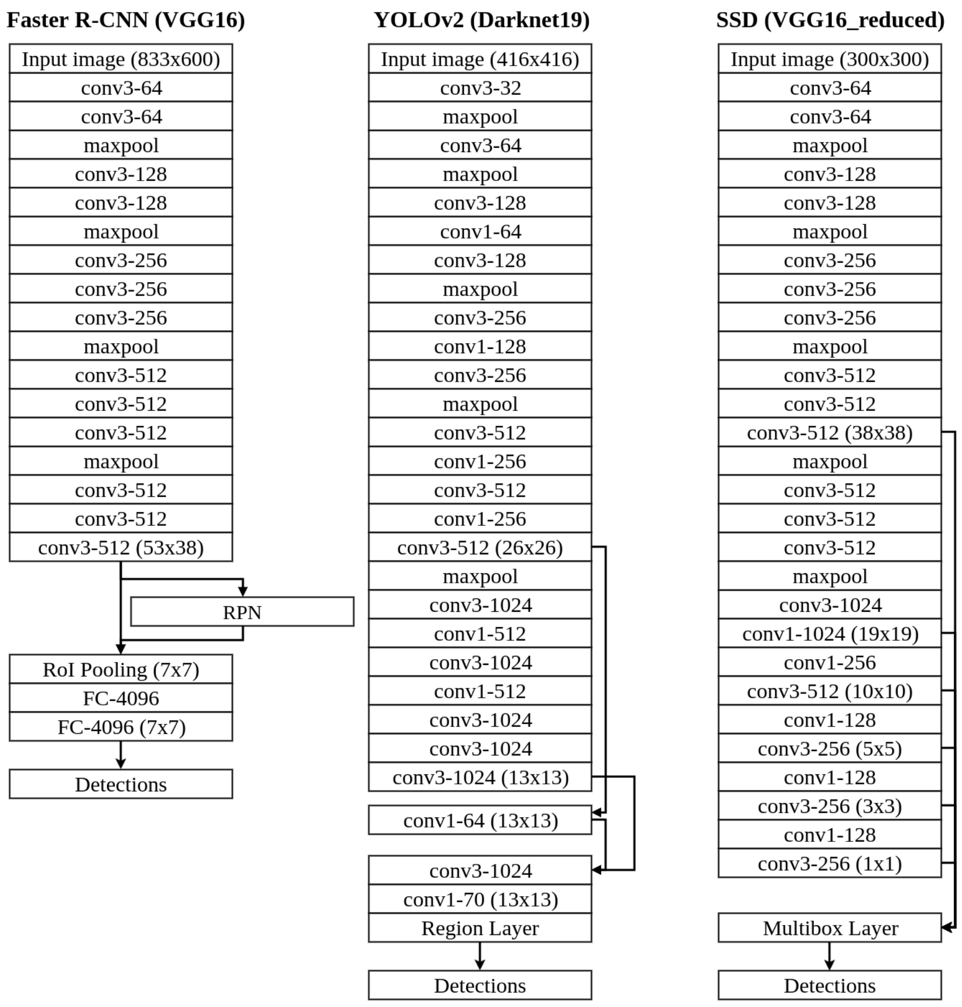


Figure 5.2: Overview of the CNN architectures used by each object detection method in our system. The convolutional layer parameters are denoted as conv[filter size]-[number of filters] and the fully connected layers are denoted as FC-[number of neurons].

6 Implementation

Our unified detection system is based on three open source frameworks used by the object detection methods described in Chapter 2. We refer to these frameworks as Py-faster-rcnn, Caffe-rcnn-ssd and Darknet. This chapter will elaborate on how we implemented the system, along with modifications and adaptations we made to the frameworks that are used.

We implemented high-level functionality in Python, including a run program and three wrapper programs that connect each object detection method to the system. The run program consists of a single main function, which takes care of setting up the configuration and calling appropriate wrapper procedures. Each wrapper acts as an intermediate stage between its respective object detection methods framework and the run program. They have separate procedures for each system action, along with procedures for creating or changing model definition files according to the dataset and configuration settings given. The data utilities and visualization script described in Section 4.1.3, 4.2 and 5.1 was also implemented in Python.

6.1 Open Source Frameworks

Caffe is an open source deep learning framework used by both Faster R-CNN and SSD. It is developed by The Berkeley Vision and Learning Center in C++ with a focus on CNNs, which makes it a popular tool in image analysis [35]. One of its benefits is the number of pre-trained networks that can be downloaded from the Caffe Model Zoo [53] and used immediately.

To train and test a CNN, Caffe uses model and solver definition files (.prototxt), where the CNN architecture, optimizer, and its parameters are specified. It has three interfaces for training and deploying CNNs, including a command line interface and a Python interface which are both used in our system.

Faster R-CNN and SSD originally use their own modified versions of Caffe containing several self-implemented CNN layers, such as the ROI pooling layer and the Multibox layer. Having two versions of Caffe would be very inconvenient, so we decided to merge them and manually solve the conflicts that arose. This resulted in a single Caffe framework that can be used by both Faster R-CNN and SSD in our system, we call it Caffe-rcnn-ssd.

Caffe-rcnn-ssd has the following dependencies:

- Required: BLAS, Boost, protobuf, glog, gflags, hdf5
- Optional: CUDA, cuDNN, OpenCV, lmdb, leveldb

Py-faster-rcnn is the official open source Python implementation of Faster R-CNN [36] and is used in our system. The following modifications and adaptations were made to Py-faster-rcnn:

- The original implementation only supports the PASCAL VOC and Microsoft COCO datasets. We modify it to work with custom datasets that are prepared and structured like Baitcam as shown in Figure 5.1.
- We adopt its YAML configuration scheme and extend it to be used by all the detection methods in our system. The configuration defines several settings, such as input image size, batch size, and options to use custom anchor boxes or OHEM. Adapting it to all methods was natural since they use similar settings.
- We implement the ability to use k custom anchor boxes generated by our k-means clustering utility, instead of using the 9 default anchor boxes for Faster R-CNN models.

- We add the ability to train Faster R-CNN models with OHEM by integrating the original OHEM implementation which was made for Fast R-CNN [50].
- We adapt Faster R-CNN model definition files to the Baitcam dataset and use them in our system.
- We remove everything except the files needed for training, evaluating and detecting with a Faster R-CNN model.
- The functionality to calculate detection metrics, which are described in Section 7.3, is found in Py-faster-rcnn. We use this functionality in the evaluation of all models in our system and extend it to plot and save precision-recall curves with the pyplot module in Matplotlib.

Py-faster-rcnn has the following dependencies:

- Required: CUDA, OpenCV, Numpy, Cython, EasyDict, PyYAML and Matplotlib.

Darknet is an open source neural network framework used by YOLOv2 [37]. It is written by Joseph Redmon in C and CUDA, with a focus on making it fast with support for both CPU and GPU computation [54]. Installing Darknet is easy with only two optional dependencies. It uses CUDA for GPU computation and OpenCV for supporting a wider variety of image types. To train and test a CNN, Darknet uses a single model definition file (.cfg), where the CNN architecture, optimizer, and its parameters are specified.

We use a C++ compilable version of Darknet in our system [55], which supports CUDA v8 and OpenCV v3. The following modifications and adaptations were made to Darknet:

- The original YOLOv2 model definition file is made for the PASCAL VOC dataset. We modify it to work with custom datasets that are prepared and structured like Baitcam as shown in Figure 5.1.
- We implement the ability to use *k* custom anchor boxes generated by our *k*-means clustering utility, instead of using the 5 default anchor boxes for YOLOv2 models.
- We implement functionality in Darknet for our systems detect action, which is described in Section 5.1.
- We remove everything except the files needed for training, evaluating

and detecting with a YOLOv2 model.

Darknet has the following dependencies:

- Optional: CUDA, cuDNN and OpenCV

We use official SSD open source code [39] to create the model definition files for SSD in our system. The following modifications and adaptations were made to this code:

- The original code creates SSD model definition files for the PASCAL VOC datasets. We modify it to create SSD model definition files for custom datasets that are prepared and structured like Baitcam as shown in Figure 5.1.

- SSD does not use a single set of anchor boxes for training, but rather multiple, because it exploits multi-scale feature maps. Originally, the code creates a set of default anchor boxes for each feature map used in the multibox layer, where each set is responsible for differently scaled objects. So the first feature map will be responsible for objects with size 10-20% of the input image, second feature map 20-30% etc. We implement the ability to create sets of anchor boxes based on k custom anchor boxes generated by our k-means clustering utility, instead of creating the default anchor boxes.

7 Evaluation

This chapter describes the experimental setup and detection metrics used to evaluate Faster R-CNN, YOLOv2 and SSD models. All models are trained on the Baitcam training set and evaluated on the Baitcam validation set that is described in Section 4.1.2. We compare the quality of detections of the models and see the effects of our training methods described in Chapter 4. The model's computational complexity is also measured to see what kind of resource allocation is needed both in training and in detection.

7.1 Experimental Platform

All experiments were run on two identical desktop computers with the following specifications: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz×8, Geforce GTX 1080 8GB GPU @ 1657MHz (2560 CUDA cores) and 32GB DDR4 RAM @ 2400MHz. They both ran on Ubuntu 16.04 LTS 64-bit with gcc v5.4.0 compiler and Python v2.7.12.

We build the caffe-rcnn-ssd framework in our system with the following dependencies:

- CUDA v8, cuDNN v5.1, OpenCV v3.2.0-dev, Boost v1.58.0.1, protobuf v2.6.1, glog v0.3.4, gflags v2.1.2, hdf5 v1.8.16 and lmdb v0.9.17

We build the py-faster-rcnn framework in our system with the following dependencies:

- CUDA v8, OpenCV v3.2.0-dev, Numpy v1.12.1, Cython v0.25.2, EasyDict v1.6, PyYAML v3.12 and Matplotlib v1.5.1

We build the darknet framework in our system with the following dependencies:

- CUDA v8, cuDNN v5.1 and OpenCV v3.2.0-dev

7.2 Experimental Design

We decide not to use identical training parameters for each object detection method because of their different training schemes. Instead, we follow each methods default parameters that are optimized for PASCAL VOC datasets and scale them up or down to better fit the Baitcam dataset. Pre-trained ImageNet models were used to initialize all of our models. We train with SGD optimizer using a base learning rate of 0.001, with 0.9 momentum and 0.0005 weight decay, following [16], [17] and [18].

- All Faster R-CNN models were trained for 100000 iterations, scaled up from default 70000 iterations used on the smaller PASCAL VOC 07 dataset. We use default batch size of 2 images and decrease the learning rate by a factor of 10 at 30%, 60%, and 90% iterations.

- All YOLOv2 models were trained for 34250 iterations, scaled down from default 80200 iterations used on the larger PASCAL VOC 07+12 dataset. We use default batch size of 64 images and decrease the learning rate by a factor of 10 at 50% and 75% iterations.

- All SSD models were trained for 50000 iterations, scaled down from default 120000 iterations used on the larger PASCAL VOC 07+12 dataset. We use default batch size of 32 images and decrease the learning rate by a factor of 10 at 66% and 83% iterations.

We train a model for each object detection method with default input image size and default anchor boxes as reference models. Subsequently, we train new models, progressively applying our training methods to see their effects on the results. The different training methods are described in Chapter 4 and can be summarized as: ”default low resolution“ vs. ”custom high resolution“, ”default anchor boxes“ vs. ”custom anchor boxes“ and ”no OHEM“ vs. ”OHEM“. Keep in mind that the ”no OHEM“ vs. ”OHEM“ option only applies to Faster R-CNN models as described in Section 4.4.

For SSD and Faster R-CNN models, we keep the input image size used in training for the evaluation procedure, as in [16] and [18]. For YOLOv2 models, we double the input image size used in training for the evaluation procedure, following [17]. The NMS threshold and confidence threshold for post-processing detections were set to 0.45 and 0.005 respectively. These thresholds were found to give good results on the Baitcam dataset.

Method	Default low resolution	Custom high resolution
Faster R-CNN	833×600	1684×1184
YOLOv2	416×416 (832×832)	832×832 (1184×1184)
SSD	300×300	608×608

Table 7.1: Overview of input image sizes used in our experimentation. Parentheses show size used in the evaluation if it differs from size used in training.

Ten different object detection models is trained and evaluated on the Baitcam dataset in our experimentation. Due to the number of models to be trained and time limitations we were not able to apply k-fold cross validation when training our models. We believe our validation set, as described in Section 4.1.2, gives a good representation of the Baitdam dataset. Post-training, we did detections on new images from the COAT dataset and manually went through them to verify the results, confirming that the models were generalizing well.

To measure the computational complexity of the models we record time usage, RAM usage and GPU memory usage during training and prediction, since these are the primary bottlenecks in CNNs [21]. We compare each object detection method to see their computational differences, and additionally compare all models in our experiments to see the computational cost of applying our training methods.

Time used in training and detection was measured with a timer class in our system. The timer class makes use of Pythons *time* module to measure the total time used between a specified start point and end point in the system. In training, we measure the time it takes for the wrapper to complete its training procedure. In detection, we measure the time it takes to run a forward pass on one image, and take the average of 100 detections. RAM usage during training and detection was recorded with Ubuntu’s system monitor, and GPU memory usage was recorded with the *nvidia-smi* command. We did not want to use profilers that could slow down the system.

7.3 Detection Metrics

We use Mean Average Precision (MAP) to evaluate the object detection models in our system. MAP is a detection metric commonly used in modern object detection challenges such as PASCAL VOC and Microsoft COCO. It is calculated as the mean of the Average Precision (AP) for each class, where AP is given by the area under a precision-recall curve for the detections [19].

Precision is defined as the ratio of True Positive (TP) detections to all detections and captures how accurate the object detection model is.

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

Recall is defined as the ratio of TP detections to ground truth instances and captures how many relevant detections are found by the object detection model.

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

Typically, precision and recall are inversely related, i.e. as recall increases, precision falls and vice-versa. A balance between the two is often preferred and can be found by creating a precision-recall curve.

Creating a precision-recall curve in object detection is done in the following steps:

- Each detection is assigned to the ground truth bounding box which it overlaps the most if any overlaps sufficiently. We follow the standard of PASCAL VOC and consider overlaps with an IOU larger than 0.5 (50%) to be sufficient [19]. IOU divides the area of overlap by the area of union between a pair of bounding boxes $BBox_A$ and $BBox_B$. This yields an overlap ratio between 0 and 1, where 0 indicates that the boxes do not overlap at all, and 1 indicates that the boxes fully overlap.

$$IntersectionOverUnion = \frac{BBox_A \cap BBox_B}{BBox_A \cup BBox_B}$$

- The detection with the highest probability score assigned to each ground truth bounding box is counted as a TP, while all other detections are counted as FPs. This point highlights the importance of post-processing detections with NMS to remove redundant detections, since they would be counted as FPs.

- Precision and recall values are computed for increasingly large subsets of detections, starting with the highest-scored detection and adding the remainder in decreasing order of their score. Plotting these precision-recall pairs as progressively lower-scored detections are included, creates a precision-recall curve as shown in Figure 7.1.

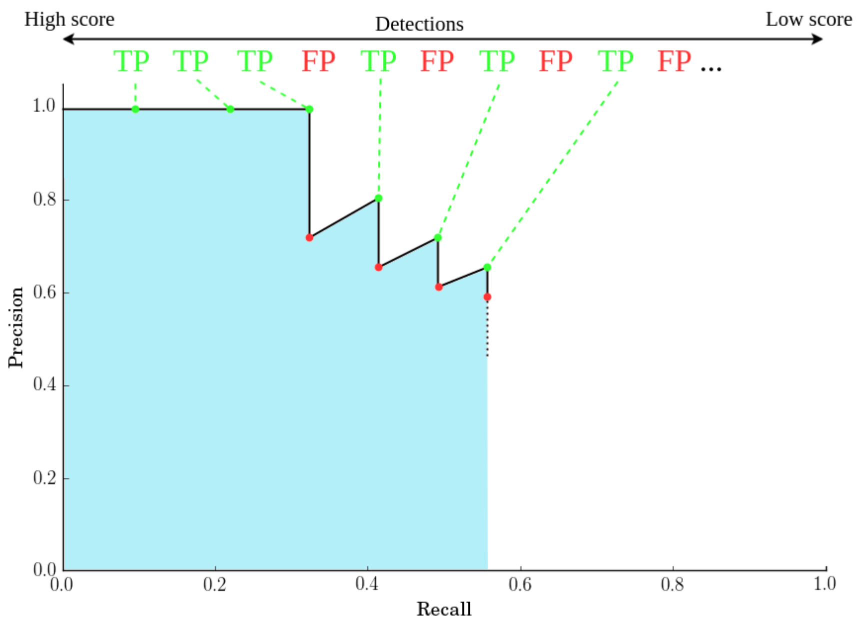


Figure 7.1: Example precision-recall curve for a single class. It shows AP (blue area) and how TP and FP detections change the curve.

The AP for each object class is measured with the Riemann sum as the true area under the curve,

$$AveragePrecision = \sum_{k=1,\ldots,N} P(k)\Delta R(k)$$

where $P(k)$ is the precision at every possible threshold value, $\Delta R(k)$ is the change in recall, and k takes on every possible recall value found in the data. To obtain a high AP score, the object detection model must have precision at all levels of recall, penalizing models that only gets a subset of detections with high precision. MAP can then be computed as the mean of the AP for each class,

$$MeanAveragePrecision = \sum_{q=1}^{Q} \frac{AveragePrecision(q)}{Q}$$

where Q is the number of classes.

7.4 Results

Table 7.2 shows the validation results after training models with default low resolution and default anchor boxes using each object detection method, and Figure 7.2 shows the precision-recall curve of the SSD model in this table.

Method	mAP	(A)Fox	Crow	(WT)Eagle	(G)Eagle	Raven	(R)Fox	Reindeer	SnowyOwl	Wolverine
Faster R-CNN	87.2	97.1	80.2	85.6	95.7	89.0	97.5	46.3	94.3	98.7
YOLOv2	86.4	93.8	78.6	85.5	95.3	88.1	93.6	55.5	90.2	96.9
SSD	87.3	96.8	82.3	88.8	97.1	89.1	97.3	36.2	98.9	99.0

Table 7.2: Results on the Baitcam validation set using each object detection method with default low resolution and default anchor boxes. It shows AP for each class and MAP.

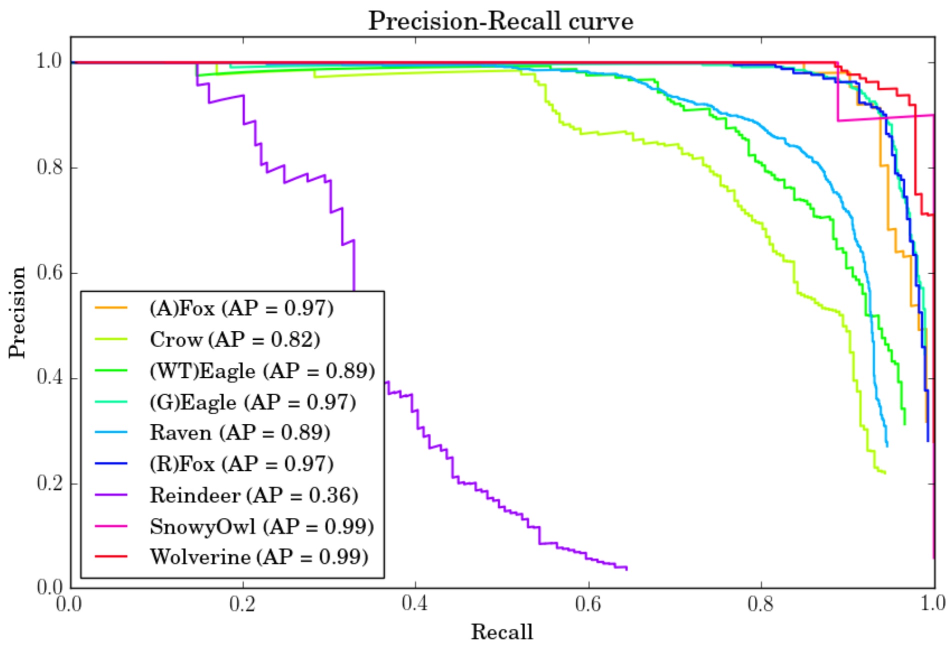


Figure 7.2: Precision-Recall curve for SSD model with default low resolution and default anchor boxes.

From our bounding box annotation, we know that Reindeer, Crow, and Raven are the most small-scale classes in Baitcam, and it is apparent that all the models are having a hard time detecting these animals. With such a low input image size, the smallest objects are scaled down to the degree that they barely appear on the feature maps generated by the CNNs. This makes it hard for the model to train and learn the patterns of these small-scale classes, causing it to make a lot of FP detections as seen with Reindeer on the precision-recall curve in Figure 7.2. The curve implies that the model is more frequently detecting reindeer that are not present or misclassifying other animals to be reindeer. Examples of such detections can be seen in Figure 7.6a.

Table 7.2 show that Reindeer is the hardest class to detect because most reindeer in Baitcam are far in the background of the images. Crows and ravens are also very small-scale, and they have the disadvantage of being similar when it comes to size and features. This means that they have a higher chance of being wrongly detected as each other. As the majority class, Ravens will most likely be favored in detection, causing them to have almost 90% AP while crows have around 80%. The same observation can be made with white-tailed eagles and golden eagles, where the former class is a minority of the two. WhiteTailedEagle has an AP of ~85%, while GoldenEagle has the favorable score at around 95%. This issue is not as noticeable with RedFox and ArcticFox, which are more easily separated by color in daytime images but can be very similar in greyscale images taken at night.

We see that MAP is similar across all methods, with SSD having the highest score of 87.3%, Faster R-CNN having a score of 87.2% and YOLOv2 having the lowest score of 86.4%. This is unexpected since SSD use significantly lower resolution than YOLOv2 and Faster R-CNN in evaluation (300×300 vs. 832×832 vs. 833×600 respectively). The multi-scale training and data augmentation in SSD are clearly beneficial to a majority of the classes in the Baitcam dataset. YOLOv2 performs best on the most small-scale class Reindeer, but worse on all other classes, compared to the other methods. Faster R-CNN is on par with SSD in terms of MAP, without using their extensive data augmentation strategies. With this in mind, a point could be made for keeping the aspect ratio of images when training, to avoid distortions on objects from scaling them to a square shape.

Table 7.3 shows the validation results after training models with default low resolution and custom anchor boxes using each object detection method, and Figure 7.3 shows the precision-recall curve of the SSD model in this table.

Our custom anchor boxes give a modest MAP increase of 0.4% and 1% for Faster R-CNN and YOLOv2 respectively. MAP for SSD, on the other hand, is decreased by 0.3%.

Method	mAP	(A)Fox	Crow	(WT)Eagle	(G)Eagle	Raven	(R)Fox	Reindeer	SnowyOwl	Wolverine
Faster R-CNN	87.6	97.1	80.6	86.1	96.2	90.8	96.9	45.0	97.2	98.8
YOLOv2	87.4	96.9	78.0	88.8	96.2	89.7	96.4	55.2	86.6	99.0
SSD	87.0	95.5	70.6	90.8	97.1	89.1	97.3	46.9	98.9	97.3

Table 7.3: Results on the Baitcam validation set using each object detection method with default low resolution and custom anchor boxes. It shows AP for each class and MAP.

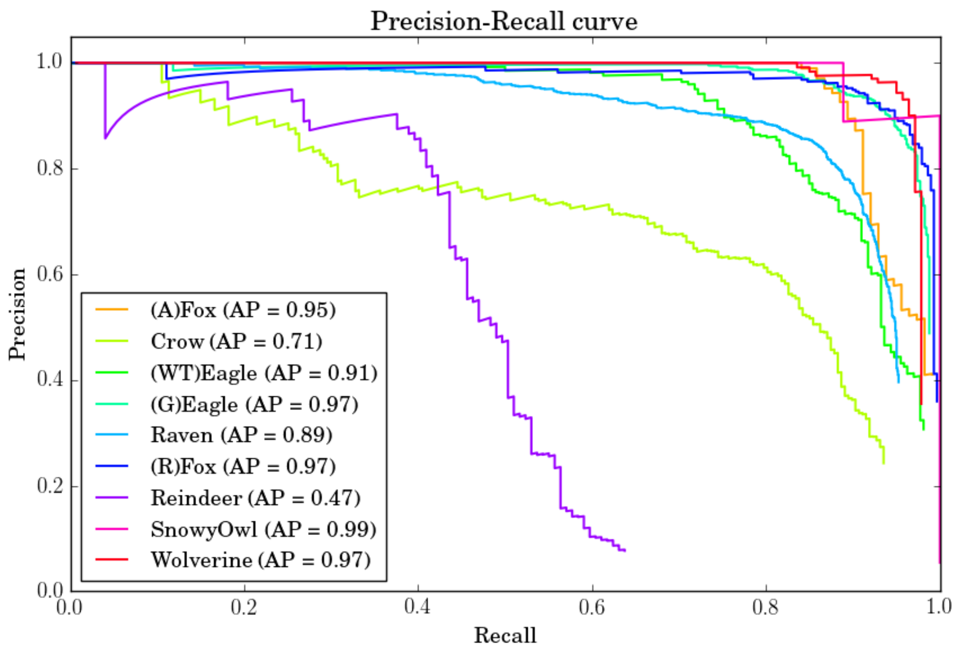


Figure 7.3: Precision-Recall curve for SSD model with default low resolution and custom anchor boxes.

Looking at individual class scores for Faster R-CNN and YOLOv2, we see that AP increases slightly for most classes, compared to Table 7.2. It is clear that our custom anchor boxes have scales and aspect ratios that are more fitting for objects in Baitcam than the default anchor boxes. Detection of reindeer did not improve in these models but got a small AP decrease instead. We know that our custom anchor boxes are much smaller than the default anchor boxes and should be more suitable for the Reindeer class. The model is most likely making more reindeer detections, but these detections are FPs because of the previously mentioned problems with small-scale objects.

For the SSD model, all classes have similar scores to the default reference model, except for Crow and Reindeer. Contrary to the other two object detection methods, our custom anchor boxes in SSD prove effective on the Reindeer class, which has increased from 36.2% to 46.9% AP. Conversely, Crow has decreased by ~11%, which explains the overall MAP decrease. Looking at the precision-recall curve of the SSD model in Figure 7.3, we see that FP detections of crows start occurring at earlier levels of recall, compared to the default reference model in Figure 7.2. Again, the model is making more small-scale detections, but it has a negative effect on the Crow class. We believe it is struggling to separate ravens and crows on such low-resolution input images, therefore detecting more FP crows. The benefits of using custom anchor boxes are clearly not being fully exploited on low-resolution input images. Figure 7.5a, 7.5b, 7.5c and 7.5d show images where our custom anchor boxes help detecting small objects with SSD and YOLOv2.

Table 7.4 shows the validation results after training models with custom high resolution and custom anchor boxes using each object detection method, and Figure 7.4 shows the precision-recall curve of the SSD model in this table. The last Faster R-CNN model in the table is trained with OHEM and bolded numbers highlight our best scores.

Method	mAP	(A)Fox	Crow	(WT)Eagle	(G)Eagle	Raven	(R)Fox	Reindeer	SnowyOwl	Wolverine
Faster R-CNN	91.5	96.3	85.9	89.9	95.3	94.2	97.4	76.5	89.2	98.8
YOLOv2	92.5	96.4	89.0	86.5	97.1	94.2	97.7	80.0	92.6	99.2
SSD	**94.1**	**98.9**	**91.1**	**93.3**	**98.0**	**95.1**	**98.4**	72.6	**100**	**99.5**
Faster R-CNN	93.1	96.9	88.1	89.0	96.1	94.1	97.6	**82.8**	93.6	99.4

Table 7.4: Results on the Baitcam validation set using each object detection method with custom high resolution and custom anchor boxes. It shows AP for each class and MAP. The last Faster R-CNN model in the table is trained with OHEM. Bolded numbers highlight our best scores.

Training with custom high resolution and custom anchor boxes gives a significant boost in MAP for all object detection methods. Compared to Table 7.3, Faster R-CNN increased by 3.9% to 91.5%, YOLOv2 increased by 5.1% to 92.5% and SSD increased by 7.1% to 94.1%. We additionally train Faster R-CNN with OHEM which further increases its MAP by 1.6% to 93.1%. Like the default models, SSD is back to having highest MAP while Faster R-CNN with OHEM and YOLOv2 follows close behind.

It is clear that our training methods are successful for the object detection methods in our system, not only for detecting small objects but also across all classes in the Baitcam dataset. The bolded numbers show that almost all classes have surpassed 90% AP, with many being close to 100%. Reindeer has the lowest AP at 82.8%, but it is the class which has improved the most, as it had only 46.3% AP in the default reference model. The AP gap between crows/ravens and white-tailed eagles/golden eagles has decreased from ~10% to ~4%. This shows that more precise ROIs from our custom anchor boxes and more features from higher input image resolution helps the models separate similar classes, mitigating the issue with majority classes being favorable in

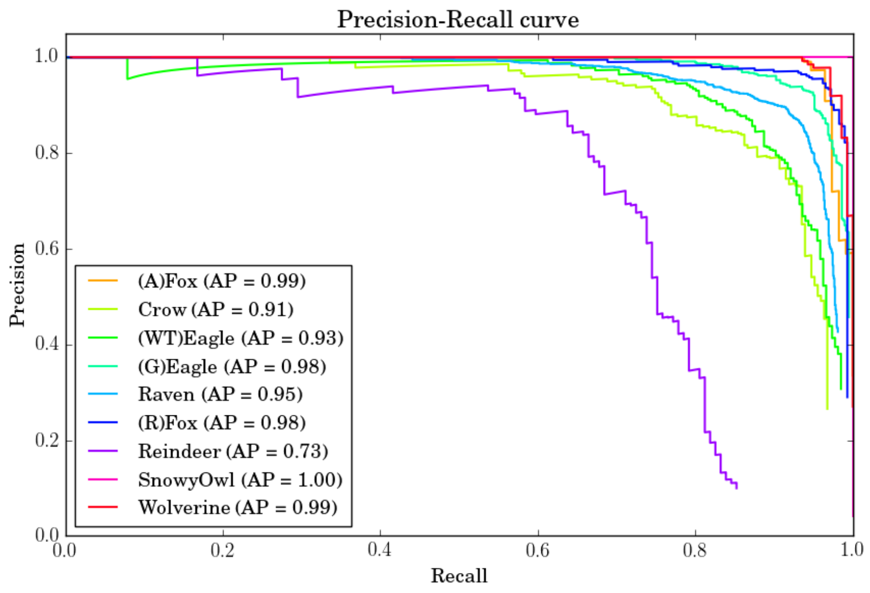
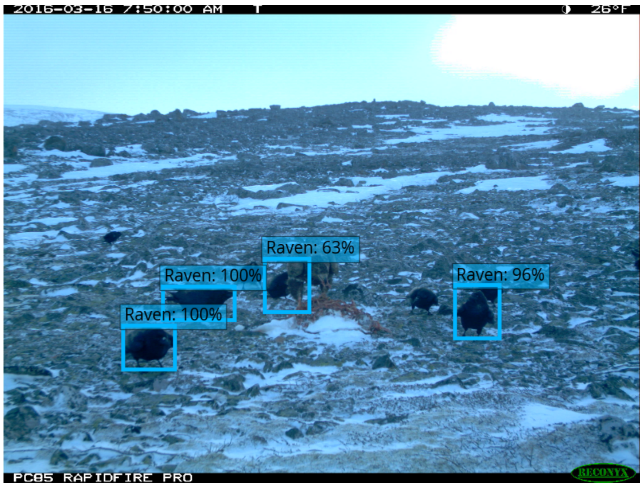


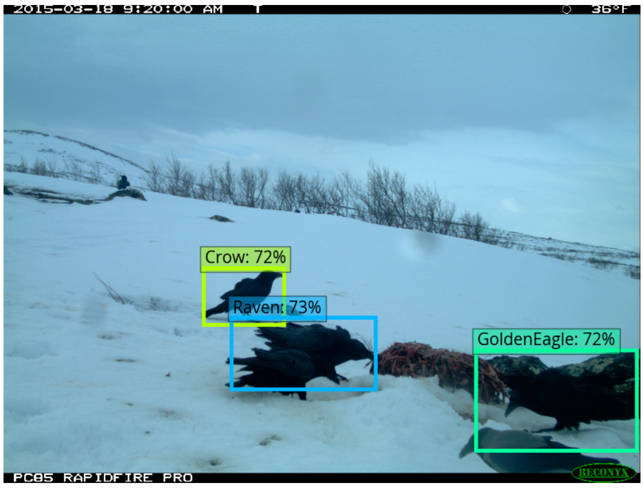
Figure 7.4: Precision-Recall curve for SSD model with custom high resolution and custom anchor boxes.

detection.

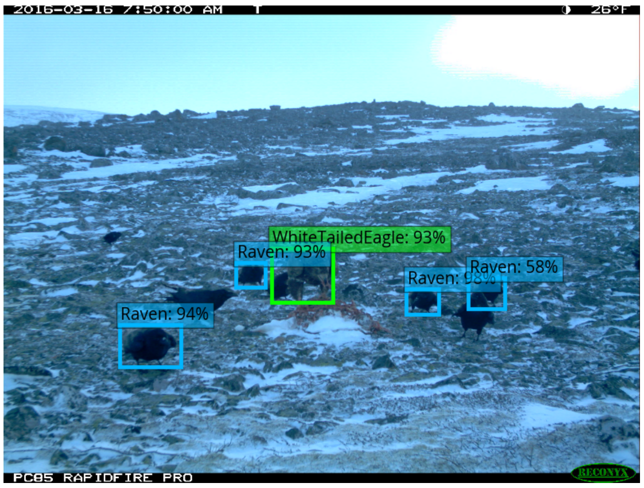
Looking the precision-recall curve for the SSD model in Figure 7.4, we see that the Crow class is in line with other classes, confirming our theory that custom anchor boxes gave more FP Crow detections because of the low-resolution input image. Furthermore, we see a more balanced curve for the Reindeer class where FP detections occur at much later levels of recall. Higher resolution input images combined with custom anchor boxes are clearly beneficial for the most small-scale objects in Baitcam. Figure 7.5e, 7.5f and 7.6b show images with detections made by the models with all our training methods applied, comparing them to the other models in our experiments.



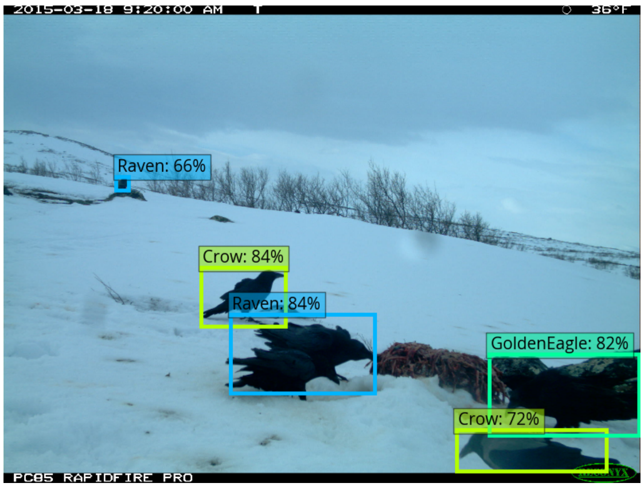
(a) SSD with default low resolution and default anchor boxes.



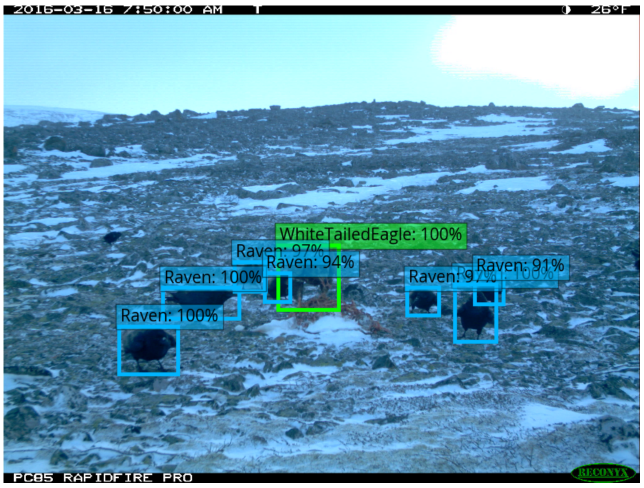
(b) YOLOv2 with default low resolution and default anchor boxes.



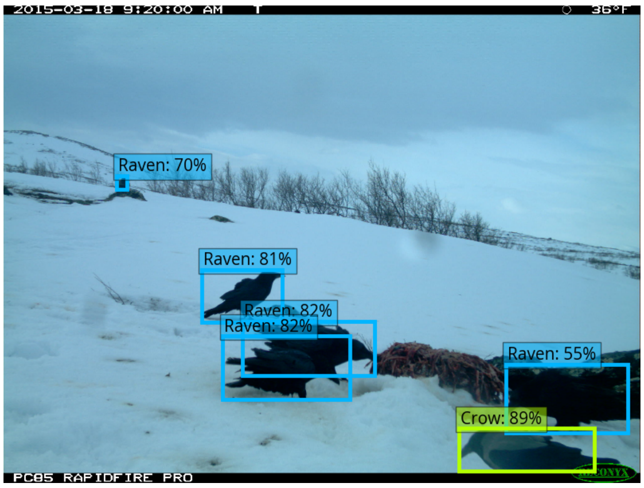
(c) SSD with default low resolution and custom anchor boxes.



(d) YOLOv2 with default low resolution and custom anchor boxes.

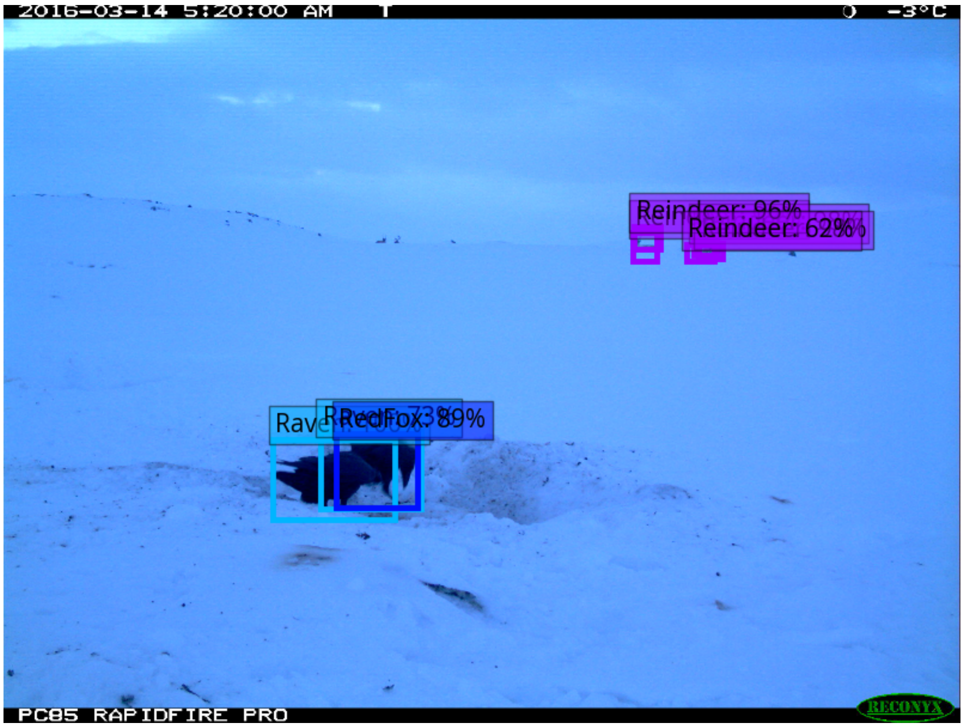


(e) SSD with custom high resolution and custom anchor boxes.

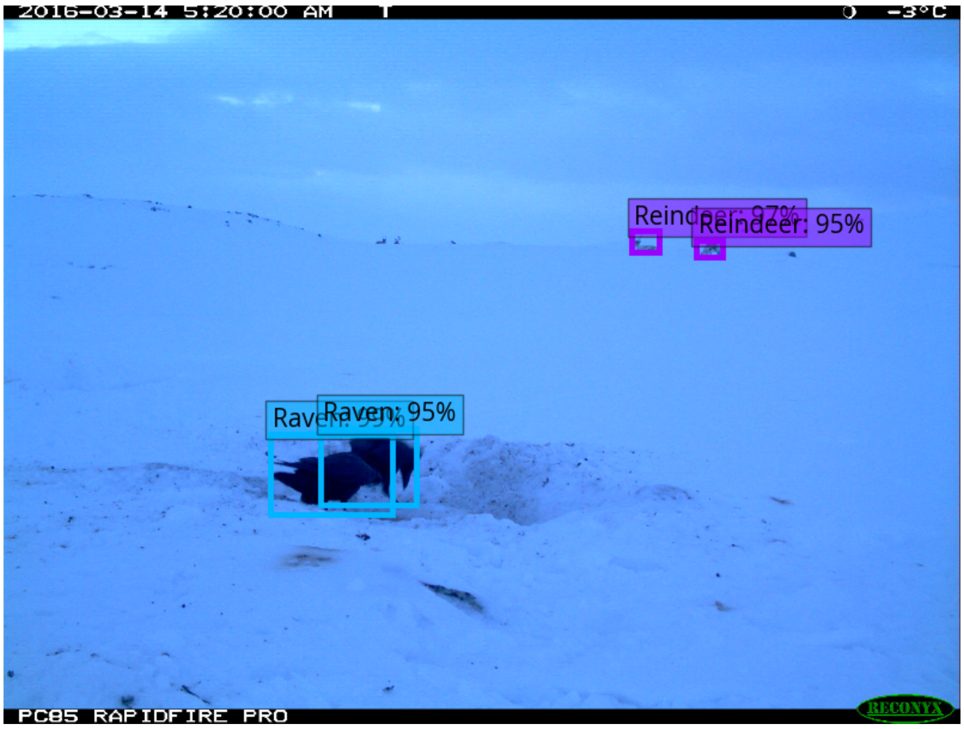


(f) YOLOv2 with custom high resolution and custom anchor boxes.

Figure 7.5: Comparing detections from SSD and YOLOv2 models with different training methods.



(a) Faster R-CNN with default low resolution and default anchor boxes.



(b) Faster R-CNN with custom high resolution, custom anchor boxes and OHEM.

Figure 7.6: Comparison of detections from Faster R-CNN models in our experiments.

Figure 7.7 shows the training time for each object detection model in our experiments. Training the default models is relatively fast, ranging from 10-19 hours. Applying custom anchor boxes did not have a large impact on training time, which is expected because we use the same number of anchor boxes as default and it will therefore not affect the number of operations needed in the model. Using the default models as reference points, we see that doubling the input image size increases training time by approximately 200%, 150% and 300% for Faster R-CNN, YOLOv2 and SSD respectively. Training Faster R-CNN with OHEM increased training time by 350%. So the boost in MAP from using higher resolution images and OHEM does require longer training times, which is a tradeoff that has to be considered when choosing a model for deployment.

Comparing each object detection method, it is evident that Faster R-CNN has the lowest training time in all experiments, which makes sense when we look at the batch size and number of iterations used in training. With a batch size of 2 and 100 000 iterations, Faster R-CNN only trains on a total of 200 000 images, compared to 2 192 000 (64·34250) for YOLOv2 and 1 600 000 (32·50000) for SSD. Even with such large differences in number of images processed in training, all object detection methods has similar performance in MAP, suggesting that the ROI pooling training scheme in Faster R-CNN is more efficient than the full image training schemes in YOLOv2 and SSD.

Figure 7.8 and 7.9 shows the maximum GPU memory and RAM usage when training each object detection model in our experiments. For the default resolution models, SSD has the highest GPU memory usage of around 5GB. This is surprising since it trains on lower image resolution than the other object detection methods, and it has fewer weight parameters as described in Section 5.2. We believe the high GPU memory usage in SSD is due to using multiple feature maps in training and the expansion data augmentation strategy that has a 50% chance of enlarging the input image by 4 times its original size. YOLOv2 and Faster R-CNN has almost identical computational needs of ~3GB and 5.3GB GPU memory for training on low- and high-resolution images respectively.

The lowest RAM usage during training is at around 500-600 MB for all models and is held by SSD, as a contrast to its GPU memory usage. This might be due to the use of LMDBs which lets the Caffe framework load images more efficiently. Faster R-CNN has a stable RAM usage of ~900 MB while YOLO has the highest usage at 1-1.8 GB. The differences when training a YOLOv2 model with low and high resolution indicate that the Darknet framework is less scalable than the Caffe framework in terms of RAM usage.

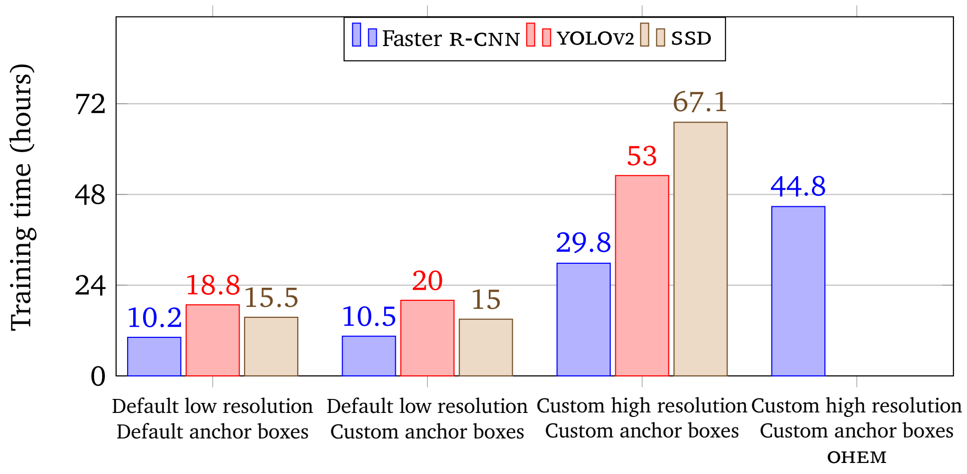


Figure 7.7: Training time (hours) for each object detection model in our experiments.

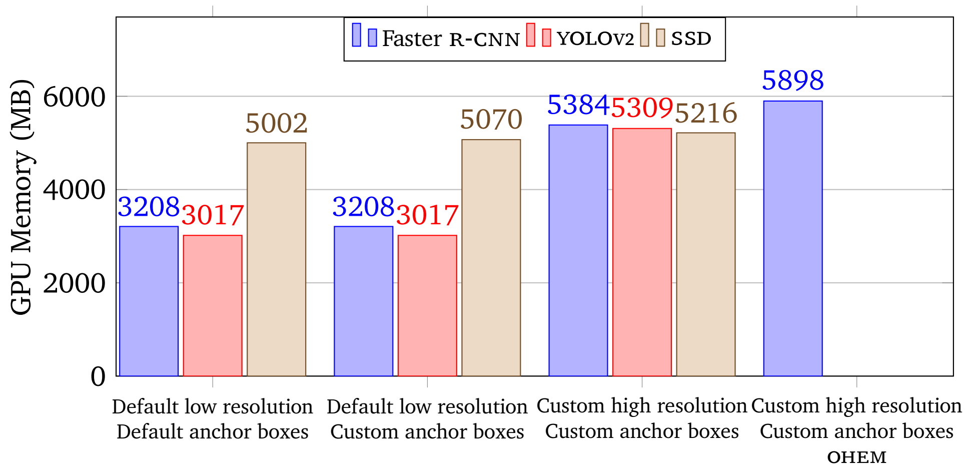


Figure 7.8: Maximum GPU memory usage (MB) when training each object detection model in our experiments.

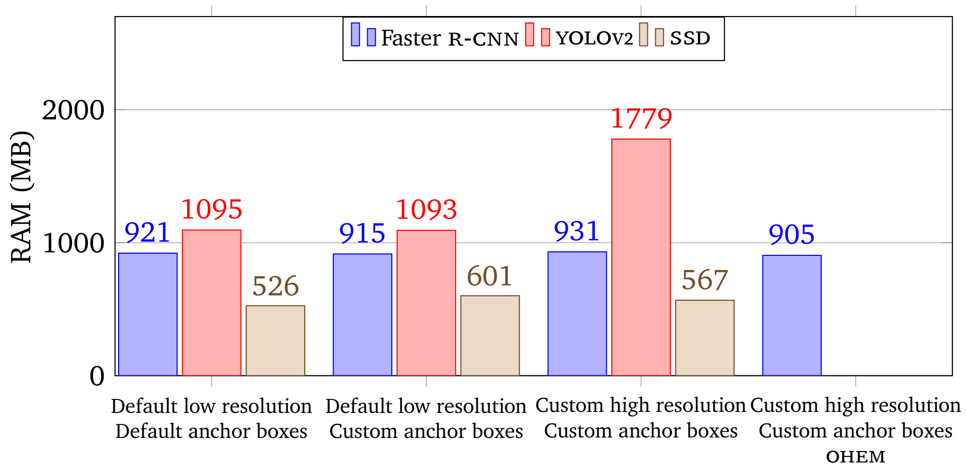


Figure 7.9: Maximum RAM usage (MB) when training each object detection model in our experiments.

Figure 7.10 shows the detection time on one image for each object detection model in our experiments. The default models are faster than the models with our training methods applied, as suggested by the computational needs in training. The discrepancies are more stable than in training time, with a 100% increase in detection time when using low-resolution vs high-resolution input images, except SSD which had a 200% increase. So doubling the input image size leads to double the detection time in most cases.

As indicated in Chapter 2, the single shot detectors have a much lower detection time than Faster R-CNN. Our default SSD model performs detection at 71 frames per second (FPS), with an average detection time of 0.014 seconds per image. YOLOv2 has a lower average detection time of 0.036 seconds and Faster R-CNN is the slowest with 0.136 seconds, corresponding to 27 and 7 FPS. With our training methods applied, the detection speeds are reduced to 21, 15 and 3 FPS.

Figure 7.11 and 7.12 shows the maximum GPU memory and RAM usage when detecting with each object detection model in our experiments. We perform detection with a batch size of 1, resulting in lower computational needs than in training. Similar to Figure 7.10, the YOLOv2 and Faster R-CNN models have almost identical GPU memory usage. In contrary to training, SSD has the lowest GPU memory usage in detection of only ~700-1200 MB. This reinforces our theory of high GPU memory needs from its data augmentation strategy, which is not performed in detection.

RAM usage is also quite different in detection compared to training, especially for YOLOv2 and SSD. Opposite from training, we see that YOLOv2 and SSD has the lowest and highest RAM requirements of around 600 MB and 1000 MB respectively. Faster R-CNN has the same RAM usage as in training at ~900 MB. The RAM usage is stable for all models in detection, even with higher input image resolution.

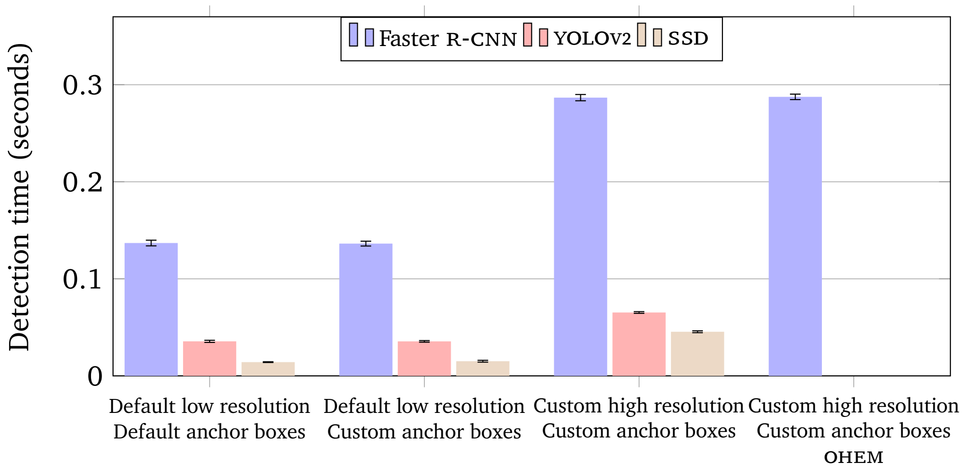


Figure 7.10: Detection time (seconds) on one image for each object detection model in our experiments. Measured as an average of 100 detections with error bars showing standard deviation.

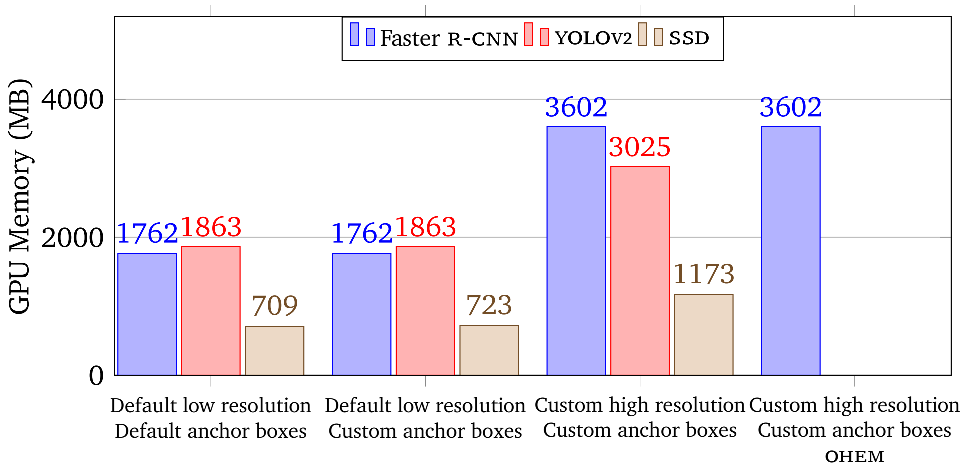


Figure 7.11: Maximum GPU memory usage (MB) when detecting with each object detection model in our experiments.

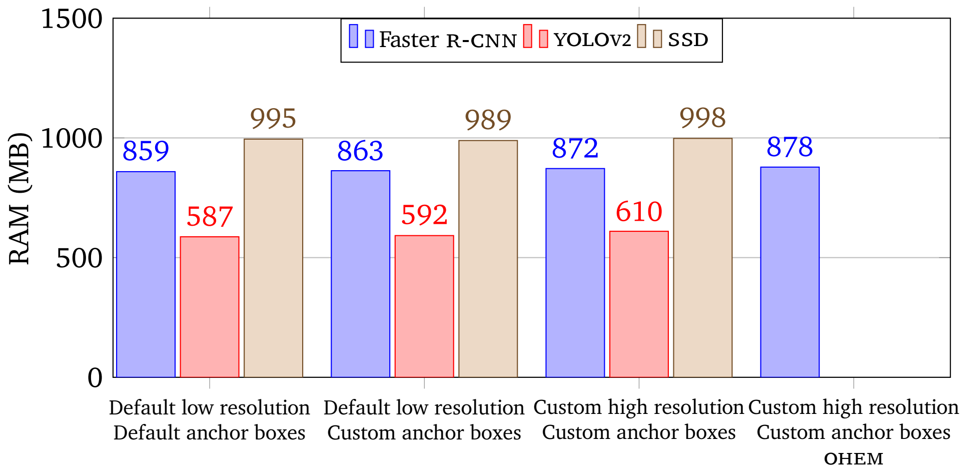


Figure 7.12: Maximum RAM usage (MB) when detecting with each object detection model in our experiments.

8 Discussion

Our evaluation compared state-of-the-art object detection methods in our unified detection system and showed the challenge of detecting wild animals in camera trap images from the Arctic tundra. We worked to solve this challenge by training several object detection models, investigating their results and figuring out how to make improvements, in an iterative fashion. This chapter discusses the process of deriving our training methods and applying them to our experimentation. The results showed that our training methods were successful, with certain tradeoffs to consider, which are also discussed in relation to the COAT project.

8.1 Detecting Small Animals

The published papers on the object detection methods used in our unified detection system recognize the difficulty of detecting small objects in images and work toward improving this problem [17] [18]. After training our first model on the Baitcam dataset, it was clear that this would be our main challenge as well. An easy fix would be to remove the most challenging images with very small-scale objects from the Baitcam dataset, only demanding reliable detection of animals that are close to the camera. We did not choose this approach because it would remove a large portion of images from the already small Baitcam dataset, and we want an animal detection model that is reliable for all cases of animals in the Arctic tundra.

From our research, we knew that anchor boxes played a big part in how well the model would detect certain objects and was a common factor for all object detection methods in our system. Since Baitcam had nine different animals, and several of those animals are similar in size, we believed that custom anchor boxes which were optimized for the animals in Baitcam would surely improve detection accuracy. An option was to manually explore the bounding box annotation data in our Baitcam training set and hard-code anchor box changes to the object detection methods. We chose another method because we wanted a more automatic and general solution, which could potentially be used to find favorable anchor boxes for other datasets and problem areas as well. This goal, combined with some guidance from Redmon et. al. [17], lead to our k-means clustering implementation for finding custom anchor boxes.

Applying our custom anchor boxes proved successful in making more small-scaled detections, but also had an unexpected negative effect because of the low input image resolution used in each object detection method. All object detection methods default settings were made for Pascal VOC datasets which consist of ~500×375 images with a wide range of object scales. The images in Baitcam have a much larger size of 2043×1472, a majority of our objects covers below 30% of the full image and our smallest object covers only ~1%. It was clear that the object detection methods needed adapting to our larger images.

Our first approach was to exploit larger feature maps from earlier layers in the CNN, since small-scale objects would have more features retained in these layers for detection. We modified the Faster R-CNN model definition files and moved the RPN from the 17th layer to 13th layer in the VGG16 network. This resulted in a 1-2% MAP increase, at the cost of more GPU memory usage and longer training times. For YOLOv2, an idea is to remove one of the initial max-pooling layers making the final feature map 26×26, but we were not able to experiment with this due to time limitations.

We tried increasing the input image size in addition to using a larger feature map in Faster R-CNN, but our GPU memory fell short of the CNNs requirements. Taking a step back, we decided to revert the RPN to the 17th layer and train solely with a higher resolution input image size and our custom anchor boxes. This training method increased MAP significantly more than using larger feature maps, as seen in our results. Consequently, we applied it to all object detection methods in our experimentation.

Shrivastava et. al. implemented OHEM for the Fast R-CNN object detection method and showed that it was helpful when dealing with smaller sized objects [49]. Additionally, our results showed that the SSD model, which uses OHEM, had the best MAP when training with custom high resolution and custom anchor boxes, while Faster R-CNN was falling behind. Based on this information we decided to integrate OHEM from Fast R-CNN into our unified detection system and apply it to Faster R-CNN as our final training method. This yielded a better performing Faster R-CNN model in terms of detection accuracy, surpassing our best YOLOv2 model and achieving best AP for the small-scale Reindeer class.

8.2 Model Deployment for Animal Detection in the Arctic Tundra

Our evaluation show that our training methods are successful for Faster R-CNN, YOLOv2 and SSD, giving a MAP increase on the Baitcam validation set by 5.9%, 6.1% and 6.8% respectively. This increase in quality of detection comes at the cost of higher computational complexity and hardware requirements for our detection system. All the models in our experiments have different computational attributes which can deem them favorable or unfavorable for deployment in the COAT project.

Based on the GPU memory and RAM requirements during training of all models, we believe they are best suited for training at a server backend and not on-site at the camera traps, where hardware is limited. With this in mind, we do not see GPU memory, RAM and time usage during training to be an obstacle, since retraining of models could be done continuously at the backend. Additionally, we believe retraining due to new animals is unlikely, as nine animal classes in the COAT dataset has been constant over the last six years of capturing images in Finnmark, Norway.

GPU memory and RAM usage of our models during detection give an indication of possible deployment on-site at the COAT camera traps. We were particularly surprised by the SSD models which only require 1.2 GB GPU memory and 1 GB RAM for detection. In the case that these hardware requirements are too high, there are other possibilities to consider. The camera traps could transmit images to a server where our unified detection system would perform detection on said images.

One of COATs goals is to create a real-time detection system. To achieve this, the models in our system has to be able to perform detection on an image at the same or faster rate that the camera traps capture images. In our case, the camera traps take an image every 5 minutes, which means that all our models surpass this goal by a large margin. The sampling rate of the camera traps could be increased by a factor of 300 and our slowest model would still be able to perform real-time detection. Based on all previously mentioned points, we believe our SSD model with the highest MAP of 94.1% is suitable for deployment in the COAT project, despite its computational complexity.

9 Conclusion

In this thesis, we have implemented a system that unifies three state-of-the-art object detection methods for automatic, real-time and accurate animal detection in camera trap images from the Arctic tundra. We have given a detailed description of dataset preparation and training methods used for animal detection. We described and analyzed Faster R-CNN, YOLOv2 and SSD, comparing their quality of detections and computational complexity on our Baitcam dataset. Our results show that SSD outperforms the other methods in both detection accuracy and speed.

Our experimentation showed that detection of small objects in images is one of the main challenges of animal detection in the Arctic tundra. We worked towards solving this challenge by deriving and applying our training methods, achieving significant improvements. Our fastest model can detect scavengers in the Arctic tundra at 71 FPS with 87.3% MAP. With our training methods, we achieve 94.1% MAP at 21 FPS. We additionally discussed the process of deriving our training methods and possible deployment of our unified detection system for animal detection in the Arctic tundra.

9.1 Future Work

Several improvements can be made to our unified detection system. A goal for the future is to port YOLOv2 to Caffe, removing the need for Darknet and simplifying our system. This could be done by implementing a few CNN layers used by YOLOv2 in Caffe, such as the region layer. For further simplification, we want to remove the use of Caffe's command line interface, making our system only rely on Caffe's Python interface instead.

Caffe2 was newly released as a lightweight and scalable deep learning framework, building on the original Caffe, A long-term goal is to port the object detection methods to the Caffe2 framework, making it the sole dependency of our unified detection system.

Adding more object detection methods is a natural development of our system, and should be relatively straightforward by creating additional wrapper programs. A few methods we are interested in is the Region-based Fully Convolutional Network (R-FCN) [56] and the Deconvolutional Single Shot Detector (DSSD) [57]. Both have shown state-of-the-art detection accuracy on the PASCAL VOC datasets and the latter was particularly impressive on small objects.

To further increase MAP for animal detection in camera trap images from the Arctic tundra, we propose experimenting with different CNN architectures for the object detection methods. We believe networks such as ResNet101 [41] and the Inception ResNet v2 [58] would improve detection accuracy compared to the VGG16 networks which are currently used, but they would require better hardware than what was used for the experimentation in this thesis. For on-site camera trap deployment we would like to try MobileNet which has been shown to achieve VGG16 level accuracy on ImageNet with only 1/30 of the computational cost and model size [59].

Because of time limitations, we had to quickly implement custom anchor boxes for SSD and we believe further improvements can be made with more experimentation. Additionally, we would like to add cropping as a preprocessing step, since we know that animals in Baitcam rarely exceeds 30% of the image size. Cropping the input images into smaller patches and training on the patches would remove the need for our high-resolution input image size, decreasing the models computational complexity. It would most likely improve detection on smaller objects as well, since the cropped patches would not need substantial downscaling like the original images.

Bibliography

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

[2] K. K. A.F. O'Connell, J.D. Nichols, *Camera traps in animal ecology: Methods and analyses*. Springer, Jan. 2011.

[3] R. Steenweg, M. Hebblewhite, R. Kays, J. Ahumada, J. T. Fisher, C. Burton, S. E. Townsend, C. Carbone, J. M. Rowcliffe, J. Whittington, J. Brodie, J. A. Royle, A. Switalski, A. P. Clevenger, N. Heim, and L. N. Rich, "Scaling-up camera traps: monitoring the planet's biodiversity with networks of remote sensors," *Frontiers in Ecology and the Environment*, vol. 15, no. 1, pp. 26–34, 2017.

[4] R. Kays, B. Kranstauber, P. A. Jansen, C. Carbone, M. J. Rowcliffe, T. Fountain, and S. Tilak, "Camera traps as sensor networks for monitoring animal communities," *LCN*, p. 811–818, Oct. 2009.

[5] R. A. Ims, J. U. Jepsen, A. Stien, and N. G. Yoccoz, "Science plan for coat: Climate-ecological observatory for arctic tundra," *Fram Centre Report Series 1*, p. 98, 2013.

[6] S. Hamel, S. T. Killengreen, J.-A. Henden, N. E. Eide, L. Roed-Eriksen, R. A. Ims, and N. G. Yoccoz, "Towards good practice guidance in using camera-traps in ecology: influence of sampling design on validity of ecological inferences," *Methods in Ecology and Evolution*, vol. 4, no. 2, pp. 105–113, 2013.

[7] O. Beijbom, C. J. Tan, S. Chan, T. Treibitz, A. Gamst, B. G. Mitchell, D. Kriegman, P. J. Edmunds, C. Roelfsema, J. Smith, D. I. Kline, B. P. Neal, M. J. Dunlap, V. Moriarty, and T. Y. Fan, "Towards automated annotation of benthic survey images: Variability of human experts and operational modes of automation.," *PLoS One*, vol. 10, p. e0130312, 2015.

[8] A. Swanson, M. Kosmala, C. Lintott, R. Simpson, A. Smith, and C. Packer, "Snapshot serengeti, high-frequency annotated camera trap images of 40 mammalian species in an african savanna," *Scientific data*, vol. 2, June 2015.

[9] J. S. Evans, *Bias in human reasoning - causes and consequences.* Essays in cognitive psychology, Lawrence Erlbaum, 1989.

[10] W. P. Colquhoun, "The effect of a short rest-pause on inspection efficiency," *Ergonomics*, vol. 2, no. 4, pp. 367–372, 1959.

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

[12] N. S. Clayton and N. J. Emery, "The social life of corvids," *Current Biology*, vol. 17, no. 16, pp. R652 – R656, 2007.

[13] T. Y.-H. Chen, L. S. Ravindranath, S. Deng, P. V. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *13th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, (Seoul, South Korea), Nov. 2015.

[14] P. Tsarouchi, S.-A. Matthaiakis, G. Michalos, S. Makris, and G. Chryssolouris, "A method for detection of randomly placed objects for robotic handling," {*CIRP*} *Journal of Manufacturing Science and Technology*, vol. 14, pp. 20 – 27, 2016.

[15] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016.

[16] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems 28* (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), pp. 91–99, Curran Associates, Inc., 2015.

[17] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016.

[18] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, "SSD: single shot multibox detector," *CoRR*, vol. abs/1512.02325, 2015.

[19] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, 2010.

[20] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014.

[21] 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.

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

[23] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of International Computer Vision and Pattern Recognition (CVPR 2014)*, 2014.

[24] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop.," in *Neural Networks: Tricks of the Trade (2nd ed.)* (G. Montavon, G. B. Orr, and K.-R. Müller, eds.), vol. 7700 of *Lecture Notes in Computer Science*, pp. 9–48, Springer, 2012.

[25] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning, 2012.

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

[27] D. M. Hawkins, "The Problem of Overfitting," *Journal of Chemical Information and Computer Sciences*, vol. 44, no. 1, pp. 1–12, 2004.

[28] S. J. Nowlan and G. E. Hinton, "Simplifying neural networks by soft weight-sharing," *Neural Comput.*, vol. 4, pp. 473–493, July 1992.

[29] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, Jan. 2014.

[30] 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.

[31] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” *CoRR*, vol. abs/1311.2524, 2013.

[32] J. Uijlings, K. van de Sande, T. Gevers, and A. Smeulders, “Selective search for object recognition,” *International Journal of Computer Vision*, 2013.

[33] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, “Object detection with discriminatively trained part-based models,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, pp. 1627–1645, Sept. 2010.

[34] R. B. Girshick, “Fast R-CNN,” *CoRR*, vol. abs/1504.08083, 2015.

[35] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv preprint arXiv:1408.5093*, 2014.

[36] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks.” `https://github.com/rbgirshick/py-faster-rcnn`, 2015. (Date last accessed 18-May-2017).

[37] J. Redmon, “Darknet: Open source neural networks in c.” `https://github.com/pjreddie/darknet`, 2016. (Date last accessed 18-May-2017).

[38] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” *CoRR*, vol. abs/1506.02640, 2015.

[39] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, “Ssd: Single shot multibox detector official open source code.” `https://github.com/weiliu89/caffe/blob/ssd/examples/ssd/ssd_pascal.py`, 2016. (Date last accessed 18-May-2017).

[40] 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.

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

[42] J. Wawerla, S. Marshall, G. Mori, K. Rothley, and P. Sabzmeydani,

"Bearcam: automated wildlife monitoring at the arctic circle," *Mach. Vis. Appl.*, vol. 20, no. 5, pp. 303–317, 2009.

[43] P. Sabzmeydani and G. Mori, "Detecting pedestrians by learning shapelet features.," in *CVPR*, IEEE Computer Society, 2007.

[44] J. Parham and C. Stewart, "Detecting plains and grevy's zebras in the realworld," in *2016 IEEE Winter Applications of Computer Vision Workshops (WACVW)*, pp. 1–9, Mar. 2016.

[45] Z. Zhang, Z. He, G. Cao, and W. Cao, "Animal detection from highly cluttered natural scenes using spatiotemporal object region proposals and patch verification," *IEEE Transactions on Multimedia*, vol. 18, pp. 2079–2092, Oct. 2016.

[46] Tzutalin, "Labelimg." `https://github.com/tzutalin`, 2016. (Date last accessed 18-May-2017).

[47] M. Everingham and J. Winn, "The pascal visual object classes challenge 2012 (voc2012) annotation guidelines.." `http://host.robots.ox.ac.uk/pascal/VOC/voc2012/guidelines.html`, 2012. (Date last accessed 18-May-2017).

[48] E. Keogh and A. Mueen, *Curse of Dimensionality*, pp. 257–258. Boston, MA: Springer US, 2010.

[49] A. Shrivastava, A. Gupta, and R. B. Girshick, "Training region-based object detectors with online hard example mining," *CoRR*, vol. abs/1604.03540, 2016.

[50] A. Shrivastava, A. Gupta, and R. B. Girshick, "Training region-based object detectors with online hard example mining." `https://github.com/abhi2610/OHEM`, 2016. (Date last accessed 18-May-2017).

[51] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[52] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," *CoRR*, vol. abs/1411.4038, 2014.

[53] Y. Jia and E. Shelhamer, "Caffe model zoo." `https://github.com/BVLC/caffe/wiki/Model-Zoo`, 2017. (Date last accessed 18-May-2017).

[54] J. Redmon, "Darknet: Open source neural networks in c." `http://`

pjreddie.com/darknet/, 2013–2016.

[55] J. Redmon and P. Sundareson, "Darknet-cpp." https://github.com/prabindh/darknet, 2016. (Date last accessed 18-May-2017).

[56] J. Dai, Y. Li, K. He, and J. Sun, "R-FCN: object detection via region-based fully convolutional networks," *CoRR*, vol. abs/1605.06409, 2016.

[57] C. Fu, W. Liu, A. Ranga, A. Tyagi, and A. C. Berg, "DSSD : Deconvolutional single shot detector," *CoRR*, vol. abs/1701.06659, 2017.

[58] C. Szegedy, S. Ioffe, and V. Vanhoucke, "Inception-v4, inception-resnet and the impact of residual connections on learning," *CoRR*, vol. abs/1602.07261, 2016.

[59] 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/11704.04861, 2017.