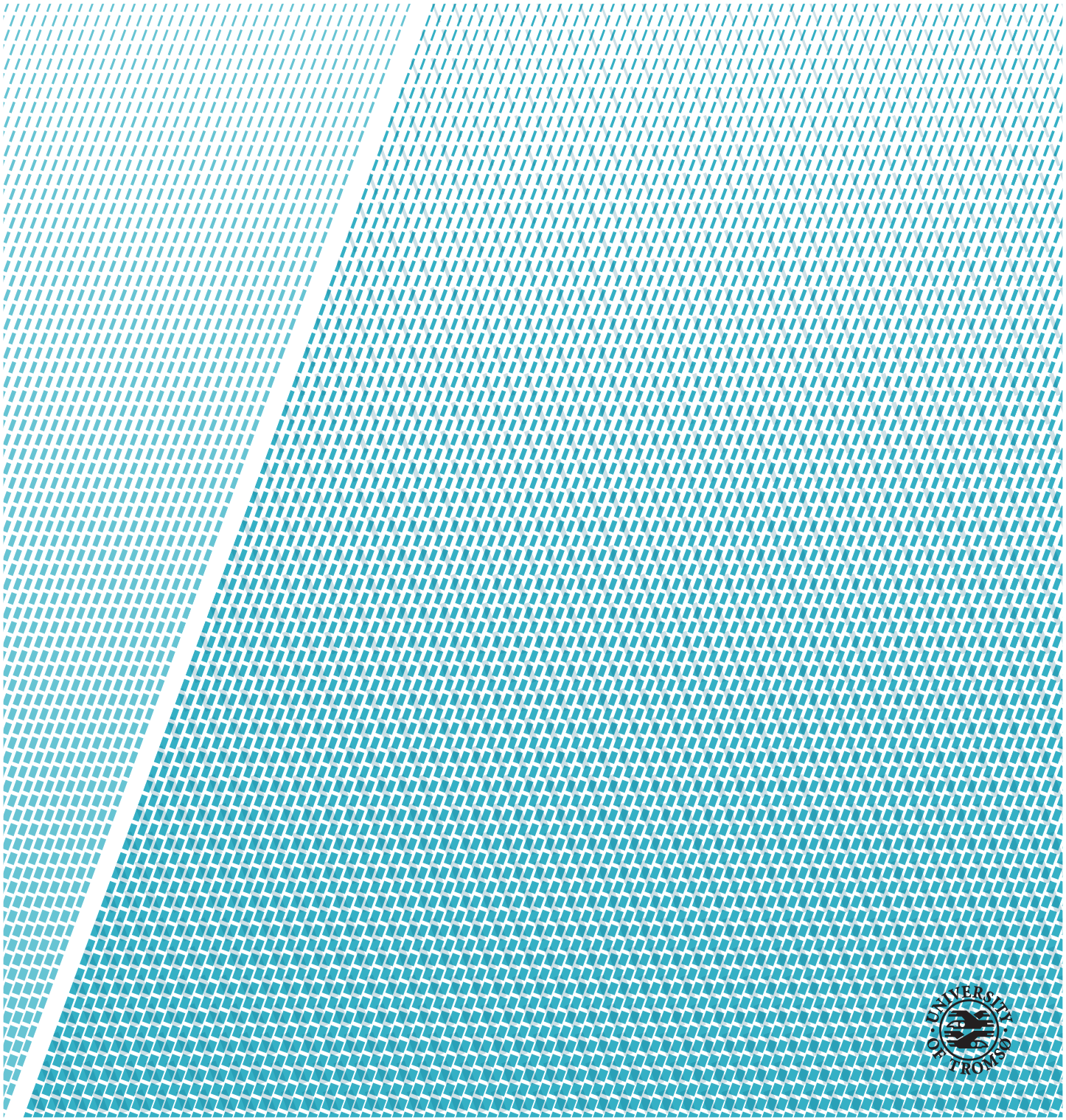
UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Automatic Rodent Identification in Camera Trap Images using Deep Convolutional Neural Networks

Håvard Thom

INF-3983 Capstone Project in Computer Science . . . December 2016



Abstract

The Climate-ecological Observatory for Arctic Tundra (COAT) project is currently monitoring rodent populations using camera traps in northern territories. These camera traps produce hundreds of thousands of images, which have to be manually examined by ecologists to identify the animal species in the image. A more efficient approach to identifying the species is required for further advancement in their research. This paper presents an animal species identification system that can automatically identify small mammals in camera trap images. The system can train three different deep convolutional neural network models which can be used to make image predictions.

The design and implementation of our system and the models are described, along with methods used to improve the prediction results for our dataset. We evaluate and compare the quality of predictions of each model and look at their computational complexity to assess possible deployment of the system at a remote camera trap. Results show that we achieve 97.84% accuracy, 97.81% precision and 93.45% recall on a dataset with 10000 camera trap images and 11 classes, exceeding the performance of previous related work. Furthermore, it shows that establishing real time identification at remote camera traps could be difficult due to high computational costs of convolutional neural networks.

Contents

List of Figures

List of Tables

1

Introduction

Motivation

With the climate changes occuring 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 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 enables documentation and understanding of climate impacts on arctic tundra ecosystems [1].

In autumn 2015, COAT was granted substantial funding which will allow them to establish a research infrastructure during 2016-2020. Part of this infrastructure is to create a real time animal detection system and we will present the first steps towards reaching this goal.

Camera Trapping

Camera trapping is one of the most cost-effective methods for collecting data on animal populations. The remotely activated cameras are equipped 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 [2].

A new camera trap prototype was recently developed by COAT to measure the quantity of small mammal activity under the snow. Since rodents are a key-stone species in northern terrestrial food webs, tracking and understanding their population fluctuations is very important, especially during the winter [3]. Warmer winters could cause irregularities in rodent population cycles which would negatively impact plant communities because of reduced rodent grazing and rodent specialist predators such as weasels [4].

These rodent camera traps produces huge amounts of monitoring data, which poses challenges in both data management and data analyzing. Currently the images are manually examined and annotated by ecologists, which is an extremely tedious approach. With hundreds of thousands of images, it would take weeks, maybe even months to go through. There is no doubt that this work force could be more useful elsewhere. The demand for a tool that can help automatically identify animal species from large volumes of images is clearly present in both the COAT project and the ecology research community in general.

Problem Definition

In this paper, we consider the problem of automatic identification of animal species from camera trap images using deep convolutional neural networks (CNN), which has previously shown promising results in this field [5]. We state that it is possible to create a fully automatic animal identification system that achieves near-human performance by using state of the art technology. To define near-human performance we look at [6] who showed that the human error rate was estimated at 5.1%, based on experiments with an expert annotator that trained on 500 images and annotated 1500 test images from the ImageNet dataset [6]. This kind of performance has, to the author's knowledge, not previously been achieved in animal identification on camera trap images.

To test our statement we present the design and implementation of a prototype system that identifies small mammals in camera trap images. The system has three different deep convolutional network models, which can be trained and used to make image predictions. A *bottleneck* model and *fine-tune* model that are based on a pretrained VGG-16 net [7], and a *custom* model that is trained from scratch. We will describe the methods used to optimize these models for our dataset, which is provided by the COAT project and their camera trap prototypes. In our evaluation, we will look at the computational complexity of our models and their quality of predictions on the following classifications:

- A binary classification to detect if the image contains an animal or not.
- A multiclass classification of small mammal families and the state of the camera trap (i.e. if the camera trap tunnel is filled with snow or water).
- A multiclass classification of specific small mammal species and the state of the camera trap.

Contributions

The contributions of this project are:

- A description of state of the art techniques used for image recognition.
- A working prototype system for automatic identification of small mammals in camera trap images. The system is also generally applicable to different datasets with other types of classifications.
- An evaluation of the system comparing the quality of predictions and computational performance of three different deep convolutional neural network models.

2

Background and Related Work

Over the past decade, image recognition and object detection has become an increasingly popular field in computer science. Partly due to advances in technology and the need for improvements on applications such as facial recognition or fingerprint detection. also due to the rise of communities such as kaggle [1], which host and promotes public machine learning competitions and provide a platform for data scientists to share and collaborate on their work.

There are several techniques used for image recognition with the traditional choice of image representation being feature extraction methods such as Bag-of-Visual-Words (BOVW) [8] and Improved Fisher Vector (IFV) [9]. However, in recent years these relatively simple methods have started to be outperformed by deep CNN [10][11] which have a more complex structure containing several layers of non-linear feature extractors.

1. kaggle.com

Bag-of-Visual-Words

The BoVW model is a standard approach in image classification inspired by the Bag-of-Words model used in document classification. The model starts by extracting local image patch descriptors, also known as *visual words*, which describes important key-points in the image. These *visual words* are then mapped to a high dimensional feature vector and pooled into a global image-level signature. Lastly, a classifier is used to train or predict on the global image signatures.

The choice of image descriptors for the *visual words* is important, and it might vary with different vision tasks. A frequently used algorithm is the Scale Invariant Feature Transform (SIFT) [12] which detect local points of interest using a histogram of gradients of the image. This descriptor generally works well when scale, rotation or illumination change happens. The Local Binary Pattern (LBP) [13] is another algorithm that describes the appearance of small-scale pixel neighborhoods in an image and it has proven to be powerful for texture classification.

There are some issues with BoVW such as computation efficiency when generating *visual words* and how to account for spatial information in the image. Several methods has overcome these problems by extending the BoVW approach and as a result, demonstrated impressive levels of performance, exceeding the state of the art on public datasets [14] [15].

Convolutional Neural Networks

Instead of using the hand-designed feature extraction methods as in BoVW, convolutional neural networks stack multiple stages, or layers of feature extractors in a connected structure with a classification layer at the end. These layers form a complete *deep* CNN architecture, as opposed to the traditional methods *shallow* architecture.

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

- **Feedforward:** 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:** A *cost function* such as the cross-

entropy loss calculates an error measurement between the true training labels and the predicted labels. This error measurement is then used by an *optimizer* that goes backwards through the network tweaking the weight parameters, so that it makes better predictions in the next feedforward step.

An *epoch* is defined as one forward pass and one backward pass of all the training images. Some standard *optimizers* are Stochastic Gradient Descent (SGD), RMSprop [16] and Adam [17], which all have parameters that can be tuned to improve the learning process. The most important parameter is the *learning rate* which controls the size of weight changes during training.

A common problem that occurs when training a model is that it tries to memorize the training images instead of trying to generalize from patterns observed in the training images. This is called *overfitting* and often happens when the model is too complex by having too many parameters. Since the model loses its generalization power, it will have very poor predictive performance on unseen test images. Fortunately, there are several techniques developed in order to minimize overfitting.

Figure 2.1 show the three main types of layers when building 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 with a set of filters (also called kernels or feature detectors), each producing one feature map of the image which contains key features like edges, lines, shape, intensity etc. Naturally, increasing the number of filters means more image features getting extracted leading to a network that is better at recognizing patterns in unseen images. The size of the filters are usually set to a small number like 3×3 or 5×5.

- Pooling layers reduce the spatial dimensions and retain the most important information of the feature maps with a down sampling operation. This controls overfitting by reducing the number of parameters and computations in the network. Additionally it makes the convolution process invariant to translation, rotation and shifting. Max-pooling and Average-pooling are commonly used. Max pooling simply iterates over the image with a small pixel neighborhood (usually 2×2) choosing the maximum value within the window. Average-pooling calculates the average value of the pixels in the window instead.

- Fully Connected layers take care of the high level reasoning of the features that are output from the previous convolution and pooling layers. The

last FC layer also known as the output layer, will give the final class probabilities.

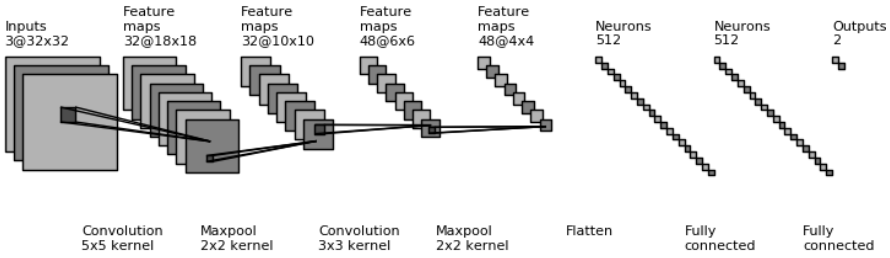


Figure 2.1: Model diagram of a deep convolutional neural network architecture

Transfer learning

When working with a small amount of data it is common to use existing CNNs that are already trained on very large datasets, such as ImageNet which contain 1.2 million images and 1000 classes [6]. 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.

2 different scenarios within transfer learning are used:

- Remove the last fully connected layers of the pretrained CNN, then treat the convolutional base of the pretrained CNN as a fixed feature extractor for the new dataset. A custom classifier can then be trained on these features, which are also called *bottleneck features*.

- After replacing and training a classifier on top of the convolutional base on the new dataset, it is also possible to fine-tune the weights of the pretrained network by continuing the backpropagation. In this scenario, it is common to keep the earlier convolution layers fixed due to overfitting concerns.

Related Work

In 2013 Yu et. al. [18] introduced improved Sparse coding Spatial Pyramid Matching (ScSPM) for automatic species identification in wildlife pictures captured by remote camera traps. The method used dense SIFT descriptors and cell-structured LBP to extract local features that represent the object of interest. Furthermore, they generated global features with weighted sparse coding and max-pooling using a multi-scale pyramid kernel, and then classified the images using a linear Support Vector Machine (SVM). With an average classification accuracy of 82% they showed that traditional image recognition techniques can be effectively used to identify wild mammals in camera trap images.

The method was evaluated using a dataset with over 7000 images containing 18 species from two different field sites. As a preprocessing step, the animals were manually cropped and images that were empty or did not contain a full animal body was removed. This probably had an effect on the results as it diminishes the difficulty of the classification considerably and actually turns the problem into an individual animal identification task.

Chen et al. [19] presented the first attempt for a fully automatic computer vision based method for species recognition in camera trap images. They used a 6-layer CNN on a challenging dataset that contained 20 animal species, and compared it against traditional BoVW methods. Before being fed to the species recognition algorithm, the animals were segmented automatically by cropping regions of interest using Ensemble Video Object Cut [20]. The experimental results showed that the DCNN achieved 38.3% accuracy and was superior to the BoVW method, but their performance did not meet the requirements of full automation in species recognition.

In 2016 Gomez et al. [5] proposed a solution based on very deep CNNs. The approach was different from [19] in two main aspects. Firstly, they manually segmented animals from images, since there currently are no automatic segmentation algorithms that provide perfect results. However, they also included images where only parts of the animal was visible, as opposed to [18]. Secondly they used CNNs that had much deeper architectures and that was pretrained in some cases (AlexNet [11], VGGNet [7], GoogLenet [21] and ResNets [22]). With these models they were able to achieve 88.9% Top-1 and 98.1% Top-5 accuracy on the Snapshot Serengeti dataset [23]. The results prove that CNNs architectures are highly capable of dealing with the animal species classification problem, even in cases where images contain only parts of the animal body.

They also did an analysis on four versions of the dataset (imbalanced, balanced, conditioned, and segmented) to compare the effects of different challenges that can occur in wild animal monitoring. Results showed that the highly imbalanced dataset performed worst while the segmented dataset gave best accuracy, which further manifest the possibilities if there is enough training data and an accurate segmentation algorithm available.

3

Methods

Dataset and Annotation

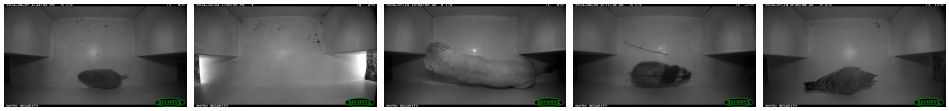
The rodentcam dataset provided by COAT consist of 74 429 unlabeled camera trap images taken from 2014-2015. The images are spread in 77 folders, where each folder represent an individual camera trap box set up at an unknown location. Some immediate observations when looking at the images is that they mostly have the same static background, with the exception of small objects appearing in the box for reasons like weather or nest building. This is very beneficial for our learning algorithms since less background noise means more focus on important features, like animals. There are some variations in image angles and point of view depending on how the camera is setup in the box, but data augmentation techniques that are presented later mitigate these differences.

All the images are grayscale with the exception of the Reconyx logo in the lower right corner. The obvious benefits are faster computation and less resource requirements in terms of disk and memory usage, which is a primary bottleneck when running CNNs. But one channel images can also be a big disadvantage, since color information could be a distinguishing factor in detecting a certain animal species. It is an important tradeoff, which has to be considered when collecting data.

(a) Normal setup **(b)** Abnormal setup

Figure 3.1: Camera setup

The initial impression of the dataset was that most of the folders have images with mouse and images that were empty. There were also some folders containing lemmings and weasels, and a few with birds. Keep in mind that these basic separations were made without the expertise of an ecologist.



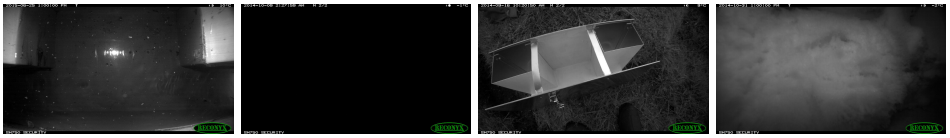
(a) Mouse **(b)** Empty **(c)** Weasel **(d)** Lemming **(e)** Bird

Figure 3.2: Initial classes

To start off simple, the first annotation approach was to manually pick out and label 100 images for each of the five classes (Mouse, Empty, Lemming, Weasel, Bird) from a couple of the folders. The images was further split into training (80%) and validation(20%) sets, then used for training a simple neural net. Even with such a small dataset, the classification accuracy on the validation set was surprisingly good at around 85%. Unfortunately this was only a false hope since the model was not predicting correctly on unseen test images, which indicates that the model was not exposed to enough images to generalize well on the whole dataset. Clearly, there were more differences between the folders than assumed initially and picking images manually for annotation was not a good approach. It was also obvious that a lot more data needed to be annotated to train a good model.

To tackle this challenge in an efficient way, a simple annotation script was made in python. The script loads all the images in the rodentcam dataset, randomly shuffles them and splits them into a training set (67%) and test set (33%). It then proceeds to show the training images one by one, and with a simple key stroke, the image is saved to its respective class folder. Around 10000 training images was annotated in 20 hours using the script and a few extra classes were discovered. The new classes was added to get more information about

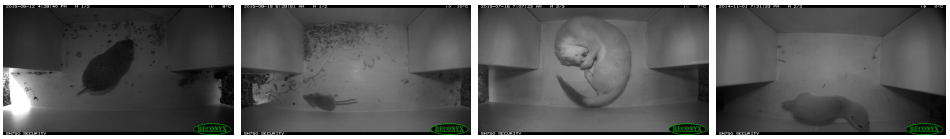
the state of the camera trap box and to cover all possible situations within the rodentcam dataset. These classifications could be useful to determine the functionality of the camera trap. For example, black images could indicate that the camera flash is not working properly, and large quantities of images with snow or water could indicate bad ground positioning of the box.



(a) Water **(b)** Black **(c)** Setup **(d)** Snow

Figure 3.3: Four extra classes added

This 9-class training set was subsequently sent to an ecologist working in the COAT project who further separated animals into specific species and corrected some mislabeled images. Shrew and Vole was derived from the Mouse class and LeastWeasel from the Weasel class. The Vole class include the species gray-sided vole and tundra vole, which are not possible to separate in grayscale images.



(a) Vole **(b)** Shrew **(c)** Weasel **(d)** LeastWeasel

Figure 3.4: Mouse and Weasel are split into species that are more specific

In the end, we have three training datasets with different types of classifications used for evaluating the system:

Class	Images
Animal	5506
NoAnimal	4406

Table 3.1: 2 class dataset

Class	Images
Bird	37
Black	289
Empty	2884
Lemming	201
Mouse	4926
Setup	34
Snow	972
Water	238
Weasel	345

Table 3.2: 9 class dataset

Class	Images
Bird	37
Black	289
Empty	2884
LeastWeasel	54
Lemming	226
Setup	34
Shrew	180
Snow	972
Vole	4703
Water	238
Weasel	365

Table 3.3: 11 class dataset

Looking at table 3.2 and 3.3 it is evident that our multiclass datasets are extremely imbalanced. Table 3.3 show that the majority class Vole make up almost half of the dataset while the minority classes Bird, Setup and LeastWeasel only cover a small fraction. This could result in biased predictions and misleading accuracies because the DCNNs will favor the classes with the most data and small classes might not provide enough information to make an accurate prediction. Luckily, there are several techniques to combat this problem since datasets with skewed class distributions is a common occurrence in the machine learning community. We will elaborate this subject more in our discussion because it plays a big role on our results and how they are interpreted.

Segmentation

Related work show that segmenting animals from the images before feeding them to the model is a popular approach [18][19][5]. On camera trap images above ground, it is easy to see the benefits of removing the background. However, in our case where we have images with a fixed background there might not be much to gain. Some experimentation with an automatic segmentation technique was done regardless. By using the following method with classic binary thresholding, we were able to get decent results:

- Create mean image by computing the average of all images in a folder.
- Traverse each image in the folder, blur it with Gaussian filter to get rid of irrelevant noise, and subtract it from the mean image to get the delta image.
- Apply a binary threshold to the delta image to only keep the large differences and dilate remaining pixels to fill in the holes.
- Find and compute a bounding box for each contour to get the resulting segmentation.

(a) Original **(b)** Mean **(c)** Delta **(d)** Threshold **(e)** Result

Figure 3.5: Automatic segmentation process

The main issue with this technique was the high illumination variance near the entrances of the camera trap box due to change between night and day. This resulted in empty bounding boxes in a majority of the images, as seen in figure 3.5e. One solution was to increase the threshold, but this caused other problems such as weasels not being detected because their brightness is similar to the background. Another idea was to create a separate segmentation algorithm for each animal, but this would be highly impractical and inefficient for large-scale problems. We concluded that this method was not good enough and proceeded to work on the original images including background. There is definitely room for improvement on this subject and an alternative will be presented in the future work section.

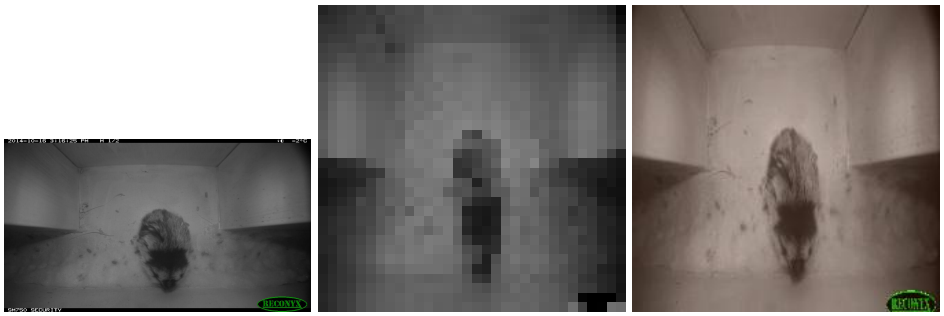
Preprocessing

We preprocess our data before training the models. Different preprocessing is applied depending on if we are using the custom model which is trained from scratch, or the bottleneck and fine-tune model that are based on the pretrained VGG-16 net. [7]

Originally the images are 1280×720 pixels which corresponds to an input vector of over 920 000 dimensions. They clearly have to be reduced in size to avoid curse of dimensionality [24] and to be able to fit in memory. The normal approach is to resize the image to a much smaller size like 48×48, then take random crops to get more training data. In our case, we skip the random cropping since it could cause loss of valuable information in images where only a small part of the animal is visible.

For our small custom model we use 32×32 grayscale images as input and for the bottleneck and fine-tune model we use 224×224 BGR images. Notice how we have to use three channel input for the pretrained models even though we have grayscale images. This means that color-specific filters learned in the VGG network will not have any effect on our images. To follow the preprocessing specified in [7] we also subtract the mean RGB value, computed on the ImageNet training set, from each pixel.

An optional preprocessing step to crop the black borders from the images is also implemented. The borders contain information like date, time and temperature, which is insignificant. It could actually serve as a negative factor since the neural nets might make predictions based on patterns learned from the date or time. Removing the borders had some effect on our results which are mentioned in chapter 6.



(a) Original image ($1280 \times 720 \times 3$)
(b) Custom model ($32 \times 32 \times 1$)
(c) Pretrained models ($224 \times 224 \times 3$)

Figure 3.6: Image before and after preprocessing

After loading, preprocessing and splitting the images to training (80%) and validation (20%) sets, we save them to train and validation folders on disk. This provides the benefit of quickly loading data in the case of retraining and it accommodates the use of data generators, which lets us load images in batches for efficient memory usage.

Data Augmentation

A training set of 10000 images with just a few hundred or thousand pictures from each class is quite small. To generate additional data we apply various transformations on our original images, also known as data augmentation. [25] This easy and powerful technique helps prevent overfitting and helps the model generalize better because it will never see the exact same picture twice. It is also computationally cheap since the data is generated in real-time on the CPU simultaneously as the GPU is training on previous batches of images.

We apply data augmentation in the form of horizontal flips, vertical flips, random 0-5% pixel width shifts and random 0-10% pixel height shifts. To avoid the possibility of losing important image information we limit the pixel shift range and omit other types of transformations, such as random rotations. Figure 3.7 show a few samples of our data augmentation strategy.

Figure 3.7: Samples of augmented data

4

Design

Model Architectures

Figure 4.2 show the architectures of the three DCNN used by the system.

The custom model has 11 weight layers and can be seen as a smaller version of the VGG nets [7]. It has four convolution blocks containing 2 convolutional layers and 1 max pooling layer each. All the convolutional layers use a small filter size of 3×3 and maintain the spatial dimensions of the input by padding it with zeros. The max-pooling layers at the end of each convolutional block applies spatial pooling. This spatial pooling will halve the feature map dimensions, which lets us double the number of filters and preserve the time complexity per layer. These design principles generally follow that of [22] and [7].

After the convolutional layers, the feature maps are flattened and sent through two fully connected (FC) layers with 256 neurons and a final FC output layer with 11 neurons. We add 0.5 dropout regularization after each FC layer. This will set the output of each neuron to zero with a probability of 0.5, to prevent overfitting and improve the models ability to generalize [26].

All layers except the output layer use the Rectified Linear unit (ReLu) activation function, which is a popular choice since it significantly speeds up the network training time [11]. The activation function for the output layer is set to softmax for multiclass output and sigmoid for binary class output.

The 3-layer bottleneck model is created entirely of FC layers since the input to this model are *bottleneck feature* maps. These *bottleneck features* are generated by feeding our data through the convolutional base of a VGG-16 net, which is pretrained on the large ImageNet dataset [6]. This pretrained net will have learned features that are useful for us because the ImageNet dataset contains classes that we use, like Mouse and Lemming. Therefore, we utilize it to get useful high-level features in our data and train our fully connected model on these features.

The model consist of two FC layers with 4096 neurons and a final 11 neuron output layer with the same activation functions as the custom model. A more aggressive dropout of 0.6 is applied and additionally we add 0.1 L2 weight penalty to each FC layer to further combat overfitting and improve generalization [27].

We create our fine-tune model by adding a pretrained bottleneck model on top of the pretrained VGG-16 convolutional base. In table 4.1 we see that fine-tune model has 134.3 million weight parameters. Due to the computational complexity that comes with handling all these parameters, we freeze all weights up to the last convolutional block, which is fine-tuned along with the fully connected layers. The features learned by low-level convolutional blocks are more general than those found higher-up, so it is sensible to freeze the first few blocks and only fine-tune the last one which has more specialized features.

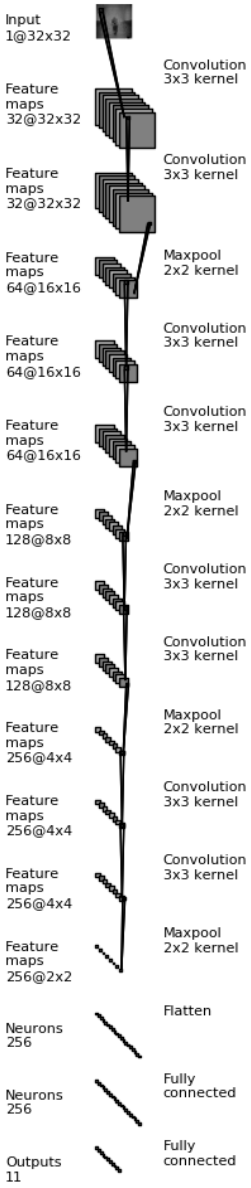


Figure 4.1: Diagram of the custom model architecture

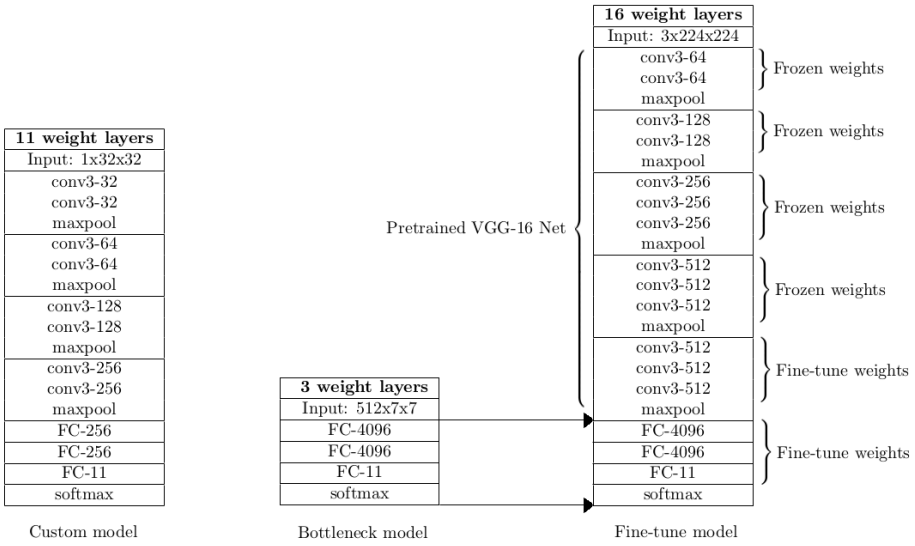


Figure 4.2: Overview of the model architectures. The convolutional layer parameters are denoted as *conv[filter size]-[number of filters]*. The fully connected layers are denoted as *FC-[number of neurons]*. Activation functions and regularization techniques are omitted for readability.

Model	Custom	Bottleneck	Fine-tune
Number of parameters	1.5	119.6	134.3

Table 4.1: Number of weight parameters (in millions)

System Workflow

Figure 6.7 shows an overview of the system
workflow when training the bottleneck model and fine-tune model. It is split up in the following 5 phases:

- **Load data:** When the system is run for the first time with a new training dataset, it loads all the images from the given training directory, preprocesses them as described in section 3.3, splits them into training and validation sets, then saves them on disk for reuse. This phase is skipped if the preprocessed images already exist.

- **Save bottleneck features:** The training and validation data generator loads preprocessed images in batches and feeds them to the pretrained VGG-16 base model which generates bottleneck features for our data and saves them on disk for reuse. This phase is skipped if the bottleneck features already exist.

- **Train bottleneck model:** The bottleneck model is created, its architecture is saved, then it is trained on the bottleneck features which are loaded from disk. During training the weights with best validation accuracy is saved. After training, the validation results are saved along with accuracy and loss trend graphs. This phase is skipped if a trained bottleneck model already exist.

- **Train fine-tune model:** The fine-tune model is created by concatenating the trained bottleneck model on top of the pretrained VGG-16 base model. The training and validation data generator loads preprocessed images in batches, applies data augmentation as described in section 3.4, and feeds them to the fine-tune model. During training the weights with best validation accuracy is saved. After training, the validation results are saved along with accuracy and loss trend graphs.

- **Predict:** This phase is optional. If a test directory is given, the system will load the test images and the trained model. Before making the predictions, the test data is preprocessed as described in section 3.3. The test results are saved when the predicting is finished.

There is also a separate workflow for training the custom model that is identical to phase 4 if we replace loading of existing models with creating the custom model. The optional prediction phase is applicable to all three models.

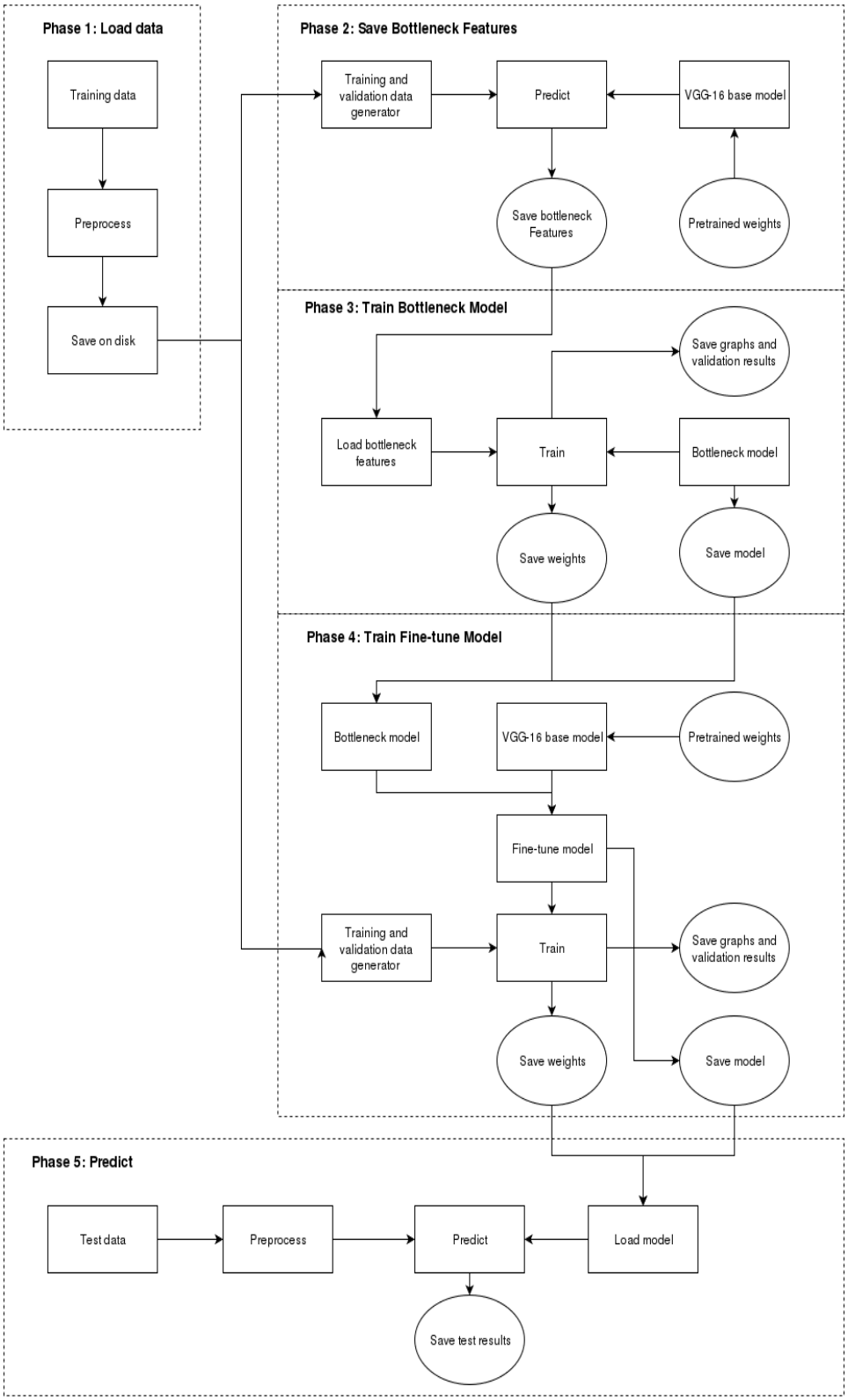


Figure 4.3: Overview of system workflow when training the bottleneck and fine-tune model

5

Implementation

The system prototype was written in python because of the extensive amount of machine learning frameworks available in this language. These frameworks are often developed in C/C++/java to maximize performance, and then integrated to a python API for easy use. Two of the top python libraries for deep learning development and research are Tensorflow [28] and Theano [29]. Both are very robust and powerful tools but they can be difficult to use directly when creating CNN models. To make model definition and training simpler, higher level frameworks such as Keras [30] and Lasagne [31] has been created to run on top of these platforms.

We use Keras v1.1.0 with Theano v0.8.2 as a backend because it allows for easy and fast model creation by combining standalone modules such as neural layers, optimizers and cost functions. The Keras API also provides *image generators* which enables us to use real time data augmentation and feed batches of images to the neural network to avoid the memory bottleneck of loading all images at once. With Theano as a backend the training is done seamlessly on the CPU or GPU and in terms of computation performance, Theano is proven to be on par with other major research software, like TensorFlow [29].

Several other computer vision libraries are used in our system:

- OpenCV v3.1.0 provides the functionality for loading and preprocessing the images.

- We use the metrics library in scikit-learn v0.18 to calculate various prediction metrics such as *Accuracy*, *Precision* and *Recall*.

- To plot graphs that show accuracy and loss trends during training we use the pyplot library in Matplotlib v1.5.3.

The fundamental package for scientific computing with Python, Numpy v1.11.2 is the backbone of our system and the previously mentioned libraries since it adds support for large, multi-dimensional arrays and matrices that is needed when dealing with image data.

6

Evaluation

This chapter describes the experimental setup and prediction metrics used to evaluate the custom, bottleneck and fine-tune model. All models were trained on each training dataset presented in section 3.1. This way we can compare the quality of predictions of the models and see the effects of our different classifications. Secondly, we will look at the models computational performance to see what kind of resource allocation is needed, both in the training and prediction phase. This is important because we want to get a sense of what is required for the system to be used in real time at a remote camera trap.

Experimental Setup

All experiments was done on a laptop with a Nvidia Geforce GTX 765m GPU, an Intel Core I7-4700HQ @ 3.40GHz×8 and 8GB DDR3 1600MHz. It ran on Ubuntu 16.04 LTS 64-bit with gcc v5.4.1 compiler. The Nvidia GPU was used as a dedicated CUDA card while the integrated Intel GPU was used for running X server. Theano was configured to use the GPU with CUDA 8, cuDNN v5 (version 5005) and data type float32.

Nvidia GTX 765m has 2 GB memory and processing power of 1326 GFLOPS, which is quite low compared to the high-end cards that is normally used in machine learning problems. This caused some model parameters such as batch size and number of epochs to be restricted due to memory usage and training

time. It also prevented us from using cross validation when evaluating the models performance, because of the computational issues of consecutively retraining the models in one session.

The training datasets described in section 3.1 were shuffled and split into training (80%) and validation (20%) sets, resulting in 7985 images for training and 1997 images for testing. These sets should give a good representation of the rodentcam dataset. Post-training we did predictions on unseen test images and manually went through them to verify the results and to confirm that the models were generalizing well.

We trained the models several times and manually tuned different parameters to find which values and combinations gave the best results for our training sets. The following parameters were used for our evaluation:

- The custom and bottleneck model was trained for 100 epochs using a batch size of 64 and we saved the weights that gave best validation accuracy during training. They were compiled with an Adam [17] optimizer with initial learning rate $1 \cdot 10^{-4}$ for the custom model and $5 \cdot 10^{-5}$ for the bottleneck model. The learning rate was reduced by half every 20 epochs.

- The fine-tune model was trained for 30 epochs using a batch size of 32 and we saved the weights that gave best validation accuracy during training. It was compiled with a SGD optimizer and initial learning rate of $5 \cdot 10^{-5}$. The learning rate was reduced by half every 10 epochs.

All models used categorical cross-entropy cost function for multiclass training and binary cross-entropy for binary class training.

To measure models performance we looked at runtime, RAM usage, CPU usage and GPU memory usage during the training and prediction phase. Total training time of the models was measured with the unix *time* command. Time used per epoch was output by Keras model training function. RAM, CPU and GPU memory usage during training was recorded with Ubuntu's system monitor and *nvidia-smi* command, we did not want to use profilers that could slow down the system.

To test prediction performance we used CPU instead of GPU, since remote camera traps will probably not have a GPU. We chose to measure the runtime, CPU usage and RAM usage of 1 image prediction to see what a camera trap would require for real time animal identification. Time used on 100 image predictions is also included for comparison. RAM usage was recorded using python memory_profiler v0.41 and we measured prediction time with pythons time module. The results were computed as the average of 10 prediction runs. Disk usage is also included to see the storage requirements of our models.

Prediction metrics

To measure the models quality of predictions we use three different metrics: *Accuracy, Precision* and *Recall*. We look at classification accuracy because it is a commonly used metric when evaluating CNNs and it will tell us the proportion of correct predictions made by the models. Additionally, we look at the average and individual class precisions and recalls which respectively measures the prediction relevancy and how many truly relevant predictions are returned.

Accuracy is measured as the fraction of correct predictions over the total number of samples and is defined as

$$Accuracy(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y}_i = y_i)$$

where $\hat{y}$ are the predicted labels and y are the corresponding true labels for all samples.

To measure the average precision and recall for our classifications we use Macro-averaging. The average precision/recall is then measured as the fraction of the sum of class precisions/recalls over the number of classes and is defined as

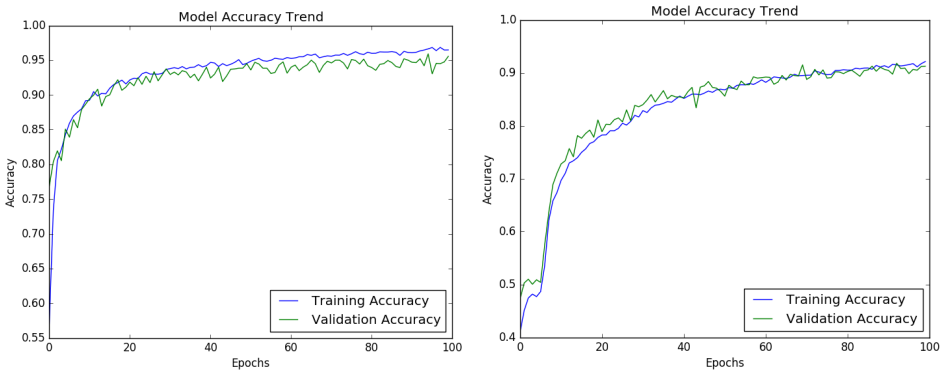
$$\text{Macro-average } Precision = \frac{1}{n_{\text{classes}}} \sum_{i=0}^{n_{\text{classes}}-1} P(y_i, \hat{y}_i), \qquad P(y, \hat{y}) = \frac{|y \cap \hat{y}|}{|y|}$$

$$\text{Macro-average } Recall = \frac{1}{n_{\text{classes}}} \sum_{i=0}^{n_{\text{classes}}-1} R(y_i, \hat{y}_i), \qquad R(y, \hat{y}) = \frac{|y \cap \hat{y}|}{|\hat{y}|}$$

where $\hat{y}_i$ are the predicted labels and y_i are the corresponding true labels of the samples in each individual class i.

Results

Figure 6.1 shows the accuracy trend of the custom model during training. We can see that the accuracy quickly increases in the beginning, and then starts to smooth out to an inverse exponential curve, which demonstrates that the model is learning at a good rate. In figure 6.1a the validation accuracy starts to slightly fall behind the training accuracy after epoch 40, but it is still increasing slowly. This indicates that the learning capacity of the model is reached faster for the 2-class training set compared to the 9-class and 11-class training set, where the validation accuracy continues to follow the training accuracy until the very end. Naturally, the model will have an easier time learning to separate 2 classes versus 9 or 11.



(a) 2-class training set **(b)** 11-class training set

Figure 6.1: Accuracy trend when training custom model. The 9-class training set is omitted because of its similarity to (b).

Validation set	Accuracy (%)	Precision (%)	Recall (%)
2-class	95.31	95.50	95.50
9-class	92.04	73.22	73.77
11-class	90.88	71.45	71.54

Table 6.1: Custom model validation results

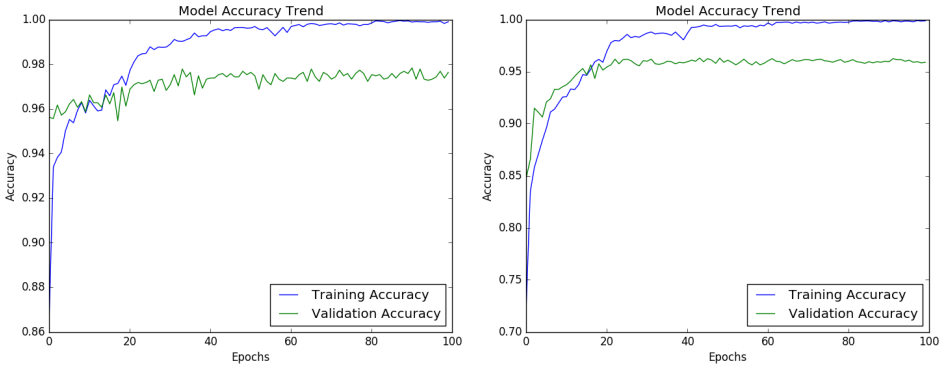
Table 6.1 shows the validation results after training the custom model. The overall prediction accuracies are between 90%-95% for all classifications which is excellent, but it does not mean that the model is making good predictions. Looking at the precision and recall for the multiclass validation sets, they are only 71%-73%, which is poor considering the high accuracy. This is caused by the imbalanced nature of these training sets. The model will obviously favor the majority classes like Empty and Vole causing them to have a high precision and recall, while the minority classes like Bird, LeastWeasel and Shrew will have a low precision and recall that brings down the average.

For example, the class LeastWeasel has 40% precision and 36% recall. This means that only 40% of the LeastWeasel predictions that are made is correct and 36% of images with a LeastWeasel are correctly classified. It does not have a high impact on the overall accuracy because they are minority classes. Figure 6.2 shows images from the validation set where the custom model misclassifies minority classes. The images display the real class, the predicted class and the probability of the prediction.

(a) Bird misclassified as Vole **(b)** Shrew misclassified as Vole

Figure 6.2: Classification of minority classes by the custom model

From the bottleneck model accuracy trend graphs we see that the initial accuracy is very high compared to Figure 6.1, starting at 70% for the 11-class and 86% for the 2-class training set. This shows that the pretrained VGG-16 convolutional base generated useful high-level features from our data, allowing fast learning for the fully connected layers in our bottleneck model. As a negative consequence, the training accuracy quickly exceeds the validation accuracy during training, causing the model to slightly overfit.



(a) 2-class training set **(b)** 11-class training set

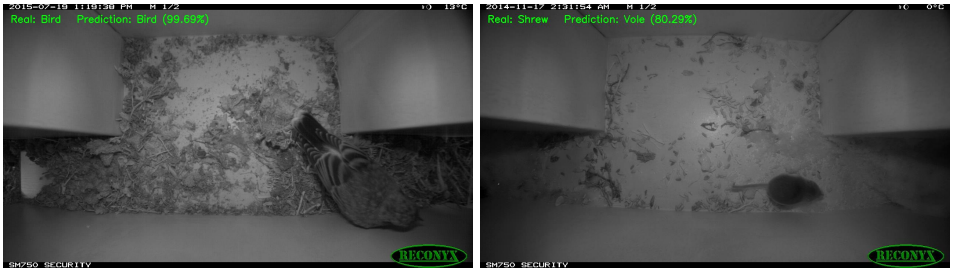
Figure 6.3: Accuracy trend when training bottleneck model

Validation set	Accuracy (%)	Precision (%)	Recall (%)
2-class	97.68	97.50	97.50
9-class	96.37	92.33	85.66
11-class	96.29	95.81	85.00

Table 6.2: Bottleneck model validation results

Table 6.2 shows the very promising results of our bottleneck model. The accuracy has increased by 2%-6% for every validation set compared to the custom model. This is a significant improvement, especially since we are in the 90-percentile scores. The 2-class validation set is nearing peak performance with only 46 misclassifications in 1997 images and the accuracy gap to the multiclass sets is much smaller than in Table 6.1.

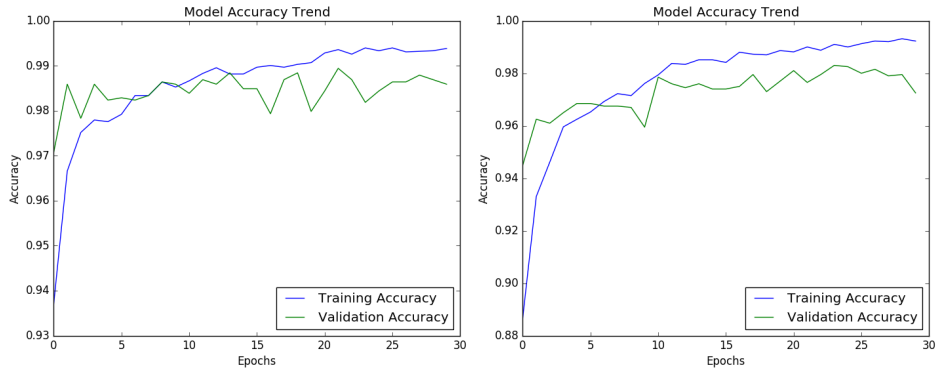
Precision has increased to the 90-percentile, which means that the predictions made by the bottleneck model are more trustworthy. Recall has also notably improved, but it is still inferior to the other metric scores. The LeastWeasel class now has 89% precision and 73% recall, clearly showing that the bottleneck model handles imbalanced datasets better than the custom model. Figure 6.4 shows the bottleneck model predictions on the same validation images shown in Figure 6.2. The bird is now correctly classified, but it still misclassifies the Shrew as a Vole.



(a) Bird correctly classified **(b)** Shrew misclassified as Vole

Figure 6.4: Classification of minority classes by the bottleneck model

We see in figure 6.5 that the initial accuracy of the fine-tune model is even higher than in figure 6.3, starting at 88% for the 11-class and 93% for the 2-class training set. Probably because the fine-tune model is already familiar with our data since it includes the the previously trained bottleneck model. With a smaller y-axis scale, the variance in the validation accuracy is more evident, but there are clear improvements during the course of the training. Around epoch 15, the model starts to overfit slightly, similar to the bottleneck model.



(a) 2-class training set **(b)** 11-class training set

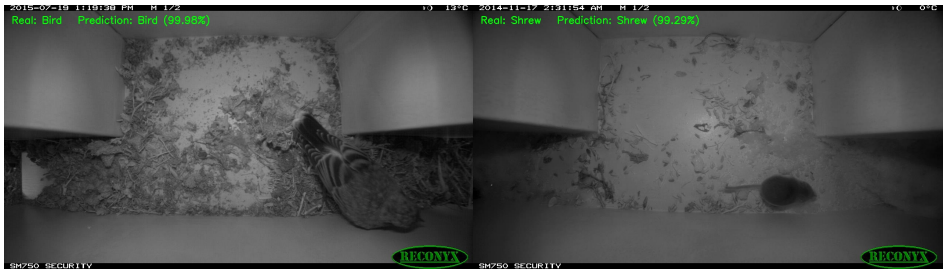
Figure 6.5: Accuracy trend when training fine-tune model

Validation set	Accuracy (%)	Precision (%)	Recall (%)
2-class	98.89	98.50	99.00
9-class	97.93	95.22	90.55
11-class	97.84	97.81	93.45

Table 6.3: Fine-tune model validation results

We achieve best results using the fine-tune model with 97.84% accuracy, 97.81% precision and 93.45% recall on the 11-class validation set. Only 43 of 1997 images are misclassified and all individual class precisions are in the range 90-100%. LeastWeasel, Bird and Shrew has the lowest recalls at 80-90%, while all other classes has 90-100%. The 9-class validation set has similar accuracy, but worse precision and recall than the 11-class which is surprising. It might be because the majority class in the 9-class training set has a few hundred images more than the majority class in the 11-class training set. This could be an indicator that the precision and recall would drop rapidly with increased class imbalance. The 2-class validation set results has also increased to 98.89% accuracy, 98.50% precision and 99% recall.

We see in Figure 6.6 that the fine-tune model correctly classifies both the Bird and the Shrew that was misclassified by the custom and bottleneck model.



(a) Bird correctly classified **(b)** Shrew correctly classified

Figure 6.6: Classification of minority classes by the fine-tune model

It is clear from figure 6.7 that the fine-tune model takes the longest to train by far with a total training time of 535 minutes and 1070 seconds per epoch. In comparison the bottleneck model used 104 minutes on 100 epochs, which is quite surprising since they almost have the same amount of weight parameters as shown in section 4.1. This indicates that most of the training time is spent in the convolutional layers and not in the fully connected layers. Generating the input features for the bottleneck model also required around 30-40 minutes and was not included in its total training time.

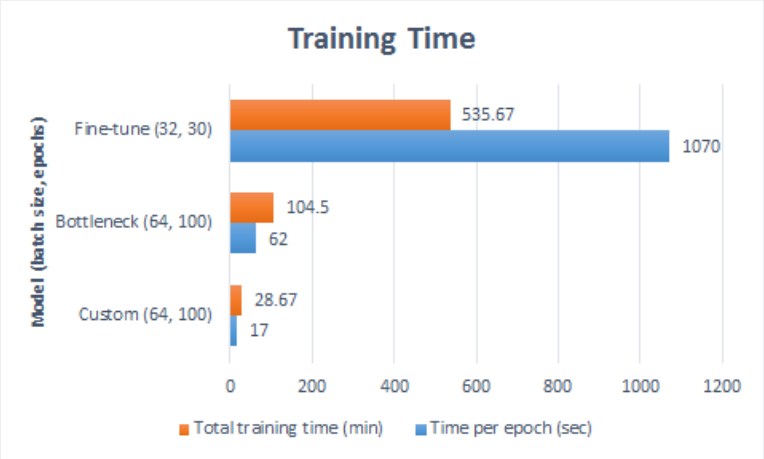


Figure 6.7: Model training times

The differences in GPU memory usage are also quite large. Our GPU was barely able to train the fine-tune model as it used 1903MB memory out of 1998MB available. Storing images in memory is the main RAM consumer and it can be problematic when dealing with a lot of data. The maximum RAM usage during training was 3481MB for the bottleneck and fine-tune model. It is further reduced from 3481MB to 1437MB when retraining, since the preprocessed images are saved on disk. Naturally, the custom model used less RAM and GPU memory because it works with 32x32 pixel grayscale images and it has far fewer weight parameters than the other models. The CPU usage was 13% for all models with 1 of 8 cores working at 100%, probably handling data augmentation.

	Custom	Bottleneck	Fine-tune
GPU memory usage (MB)	225	1437	1903
RAM usage (MB)	640	3481	3481

Table 6.4: Maximum memory usage during training

Figure 6.8 shows three different measurements of time when predicting 1 image. We chose these measurements because we assume that the system can be in different states at the camera trap. For example, that it is running with the models initialized in some kind of sleep mode, only awakening when the camera takes a picture. The time it takes to make a prediction on that image would then be preprocessing + prediction time.

Looking at the total system time it is clear that system and model initializing time is higher for the bottleneck and fine-tune model compared to the custom model due to their size. It is highest for the bottleneck model because of the additional time generating bottleneck features. The fine-tune model has the longest prediction time of 1.55 seconds, while the bottleneck and custom model are almost identical at around 0.5 seconds.

Prediction time for 100 images is shown in figure 6.9 for comparison. All time measurements where the pretrained VGG-16 convolutional base is used has increased by 10 times, while total system time and prediction time for the custom model has barely increased. It is apparent that if very fast prediction is a high priority, then the custom model would be preferred.

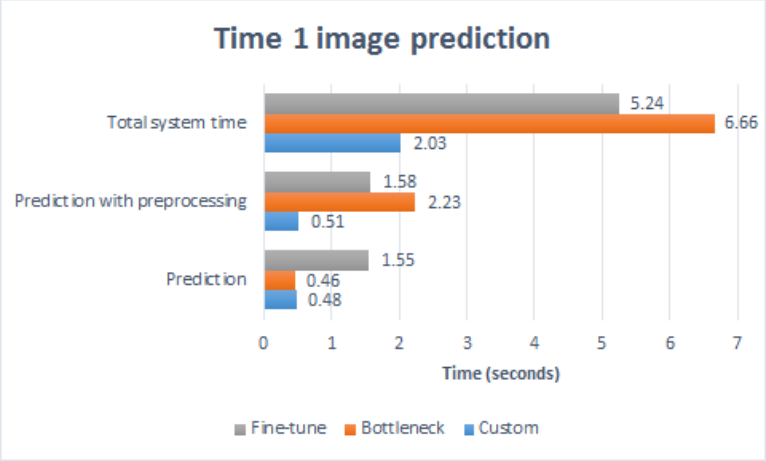


Figure 6.8: Time spent on 1 image prediction

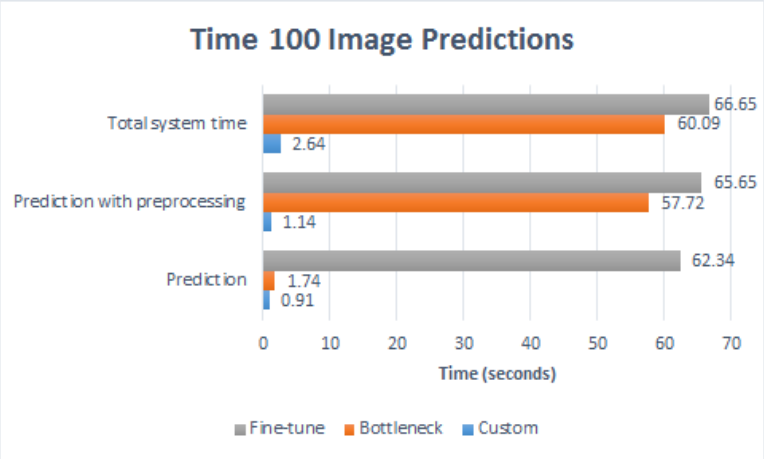


Figure 6.9: Time spent on 100 image predictions

	Custom	Bottleneck	Fine-tune
Disk usage (MB)	6	478	537

Table 6.5: Model disk usage

	Custom	Bottleneck	Fine-tune
RAM usage (MB)	281	1652	1622

Table 6.6: RAM usage for 1 image prediction

Saving the models on disk for reuse requires some storage space. Model architectures are saved in JSON files and only use a couple of KB. The model weights file is saved in HDF5 format and its size is obviously dependent on how many weight parameters the model has. Table 6.5 shows the disk usage for our models.

Since we are predicting with the CPU, model architecture and weights has to be loaded in RAM instead of GPU memory. This is why the RAM usage is so high when only predicting 1 image. CPU usage was similar to the training phase with 1 of 8 cores working at 100% for all models.

Discussion

Our evaluation show that the fine-tune model clearly outperforms the other models in terms of quality of predictions but it comes at the cost of higher computational complexity and hardware requirements. This is an important tradeoff to consider when choosing a model for deployment. In our case it might not be possible to use the fine-tune model in real-time at a remote camera trap due to its hardware requirements, and the custom model might be a better choice if 90% prediction accuracy and 70% precision/recall is considered good enough. Images with low prediction probability could also be manually checked by ecologists to see if misclassifications have been made. If the best possible model and predictions are desired, other solutions could be tested. For example using the fine-tune model at a server that receive images from remote camera traps and handles the predictions.

Class Imbalance

Highly imbalanced datasets was brought up as a common issue with camera trap images in [5] and it was proven true for our dataset as well. It is clear from our results that this caused the custom and bottleneck model to often misclassify minority classes. The fine-tune model however, managed to overcome this issue and provide satisfactory results in all our prediction metrics. This proves that deep pretrained neural nets can handle class imbalance much better than models that are trained from scratch.

It also shows how important it is to look at several different metrics when evaluating a CNN. If we only used accuracy as our metric, it would be quite misleading since it does not show if the model can predict all classes well.

Some solutions were considered to defeat class imbalance in our dataset. We tried adding class penalties to the training process, which means that the model will prioritize learning the minority classes and pay less attention to the majority classes. The models were not able to train properly using these penalties, causing large variations in their results which indicates that the class imbalance in our dataset was too high to be handled by such penalties.

Using two models in the classification process could also help reduce class imbalance in our dataset. The first model would be trained on the 2-class training set to separate images with and without animals, this dataset is balanced as seen in section 3.1. Images predicted to contain an animal would then be sent through a second model trained only on the animal classes for identification. This would effectively get rid of all classes not containing an animal including the majority class Empty, making animal identification easier for the second

model. There are some negative consequences to this approach. We would not be able to predict the state of the camera trap, which can be useful in some cases, and if the first model makes misclassifications, we would lose important images containing animals. Additionally, the second model would still have a class imbalance problem with the animal classes, but it would be less severe than it was originally. Some other methods that could potentially help against our imbalanced dataset are mentioned in the future work section.

Hyperparameters

From our experimentation, we learned that hyperparameters have a large impact when training a neural network and it is very important to tune them to fit the dataset. We tweaked many training parameters until we achieved our results. All models improved by at least 7% due to this tweaking. The most dramatic change was switching the optimizer from RMSprop to Adam and finding the right learning rate for our bottleneck and custom model, this increased their validation accuracy by around 3%. Leaning rate decay also had a big influence on the results. We trained with a fixed learning rate at first, to see at what epoch the validation accuracy stopped improving. Then we applied learning rate decay at this epoch. The positive effects of this are clearly shown at epoch 10 and 20 in figure 6.10b.

(a) Fine-tune model without learning rate decay

(b) Fine-tune model with learning rate decay

Figure 6.10: Effects of adding learning rate decay

After each training phase, we manually went through misclassified images in the validation set to see what images the model struggled with and to occasionally find mislabeled images, which were corrected. Cropping unnecessary information from our images as mentioned in section 3.3 also increased validation accuracies by 1%, which indicate that the models made some predictions

based on the date and time that was in the image borders. All these improvements might seem small when looking at them individually, but it quickly adds up when combined.

/7 Conclusion

As part of this paper, we have successfully designed and implemented a prototype system that can automatically identify small mammals in camera trap images using deep CNNs. Classification results show that our methods and hyperparameter tuning alongside the customized model architectures allowed us to achieve near-human performance [6] with 97.84% accuracy, 97.81% precision and 93.45% recall on our validation set containing 11 classes. These results exceed previous state of the art in animal identification on camera trap images. [18] [19] [5]

A comparison of three different models used by the system show that the pretrained bottleneck model and fine-tune model sufficiently overcome class imbalance and outperforms the custom model that is trained from scratch. This comes at the cost of much higher computational complexity, which can be a problem in the sense of deploying the system at a remote camera trap.

Future work

There are further improvements that can be made to our system. Currently it can identify a single animal in each image, which is problematic if an image contains several animals of different species. A superior approach would be to first detect all animals in an image, localize them, and then identify them separately.

Recent work show that CNN can be successfully used for these purposes. [32] replaces the classifier layers of the CNN and adds a regression network that can learn to predict object bounding boxes, effectively segmenting the objects in the image. Furthermore, it combines the bounding box predictions with classification results to identify the object. This approach has obtained very competitive results for detection and classifications tasks, and could be the next step for our system.

The COAT project have experimented with camera traps that capture color images of the rodents. It would be interesting to compare the models results on these color images against current grayscale images to see if there are improvements to be made. Color images also have the advantage of separating more species, such as the gray-sided vole and tundra vole, which are not possible to separate in grayscale 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] 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.

[3] E. M. Soininen, I. Jensvoll, S. T. Killengreen, and R. A. Ims, "Under the snow: a new camera trap opens the white box of subnivean ecology," *Remote Sensing in Ecoogy and Conservation*, pp. 29–38, Apr. 2015.

[4] 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. 52, 2013.

[5] A. Gómez, A. Salazar, and F. Vargas, "Towards automatic wild animal monitoring: Identification of animal species in camera-trap images using very deep convolutional neural networks," *CoRR*, vol. abs/1603.06169, 2016.

[6] 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.

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

[8] G. Csurka, C. R. Dance, L. Fan, J. Willamowski, and C. Bray, "Visual categorization with bags of keypoints," in *In Workshop on Statistical Learning in Computer Vision, ECCV*, pp. 1–22, 2004.

[9] F. Perronnin, J. Sánchez, and T. Mensink, "Improving the fisher kernel

for large-scale image classification," in *Proceedings of the 11th European Conference on Computer Vision: Part IV*, ECCV'10, (Berlin, Heidelberg), pp. 143–156, Springer-Verlag, 2010.

[10] D. C. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," *CoRR*, vol. abs/1202.2745, 2012.

[11] 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.

[12] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, pp. 91–110, Nov. 2004.

[13] T. Ojala, M. Pietikäinen, and D. Harwood, "A comparative study of texture measures with classification based on featured distributions," *Pattern Recognition*, vol. 29, no. 1, pp. 51 – 59, 1996.

[14] S. Lazebnik, C. Schmid, and J. Ponce, "Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories," in *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR '06, (Washington, DC, USA), pp. 2169–2178, IEEE Computer Society, 2006.

[15] X. Zhou, K. Yu, T. Zhang, and T. S. Huang, "Image classification using super-vector coding of local image descriptors," in *Proceedings of the 11th European Conference on Computer Vision: Part V*, ECCV'10, (Berlin, Heidelberg), pp. 141–154, Springer-Verlag, 2010.

[16] 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.

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

[18] X. Yu, J. Wang, R. Kays, P. A. Jansen, T. Wang, and T. Huang, "Automated identification of animal species in camera trap images," *EURASIP Journal on Image and Video Processing*, vol. 2013, no. 1, p. 52, 2013.

[19] G. Chen, T. X. Han, Z. He, R. Kays, and T. Forrester, "Deep convolutional neural network based species recognition for wild animal monitoring.," in *ICIP*, pp. 858–862, IEEE, 2014.

[20] X. Ren, T. X. Han, and Z. He, "Ensemble video object cut in highly dynamic scenes.," in *CVPR*, pp. 1947–1954, IEEE Computer Society, 2013.

[21] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015.

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

[23] 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, 6 2015.

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

[25] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis," in *Proceedings of the Seventh International Conference on Document Analysis and Recognition - Volume 2*, ICDAR '03, (Washington, DC, USA), pp. 958–, IEEE Computer Society, 2003.

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

[27] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 4*, pp. 950–957, Morgan Kaufmann, 1992.

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[29] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[30] F. Chollet, "keras." `https://github.com/fchollet/keras`, 2015.

[31] S. Dieleman, J. Schlüter, C. R. others(and Eben Olson, S. K. Sønderby, D. Nouri, D. Maturana, M. Thoma, E. Battenberg, J. Kelly, J. D. Fauw, M. Heilman, D. M. de Almeida, B. McFee, H. Weideman, G. Takács, P. de Rivaz, J. Crall, G. Sanders, K. Rasul, C. Liu, G. French, and J. D. others), "Lasagne: First release.," Aug. 2015.

[32] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *CoRR*, vol. abs/1312.6229, 2013.